基于 DTW 距离度量的 Shapelet 并行算法研究

（申请清华大学工程硕士专业学位论文）

培 养 单 位: 软 件 学 院

工 程 领 域: 软 件 工 程

申　请　人: 姜　友　友

指 导 教 师: 邓　仰　东　副　教　授

二〇一八年六月

Research on Parallel Shapelet Algorithm Based on Distance Measure of DTW

Thesis Submitted to

Tsinghua University

in partial fulfillment of the requirement

for the professional degree of

Master of Engineering

by

Jiang Youyou

(Software Engineering)

Thesis Supervisor : Associate Professor Deng Yangdong

June, 2018

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容。

本人保证遵守上述规定。

（保密的论文在解密后应遵守此规定）

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

摘 要

在信息时代，数据充斥着各领域并用于帮助我们进行分析和决策，人们也逐渐看到数据潜在的价值。而在大量数据之中，主要是以时间序列形式存在的。越来越多的人研究时间序列挖掘以期获取数据内在的价值，时间序列分类就是时间序列挖掘的重要分支之一。目前在时间序列分类领域，相似度/距离计算仍然占有主要地位，Shapelet 分类算法就是其中一种比较重要基于相似度/距离度量的时间序列分类算法。Shapelet 算法也有它的缺陷，训练过程比较消耗时间和资源。针对 Shapelet 分类准确率高而训练时间长这一问题，很多人提出了不少剪枝、降维等方法，但是时间消耗仍然是 Shapelet 算法的一个关键问题。

本文针对 Shapelet 发现过程耗时问题，提出了基于动态时间规整 (Dynamic Time Warping, DTW) 距离度量的 Shapelet GPU 并行算法。该算法主要从下面几个方面对于 Shapelet 运行时间进行了优化。

1. 提出 Shapelet 并行算法框架，可以针对较大数据集进行计算，针对不同的数据集大小、时间序列长度选择不同的计算方式，可以保证更高效的计算。

2. 在距离计算阶段，采用“重用”策略降低时间复杂度，并且使用统一计算设备架构 (Compute Unified Device Architecture, CUDA) 的一些优化技术如合并内存访问、归约技术、存储体冲突使有限的图形处理器资源上进行更多的计算。

3. 在计算最佳分割点，使用一种启发式计算最佳分割点的方法在保证准确率的基础上来减少计算信息增益需要的时间。

实验表明，本文提出的算法能够大幅提高 Shapelet 发现的并行速度，相比已有优化算法效率都有较大幅度提高，其中相比原始 CPU 算法提高 834 到 3692 倍，平均 1945 倍；相比基于剪枝的算法提高 21 倍到 51 倍之间，平均 41 倍；相比已有 GPU 并行算法提高 4 倍到 11 倍，平均 6.3 倍。同时，本文并行算法对于大数据集也进行了实验，并取得了较好的结果。

关键词：时间序列；GPU；CUDA；并行；距离

Abstract

In the information age, data is flooded in various fields and is used for analyzing to make some strategic decisions,thus its potential value is gradually discovered by people. Among these data,most are time series. Time series data mining has attracted more and more people to study for their intrinsic value. Classification of time series is one of the most important branches of time series data mining. At present, Similarity/distance calculation is dominating in the time series classification field, Shapelet is one of the important algorithms based on similarity/distance calculation.Shapelet is one of subsequences of time series that has an excellent discriminative power. However,the shapelet algorithm also has its drawbacks,that is, Lots of time and resources need to be consumed in the training process. Considering that Shapelet is excellent in high accuracy but is long in training, researchers suggested methods such as pruning, dimensionality reduction. Notwithstanding, time consumption is still a key issue in the Shapelet discovery.

To solve the time-consuming problem of Shapelet discovery, A parallel Shapelet algorithm based on distance measure of DTW(Dynamic Time Warping) has been proposed. Many works and optimizations are spent on the algorithm, including the following aspects:

1. A parallel algorithm framework of Shapelet discovery has been put forward, which can be applied to large datasets and can choose different methods according to the length of time series and the size of the dataset guaranteeing more efficient calculations.

2. At the distance calculation stage, the "reuse" strategy is used to reduce the time complexity, and CUDA's optimization techniques such as combined memory access, Reduced technology,and Bank-Conflict are used to take more avantage of limited graphics processor resources to do more calculations.

3. In the optimal split point calculation stage, a heuristic method used for calculating the optimal split point has been used, which significantly reduced the elapsed time on condition of achieving comparable classification accuracy.

Experiments show that the algorithm proposed in this paper can greatly increase the parallel speed of Shapelet discovery,which has a significant increasement in efficiency compared with the existing optimization algorithms. The proposed algorithm increased from 834x+ to 3692x+, with an average of 1945x+ compared to the CPU Naive algorithm.

The speedup of the algorithm from the pruning-based algorithm is between 21x+ and 51x+, 41x+ in average. the algorithm is faster than the existing GPU parallel algorithms by 4x+~11x+, 6.3x+ in average. At the same time, the parallel algorithm of this paper has also conducted experiments on large datasets and achieved excellent perfomance.

Key words: Time Series; GPU; CUDA; Parellel; Distance

目 录

主要符号对照表

D	时间序列数据集 (time series dataset), 包含多个时间序列-类标对 $(T_j, y_j, j = 1, 2, \cdots, N)$
T_j	D 数据集第 j 个位置的时间序列
y_j	D 数据集第 j 个位置的时间序列对应的类标
N	D 数据集大小
L	D 中时间序列 T_j 的长度
S	候选序列或时间子序列 (Subsequence)
$\mathcal{F}$	$(SubDist(S, T_j), y_j), j = 1, 2, \cdots, N$ 的集合
EDR	实数序列的编辑距离 (Edit Distance on Real Sequence)
DTW	动态时间归整 (Dynamic Time Warping)
w	$DTW(A, B, w)$ 距离中控制限制区域大小的参数
TSC	时间序列分类 (Time series classification)
GPU	图形处理器 (Graphics Processing Unit)
CUDA	统一计算设备架构 (Compute Unified Device Architecture)
SIMD	单指令多数据并行方式 (Single Instruction Multiple Data)
SM	流处理器簇 (Stream Multiprocessors)
SP	流处理器 (Stream Processor)
ARIMA	自回归移动平均模型 (Auto Regressive Integrated Moving Average Model)
DNN	深度神经网络 (Deep Neural Networks)
SOINN	增强自组织增量神经网络 (self-organizing incremental neural network)
Coalesced	合并内存访问 (Coalesced Memory Access)

第 1 章　引言

1.1　研究背景和意义

在信息时代，数据已经渗透到商业、经济和其他领域，并且逐渐取代经验和直觉成为支持和影响决策的主要因素之一[1]。互联网、金融、工业、医药等领域都充斥着大量数据[2]，利用这些海量数据进行分析将会给各行各业带来商业模式的转化和传统工作方法的革新。而在这些海量数据中主要是的存在形式是时间序列，比如复杂工业装备成千上万的传感器数据[3]，金融领域的股票期货价格、通胀率[4]，医学领域的心电图[5]，制造行业的传感器连续测量的数据流记录都是时间序列应用的特定示例。越来越多的学者和专业人士想通过数据挖掘的方法分析时间序列数据，目的是通过发现时间序列内在知识和信息来帮助人们更好地做出决策和提前预知风险等。数据挖掘逐步应用于时间序列，并且在部分领域（例如金融、工业）取得了一定的成果[6]。但是传统的数据挖掘方法不能完全应用于时间序列，原因有两点：时间序列表示随时间的连续测量获得的值的集合[7]，属性是有序的，分析和挖掘时，必须考虑值与值之间的前后相关性；时间序列数据长度不一致，不符合传统数据挖掘方法固定特征个数的要求。时间序列普遍存在于各领域，人们需要大量时间序列中蕴藏的知识和信息来给予我们意见和指导我们进行决策，因此时间序列数据挖掘 (Time Series Data Mining, TSDM) 已逐渐成为数据挖掘的重要分支之一。我们将这种试图从大量的时间序列数据中提取人们事先不知道的、与时间序列属性相关的有用信息和知识的方法称为时间序列数据挖掘。目前时间序列分类面临的主要问题有两个：一是计算复杂度高，主要源于时间序列长度过大和数量过多；二是时间序列数据具有长尾效应，样本极不均衡，甚至上万时间序列数据只有其中一条负样本，例如复杂设备故障数据。

时间序列数据挖掘的主要任务包括内容查询、异常检测、预测、聚类、分类和分割[7]。其中，时间序列分类（Time Series Classification, TSC）是时间序列数据挖掘的重要分支之一，是将时间序列当做输入，目的在于给时间序列赋予一个预定义的类标[8]。时间序列分类方法又可以分为三种，分别是基于相似度的、基于模型的、基于特征的分类[9,10]。比较重要的方法包括包括 Shapelet 方法[11,12]（基于相似性）、自回归移动平均模型[13]（基于模型）、深度神经网络[14]（基于模型）等方法，这几种方法各有优劣。自回归移动平均模型[15](Auto Regressive Integrated Moving Average Model, ARIMA)，是统计模型中最常见的一种用来做时间序列预测的模型，ARIMA 优点在于模型十分简单，只需要内生变量而不需要借助外生变量；

ARIMA 模型也存在不足的的地方：要求数据是稳定或者经过差分变化之后是稳定的，并且不能捕捉非线性模型，比如 ARIMA 就不用于股票时间序列数据。深度神经网络[16](Deep Neural Networks, DNN) 是近年来比较流行的机器学习算法，DNN 具有模拟复杂模型、高准确率等优点，但是也存在很多不足的地方：DNN 数据每种类别需都要较多的数据，与时间序列数据长尾效应相矛盾；DNN 的特征不具有可解释性，不能给予过多指导意义。Shapelet 是一种基于相似度计算的子序列发现算法，能够从时间序列中提取具有高分类能力的时间序列子序列[17]，具有高准确率、易解释，Shapelet 是一种基于模式分类的方法，能够为其他领域提供准确、可解释的分类结果。

Shapelet 分类有很多重要的应用，能够广泛应用于制造业、医药业、航天领域等。在制造业领域，Shapelet 分类能够应用于故障检测，可以通过对于故障机器的时间序列数据进行检测，推断发生故障的时间，便于对故障发生原因进行分析。在医药领域，能够通过检测表征人体的一些状态变化的数据是否存在某种表征病变的模式来判断是否生病，例如心电图数据可以用于判断某个病人是否有缺血症状。在航天领域，Shapelet 分类可以根据发动机声学特征的时间序列判断发动机是否处于异常状态，通过对异常状态进行预测及修复从而能够有效地保证生命财产安全。

除了上述优点和广泛应用，Shapelet 也存在很大的不足，Shapelet 发现（寻找具有高分类能力的子序列的过程）的过程非常耗时。因此，很多业界人士使用多种优化方法提高 Shapelet 的发现效率，总的来说优化方法可以划分为四类，分别是基于剪枝的，基于学习的，基于降维的，基于并行的。首先基于剪枝的 Shapelet 发现算法，主要是通过减少候选集的搜索空间即候选序列集合的大小。Random-Shapelet[18] 是以一定的概率从候选序列集合中获得子集，在子集上进行 Shapelet 的发现过程，可以减少执行时间，在子集中发现 Shapelet 主要基于高判别能力的 Shapelet 在时间序列中出现的频次比较高的原因，而采样概率是准确率和时间的权衡的结果。Yang 等人[19] 使用增强自组织增量神经网络[20](Self-Organizing Incremental Neural Network, SOINN) 将时间序列数据集中相似的子序列用一个子序列形状来代表，减少候选序列集合，在较小的候选序列集合中发现 Shapelet，从而减少了执行时间。基于学习的 Shapelet 发现算法，主要是通过建立一个模型 f 直接学习获得高分类能力的候选序列。Hou 等人[17] 通过结合广义特征向量方法和融合 lasso 正则方法来产生一个稀疏块状解。GRabocka 等人[21] 利用神经网络的方法学习获得高判别能力的候选序列。基于降维的 Shapelet 发现算法，主要使用低维空间来进行时间序列表示，然后使用低维空间进行相似度计算。Fast-Shapelet[22] 就是其中的代表，Fast-Shapelet 是使用 SAX 方法结合随机投影[23] 将时间序列降维到成低维空间，然

