

Weekly Research Progress Report

Student: 从 兴 Date: 2024/05/06 - 2023/05/21

List of accomplishments this week

- (1) 第一周集中精力准备申请博士的资料，准备答辩 PPT，做相关代码机试题。
- (2) HYCOM 已经在曙光的机器上跑通 example 模型，测试并编写相关的编译、执行和作业提交脚本。
- (3) 放弃更改 TileSpMV 源代码，重新编写 BWM(Block Wise Matrix) 的代码，目前基本框架已经搭建完成。
- (4) 阅读论文《DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication》

Paper summary

Name: DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication

Motivation: 尽管已有大量研究改善了 SpMV 的内存访问性能，但实验发现内积计算仍然占据了大量的计算开销（这一方面在现有工作中被忽略）。此外，现有方法无法充分利用带宽（无法使带宽性能接近峰值）。与此同时，GPU 等并行处理器引入了专用的矩阵乘法累加 (MMA) 单元，显著提升了稀疏矩阵-密集矩阵乘法的性能。然而，由于稀疏矩阵的非零元素分布非常不规则，而 MMA 单元需要严格的数据布局才能充分利用硬件资源，因此，将 MMA 单元应用到 SpMV 中仍存在挑战。

Solution: 该论文采用了三种方式来对基于 MMA 单元的 SpMV 计算进行优化，首先，设计了新的数据结构，分析稀疏矩阵中每行的非零元素分布，将行分为长、中、短三类，并对不同的类别进行以适应 MMA 单元的调整。同时，由于设计了新的数据结构，所以，针对不同类别的行分别设计了不同的计算策略。最后，还使用了一些优化技术，比如：缓存旁路技术来提高向量 x 在缓存中的命中率，采用自适应分工作负载分配策略，从而进一步提高计算效率。

Related to us: 这篇文章的思路很新奇，从充分利用硬件的角度出发，针对某个新添的硬件进行设计存储结构和计算方式，从而达到加速计算的目的。这种方法虽然新

奇，但是，编码涉及大量的硬件语言，甚至涉及到了汇编指令，因此，并不具有通用性。而且，使用硬件语言来获得计算的加速，不知道能不能经历起时间的考验！

Work summary

(1) HYCOM 的相关进度

(a) 曙光机器的配置

与天河上不同的是，曙光机器支持的软件更多，所以，更换了 fortran 和 MPI 编译器，改为使用 Intel 相关的编译器进行 HYCOM 源码的编译。

在曙光机器上需要添加的软件包为：

```
mpi/intelmpi/2020.1.217  
compiler/intel/2020.1.217
```

(b) 配置文件

配置文件需要有特别的命名规则：机器类型_采用的执行方式：one/mpi/omp/ompi，本次测试使用的曙光机器，采用的类型是 openMP+MPI。因此，命名为：*sg_ompi*，配置文件中的内容为：

```
FC = mpiifort  
  
FCFLAGS = -traceback -xHost -O3 -fp-model precise -no-fma -ftz -align \  
          array64byte -qopenmp -r8 -warn nogeneral \  
          -diag-disable 10212 -mcmodel=small  
  
CC = mpiicc  
  
CFLAGS = -traceback -xHost -O -mcmodel=small  
  
CPP = cpp -P  
  
CPPFLAGS = -DREAL8 -DMPI -DENDIAN_IO -DNAN2003 -DTIMER -DRELO $(CPP_EXTRAS)  
  
LD = $(FC)  
  
LDFLAGS = -V $(FCFLAGS)  
  
  
SHELL = /bin/sh  
  
RM = \rm -f  
  
.c.o : $(CC) $(CPPFLAGS) $(CFLAGS) -c $*.c  
.F90.o : $(FC) $(CPPFLAGS) $(FCFLAGS) -c $*.F90
```

