基于 NiuTensor 的 8 位整型量化方法与系统实现

姓名：丁妍
学号：2101705
班级：计硕 2102 班
专业：计算机科学与技术

1 系统简介

本文基于 NiuTensor 深度学习框架，针对 Transformer 模型，设计实现了一个 8 位整型量化系统。该系统设计并实现了一个合理的量化方法，包括量化、反量化和重量化。同时针对底层的张量计算以及 Transformer 模型的各个部分，本文也设计了具体的量化操作，使得模型能够使用 8 位整型数进行推理。在数据读取以及存储方面，能够对浮点类型的模型文件进行读取并转换为 8 位整型，在保证推理结果正确性的基础上解决参数的冗余问题，提高计算效率，降低存储需求。

2 系统实现

系统实现是系统开发中的至关重要的阶段，系统的编码工作主要集中在此阶段。本小节主要从系统的开发环境，量化方法实现、量化操作实现、量化存取实现四方面来对系统的整体实现进行详细描述。

2.1 系统开发环境

2.1.1 开发环境

本系统基于 Windows 10 系统开发，使用 Microsoft Visual Studio 2017 及 PyCharm 2018.1 集成开发环境， Python 接口封装工具为 pybind11，使用的 Python 编程语言版本为 3.6.5，使用的其他工具有文本编辑工具 NotePad++。表 5.1 详细列出了系统开发的软件需求。

本系统主要使用 Windows10 系统进行开发，使用 Microsoft Visual Studio 2019 进行 8 位整型量化系统的编码实现工作，使用 Cmake 进行 NiuTensor 框架的编译，同时也使用 Visual Studio Code 进行了部分的编码实验。表 2.1 列出了在开发过程中使用的软件环境。

表 2.1 系统开发软件需求表

名称	版本	功能
Windows	10	操作系统
Visual Studio	2019	集成开发环境
Visual Studio Code	1.56.2	部分编码实验

Cmake	3.19.0	编译工具

2.2 量化方法实现

2.2.1 数据结构

在 NiuTensor 的计算框架中，XTensor 类被设计用来表示张量，表 2.2 将对其成员变量进行简要介绍。

由上一章的系统分析可知，在对张量进行量化后，本课题采用了在将张量的量化的尺度 scale 在计算时进行传播的方式，来避免在每一个计算操作间都使用量化和反量化的方法。这种将 scale 传播的方法减少了频繁量化反量化带来的计算代价，收益较高。由于量化是沿着张量的最后一个维度进行，所以 scale 也是具有维度和形状大小的，需要内存来存储数据。所以使用了如表 2.3 所示的数据结构对 scale 进行实现。

表 2.2 XTensor 部分成员变量及其功能

成员变量	功能
void* data	用来保存张量元素的数组
int order	张量的维度
int dimSize[MAX_TENSOR_DIM_NUM]	表示张量每一个维度的大小
TENSOR_DATA_TYPE datatype	记录张量的数据类型
unitSize	记录张量每个数据的大小（字节单位）
unitNum	张量中元素的个数
bool isSparse	表示张量是否稠密
Float denseRatio	表示张量的稠密度

表 2.3 scale 类构成

Scale 类

```
1  class Scale {
2  float *  values = NULL;
3  int order;
4  int unitnum;
5  int dimsize[MAX_SCALE_DIM] = {0};
6  bool allocated=false;
```

```
7    void setData(float *d,int unitnum);
8    void initScale(int order,const int * dimsize, float *d);
9    void initScale(int order, const int* dimsize);
10       };
```

其中 values 是保存 scale 数据的数组，order 代表 scale 的维度，dimsize 为 scale 每一维度的大小，allocated 代表 values 是否已经被分配内存。setData 方法是对 scale 的数据进行分配，因为有时候通常要先进行 scale 大小的初始化，再对 scale 的值进行计算，计算后再分配 scale 的值，所以该方法的设计实验是必要的。initScale 方法重载了两个实现，一个是是对 scale 的维度，各维度大小，数据进行初始化，一个是仅仅对维度和维度大小进行初始化，这两种方法都开辟了 scale 数值的内存空间。