后在低维空间进行相似度搜索。基于图形处理器并行的 Shapelet 发现算法，对于不同候选序列对应的距离计算在图形处理器上进行并行，Chang 等人[24] 利用图形处理器进行距离计算，明显减少执行时间。虽然经过这么多研究，但是 Shapelet 发现过程的耗时问题仍然是一个难题。

1.2　研究内容和主要创新点

本文针对 Shapelet 发现过程非常耗时这一点，我们利用图形处理器 (Graphics Processing Unit, GPU) 对于 Shapelet 发现过程进行并行加速。论文的主要研究内容如下：

1. 针对 Shapelet 发现过程耗时这一点，提出关于解决耗时问题的并行框架，能够针对不同数据集大小 N，不同时间序列长度 L 选择不同的并行方案，另外并行方案中使用的距离度量方法为动态时间规整 (Dynamic Time Warping, DTW)。

2. 分别就 Shapelet 发现过程各阶段（距离计算阶段、最佳分割点计算阶段、候选序列筛选阶段）提出不同的并行解决方案。

3. 运用统一计算设备架构 (Compute Unified Device Architecture, CUDA) 相关技术和并行算法结合，使在有限的计算资源上尽可能进行更多的计算，使用的 CUDA 优化技术包括矩阵转置、合并内存访问、存储器冲突、线程束分歧、归约等。

本文针对 Shapelet 发现过程的耗时问题提出的一个基于 DTW 度量的 Shapelet 并行算法框架中，主要创新点包括以下三个方面：

1. Shapelet 并行算法框架，可以针对大数据集进行计算，针对不同得数据集大小、时间序列长度选择不同的计算方式。

2. 在距离计算阶段，采用“重用”策略降低时间复杂度；在最佳分割点计算阶段，使用一种启发式计算信息增益的方法来降低时间复杂度。

3. 使用定制化的矩阵转置大幅度降低大量的全局内存访存时间。

1.3　论文组织架构

本文围绕基于 DTW 距离度量的 Shapelet 并行算法进行阐述：

第二部分：详细介绍了 Shapelet 的相关定义及计算流程进行了详细的介绍，对于 Shapelet 通用算法进行了分析，指明了 Shapelet 发现过程耗时的原因；对于多种相似度计算/距离计算方法进行比较说明并说明了部分相似度计算/距离之间的关系；对于 GPU/CUDA 并行原理和本文用到的 CUDA 优化技术进行了详细描述。

第三部分：这里首先介绍基于 DTW 距离度量的 Shapelet 并行框架并对于并行框架中各模块功能进行了描述；对于并行框架中各模块之间执行路径进行了描述；

对于并行化之前需要解决的问题进行了识别和解决。

第四部分：本章对于并行总体方案中三个主要模块进行算法介绍，其中包括 w>0 距离计算模块并行方案，w=0 距离计算模块并行方案，最佳分割点计算模块并行方案，分别介绍三个模块的并行策略、算法设计、实现细节和性能考虑。并行策略是如何避免重复计算和算法如何被多线程利用。实现细节和性能考虑主要是本算法如何和 CUDA 技术进行结合，主要包括 w>0 距离计算模块和全局内存访问以及存储体冲突结合，w=0 距离计算模块和线程束分歧以及矩阵转置结合。这里 w 是一个控制相似度匹配时弯曲程度的参数，会在 2.3.2介绍。

第五部分：针对论文的研究目的，设计了一系列的实验，就基于 DTW 度量的 Shapelet 并行方法的准确率、执行时间在多个数据集中进行验证，并和已有优化方法进行了比较。

第六部分：总结本文的主要工作，并提出了新的问题以及可能的研究方向。

第 2 章　相关定义

本章主要对于本文的基础算法和涉及到的 GPU/CUDA 加速技术进行了介绍。首先介绍 Shapelet 算法的相关定义和通用算法，并对通用算法时间复杂度进行了分析。然后对于不同的相似度度量方法进行了分析和比较，并对部分相似度/距离度量之间关系进行了介绍。最后，详细介绍了 GPU/CUDA 并行原理以及本文并行工作使用到的 CUDA 优化技术。

2.1　Shapelet 相关定义

本章节介绍 Shapelet 的相关定义[25]，这些定义是理解 Shapelet 发现过程的前提，后文主体部分都是依照本章节进行展开的。

定义 2.1：　数据集的表示：对于一个时间序列数据集 D，和其他数据集一样，都是由输入 (时间序列) 与输出（类标）组成，数据集通常表示为公式 2-1形式。其中 T_j 是一个时间序列，y_j 是其类标，其中 $y_j \in \{A, B\}$，这里及后文中 N 表示时间序列个数，即数据集大小。

$$D = \{(T_1, y_1), (T_2, y_2), \cdots (T_N, y_N)\} \tag{2-1}$$

定义 2.2：　时间序列 $T_j = t_1, \cdots, t_L$ 是由 L 个实数组成的有序集合。$t_1, \cdots, t_L$ 是按时间顺序排列并且具有相同的时间间隔。时间序列 T_j 上的元素 t_p 可以通过 $T_{j,p}$ 来表示。时间序列 T_j 的长度可以用 $|T_j|$ 表示，也可以通过 L 来表示。后文提到的 L 都是数据集中时间序列的长度，而且本文中提到的时间序列都是等长的，长度为 L。

定义 2.3：　子序列 $(Subsequence, S)$，对于长度为 L 的时间序列 T_j，对于任何一个子序列可以由两个参数（子序列起始位置 s 和长度 len）确定唯一的子序列 $T_{j,s}^{len}(s.t. \quad s + len <= L)$。对于任意一个子序列 S 的长度可以通过 $|S|$ 来表示，但不能通过 L 表示。

时间序列 T_j 的子序列集合可以通过 $Subset(T_j)$ 表示，$Subset(T_j)$ 集合大小为 $\frac{L(L+1)}{2}$（这里不包括长度为 0 的时间序列）。数据 D 的子序列集合可以用 $SubSet(D)$ 表示，数据集 D 有 N 个长度为 L 的时间序列，则 $SubSet(D)$ 的集合大小为 $\frac{NL(L+1)}{2}$。

因为每一个子序列都是 Shapelet 的候选序列，因此我们把子序列由称为候选序列，D 对应的子序列集合 $SubSet(D)$ 又称为候选序列集（合）。

定义 2.4：时间（子）序列 A, B 之间的距离 $Dist(A, B)$，用于表示具有相同长度的时间 (子) 序列 A 和时间 (子) 序列 B 的相似度。

原始 Shapelet 算法使用欧式距离 (Euclidean distance) 作为子序列相似度度量，但是作为距离，只要满足对称性、非负性、三角不等式形式的，就可以满足 $Dist(A, B)$ 的要求，这里 $Dist(A, B)$ 可以扩展到其他广义距离，包括曼哈顿距离，欧氏距离，DTW 距离等。

定义 2.5：候选序列 S 到时间序列 T_j 之间的距离（必须满足 $|S| <= |T_j|$），是 T_j 序列中所有长度为 $|S|$ 的子序列与 S 距离 $Dist(T_{j,p}^{|S|}, S), p = 1 \rightarrow L - |S| + 1$ 的最小值，如公式 2-2。

$$SubDist(S, T_j) = \min_{p=1 \rightarrow L-|S|+1}(Dist(S, T_{j,p}^{|S|})) \tag{2-2}$$

候选序列 S 和数据集 D 的所有时间序列 $T_j, j = 1, 2, \cdots, N$ 计算距离，可以获得一个 $SubDist(S, T_j)$ 的向量，记作 $dist$，如公式 2-3。

$$dist = \{SubDist(S, T_1), SubDist(S, T_2), \cdots, SubDist(S, T_N)\} \tag{2-3}$$

将 $dist$ 中的元素和类标一一对应起来组成的“距离-类标对”的向量，记作 $\mathcal{F}$，如公式 2-4，其中有 $\mathcal{F}_{j,1} = SubDist(S, T_j)$，$\mathcal{F}_{j,2} = y_j$。

$$\mathcal{F} = \{(SubDist(S, T_1), y_1), (SubDist(S, T_2), y_2), \cdots, (SubDist(S, T_N), y_N)\} \tag{2-4}$$

定义 2.6：信息增益，根据特征 A 将训练数据集 D 划分为 M 个子集 $D_1, D_2, \cdots, D_M$，特征 A 对于数据集 D 的信息增益 $g(D, A)$ 是数据集 D 的信息熵 $H(D)$ 和在 A 给定条件下经验条件熵 $H(D|A)$ 之差，即公式 2-5。A 特征对应的信息增益越大，则说明特征 A 的分类能力越强，Shapelet 使用信息增益作为候选序列筛选的标准。

$$g(D, A) = H(D) - H(D|A) = H(D) - \sum_{i=1}^{M} \frac{|D_i|}{|D|} H(D_i|A) \tag{2-5}$$

候选序列 S 与数据集 D 中每一个时间序列 $T_j, j = 1, 2, \cdots, N$ 的距离 $SubDist(S, T_j)$，其中 $SubDist(S, T_j)$ 的所有取值分布在 $(0, \infty)$ 区间，如图 2.1。候选序列 S 和阈值 d_{th} 共同作为特征 (S, d_{th}) 将数据集 D 分为 $D_1 = \{T_j, SubDist(S, T_j) < d_{th}, j = 1 \rightarrow N\}$ 和 $D_2 = \{T_j, SubDist(S, T_j) \geq d_{th}, j = 1 \rightarrow N\}$ 两部分。特征 (S, d_{th}) 相对数据集的信息增益为数据集 D 的信息熵减去子集 D_1 和子集 D_2 的信息熵的权重和，如公式 2-6

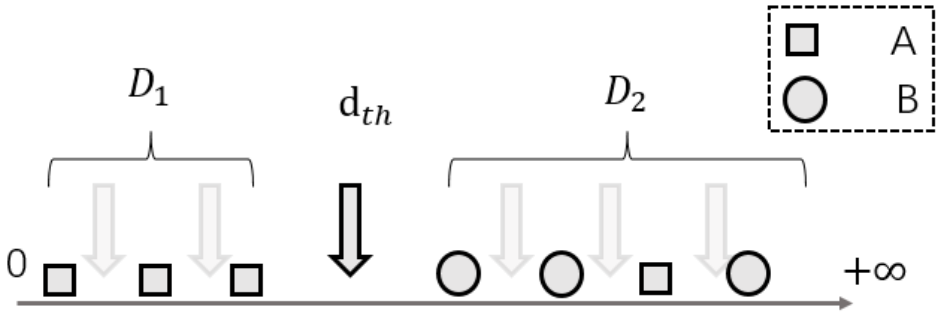

图 2.1　$SubDist(S, T_j), T_j \in D$ 在 $(0, \infty)$ 的分布

$$g(D, (S, d_{th})) = H(D) - H(D|(S, d_{th})) = H(D) - (\frac{|D_1|}{|D|}H(D_1) + \frac{|D_2|}{|D|}H(D_2)) \quad (2\text{-}6)$$

