

2023 年 CCFSys 定制计算挑战赛
CCFSys Customized Computing Challenge 2023
FFT 设计报告

参赛学校：华东师范大学
指导教师：徐飞老师
参赛队员：白卓岩、堵仪萱、徐珑珊

2023 年 7 月 8 日

目录

1 概述	1
1.1 选题	1
1.2 完成情况	1
2 算法设计	1
2.1 非递归 FFT	1
2.1.1 FFT 基本原理	1
2.1.2 蝴蝶操作	2
2.2 分布式 FFT	2
3 具体实现	3
3.1 AIE	3
3.1.1 1K-point FFT	3
3.1.2 Stage-2 FFT	4
3.1.3 Graph 连接	5
3.1.4 Array 图片	6
3.2 PL	8
3.2.1 PL 和 AIE 的数据传输	8
3.2.2 host 调用	8
3.2.3 运行结果	8
4 实验结果	8
5 总结	9
参考文献	10

1 概述

1.1 选题

1.2 完成情况

赛题的完成情况如下。

- 根据对 FFT 和 AIE 的理解，使用 AIE API 完成了对 1024 点小规模 FFT 算法的单核设计、优化和仿真；
- 探索 AIE kernel 之间的连接，扩展更大点数的 FFT 实现；
- 实现 PL 和 AIE 的数据连接，可以在 VCK5000 硬件上运行。

表 1: 性能矩阵

点数	AIE 数量	GSPS	数据类型	twiddle 类型	Power	GSPW
4096	5		cint16	cint16		

2 算法设计

傅里叶变换（Fourier transform, FT）是一种重要的数字信号处理技术。傅里叶变换可以将时域信号转换至频域，以便查看不同频率下信号的强度，实现去除低频、高频噪声信号等功能，在音频、图像、雷达、通信等领域中都有着广泛的应用。

快速傅里叶变换（fast Fourier transform, FFT）是一种优化算法，可以高效计算傅里叶变换，将计算复杂度从 $O(n^2)$ 降低到 $O(n \log n)$ 。FFT 在具体实现上可以采用不同的优化方案，例如使用分治算法将一个大规模的傅里叶变换分解成若干个小规模的傅里叶变换进行并行计算，这使得 FFT 在实际应用中可以高效地实现。

本节介绍本项目所采用的 FFT 算法设计。第 2.1 小节介绍 FFT 的非递归实现方法，用于计算基础的 1024 点的 FFT。第 2.2 小节介绍分布式的 FFT 实现方法，侧重于点数规模的扩展。

2.1 非递归 FFT

基于递归实现的 FFT 是较为基础的版本，但递归过程开销较大，本项目采用非递归的方式来实现。

2.1.1 FFT 基本原理

傅里叶变换的基本计算公式为：

$$Z_k = \sum_{n=0}^{N-1} \omega_N^{nk} x_n, \quad k = 0, 1, \dots, N.$$

其中, N 为采样点数, ω_N^{nk} 叫做旋转因子 (twiddle factor), 可看作频域下的基。该式子的作用就是通过线性变换, 将时域下的采样点转换至频域下。由于傅里叶变换所做的是线性变换, 其公式也可以写成矩阵向量乘法的形式, 即:

$$\begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ Z_N \end{bmatrix} = \begin{bmatrix} \omega_N^0 & \omega_N^0 & \cdots & \omega_N^0 \\ \omega_N^0 & \omega_N^1 & \cdots & \omega_N^{N_2-1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_N^0 & \omega_N^{N_1-1} & \cdots & \omega_N^{(N_1-1)(N_2-1)} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}.$$

可以看出该矩阵向量乘法的计算复杂度为 $O(n^2)$ 。

FFT 利用了两个的技巧, 将傅里叶变换的复杂度降低为 $O(n \log n)$ 。

技巧 1. twiddle factors 均匀分布在复平面单位圆上。如图 1 所示, 当 $N = 8$ 时,

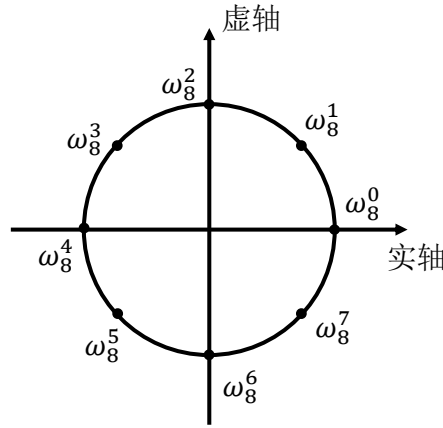


图 1: 复平面单位圆

技巧 2. Z_k 可以分成奇偶两部分。

即:

$$A(x) = (a_0 + a_2x^2)$$

2.1.2 蝴蝶操作

2.2 分布式 FFT

分布式 FFT 采用 Cooley-Tukey 技巧将 N -point FFT 分解为 N_1 -point 和 N_2 -point FFT, 则 FFT 的计算公式变为:

$$Z_k = \sum_{n=0}^{N-1} \omega_N^{nk} x_n = \sum_{n_1=0}^{N_1-1} \omega_{N_1}^{n_1 k_1} t_{n_1, k_2} \sum_{n_2=0}^{N_2-1} \omega_{N_2}^{n_2 k_2} x_{n_2 N_1 + n_1}$$

其中 $N = N_1 \times N_2$, $k = k_1 N_2 + k_2$, $k_1 = 0, \dots, N_1-1$, $k_2 = 0, \dots, N_2-1$, t_{n_1, k_2} 是 twiddle factor, $t_{n_1, k_2} = \omega_N^{k_2 n_1}$ 。通过这个变换, N_2 -point FFT 可在 N_1 个处理器中并行计算, 最后再将这 N_1 组结果进行 N_1 -point FFT 的计算得出最终结果。具体步骤如下:

1) 将输入信号 X 分为 $X_1^{(\text{row})}$ 到 $X_{N_1}^{(\text{row})}$, 分别传入 N_1 个 N_2 -point FFT:

$$X = \begin{bmatrix} x_1 & x_{N_1+1} & \cdots & x_{(N_2-1)N_1+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N_1} & x_{2N_1} & \cdots & x_{N_1N_2} \end{bmatrix} = \begin{bmatrix} X_1^{(\text{row})} \\ \vdots \\ X_{N_1}^{(\text{row})} \end{bmatrix}$$

2) 在每个处理器中进行 N_2 -point FFT 的计算得到 $Y_i^{(\text{row})} = X_i^{(\text{row})} F_{N_2}$, $i = 1, \dots, N_1$

3) 在每个处理器中将 Y_i 乘以 twiddle factor: $Y_i^{(\text{row})} = T_{N,i}^{(\text{row})} \circ Y_i^{(\text{row})}$, 其中 \circ 表示对应元素相乘,

$$T_N = \begin{bmatrix} \omega_N^0 & \omega_N^0 & \cdots & \omega_N^0 \\ \omega_N^0 & \omega_N^1 & \cdots & \omega_N^{N_2-1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_N^0 & \omega_N^{N_1-1} & \cdots & \omega_N^{(N_1-1)(N_2-1)} \end{bmatrix} = \begin{bmatrix} T_{N,1}^{(\text{row})} \\ \vdots \\ T_{N,N_1}^{(\text{row})} \end{bmatrix}$$

4) 在第 $N_1 + 1$ 个处理器中计算出最终结果 Z :

$$Y = \begin{bmatrix} Y_1^{(\text{row})} \\ \vdots \\ Y_{N_1}^{(\text{row})} \end{bmatrix} = \begin{bmatrix} Y_1^{(\text{col})} & \cdots & Y_{N_2}^{(\text{col})} \end{bmatrix}$$

$$Z_i^{(\text{col})} = F_{N_1} Y_i^{(\text{col})}, i = 1 \dots N_2$$

3 具体实现

本项目的核心在于优化 AIE 上 FFT 的设计和实现, 这一部分将在第 3.1 小节介绍。另外, 也尝试了连接 PL 和 AIE, 从 host 端进行调用, 在 VCK5000 上运行任务, 见第 3.2 小节。

3.1 AIE

3.1.1 1K-point FFT

1K-point FFT 为一个基本的计算 kernel, 输入和输出都采用 buffer 的方式。该 kernel 首先需要对输入信号进行位置交换, 然后用矩阵乘法完成 8-point FFT 的计算, 再进行 16-point 到 1024-point 的 butterfly 操作, 共 7 次, 最后将结果乘以对应的 twiddle factor。为了提高访存效率, 该 kernel 将输入和输出 buffer 作为 ping-pong buffer 的方式, 以进行上述每一次操作计算结果的读入和写出。