(c) 编译 HYCOM 源文件脚本

这里有个问题，源码中自带的 make 脚本中，默认的海洋模拟是局部的，在用到 example 中时，会发生错误，因为 example 是全球的，需要进行一个参数的修改。

```
#!/bin/csh

set echo

cd $cwd

setenv ARCH sg

setenv TYPE `echo $cwd | awk -F"_" '{print $NF}'` 

echo "ARCH = " $ARCH " TYPE = " $TYPE

if (! -e ./config/${ARCH}_${TYPE}) then

    echo "ARCH = " $ARCH " TYPE = " $TYPE " is not supported"

    exit 1

endif

setenv OCN_EOS -DEOS_17T ## EOS 17-term

setenv OCN_GLB -DARCTIC ## global tripolar simulation 修改此处

setenv OCN_KAPP ""

setenv OCN_MISC "-DMASSLESS_1MM"

setenv CPP_EXTRAS "${OCN_SIG} ${OCN_EOS} ${OCN_GLB} ${OCN_KAPP} ${OCN_MISC}"

make hycom ARCH=$ARCH TYPE=$TYPE
```

(d) 提交 HYCOM 作业的脚本

在 HYCOM-example 中，默认提供的作业脚本是使用 PBS 调度器写的，因此，此处需要针对曙光机器的 Slurm 调度器进行重新的修改。

还有一个 bug 是，在 example 中的 *patch.input* 文件中，由于 HYCOM-src 进行了更新，因此在 *patch.input* 文件中，缺少了一个参数配置，需要在运行 example 之前，将该参数加入到 *patch.input* 文件中。

缺少的参数为： 0 'mtracr' = number of diagnostic tracers

脚本为：

```
#!/bin/csh

#SBATCH --job-name=HY01525

#SBATCH --output=HY01525.log
```

```

#SBATCH --error=HY01525.log

#SBATCH --ntasks=1525      # MPI 进程的数量

#SBATCH --nodes=48

#SBATCH --ntasks-per-node=32

#SBATCH --cpus-per-task=2

#SBATCH --exclusive

#SBATCH --partition=xahcnormal

source /work/home/congxing/HYCOM/examples/GLBT0.08/expt_73.7/Linux/load.sh

setenv NOMP 2

setenv NMPI 1525

cd      /work/share/acjhjlnmhg/HYCOM/DATA/GLBT0.08/expt_73.7/DATA

rm hycom patch.input

setenv A "g"

setenv B "h"

setenv Y01 "001"

setenv YXX "001"

echo "Y01 = $Y01 YXX = $YXX A =" ${A} "B =" ${B}

/bin/cp /work/home/congxing/HYCOM/src_sg_ompi/hycom ./

setenv NPATCH `echo $NMPI | awk '{printf("%05d", $1)}'` 

if (-e patch.input_${NPATCH}s8) then

  ln  patch.input_${NPATCH}s8 patch.input

else

  echo "ERROR - requested patch file" patch.input_${NPATCH}s8 "does not exist"

  exit

endif

setenv OMP_NUM_THREADS $NOMP

mpirun -np $NMPI ./hycom

```

(e) HYCOM01525 运行结果

```
timer statistics, processor 293 out of 1525
-----
xcaget    calls = 2324    time = 5.87015    time/call = 0.00252588
xcaput    calls = 330     time = 0.22571    time/call = 0.00068396
xcsum     calls = 3758    time = 197.19908   time/call = 0.05247448
xcmaxr    calls = 7047    time = 320.04184   time/call = 0.04541533
xctilr    calls = 83706   time = 1149.04671  time/call = 0.01372717
zaio**    calls = 68      time = 7.11530    time/call = 0.10463669
zaiord    calls = 330     time = 76.45293   time/call = 0.23167556
zaiowr    calls = 2324   time = 444.27128  time/call = 0.19116664
xc****    calls = 1       time = 3342.55012  time/call = 3342.55011995
cnuity    calls = 720     time = 256.02532  time/call = 0.35559072
tsadvc    calls = 720     time = 268.26082  time/call = 0.37258448
momtum    calls = 720     time = 402.06764  time/call = 0.55842727
barotp    calls = 720     time = 593.22508  time/call = 0.82392372
thermf    calls = 720     time = 28.88789   time/call = 0.04012208
ic****    calls = 720     time = 140.44647  time/call = 0.19506454
mx****    calls = 720     time = 383.37529  time/call = 0.53246569
conv**    calls = 720     time = 0.00005   time/call = 0.00000006
diapf*    calls = 720     time = 0.00005   time/call = 0.00000006
hybgen    calls = 720     time = 167.46184  time/call = 0.23258588
overtn    calls = 1       time = 1.65750   time/call = 1.65749601
archiv    calls = 25      time = 451.80732  time/call = 18.07229282
incupd    calls = 720     time = 0.00037   time/call = 0.00000052
aslsav    calls = 720     time = 20.92804  time/call = 0.02906672
asseln    calls = 720     time = 26.02908  time/call = 0.03615149
total     calls = 1       time = 3031.60495 time/call = 3031.60495153
```

