Make Einsum 1000x faster and scalable

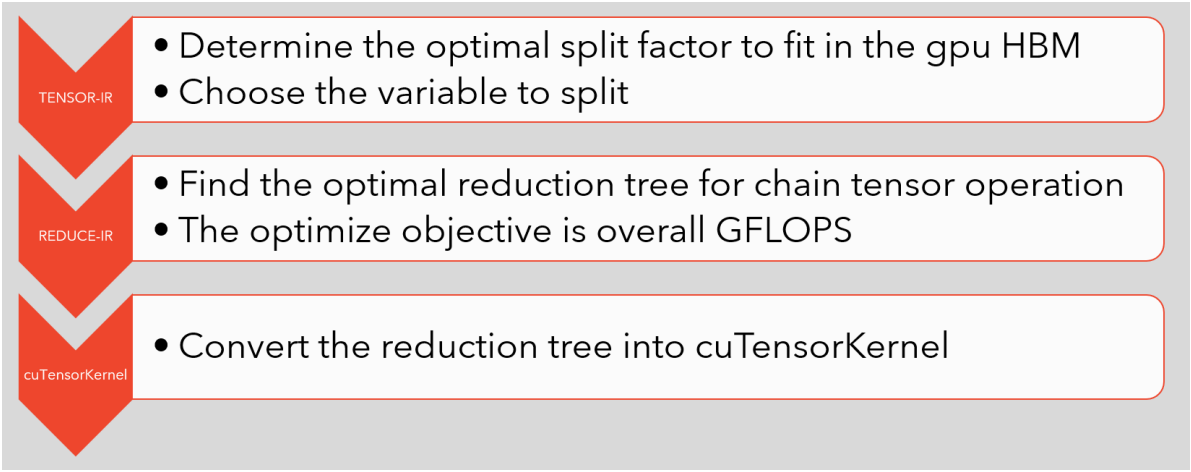
Intro

张量计算在科学计算中有着重要的作用，我们在黑客松比赛中， 结合开源工具和英伟达高性能的cuTensor库， 利用GPU加速相对于原$np.einsum$函数实现了1000倍以上的加速比，同时我们的解决方案并不受到GPU显存的限制，拓展到了上千万格点的矩阵组装。我们的解决方案也不受到该特定数据流的限制， 可以加速任何的$einsum$函数。

Contribution

- 思考了如果张量大小如果无法放进GPU显存时的情况并提出了解决方案
- 结合开源代码极大地优化了组装单元矩阵的过程

Overview



Tensor IR（Loop split)

对于任意形式的张量计算，我们都可以用循环来表示。

对于样例中的数据流

$$e_{j,k,m} = a_i b_{i,j,k,l} c_{i,j,m,l} d_j$$

可以被表达成如下的循环

```
for (int i = 1;i <= extent[i];++i)
for (int j = 1;i <= extent[j];++j)
for (int k = 1;i <= extent[k];++k)
for (int m = 1;i <= extent[m];++m)
for (int l = 1;i <= extent[l];++l)
    e[j][k][m]+=a[i]*b[i][j][k][l]*c[i][j][m][l]*d[j]
```

对于出现在结果中的下标，我们可以定义这些下标为$output_index$，而通过对$output_index$进行适当的划分，我们可以把工作切分成互不影响的子部分。

比如刚才的循环，就可以被转换为

```
for (int j_1 = 1;j_1 <= extent[j]/stride[j]+1;++j_1)
    workload(j_1);

workload(j_1):
 for (int j_2 = 1;j_2 <= stride[j_2];++j_2)
 for (int i = 1;i <= extent[i];++i)
 for (int k = 1;i <= extent[k];++k)
 for (int m = 1;i <= extent[m];++m)
 for (int l = 1;i <= extent[l];++l)
     j=j_1*stride[j]+j_2
     e[j][k][m]+=a[i]*b[i][j][k][l]*c[i][j][m][l]*d[j]
```