由于在高度量化的系统中需要每一个张量都有其对应的 scale，方便 scale 的传播，所以讲 scale 添加至 XTensor 的成员变量中。相关的类图如图 2.1 所示。

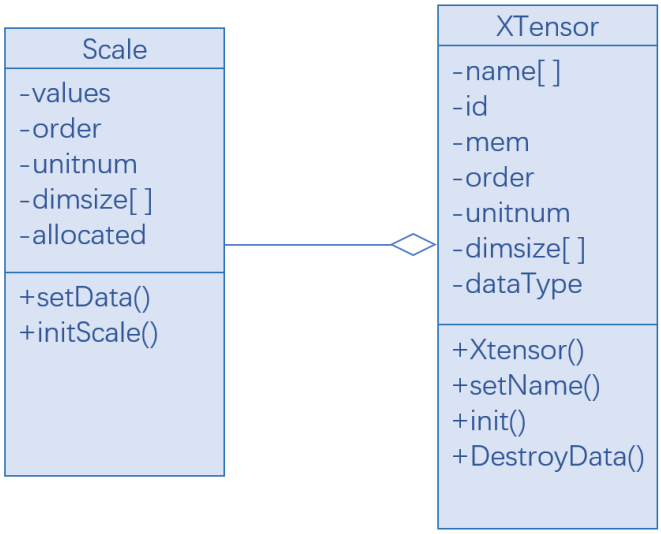

图 2.1 XTensor 与 scale 类图

2.2.2 量化实现

量化方法包括量化、反量化、重量化的操作，涉及这些的相关方法都在 quantize.cpp 中进行了实现，本小节主要介绍量化方法的实现。量化方法包括 quantize()，quantizeMatrix2DinCol()，quantizeMatrix2DinColtoScale()。下面将对其功能和实现进行具体介绍。

（1）void quantize(XTensor* input)

该方法重载了两种实现方式，一种具有 XTensor*类型的形参，另外一种具有 XTensor 和 float*类型的形参。都是对 XTensor 的数据进行量化，XTensor 的数据类型可以是浮点类型也可以是 8 位整型。将 XTensor 的数据沿着最后一个维度进行计算后，对其成员变量 scale 进行初始化。

（2）void quantizeMatrix2DinCol(XTensor* input)

根据第四章的系统设计可知，当进行矩阵乘法时，scale 的维度与矩阵乘法需要的维度不匹配，可能会对其另外一个维度进行量化。对于二维矩阵来说，普通的 quantize 方法相当于沿着矩阵的行进行量化，所以 quantizeMatrix2DinCol()实现的是对列进行量化。其形参是 XTensor 类型指针，直接对其 scale 进行修改，无返回值。

（3）float* quantizeMatrix2DinColtoScale(XTensor* input, float* d)

该方法的具体实现和形参与 quantizeMatrix2DinCol()相同，只不过其返回值为重新量化后的 scale，而不对形参 XTensor 的 scale 进行改变。使用这个方法，如果接下来的计算不需要沿着矩阵的列进行量化的话，就无须对张量重新沿着行进行量化，减少计算代价。

2.2.3 反量化实现

（1）void dequantize(XTensor* input)

该方法与 quantize 方法正好相反，是将张量还原为浮点数，其参数为 XTensor 类型的指针。通过公式 4.3 计算好浮点类型的数据后，再使用类型转换函数将 XTensor 的数据类型由 X_INT8 转换为 X_FLOAT。

（2）void dequantizeMatrix2DinCol(XTensor* input)

该方法是当输入 XTensor 为 2 位张量时，将其沿着列的方向进行反量化。该函数专门由于给用列进行量化的张量进行反量化。其具体实现方法与 dequantize()类似。

