Weekly Research Progress Report

Student: 从 兴 **Date:** 2024/4/1 - 2024/4/15

List of accomplishments this week

（1）准备组会分享论文的 PPT。

（2）阅读论文：《Overhead-Conscious Format Selection for SpMV-Based Applications》

（3）研究 HYCOM 相关的编译方法，HYCOM 的执行方法，HYCOM 的作业提交方法。

（4）研究 TileSpMV 的源代码，目前基本结构已经了解，下一步准备进行代码的改写。

（5）编写自适应感知内存分配颗粒的 Demo，基本实现了自适应感知内存分配颗粒的功能。

Paper summary

Name：Overhead-Conscious Format Selection for SpMV-Based Applications

Motivation：在为稀疏矩阵选择最佳的存储格式时，以往的研究通常只关注计算性能，而忽略了存储格式选择的开销。作者提出，这可能导致在实际的应用中，尽管预测精度较高，但是总体性能并没能得到显著提升，甚至有时会减慢执行速度。因此，作者提出了一种新的方法，该方法考虑了存储格式选择的开销，以提高性能。

Solution：以前的方法忽视了运行时开销和格式转换的成本，尽管这些开销部分可能会抵消由于选择了更好的格式而带来的性能提升。因此，该文提出来一个考虑这些开销的两层预测模型（时间序列预测模型 + 回归分析模型），分别预测 SpMV 的执行次数、转换耗时、计算时间等，旨在全面评估格式转换的真实收益，通过更精确的开销和益处评估来指导格式的选择，以确保在多次使用中，格式转换后的性能提升能够超过相关开销，从而实际提高应用程序的整体性能。

Related to us：在本篇论文中，我认为有两点可以用到以后的论文中，第一，作者使用了时间序列分析模型来预测 SpMV 的执行次数，**时间序列分析模型的优点是可以很好的动态预测未来的数据**，我想这个可能会对后面的实时分析系统的运行状态来动态选择任务分配时的任务数量会有帮助。第二，作者使用了 XGBoost 决策树集成方法

来预测最佳的存储格式，并且 **XGBoost 可以从训练好的预测模型中自动确定特征的重要性**，因此，对于我们后面的块内存储格式自动选择，可能会有一定的借鉴意义。