同时为了发掘更多的并行性，我们也可以切分更多的output_index维度然后通过交换顺序使得workload的下标具有更多维度。

```
for (int j_1 = 1;j_1 <= extent[j]/stride[j]+1;++j_1)
    for (int k_1 = 1;k <= extent[k]/stride[k]+1;++k_1)
        workload(j_1,k_1);

workload(j_1,k_1):
 for (int j_2 = 1;j_2 <= stride[j_2];++j_2)
 for (int k_2 = 1;k_2 <= stride[k_2];++k_2)
 for (int i = 1;i <= extent[i];++i)
 for (int m = 1;i <= extent[m];++m)
 for (int l = 1;i <= extent[l];++l)
     j=j_1*stride[j]+j_2
     k=k_1*stride[k]+k_2
     e[j][k][m]+=a[i]*b[i][j][k][l]*c[i][j][m][l]*d[j]
```

在实际工作中我们需要保证切分后的单元映射到连续的内存区域，所以我们最后手动切分了j这个维度并且更改了张量的模式顺序。

利用这个切分顺序，我们8秒内处理了2百万个格点的数据(p=3,q=3)。

Reduce IR（Determine the optimal reduce order）

上一步主要是解决计算受显存限制的问题， 这一步是为了性能的优化。

以下搜索内容在开源软件opt_einsum中已经被实现了，我们提交的代码直接利用了他们的实现。

一个张量连乘表达式的计算过程可以被抽象为一个表达式树。 因为实际上我们一次只能合并两个张量，而合并的顺序会影响性能。

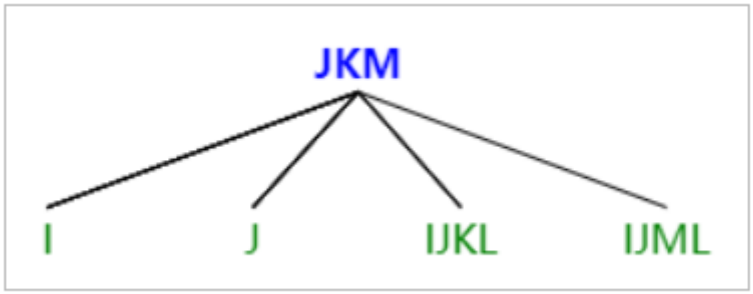
对于样例中的表达式，

$$e_{j,k,m} = a_i b_{i,j,k,l} c_{i,j,m,l} d_j$$

假设数据集大小为

$$[I, J, K, L, M] = [10, 100000, 20, 3, 20]$$

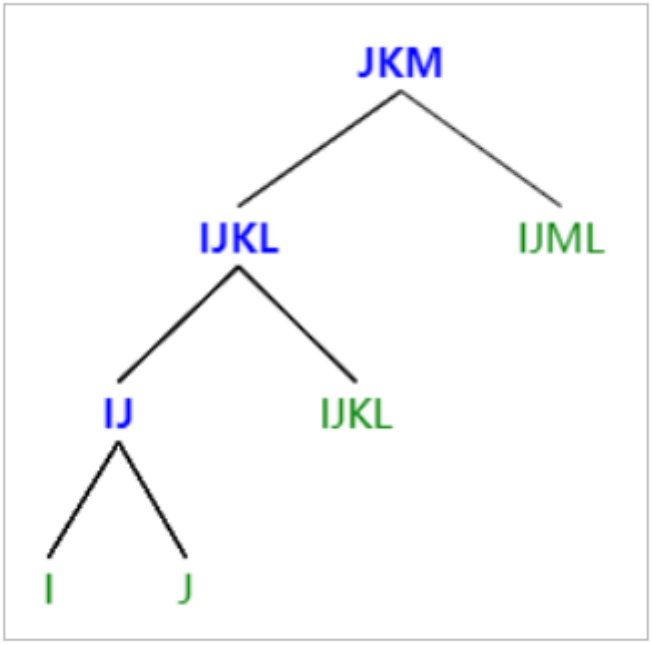
可以抽象到如下的表达式树， （I代表a, J代表b, 等等）



表达式树的运算代价就是所有非叶节点的权值，非叶节点的权值定义为该节点$reduce$操作的代价（忽略掉一些特殊情况，可以认为是该节点子节点和该节点上下标并集的大小乘以（子节点的个数减一）），在这棵树里面所有非叶节点的权值为（乘以3是因为需要连着三个乘法$a*b*c*d$）

