

Weekly Research Progress Report

Student: 从 兴 Date: 2024/3/13 - 2024/3/31

List of accomplishments this week

- (1) 阅读论文: TileSpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs。
- (2) 阅读论文: Revisiting thread configuration of SpMV kernels on GPU: A machine learning based approach。
- (3) 利用曙光超算, 编写基于结构体存储的 SpMV 示例, 对比与普通 CSR 格式的性能差异。
- (4) 配置本地的 CUDA 环境, 将在曙光超算上的 SpMV 示例移植到 GPU 上, 发现一些 AMD 的 GPU 与 NVIDIA 的 GPU 的差异。
- (5) 整理最近三年来, 发表在 SC、PPoPP、ICS、HPCA 和 ASPLOS 上关于 SpMV 的论文。

Paper summary

Name: TileSpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs。

Motivation: 在 GPU 上加速 SpMV 的研究中, 研究者们通常会使用 CSR 格式, 但是 CSR 格式在 GPU 上的性能并不理想, 因为它的内存访问模式是不规则的。同时, 虽然有很多技术被应用, 比如: 增加并行性、减少负载不平衡和利用机器学习来选择最佳格式。但是, 这些技术并没有很好的解决这些问题。同时, 作者发现, 在基于 GPU 的 SpMV 工作中, 二维空间稀疏结构还没有得到很好的开发。

Solution: 作者首先开发了稀疏矩阵的二维空间结构来对 SpMV 来进行优化, 随后提出了一种基于 Tile 的 SpMV 算法, 该算法可以在 GPU 上高效地执行。这个思想和 TileGEMM 中的思想很一致。同时实现了使用七种不同的稀疏矩阵格式, 包括 CSR、ELL、HYB 等等, 并且设计了一种选择方法, 来找到每个块的最佳格式和 SpMV 实现, 从而提高了整体性能。

Related to us: 作者使用的存储格式的基本单位为 Tile, 它是一个 16*16 的小矩阵, 但我看它的原代码对小矩阵内的结构虽然使用了结构体包装, 但进行 GPU 内存分配时,

还是使用的分开 `malloc` 的方法，因此，有可能使用结构体块的方法对其的空间进一步压缩，同时，提高它的数据局部性，从而加快 `kernel` 的计算速度。

Name: Revisiting thread configuration of SpMV kernels on GPU: A machine learning based approach.

Motivation: 由于内存访问不规则和工作负载不平衡, GPU 上的 SpMV 操作优化一直是个挑战。现有的大多数方案都是基于经验的, 而且很难找到一个通用的方案。

Solution: 作者提出了一种基于机器学习的方法，来预测矩阵近似最佳的线程配置。同时，作者对前人关于机器学习来预测最有配置的方法进行了改进，提出了将矩阵进行分块，然后对每个块进行预测的方法。这种方法可以更好的适应不同的矩阵。最后，作者还深入研究了训练后的模型，大致说了一下稀疏矩阵的特征与最佳线程配置之间的联系。

Related to us: 这篇文章的方法可以用来预测 SpMV 的最佳线程配置，这样可以更好的适应不同的矩阵。和后期的工作结合，可以用来预测 SpMV 的最佳存储格式。

Work summary

1. TileSpMV 论文总结

(1) TileSpMV 的存储格式

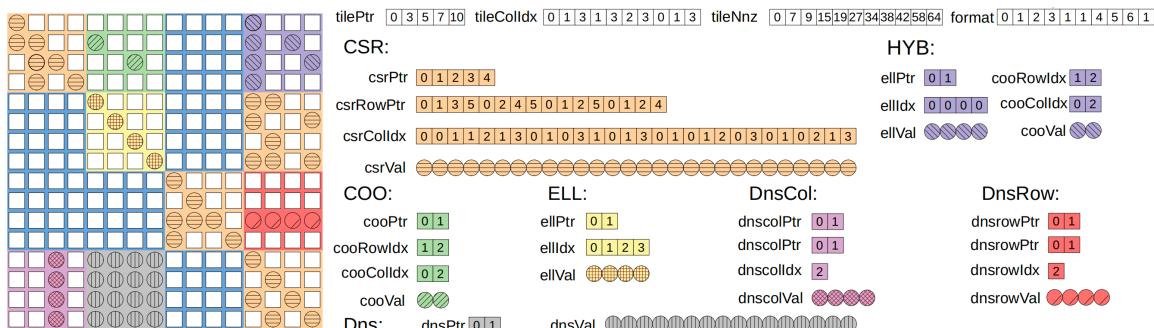


Fig. 3. An example matrix A of size 16-by-16 stored in 10 sparse tiles of size 4-by-4. The tile structure includes three arrays `tilePtr`, `tileColIdx` and `tileNnz` representing the memory offsets of tiles, tile column indices and the offsets for the number of nonzeros in sparse tiles. According to the format selection method, four of the 10 tiles keep the CSR format unchanged and use three arrays to store their information. The remaining six tiles are transformed to different formats, including COO, ELL, HYB, Dns, `DnsRow` and `DnsCol`. Each format has several corresponding arrays to store the nonzeros and their indices.