(2) BWM 的相关进度

矩阵第一层结构：使用 COO 格式对每个分块进行存储。

矩阵第二层结构：构建 WISE_BLOCK 基类，根据每个分块中非零元素的特征不同，选择合适的存储格式，不同的存储格式也就对应基于基类实现的不同种类的分块，在每种分块类中实现内存的优化。由于上层结构使用 COO 格式来进行存储和指示，因此，在将每个子块数据进行存储时，使用 Map 来加快对每个子块的读取，不再采用遍历的方式：`#define WISE_BLOCK_MAP map<tuple<int, int>, shared_ptr<WISE_BLOCK> >`

```
struct GLOBAL_BLOCK{
    // 上层结构使用COO格式进行存储
    int m;           // 矩阵的行数
    int n;           // 矩阵的列数
    int nnz;         // 矩阵的非零元素个数
    int gb_nb;       // 划分子块后，子块的个数
    int gb_row;      // 划分子块后，全局的行数
    int gb_col;      // 划分子块后，全局的列数
    int gb_nnz;      // 划分子块后，拥有非零元素子块的个数
    vector<int> gb_row_index; // 子块的行索引
    vector<int> gb_col_index; // 子块的列索引
    vector<int> gb_nnz_cnt; // 每个子块内非零元素个数
    GLOBAL_BLOCK(int m, int n, int nnz) : m(m), n(n), nnz(nnz) {
        gb_row = (m + BLOCK_SIZE - 1) / BLOCK_SIZE; // 划分后的全局行数
        gb_col = (n + BLOCK_SIZE - 1) / BLOCK_SIZE; // 划分后的全局列数
        gb_nb = gb_row * gb_col;
        gb_nnz = 0; // 初始子块数为0，可以根据需要更新
    }
};

class WISE_BLOCK {
public:
    virtual ~WISE_BLOCK() = default;
};

class CSR_BLOCK : public WISE_BLOCK {
public:
    // 不使用内存优化的参数
    int wb_row;           // WISE_BLOCK的行数
    int wb_nnz;          // WISE_BLOCK的非零元素个数
    // 使用内存优化的数组
    int* row_ptr;         // wise_block行指针数组指针
    int* col_index;       // wise_block列索引数组指针
    double* val;          // wise_block非零元素值数组指针
    double* val_img;      // wise_block非零元素虚部数组指针
};

class COO_BLOCK : public WISE_BLOCK {
public:
    // 不使用内存优化的参数
    int wb_nnz;          // WISE_BLOCK的非零元素个数
    // 使用内存优化的数组
    int* row_index;       // wise_block行索引数组指针
    int* col_index;       // wise_block列索引数组指针
    double* val;          // wise_block非零元素值数组指针
    double* val_img;      // wise_block非零元素虚部数组指针
};
```

