大型张量的 CP 分解算法

张晨

(计算机系，学号：2017011307，手机号：13730916126)

摘　要: 本文讨论了张量 CP 分解算法的优化方法，并在使用近年来对张量分解的优化方法的基础上，测试了使用并行、GPU 进行张量分解的效果。实验表明，在对张量分解过程进行简单调整后，张量分解具有很好的扩展性，能较好利用矩阵的稀疏性，且十分适用于使用 GPU 进行运算。

关键词: CP 分解，并行张量分解，GPU 优化

1. 引言

张量，即高维数组，可以对现实世界中的多种数据进行建模。张量分解可提取出数据中的潜在信息，被广泛地应用于各类数据挖掘任务中，如社交网络分析、推荐系统、音频视频处理等[1]。常见的张量分解方法有 CP 分解、Tucker 分解、DEDICOM 等等。CP 分解的表达较为简洁，分解结果具有一定的可解释性，本文将对其进行着重讨论。

CP 分解[2]将一个 I*J*K 的三维矩阵分解成 F 个外积之和：

$$\mathrm{X} = \sum_{f=1}^{F} a_f \circ b_f \circ c_f \tag{1}$$

∘表示向量外积，满足

$$a_f \circ b_f \circ c_f(\mathrm{i},\mathrm{j},\mathrm{k}) = a_f(i)b_f(j)c_f(k)$$

其中，$1 \le \mathrm{i} \le \mathrm{I}, 1 \le \mathrm{j} \le \mathrm{J}, 1 \le \mathrm{k} \le \mathrm{K}, \mathrm{a_f} \in R^{I\times 1}, b_f \in R^{J\times 1}, \mathrm{c_f} \in R^{K\times 1}$。最小的使式(1)成立的 F 的值被称为张量 X 的秩。在一定条件下，张量的 CP 分解是唯一的[1]，即给定张量 X 和它的秩 F，仅有一种将 X 分解成 F 个秩为 1 的张量相加的方法。因此，CP 分解本身便反映出数据的一些性质。

寻找秩为 F 的 CP 分解，使之与 X 之差的 F-范数最小，即最小化$\left\|X - \sum_{f=1}^{F} a_f \circ b_f \circ c_f\right\|_F$ 是一个 NP-hard 的问题，但是 ALS(alternating least squares)算法可以获得此问题的一个很好的解。然而，ALS 算法难以扩展到较大的张量上，其最大原因是 ALS 算法的中间过程的数据量远大于其结果的数据量，这被称作“中间数据爆炸问题”[3]。此外，在 ALS 的每轮迭代中，整个张量中的数据都要被访问多次，且访问模式不同，这使得运算过程中难以利用缓存，且在分布式内存系统中运算会带来巨大的通信开销[2]。

PARACOMP(parallel randomly compressed tensor decomposition)[5]是进行大张量 CP 分解的一种备选方案，它将张量 X 随机压缩成若干个较小的张量，分别对这些张量进行分解，将分解结果合并得到张量 X 的 CP 分解。如果张量 X 的秩较小，且运行过程中选取了较为合适的超参数，此方法会有很好地效果。“分别分解这些张量”的并行化十分容易，但是“随机压缩”过程却难以并行 。

鉴于此，Ravindran[2]提出两种方案，分别对 ALS 和 PARACOMP 进行优化，在总的浮点运算次数基本不变的情况下，降低了二者的内存用量，并使得这二者易于并行。本文将对这两种方法的实际效果进行实验验证。

张量分解的主要瓶颈为大量的矩阵乘法运算，十分适合使用 GPU 计算。本文将 Ravindran 优化后的 ALS 移植到 GPU 上，取得了较好的加速效果。

2. 记号

本文中用到的记号如表 1 所示，记号取自文献[4].

表 1 本文中用到的记号

记号	定义
diag(a)	以向量 a 为对角线的对角阵
A^+	矩阵 A 的广义逆
X(:,i)	矩阵 X 的第 i 列
X(i,:)	矩阵 X 的第 i 行
vec(X)	将矩阵 X 向量化

本文中还涉及以下运算：

定义 1(向量外积). 对于两个向量$\mathbf{a} \in \mathrm{R}^{\mathrm{I}}$，$\boldsymbol{b} \in R^{J}$，它们的外积$\mathbf{a} \circ \mathbf{b}$是一个$\mathrm{I} \times \mathrm{J}$维矩阵，矩阵第 i 行第 j 列的元素是a(i)b(j)。

定义 2(张量积). 对于两个矩阵$\mathrm{A} \in \mathrm{R}^{I\times J}$，$B \in R^{K\times L}$，他们的张量积是一个IK × JL维矩阵：

$$\mathrm{A} \otimes \mathrm{B} = \begin{bmatrix} A(1,1)B & \cdots & A(1,J)B \\ \vdots & \ddots & \vdots \\ A(I,1)B & \cdots & A(I,J)B \end{bmatrix}$$