2.2.4 重量化实现

重量化是当进行运算后，运算结果发生溢出情况时使用的操作。为了避免精度损失，对于张量加法、点乘、乘法等大概率会发生溢出的操作，通常都会先使用一个较大的数据类型数组来保存计算结果，比如 X_INT 和 X_FLOAT。如果发生溢出，再对其进行重量化操作。系统中重量化的实现为 void requantize(XTensor* input,

float* data)，其输入为 XTensor 指针和计算后存储结果的 X_INT 或 X_FLOAT 数组。对其使用公式 4.4 进行重量化计算后再重新为 XTensor 的元素和 scale 元素赋值。

表 5.3 介绍了量化部分的主要方法实现。

2.3 量化操作实现

本小节主要介绍关于张量计算的量化方法的实现。

2.3.1 张量代数计算

（1）void matchscale(XTensor* a, XTensor* b)

该方法为 scale 校准操作的实现，a 和 b 分别为两个需要校准的张量，先对其 scale 进行校准，再对数据进行校准。

（2）void Sum(const XTensor &a, const XTensor &b，XTensor &c，DTYPE beta=(DTYPE)1.0)

该函数为张量的加法。其中 a 和 b 为两个加数，c 为和。beta 为缩放系数。在计算前如果检测到 a 和 b 的数据类型都为 X_INT8，会对其 scale 进行加权求和，并赋值给 c 的 scale。先使用 X_INT 类型的数组对 a 和 b 求和的结果进行存储，如果 a 和 b 求和后发生了溢出，会对 c 进行重量化。

表 2.3 量化主要方法表

函数名称	函数功能
quantize(XTensor* input)	量化
void quantizeMatrix2DinCol(XTensor* input)	沿着 2 维张量的列进行量化
float* quantizeMatrix2DinColtoScale(XTensor* input, float* d)	沿着 2 维张量的列进行量化并返回 scale 数据
void dequantize(XTensor* input)	反量化
void dequantizeMatrix2DinCol(XTensor* input)	沿着 2 维张量的列进行反量化
void requantize(XTensor* input, float* data)	重量化

（3）void Sub(const XTensor &a, const XTensor &b, XTensor &c, DTYPE beta = (DTYPE)1.0)

该函数为张量的减法，其中 a 和 b 为两个减数，c 为差。其计算过程与张量的加法相同，只是将加法替换为减法。

（4）void Multiply(const XTensor &a, const XTensor &b, XTensor &c, DTYPE alpha = 0.0, int leadingDim = 0)

该函数为张量的点乘。其中 a 和 b 为两个乘数，c 为乘积。在依次对 a 和 b 的元素进行相乘之后，也用 X_INT 的类型数组对其进行存储，并对 a 和 b 的 scale 也进行乘法运算，赋值给 c 的 scale。如果发生溢出，则进行重量化。

（5）void Div(const XTensor &a, const XTensor &b, XTensor &c, DTYPE alpha = 0.0, int leadingDim = 0)

该函数为张量的除法。其计算方式同点乘类似。其中 a 和 b 分别为除数和被除数，c 为商。由于是除法，不会产生溢出，所以可以直接将计算结果存入 c 中，无需进行重量化。

（6）void MatrixMul(const XTensor &a, MATRIX_TRANS_TYPE transposedA, const XTensor &b, MATRIX_TRANS_TYPE transposedB, XTensor &c,DTYPE alpha = (DTYPE)1.0, DTYPE beta = 0, XPRunner * parallelRunner = NULL)

该函数为矩阵乘法，a 和 b 为两个输入矩阵。transposedA 和 transposedB 分别决定 a 和 b 在进行乘法时是否进行转置。a 和 b 的转置情况决定了是否将 a 和 b 按照列进行量化。在对 a 和 b 的矩阵相乘之后，对 a 和 b 的 scale 也进行矩阵相乘，再使用 dequantize2D()和 quantize()进行反量化和量化。

