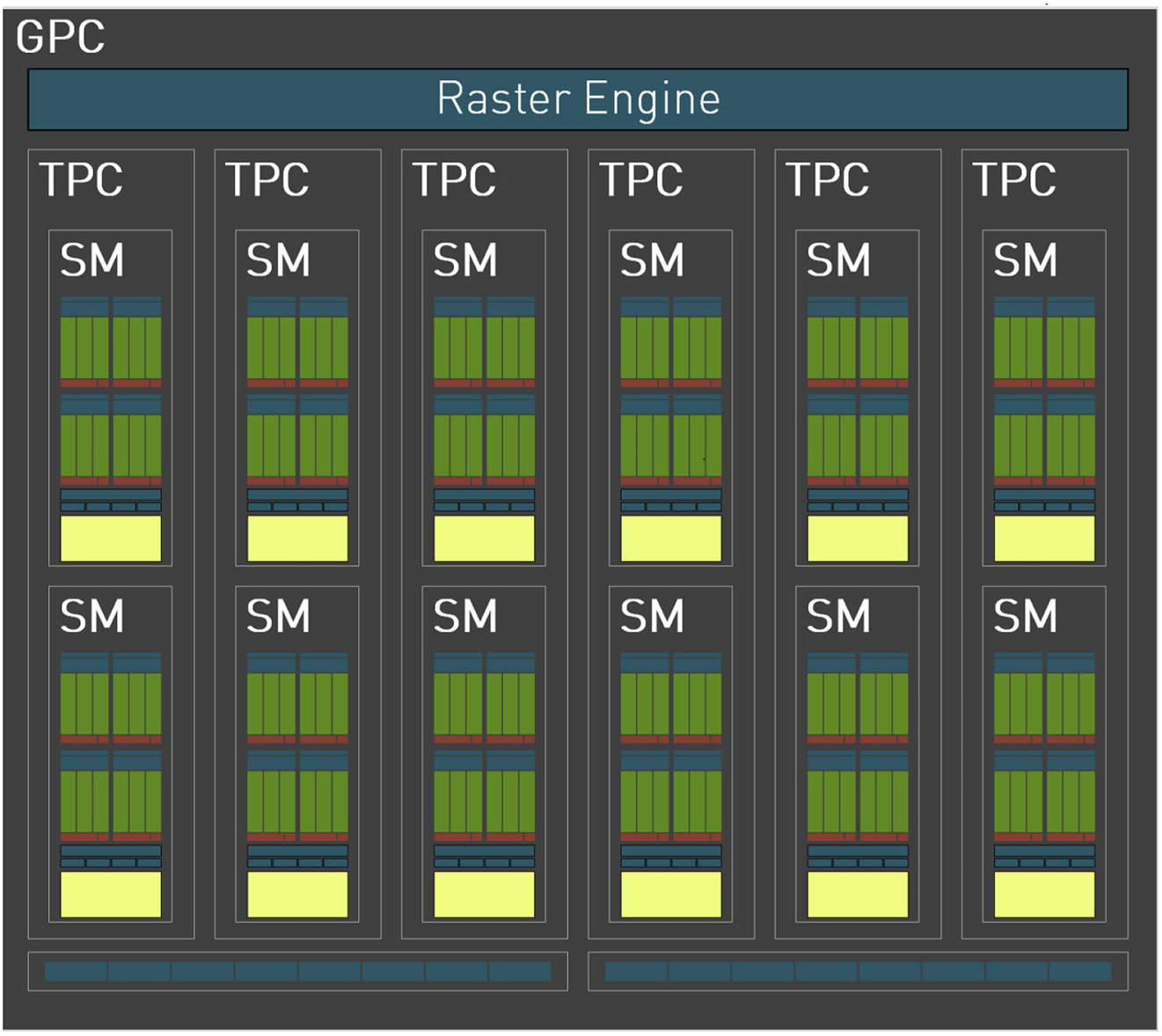
GPU调优和工具

Intro. 现代GPU体系架构

以Ada Lovelace为例

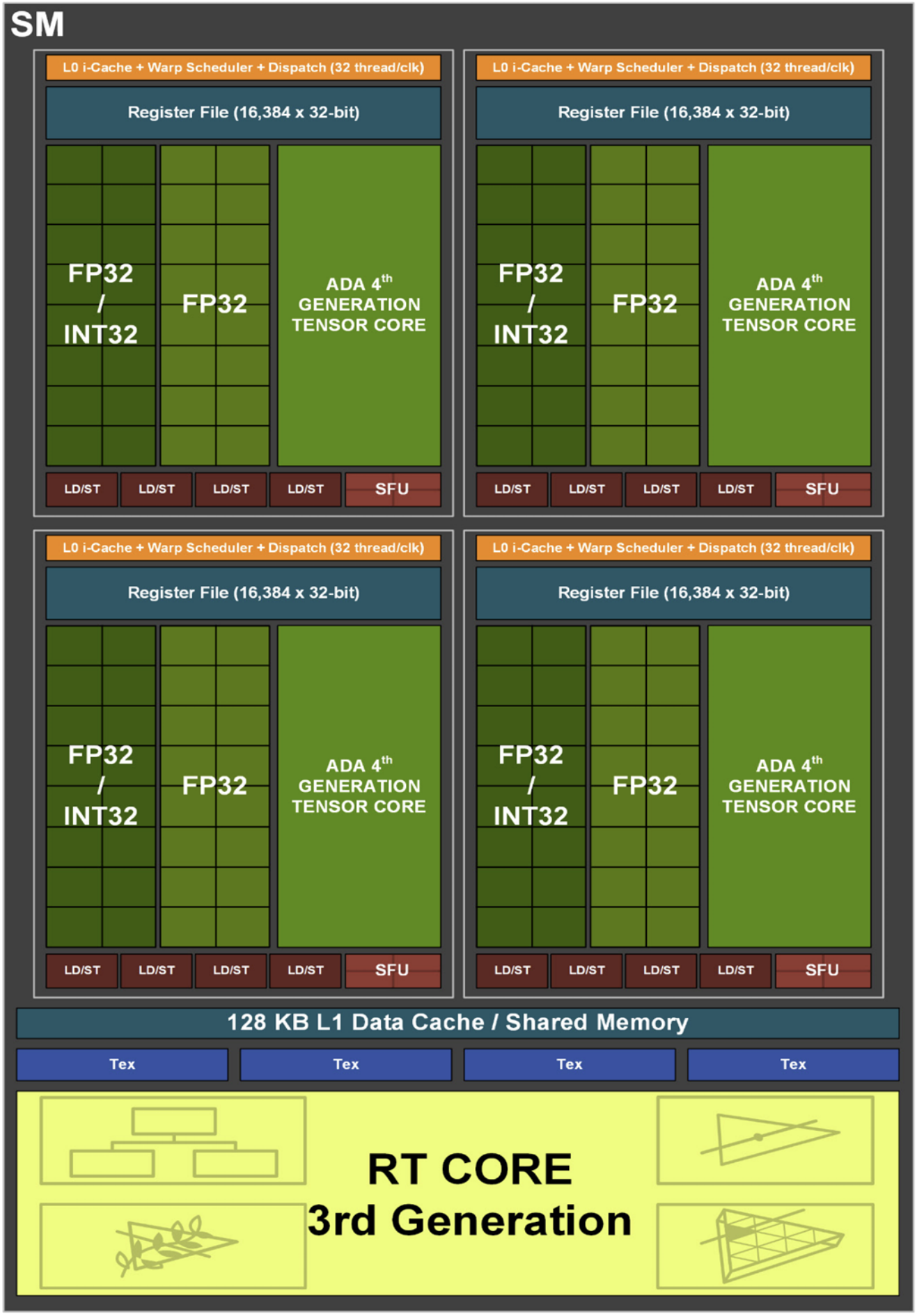
现代GPU架构如图所示，在一个完整的GPU中存在许多GPCs，这些GPCs共享同一块L2cache。

Graphics Processing Cluster



在一个GPC中存在多个Texture Processing Cluster。

Streaming Multiprocessor



在SM中存在许多核心，这些核心被分为四个相同的部分这四个部分共享一个**L1 cache (shared mem的物理位置)**。每一个部分都包含**64KB register**， **L0 instruction cache**，**warp scheduler**，**dispatch unit**以及16个fp32核心和16个

fp32/int32核心还有tensor core。除此之外还有Load/Store unit负责从L1/L2甚至DRAM中读写数据。
在执行cuda程序时，block的执行在硬件层是落实到SM中的，所以一个block能调度的最大线程数量受到SM所能调度的数量限制。在一些架构中这个限制是2048，也有1024。