3.1.1.1 Shuffle 本项目的 1K-point FFT 计算采用非递归的 FFT 算法实现，从 8-point FFT 向上迭代。该方法首先需要将输入信号的顺序进行交换，以便后续的迭代计算。为了完成交换步骤，首先要利用递归实现的 FFT 迭代到 8-point 来获取 8-point FFT 开始时各输入值的位置，然后确定如何把输入信号通过交换转变为 8-point FFT 开始时信号序列，需要交换位置的信号值可以分为多个最小的序列组，每组分别进行交换即可完成整个序列的交换。例如第 1 个值最终交换到第 512 个，第 512 个到第 4 个，第 4 个到第 128 个，第 128 个到第 1 个，则第 1、512、4 和 128 个值可分为一个长度为 4 的组进行交换。经过分析，4096 个输入值一共分为 120 个长度为 2 的组和 384 个长度为 4 的组。在代码实现方面，可以通过两个循环实现长度为 2 和 4 的序列组的交换，同时由于每个组的数据中间没有重复，因此每个迭代间也没有数据依赖，可通过 unroll 来提高循环的并行度。

3.1.1.2 Butterfly 在传统快速傅立叶变换的实现中，根据傅立叶变换的特殊性质，可将长度为 N 的离散傅立叶变换 (discrete Fourier transform, DFT) 递归地表述为两个大小为 $N/2$ 的离散傅立叶变换，直至 N 等于 2。通过重用中间计算结果，两个大小为 $N/2$ 的傅立叶变换结果可以同时得到。从而将傅立叶变换的计算复杂度由 $O(n^2)$ 降低为 $O(n \log n)$ 。

在本实现中，为避免大量的函数调用及存储开销，我们将递归操作修改为非递归实现，同时以 8-point FFT 为底开始运算。即首先采用传统傅立叶计算方式（矩阵运算）计算 8-point FFT，再在此计算结果上进行多次蝶形运算的迭代。虽然表面上看 8-point FFT 的计算复杂度有所提升，但将标量运算转化为矢量计算，能够更好地利用 AIE 的单指令多数据流 (Single Instruction/Multiple Data, SIMD) 特性，实现对各个数据元素的并行处理，从而大大提升处理速率。同时，选择 8-point 为底具有可接受的存储开销及时间复杂度。

在进行多次蝶形运算的迭代过程中，为充分利用 AIE 的并行计算能力，我们将尽可能多地载入矢量数据进行运算（数据类型为 `cint16` 时矢量大小至多为 32）。此时数据的读入与读出将成为主要的性能瓶颈，为避免读入数据的地址空间不连续所导致的读入耗时过长，我们以可接受的存储冗余开销换取读入时间的大幅减少。具体而言，我们分别存储了 16-256 以及 1024 point 下所需的 twiddle factor 数组，从而使得蝶形运算过程的访存和计算过程达到相对平衡。

同时，在该过程中我们借鉴了 Ping-Pong Buffer 的思路，主要考虑避免在一个循环过程中，数据的读取与写入在同一块地址空间中进行，以避免数据冲突和处理延迟。为此，我们利用了两个独立的 buffer，它们在操作过程中会轮流切换角色，即在一个时间片段内，一个 buffer 负责数据的读取，而另一个 buffer 则负责数据的写入。在下一个时间片段，它们的角色互换。该策略大大提高了数据处理的效率。

3.1.1.3 乘操作 在前四个 kernel 分别完成 1024-point FFT 后，需将所得结果分别与对应的 twiddle factor 相乘，之后由最后一个 kernel 汇总所有数据并计算 1024 个 4-point FFT。同样，在此过程中为充分利用 AIE 的并行计算能力，我们也尽可能多地载入矢量数据进行运算，使得 32 个数据同时进行乘操作。

3.1.2 Stage-2 FFT

AIE 提供硬件支持来加速一种称为 sliding multiplication 的多通道乘法。它允许多通道并行进行乘累加 (multiplyaccumulate, MAC) 操作，并将结果添加到累加器中。在该阶段我们利用 AIE 的这一特性进

行 1024 个 4-point FFT 运算。参数配置及计算过程如下图所示。

$$\text{auto acc_buff} = \text{aie::sliding_mul_ops} \langle 8, 4, 1, 8, 1, \text{cint16}, \text{cint16}, \text{cacc48} \rangle :: \text{mul}(\text{coeff_buff}, 0, \text{data_buff}, 0);$$