定义 2.7：最佳分割点: 在候选序列 S 下，有多个阈值 d_{th}，如图 2.1，在这些分割点中，存在一个阈值 $d_{osp(S)}$（或者是一段区间），满足 $g(D, (S, d_{osp(S)})) \geq g(D, (S, d_{th})), \forall d_{th} \in \mathbb{R}_+$，则将 $d_{osp(S)}$ 记为候选序列 S 对应的最佳分割点。

数据集 D 中有 $O(NL^2)$ 个候选序列，即有 $O(NL^2)$ 个特征组合 $(S, d_{osp(S)})$，需要选取分类能力最高的特征组合，分类能力最高特征组合中的候选序列即为 Shapelet，如定义 2.8。

定义 2.8：数据集 D 中的 Shapelet，使用 $Shapelet(D)$ 表示。对于给定数据集，存在一个子序列 $Shapelet(D)$，使 $g(D, (Shapelet(D), d_{osp(Shapelet(D))})) \geq g(D, (S, d_{osp(S)})), \forall S \in SubSet(D)$。$(Shapelet(D), d_{osp(Shapelet(D))})$ 特征将作为 Shapelet 算法那最终的分类器。

定义 2.9：Shapelet 计算过程三阶段: 本文将 Shapelet 计算过程分为三个阶段，分别为距离计算阶段、最佳分割点计算阶段、候选序列筛选阶段。其中距离计算阶段是指从候选序列 S 和数据集 D 到 S 相对数据集 D 所有时间序列计算距离获取距离-类标对的集合 $\mathcal{F}$ 的过程，即计算 $\mathcal{F}$ 的过程；最佳分割点计算阶段是指从 $\mathcal{F}$

到对应的最佳分割点 $d_{osp(S)}$ 以及对应的信息增益的过程，相当于定义 2.7过程；候选序列筛选阶段是指从多个候选序列中选出分类能力最高的候选序列，相当于定义 2.8过程。章节 3.2.1解释了为什么从中间结果 $\mathcal{F}$ 处分成两个阶段，章节 3.1.2解释了为了要从最佳分割点处分为两个阶段。

从以上定义来看，时间序列可以分类的前提是，数据集中的一类（比如 A 类）存在一种模式（波形），这种模式和 A 类时间序列 $(T_j, y_i \in A)$ 表现出较小的距离，和另一类 B 类时间序列 $(T_j, y_i \in B)$ 表现出较大的距离。而 Shapelet 算法那就是负责寻找能够区分这两类时间序列的模式以及对应的距离阈值的算法。

2.2　通用算法分析

算法 2.1 Shapelet 原始算法

```
1:  function ShapeletNaiveAlg(D)
2:      lastS ← φ, lastinfogain ← 0, lastdosp ← 0, leftisAorB ← A
3:      for all S ∈ SubSet(D) do //O(NL^2) 个候选序列
4:          F = {(..., SubDist(S, T_j), y_j), ...}, j = 1, 2, ··· , N //O(NL^2)
5:          计算 g(D, (S, d_osp(S))) //O(N log(N))
6:          if g(D, (S, d_osp(S))) > lastinfogain then
7:              更新 lastinfogain,lastS,lastdospd_osp(S),leftisAorB
8:          end if
9:      end for
10:     return lastinfogain, lastS, lastdops, leftisAorB
11: end function
```

如算法 2.1为 Shapelet 发现的通用算法，是所有子序列中发现具有最佳分类能力的子序列的过程。其中：

line 4是对于计算一个候选序列 S 相对于数据集 D 中每个时间序列 T_j 的过程，时间复杂度为 $O(NL^2)$;

line 5是寻找一个阈值 $d_{osp(S)}$ 与候选序列 S 共同作为特征满足 $g(D, (S, d_{osp(S)})) \geq g(D, (S, d_{th})), \forall d_{th} \in \mathbb{R}_+$，时间复杂度为 $O(N \log(N))$。

line 3:$SubSet(D)$ 集合大小为 $O(NL^2)$，因此 *line* 3处循环为 $O(NL^2)$ 次；循环内部时间复杂度为 $O(NL^2)$。

因此 Shapelet 通用算法的时间复杂度为 $O(N^2L^4)$，这个时间复杂度非常高，特别 N, L 比较大的情况，严重地影响执行时间。

2.3　相似度/距离度量方法

章节 2.1描述的相似度/距离计算都是基于两个时间（子）序列 A, B 的相似性度量 $Dist(A, B)$ 进行的，本章节就 $Dist(A, B)$ 展开介绍。时间序列相似性/距离是许多计算系统的核心，同时也是时间序列聚类和分类的最重要的组成部分之一[26]。正是由于相似度/距离度量的重要性，多种相似度/距离度量的方法先后被提出。在本章节中，我们对于几个具有代表性的时间序列相似度/距离度量及其所属类别进行评估，相似度/距离计算可以简单分为[27]：锁步距离测量（欧氏距离）、基于特征测量（傅里叶系数）、基于模型测量（自回归）、弹性测量（动态时间规整、实数序列的编辑距离）。

在上述多种相似性计算当中，基于特征测量和基于模型测量都没有直接利用时间序列之间的距离而是使用时间序列提取的特征进行计算相似性的，和 Shapelet 算法的滑动窗口距离取最小值的方法不符，因此对于这两种距离不予考虑。

对于两个时间序列 $A = \{a_1, a_2, \cdots, a_M\}$ 和 $B = \{b_1, b_2, \cdots, b_M\}$，本章主要讨论欧氏距离、实数序列的编辑距离、动态时间规整距离进行介绍并比较。

2.3.1　欧氏距离

欧氏距离是使用最普遍的距离度量方法，是度量 M 维空间上的两个点的直线距离，使用 $Dist(A, B)$ 或者 $Euclid(A, B)$ 表示。将欧氏距离应用在时间（子）序列上，要求两个时间序列 A, B 的长度相等 $|A| = |B|$。在实际使用中经常将欧式距离的定义会进行扩展，使用两个长度相等时间序列 A, B 的差的 L_n 范数作为欧氏距离[28]，如公式 2-7。有时候，为了便于计算，经常使用 $\sum_{i=1}^{M}(A_i - B_i)^n$ 记做欧氏距离，其中 $n = 1$，为曼哈顿距离，$n = 2$，为狭义欧氏距离，n 一般会作为一个底层距离参数，可以根据情况调节。

$$Euclid(A, B) = (\sum_{i=1}^{M}(A_i - B_i)^n)^{\frac{1}{n}} \tag{2-7}$$

欧氏距离的优点在于计算简单，复杂度低；缺点要求两个时间序列各元素一一对应，不能弹性地计算时间序列之间距离，不能扭曲对应，这也是欧氏距离属于锁步距离的原因。

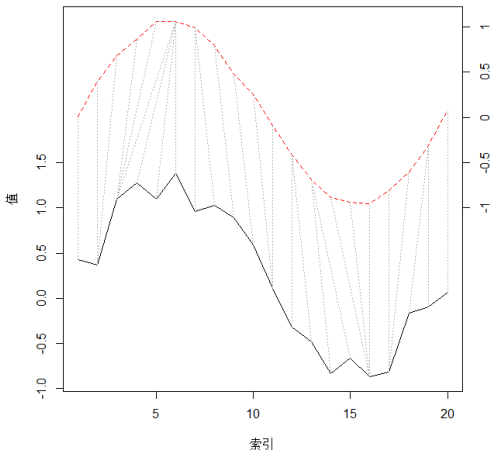

图 2.2　时间序列 A, B 对齐方式

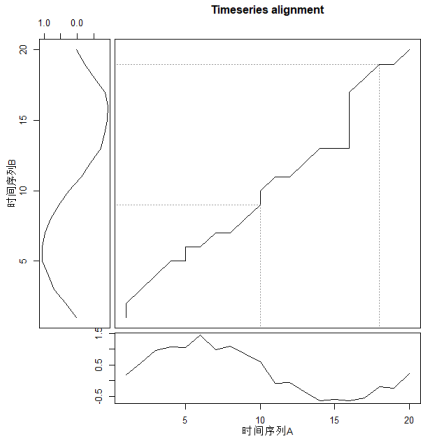

图 2.3　时间序列 A, B 规整路径和代价矩阵

2.3.2　动态时间规整

动态时间规整[29](Dynamic Time Warping, DTW) 是一种通过将一个时间序列 A 延时间轴进行非线性的延展或者压缩，目的使 A 序列和另一时间序列 B 能够很好地对齐，对齐方式如图 2.2。这种对齐方式是可以通过一种动态规划的算法计算获得，如图 2.3，图中从 $(0,0)$ 到 $(|A|,|B|)$ 的路线称为规整路径。

A 和 B 之间的 DTW 距离用 $DTW(A,B)$ 来表示，$DTW(A,B)$ 的递归方程如公式 2-8[30]①。

$$
\begin{aligned}
&Dist(A,B) = d(M,M) \\
&d(i,j) = |a_i - b_j|_n + \min \begin{cases} d(i-1,j) \\ d(i,j-1) \\ d(i-1,j-1) \end{cases} \\
&d(0,0) = 0; d(i,0) = \infty; d(0,j) = \infty; i = 1,2,\cdots,M; j = 1,2,\cdots,M
\end{aligned}
\tag{2-8}
$$

我们要求上面递推公式 2-8比较的两个时间序列长度是相等的 $|A| = |B| = M$，但其实 DTW 不要求时间（子）序列长度一致。这里要求 $|A| = |B| = M$ 主要基于两个原因考虑：第一，这样对于任何一个子序列 S 与 T_j 的距离，如果不要求（子）序列长度一致，就必须计算 T_j 所有子序列和 S 的 DTW 距离，然后在这些距离中取最低值，这样 S 需要和 $O(L^2)$ 个子序列计算距离，使计算的复杂度变高；第二，不能判断哪个时间序列更具相似性，比如有 D, E, F 三个时间序列，其中 $|D| << |E| < |F|$,$DTW(D,E) < DTW(F,E)$，但是这里不能就此认定对于 E，D 比

① n=1,2

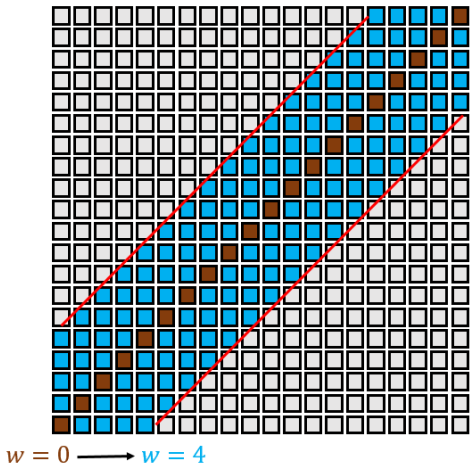

图 2.4　Sakoe-Chuba-band 限制区域

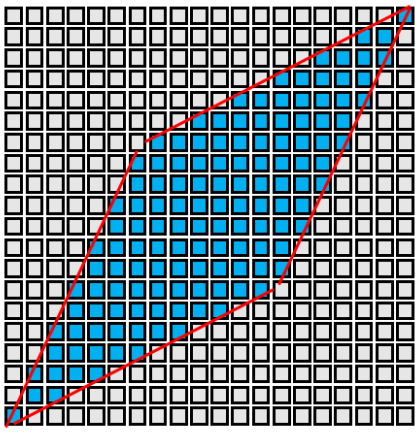

图 2.5　Itakura-band 限制区域

F 更具有相似性。

递归方程 2-8计算 DTW 距离的时间复杂度和空间复杂度都是 $O(M^2)$(如果 $|A| = |B| = M$)，相比于欧氏距离都很大。因为 DTW 时间空间复杂度高的原因，有人提出了 FastDTW[31]，通过将规整路径限制在一定区域内来加速计算，限制区域主要包括 $Sakoe-Chubaband$（如图 2.4）和 $Itakuraband$（如 2.5）两种方法。这两种限制区域的方式实质上将时间序列扭曲对应限制在一定的范围内。