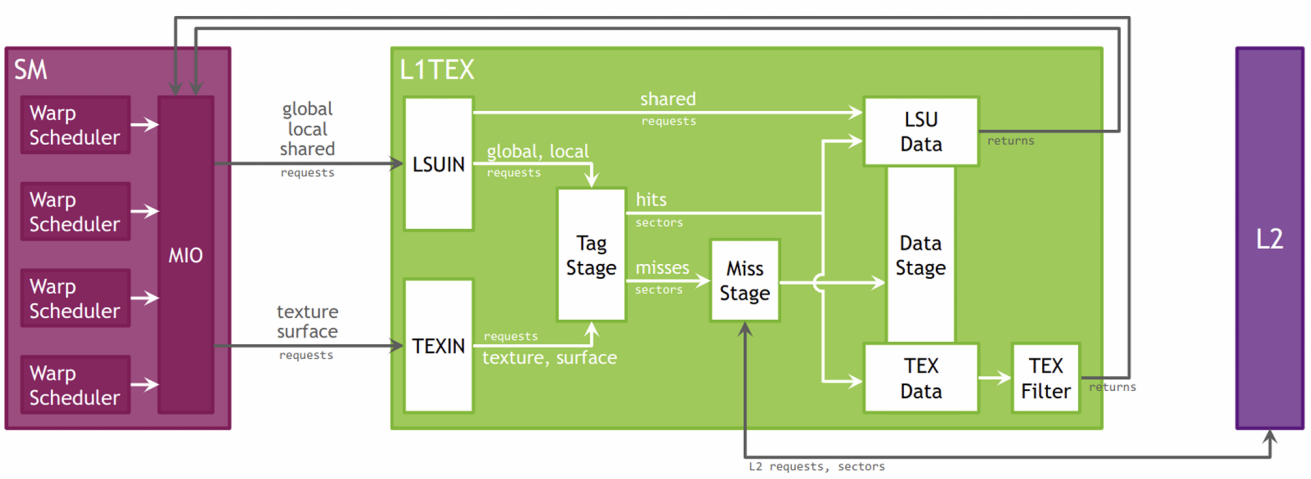
bank

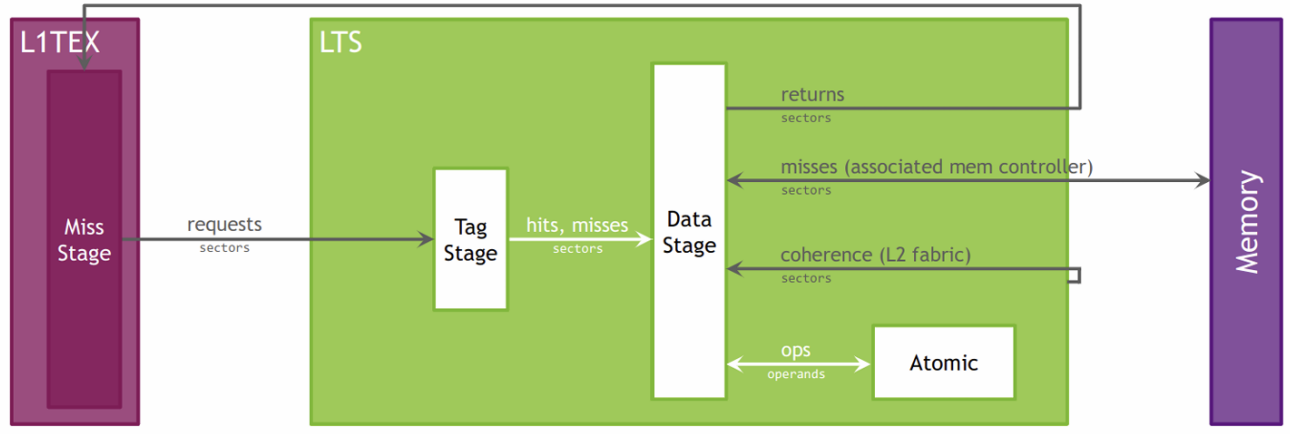
为了提高warp对shared mem的访存带宽，shared mem被拆成32个bank，每个bank带宽是4B/cycle。如果同warp多个thread访问**同一个bank下的不同数据**，则会发生bank conflict。如果是同一个bank下的**同一个数据**，则会触发broadcast。另外，**不同warp之间不存在bank conflict。**

访存效率

一些背景知识

数据传输





SM（一个warp）发出读写req，req请求的数据的单位是sectors（32B）sector之间是串行的，数量根据情况而定，请求会发送到L1

L1
LSUIN: 如果发现req的数据在shared中，则会通过Load/Store Unit传回到核心
如果请求的数据来自于global或者local则会进入Tag Stage。
Tag stage: 在这一阶段L1会查找所需数据，如果miss则进入L2中查找
L2
类似，如果cache hit会逐级返回，miss时则会会通过mem controller从DRAM传回。
#L1如何查找数据 在L1 cache中，数据按照cache line (128B)储存。当L1接收到读写请求时会根据请求的内存地址的**中间部分**的达到index，根据index查找对应的cache line。随后通过内存地址的**高位**的到tag，再与索引到的cache line中的数据的内存地址高位相对比。如果tag对应就视为cache hit，数据被LSU处理返回给核心。