$$3*IJKML=3*1.2e9=3.6e9$$

而经过搜索，最优的表达式树应该是这样的。



这时，所有非叶节点的权值为

$$IJKL+IJKML+IJ=6e7+4e7+1e6=1.3e9$$

cuTensor IR

上述reduce的结果会被转换成cuTensor Kernel(contraction)来最大化对GPU的利用。

Results & Experiment setup

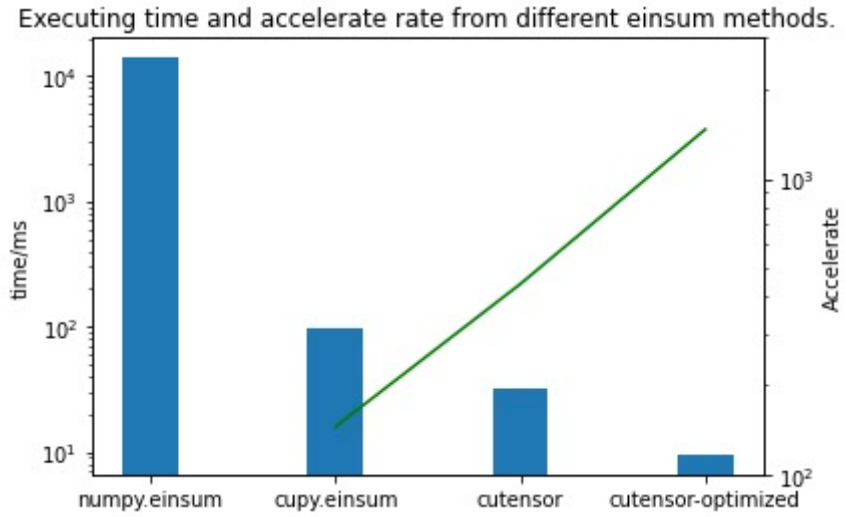
我们取得了非常好的效果，是原$np.einsum$速度的千倍以上，同时也是目前网上我们找到的最快的代码（$cupy.einsum$)的十倍以上。

注意由于cuTensor第一次调用会用1s左右时间加载handle和搜索一些超参数，我们把第一次调用的运行时间排除了。

我们数据集的张量由以下函数生成

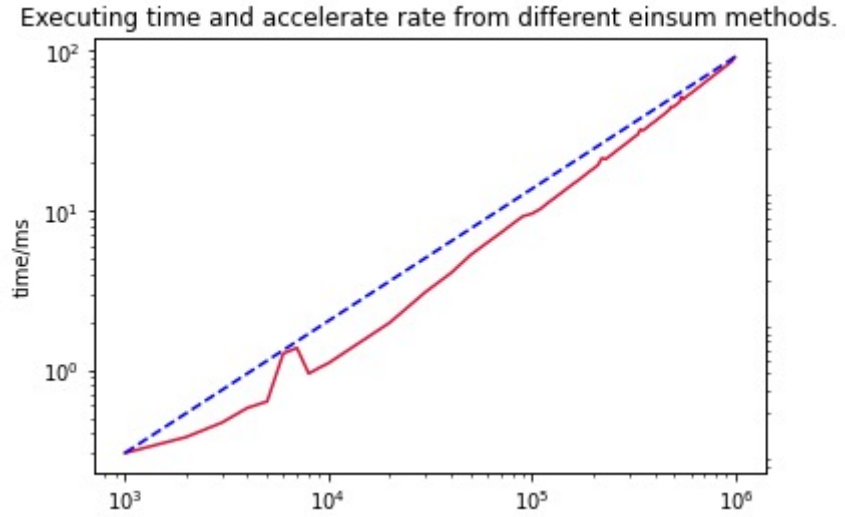
```
def generateTestDataDim(p, q, numberOfCell, geometryDim = 3, topologyDim = 3):
    I = q * (q+1) * (q+2) / 6
    K = M = (p+1) * (p+2) * (p+3) / 6
    J = numberOfCell
    L = geometryDim
    return list(map(int,(I,J,K,L,M)))
```

相关维度的大小均由样例代码推理得到。



这组数据是在$[p, q, nc] = [3, 3, 100000]$的时候得到的，坐标是对数坐标，最优的加速比达到了1470倍。

以下的数据展示我们的算法的拓展性



可以看到我们几乎保持了线性地增长。 注意两个坐标都是对数坐标。