blocksparse_matmul_op_gpu.cu

gemm_blocksparse_?x?x?x?_xprop

```
template <bool Fprop, typename TW, typename TX, typename TY>
__global__ void __launch_bounds__(32) gemm_blocksparse_08x64x08x8_xprop(
    const  int2* __restrict__ Lut,
    const    TW* __restrict__ W,
    const    TX* __restrict__ X,
    TY* Y, int* Lock, int locks, int N)
```

parameter

参数名	参数类型	参数说明
Lut	const int2* __restrict__	包含块稀疏矩阵LUT（Look Up Table）的数组
W	const TW* __restrict__	块稠密权重矩阵
X	const TX* __restrict__	稀疏输入矩阵
Y	TY*	输出矩阵
Lock	int*	用于同步的锁
locks	int	锁的数量
N	int	输入矩阵的大小

procedure

判断Fprop来确定共享内存

在前向传播中，由于卷积核和输入在每个位置进行点乘，因此需要8行8列的数据块。在反向传播中，需要多两行来缓存一些临时变量。因此，在反向传播中需要更多的共享内存。

初始化变量

初始化一些变量和偏移量，用于之后的计算

GEMM（general matrix multiplication）计算

对于每个lut表中的条目，先将对应的W和X从全局内存中加载到共享内存中，然后在共享内存中对它们进行8x64x8的矩阵乘法运算，最后将结果保存到累加寄存器regY中。

锁和累加操作

如果当前线程是第一个到达临界区的线程，就先将 `Lock` 标记为 1，表示当前临界区已被占用。然后，所有线程都需要同步以确保 `Count` 能被正确更新。更新 `Count` 的值，并检查当前已经有多少个线程进入了临界区，如果是第一个进入的线程，则直接将结果写入内存。否则，需要先将之前的结果加载到寄存器中，然后累加结果，并将新结果写入内存中。最后，所有线程再次同步以确保数据写入内存后才将锁标记为 0，以允许其他线程进入临界区。

gemm_blocksparse_?x?x?x?_updat

```
template <typename TX, typename TE, typename TU>
__global__ void __launch_bounds__(32) gemm_blocksparse_08x64x08x8_updat(
    struct Plist<TX,8> X, struct Plist<TE,8> E,
    const  int2* __restrict__ Lut,
    TU* U,
    int params8, int N, int loops, float alpha, float beta)
```

parameter

参数名	参数类型	参数说明
X	Plist<TX,8>	输入矩阵的指针
E	Plist<TE,8>	损失梯度矩阵的指针
Lut	const int2* **restrict**	包含块稀疏矩阵LUT（Look Up Table）的数组
U	TU*	权重矩阵
params8	int	U 的列数，等于 W 的行数
N	int	输入矩阵的大小
loops	int	迭代次数
alpha	float	学习率
beta	float	权重衰减率

procedure

相较于前面的 `xprop` 函数，这个函数涉及到的矩阵乘法的计算顺序是不同的，其计算顺序为 `U = alpha * E * X + beta * U`，即先将稀疏矩阵 `E` 乘上稠密矩阵 `X`，然后将其与稠密矩阵 `U` 相加得到结果。此外，该函数使用了共享内存来提高计算效率。

params 结构体

```
typedef struct bsmm_params
{
    const int* Lut;
    const float* Gate;
    int* Lock;
    //float4* Scratch;
    int blocks;
    int bsize;
    int segments;
    int locks;
    int C;
    int K;
    int N;
    int shared;
    int pcount;
    uint blk_a;
    uint blk_A;
    uint blk_b;
    uint blk_B;
    float alpha;
    float beta;
    CUstream stream;
} bsmm_params;
```

成员名	成员类型	成员说明
Lut	const int*	稀疏矩阵的索引表
Gate	const float*	激活函数的门限值?
Lock	int*	互斥锁数组
blocks	int	稀疏块的数量
bsize	int	稀疏块的尺寸
segments	int	数据分割的数量
locks	int	互斥锁的数量
C	int	卷积核的通道数
K	int	卷积核的数量
N	int	输出特征图的像素数
shared	int	共享内存的大小
pcount	int	不为0的元素数量? 好像默认值都=1
blk_a	uint	稠密矩阵块的大小
blk_A	uint	稠密矩阵行方向的步长
blk_b	uint	稠密矩阵块的大小
blk_B	uint	稠密矩阵列方向的步长
alpha	float	学习率
beta	float	权重衰减率
stream	CUstream	CUDA流对象