CoeffStep DataStepX
 Lanes Points DataStepY

图 2: sliding multiplication 参数配置

	d ₀	d ₁	d ₂	d ₃	d ₄	d ₅	d ₆	d ₇	d ₈	d ₉	d ₁₀	d ₁₁	d ₁₂	d ₁₃	d ₁₄	d ₁₅	d ₁₆	d ₁₇	d ₁₈	d ₁₉	d ₂₀	d ₂₁	d ₂₂	d ₂₃	d ₂₄	d ₂₅	d ₂₆	d ₂₇	d ₂₈	d ₂₉	d ₃₀	d ₃₁	
y ₀	c ₀								c ₁								c ₂								c ₃								
y ₁		c ₀								c ₁								c ₂								c ₃							
y ₂			c ₀								c ₁								c ₂									c ₃					
y ₄				c ₀								c ₁								c ₂									c ₃				
y ₅					c ₀								c ₁								c ₂									c ₃			
y ₆						c ₀								c ₁								c ₂									c ₃		
y ₇							c ₀								c ₁								c ₂									c ₃	
y ₈								c ₀								c ₁									c ₂								c ₃

图 3: sliding multiplication 计算过程

由计算过程可知，一次 sliding multiplication 操作可完成 32 次乘累加操作，并得到 8 个输出作为 4096-point FFT 的最终结果。在后续操作中，只需更新四次系数 (coefficient) 值，即可得到 32 个输出结果。不断重复操作（循环 128 次），即可得到所有输出结果。经检验，该操作可大大降低运算时长。

3.1.3 Graph 连接

本项目一共使用 5 个 kernel 实现 4K-point FFT，其中 4 个 kernel 用于 1K-point FFT 以及与 twiddle factor 相乘的计算，1 个 kernel 用于 1024 个 4-point FFT 的计算。如图4，四个计算 1K-Point FFT 的 kernel 分别为 fft0、fft1、fft2 和 fft3，都通过 buffer 进行输入输出数据的传输，每组输入数据是从原长度为 4096 的输入信号中每隔 4 个取一个组成。当这 4 个 kernel 将计算结果都写入输出 buffer 后，用于计算 1024 个 4-point FFT 的 kernel fft_stage2 开始进行计算，并将以 4 个长度为 1024 的结果作为输出，这 4 个输出按顺序拼接即可得出长度为 4096 的结果。