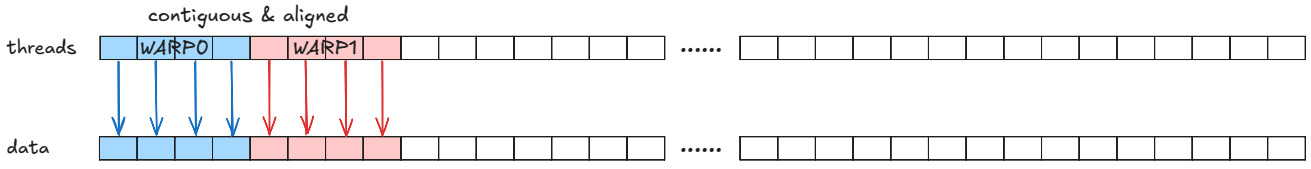
访存和合并访存

#def 合并访存：当指令所需的数据即是请求到的所有数据时被称为combined access。同时可以衍生出合并度以衡量合并的情况。
由上数据的读写机制可知，发出读写请求的最小单位是warp，即32个线程。我们需要优化程序使得每一个warp的读写的sector的数量（因为sector是读写的最小单位）都是完成其访存需求的最小数量。这样就可以**减少内存事务**提高访存效率（即我们只需要读写所需的数据而不浪费）。要达到这一目的我们需要访问的数据的内存是**对齐**的，并且是**连续**的（同warp不同线程可以乱序访问）。

以add和cudaTranspose为例通过NCU分析访存

add

add1



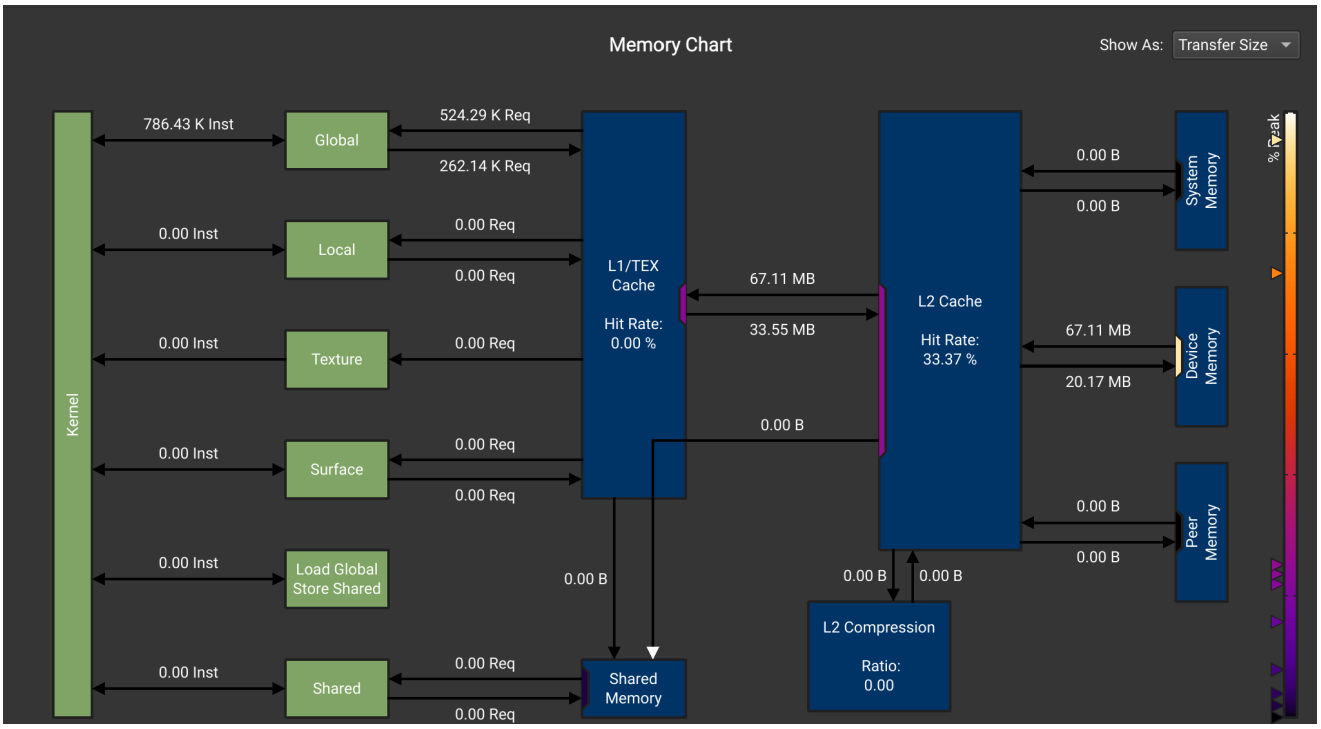

```
void __global__ add1(float *A, float *B, float *C)
{

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

    C[global_tid] = A[global_tid] + B[global_tid];

}
```