图 1 两层分块结构的实现

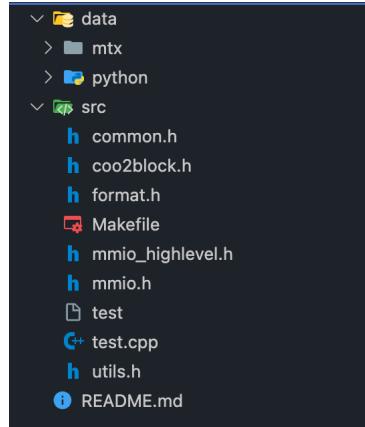


图 2 目前的代码框架

(3) DASP 论文笔记

(a) MMA 实现的矩阵乘法



Listing 1 The FP64 precision mma_m8n8k4 instruction.

```
1: % __device__ __forceinline__
2: void mma_m8n8k4(double *acc, double &frag_a,
3:                   double &frag_b){
4:     asm volatile(
5:         "mma.sync.aligned.m8n8k4.
6:          row.col.f64.f64.f64.f64"
7:         " { %0, %1 }, "
8:         " { %2 }, "
9:         " { %3 }, "
10:        " { %0, %1 };"
11:        : "+d"(acc[0]), "+d"(acc[1]),
12:          "d"(frag_a), "d"(frag_b));
13: }
```

图 3 利用 MMA 实现稠密矩阵乘法

行的种类划分方式：

- 长行: $Row_len > MAX_LEN$
- 中行: $4 < Row_len \leq MAX_LEN$
- 短行: $Row_len \leq 4$

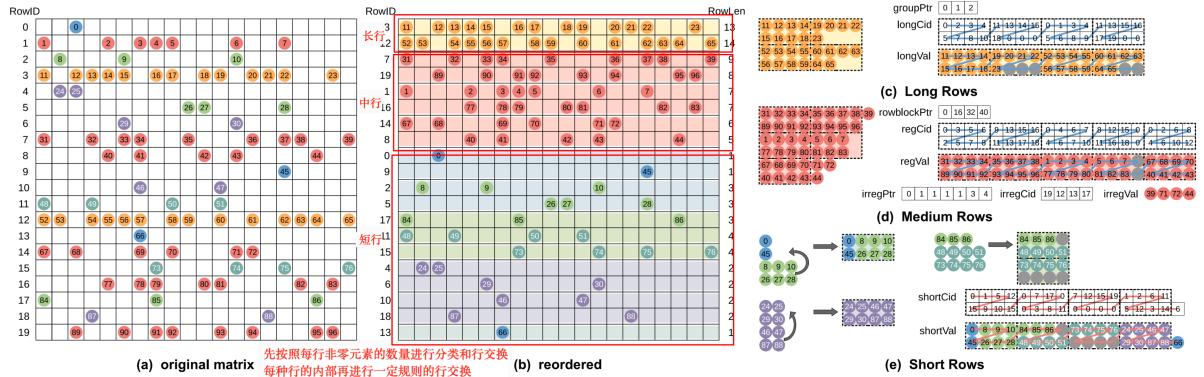
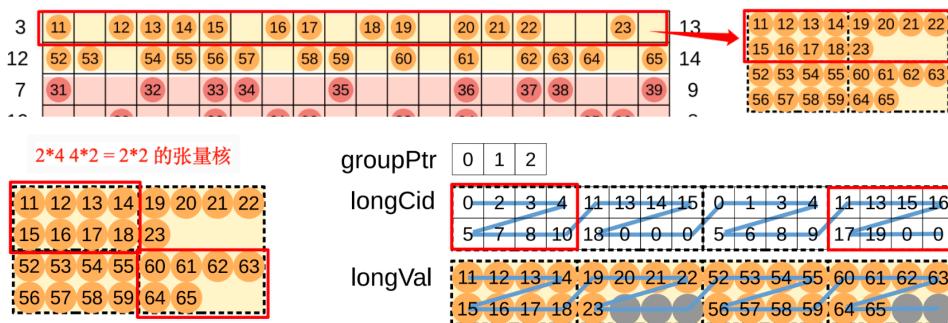