（7）XTensor MatrixMulBatched(const XTensor &a, MATRIX_TRANS_TYPE transposedA, const XTensor &b, MATRIX_TRANS_TYPE transposedB,DTYPE alpha = (DTYPE)1.0, XPRunner * parallelRunner = NULL)

矩阵 Batch 的实现方法与矩阵乘法类似。其内部实现调用了 MatrixMul 函数。在处理 scale 的时候，应该按照 batch 的大小对 scale 依次取出进行计算。

（8）XTensor MulAndShift(const XTensor &x, MATRIX_TRANS_TYPE transposedX, const XTensor &w, MATRIX_TRANS_TYPE transposedW, const XTensor &b, DTYPE alpha = (DTYPE)1.0, XPRunner * parallelRunner = NULL)

该函数为线性变化的实现。在其内部调用了矩阵乘法和矩阵加法的实现，所以对 scale 的操作已经包含在矩阵乘法和矩阵加法的实现中，无需进行其他操作。但是要注意在乘法运行后会有一个临时变量来存储乘法的结果，在实现的时候应该注意 scale 值的复制。

2.3.2 张量存取

（1）XTensor ConvertDataType(const XTensor & input, TENSOR_DATA_TYPE dataType)

该函数为张量类型转化的实现方法。对于 INT_8 类型的变量，将其类型转化为其他数据类型的时候，要及时释放掉 scale 中储存值的数组内存，防止内存泄漏，或者是以后对其量化的时候无法重新对其进行分配。

（2）void XTensor::SetDataInt8(const void* d, int num, int beg)

该函数用来给数据类型为 X_IN8 的张量赋值。由于在乘法计算后使用 32 位 int 数组来存放结果，如果结果无需量化，将 32 位整型值赋值给 8 位整型值即可。由于在内存中，整型数的 32 位地位字节在内存的前面存放，所以直接读取每个数的第一个字节作为 8 位整型的数值即可。

2.3.3 数学运算

（1）XTensor LN::Make_int8(XTensor& input)

该函数为 Transformer 子层中标准化操作的具体实现。其输入和返回值都为 XTensor 张量。由第四章的系统设计可知，本课题使用 L1 标准化来替代传统的标准化。其内部使用 ReduceMeanandScale_int8()来计算均值，同时这个函数也可以被用来计算 1 范数，只需要将其参数进行特殊设计即可。在 L1 标准化中的所有张量代数计算都使用了量化后的操作。

（2）XTensor ScaleAndShift(const XTensor &a, DTYPE scale, DTYPE shift, bool inplace)

该函数为缩放平移的实现，其内部实现首先调用 scale()进行缩放，使用临时张量 tmp 对其结果进行存储，然后调用了 sum()，对 scale 的具体操作都包含在 scale()和 sum()方法中，只需要在计算过程中对 tmp 的 scale 进行初始化和传递即可。

2.3.4 规约操作

（1）XTensor ReduceMeanandScale_int8(const XTensor& input, int dim, XTensor* shift, DTYPE power)

该函数是规约操作的实现。通过对 power 和 shift 的设置可以实现不同的功能。如 power=1，shift 为 NULL 时，可以求其均值。当 power=3，shift 为均值张量时，可以求其 1 范式，也就是应用于标准化层中的分母。

2.3.5 形状转换

（1）XTensor Concatenate(const XTensor &smallA, const XTensor &smallB, int dim)

该函数为级联操作的实现，smallA 和 smallB 为两个要合并的张量，dim 为合并的维度。当检测到输入张量的类型为 X_INT8 后，分别对其 scale 进行复制，复制到结果张量的 scale 数组中。根据合并维度的不同设置每次复制的步长和大小。

（2）XTensor Merge(const XTensor &s, int whereToMerge, int leadingDim)

该函数为合并操作的实现，whereToMerge 为合并的维度，leadingDim 为被合并掉的维度。如果检测到输入张量的类型为 X_INT8，将其 scale 按照新的形状复制到结果张量中即可。

（3）void Split(const XTensor &s, XTensor &t, int whereToSplit, int splitNum);