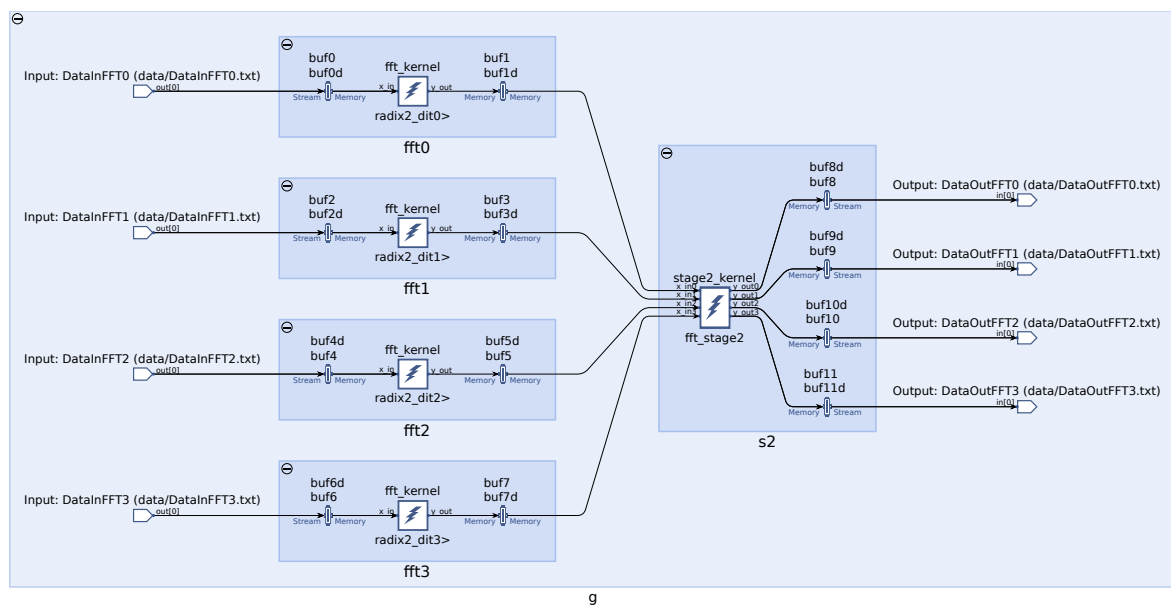


图 4: 4K-point FFT Graph

3.1.4 Array 图片

换图片。。。

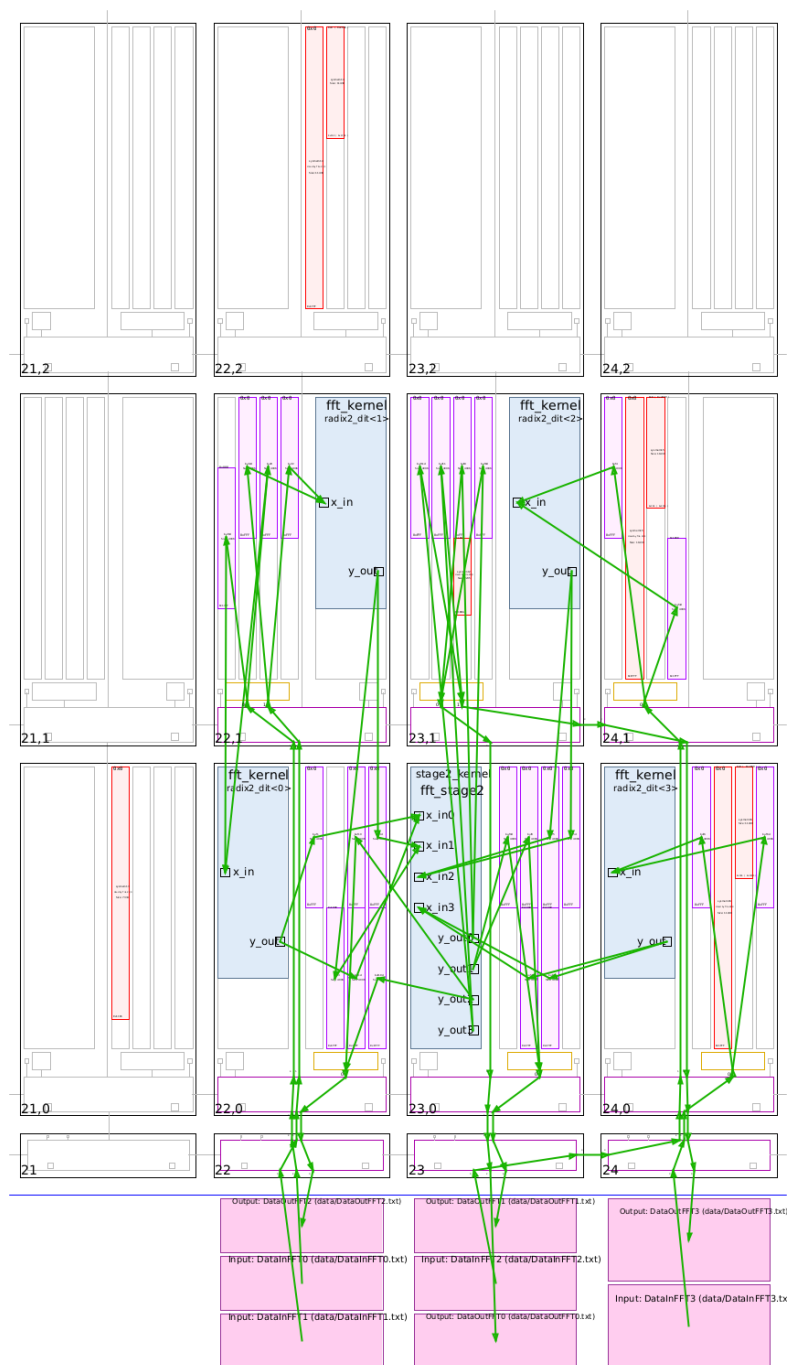


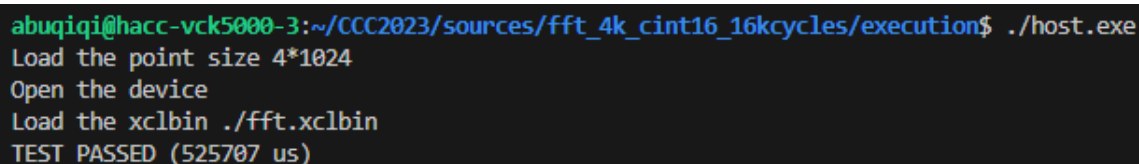
图 5: 4K-pint FFT Array 布局

3.2 PL

3.2.1 PL 和 AIE 的数据传输

3.2.2 host 调用

3.2.3 运行结果



```
abuqiqi@hacc-vck5000-3:~/CCC2023/sources/fft_4k_cint16_16kcycles/execution$ ./host.exe
Load the point size 4*1024
Open the device
Load the xclbin ./fft.xclbin
TEST PASSED (525707 us)
```