如图所示add1的线程访问的内存片段是对齐且连续的所以此时是**合并访存**。

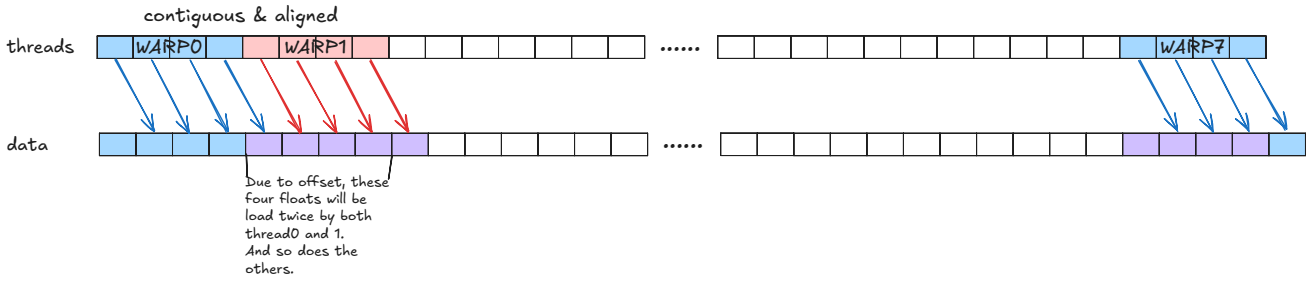


L2 cache hit rate符合推理。因为2次读1次写，最后1次写所需的数据可以在L2中找到，这是因为两次读后计算的结果已经在cache中了。

	Sectors/Req
Local Load	0
Global Load	4
Global Load To Shared Store (access)	0
Global Load To Shared Store (bypass)	0
Surface Load	0
Texture Load	0
Global Store	4

L1的访存数据也符合合并访存的模式，因为读写都是4sectors/rep。即一个warp读取了4个sector(32B)共计$4 \times 32B = 128B$的数据，相当于32个float。这说明一个warp只读取了他所需要的数据，写回同理。

add2




```
void __global__ add2(float *A, float *B, float *C)
{

    int global_tid = blockIdx.x * blockDim.x + threadIdx.x + 1;
//+1 offset, un-aligned memory layout

    C[global_tid] = A[global_tid] + B[global_tid];

}
```

在add2中，由于线程所访问的数据有+1的offset，导致访问不再是对齐的。这样会导致每一个warp需要在第一个sector中多读取一个没用的数据，并且要为第31位数据多读一个sector，这就是非合并访存。

	Sectors	Sectors/Req	Hit Rate	Bytes
Local Load	0	0	0	0
Global Load	2621440	5	13.23	83886080
Global Load To Shared Store (access)	0	0		0
Global Load To Shared Store (bypass)	0	0	-	0
Surface Load	0	0	0	0
Texture Load	0	0	0	0
Global Store	1310720	5	31.15	41943040