该函数为切分操作的实现，whereToSplit 为切分的 ，splitNum 为切分的份数。检测到输入张量为 X_INT8 类型后，也是将其 scale 按照新的形状复制到结果张量中。

（4）void Unsqueeze(const XTensor &a, XTensor &b, int dim, int dSize)

该函数为张量扩展操作的实现。a 为输入张量，b 为结果张量，dim 为插入维度，dSize 为插入维度大小，如果检测到张量为 X_INT8 类型，也将其 scale 按照新的形状进行扩充和复制。

2.3.6 激活函数

（1）XTensor Rectify(const XTensor &x)

该函数为 ReLU 激活函数的实现，由系统设计中的分析可知，该函数无需对其 scale 进行操作。

2.4 量化存取实现

2.4.1 参数加载实现

（1）void Model::ReadandConvert(FILE* file)

file 为存储模型参数的文件，将其按照参数列表进行读取吼，再调用 quantize() 直接将其量化。

2.4.2 参数存储实现

（1）void Dump(const char* fn)

该函数为存储参数的实现。其中 fn 为要存储文件的名字。首先在文件开头存储模型信息，其次按照 8 位整型存储模型参数，参数包括每个张量的数据和其对应的 scale。

2.5 量化整体实现

量化的整体实现流程图如图 5.1 所示。

2.6 技术难点

本课题首次将 8 位整型量化方法应用于 NiuTensor 框架中。NiuTensor 是东北大学自然语言处理实验室开发的深度学习框架，由 C++编写。具有高性能，轻量化等优点。本课题在此基础上，针对该框架的数据结构进行了量化系统的设计与实现。在实现的过程中，主要遇到了三个难点。

（1）代码理解。要实现对张量计算和模型的正确量化，就要对 NiuTensor 框架其内部实现有深入的理解。对于 NiuTensor 框架来说，需要明白是用何种数据结构来实现张量的存储的，其实现都包含那些成员变量以及成员函数，都对应什么功能。同时该框架是如何实现张量计算的，如何对张量内部的数据进行操作，存储和传递。对于 Transformer 模型来说，需要明白它是如何进行推理的，在推理过程中应用到了哪些张量计算，哪些张量计算需要进行量化，哪些不需要进行量化，哪些无法进行量化。这些都是在进行系统实现时必要的步骤与重难点。

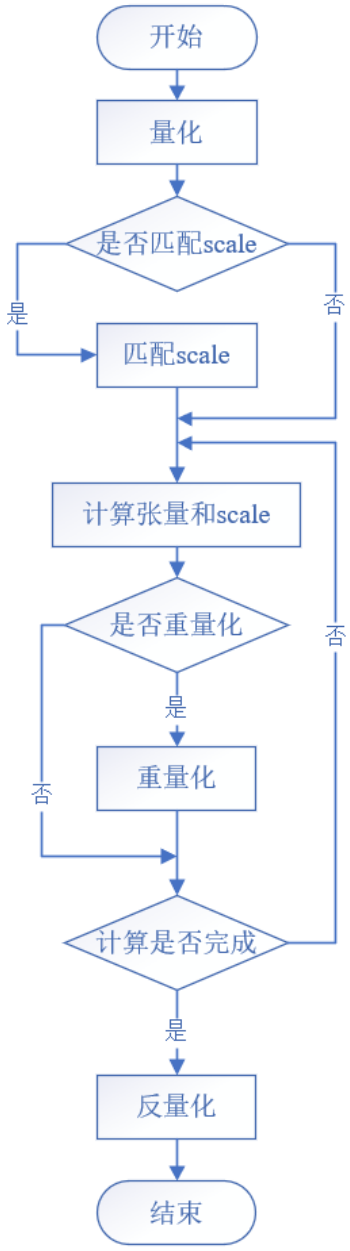

图 2.2 量化整体实现流程图

（2）代码实现。在具体实现的时候，由于对整个框架使用最多，最基础的类 XTensor 进行了改变，对其添加了新的成员变量 Scale，所以在量化过程中对代码底层的改动还是相当大的。同时要对每一个 Transformer 涉及到的运算都添加对 scale 的传播操作，且不同的运算对应的 scale 传播方法不同，在实现的时候进行了各种尝试和优化。因此代码的实现也是整个系统设计的重难点之一。