定义 3(克罗内克积). 对于两个矩阵$\mathrm{A} \in \mathrm{R}^{\mathrm{I}\times\mathrm{K}}$,$\mathrm{B} \in \mathrm{R}^{\mathrm{J}\times\mathrm{K}}$,它们的克罗内克积是一个(I ∗ J) × K维矩阵。

$$\mathrm{A} \odot \mathrm{B} = \begin{bmatrix} A(1,1)B(:,1) & \cdots & A(1,K)B(:,K) \\ \vdots & \ddots & \vdots \\ A(1,I)B(:,1) & \cdots & A(I,K)B(:K) \end{bmatrix}$$

定义 4(张量的矩阵展开). 一个 N 维张量$\mathrm{X} \in \mathrm{R}^{I_1\times I_2\times\ldots\times I_N}$在第 n 维的矩阵展开记为$\mathrm{X}_{(\mathrm{n})} \in \mathrm{R}^{I_n\times I_1\ldots I_{n-1}I_{n+1}\ldots I_N}$，它是平铺矩阵 X 在第 n 维的所有向量而得到的，[4]提供了一个较为易懂的示例。

3. 张量分解算法

3.1 ALS算法

这里仅讨论三维张量，对于一般化的 n 维张量，请参考[4]。

令因子矩阵 A、B、C 表示 X 的 CP 分解结果排列而成的矩阵：

$$\mathrm{A} = [\mathrm{a}_1\ \mathrm{a}_2\ \ldots\ \mathrm{a}_\mathrm{F}] \in \mathrm{R}^{\mathrm{I}\times\mathrm{F}},\ \mathrm{B} = [\mathrm{b}_1\ b_2\ \ldots\ b_F] \in R^{J\times F},\ \mathrm{C} = [\mathrm{c}_1\ c_2\ \ldots\ c_F] \in R^{K\times F}$$

则有

$$\mathrm{X}_{(1)} = A(C \odot B)^T,\ \mathrm{X}_{(2)} = B(C \odot A)^T,\ \mathrm{X}_{(3)} = C(B \odot A)^T$$

求解的目标函数可写为

$$\min_{\mathrm{A,B,C}} \left\| X_{(1)} - A(C \odot B)^T \right\|_F$$

若固定 B、C，则可求得

$$\mathrm{A} = \mathrm{X}_{(1)}(C \odot B)^+$$

利用克罗内克积的性质[4]

$$(\mathrm{A} \odot \mathrm{B})^+ = \left((A^TA)(B^TB)\right)^+(A \odot B)^T$$

可得

$$\mathrm{A} = \mathrm{X}_{(1)}(C \odot B)(C^TC * B^TB)^+ \tag{2}$$

相似的，有

$$\mathrm{B} = \mathrm{X}_{(2)}(C \odot A)(C^TC * A^TA)^+ \tag{3}$$

$$\mathrm{C} = \mathrm{X}_{(3)}(B \odot A)(B^TB * A^TA)^+ \tag{4}$$

不断利用式(2)-(4)进行迭代计算，即可获得张量 X 的秩为 F 的 CP 分解。见算法 1。

算法 1： ALS 算法计算 CP 分解

输入：张量$\mathrm{X} \in \mathrm{R}^{I\times J\times K}$，秩 R

输出：CP 分解结果，$\mathrm{A} \in \mathrm{R}^{I\times R}, B \in R^{J\times R}, C \in R^{K\times R}$

1：A，B，C 随机初始化

2：**while** 不达到收敛条件 **do**

3：　$\mathrm{A} \leftarrow \mathrm{X}_{(1)}(C \odot B)(C^TC * B^TB)^+$

4：　$\mathrm{B} \leftarrow \mathrm{X}_{(2)}(C \odot A)(C^TC * A^TA)^+$

5：　$\mathrm{C} \leftarrow \mathrm{X}_{(3)}(B \odot A)(B^TB * A^TA)^+$

6：**end while**

3.2 PARACOMP

PARACOMP 的主要流程如图 1 所示。

Fork 过程将 P 个三元组$(\mathrm{U}_\mathrm{p}, V_p, W_p)$分别与张量 X 相乘，得到 P 个压缩后的矩阵$\{\mathrm{Y}_\mathrm{p}\}_{\mathrm{p}=1}^{P}$。具体的，对于给定的参数矩阵$\mathrm{U}_\mathrm{p} \in R^{I\times L_P}$，$\mathrm{V}_\mathrm{p} \in R^{J\times M_P}$，$\mathrm{W}_\mathrm{p} \in R^{K\times N_P}$，压缩后的矩阵$\mathrm{Y}_\mathrm{p} \in R^{L_p\times M_p\times N_p}$满足：

$$\mathrm{Y}_\mathrm{p}(l,m,n) = \sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K} U_p(l,i)V_p(m,j)W_p(n,k)X(i,j,k) \tag{5}$$