BsmmXprop_CN

```
template <bool Fprop, CTYPE(T)>
cudaError_t BsmmXprop_CN(const T* X, const T* W, T* Y, bsmm_params* params)
```

parameter

参数名	参数类型	参数说明
X	const T*	输入张量的指针
W	const T*	权重张量的指针
Y	T*	输出张量的指针
params	bsmm_params*	存储矩阵乘法参数的结构体指针，包括输入输出张量的形状、内存布局、卷积参数、卷积方式等信息。

procedure

实现基于向前传播稀疏矩阵乘法的操作。在这个函数中，会根据输入的参数和数据类型选择一个不同的核函数(`gemm_blocksparse_?x?x?x?_xprop`)进行计算

BsmmUpdat_CN

```
template <CTYPE(T)>
cudaError_t BsmmUpdat_CN(const T* X, const T* E, T* U, bsmm_params* params)
```

parameter

参数名	参数类型	参数说明
X	const T*	输入张量的指针
E	const T*	损失梯度的指针
U	T*	权重梯度的指针
params	bsmm_params*	BSpMM运算的参数，包含了稀疏矩阵LUT，权重矩阵W的指针，以及其他控制参数等。

procedure

实现基于反向传播稀疏矩阵乘法的操作。在这个函数中，会根据输入的参数和数据类型选择一个不同的核函数(`gemm_blocksparse_?x?x?x?_updat`)进行计算

identity_init_CK

```
template <uint BSIZE, uint THREADS>
__global__ void __launch_bounds__(THREADS) identity_init_CK(float* W, const
int2* __restrict__ lut, int CB, int KB, float scale)
```

用于将权重矩阵中每个卷积核的对角线元素设置为一个指定的比例

parameter

参数名	参数类型	参数说明
W	float*	要初始化的权重
lut	const int2* **restrict**	包含CB和KB索引的LUT
CB	int	计算行块大小
KB	int	计算列块大小
scale	float	要乘以的权重初始化缩放因子
BSIZE	uint	每个块的大小
THREADS	uint	线程块大小

procedure

首先从索引表中读取当前线程块所处理的卷积核和输入通道的索引。然后，每个线程使用 `u*THREADS+tid` 计算当前卷积核中的权重矩阵元素的索引 `i`。接下来，如果 `c=k` 并且当前卷积核的索引 `c_b` 与输入通道的索引 `k_b` 对应（即 `(c_b \bmod KB) = (k_b \bmod CB)`），则将 `W[bid\times BSIZE\times BSIZE+i]` 的值设置为 `scale`，否则设置为 `0`。最后，线程将其计算得到的权重矩阵元素写回到权重矩阵中。

---

blocksparse_matmul_gated_op_gpu.cu

LutEntry 结构体

```
typedef struct __align__(16) LutEntry
{
    int offsetX;
    int offsetW;
    float gate;
    float unused;
} LutEntry;
```

成员名	成员类型	成员说明
offsetX	int	在输入矩阵 X 中的偏移量
offsetW	int	在权重矩阵 W 中的偏移量
gate	float	对应的门控值，用于门控循环单元（GRU）网络
unused	float	未使用的浮点数，用于内存对齐

gemm_blocksparse_gated_?x?x?x?_xprop

```
template <bool Fprop, typename TW, typename TX, typename TY>
__global__ void __launch_bounds__(32) gemm_blocksparse_gated_08x64x08x8_xprop(
    const  int2* __restrict__ Lut,
    const float* __restrict__ Gate,
    const    TW* __restrict__ W,
    const    TX* __restrict__ X,
    TY* Y, int* Lock, int locks, int N /* N is in units of groups of 8 elements
each (N/8) */)
```

parameter

参数名	参数类型	参数说明
Lut	const int2* __restrict__	包含块稀疏矩阵LUT（Look Up Table）的数组
Gate	const float* __restrict__	门控值
W	const TW* __restrict__	块稠密权重矩阵
X	const TX* __restrict__	稀疏输入矩阵
Y	TY*	输出矩阵
Lock	int*	用于同步的锁
locks	int	锁的数量
N	int	输入矩阵的大小

procedure

与`gemm_blocksparse?x?x?x?xprop`相比多了一个步骤：使用了一个名为`Gate`的浮点数组作为输入参数，表示每个权重的门控系数。同时，在LUT表构建时，将Gate值乘以该LUT条目对应的权重值，以实现门控。此外，还在计算过程中加入了一个新的累加操作，用于将不同门控系数的权重分别计算得到的结果加起来。