（3）代码维护。由于 Transformer 的模型设计的计算操作特别多，在计算时应该注意数据的传递和存储，不能出现数据丢失或者内存泄漏的情况。要对构造函数、

析构函数、拷贝函数、等号赋值以及数据清空等操作进行详细的修改和排查，以免程序出现内存漏洞。同时也要将无用的内存空间及时释放，管理指针的正确性，避免出现野指针、悬空指针等情况。

3 系统测试

3.1 正确性测试

本小节首先对 Transformer 推理过程中使用率较高的几个张量计算进行测试，其次对 LayerNorm 层进行测试，最后对系统的整体正确性进行测试。本次测试的输入张量为大小为（100,100）的张量，其取值为-1 到 1 的随机数。

表 3.1 量化主要计算误差表

函数名称	误差
quantize()	0.4%±0.4%
Sum()	2%±1%
Sub()	2%±1%
Multiply()	0.5%±0.5%
Div()	0.5%±0.5%
Scale()	1%±1%
MatrixMul()	3%±1%
MatrixMulBachted()	3%±1%
Split()	0.4%±0.4%

接下来使用 BLEU 分数来验证 LayerNorm 的正确性和整体的正确性。Bleu 通过将待测试翻译样本与参考样本进行比较，分数越高，相似度越高。其测试结果如表 6.2 所示，其中 baseline 是未经过量化的系统，使用 Transformer 模型，编码器和解码器的层数都为 6 层，词嵌入的维度大小为 512，读入句子的 batch 大小为 64。Layernorm 是标准化层经过量化操作的系统，Fully 是整体经过量化的系统。从表 6.2 中可以看出，只量化 Layernorm 的模型 bleu 值下降了 0.46，整体的 8 位整型量化系统下降了 0.89，虽然产生了一定的精度损失，但是正确性仍能保证。

表 3.2 BLEU 分数表

模型方案	Bleu 分数
Baseline	34.01
Layernorm	33.55
Fully	33.12

3.2 性能测试

在本节中，使用 IWSLT-14 DE-EN 测试集，将对量化后的系统和 baseline 进行存储、速度、运行内存方面的测试比较。其结果如表 3.3 所示。

表 3.3 性能测试表

模型方案	模型参数存储/MB	运行内存/GB	加速比
Baseline	179	2	-
Fully	45.2	0.78	×3

相对于 baseline，模型参数量化后接近原存储的四分之一，之所以没有严格达到四分之一的大小，是因为在文件中还存储了 Transformer 的层数信息，以及各个张量的 scale 值。同时运行内存也由 2GB 降低至 0.78G，推测是由于要存储程序的其他信息，以及张量的其他部分，所以运行内存没有将至其四分之一。对于整个系统的加速比，由于目前实验所能使用的 CPU 都缺乏支持 int8 加速运算的 VNNI 指令，所以按照每个操作会有六倍的速度收益来对系统整体的加速效果进行估计。但是在目前实验使用的 CPU 上，还是存在一定的速度收益，例如矩阵乘法，如图 6.1 所示，可以发现，随着矩阵变大，速度的提升越大。