本文使用到的是 Sakoe-Chuba-band 限制区域方法，这里需要对它进行介绍。如图 2.4，Sakoe-Chuba-band 限制区域沿反对角线方向向两边延展，而 w 是一个控制限制区域宽度的参数，这里使用 $DTW(A, B, w)$ 表示时间序列 A, B 之间的 w 参数下限制区域 DTW 距离。$w(w >= 0)$ 作为限制区域的宽度的参数，当 $w = 0$ 时，限制区域为反对角线；当 $w = M$（与时间序列 A, B 长度相等）时，限制区域为整个代价矩阵区域，这个时候限制区域 DTW 距离计算退化为于原始的 DTW 距离。因为 $DTW(A, B)$ 或者 $DTW(A, B, w = M)$ 距离是限制区域 $DTW(A, B, w)$ 距离的特殊形式，后文提到的 DTW 距离都是指 $DTW(A, B, w)$ 距离，而原始的 DTW 距离将会用 $DTW(A, B, w = M)$ 来表示。

算法 2.2是 $DTW(A, B, w$ 距离计算的算法，时间复杂度和空间复杂度都为 $O(wM)$。

Algorithm 2.2 DTW 距离计算（$Sakoe-Chuba-band$ 限制区域）

```
1: function DTW(A, B, w) //|A| = |B| = M
2:     D = array(M + 1, M + 1)
3:     for m = 0 to M do
4:         if m < w then
5:             D_{m,0} ← ∞, D_{0,m} ← ∞
```

```
        else
            D_{m,w+m} ← ∞, D_{w+m,m} ← ∞
        end if
    end for
    for m = 1 to M do
        for n = max(m − w, 1) to min(M, m + w) do
            D_{m,n} = min(D_{m−1,n−1}, D_{m−1,n}, D_{m,n−1}) + d(a_m, b_n)
        end for
    end for
    return D_{M,M}
end function
```

2.3.3　实数序列的编辑距离

实数序列的编辑距离[32](Edit Distance on Real Sequence, EDR) 可以视为原始编辑距离算法在实数序列领域的扩展，递归方程如公式 2-9。

$$
\begin{aligned}
&EDR(A,B) = d(M,M)\\
&d(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ d(i-1,j-1) & \text{if } \theta(a_i - b_j, \epsilon) \\ \min(d(i,j-1), d(i-1,j-1), d(i-1,j)) + 1 & \text{otherwise} \end{cases}
\end{aligned} \tag{2-9}
$$

EDR 要求时间序列长度相等 $|A| = |B|$，主要是基于和 DTW 距离相同的考虑。其中，θ 是一个阶跃函数，参数 ϵ，$\epsilon \in [0, \infty)$ 是一个控制两个序列样本 a_i, b_j 是否匹配的阈值参数，在实际的 EDR 计算中，ϵ 作为一个超参数需要提前确定。但是如果将 EDR 应用到 Shapelet 中，这样的参数 ϵ 难以确定，对于某一个数据集适应的 ϵ 不一定适应于其他数据集。

2.3.4　欧式距离与动态时间规整比较

如图 2.4，当 $w = 0$ 时，DTW 代价矩阵将会退化成代价矩阵的对角线，$D(i, j)$ 的递推方程也将变为沿对角线递归，如图 2.4黄色部分和 2-10递归方程，因此，当 $w = 0$ 时，时间序列的距离 $DTW(A, B, 0)$ 则退化欧式距离。

当 $w = 0$ 时，在 DTW 距离计算等价于欧氏距离，但是相比欧式距离会多出 $2M$ 个比较计算过程和 $2M$ 次访存过程。

$$d(i,j) = |a_i - b_j|_n + \min \begin{cases} d(i-1,j) \\ d(i,j-1) \\ d(i-1,j-1) \end{cases} = |a_i - b_j|_n + d(i-1,j-1) \tag{2-10}$$

$$d(0,0) = 0$$

$$DTW(A,B,0) = d(M,M) = \sum_{i=1}^{M} |a_i - b_i|_n = Euclid(A,B)$$

2.3.5　相似度/距离度量方法选择

在章节 2.3前面部分介绍各种相似度/距离以及存在的联系。本章节需要选出适用的相似度/距离计算作为 Shapelet 的相似度/距离调用子程序。

欧式距离 $Euclid(A,B)$ 是 Shapelet 原始算法的相似度/距离度量，是时间序列数据挖掘应用上有很健壮的性能指标。欧式距离相比其他距离度量的优势在于时间复杂度低，但由于欧氏距离是锁步距离，要求序列上的元素一一对应。

DTW 距离 $DTW(A,B,w)$ 容许时间序列进行局部延长和压缩，能够克服欧氏距离由于时间序列发生扭曲而无法进行匹配的问题。DTW 距离的时间复杂度为 $O(wM)$，这里 w 作为控制时间序列扭曲程度一个参数，一般比较小（只有时间序列过度扭曲才需要很大的限制区域即 w 很大来包含时间序列之间的规整路径），因此，$DTW(A,B,w)$ 中 w 取值范围为 $[0,C), s.t. C << M$，时间复杂度仍为线性。章节 2.3.4对于欧氏距离和 DTW 关系进行了介绍，欧式距离是 DTW 的特殊形式，存在 $DTW(A,B,w) \leq DTW(A,B,w=0) = Euclid(A,B)$ 关系，使用 $DTW(A,B,w)$ 作为相似度/距离调用在保持欧式距离的健壮性基础上，可以使时间序列进行一定扭曲来进行匹配。

EDR 需要提前确定一个超参数 ϵ 或者对于 $EDR(A,B)$ 中 A,B 进行标准化，所以不适合作为 Shapelet 作为相似度/距离调用。

综上所述，本文选择 $DTW(A,B,w), w \in [0,C), s.t. C << M$ 作为 Shapelet 的相似度/距离度量。当 $w > 0$ 时，距离计算阶段采用的距离度量是 $DTW(A,B,w)$；而当 $w = 0$ 时，为了避免不必要的计算，距离计算阶段采用 $DTW(A,B,w)$ 距离度量的等价方案 $Euclid(A,B)$。因此，距离计算阶段会根据 w 的不同分为：w>0 距离计算模块和 w=0 距离计算模块。

2.4　GPU/CUDA 并行原理及相关优化技术

GPU/CUDA 并行采用单指令多数据 (Single Instruction Multiple Data, SIMD) 并行方式的。SIMD 并行是将数据元素映射到并行处理线程交由相同的程序处理。本章节首先从 GPU 硬件架构和 CUDA 并行原理方面进行介绍并行技术，其中，GPU 硬件结构部分介绍 GPU 的硬件结构组成构成并行的基础，CUDA 并行原理部分来说明 CUDA 如何利用 GPU 硬件完成并行任务的。然后在 GPU/CUDA 并行原理的基础上介绍本文用到的 CUDA 优化技术。GPU 加速并行并没有改变算法的时间复杂度，而是在 CPU 执行时间的基础上除以一个常数，这个常数由两个因素决定，一是 GPU 结构及流处理器簇个数和每个流处理器簇中流处理器个数，二是与优化技术和算法的结合以及算法本身的实现有关。硬件基础没有办法改变，更多的是依靠算法实现和优化技术，一个高效的优化算法相比一般的优化算法也会有很大的速度区别，比如归约中的 $kernel7$ 相比 $kernel1$ 在 $4M$ 元素下加速倍数为 21 倍[33]。章节 2.4.3-章节 2.4.6详细地介绍了本文用到的 CUDA 优化技术，包括合并内存访问、隐藏延时、线程束分歧、存储体冲突等

本章节的图 2.6和图 2.7全部根据文献[34] 和文献[35] 综合绘制。本章节介绍 GPU/CUDA 并行和 CUDA 相关技术都是以英伟达的图形处理器为例介绍的，对于其他加速设备以及并行技术没有覆盖。

2.4.1　GPU 硬件结构

图 2.6介绍了 GPU 的典型硬件结构，GPU 硬件包括了流处理器、流处理器簇、内存（全局、共享、常量等）几个关键部分。下面从这个几个关键部分开始介绍 GPU 硬件结构。

流处理器 (Streaming Processor,SP) 是 GPU 最基本的单元（计算单元），也是执行一个线程的基本单元，因此 SP 又称为 CUDA 核，如图 2.6的 SP。

流处理器簇 (Stream Multiprocessors,SM) 是由一定数量的 SP 加上指令单元、寄存器、共享内存、L1/L2 缓存等组成。从图 2.6可以看出，每个 SM 是一个 SIMD 处理单元，由多个 SP 和一个指令单元组成。而 GPU 实际上是一个 SM 的阵列，而 SM 又是由多个 SP 组成，GPU 进行并行计算的本质就是众多 SP 共同进行数据计算的过程。以 GTX-1080 来讲，GPU 含有 20 个 SM，每个 SM 含有 128 个 SP，即在 GPU 最多可以同时运行 $20 * 128$ 个线程。

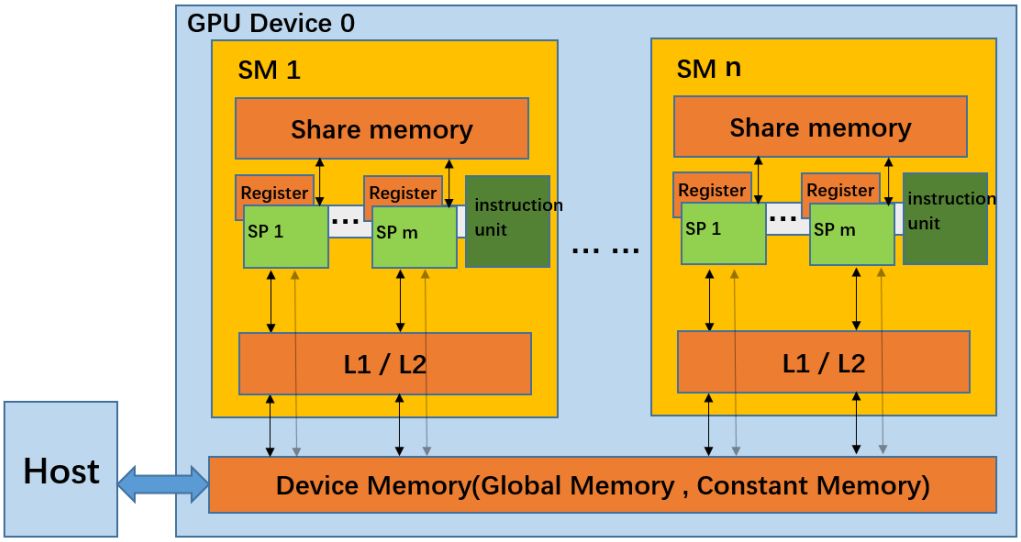

图 2.6　GPU 硬件结构

全局内存 (Global memory) 是独立于 GPU 内核的内存，同时也是 GPU 中空间最大的内存，但访问延时最长，如图 2.6。全局内存有以下特点：对于所有 SP 可见（所有 SP 都可以读写全局内存）；可以通过和主存建立主机和 GPU 之间的数据通信；两个并行任务之间的中间结果必须使用全局内存来处理。

共享内存 (Share memory)，是存在 SM 内部可以由 SM 内部多个线程共享使用的内存，速度是全局内存的几百倍。可以通过共享内存完成线程之间的通信和协作计算。共享内存对于某些特定的线程可见，具体后文会提及。