Work summary

1. 论文阅读笔记

(1) 论文提出的背景

在为稀疏矩阵选择最佳的存储格式时，以往的研究通常只关注计算性能，而忽略了存储格式选择的开销。就如下面流程图 1 中所展示的这样：

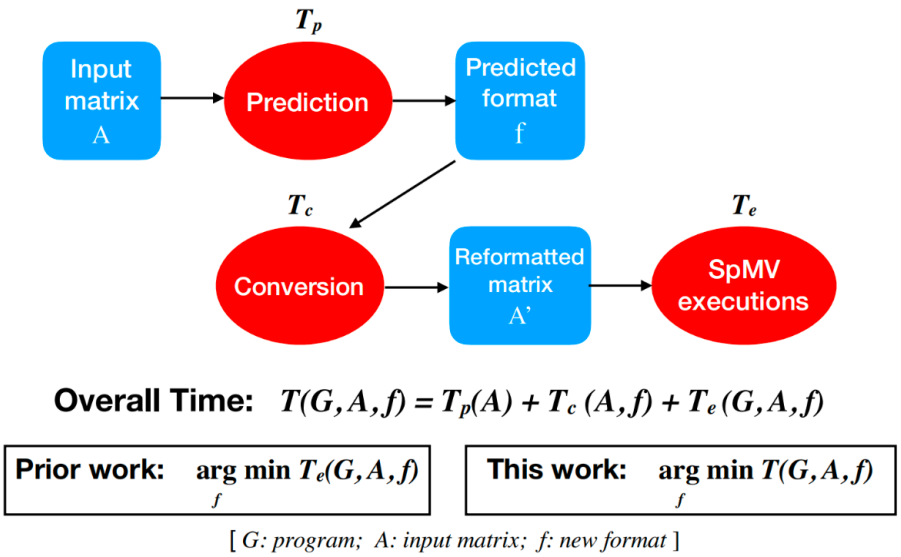

图 1 一个受益于改进矩阵格式应用程序的典型工作流程

可以看到，总耗费的时间包括以下三个部分：

$$\texttt{Overall Time:}\quad T(G,A,f) = T_p(A) + T_c(A,f) + T_e(G,A,f)$$

但是，以往的研究只关注了 $T_e(G,A,f)$，而忽略了 $T_p(A)$ 和 $T_c(A,f)$，这可能导致在实际的应用中，尽管预测精度较高，但是总体性能并没能得到显著提升，甚至有时会减慢执行速度。情况如下图 2 所示，可以看到，有将近 63% 的矩阵在选择了最佳的存储格式的背景下，将其他开销加入后，性能提升并不明显，甚至会变慢。

(2) 论文的创新点

(a) 采用显式的开销模型

显式的开销模型：为每种格式转换的开销建立一个模型，以便在选择格式时考虑这

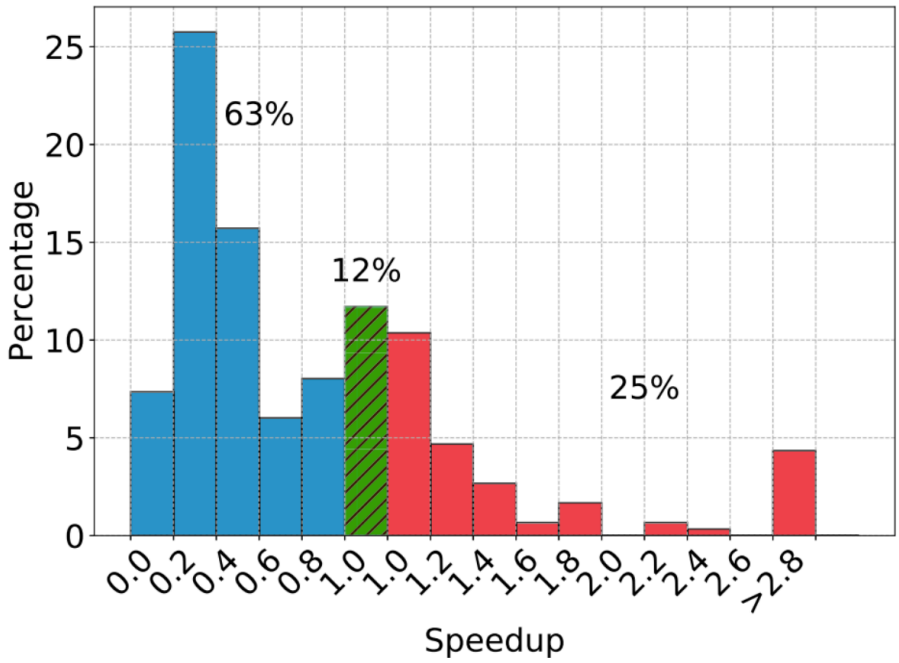

图 2 程序 PageRank 使用以前的基于决策树的预测器时的整体加速度的直方图

些开销。将格式选择可以产生的性能提升与格式转换的开销进行比较，从而最大化性能提升。

隐式的开销模型：构建一个单一的预测器，该预测器只考虑输入矩阵和程序特征，直接进行预测出最佳的存储格式。

$$T_{overall} = T_{predict} + T_{convert} + (\sum_{i} T_{spmv(i)} * N_i) + T_{other}$$

正如上面公式所展示的，论文中将开销进行分解，也就是采用了显式的开销模型，然后，**构建单独的预测器**，直接预测开销、单个 SpMV 时间 $T_{spmv(i)}$ 和 N_i。从而，它们可以跨程序、矩阵和格式工作，具有很好的通用性。同时，它们避免了许多隐式设计所面临的复杂性。