图 2 是张量压缩过程的示意图。

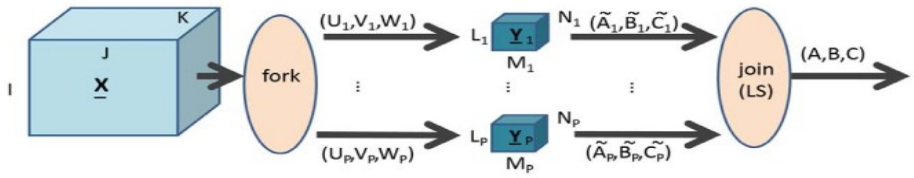

图 1 PARACOMP 算法流程

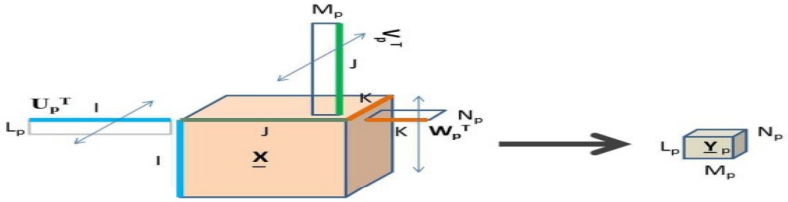

图 2 张量压缩过程示意

之后，单独分解每一个矩阵$\mathrm{Y_p}$，得到分解结果$(\widetilde{A_p},\widetilde{B_p},\widetilde{C_p})$。

注意到(5)式满足

$$vec(\mathrm{Y_p}) = (U_p^T \otimes V_p^T \otimes W_p^T)vec(X) \tag{6}$$

且由于克罗内克积满足性质

$$(\mathrm{U_p^T} \otimes \mathrm{V_p^T} \otimes \mathrm{W_p^T})(\mathrm{A} \odot \mathrm{B} \odot \mathrm{C}) = (\mathrm{U^T A}) \odot (\mathrm{V^T B}) \odot (\mathrm{W^T}C)$$

故

$$vec(Y_p) = \left((\mathrm{U_p^T A}) \odot (\mathrm{V_p^T B}) \odot (\mathrm{W_p^T}C)\right)\mathbf{1}$$

因而分解结果满足$\widetilde{A_p} = \mathrm{U_p^T A}, \widetilde{B_p} = \mathrm{V_p^T B}, \widetilde{C_p} = \mathrm{W_p^T}C$，据此可恢复出张量 X 的分解结果。

对于 PARACOMP 算法的可行性，有如下定理

定理 1(压缩后矩阵分解唯一的条件). 令$\mathrm{x} = (\mathrm{A} \odot \mathrm{B} \odot \mathrm{C})\mathbf{1} \in \mathrm{R^{IJK}}$，其中 A、B、C 分别为$\mathrm{I \times F}$、$\mathrm{J \times F}$、$\mathrm{K \times F}$维矩阵，第 p 个压缩结果$\mathrm{y_p} = (U_p^T \otimes V_p^T \otimes W_p^T)x = ((U^TA) \odot (V^TB) \odot (W^TC))\mathbf{1} = (\widetilde{A_p} \odot \widetilde{B_p} \odot \widetilde{C_p})\mathbf{1} \in R^{LMN}$，其中参数矩阵$\mathrm{U(I \times L, L \le I), V(J \times M,\ M \le J), W(K \times N, N \le K)}$分别从$R^{I \times L_P}$，$R^{J \times M_P}$，$R^{K \times N_P}$的连续分布中独立选取，A、B、C 都是秩为 F 的列满秩矩阵，$\mathrm{L \le M \le N}$，且

$$(\mathrm{L}+1)(\mathrm{M}+1) \ge 16F$$

则 $\tilde{A}_p$，$\widetilde{B_p}$，$\widetilde{C_p}$除顺序和相对大小外几乎可以由 y 唯一确定。

定理 2(合成结果唯一的条件). 在满足定理 1 的前提下，若每个$\mathrm{W_p}$有两个公共列，且

$$\mathrm{P} \ge \max\left(\frac{\mathrm{I}}{\mathrm{L_P}}, \frac{\mathrm{J}}{\mathrm{M}}, \frac{K-2}{N-2}\right)$$

则(A,B,C)除顺序和相对大小外基本可以由$\{(\tilde{A},\ \tilde{B},\ \tilde{C})\}_{\mathrm{p=1}}^{\mathrm{P}}$唯一确定。