寄存器 (Registers) 是 GPU 中最快的内存，用于存储线程中的临时变量，只在线程内部可见。

本地内存 (Local memory) 是存储堆栈中无法容纳的所有内容的临时变量，本地内存存储在全局内存中，但对于线程可见。

表 2.1　GPU 中各种存储之间的比较

内存种类	访 存 延 时 (时钟周期)	内存大小	可见范围	周期
全局内存	~400-600	8G	线程网格	整个过程
共享内存	~20	48K(每个线程块)	线程块	线程
寄存器	~1	64K(每个线程块)	线程块	线程
本地内存	~400-600	同全局内存	线程块	线程

常量内存和纹理内存在本文中没有涉及，在这里不做介绍。

表 2.1对于使用到的内存的性能进行比较，更好地理解各内存在程序中所起的

功能。

2.4.2　CUDA 并行原理

前面从硬件角度介绍 GPU 的硬件结构，本章节将从软件角度叙述 CUDA 如何在 GPU 的硬件基础进行并行计算的，首先介绍几个 CUDA 的基本概念。

线程 (Thread) 是并行程序的基本单元，一个并行程序是由很多线程共同执行，图 2.7带箭头的曲线表示一个线程。

线程块 (Block) 由多个线程组成，在一个 Block 中线程可以进行同步（需要通过程序控制），也可以通过共享内存进行通信。如图 2.7，每个 $Block(x, y)$ 也是有线程的阵列组成，如 $Block(1, 1)$。

线程束 (Warp)，将 Block 连续线程 ID 聚合起来的 32 个线程（比如 thread 0-31,32-63）定义为一个 Warp，Warp 是调度和运行的基本单元，Warp 中所有线程并行的执行相同的指令并且是严格同步的，这种同步是由硬件底层完成而不需要通过程序来控制。Block 会将连续的 32 个线程聚合成一个 Warp，当一个 Block 的线程个数不是 32 个整数倍时，硬件会帮助不足 32 个线程的部分凑足 32 个线程形成一个 Warp。

线程网格 (Grid)，多个 Block 构成线程网格，Grid 是主机程序调度的接口。如图 2.7所示，主机调用了两个 Grid 程序，其中 Grid 1 是由 2*3 个 Block 组成。

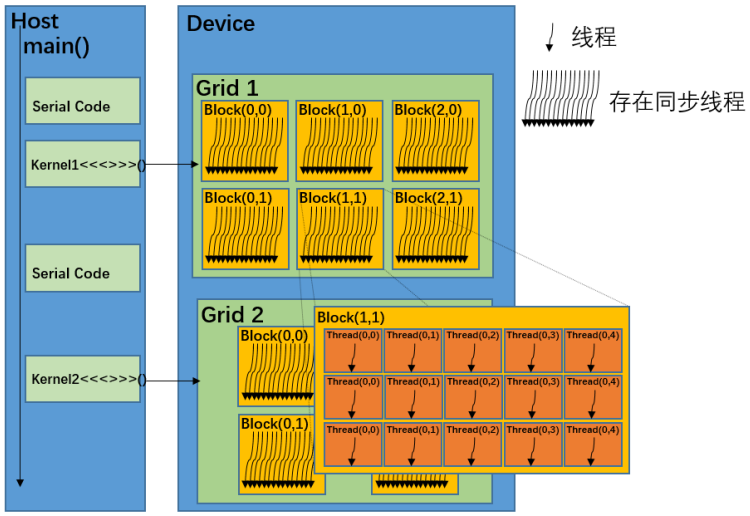

图 2.7　NVIDIA CUDA 架构

以 GTX 1080 为例，一个 Grid 最多可以有 $2^{31} - 1$ 个 Block，一个 Block 最多可以有 1024 个线程，但是在硬件上每个 GPU 只有 20 个 SM，每个 SM 只有 128 个。

正常情况下 2560 个 SP 是不可能运行这么多线程的。

一个 SP 只能可以执行一个线程，但是实际上并不是所有的线程能够在同一时刻执行，SM 运行线程也是采用了分时复用的思想。连续的 32 个线程形成一个 Warp，一个 Block 中的 1024 个线程分成 32 个 Warp。而 GTX 1080 一个 SM 有 4 个 Warp 调度单元，可以同时容纳 4 个 Warp 运行。起初所有 Warp 都在就绪队列中，首先从就绪对列取出 4 个 Warp 在 SM 上执行；当某个 Warp 因为读取内存操作或者其他原因而挂起时，Warp 调度单元从就绪队列取出一个 Warp 执行；当正在内存请求的 Warp 完成内存请求之后，线程束将会进入就绪状态；当线程束执行完任务时进入结束状态；这样不断循环完成一个 SM 内所有 Warp 的执行。

多个 Warp 在 4 个 Warp 调度单元通过分时复用的方式进行并行运行，同样道理，一个 Grid 中大量的 Block 在 20 个 SM 中运行的原理是一致的，都是利用计算资源分时复用的方法。CUDA 并行是由成千上万的线程并行完成的，这些线程在软件层面是同时并行的，但是在物理层面最多只有 2560 个线程在运行，其他都在挂起或者就绪状态。

2.4.3　合并内存访问

从表 2.1可以看出访问全局内存是内存访问耗时最长的环节，但全局内存又是 GPU 内存最大的部分，当处理大量数据的并行时，不可避免地使用全局内存，这样必然导致大量的全局内存读写操作而产生访存延时，全局内存访问有可能成为性能优化的瓶颈。这里可以通过合并内存访问[36] 来解决全局内存访问耗时问题。

合并内存访问 (Coalesced Memory Access, Coalesced)，是当特定的访问条件满足时，设备可以将一个 Warp 内所有线程的全局内存访问操作合并成一个事务。这里特定的条件是一个 Warp 内线程访问连续的 128B 空间，而且线程和全局内存地址一一对应（但不要求严格顺序对齐），如图 2.8[37]。这种合并内存访问模式只需要经过一次 L1 缓存的过程，大大地提高了全局内存访问速度，由图 2.8红色区域表示。

图 2.8　合并内存访问 (所有线程一个 L1 缓存行)

合并内存访问加速全局内存访问可以通过 Cache 命中观点来解释：当一个

Warp 的 32 线程同时读写连续的 128B 空间，假设某个线程首先读取了 128B 空间中的某个地址，设备会将这连续的 128B 空间缓存到 L1 缓存行中 (设备中 L1 缓存行的大小为 128B)，其他 31 个线程再进行读写这个 128B 地址空间时，可以直接命中到 L1 缓存行, 而 Cache 的访问时延远小于全局内存，不需要经过内存读写挂起的过程，大大地降低了访问时延。

当一个 warp 的连续线程访问连续空间但没有按照 L1 对齐时，会请求两个 128B 的 L1 缓存行，如图 2.9，会出现两次缓存 L1 的过程，至少需要经过两个全局内存时延等待。而且这种情况一般会伴随着同一个 128B 地址被两个 warp 访问，如果同一个 128B 地址被两个 warp 访问，而且这两个 warp 又不在一个 SM 中，就会因为考虑数据一致性的问题而导致延时。

图 2.9　非对齐顺序地址访问 (使用两个 128B)

合并内存访问大大减少了全局内存访问时延，一般数据量过大的并行计算都是需要全局内存配合合并内存访问来解决。同时，合并内存访问、全局内存、共享内存结合使用可以完成很多复杂操作，比如矩阵转置等。

2.4.4　延时隐藏

线程不可避免会出现访问各种内存访问而产生的访存延时，为了使访存延时不影响执行速度，可以使用隐藏延时的方法来解决。SM 使通过调度 Warp 来执行线程, 可以通过增加 Warp 的个数来使 Warp 调度单元尽可能选择已经就绪的 Warp，避免选择正在因为延时而挂起的 Warp。将通过执行多个 Warp 来获取较高吞吐量的方法称为隐藏延时。

对于这种性能度量通常通过占用率来评价，占用率通过并行执行的 Warp 数量除以并发运行的最大可能 Warp 数量。高占用率意味着 Warp 调度器有很多 Warp 可供选择，能够隐藏延时访存等。

在实际优化中，可以对于一个 Block 中执行的 Warp 个数进行调节，选择执行时间最短的。另外尽量避免全局内存访存之后出现同步操作，这样容易使访存时间没有办法隐藏。

2.4.5　线程束 Warp 分歧

因为线程束 Warp 使 GPU 调度得基本单元，每个 SM 又是 SIMD 处理器（只有一个指令单元，但有多个处理器），因此 Warp 内的线程只能执行同一条指令。当遇到分支语句时，不同的线程需要执行不同的语句，这和 Warp 内线程只能执行一个指令互相矛盾。为了解决这个矛盾，分支语句会被序列化，一个 Warp 中同时只能执行一个分支即部分线程执行，其他分支的线程都处于挂起状态，这样会导致性能的降低，这种现象称为线程束分歧 (Warp divergence,Warp 分歧)。

为了 Warp 分歧影响 GPU 并行效率，尽量避免同一个 Warp 存在不同的分支路径，如果不能避免 Warp 分歧的出现，则尽量保证 Warp 内少数线程执行的分支内容尽可能少，或者提前并行计算。

2.4.6　共享内存使用和存储体冲突

前面提到共享内存，相比全局内存拥有更小的时延和更高的带宽，并且线程之间可以通过共享内存进行通信和协作计算，但是使用共享内存有一个前提条件：不能出现存储体冲突。

首先看一下 SM 中共享内存结构，一个 SM 上的共享内存实际上是被分为可以同时访问的同等大小的内存块 (Bank)，因此可以达到共享通信和增加带宽的作用。共享内存中的地址以 4bytes 为单位依次分配到 16 个存储体 (Bank) 中，如图 2.10，$\_\_share\_\_\ \ int\ \ data[128]$，假设，那么 $data[0], data[16], \cdots$ 属于 Bank0，$data[1], data[17], \cdots$ 属于 Bank1，...。

当每个线程 (半个 Warp) 访问不同 Bank 时，可以快速读写共享内存，如图 2.11。但当出现多个线程 (半个 Warp 内) 同时访问一个 Bank 的不同地址时，就会出现存储体冲突 (Bank Conflict)，这多个线程的访问将被序列化，分先后访问存储体 (这部分工作是由硬件完成的)，会造成多倍的访问时延，如图 2.12，每个存储体被两个线程访问称为 2-way Bank Conflict。但是有一个特殊情况，当一个 Warp 内的所有线程访问一个存储体中的同一个地址时，共享内存会将这个地址下的数据广播到 Warp 所有线程中，本文没有涉及，不作详述。

Bank Conflict 经常出现在多个线程访问不连续的共享内存的情况下，当线程之间访问的共享内存间隔 (Stride,4Bytes 计为一个间隔) 为固定的某个数比如 2 时，则每两个线程 (thread0 和 thread9，thread1 和 thread10...) 就会访问同一个存储体 Bank，这样就会产生存储体冲突 (Bank Conflict)。为了避免出现存储体冲突 (Bank Conflict) 现象，可以通过以下方法避免：