图 4 三种行种类以及存储格式总览

(b) 长行对应的存储结构和计算方式



(c) Long Rows

图 5 长行存储格式示意图

Algorithm 2 A pseudocode of warp-level Long-Rows SpMV.

```

1: for laneid = 0 to 31 in parallel do
2:   fragY[2], fragA, fragX ← 0
3:   idx = (3 + laneid) + (laneid > 2) × MMA_K
4:   for i = 0 to 1 do
5:     fragA ← longVal[offsetA + idx]
6:     fragX ← valX[longCid[offsetA + idx]]
7:     mma.m8n8k4(fragY, fragA, fragX)
8:     idx += MMA_M × MMA_K
9:   end for
10:  fragY[0] += __SHFL_DOWN_SYNC(0xffffffff, fragY[0], 9)
11:  fragY[0] += __SHFL_DOWN_SYNC(0xffffffff, fragY[0], 18)
12:  fragY[1] += __SHFL_DOWN_SYNC(0xffffffff, fragY[1], 9)
13:  fragY[1] += __SHFL_DOWN_SYNC(0xffffffff, fragY[1], 18)
14:  fragY[0] += __SHFL_SYNC(0xffffffff, fragY[1], 4)
15:  if laneid == 0 then
16:    warpVal[warpid] ← fragY[0]
17:  end if
18:  threadVal ← 0
19:  for i = laneid to row_warp_len step WARP_SIZE do
20:    threadVal += warpVal[offsetW + i]
21:  end for
22:  threadVal = warpReduceSum(threadVal)
23:  if laneid == 0 then
24:    valY[warpid] ← threadVal
25:  end if
26: end for

```

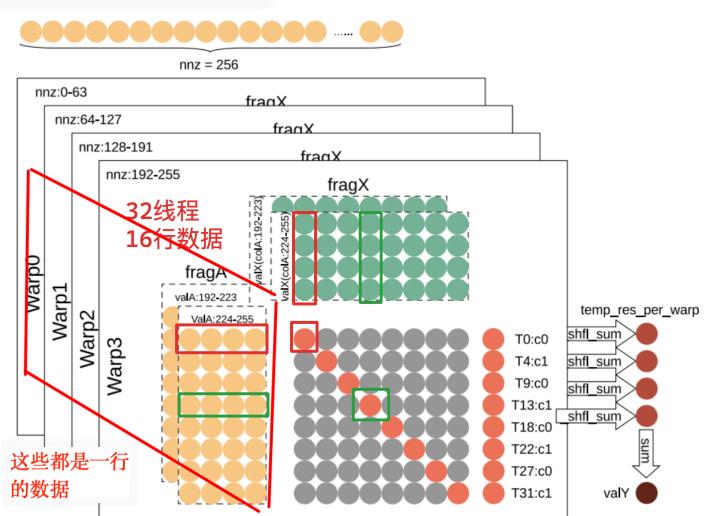
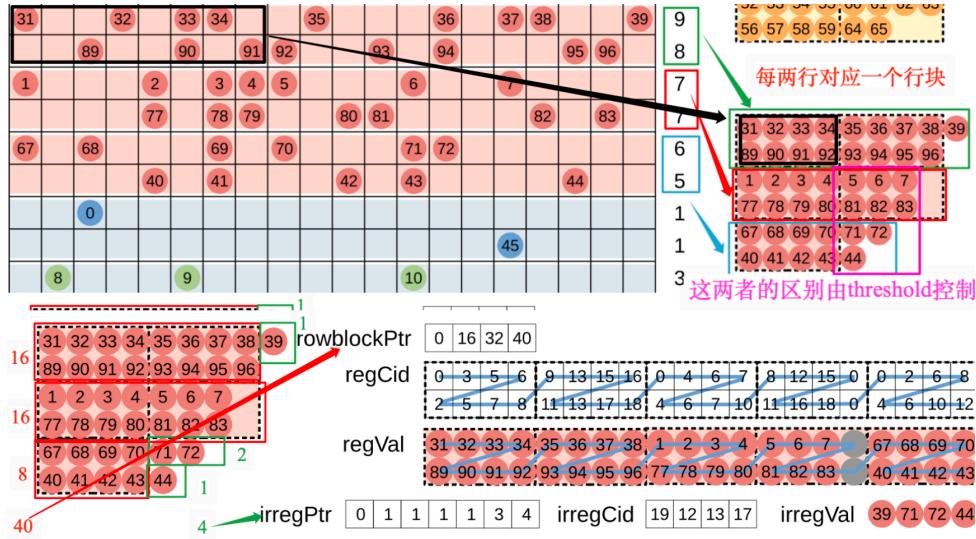