4. 张量分解算法的瓶颈及优化

4.1 ALS算法

ALS 算法的核心思想是式(2)至式(4)的迭代计算。

在式(2)中，定义 MTTKRP 操作为

$$\mathrm{Y} = \mathrm{X_{(1)}}(C \odot B)$$

在张量 X 的元素个数$\mathrm{I \times J \times K}$很大时，计算克罗内克积$\mathrm{C \odot B}$以及 MTTKRP 十分困难，这使得 ALS 算法难以应用在大型的张量上，这被称作“中间数据爆炸问题”。注意式(4)中剩余的部分$\mathrm{C^T}CB^TB$仅为一个$\mathrm{F \times F}$的矩阵，故求其广义逆运算量较小。

此外，Ravidran[2]指出，$\mathrm{X_{(1)}}, X_{(2)}, X_{(3)}$中对 X 的数据访问模式是不同的，这使得在计算过程中较难通过利用 cache 加快对内存中张量 X 的数据的访问（除非在内存中将这个张量在内存中复制三遍，分别按$\mathrm{X_{(1)}}$、$\mathrm{X_{(2)}}$、$\mathrm{X_{(3)}}$的格式存储）。此外，大型张量有可能是分布式存储，在这种情况下，访问模式不同的问题严重阻碍 ALS 算法的并行。

算法 2：计算$\mathbf{X}_{(3)}(\mathbf{B}\odot\mathbf{A})$

输入：张量$\mathrm{X}\in \mathrm{R}^{I\times J\times K}, A\in R^{I\times F}, B\in R^{J\times F}$

输出：$\mathrm{M}\leftarrow \mathbf{X}_{(3)}(\mathbf{B}\odot\mathbf{A})\in \mathbf{R}^{K\times F}$

1：$\mathrm{M}\leftarrow 0$

2：**for** $\mathrm{k}=1,\ldots,\mathrm{K}$ **do**

3：　$\mathrm{M}(\mathrm{k},:,:)\leftarrow 1^{T}(A*(X(:,:,k)B))$

4：**end for**

算法 3：计算$\mathbf{X}_{(1)}(\mathbf{C}\odot\mathbf{B})$

输入：张量$\mathrm{X}\in \mathrm{R}^{I\times J\times K}, B\in R^{J\times F}, C\in R^{K\times F}$

输出：$\mathrm{M}\leftarrow \mathbf{X}_{(1)}(\mathbf{C}\odot\mathbf{B})\in \mathbf{R}^{I*F}$

1：$\mathrm{M}\leftarrow 0$

2：**for** $\mathrm{k}=1,\ldots,\mathrm{K}$ **do**

3：　$\mathrm{M}(\mathrm{k},:,:)\leftarrow M+X(:,:,k)B\ diag(C(k,:))$

4：**end for**

算法 4：计算$\mathbf{X}_{(2)}(\mathbf{C}\odot\mathbf{A})$

输入：张量$\mathrm{X}\in \mathrm{R}^{I\times J\times K}, A\in R^{I\times F}, C\in R^{K\times F}$

输出：$\mathrm{M}\leftarrow \mathbf{X}_{(2)}(\mathbf{C}\odot\mathbf{A})\in \mathbf{R}^{J*F}$

1：$\mathrm{M}\leftarrow 0$

2：**for** $\mathrm{k}=1,\ldots,\mathrm{K}$ **do**

3：　$\mathrm{M}(\mathrm{k},:,:)\leftarrow M+X(:,:,k)A\ diag(C(k,:))$

4：**end for**

Ravidran[2]的主要思想如算法 2-算法 4 所示

使用算法 3 计算$\mathbf{X}_{(1)}(\boldsymbol{C}\odot\boldsymbol{B})$，仅需要 O(NNZ)的额外内存，其中，NNZ(number of non-zero elements)表示张量 X 中非零元素的个数。令I_k表示第 k 个I × J的张量“切片”X(:,:,k)中非零行的个数，J_k表示 X(:,:,k)中非零列的个数，令

$$\mathrm{NNZ}_1=\sum_{k=1}^{K}I_k$$

$$\mathrm{NNZ}_2=\sum_{k=1}^{K}J_k$$

分别表示张量 X 中非零“行”和非零“列”的个数，则算法 1 的总浮点运算次数(flop)为

$$\mathrm{F}*\mathrm{NNZ}_1+\mathrm{F}*\mathrm{NNZ}_2+2\mathrm{F}*\mathrm{NNZ}$$

其中，计算$\mathrm{B}*\mathrm{diag}(\mathrm{C}(\mathrm{k},;))$时仅需计算 X(:,:,k)对应列非空时的结果，共需$\mathrm{F}*\mathrm{NNZ}_2$次浮点运算；$\mathrm{X}(:,:,\mathrm{k})*(\mathrm{B}*\mathrm{diag}(\mathrm{C}(\mathrm{k},;)))$总共需要 2F*NNZ 次浮点运算（乘法和加法）；只有对应 X(:,:,k)的非空行的M_1会被更新，更新每行需要 F 次运算，共计$\mathrm{F}*\mathrm{NNZ}_1$次。故算法 3 在稀疏矩阵上表现良好。算法 3 的并行实现，仅需使用 K 个线程并行执行步骤 2-3。且由于每个线程仅需要访问矩阵 X 的一个切片 X(:,:,k)，此算法也很容易拓展到分布式内存系统上。