1. 最好确保一个 Warp 的线程访问连续的一块共享内存，线程就能够访问到不

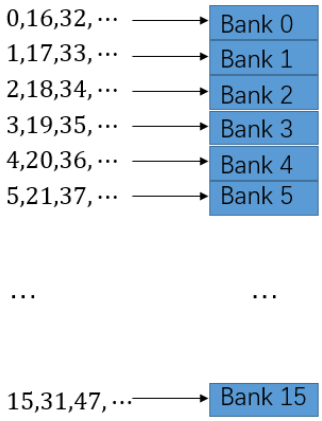

图 2.10　共享内存存储体

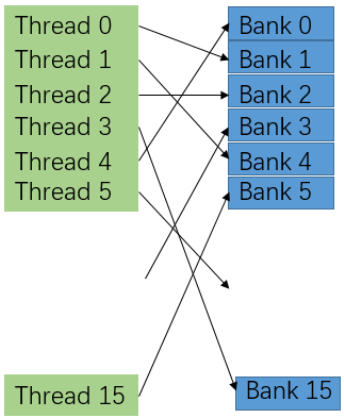

图 2.11　No bank Conflict

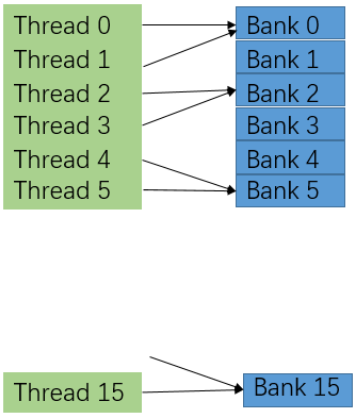

图 2.12　bank Conflict

同的存储体中；

2. 使连续线程访问不连续共享内存地址，可以对其进行调整，比如访问矩阵的列，可以对矩阵先进行转置；

3. 使用数据对齐的方式，尽量使用 *int*, *float* 占据 4Bytes 的格式，避免使用 *double* 等数据格式；

4. 当线程不能访问连续的地址，而且访问的地址之间存在一个 stride，需要保证 stride=1,3,5,7,9... 等奇数。

2.5　本章小结

本章首先介绍了 Shapelet 的相关定义和计算过程，为本文展开工作铺垫基础。然后对于 Shapelet 通用算法进行了描述，并分析了其时间复杂度，指明了 Shapelet 发现过程耗时的原因。然后对不同的相似度/距离度量方法进行了比较和分析，并对部分相似度/距离度量之间关系进行了介绍，选择了 $DTW(A, B, w)$ 作为本文 Shapelet 方案的度量距离。最后对于 GPU/CUDA 并行原理和后文使用到的 CUDA 优化技术进行了详细介绍，为后文并行工作的描述打下基础。

第 3 章　基于 DTW 距离度量的 Shapelet 并行总体方案

本章主要包括基于 DTW 距离度量的 Shapelet 并行总体方案和并行化需要解决的问题。并行总体方案部分主要包括 Shapelet 发现并行框架，并行框架中各个模块的功能，并行框架执行路径的选择。并行化需要解决的问题部分主要是并行设计需要解决的一些关键问题。

3.1　基于 DTW 距离度量的 Shapelet 并行总体方案

Shapelet 在实际应用既需要训练模块的离线部分，又需要预测模块的在线部分，因此 Shapelet 总体方案包括两个部分：离线训练部分和在线预测部分，如图 3.1。离线训练部分是一个训练模型的过程，输出一个最具有分类能力的子序列。在线预测部分主要是用于对未知的时间序列进行分类和评估预测模型的准确率。

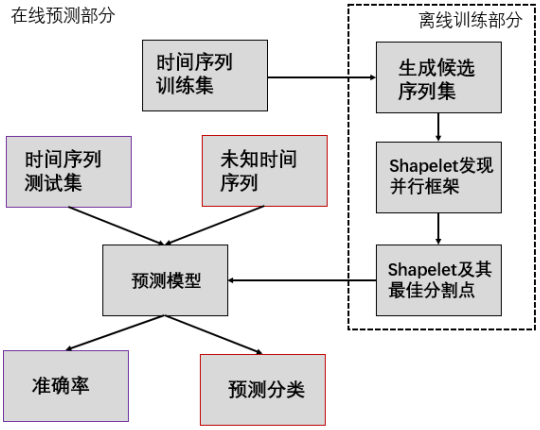

图 3.1　Shapelet 总体方案

离线训练部分：输入时间序列训练集，根据训练集产生候选序列集，然后在候选序列集上执行 Shapelet 并行发现过程，将分类能力最高的候选序列及其对应的最佳分割点作为模型输出。模块训练的过程即 Shapelet 发现的过程非常耗时，模块训练是并行总体方案的重点，本文主要针对这一部分展开叙述的。

在线预测部分：将未知时间序列输入达到预测模型中，预测模型输出其对应的类标。这一部分对于时间消耗非常少，因此不需要进行并行化。

因为离线训练部分 Shapelet 发现过程非常消耗时间，我们设计了 CPU 和 GPU 联合执行的 Shapelet 发现并行框架，其中 CPU 负责调度（负责少量数据计算），

GPU 负责执行数据计算，章节 3.1.1主要介绍发现过程的整体框架。发现过程方案中的多个模块对于 Shapelet 发现过程的三阶段功能进行了实现，章节 3.1.2主要介绍各个模块的功能。因为数据集和距离参数的不同，Shapelet 发现过程在同一阶段的执行中会选择不同的模块，章节 3.1.3介绍了数据流向如何根据数据集和距离参数进行选择。

3.1.1　Shapelet 发现并行框架

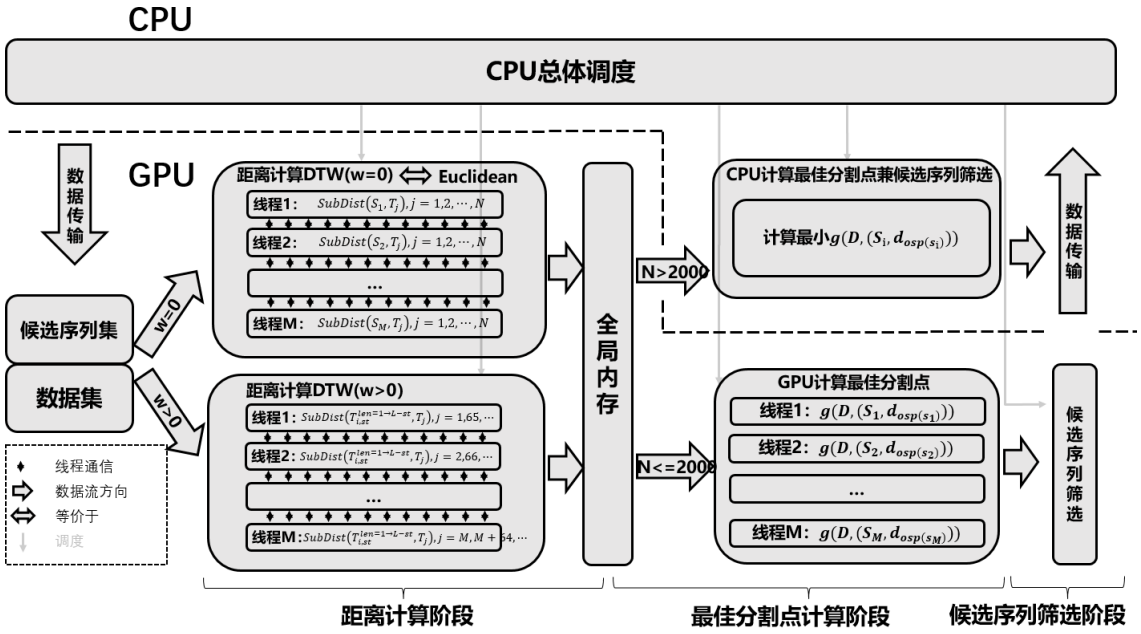

图 3.2　Shapelet 发现并行框架

图 3.2基于 DTW 距离度量的 Shapelet 并行算法发现过程的并行框架图。输入数据是数据集和候选序列集，其中候选序列集是对于时间序列的所有子序列枚举获得。输出数据是一个具有高分类能力的候选序列及其最佳分割点等作为预测模型。框架部分包括六个模块：w>0 距离计算模块、w=0 距离计算模块、GPU 最佳分割点计算模块、CPU 最佳分割点计算模块、候选序列筛选模块、全局内存（中间变量存储），这六个模块分别对于 Shapelet 发现过程的三阶段进行实现，各模块的功能会在章节 3.1.2介绍。三个阶段的功能如下：

1. 距离计算阶段：输入候选序列集 S 和数据集 D，输出多个 $\mathcal{F}$ 并且存储在全局内存中 (见定义 2.5，公式 2-4)，其中包括 w>0 距离计算模块和 w=0 距离计算模块。

2. 最佳分割点计算阶段：输入多个 $\mathcal{F}$（从全局内存读入），输出信息增益 $g(D,(S,d_{osp(S)}))$ 以及对应的 S,$d_{osp(S)}$，包括 CPU 最佳分割点计算模块和 GPU 最佳

分割点计算模块。

3. 候选序列筛选阶段: 输入多个候选序列 S 以及对应的信息增益 $g(D,(S,d_{osp(S)}))$, 最佳分割点 $d_{osp(S)}$, 输出具有最大信息增益的 S 以及对应的 $d_{osp(S)}$, 实现模块为候选序列筛选模块。

Shapelet 发现过程根据不同的数据集和不同的扭曲匹配程度选择不同的执行模块，如何进行执行路径的选择会在章节 3.1.3介绍。

3.1.2 并行框架各模块功能

本章节主要介绍并行框架中的六个模块：w>0 距离计算、w=0 距离计算、GPU 最佳分割点计算、CPU 最佳分割点计算、候选序列筛选、中间变量存储（全局内存）。

中间变量存储模块（全局内存）：全局内存承担距离计算阶段和最佳分割点计算阶段的中间媒介，距离计算阶段将距离计算结果 $\mathcal{F}$ 存入全局内存，最佳分割点计算阶段再从全局内存读取 $\mathcal{F}$ 来计算最佳分割点，其中距离计算阶段和最佳分割点计算阶段都使用合并内存访问 Coalesced 的方法，从而减少全局内存访存延时。

w>0 距离计算模块：距离度量采用 $DTW(A,B,w)$，每个线程主要负责 T_i 中以 s 位置为起始的所有候选序列 $\left\{T_{i,s}^{len}, len=1\rightarrow L-s\right\}$ 和数据集 D 中几个时间序列 T_j 的距离计算，然后将其存入全局内存相应位置，如公式 3-1，具体实现细节会在章节 4.1介绍。

$$\begin{cases} \left\{SubDist(T_{i,s}^{len}, T_j)\right\} \\ \text{subject to:} \\ \qquad len = 1 \rightarrow (L-s) \\ \qquad j = \{tid_x, tid_x + dim_{tidx}, tid + 2 * dim_{tidx}\} \end{cases} \tag{3-1}$$

w=0 距离计算模块：距离度量采用 $Euclid(A,B)$，每个线程执行一个候选序列 S 相对于数据集 D 所有时间序列 T_j 计算距离 $\mathcal{F}=SubDist(S,T_j), j=1\rightarrow N$，最后将 $\mathcal{F}$ 存入全局内存相应位置。某个线程块 Block 负责一个时间序列的长度为 len 的子序列，比如线程块 $Block(i,len)$ 计算候选序列就是 $T_{i,s}^{len}, s=1,2,\cdots,L-len+1$。$w=0$ 距离计算模块算法中线程存在依赖关系，需要线程之间进行协作计算，详见章节 4.2。相比于 w>0 距离计算模块，本模块减少不必要的访存过程和比较过程。当然算法也是有缺陷的，时间序列长度 L 不能超过 1024，因为线程之间需要通信才能完成依赖计算，GTX 1080 一个线程块 Block 中最多允许 1024 个线程，超过 1024 个线程的依赖关系不能建立。

GPU 最佳分割点计算模块：从全局内存读取 $\mathcal{F}$，然后并行计算候选序列对应的最佳分割点 d_{osp}，这里使用了启发式算法计算最佳分割点，详细设计见章节 4.3。本模块会根据不同 N 大小使用不同的共享内存及寄存器改变相应的 $grid$ 和 $block$ 参数，使尽可能多的线程同时并行计算。

CPU 最佳分割点计算模块：采用的算法和 GPU 最佳分割点计算模块的算法是相同的，不同的是在 CPU 中执行。当 N 增大到一定程度时，流处理器簇 SM 上的资源不再足够 32 个线程执行，就会切换到 CPU 执行。