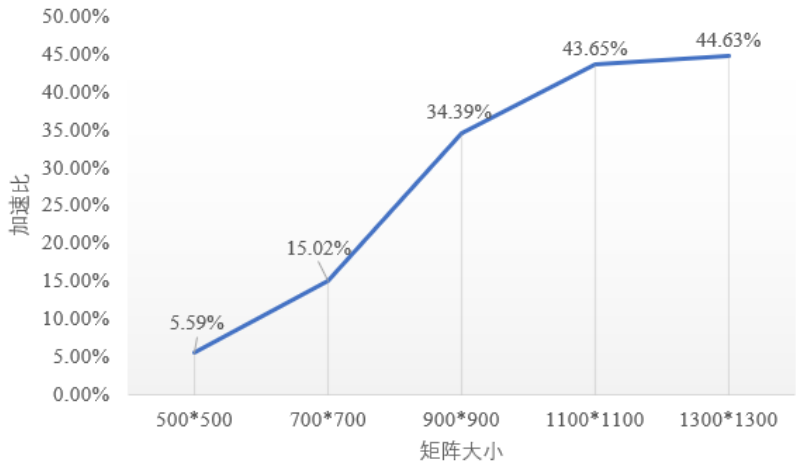

图 3.1 矩阵大小与其矩阵乘法加速比折线图

对于矩阵 Batch 乘法，量化后的操作在本地也获得了一定的加速收益。如图 3.2 所示。可以发现，加速收益也是随着矩阵增大而增大。但是对比矩阵乘法可以发现，其维度较低时，加速收益没有矩阵乘法那么多，推测是因为矩阵 Batch 乘法在运算时要对张量的 scale 进行切割操作，添加了计算成本。

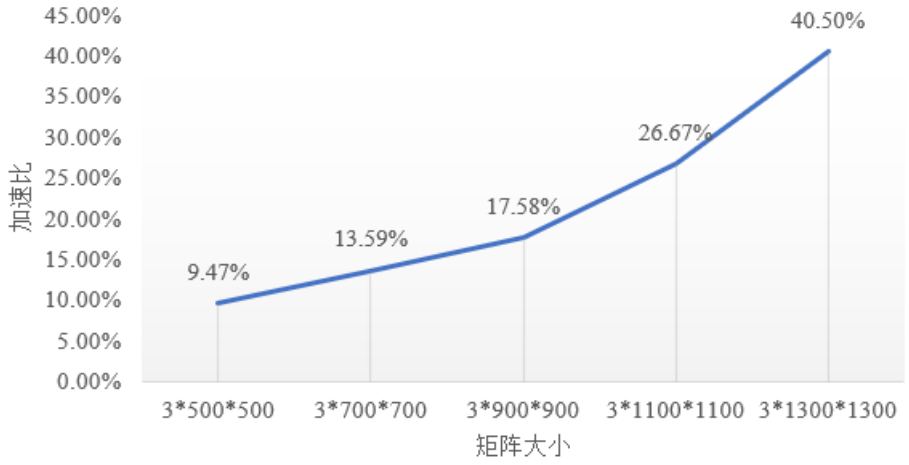

图 3.2 矩阵大小与其矩阵 Batch 乘法加速比折线图

对于张量的缩放操作，量化后带来的速度收益要比较明显，如图 3.3 所示。分析是因为该操作只对 scale 进行计算，相当于减少了一个维度的计算量，并且无需对数据进行重量化，所以收益较大。