算法 4 计算$\mathbf{X}_{(2)}(\boldsymbol{C}\odot\mathrm{A})$)，其过程与算法 3 基本一致，故内存、计算开销与算法 3 相同。为了保证仍是按“切片”对张量 X 进行访问，算法 2 计算$\mathbf{X}_{(3)}(\boldsymbol{B}\odot\mathrm{A})$)的方法明显有异于算法 3、4，但内存、计算开销与算法 3、4 相似。算法 2 的并行也较为容易，仅需使用 K 个线程分别执行步骤 1-2.。

4.2 PARACOMP算法

表面上，PARACOMP 算法很适合张量的并行分解，因为在 Fork 过程完成后，每个线程仅需要访问一个压缩后的张量。然而，Fork 过程实际上是 PARACOMP 算法的瓶颈之一。

实现(5)式的一种方法是，仅枚举X(i, j, k)非零的三元组(i, j, k)，这样几乎不需要额外的存储空间，且能充分利用矩阵的稀疏性。但是，此方法应用于稠密矩阵、稀疏矩阵的时间复杂度分别为 O(LMNIJK)和 O(LMN(NNZ))，在矩阵中元素较多时开销很大。

另一种方法，可利用求和的性质，将(5)式化为

$$\mathrm{Y_p}(l,m,n)=\sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K}U_p(l,i)V_p(m,j)W_p(n,k)X(i,j,k)$$
$$=\sum_{k=1}^{K}W_p(n,k)\sum_{j=1}^{J}V_p(m,j)\sum_{i=1}^{I}U_p(l,i)\,X(i,j,k)$$

故可用如下方法计算压缩后的矩阵：

$$\mathrm{T_1}(l,j,k)=\sum_{i=1}^{I}U(l,i)X(i,j,k) \tag{7}$$

$$\mathrm{T_2}(l,m,k)=\sum_{j=1}^{J}V(m,j)T_1(l,j,k) \tag{8}$$

$$\mathrm{Y(l,m,n)}=\sum_{k=1}^{K}W(n,k)T_2(l,m,k) \tag{9}$$

此方法的时间复杂度仅为$\mathrm{O(LIJK+MLJK+NLMK)}$。在$\frac{\mathrm{I}}{\mathrm{L}}=\frac{J}{M}=\frac{K}{N}$，即每一维压缩率相同时，为使运算次数达到最小，会首先对张量 X 进行转置操作，使之满足$\mathrm{I}\le\mathrm{J}\le\mathrm{K}$。此算法对稠密阵有很好的效果，而对于稀疏矩阵，尽管式(7)可以仅访问 X 中的非零元素，但是通常情况下，运算的中间结果$\mathrm{T_1},T_2$是稠密矩阵，可能比 X 使用更多的内存，因此该方法在稀疏矩阵下表现并不好。

综合以上两种方法，Ravindran[2]提出了一种基于分块的计算方法(算法 5)，对稠密阵，其浮点运算次数与式 7-9 相同，而对于稀疏阵，其浮点运算次数小于式(5)。

算法 5：将 X 压缩为$\mathbf{Y_p}$

输入：张量$\mathrm{X}\in\mathrm{R}^{I\times J\times K},U_p\in R^{I\times L_p},V_p\in R^{J\times M_p},W_p\in R^{K\times N_p}$,块的大小 B
输出：$\mathrm{Y_p}\in R^{L_p\times M_p\times N_p}$

```
for k = 1, ..., K do
    T2' ← 0
    for b = 1, B+1, 2B+1, 3B+1, ..., J do
        T1' ← Up^T X(:, b:(b+B−1), k)
        T2' ← T2' + T1' Vp(b:(b+B−1), :)
    end for
    for n = 1, ..., Np do
        Yp(:, :, n) ← Yp(:, :, n) + W(n, k) T2'
    end for
end for
```

使用此算法，仅需要存储$\mathrm{T_1'}\in R^{L\times B},T_2'\in R^{L\times M}$两个矩阵的中间数据。取$\mathrm{B}=1$最节约内存，但是一般情况下会将 B 的值取为一个较大的数，从而能充分利用 cache。