图 1 TileSpMV 的存储格式

TileSpMV 首先将输入矩阵 A 划分为大小相同 (在本文中为 16×16) 的稀疏块，并使用稀疏块作为基本工作单元。分区之后，将生成两个级别的信息，这些信息表示为一组数组，用于存储稀疏的 tile。信息的两个级别分别存储 矩阵的 tile 结构和 每个稀疏 Tile 的内部信息。

- 第一层：存储矩阵的 Tile 结构

- (1) 大小为 $tilem_A + 1$ 的 $tilePtr$ 数组，其中 $tilem_A$ 是矩阵 A 的 tile 行数，相当于 CSR 存储格式中的 row_ptr 数组。
- (2) 大小为 $numtile_A$ 的 $tileColIdx$ 数组，其中 $numtile_A$ 是矩阵中稀疏 tile 的数量，用于存储每个具有非零元素 tile 的列索引，相当于 CSR 存储格式中的 col_idx 数组。
- (3) 大小为 $numtile_A + 1$ 的 $tileNnz$ 数组，其中存储稀疏 tile 中非零元数量的偏移量。通过该数组，可以计算得到某个非零元 tile 中的非零元素的个数。
- 第二层：存储每个稀疏 Tile 的内部信息
 - 第二层将非零元素及其索引以不同的格式存储在每个稀疏 Tile 中。这里有七种选择：CSR、COO、ELL、HYB、稠密 (Dns)、稠密行 (DnsRow) 和稠密列 (DnsCol)。
 - 作者在这里写了个算法，实现了对每个 Tile 的最佳格式的选择。

(2) TileSpMV 的算法

作者还实现了关于七种不同的稀疏矩阵格式的 SpMV 算法，如下图所示。

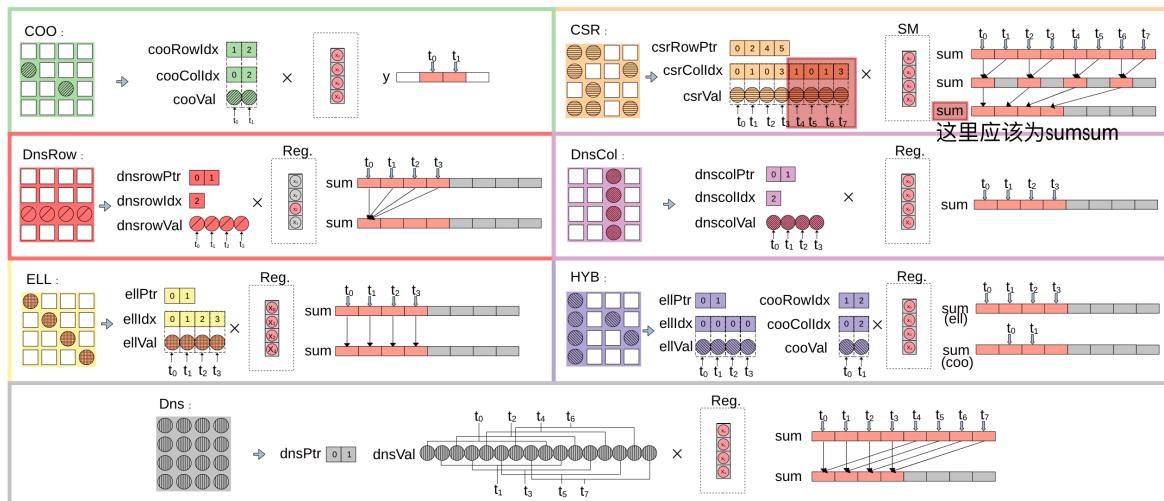


图 2 七种存储格式的 SpMV 算法

作者在文中一一介绍了这七种存储格式的 SpMV 算法，这里就只介绍一下 CSR 格式的 SpMV 算法。

注：文章中，作者好像有一些错误，通过查看源代码，发现了一些错误，这里进行了修正。

在 warp 级 CSR-SpMV 算法中，32 个线程的 warp 始终处理一个具有 16 行的 tile，这意味着每两个连续的线程处理一行。在计算之前，将向量 x 中的 16 个条目对应段加