图 6 长行计算方式示意图

这一行中的数，就按照列的顺序来进行排布，每个组的大小为 16，每个组包括两个小的矩阵，矩阵的大小都为 $2 * 4$ 。每行的元素首先会填充第一组的左边的矩阵，随

后会填充右边的矩阵，如果该组的左右两侧的矩阵都被填充完全，再启用新的组，以此类推，如果某一行的所属最后一个组的左侧或右侧矩阵没有被填满，则使用零元素来进行填充。也就是说，一个行非零元素越多，则它可以分成的组数就越多。（每种行的分组策略都是不同的！！）

(c) 中行对应的存储结构和计算方式



(d) Medium Rows

图 7 中行存储格式示意图

```

Algorithm 3 A pseudocode of warp-level Medium-Rows SpMV.
1: for laneid = 0 to 31 in parallel do
2:   for i = 0 to LOOP_NUM do
3:     fragY[2], fragA, fragX, res ← 0
4:     idx = (3 & laneid) + (laneid >> 2) × MMA_K
5:     bid = wid×LOOP_NUM + i
6:     len = rowblockptr[bid + 1] - rowblockptr[bid]
7:     for j = 0 to len step MMA_K do
8:       fragA ← regVal[(fsetA + idx)]
9:       fragX ← valX[regCid[(fsetA + idx)]]
10:      mma_m8n8k4(fragY, fragA, fragX)
11:      idx += MMA_M × MMA_K
12:    end for
13:    target = ((laneid - i × 8) >> 1) × 9
14:    fragY[0] = __SHFL_SYNC(0xffffffff, fragY[0], target)
15:    fragY[1] = __SHFL_SYNC(0xffffffff, fragY[1], target + 4)
16:    if (laneid > 3) == i then
17:      res = (1 & laneid) == 0 ? fragY[0] : fragY[1]
18:    end if
19:  end for
20:  if (laneid >> 3) < LOOP_NUM then
21:    cur_row = wid×LOOP_NUM × MMA_M + laneid
22:    for i = irregPtr[cur_row] to irregPtr[cur_row + 1] do
23:      res += irregVal[i] × valX[irregCid[i]]
24:    end for
25:    valY[cur_row] ← res
26:  end if
27: end for

```

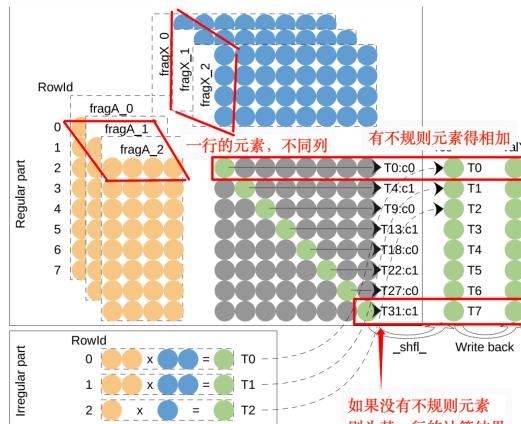


图 8 中行计算方式示意图

当行为中行时，每 MMA_M 行被视为一个行块 (row-block)，在图中为 2，也就是每两行对应一个行块，也可以理解为一个组，这时，每两行只对应一个组，组内的子矩阵块的数量会根据这两行的非零元素的数量而变化，不再仅仅是左右两个，有可能是很多个。如果，非零元素无法正好填充好一个子块，则利用 threshold 来进行控制，判断这个多出来的非零元素是属于新的子矩阵块中的元素，后面用零元素补充，还是认为为不规则元素，进行一个重新处理。