算法 5 的时间复杂度为$\mathrm{O(L(NNZ)+LM(NNZ_2)+LMNK)}$，其中$\mathrm{L(NNZ),LM(NNZ_2),LMNK}$分别表示第 4、5、8 步的复杂度。对稀疏矩阵，式 5 的时间复杂度为 O(LMN(NNZ))。一般而言，$\mathrm{NNZ}>\mathrm{NNZ_2},NNZ>K$，故算法 5 比式 5 的浮点运算次数更少。

然而，Ravindran[2]也指出，算法 5 仅在对第一维进行压缩时充分利用了矩阵的稀疏性，在压缩第二维时仅能跳过全空的“列”，在压缩第三维时与稠密阵没有区别。因此 Ravindran[2]认为，此算法还有进一步的优化空间。

此算法最直接的并行方法是使用 P 个线程分别运行 P 个压缩过程，但这意味着每个线程都需要访问张量 X 的所有数据。更理想的方法是使用 K 个线程并行化算法 5 中的步骤 1，这样每个线程仅需要访问张量 X 的一个“切片”X(:,:,k)。

5. 实验与讨论

本节通过随机生成的矩阵，对 Ravindran[2]提出的两个优化及 CP 分解的 GPU 实现的性能进行测试。实验所用 CPU 为 Intel 酷睿 i7 7700HQ，GPU 为 GeForce GTX 1060。实验的操作系统为 Windows 10，g++编译器为 g++-5.1.0-tdm-1，python 版本为 python3.7.0。串行算法使用 C++实现，并行版本使用 C++ 与 openmp 实现，GPU 版本

使用 python+pytorch1.0.0。

5.1 ALS算法的优化效果

Ravindran[2]通过改变矩阵乘法的顺序，使 MTTKRP 操作充分利用系统的 cache。本实验随机生成了$n \times n \times n$的张量 X 及$n \times F$的矩阵 A、B、C，测试该优化对稠密矩阵的效果，如图 3、4 所示。

使用经典方法计算$X_{(3)}(B \odot A)$明显慢于计算$X_{(1)}(C \odot B)$和$X_{(2)}(C \odot A)$，这是因为在计算过程中，对张量 X 的访问不是连续的，会存在如下代码片段：

```
for (int k = 0; k < K; k++)
    for (int i = 0; i < I; i++)
        for (int j = 0; j < J; j++)
            /*...*/x[i][j][k]/*..*/;
```

而在优化版的算法中，三次计算对张量 X 的访问都是连续的，且相比于经典算法计算$X_{(1)}(C \odot B)$和$X_{(2)}(C \odot A)$没有明显的性能损失。由于计算方法的改变，algorithm1-3 与 baseline1-3 在计算性能上没有一一对应关系，故使用 algorithm1-3 中最慢的与 baseline1-3 中最慢的（即 baseline3）计算加速比。从图 3 中可以发现，在张量 X 的规模 n 较大时，该优化约能获得 1.1×的加速比。

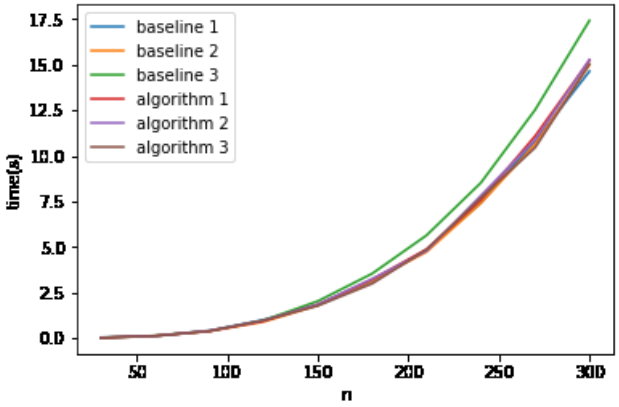

图 3 MTTKRP 各部分运行时间对比。baseline1-baseline3 分别使用经典方法计算$X_{(1)}(C \odot B)$、$X_{(2)}(C \odot A)$、$X_{(3)}(B \odot A)$，algorithm1-algorithm3 分别对应于本文中的算法 2-4。实验中取 F=100.

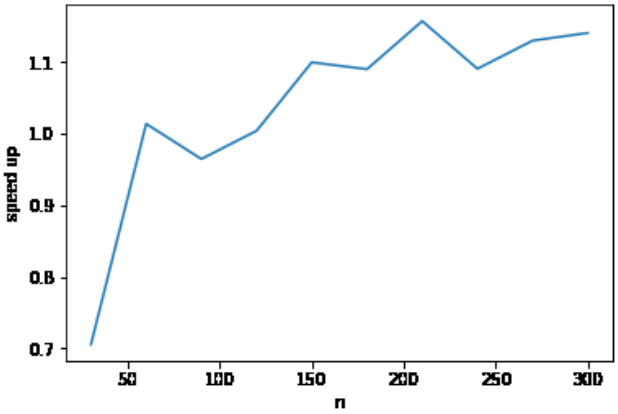

图 4 优化版 MTTKRP 的加速比，计算公式为 $\frac{t(\text{baseline3})}{\max(t(algorithm1), t(algorithm2), t(algorithm3))}$ 其中 t()表示响应算法的运行时间。