候选序列筛选模块：最佳分割点阶段产生 $O(NL^2)$ 个候选序列 S 及其对应的信息增益 $g(D,(S,d_{osp(S)}))$ 和分割点 $d_{dsp(S)}$，在本模块中需要选出具有最大信息增益的候选序列及其对应的分割点。本模块的本质是一个特大向量求最大值的过程，是一个典型的归约过程，可以采用并行归约的算法求最大信息增益，时间复杂度为 $O(\log(N)+\log(L))$。我们这里并没有直接使用归约算法，而是将原始归约算法[33]的求最大值过程改造成求最大值索引过程，这样能够避免候选序列和分割点跟随信息增益一起归约。因为并行求最大信息增益过程总的时间占比不超过 1%，本文不做过多的介绍。另外，正因为候选序列筛选模块是一个归约过程，为了保持高效率，使用独立的并行计算完成。

3.1.3　并行框架执行路径的选择

w 的取值是控制扭曲匹配程度的参数，当 $w > 0$ 时，距离计算阶段采用的距离度量是 $DTW(A,B,w)$；而当 $w = 0$ 时，为了避免不必要的计算，距离计算阶段采用 $DTW(A,B,w)$ 距离度量的等价方案 $Euclid(A,B)$。因此，距离计算阶段会根据 w 的不同分为不同的模块：w>0 距离计算模块和 w=0 距离计算模块。另外，一个线程块 Block 中最多出现 1024 个线程，$w = 0$ 距离计算模块允许最长的时间序列长度 L 为 1024（线程之间需要协作）。

因此，当 $w > 0$ 或 $L > 1024$ 时，执行 w>0 距离计算模块；而当 $w = 0$ 且 $L <= 1024$ 时，执行 w=0 距离计算模块。

GPU 上多线程计算最佳分割点，每个线程需要 $O(N)$ 的空间，但是硬件环境每个线程块 Block 只能提供 48K 的共享内存和 64K 的寄存器大小（见表 5.2），可能会出现内存不足或者过大内存需求导致每个流处理器簇 SM 上运行的线程束 Warp 个数减小。虽然通过调节线程块 Block 的参数能够一定程度上缓解这种情况，但是依然存在这样一个阈值，当 N 大于这个阈值时，GPU 将不能提供足够的资源进行最佳分割点并行计算。

因此，设定一个阈值 $N_{th} = 2000$（根据一个 $block$ 执行 32 个线程所需资源

计算），当 $N \leq N_{th}$ 时，最佳分割点计算时，执行 GPU 最佳分割点计算模块；当 $N > N_{th}$ 时，执行 CPU 最佳分割点计算模块，其中 CPU 最佳分割点模块的算法和 GPU 最佳分割点模块相同。

对于 w>0 距离计算模块、w=0 距离计算模块、最佳分割点计算模块，会估计不同数据集即不同 w, N, L 参数下每个线程块 Block 中共享内存和寄存器的使用情况。本文对于不同的 w, N, L 设置了资源使用上线，会根据不同的参数调节线程块 Block 和计算网格 Grid 参数，避免单个 Warp 占用资源太多，而影响流处理器 SM 上同时执行的线程束 Warp 个数，这样可以使有限的资源上进行更多的计算。

3.2　并行化需要解决的问题

本章节主要介绍设计并行化之前需要解决的问题，包括中间变量 $\mathcal{F}$ 的存储问题、特定需求的矩阵转置问题、最佳分割点计算的时间复杂度问题。

3.2.1　中间变量 $\mathcal{F}$ 的存储问题

$\mathcal{F}$ 是计算 Shapelet 中计算距离阶段和最佳分割点阶段的中间变量，空间复杂度为 $O(N)$，对于整个候选序列集而言，$\mathcal{F}$ 存储的空间复杂度为 $O(N^2L^2)$，所需要的空间大小取决于数据集的大小，根据不同的数据集大小，有不同的方案，具体如下：

当数据集大小及时间序列长度 N, L 很小的时候，我们采用寄存器存储 $\mathcal{F}$，不会涉及到访存延时，计算速度会很快。

当 N, L 增大时，寄存器不见不能满足存储要求，系统会逐渐使用局部内存（局部内存实际上占用全局内存，和全局存储速度一致，但是又没有使用合并内存访问，会有很大的访存延时），因此各线程会出现频繁的访存延时而使性能非常差。因为有限的寄存器和共享内存都不能满足 $\mathcal{F}$ 存放的要求，这里我们考虑将 $\mathcal{F}$ 放在全局内存中存储。使用全局内存保证 $\mathcal{F}$ 的存储空间，为了保证执行效率，距离计算阶段必须通过合并内存访问 Coalesced 的方法将 $\mathcal{F}$ 存储到全局内存，最佳分割点计算阶段必须通过合并内存访问 Coalesced 的方法从全局内存读取 $\mathcal{F}$。

将 $\mathcal{F}$ 放在全局内存中存储，只能一定程度上解决资源不足的问题。当 N, L 继续增大时，全局内存依旧不能满足 $\mathcal{F}$ 的资源需求，全局内存肯定不能提供平方级别的空间需求，$O(N^2L^2)$ 的空间复杂度会使 $\mathcal{F}$ 需要的内存很快超过全局内存大小。对于全局内存不能中间结果 $\mathcal{F}$ 空间不足的情况，这里使用换入换出[38] 的思路去解决，将整个候选序列集分为多个子集，每次对于一个候选序列子集在 GPU 中进行 Shapelet 并行发现过程，再在 CPU 上综合多次计算的结果，同时使用合并内存

访问和延时隐藏保证效率，具体如图 3.3。当然有多块 GPU 可供使用时，也可以多个 GPU 同时并行多个候选序列子集，然后再统一归约。这里涉及的换入换出、数据在主机和 GPU 之间拷贝，不再这里详述。

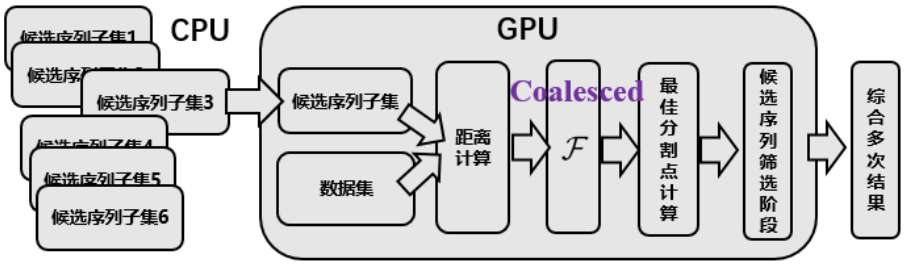

图 3.3　中间变量 $\mathcal{F}$ 存储空间需求过大的解决方案

这里解释一下为什么从 $\mathcal{F}$ 中间变量处分为距离计算和最佳分割点计算两个阶段，主要三个原因：

1. 距离计算阶段和最佳分割点计算阶段是不同性质的计算，将两者分开更有利于保持各线程均衡，有利于资源的充分利用；

2. 在距离计算阶段，后文根据 w 的不同分为 $w > 0$ 距离计算模块和 $w = 0$ 距离计算模块，两者访问 $\mathcal{F}$ 的方式不同；

3. 从 $\mathcal{F}$ 处分开有利于算法实现，使每个 *kernel* 的功能都比较清晰。

3.2.2　特定需求的矩阵转置

在 $w = 0$ 距离计算模块中，每个线程块 Block 都会产生一个矩阵，我们需要将每一个矩阵中的列向量变成行向量，这时候就需要采用矩阵转置过程。但是这里我们不能使用 CUBLAS[39] 中库或者或现成算法，原因有两点：

第一：如图 3.4，每一个线程块 Block 产生一个 $M * N$ 的矩阵都存储在全局内存一个 $C * R$ 的一个块中，矩阵并没有占据整个块的空间；每个线程块 Block 产生矩阵列数 M 不一致，但是 C, R 的大小是根据最大的 M, N 确定，而 M 的差别有可能很大，如图 3.4中的 $Block_1$ 和 $Block_2$。调用算法的矩阵转置都是按照 $C * R$ 的大小进行转置的，有可能造成大量计算资源浪费。

第二：我们这里要求转置之后的矩阵行与行之间是拼接起来的，如图 3.4右侧部分。如果调用现成算法的矩阵转置之后的每个矩阵之间都会出现空白，如果需要按行拼接起来，必须再进行一次全局内存拷贝过程，这样同样违背我们加速的目的。

我们需要设计一个矩阵转置算法能够做到资源合理利用，可以从以下两个角度进行：

1. 对于不同大小的矩阵，根据矩阵的实际大小进行转置，转置的大小不能 $(M+32, N+32)$；

2. 计算每个矩阵转置拼接之后的偏移和行数，方便直接转置到偏移位置。

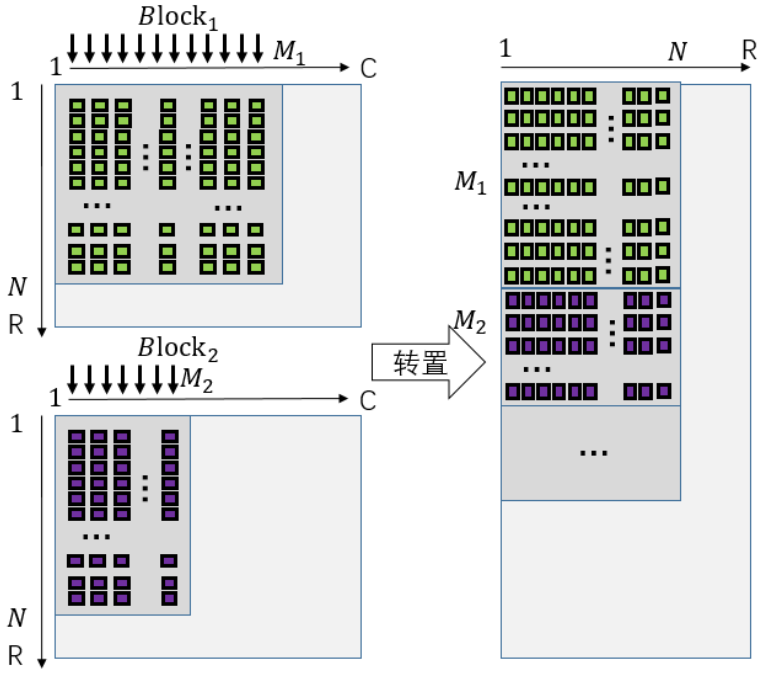

图 3.4　特定矩阵转置要求

3.2.3　最佳分割点计算的时间复杂度问题

距离计算阶段的时间复杂度为 $O(N^2L^4)$，距离计算是 Shapelet 发现复杂度最高的计算。但是距离计算阶段的时间复杂度可以降到 $O(N^2L^3)$，最佳分割点计算阶段的时间复杂度为 $O(N^2L^2\log(N))$。

当数据集 D 很大时，即 N 很大时，最佳分割点阶段有可能成为 Shapelet 发现过程占比时间最大的计算。我们需要降低最佳分割点计算的时间复杂度，从而降低执行时间。

传统的最佳分割点计算，对于所有可能的阈值 d_{th} 都进行了尝试，如 3-2。一般会将 $\mathcal{F}$ 按照距离进行排序（排序时间复杂度 $O(N\log(N))$）之后，以每两个距离之间的值作为阈值 d_{th} 进行尝试，这是以一种遍历的方法寻找全局最优值。

$$g(D,(S,d_{osp(S)})) \geq g(D,(S,d_{th})), \forall d_{th} \in \mathbb{R}_+ \tag{3-2}$$