gemm_blocksparse_gated_?x?x?x?_updat

```
template <typename TX, typename TE, typename TU>
__global__ void __launch_bounds__(32) gemm_blocksparse_gated_08x64x08x8_updat(
    struct Plist<TX,8> X, struct Plist<TE,8> E,
    const  int2* __restrict__ Lut,
    const float* __restrict__ Gate,
    TU* U,
    int params8, int N, int loops, float alpha, float beta)
```

parameter

参数名	参数类型	参数说明
X	Plist<TX,8>	输入矩阵的指针
E	Plist<TE,8>	损失梯度矩阵的指针
Lut	const int2* __restrict__	包含块稀疏矩阵LUT（Look Up Table）的数组
Gate	const float* __restirct__	门控值
U	TU*	权重矩阵
params8	int	U 的列数，等于 W 的行数
N	int	输入矩阵的大小
loops	int	迭代次数
alpha	float	学习率
beta	float	权重衰减率

procedure

由于加入了门控，所以计算更新时使用的公式也有所变化，需要根据门控的值进行加权计算。因此，在计算更新时，`gemm_blocksparse_gated_08x64x08x8_updat`需要使用`E * (alpha * Gate) + beta * U`来更新权重矩阵`U`。

BsmmGatedXprop_CN

```
template <bool Fprop, CTYPE(T)>
cudaError_t BsmmGatedXprop_CN(const T* X, const T* W, T* Y, bsmm_params* params)
```

parameter

参数名	参数类型	参数说明
X	const T*	输入张量的指针
W	const T*	权重张量的指针
Y	T*	输出张量的指针
params	bsmm_params*	存储矩阵乘法参数的结构体指针，包括输入输出张量的形状、内存布局、卷积参数、卷积方式等信息。

procedure

基本与没有gate的没有区别

BsmmGatedUpdat_CN

```
template <CTYPE(T)>
cudaError_t BsmmGatedUpdat_CN(const T* X, const T* E, T* U, bsmm_params* params)
```

parameter

参数名	参数类型	参数说明
X	const T*	输入张量的指针
E	const T*	损失梯度的指针
U	T*	权重梯度的指针
params	bsmm_params*	BSpMM运算的参数，包含了稀疏矩阵LUT，权重矩阵W的指针，以及其他控制参数等。

procedure

基本与没有gate的没有区别

blocksparse_matmul_op_gpu.cu

hgemm_blocksparse_nx_dsd

```
cudaError_t hgemm_blocksparse_nx_dsd(const ehalf* X, const ehalf* W, ehalf* Y,
bsmm_params* params, uint op)
```

parameter

参数名	参数类型	参数说明
X	const ehalf*	乘数矩阵的数据指针
W	const ehalf*	权重矩阵的数据指针
Y	ehalf*	结果矩阵的数据指针
params	bsmm_params*	存储稀疏矩阵乘法相关参数的结构体指针
op	uint	表示矩阵乘法的类型，0表示普通乘法(OP_N)，1表示转置乘法(OP_T)

procedure

根据参数params计算kernel的grid和block尺寸，在CUDA设备上调用核函数(`hgemm_blocksparse_64x?x?_nx_dsd`)执行稀疏矩阵乘法操作，最后返回CUDA的错误码，如果为0则表示执行成功，否则表示执行失败

hgemm_blocksparse_tn_dds

```
cudaError_t hgemm_blocksparse_tn_dds(const ehalf* X, const ehalf* E, ehalf* U,
bsmm_params* params)
```

parameter

参数名	参数类型	参数说明
X	const ehalf*	乘数矩阵的数据指针
E	const ehalf*	损失矩阵的数据指针
U	ehalf*	权重矩阵的数据指针
params	bsmm_params*	存储稀疏矩阵乘法相关参数的结构体指针
op	uint	表示矩阵乘法的类型，0表示普通乘法(OP_N)，1表示转置乘法(OP_T)

procedure

与`hgemm_blocksparse_nx_dsd`主要的区别就在于矩阵乘法的操作顺序不同，即一个是做了转置，一个没有。具体来说，`hgemm_blocksparse_nx_dsd`函数实现的是普通的稀疏矩阵乘以稠密矩阵的操作，而`hgemm_blocksparse_tn_dds`函数则是转置后的稀疏矩阵乘以稠密矩阵的操作。