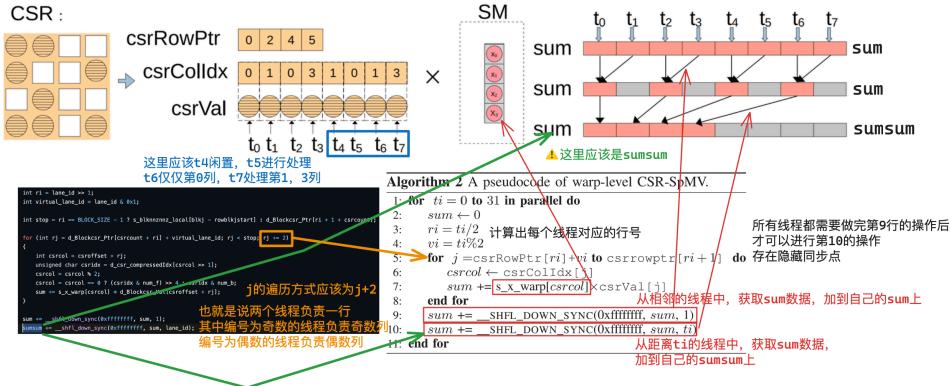


图 3 Tile 中使用 CSR 格式进行存储的 SpMV 算法

载到芯片上共享内存中以获得更好和可控的数据局部性。在线程计算后，部分 y 被相加在一起。如上图的示例，假设有 8 个线程 (t_0 - t_7) 来处理 CSR 格式中的 4×4 tile，并且每两个连续线程处理一行。值得注意的是第三行只有一个元素，因此 t_4 可以单独计算它，而 t_5 不执行任何操作（应该是 t_5 单独计算，投 t_4 不执行任何操作）。相反，第四行有三个元素，所以 t_6 需要处理两个元素（应该是 t_7 需要处理两个元素，分别是第 3 行的第 1 列和第 3 列的元素（计数从 0 开始计数））。然后使用 sum 来存储每个线程的计算结果，并且 $shuffle$ 将被用于两次添加相邻线程的结果，并传输以适应 sum （应该是 $sumsum$ ）。

(3) Tile 中存储格式的选择方法

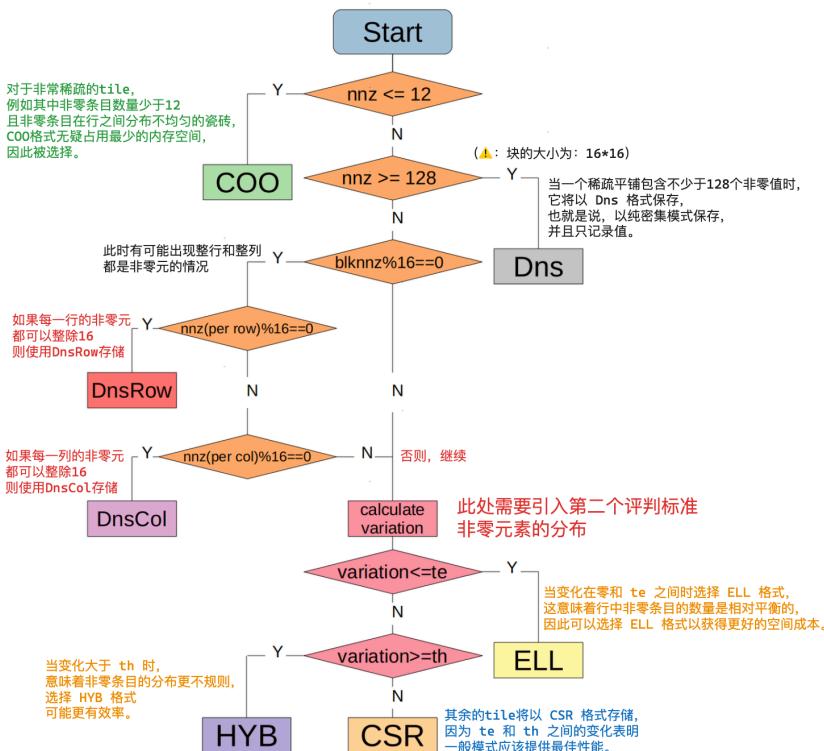


图 4 Tile 中存储格式的选择方法

在本文的方法中，通过实验将 te 和 th 分别设置为 0.2 和 1.0，因为在实验过程中发

现这两个阈值通常能给出最佳性能。除了 tile 格式的 SpMV 实现外，改善负载平衡也应该被考虑进来。

尽管使用固定大小的稀疏瓦片作为基本工作单元已经可以在一定程度上自然地避免负载不平衡，但作者还需要将非常长的瓦片行分割成小块以获得更加均匀的工作量。在文章的实现中，作者添加了一个名为 `tbalance` 的参数，并让一个 warp 处理不超过 `tbalance` 个 Tile。如果一个 Tile 行中稀疏 Tile 数量大于 `tbalance`，则会划分该 tile 行并使用多个 warps 一起处理它。