(b) 采用 XGBoost 决策树集成方法

以前的格式选择不考虑开销项，只需要进行定性预测，也就是说，哪种格式给出最短的 SpMV 时间，就选哪个格式作为存储格式，因此，**它们都将问题形式化为一个分类问题**。而在本文中，由于需要考虑开销，并需要与收益做比较，则 **需要预测器来提供定量预测**。不能再把它作为一个分类问题来建模了，故需要建立**回归模型**来预测数值。

本文采用了 **XGBoost 决策树**集成方法，它是一种基于树的模型，它们在简单数据准备、对非线性关系的稳健性能以及易于解释结果方面具有优势。它们产生的回归模型也运行速度快，因为只涉及少量关于数据特征和几个线性代数操作的问题。此外，当与提升方法结合时，在各种问题中显示出最佳预测结果和稳健性。

(c) 两层预测模型

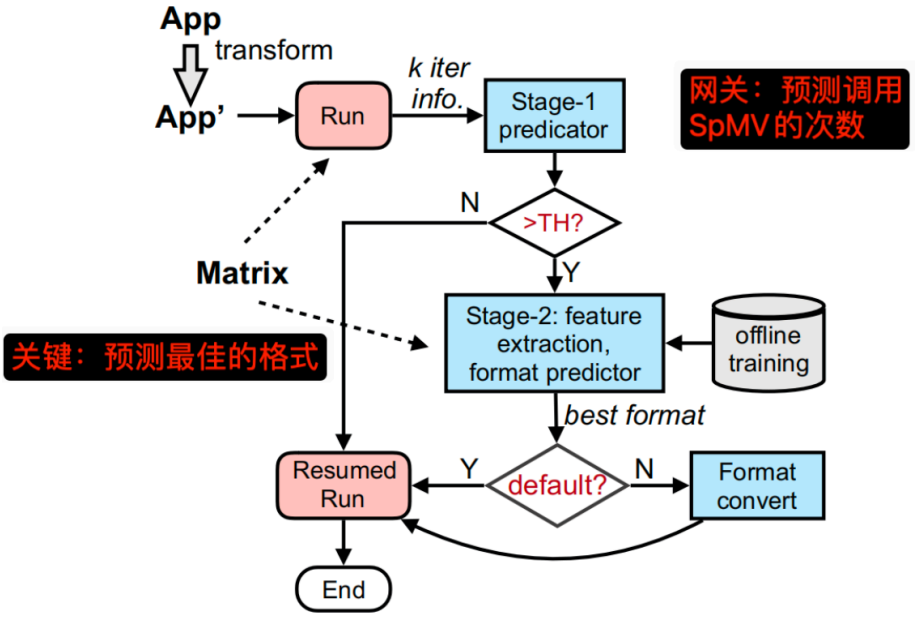

Fig. 4: The two-stage prediction system.

图 3 两阶段预测模型

该方案由两个预测阶段组成：

- 第一个是矩阵上调用 SpMV 次数的轻量级预测器。
  - 第一个预测器充当网关；这个预测器的模型是一个简单的时间序列预测模型，它可以在几乎没有开销的情况下提供合理的预测。
  - 通过比较预测值 LC 与阈值 TH，它决定是否值得调用第二个预测阶段
  - 如果 LC < TH，则不会进行进一步的预测，程序将以默认格式运行；
- 第二阶段进行更复杂的预测，并决定使用哪种格式最佳。
  - 这个预测器是回归分析模型，基于 XGBoost 决策树集成方法。
  - 根据预测结果，矩阵可能保持原样或转换为新格式并在程序执行的其余部分中使用。

两层方案可以确保第一阶段几乎不需要花费时间，但仍能提供合理的预测精度，同时尽可能控制其预测误差的影响。

关于第一层预测器的说明：首先，提到的应用环境是围绕着 SpMV 的应用，这种**应用通常包含一个用于收敛的循环结构**。在这样的循环中，例如在一个线性求解器中，每次迭代都会计算当前解的误差，并且当误差小于预定义的阈值时循环终止。这里的“误差”被称为循环的进度指示器。在不同的应用中，进度指示器的形式可能不同。基于这样的观察，他们建立了一个预测器，**这个预测器可以根据循环的前 k 次迭代中进度**