从ncu的L1内存表格中可以反映出这次读写的数据都有增加，并且sector/req也从4增加到了5。

cudaTranspose

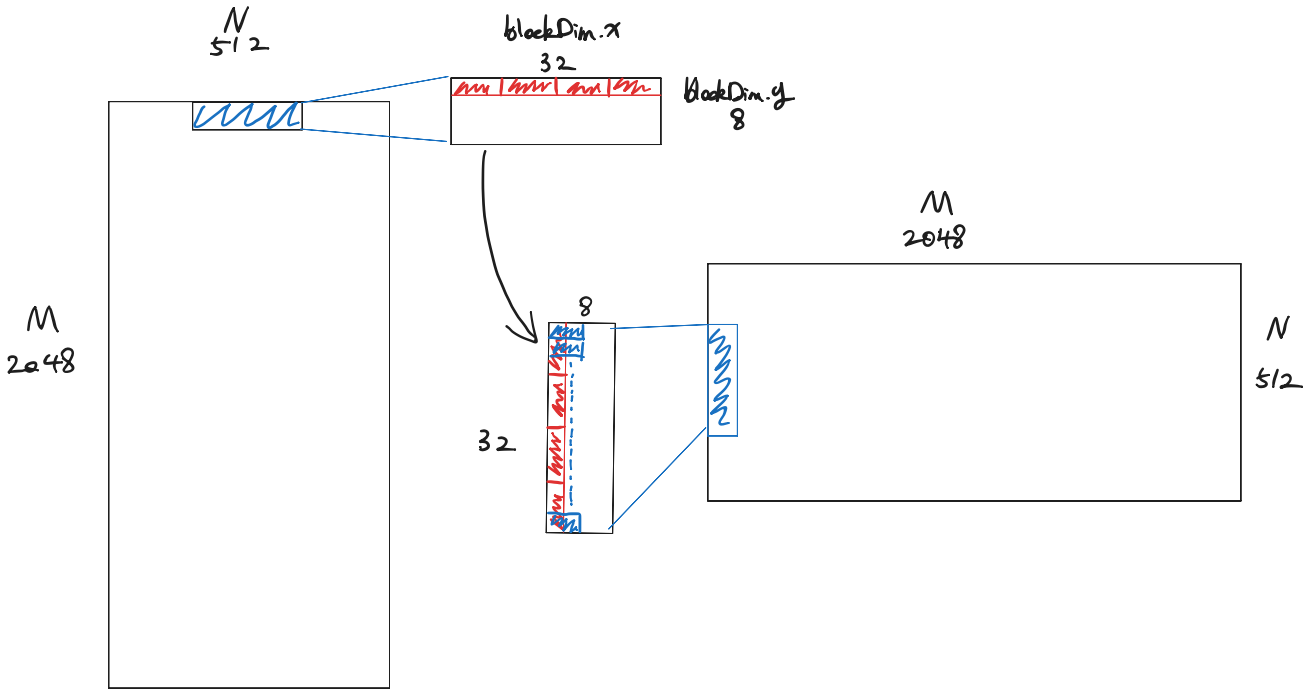
cudaTranspose_naive

naive版本的转置每个线程负责一个元素的转置。只需在每一个线程中计算当前数据转置后的位置即可。

```
__global__ void cudaTranspose_naive(float *d_input, float
*d_output, const int M, const int N)
{
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    d_output[j * M + i] = d_input[i * N + j];
}
```

32×8 / Block



在32×8的Block中，读取时是合并的。但是在写入过程中，由于sector和内存的排布和所需写入部分的排布是正交的，每写入一个数据就需要一个sector的访存，所需一个warp就要写入32个sector。