最后，由 warps 生成的部分 y 属于同一条 tile row 的数据会通过原子加法相加。通过这种方式，可以确保每个 warp 具有类似任务以改善负载平衡。

注：这是实现的负载均衡，只是简单的进行了实现。

2. 分块机器学习

(1) 基于机器学习技术的线程配置预测

矩阵的特征与最优线程配置之间的关系并不直接，难以用公式建立起关系。因此，作者提出了一种基于机器学习的方法，来预测矩阵的最佳线程配置。

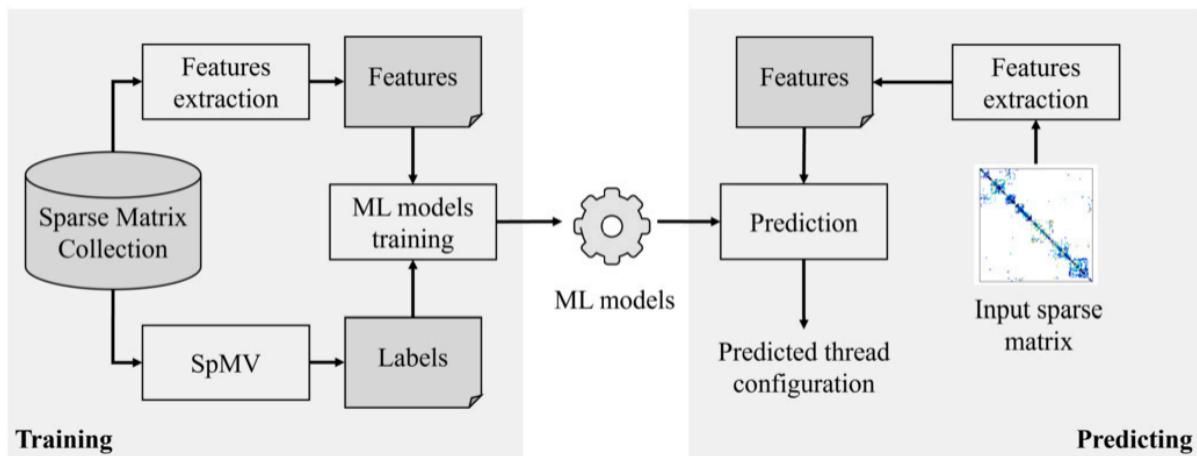


图 5 使用 AutoML 框架训练和预测最佳线程配置

作者首先选择稀疏矩阵的一些特征，然后用最优 TpR 标记每个矩阵。然后，将这些标记矩阵输入不同的 ML 模型进行训练。接下来，选择在测试集中达到最高精度的模型。最后，给定一个输入稀疏矩阵，训练后的 ML 模型根据其特征预测最优 TpR 设置。

作者在本文中，使用的是一个开源的模型框架—AutoGluon，它是一个自动机器学习框架，可以自动选择最佳的模型和超参数。作者选择了一些特征，如上图所示，然后将这些特征输入到 AutoGluon 中，训练出一个模型，用于预测最佳线程配置。AutoGluon 可以适用于各种机器学习任务，如分类、回归和时间序列预测，可以有效提高模型开发的效率和资源利用率。

Table 2
The matrix features used for model training.

Feature	Description	Complexity
m	The number of rows	$O(1)$
n	The number of columns	$O(1)$
nnz	The number of non-zero elements	$O(1)$
d	The density of non-zero elements	$O(1)$
min	The minimum number of non-zero elements per row	$O(n)$
max	The maximum number of non-zero elements per row	$O(n)$
$mean$	The average of non-zero elements per row	$O(1)$
var	The variance on non-zero elements per row	$O(2n)$
max_mu	The difference between max and $mean$	$O(1)$
$SqMean$	The square root of $mean$	$O(1)$
cov	The covariance on non-zero elements per row	$O(1)$

图 6 用于模型训练的矩阵特征

(2) 对划分矩阵块后的线程配置

作者在文章中提出了根据不同行中非零元素的分布情况对输入稀疏矩阵进行分块，然后预测不同块的最佳线程配置的方法。也就是使用上面提到的机器学习模型，对每个块进行预测，然后选择最佳的线程配置。

Algorithm 1: Partition algorithm.

```

Input:  $M$ ,  $nnz$ ,  $csrRow$ 
Output:  $nBlk$ ,  $rowBlk$ ,  $nnzBlk$ ,  $isLong$ .
1  $nBlk \leftarrow 0$ ;  $sumNnz \leftarrow 0$ ;  $rowBlk[0] \leftarrow 0$ ;
2 if  $csrRow[1] - csrRow[0] \geq T\_LONG$  then
3   |  $lastIsLong \leftarrow 1$ ;
4 else
5   |  $lastIsLong \leftarrow 0$ ;
6 end
7 for  $r \leftarrow 0$  to  $M$  do
8   |  $curNnz \leftarrow csrRow[r + 1] - csrRow[r]$ ;
9   |  $sumNnz \leftarrow sumNnz + curNnz$ ;
10  | if  $[curNnz \geq T\_LONG] \& \& [!lastIsLong]$  then
11    |   |  $rowBlk[nBlk + 1] \leftarrow r$ ;
12    |   |  $nnzBlk[nBlk] \leftarrow sumNnz - curNnz$ ;
13    |   |  $sumNnz \leftarrow curNnz$ ;
14    |   |  $nBlk \leftarrow nBlk + 1$ ;  $lastIsLong \leftarrow 1$ ;
15  end
16  | if  $[curNnz < T\_LONG] \& \& [lastIsLong]$  then
17    |   |  $rowBlk[nBlk + 1] \leftarrow r$ ;
18    |   |  $nnzBlk[nBlk] \leftarrow sumNnz - curNnz$ ;
19    |   |  $isLong \leftarrow isLong + pow(2, nBlk)$ ;
20    |   |  $sumNnz \leftarrow curNnz$ ;
21    |   |  $nBlk \leftarrow nBlk + 1$ ;  $lastIsLong \leftarrow 0$ ;
22  end
23 end
24  $rowBlk[nBlk + 1] \leftarrow M$ ;  $nnzBlk[nBlk] \leftarrow sumNnz$ ;
25 if  $lastIsLong$  then
26   |  $isLong \leftarrow isLong + pow(2, nBlk)$ ;
27 end
28  $nBlk \leftarrow nBlk + 1$ ;