BlocksparseFeatureReduceNC

```
bool BlocksparseFeatureReduceNC(CUstream stream, ehalf* Y, const struct
Plist<ehalf,8>* X8, uint params, uint C, uint N, uint bshift, uint norm_type)
```

parameter

参数名	参数类型	参数说明
stream	CUstream	CUDA stream 对象，用于在GPU上执行异步操作。
Y	ehalf*	特征值的归一化结果，是一个大小为C的数组。
X8	const struct Plist<ehalf,8>*	稀疏矩阵，结构体 Plist 包含稀疏矩阵的 LUT（Look Up Table）和数据数组，其中数据数组每个元素包含8个ehalf类型的数值。
params	uint	用于确定每个线程块（block）所处理的列数，计算方法为 32 / params。
C	uint	稀疏矩阵的列数。
N	uint	稀疏矩阵的行数。

参数名	参数类型	参数说明
N	uint	稀疏矩阵的行数。
bshift	uint	用于确定线程块大小的移位量。在该函数中，bshift的值必须是5或6。
norm_type	uint	归一化的方式，可以是MAX_NORM或L2_NORM。其中MAX_NORM用于将所有值除以该行的最大值，L2_NORM用于将每行的值除以其L2范数。

procedure

计算一个稀疏矩阵的特征值（Feature）的归一化值
首先计算了需要的线程块（block）数量和每个线程块内线程数量，然后根据输入参数计算出每个线程块需要处理的列数。接着根据bshift参数的不同值选择不同大小的线程块，然后调用不同的CUDA kernel函数进行计算。函数返回true表示计算完成。

hGemmTN

```
bool hGemmTN(CUstream stream, const ehalf* A, const ehalf* B, float* C, uint M,
uint N, uint K, uint blk_a, uint blk_b, uint blk_A, uint blk_B, uint accumulate,
float scale)
```

parameter

参数名	参数类型	参数说明
stream	CUstream	CUDA流
A	const ehalf*	输入矩阵 A
B	const ehalf*	输入矩阵 B
C	float*	输出矩阵 C
M	uint	矩阵 A 的行数
N	uint	矩阵 B 的列数
K	uint	矩阵 A 的列数，也是矩阵 B 的行数
blk_a	uint	A 矩阵块的行数
blk_b	uint	B 矩阵块的列数
blk_A	uint	A 矩阵块的列数
blk_B	uint	B 矩阵块的行数
accumulate	uint	是否累加到输出矩阵 C 上，0 表示不累加，1 表示累加
scale	float	缩放因子

procedure

实现的是一个矩阵乘法的操作，将矩阵 A 和矩阵 B 进行转置后相乘，结果保存在矩阵 C 中，支持累加和缩放操作。参数 `blk_a、blk_b、blk_A、blk_B` 是为了将输入矩阵按照一定的块大小进行拆分，以便在 GPU 上并行计算，加快矩阵乘法的计算速度。

CUDA_ARCH >= 700

blocksparse_feature_reduce_nc

```
template <uint BSIZE, uint NSIZE, uint NORM>
__global__ void __launch_bounds__(256) blocksparse_feature_reduce_nc(
    const struct Plist<ehalf,8> X8, ehalf* Y, uint N, uint C)
```

parameter

参数名	参数类型	参数说明
X8	Plist<ehalf,8> struct	包含输入稀疏矩阵数据的 Plist（Plist 是一个用于在 kernel 函数中传递指针数组的模板类）
Y	ehalf*	输出稀疏矩阵数据
N	uint	稀疏矩阵行数
C	uint	稀疏矩阵列数
BSIZE	uint	每个线程块（block）处理的列数
NSIZE	uint	每个线程块（block）处理的行数
NORM	uint	归一化方式，取值为 0 或 1，其中 0 表示 L2 归一化，1 表示 L∞ 归一化

procedure

实现了对一个稀疏矩阵的特征进行归一化

stg_64x?x?_nx

```
template <bool N64>
__device__  __noinline__  void stg_64x64x64_nx(ehalf* Y, uint offsetY, uint
loadY, uint N, uint K, uint n, uint i)
```

parameter

参数名	参数类型	参数说明
Y	ehalf*	存储计算结果的全局内存地址
offsetY	uint	Y 的偏移量
loadY	uint	共享内存中存储 Y 数据的地址
N	uint	数据总数
K	uint	数据维度
n	uint	当前处理的数据下标
i	uint	当前处理的线程块内下标