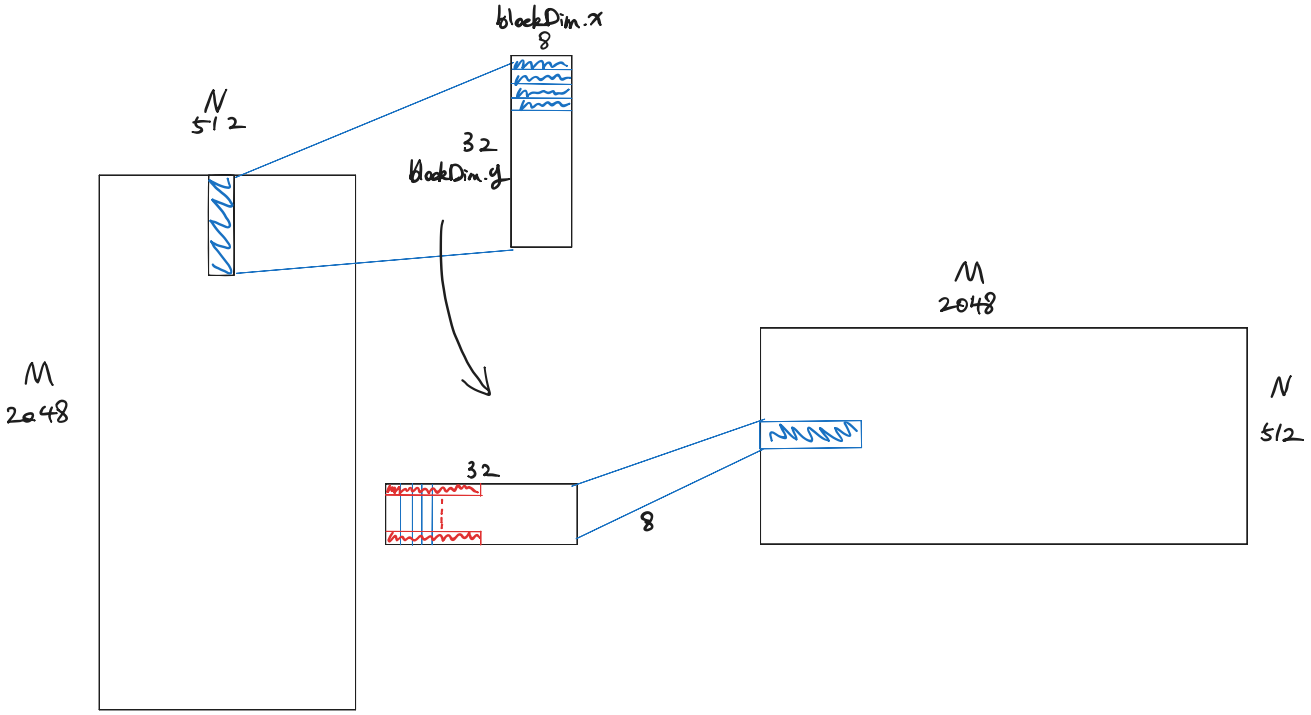
ncu性能分析印证观点：

	Instructions	Requests	Wavefronts	% Peak	Sectors	Sectors/Req
Global Load	32768	32768			131072	4
Global Load To Shared Store (access)	0	0	32768	0.73	0	0
Global Load To Shared Store (bypass)	0	0			0	0
Surface Load	0	0	0	0	0	0
Texture Load	0	0	0	0	0	0
Global Store	32768	32768	268974	6.01	1048576	32
Local Store	0	0	0	0	0	0
Surface Store	0	0	0	0	0	0
Global Reduction	0	0	0	0	0	0
DSMEM Reduction						-
Surface Reduction	0	0	0	0	0	0
Global Atomic ALU	0	0	0	0	0	0
Global Atomic CAS	0					
Surface Atomic ALU	0	0	0	0	0	0
Surface Atomic CAS	0					
Loads	32768	32768	32768	0.73	131072	4
Stores	32768	32768	268974	6.01	1048576	32
Atomics & Reductions	0	0	0	0	0	0
Total	65536	65536	301742	6.74	1179648	18

这个版本的吞吐量是：$35,75.81\%$

8×32 / Block

要想解决写入时访存不合并的问题，我们可以通过改变block的形状，使得每个sector可以写入更多的数据。与此同时blockDim.x不能不能**小到使得读取的sector的完整性**，也就是32B，即8个float。



ncu性能分析印证观点：

	Instructions	Requests	Wavefronts	% Peak	Sectors	Sectors/Req
Global Load	32768	32768			131072	4
Global Load To Shared Store (access)	0	0	32892	3.48	0	0
Global Load To Shared Store (bypass)	0	0			0	0
Surface Load	0	0	0	0	0	0
Texture Load	0	0	0	0	0	0
Global Store	32768	32768	65589	6.94	262144	8
Local Store	0	0	0	0	0	0
Surface Store	0	0	0	0	0	0
Global Reduction	0	0	0	0	0	0
DSMEM Reduction						-
Surface Reduction	0	0	0	0	0	0
Global Atomic ALU	0	0	0	0	0	0
Global Atomic CAS	0					
Surface Atomic ALU	0	0	0	0	0	0
Surface Atomic CAS	0					
Loads	32768	32768	32892	3.48	131072	4
Stores	32768	32768	65589	6.94	262144	8
Atomics & Reductions	0	0	0	0	0	0
Total	65536	65536	98481	10.41	393216	6

虽然这样已经达到了naive算法能达到的最高合并率，但是写入的合并率也只有50%仍有提升空间。并且在读取和写入阶段，sector之间是不连续的，虽然不会成为性能瓶颈（推测）但也是可以优化的。

cudaTranspose_inner_4x4

```
#define FETCH_FLOAT4(ptr) (reinterpret_cast<float4 *>(&(ptr))[0])

__global__ void cudaTranspose_inner_4x4(float *d_input, float
*d_output, const int M, const int N)
```