(d) 短行对应的存储结构和计算方式

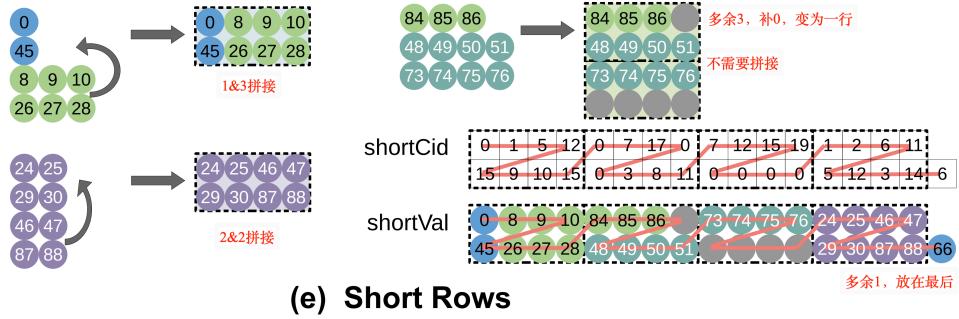


图 9 短行存储格式示意图

短行部分采用 拼接成块的策略来提高 MMA 单元的利用率。

短行部分的数据被分为四类：1&3 拼接行、行长为 4 的行（包括填充零元素的行）、2&2 拼接行和行长为 1 的行。

```
Algorithm 4 A pseudocode of warp-level Short-1&3-Rows SpMV.
1: for laneid = 0 to 31 in parallel do
2:   fragA, fragX, res ← 0
3:   idx = (3 & laneid) + (laneid >> 2) × MMA_K
4:   for i = 0 to 3 do
5:     fragY[2] ← 0
6:     cidA ← shortCid[offsetA + idx]
7:     if 1 & i == 0 then
8:       fragA ← shortVal[offsetA + idx]
9:       fragX = 3 & laneid == 0 ? valX[cidA] : 0
10:    else
11:      fragX = 3 & laneid == 0 ? 0 : valX[cidA]
12:      idx += MMA_M × MMA_K
13:    end if
14:    mma_m8n8k4(fragY, fragA, fragX)
15:    target = ((laneid - i × 8) >> 1) × 9
16:    fragY[0] = __SHFL_SYNC(0xffffffff, fragY[0], target)
17:    fragY[1] = __SHFL_SYNC(0xffffffff, fragY[1], target + 4)
18:    if (laneid >> 3) == i then
19:      res = (1 & laneid) == 0 ? fragY[0] : fragY[1]
20:    end if
21:  end for
22:  valY[offsetY + laneid] ← res
23: end for
```

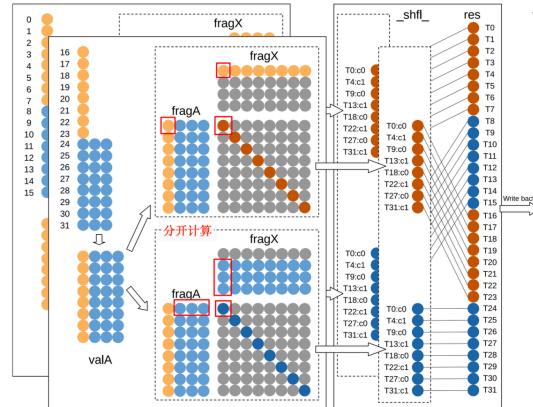


图 10 短行计算方式示意图

Next

- (1) 继续编写 BWM 的代码
- (2) 专利交底书更改
- (3) 继续阅读关于 SpMV 以及 SpMM 相关领域的论文