指示器的序列来预测整个循环所需的迭代总次数。它只在目标程序的循环执行超过前 k 次迭代之后才被调用，也就说，要等到有足够的数据（即前 k 次迭代的数据）后，才开始预测未来的行为。这样可以在不影响程序初始执行效率的情况下，为后续的执行提供决策支持，比如调整资源分配或预测执行时间等。这种方法非常适合于那些迭代次数可能不确定或者对性能优化有较高需求的场景。（PS：**所以，有可能把这个思想用到之前的构想中，通过动态获取系统信息，然后去进行任务的调度和分配**。）

关于第二个阶段的说明：该阶段分别预测每种矩阵格式上的 **转换时间**和 **SpMV 时间**。对于给定的旧矩阵格式和新格式，这两个时间主要由 **矩阵特征**决定。其中，这个训练过程，和之前做的有些类似，也是先利用已经存在的矩阵，进行格式转换时间的计算，然后，利用这些矩阵的特征作为 X 向量和计算所得的时间作为 y 向量，从而得到训练的样本。值得注意的是，论文使用了 **XGBoost 决策树集成方法**来进行预测，它们可以从训练好的预测模型中自动确定特征的重要性，可以 **在不牺牲预测性能的情况下，可以自动删除低重要性得分的特征，直到保留最小特征集为止**。（PS：这对于我们后面的块内存储格式自动选择，可能会有一定的借鉴意义。）

subsubsection*(3) 论文的实验结果

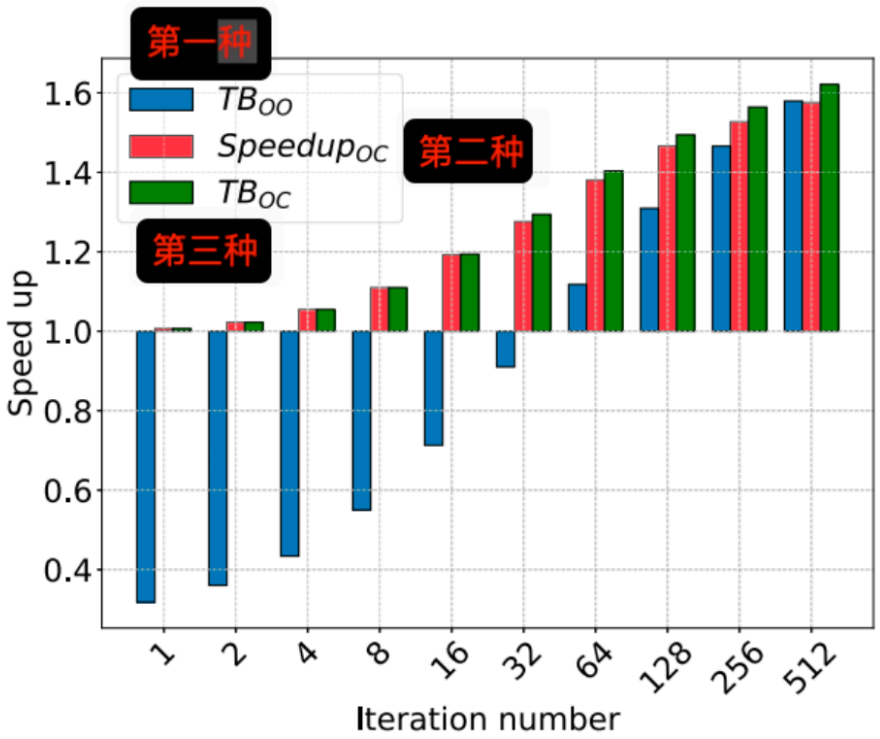

图 4 不同迭代次数，不同选择策略下的性能对比

- 第一种是：没有感知开销，只使用忽略开销方法进行预测最佳存储格式所能取得的加速比上限。

- 第二种是：采用本文的方法–感知开销预测最佳存储格式（将转换、计算、预测时

间最小化），所能取得的加速比。

- 第三种是：取得最佳预测结果，也就是假设都预测正确，所能取得到加速比上限。

如图 4 中的结果所示，由于转换开销的影响，当 SpMV 循环有少量迭代时，来自 TBOO 的决策会导致很大的减速。当迭代次数变大时，该方法的加速比考虑开销方法的加速要低。SpeedupOC 条和 TBOC 条之间的差异表明，在所有不同数量的循环迭代中，**由于主预测器的预测错误导致的加速比损失很小**。

TABLE VI: Speedup of applications.