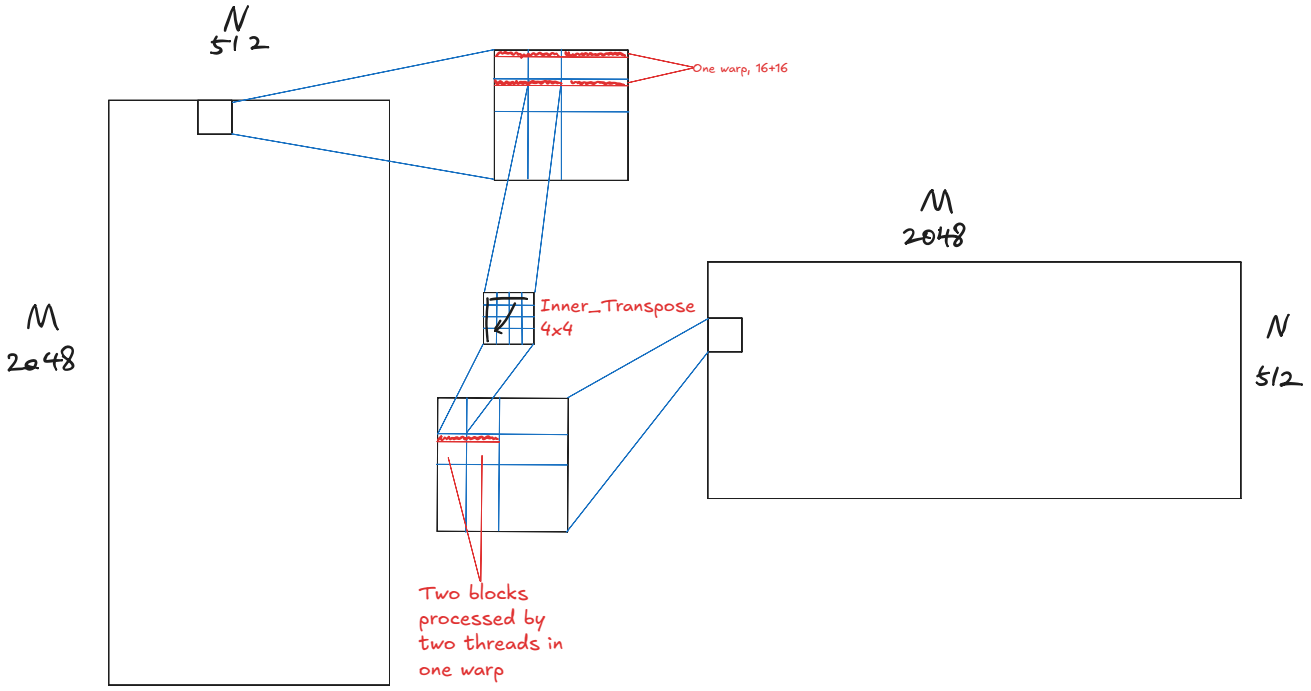
```
{
    float src[4][4];
    float dst[4][4];

    int global_i = blockIdx.y * blockDim.y + threadIdx.y;
    int global_j = blockIdx.x * blockDim.x + threadIdx.x;
    int src_i = (global_i * N + global_j) << 2;
    int dst_i = (global_j * M + global_i) << 2;

    #pragma unroll
    for (int i = 0; i < 4; ++i)
    {
        FETCH_FLOAT4(src[i]) = FETCH_FLOAT4(d_input[src_i + i *
N]);
    }

    #pragma unroll
    for (int i = 0; i < 4; ++i)
    {
        FETCH_FLOAT4(dst[i]) = make_float4(src[0][i], src[1][i],
src[2][i], src[3][i]);
    }

    #pragma unroll
    for (int i = 0; i < 4; ++i)
    {
        FETCH_FLOAT4(d_output[dst_i + i * M]) = FETCH_FLOAT4(dst[i]
[0]);
    }
}
```



直接设置block为16×16，每个线程负责处理4×4的矩阵，在线程内通过寄存器进行转置可以使得读和写的访存合并度都是100%，最大化利用带宽。

ncu性能分析印证观点：

	Instructions	Requests	Wavefronts	% Peak	Sectors	Sectors/Req
Local Load	0	0	0	0	0	0
Global Load	8192	8192			131072	16
Global Load To Shared Store (access)	0	0	16875	2.03	0	0
Global Load To Shared Store (bypass)	0	0			0	0
Surface Load	0	0	0	0	0	0
Texture Load	0	0	0	0	0	0
Global Store	8192	8192	35310	4.24	131072	16
Local Store	0	0	0	0	0	0
Surface Store	0	0	0	0	0	0
Global Reduction	0	0	0	0	0	0
DSMEM Reduction						-
Surface Reduction	0	0	0	0	0	0
Global Atomic ALU	0	0	0	0	0	0
Global Atomic CAS	0					
Surface Atomic ALU	0	0	0	0	0	0
Surface Atomic CAS	0					
Loads	8192	8192	16875	2.03	131072	16
Stores	8192	8192	35310	4.24	131072	16
Atomics & Reductions	0	0	0	0	0	0

此时的吞吐量已经达到了$186, 72.89\%$。