5.2 并行MTTKRP过程

MTTKRP 过程中，各线程基本没有依赖关系，且本实验采用了共享内存方式，故 MTTKRP 过程扩展性良好实验结果如图 5-6 所示。从理论上分析，执行 MTTKRP 过程仅需要同步计算得到的 ABC，通信量较 X 的规模可以忽略不计，故可认为在分布式系统中，MTTKRP 过程也有很好的表现。

5.3 MTTKRP过程的GPU实现

本实验使用了 pytorch 的 API，实现了 GPU 上的 MTTKRP 过程。实验结果如表 2 所示。在$X \in R^{900\times900\times900}, B \in R^{900\times100}, C \in R^{900\times100}$时，仅需要 0.03 秒。作为对比，使用 CPU，8 个线程运行$X \in R^{300\times300\times300}, B \in R^{300\times100}, C \in R^{300\times100}$的计算任务，需要约 4.38 秒。受内存与显存的限制，本实验未测试更大规模的矩阵。可见，十分适合使用 GPU 进行此过程的计算。

表 2 $X \in R^{n\times n\times n}, B \in R^{n\times100}, C \in R^{n\times100}$时 MTTKRP 运行时间。由于总的计算时间很小，未能观测到输入数据规模不同时运算时间的显著差异。

n	运行时间(s)	n	运行时间(s)
90	0.0020	540	0.0033
180	0.0033	630	0.0036
270	0.0054	720	0.0036
360	0.0061	810	0.0262
450	0.0044	900	0.0293

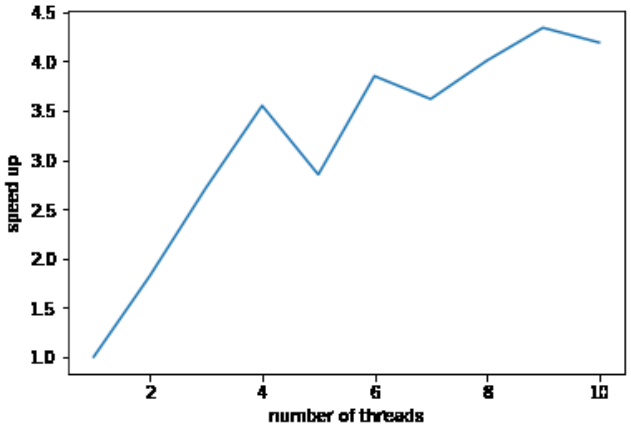

图 5 MTTKRP algorithm2 并行实现的加速比。张量 X 为随机生成的$300 \times 300 \times 300$的张量，取 F=100。加速比的计算公式为 $\text{speed up} = \frac{\text{线性运行时间}}{\text{使用当前线程数的运行时间}}$

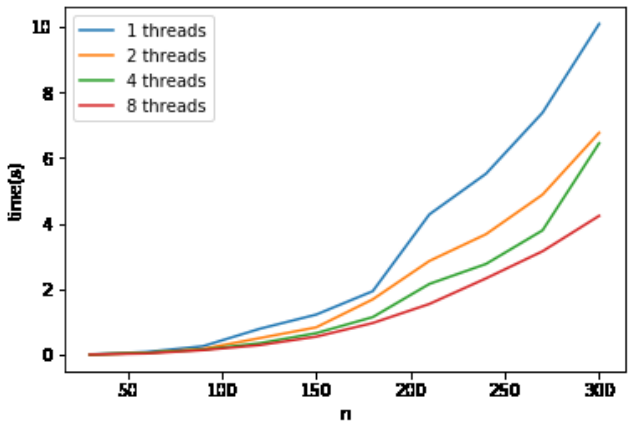

图 6 MTTKRP algorithm2 不同规模，不同进程数的运行时间对比。张量 X 为随机生成的$n \times n \times n$的张量，取 F=100。

5.4 Paracomp中优化版矩阵压缩的性能

本实验验证了优化版矩阵压缩对于矩阵稀疏性的利用，实验结果如图 7 所示。尽管在$T_2(l,m,k) = \sum_{j=1}^{J} V(m,j)T_1(l,j,k)$、$Y(l,m,n) = \sum_{k=1}^{K} W(n,k)T_2(l,m,k)$两步中没有利用矩阵的稀疏性，但这两步计算使用的矩阵规模较第一步$T_1(l,j,k) = \sum_{i=1}^{I} U(l,i)X(i,j,k)$已经有所缩小，故未对整体性能造成太大影响。实验表明，优化版压缩总体运行较快，且能较好利用矩阵的稀疏性。再加入用以节约内存的分块算法，可较好完成矩阵压缩任务。