procedure

实现一个矩阵乘法中的存储操作，将计算后的结果存储到设备端的内存中。其具体步骤是将计算结果按照一定的规律存储到指定的内存地址中，同时考虑数据类型为half的情况

循环 `j` 两次，表示处理一次共 `64` 个数据；判断是否需要继续处理，`N64` 为 `true` 或者 `n + i * 16 + j * 32 < N` 时继续；从共享内存中读取两个 `float4` 类型的数据；将读取的两个 `float4` 数据按位相加；将相加后的结果转换成 `half4` 类型，并存储到全局内存中。

red_64x?x?_nx

```
template <bool N64>
__device__  __noinline__  void red_64x64x64_nx(ehalf* Y, uint offsetY, uint
loadY, uint N, uint K, uint n, uint i, uint stdC)
```

parameter

参数名	参数类型	参数说明
Y	ehalf*	指向输出矩阵的指针
offsetY	uint	输出矩阵的起始偏移量
loadY	uint	指向输入矩阵的指针
N	uint	输入矩阵的行数
K	uint	输出矩阵的列数
n	uint	输入矩阵的起始行
i	uint	输出矩阵的起始列
stdC	uint	输入矩阵的列步幅

procedure

将一个64x64的矩阵的每个元素相加得到一个64维的向量，并将该向量中每16个元素进行求和，得到一个大小为4的向量。

循环变量`j`和`k`分别在0~1之间循环，遍历矩阵的4个子块。利用CUDA中共享内存`__shared__`特性，将64x64的矩阵按行拆分成8个大小为64的向量，每个向量由两个32维向量拼接而成。然后将这8个向量相应元素相加，得到4个32维向量。最后将这4个向量中的每个元素类型转换为`half2`类型。根据参数`offsetY`和`K`计算出写入全局内存中的位置，将得到的`half2`类型的向量与已经存在全局内存中的值进行求和，并将结果写回全局内存。如果参数`N64`为真或`n + j*32 + k*8 < N`，则调用函数`reduce_half2`将每个32维向量中的每16个元素相加，得到4个元素的向量。如果`N64`为假且`n + j*32 + k*8 >= N`，则不进行求和操作。

hgemm_blocksparse_64x?x?_nx_dsd

```
template <uint OP_B, bool N64, bool GATED>
__global__ void __launch_bounds__(256) hgemm_blocksparse_64x64x64_nx_dsd(
    const uint2* __restrict__ Lut,
    const float* __restrict__ Gate,
    const ehalf* __restrict__ X,
    const ehalf* __restrict__ W,
          ehalf*              Y,
    uint* Lock, uint locks, uint N, uint C, uint K, uint blk_a, uint blk_b, uint
blk_N)
```

parameter

参数名	参数类型	参数说明
Lut	const uint2* __restrict__	包含稀疏矩阵块的Look-Up Table
Gate	const float* __restrict__	门限值
X	const ehalf* __restrict__	稀疏矩阵
W	const ehalf* __restrict__	权重矩阵W
Y	ehalf*	结果矩阵Y
Lock	uint*	实现多线程的锁定机制
locks	uint	锁的数量
N	uint	入矩阵的行数
C	uint	输入矩阵的列数
K	uint	输出矩阵的列数
blk_a	uint	块A的大小
blk_b	uint	块B的大小
blk_N	uint	块N的大小
OP_B	uint	矩阵乘法运算中B矩阵的操作类型（OP_N或OP_T）
N64	bool	控制N的处理方式
GATED	bool	是否使用门向量

procedure

是用于进行矩阵乘法运算的

该函数通过对输入的LUT（Look-Up Table）进行分块，依次处理每个块。在每个块中，首先将LUT和Gate（门）数据预取到共享内存中，然后进行多个矩阵乘累加操作，最后将结果写回到全局内存中。

hgemm_blocksparse_?x?x64_tn_dds

```
template <bool N64, bool GATED>
__global__ void __launch_bounds__(128,6) hgemm_blocksparse_32x32x64_tn_dds(
    struct Plist<ehalf,8> X,
    struct Plist<ehalf,8> DY,
    ehalf*                DW,
    const uint2* __restrict__ Lut,
    const float* __restrict__ Gate,
    uint params8, uint N, uint C, uint K, uint loops, uint accumulate)
```