```

图 7 关于矩阵块的划分方法

由于对所有块使用 CSR 格式，因此上文提到的分区是虚拟进行的，只需要扫描一次 CSR 的行指针数组。因此，在设计中没有格式转换和数据重新组织。详细的分区算法在 Algorithm 1 中呈现。使用标量 $nBlk$ 来表示已分区块数。 $rowBlk$ 和 $nnzBlk$ 是两个数组，分别存储每个块的行索引和非零元素数量。使用位向量 (bool 类型，1 比特的

数) *isLong* 来区分长行和短行块。

考虑图 7 中展示的矩阵 shermanACa 作为例子，它被划分为三个块。*isLong* = (2)₁₀ = (010)₂ 表示第一个和第三个块是两个短行块，而第二个是一个长行块。如果当前行很长，并且上一行很短，则是时候开始一个新的块 (lines 10-15)，反之亦然 (lines 16-22)。通过这种方式，具有相似非零元素分布的连续行被打包到一个块中。

作者这边还有一个思想值得借鉴，就是：数据集的获取，作者的数据集是从 **SuiteSparse** 中获取的，随后，为了扩充数据集，作者又对 **SuiteSparse** 中的数据集进行了切片处理，这样可以得到更多的数据集，从而提高模型的泛化能力。

(3) SpMV kernel 的优化

作者设计了一种基于 CSR-vector 和 CSR-adaptive 的新 SpMV 内核，以提高性能。在分区后，每个块中非零元素的分布是规则的。因此，作者的主要想法是：

- 使用 CSR-vector 计算短行块，为每行分配固定数量的线程，并将中间结果缓存到寄存器中。
- 使用 CSR-adaptive 计算长行块，为每行分配一个线程块，并将中间结果缓存在共享内存中。

```
1  __global__ void spmv_kernel(nBlk, isLong, rowBlk[], TpRblk[],  
2                               nThdBlk[], ...)  
3  {  
4      __shared__ double sdata[T_LONG];  
5      extern __shared__ double ptrs[] [2];  
6      for (i = 0; i < nBlk; i++) {  
7          thdBlkPtr_0 = nThdBlk[i];  
8          thdBlkPtr_1 = nThdBlk[i+1];  
9          nThdBlks = thdBlkPtr_1 - thdBlkPtr_0;  
10         rStart = rowBlk[i];  
11         rStop = rowBlk[i+1];  
12         TpR = TpRblk[i],  
13             if (blockIdx.x >= thdBlkPtr_0 && blockIdx.x < thdBlkPtr_1  
14                 ) {  
15                 if ((isLong && 1) == 0) { /* short-row block */  
16                     thdId = blockDim.x * nThdBlks + threadIdx.x;  
17                     thdLane = threadIdx.x & (TpR - 1);  
18                     vecId = thdId / TpR;  
19                     vecLane = threadIdx.x / TpR; 使用csr-vector, 为每行分配固定的线程数  
20                     nVec = (blockDim.x / TpR) * nThdBlks;  
21                     for (r = rStart + vecLane; r < rStop; r += nVec)  
22                         // ...  
23                 } else { /* long-row block */  
24                     vecLane = (blockIdx.x - thdBlkPtr_0);  
25                     for (r = rStart + vecLane; r < rStop; r += nThdBlks)  
26                         // ... 使用csr-adaptive, 为每行分配线程块  
27                 }  
28             }  
29         isLong = isLong >> 1;  
30     }  
31 }
```