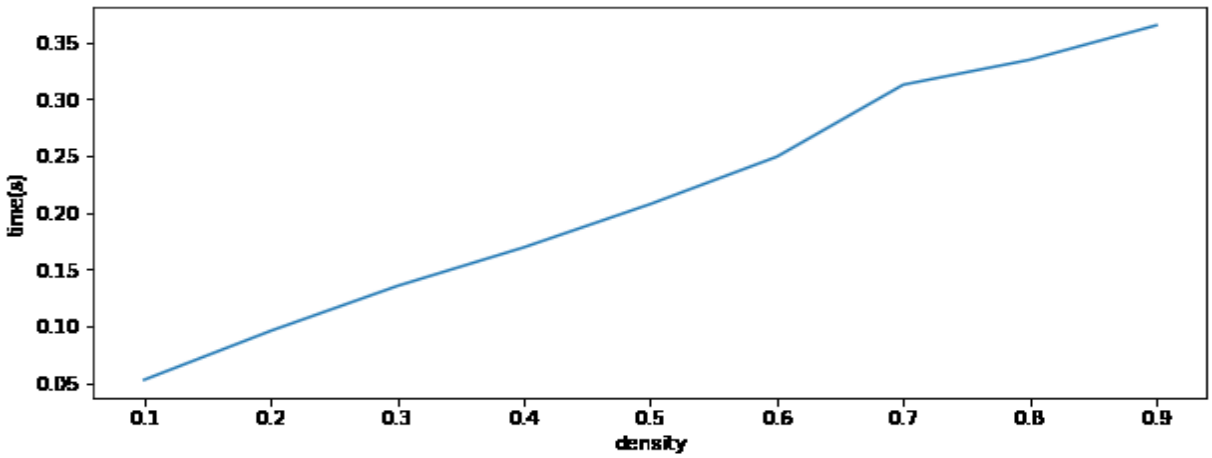

图 7 优化版 paracomp 对矩阵稀疏性的利用。实验随机生成有$n^3 * density$个非零元素的张量 X，张量$X \in R^{n \times n \times n}$，压缩使用的参数矩阵为$n \times P$的张量 U、V、W。取 n=200，P=10.

6. 总结

本文讨论了较大规模张量的 CP 分解方法。无论是 ALS 还是 Paracomp，其瓶颈都在较为简单的矩阵乘法运算，这些矩阵运算导致 CP 分解中间过程中消耗大量额外内存，占用大量运算时间。实验表明，通过适当调整运算顺序、分块计算等方法，可以极大减小额外内存的消耗量；充分利用系统的 cache，减小算法的常数；使程序易于并行化或 GPU 移植。受计算资源所限，本文未真正使用大规模张量进行测试，仅对运算过程中的核心步骤进行实验验证，但这些实验足以表明在大规模集群上进行张量分解的可行性。在大规模集群上使用 ALS 进行张量 CP 分解的一种可行做法是，将$I \times J \times K$的张量X拆分为 K 个$I \times J$的“切片”$X(:,:,k)$ $k \in 1,2..K$，分布式存储在集群内的各台服务器内，各服务器使用 GPU 迭代计算A，B，C并对A，B，C进行同步。使用 Paracomp，则可按相同方法进行分布式存储，生成压缩后矩阵。之后将压缩后的矩阵分配到各个服务器上进行分解操作，最后进行合成。

参考文献

[1] Evangelos E. Papalexakis, Christos Faloutsos, and Nicholas D. Sidiropoulos. 2016. “Tensors for Data Mining and Data Fusion: Models, Applications, and Scalable Algorithms.” ACM Trans. Intell. Syst. Technol. 8, 2, Article 16 (October 2016), 44 pages.

[2] N. Ravindran, N. D. Sidiropoulos, S. Smith and G. Karypis, "Memory-efficient parallel computation of tensor and matrix products for big tensor decomposition," *2014 48th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, 2014, pp. 581-585.

[3] Brett W. Bader and Tamara G. Kolda. 2007. “Efficient MATLAB Computations with Sparse and Factored Tensors.” *SIAM J. Sci. Comput.* 30, 1 (December 2007), 205-231.

[4] Tamara G. Kolda and Brett W. Bader. 2009. “Tensor Decompositions and Applications.” SIAM Rev. 51, 3 (August 2009), 455-500.

[5] N. D. Sidiropoulos, E. E. Papalexakis and C. Faloutsos, "A parallel algorithm for big tensor decomposition using randomly compressed cubes (PARACOMP)," *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Florence, 2014, pp. 1-5.