parameter

参数名	参数变量	参数说明
X	struct Plist<ehalf,8>	半精度数据，结构体类型，包含8个ehalf型的指针，每个指针指向一个半精度矩阵的起始地址。
DY	struct Plist<ehalf,8>	半精度权重，结构体类型，包含8个ehalf型的指针，每个指针指向一个半精度矩阵的起始地址。
DW	ehalf*	输出矩阵的地址，ehalf为半精度数据类型。
Lut	const uint2* __restrict__	卷积核表，为一个包含若干个uint2类型元素的数组，每个uint2元素包含两个uint型的数据，分别代表X和DY中需要用于计算的矩阵的下标。
Gate	const float* __restrict__	Gate值数组，为一个包含若干个float型数据的数组，数组长度为Lut数组的长度。
params8	uint	用于存储卷积核参数的uint型变量，代表一组8个半精度参数。
N	uint	输入矩阵的高（行数）。
C	uint	输入矩阵的宽（列数）。
K	uint	输出矩阵的宽（列数）。
loops	uint	循环次数，代表每个半精度参数（params8）的使用次数。
accumulate	uint	累加标志，当值为0时，表示DW中的数据需要被覆盖，否则DW中的数据需要被累加。

procedure

计算稀疏矩阵乘法

定义共享内存`hShare`，用于保存输入矩阵和权重矩阵的数据以及计算过程中的临时结果。获取当前线程在线程块中的索引`tid`和线程块的索引`bid`，以及对应的门控变量`gate`（如果启用了门控）。如果`gate`不为0，则进行稀疏矩阵乘法计算，否则直接将输出矩阵清零。根据`Lut`数组中存储的稀疏矩阵的行索引和列索引，以及输入矩阵和权重矩阵的维度信息，计算出输入矩阵和权重矩阵的偏移量，并将它们加载到共享内存中。使用CUDA的半精度（16位）浮点数运算函数，将共享内存中的输入矩阵和权重矩阵中的数据加载到片上寄存器中，并进行矩阵乘法计算。将计算结果保存到共享内存中，并使用半精度浮点数运算函数将共享内存中的数据写回到输出矩阵中。重复步骤4~6，直到将所有的输入矩阵和权重矩阵都加载到共享内存中并进行了矩阵乘法计算。将共享内存中的输出矩阵数据写回到全局内存中。如果启用了累加选项，则将计算结果累加到原有的输出矩阵中。

CUDA_ARCH < 700

blocksparse_feature_reduce_nc

```
template <uint BSIZE, uint NSIZE, uint NORM>
__global__ void __launch_bounds__(256) blocksparse_feature_reduce_nc(
    const struct Plist<ehalf,8> X8, ehalf* Y, uint N, uint C)
```

parameter

参照CUDA_ARCH >= 700

procedure

参照CUDA_ARCH >= 700

hgemm_64x64x32_tn

```
template <bool M64, bool ACCUMULATE>
__global__ void __launch_bounds__(256) hgemm_64x64x32_tn(
    const ehalf* A,
    const ehalf* B,
          float* C,
    uint M, uint N, uint K, uint blk_a, uint blk_b, float scale)
```

parameter

参照CUDA_ARCH >= 700

procedure

参照CUDA_ARCH >= 700

hgemm_blocksparse_64x?x?_nx_dsd

```
template <uint OP_B, bool N64, bool GATED>
__global__ void __launch_bounds__(256) hgemm_blocksparse_64x64x64_nx_dsd(
    const uint2* __restrict__ Lut,
    const float* __restrict__ Gate,
    const ehalf* __restrict__ X,
    const ehalf* __restrict__ W,
          ehalf*              Y,
    uint* Lock, uint locks, uint N, uint C, uint K, uint blk_a, uint blk_b, uint
blk_N)
```

parameter

参照CUDA_ARCH >= 700

procedure

参照CUDA_ARCH >= 700

hgemm_blocksparse_?x?x64_tn_dds

```
template <bool N64, bool GATED>
__global__ void __launch_bounds__(256,3) hgemm_blocksparse_64x64x64_tn_dds(
    struct Plist<ehalf,8> X,
    struct Plist<ehalf,8> DY,
    ehalf*                DW,
    const uint2* __restrict__ Lut,
    const float* __restrict__ Gate,
    uint params8, uint N, uint C, uint K, uint loops, uint accumulate)
```

parameter

参照CUDA_ARCH >= 700

procedure

参照CUDA_ARCH >= 700