图 8 新的 SpMV 内核代码

(4) 最优的模型

作者发现，优秀的模型都是基于决策树构建的集成模型。然后研究了稀疏矩阵的特征与最佳线程配置之间的关系。可视化决策树如下图所示：

根据这棵树，稀疏矩阵最重要的特性是从根部到叶子节点依次为：最大值、密度（非零元素）、方差（每行非零元素）、 m 、 n 、 nnz 和平均值。

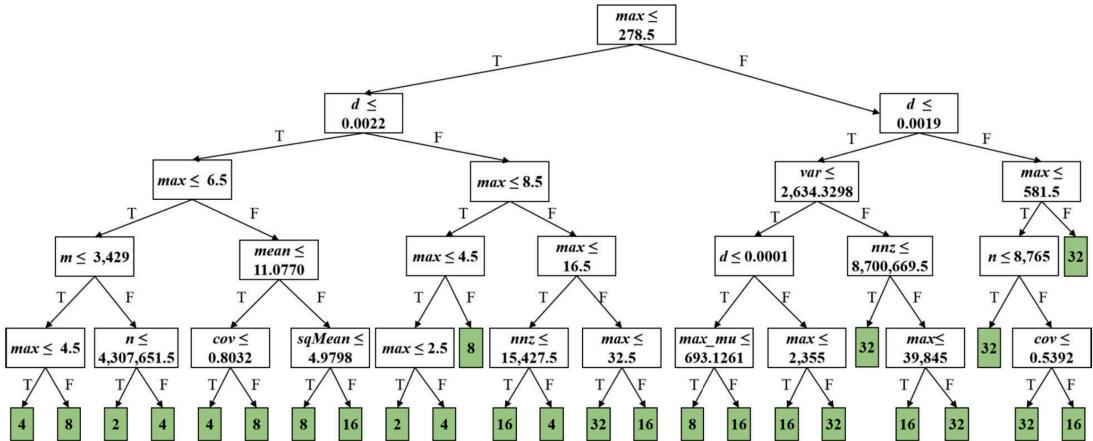


Fig. 13. Visualized 5-level decision tree. The nodes filled with green color represent the thread configurations decided by the tree.

图 9 可可视化的 5 层决策树，用绿色填充的节点表示由决策树决定的线程配置

- 最大值 (max)：每行非零元素的最大数目。
- 密度 (d)：非零元素的密度。
- 方差 (var)：每行非零元素的方差。
- 行数 (m)：矩阵的行数。
- 列数 (n)：矩阵的列数。
- 非零元的数目 (nnz)：矩阵中非零元的总数。
- 平均值 ($mean$)：每行非零元素的平均值。

3. 结构体-内存块-优化思路

(1) 目前存在的问题

CSR 格式主要包含三个数组，分别是 `row_ptr`、`col_idx`、`val`，这三个数组的类型分别是 `int`、`int`、`double`。

目前现存的研究，都是对这三个数组分别进行 `malloc`，这可能会导致三个数组所在的空间并不是连续的，因此，使用这种方法，把数据在 CPU 向 GPU 进行拷贝时，可能会导致拷贝多个分块，造成多余的数据浪费。同时，使用该方法在 kernel 进行计算时，会导致数据局部性不佳，因为，虽然是同时对这三个数组进行访问，但是由于三个数组

所在的区块并不是同一个区块，会导致在计算时，计算单元需要访问多个不同的区块，导致相应的计算延迟。

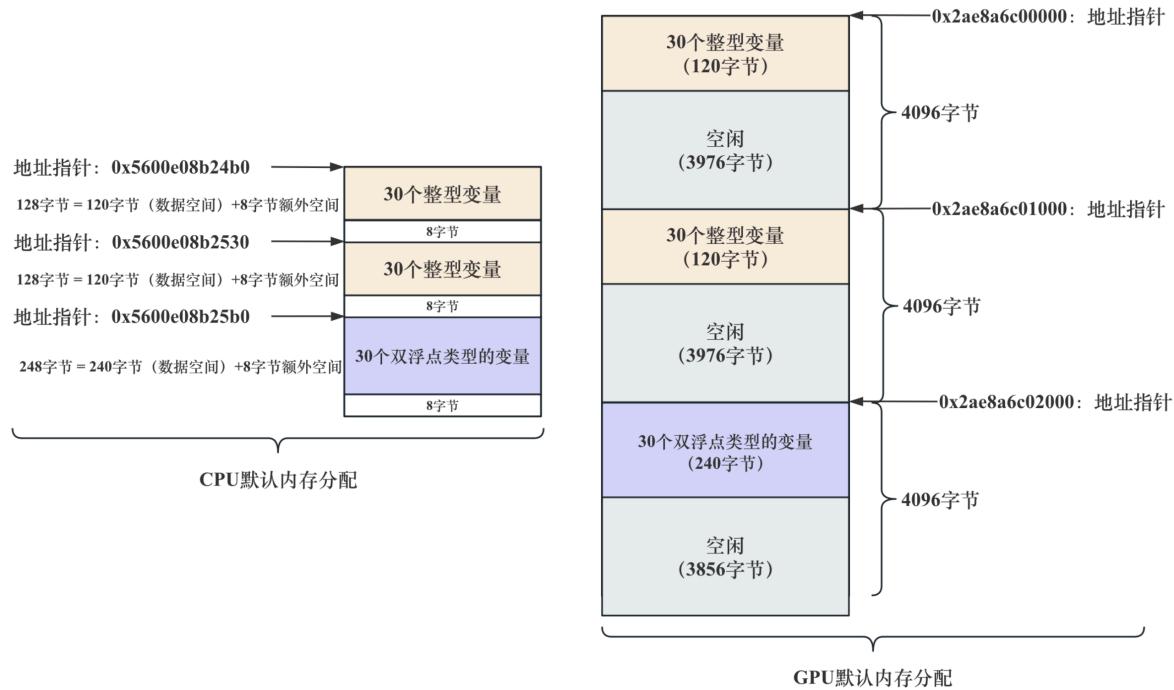


图 10 CPU 内存分配与 GPU 内存分配的比较

注意：此处的 CPU 进行内存分配时，会对分配的内存多 8 个字节的空间，原因是：编译器和运行时环境会为了管理这些动态分配的内存而额外分配一些空间。这些额外的空间通常用于存储与动态分配的内存块相关的元数据，例如数组的大小（即元素的数量），这对于后续的 `delete[]` 操作是必要的，以确保能够正确地释放整个数组。因此，CPU 的内存分配实际上是连续的。