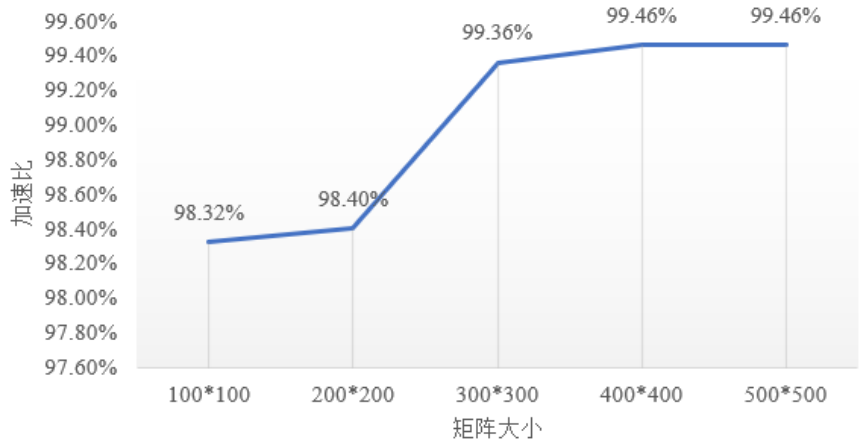

图 3.2 张量大小与其缩放操作加速比折线图

3.3 本章小结

本章从系统的正确性以及性能等方面对模型进行了测试，证实了 8 位量化系统的正确性。在性能方面，无论是模型参数存储还是程序运行内存，8 量化系统都有明显的收益。但是由于硬件的限制，无法测量系统整体的实际速度，只能对其进行理论估计。同时在本地的 CPU 上，对于矩阵乘法，虽然效果不是很明显，还是有速度收益的，随着矩阵维度变大，收益也随之增加。

4 总结和展望

4.1 总结

（1）

通过查阅相关文献、书籍，对现有的量化方法进行了学习和总结。同时也对深度神经网络、机器翻译、Transformer 模型进行了学习，了解的 Transformer 的具体架构。同时由于本课题是基于 NiuTensor 进行实现的，所以也对 NiuTensor 开源框架进行了代码阅读和实验。同时对 Transformer 在 NiuTensor 上的具体实现也进行了学习。在了解完相关知识后，对量化系统进行需求分析说，分析其所要实现的整体系统效果，以及包括量化方法、量化操作、量化存取在内的各个功能模块。最后对课题从经济和技术两方面进行了可行性分析，保证课题的顺利完成。

（2）

进行方法以及系统的设计和实现工作。首先设计了系统的量化方法，包括量化、反量化和重量化。量化是将张量由 32 位浮点数转换为 8 位整型，反量化是将张量由 8 位整型转换为浮点数。重量化是将溢出的数据重新量化至 8 位整型的范围内。其次对量化操作的设计。量化操作主要包括张量的代数运算，数据存取，数学运算，规约操作，形状转换，激活函数。通过对量化操作的设计和实现，使得张量在计算的同时，也能对 scale 进行传播，避免在各个运算之间频繁地使用量化和反量化操作，造成额外的计算代价。然后对张量的存取操作进行设计，使得系统能够读取浮点数参数并将其转换为 8 位整型，以及将 8 位整型的参数写入模型文件。最后对系统的整体架构以及流程进行设计和实现。

（3）

主要进行系统的完善和测试工作。首先将代码进行进一步的优化，使其具有可靠性、可移植性、可维护性，从可读性以及逻辑等方面对系统进行完善。然后对系统进行正确性以及性能的测试。对于正确性，本课题对先分模块验证了各个操作的正确性，再从整体方面使用 BLEU 值验证系统的正确性。对于系统的性能，本课题从运行速度、运行内存、模型存储三个方面对其进行测试。虽然运行速度由于硬件问题无法进行实际测算，只对其进行了理论估计，但是对于某些运算，例如矩阵乘法，在本地 CPU 上也有高达 40%的速度增益。

4.2 展望

虽然本课题实现的系统已经达到了预期的效果，但是还有可以进行改进的地方。比如本文只是针对 Transformer 模型进行的量化系统实现。可以将其设计得更具有普适性，对于其他深度神经网络模型也能实现 8 位整型量化。同时，本系统只是在 CPU 上进行了实现，下一步可以将其扩展至 GPU 上，进一步提高量化系统的性能和应用范围。

5.致谢

很高兴可以顺利完成本次机器翻译课程的实验，我深知这一切都离不开肖桐老师的谆谆教诲，肖老师亦师亦友，严谨的学术态度幽默的授课风格有机地融为了一体，为我们营造了生动活泼的良好学习氛围。

特别感谢实验室的胡驰学长，每次都用心准备课堂上的实验，让我们能够更好地理解当天学习的知识并增加实操经验，同时也耐心解决我们的问题。经历了本课程的学习，我受益匪浅。在未来我也会秉持探索求真的态度，脚踏实地认真学习。