Application	Speedup		
	TB_{OO}	TB_{OC}	$Speedup_{OC}$
PageRank	1.0762	1.4368	1.4307
BiCGSTAB	1.2454	1.3975	1.3375
CG	0.8246	1.1449	1.1416
GMRES	1.0136	1.2505	1.2034

图 5 基于 SpMV 应用在不同策略下的加速比

考虑开销的方法明显优于不考虑方法的上限。平均加速是显著的，从 1.14 X 到 1.43 X 不等。

2. 关于 HYCOM

(1) Make.csh 更改

图 6 Make.csh 更改

在 Make.csh 文件中，还设置了许多环境变量，都和海洋模型的参数相关，暂时不知道具体的用处。

(2) 配置文件设置

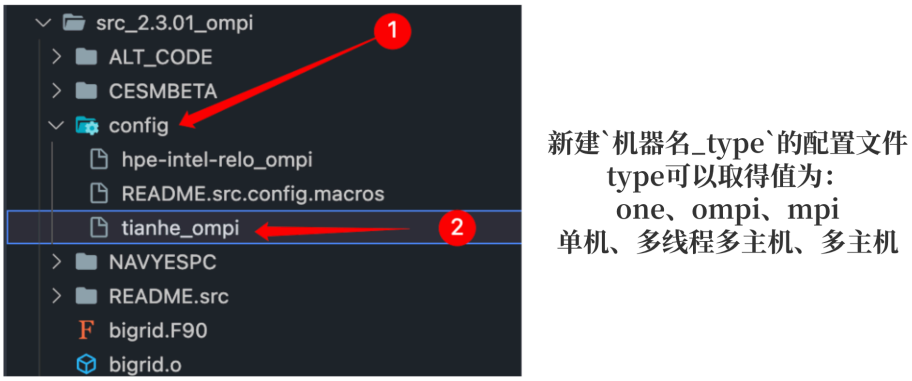

图 7 配置文件的设置（1）

图 8 配置文件的设置（2）

(3) 执行脚本

`sh Make.csh` 执行命令即可生成 hycom 的可执行文件。如果是配置了 mpi，则运行需要使用 `mpirun -n num ./hycom` 。

3. 关于 TileSpMV

(1) 了解大致框架

(2) 一些问题

这样可能导致 block 会比较大，但是，这样做，局部性会高吗?

对 csr 格式存储的一个整体内部的是分别 malloc，但是，这样块就比较大了!