(2) AMD GPU 与 NVIDIA GPU 的差异

hipMalloc(&int_test1, 1000*sizeof(int));	// 1000*4 = 4000 应该分一个块
hipMalloc(&int_test2, 2000*sizeof(int));	// 2000*4 = 8000 应该分两个块
hipMalloc(&int_test3, 10*sizeof(int));	// 10*4 = 40 应该分一个块
hipMalloc(&char_test1, 2000*sizeof(char));	// 2000*1 = 2000 应该分一个块
hipMalloc(&char_test2, 5000*sizeof(char));	// 5000*1 = 5000 应该分两个块
hipMalloc(&char_test3, 10*sizeof(char));	// 10*1 = 10 应该分一个块
hipMalloc(&double_test1, 500*sizeof(double));	// 500*8 = 4000 应该分一个块
hipMalloc(&double_test2, 1000*sizeof(double));	// 1000*8 = 8000 应该分两个块
hipMalloc(&double_test3, 10*sizeof(double));	// 10*8 = 80 应该分一个块

int_test1:	0x2b7ce9a06000
int_test2:	0x2b7ce9a07000
int_test3:	0x2b7ce9a09000
char_test1:	0x2b7ce9a0a000
char_test2:	0x2b7ce9a0b000
char_test3:	0x2b7ce9a0d000
double_test1:	0x2b7ce9a0e000
double_test2:	0x2b7ce9a0f000
double_test3:	0x2b7ce9a11000

图 11 AMD GPU 的内存分配

从上面的结果可以看出，在 N 卡中，内存的分配方式也是一样，也是按照块进行分配的。

其中，不同的是，AMD 显卡中内存分配的基本单位是 $4096B = 4KB$ ，但是在 Nvidia 显卡中，内存分配的基本单位是 $512B$ 。这就导致了在 AMD 显卡中，如果分配的内存

```

C:\Users\cx175\Desktop\base_struct\Test>nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Sep_21_10:41:10_Pacific_Daylight_Time_2022
Cuda compilation tools, release 11.8, V11.8.89
Build cuda_11.8.r11.8/compiler.31833905_0

C:\Users\cx175\Desktop\base_struct\Test>test1
int_test1: 0000000702600000
int_test2: 0000000702601000
int_test3: 0000000702603000
char_test1: 0000000702603200
char_test2: 0000000702603A00
char_test3: 0000000702604E00
double_test1: 0000000702605000
double_test2: 0000000702606000
double_test3: 0000000702608000