图 6: 主机端输出结果

核心代码

1. CNN (text_analysis/models.py)

CNN 模块建立神经网络结构，原代码中使用了一个嵌入层，3 组卷积层 + 池化层，Dropout 层防止过拟合，全连接层投影成文档隐向量。仿照原代码实现的 CNN 模块如下（只有 1 组卷积 + 池化层）：

```
1 def train(self, x_train, y_train):
2     history = self.model.fit(x_train, y_train, epochs=10,
3                             batch_size=32, validation_split=0.2)
4     return history
```

4 实验结果

由于时间缘故，有些模型来不及进行实验或者由于代码版本不同导致实验无法进行（如 NARRE、TAR-MF、MPCN 由于 python 版本要求 2.X），因此最后仅对 NRPA 模型展开了实验。

代码来源

<https://github.com/microsoft>

运行结果

```
1086it [14:27, 1.25it/s]
8874it [14:39, 10.08it/s]

at epoch 1
train info: logloss loss:1.5243750812818553
eval info: group_auc:0.5767, mean_mrr:0.2513, ndcg@10:0.3362, ndcg@5:0.2698
at epoch 1, train time: 867.4 eval time: 887.4

1086it [14:31, 1.25it/s]
8874it [14:34, 10.14it/s]

at epoch 2
train info: logloss loss:1.411583351430312
eval info: group_auc:0.5927, mean_mrr:0.2637, ndcg@10:0.352, ndcg@5:0.2883
at epoch 2, train time: 871.9 eval time: 882.2

1086it [14:56, 1.21it/s]
8874it [14:51, 9.95it/s]

at epoch 3
train info: logloss loss:1.35070926851111
eval info: group_auc:0.5893, mean_mrr:0.2664, ndcg@10:0.3517, ndcg@5:0.2867
at epoch 3, train time: 896.9 eval time: 901.0

1086it [15:19, 1.18it/s]
8874it [14:26, 10.25it/s]

at epoch 4
train info: logloss loss:1.3011221738041654
eval info: group_auc:0.5927, mean_mrr:0.2695, ndcg@10:0.3558, ndcg@5:0.2897
at epoch 4, train time: 920.0 eval time: 873.4

1086it [14:10, 1.28it/s]
8874it [14:16, 10.37it/s]

at epoch 5
train info: logloss loss:1.262577884105029
eval info: group_auc:0.5887, mean_mrr:0.2664, ndcg@10:0.3518, ndcg@5:0.2865
at epoch 5, train time: 850.4 eval time: 863.1
```

图 7: NPRA 运行结果

运行环境

- Anaconda3-5.2.0-Windows-x86_64
- Python 3.6
- Tensorflow 2.9.1

5 总结

KNN 分类器通过计算文本的距离，将文本聚类，从而得到不同的类别。相似度高的文本会被归到同一类别中。

基于内容的推荐优点：

- 挖掘评论中隐含的用户偏好和物品特征，缓解用户-物品交互稀疏性的问题；
- 增强系统的可解释性，能够展示出从用户评价中提取出的信息，比如偏好标签。

参考文献

- [1] Haewon Jeong, Tze Meng Low, and Pulkit Grover. Masterless coded computing: A fully-distributed coded FFT algorithm. In 2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton), pages 887–894. IEEE, 2018.
- [2] AI engine kernel and graph programming guide (UG1079). <https://docs.xilinx.com/r/en-US/ug1079-ai-engine-kernel-coding/>.
- [3] AI engine API user guide: Fast fourier transform (FFT). https://www.xilinx.com/htmldocs/xilinx2022_2/aiengine_api/aie_api/doc/.
- [4] AUP AI engine tutorial. https://xilinx.github.io/xup_aie_training/.
- [5] ccfsys-ccc/ccc2023. <https://github.com/ccfsys-ccc/ccc2023>.
- [6] Xtra-computing/hacc_demo. https://github.com/Xtra-Computing/hacc_demo.
- [7] VitisTM in-depth tutorials. <https://github.com/Xilinx/Vitis-Tutorials>.
- [8] XRT native library c++ API. https://xilinx.github.io/XRT/master/html/xrt_native.main.html.
- [9] Xilinx/xup_aie_training: Hands-on experience programming AI engines using vitis unified software platform. https://github.com/Xilinx/xup_aie_training.