图 9 TileSpMV 代码库结构

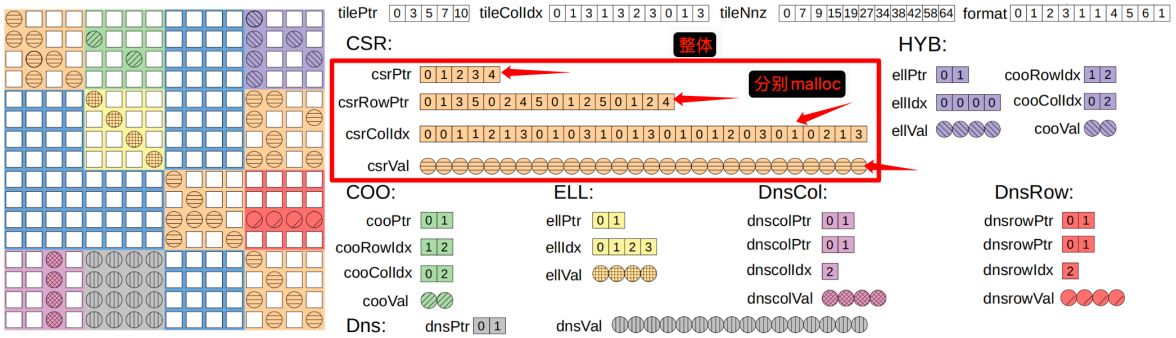

图 10 一个格式的看作一个整体

```
// CSR
unsigned char *d_csr_compressedIdx = (unsigned char *)malloc((csr_csize) * sizeof(unsigned char));
MAT_VAL_TYPE *d_Blockcsr_Val;
unsigned char *d_Blockcsr_Ptr;

cudaMalloc((void **)&d_csr_compressedIdx, (csr_csize) * sizeof(unsigned char));
cudaMalloc((void **)&d_Blockcsr_Val, (csrsize) * sizeof(MAT_VAL_TYPE));
cudaMalloc((void **)&d_Blockcsr_Ptr, (csrptrlen) * sizeof(unsigned char));

cudaMemcpy(d_csr_compressedIdx, csr_compressedIdx, (csr_csize) * sizeof(unsigned char), cudaMemcpyHostToDevice);
cudaMemcpy(d_Blockcsr_Val, Blockcsr_Val, (csrsize) * sizeof(MAT_VAL_TYPE), cudaMemcpyHostToDevice);
cudaMemcpy(d_Blockcsr_Ptr, Blockcsr_Ptr, (csrptrlen) * sizeof(unsigned char), cudaMemcpyHostToDevice);

// COO
unsigned char *d_coo_compressed_Idx;
MAT_VAL_TYPE *d_Blockcoo_Val;

cudaMalloc((void **)&d_coo_compressed_Idx, (coosize) * sizeof(unsigned char));
cudaMalloc((void **)&d_Blockcoo_Val, (coosize) * sizeof(MAT_VAL_TYPE));

cudaMemcpy(d_coo_compressed_Idx, coo_compressed_Idx, (coosize) * sizeof(unsigned char), cudaMemcpyHostToDevice);
cudaMemcpy(d_Blockcoo_Val, Blockcoo_Val, (coosize) * sizeof(MAT_VAL_TYPE), cudaMemcpyHostToDevice);
```

分别进行的malloc

图 11 分别进行的 malloc

4. 关于自适应感知内存分配颗粒

想法：通过检查预定义的宏来实现。nvcc 和 hipcc 这两个编译器都有特定的预定义宏，这些宏可以在编译时用来识别编译器。从而实现特定的代码，然后在代码中，通过这些宏来进行内存分配，计算，从而实现自适应感知内存分配颗粒。

```cpp
#include <iostream>
# ifdef __HIPCC__
#include <hip/hip_runtime.h>
# endif
# ifdef __CUDACC__
#include <cuda_runtime.h>
# endif
using namespace std;
int main() {
    #if defined(__CUDACC__)
    cout << "Compiled with nvcc (CUDA compiler).\n";
    int *d_a;
    cudaMalloc(&d_a, 10*sizeof(int));
    cout<<"d_a: "<<d_a<<endl;
    int *d_b;
    cudaMalloc(&d_b, 10*sizeof(int));
    cout<<"d_b: "<<d_b<<endl;
    cout<<"the block size is: "<<(d_b - d_a)*sizeof(int)<<endl;
    #elif defined(__HIPCC__)
    cout << "Compiled with hipcc (HIP compiler).\n";
    int *d_a;
    hipMalloc(&d_a, 10*sizeof(int));
    cout<<"d_a: "<<d_a<<endl;
    int *d_b;
    hipMalloc(&d_b, 10*sizeof(int));
    cout<<"d_b: "<<d_b<<endl;
    cout<<"the block size is: "<<(d_b - d_a)*sizeof(int)<<endl;
    #else
    cout << "Compiled with another compiler.\n";
    #endif
    return 0;
}
```

Next

（1）阅读 CSR5 的论文

（2）完善专利的代码

（3）继续研究 TileSpMV 源代码