```

图 12 Nvidia GPU 的内存分配

小于 4KB，那么会分配 4KB 的内存，而在 Nvidia 显卡中，如果分配的内存小于 512B，那么会分配 512B 的内存。

(3) 优化思路

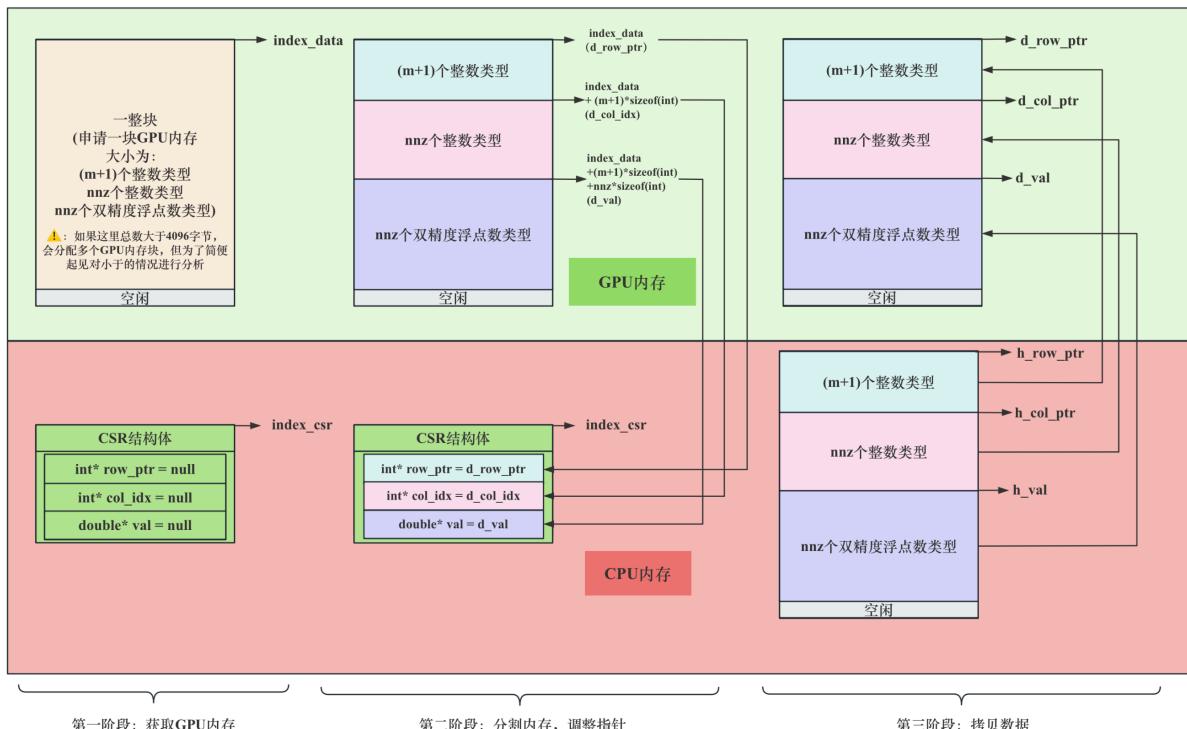


图 13 优化示意图

基于上述困境，提出一种新的存储结构，该结构把这三个数组统一存储在一个结构体中，每个结构体中仅仅存储这三个数组的指针，同时，在进行主机到 GPU 的拷贝时，摒弃以往的三个数组分别进行 `malloc` 的方法，而是根据输入的矩阵信息，向 GPU 动态

地申请一个足矣容纳这三个数组的内存块，随后，利用动态分配算法，将这块大的内存块，根据输入的矩阵信息，动态的分为三个块，分别存储上述提到的三个数据信息。这样，可以保证：第一，这个结构体所在的空间是连续的，第二，该结构体内部数组指针指向的地址空间也是连续的。因此，如果矩阵较小的话，就不需要进行多次块的传输，以及多次块的访问，就可以实现 CPU 数据到 GPU 的传输以及在 GPU 上 kernel 的计算。

(4) 实验测试

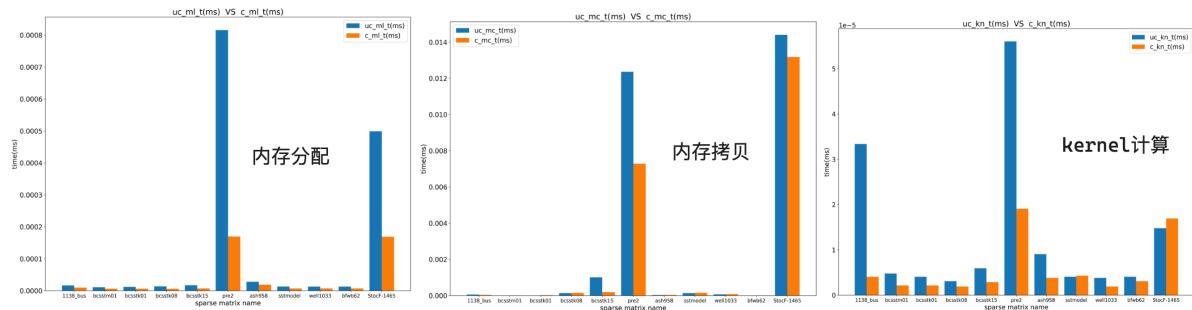


图 14 AMD GPU 上的实验结果

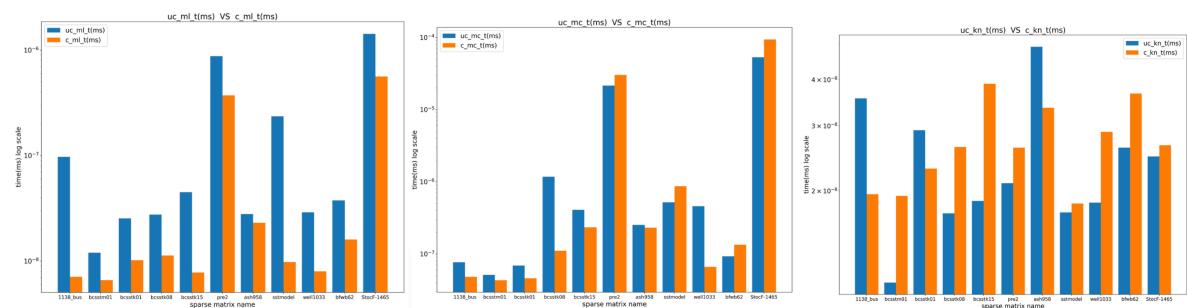


图 15 Nvidia GPU 上的实验结果

注：Nvidia GPU 实验结果上的 y 坐标使用了 log10 的刻度。

4. 近年来论文情况

分析近年来关于 SpMV、SpMM 的论文，发现并没有使用这一点优化措施。

Next

- (1) 继续阅读关于 SpMV、SpMM 的论文，寻找更多的优化方法。
- (2) 对 TileSpMV 进行修改，将其改为使用结构体存储的方法。