但是我们的目的不是对于距离进行排序，我们是需要找一个阈值 d_{th}，将数据集 D 中的两类尽可能分开。这里可以采用一种启发式的搜索方法，搜索满足条件

的阈值。因为候选序列非常多，可以使用不同的搜索方式来使其中一种候选序列产生的阈值尽量靠近全局最优值，具体的方案将在章节 4.3中介绍。

3.3　本章小结

本章首先从应用角度介绍了基于 DTW 距离度量的并行 Shapelet 的并行总体方案，包括在线预测部分和离线训练部分。因为离线训练部分比较耗时，我们提出了 Shapelet 发现的并行框架，并对于框架内的各个模块功能和如何在模块之间进行数据流向的选择进行了介绍。然后本章还对于并行化之前需要解决的问题进行了详细叙述。

第 4 章　基于 DTW 距离度量的 Shapelet 并行算法设计

第三章主要介绍了基于 DTW 距离度量的 Shapelet 算法并行总体方案和流程以及并行化需要解决问题。本章主要对于各模块的并行算法进行详细介绍，主要包括 w>0 距离计算模块并行方案、w=0 距离计算模块并行方案、GPU 最佳分割点计算模块并行方案。其他三个模块不在本章介绍，中间变量的存储在 3.2.1已经介绍；CPU 最佳分割点计算模块采用和 GPU 最佳分割点计算采用相同的算法，只是在 CPU 中执行；候选序列筛选模块是一个典型的规约过程，可以参考文献[33] 中规约过程。

4.1　w>0 距离计算模块并行方案

本章主要介绍 w>0 距离计算模块并行所使用的并行策略、并行算法设计、以及实现细节和性能考虑。在这一章讨论的都是基于 $DTW(A, B, w)$ 距离讨论的。

4.1.1　并行策略

这一部分主要从重复性和依赖性两个角度介绍 w>0 距离计算模块并行策略。通过“重用”策略降低该模块的算法时间复杂度和空间复杂度，通过减少依赖性使该模块已于并行。

首先，讨论一下 w>0 距离计算模块存在的重复计算。在 w> 距离计算模块中，对于某个候选序列 $S = T_{i,s}^{len}$，都会存在距离 $\mathcal{F}$ 的计算，如公式 4-1，其中的最基本运算是 $DTW(T_{i,s}^{len}, T_{j,p}^{len}, w)$。但是在整个 w>0 距离计算模块即对于所有候选序列 $T_{i,s}^{len}$ 而言，数据集 D 中任何两个长度相等的子序列 $T_{i,s}^{len}$ 和 $T_{j,p}^{len}$ 都会计算发生 DTW 距离计算过程，公式 4-2是所有距离计算的集合。数据集 D 中每两个长度相等的子序列都存在距离计算，因此 w>0 距离计算模块存在重复计算。

$$\begin{aligned}\mathcal{F} &= \left\{SubDist(S, T_j), j = 1, 2, \cdots, N\right\} \\ &= \left\{\min_{p=1\rightarrow L-len+1}(DTW(T_{i,s}^{len}, T_{j,p}^{len}, w)), j = 1, \cdots, N\right\}\end{aligned} \tag{4-1}$$

$$\begin{cases} \left\{DTW(T_{i,s}^{len}, T_{j,p}^{len}, w)\right\} \\ \text{subject to:} \\ \quad len = 1 \rightarrow L \\ \quad i = 1 \rightarrow N, j = 1 \rightarrow N \\ \quad s = 1 \rightarrow L - len + 1, p = 1 \rightarrow L - len + 1 \end{cases} \tag{4-2}$$

然后分析如何利用重复性降低该模块算法的时间复杂度? 这里使用这里使用 $A_{1:len} = T_{i,s}^{len}$ 来代替一组起点位置相同的候选序列 $T_{i,s}^{len}, len = 1 \rightarrow M$，使用 $B_{1:len} = T_{j,p}^{len}$ 来代替 $T_{j,p}^{len}, len = 1 \rightarrow M$ 来讨论重复计算。如图 4.1所示，在 $DTW(A_{1:M}, B_{1:M}, w)$ 的动态计算过程中可以囊括 M 个距离计算 $DTW(A_{1:m}, B_{1:m}, w), m = 1, 2, \cdots, M$。$DTW(A_{1:m+1}, B_{1:m+1}, w)$ 距离计算的结果可以视为在 $DTW(A_{1:m}, B_{1:m}, w)$ 的基础上完成，时间复杂度均摊到每一对距离计算 $DTW(A_{1:m}, B_{1:m}, w)$ 为 $O(w)$ 而不是 $O(wL)$ 或 $O(wM)$ 的时间复杂度。候选序列 S 计算 $\mathcal{F}$ 的过程包含 $O(NL)$ 个 $DTW(T_{i,s}^{len}, T_{j,p}^{len}, w)$ 计算，因此 w>0 距离计算模块 $\mathcal{F}$ 的均摊时间复杂度为 $O(wNL)$。在数据集 D 所有候选序列集合 $SubSet(D)$ 大小为 $O(NL^2)$，因此使用重用策略时，基于 DTW 距离度量的 Shapelet 算法时间复杂度为 $O(NL^2 * wNL) = O(wN^2L^3)$。而原始基于 DTW 度量的 Shapelet 发现算法时间复杂度为 $O(wN^2L^4)$，通过重用策略将整个算法的时间复杂度 L 倍，从而降低执行时间。算法 4.1是使用重用策略下基于 DTW 距离度量的 Shapelet 串行算法。

并行过程能否顺利执行和 DTW 计算的空间复杂度有很大关系，在并行计算中，DTW 计算需要占据共享内存空间，使用过多的资源有可能使并行线程个数降低，因此 DTW 计算的空间复杂度必须降低。如何降低 DTW 计算的空间复杂度，$DTW(A_{1:m+1}, B_{1:m+1}, w)$ 都是依赖 $DTW(A_{1:m}, B_{1:m}, w)$ 的计算结果而来的，而 $DTW(A_{1:m+1}, B_{1:m+1}, w)$ 的计算和 $DTW(A_{1:m}, B_{1:m}, w)$ 之前的代价矩阵元素没有关系，因此计算时只需要保留和 $DTW(A_{1:m}, B_{1:m}, w)$ 之间的代价矩阵部分元素即可，如图 4.1中并行依赖计算部分，因此空间复杂度为 $O(w)$，这样可以大大减少共享内存的使用。

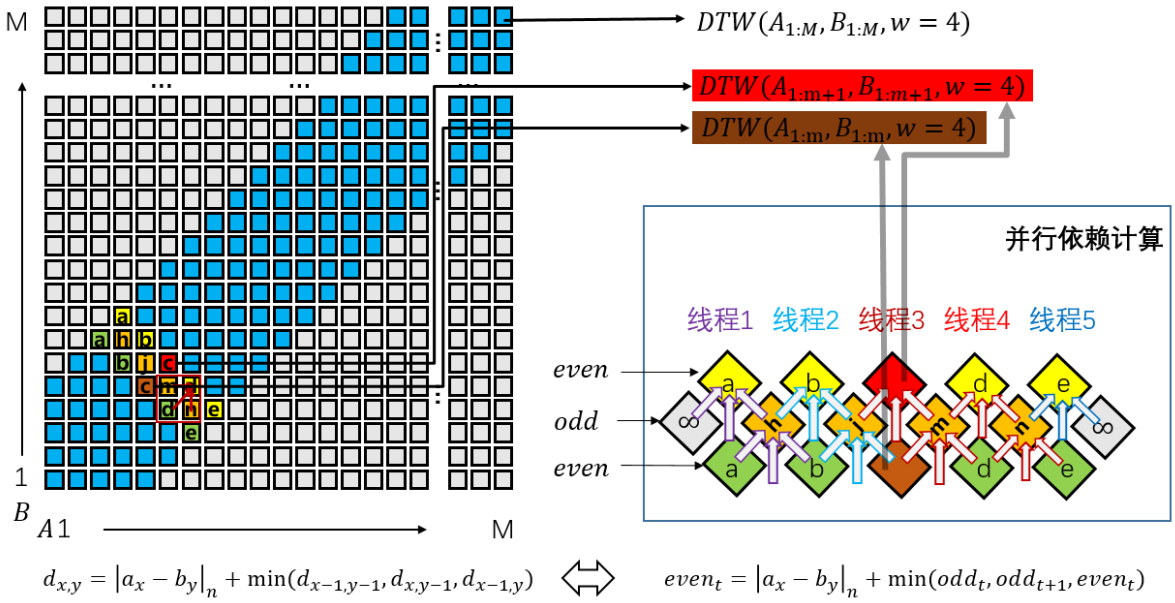

图 4.1　w>0 距离计算模块的重复性和依赖性分析

算法 4.1 基于 DTW 距离度量的 Shapelet 串行算法 (使用重用策略之后)

function SHAPELETALG(D)
　$lastS \leftarrow \phi, lastinfogain \leftarrow 0, lastdosp \leftarrow 0, leftisAorB = A$
　for all $S \in \{T_{i,s}^{L-s}, i = 1, 2, \cdots, N, s = 1, 2, \cdots, L\}$ **do**
　　for $i = 1$ to $|S|$ **do**
　　if $1 = i$ **then**
　　计算 $SubDist(S_{1:1}, T_j, j = 1, 2, \cdots, N)$ 获得 $\mathcal{F}_{S(1:1)}$
　　else
　　在 $\mathcal{F}_{S(1:i-1)}$ 的基础上计算 $\mathcal{F}_{S(1:i)}$
　　end if
　　计算 $g(D, (S, d_{osp(S(1:i))}))$
　　if $g(D, (S, d_{osp(S(1:i))})) > lastinfogain$ **then**
　　更新 $lastinfogain$,$lastS$,$lastdosp$,$leftisAorB$
　　end if
　　end for
　end for
　return $lastinfogain$,$lastS$,$lastdosp$,$leftisAorB$
end function

最后从依赖性角度来叙述一下如何使用多线程对于 $DTW(A_{1:m+1}, B_{1:m+1}, w)$ 进行并行计算。将从 $DTW(A_{1:m+1}, B_{1:m+1}, w)$ 到 $DTW(A_{1:m}, B_{1:m}, w)$ 之间计算过程放大，如图 4.1右侧并行依赖部分，分别 $even$ 和 odd 表示偶数和奇数对角线的元素，可以使原本代价矩阵中的递归方程将 $d_{x,y} = |a_x - b_y|_n + \min(d_{x,y}, d_{x-1,y}, d_{x,y-1})$ 横平竖直的更新方式变为公式 4-3方程斜对角交替更新方式，这样大大地降低了线程之间的依赖从而降低执行时间。这样就可以通过 $even$ 和 odd 交替更新完成整个代价矩阵上的元素 $D_{x,y}$ 的更新，而且可以通过多并行线程完成。至于 t 和 x, y 的关系可以通过一定的对应方式计算，这里不做过多介绍。这里并行计算 $DTW(A, B, w)$ 的线程个数为 $w + 1$ 的公约数 $y, s.t. mod(w + 1, y) = 0$ 为宜，因为并行依赖区域的宽度最大为 $w+1$，使用 y 个线程并行计算 $D_{x,y}$ 可以保证多个线程的执行内容是均衡的，不会造成计算资源的浪费。

$$D_{x,y} = \begin{cases} even_t = |a_x - b_y|_n + \min(even_t, odd_t, odd_{t+1}) & \text{if } x + y \text{ is even} \\ odd_{t+1} = |a_x - b_y|_n + \min(odd_{t+1}, even_t, , even_{t+1}) & \text{if } x + y \text{ is odd} \end{cases} \tag{4-3}$$

4.1.2　并行算法设计

有了并行策略介绍之后，然后介绍一下每个线程的内容以及线程使如何进行组织的。这里每个线程负责一组候选序列 $T_{i,s}^{len}, len = 1 \rightarrow L - s$ 和一个或多个时间序列 $T_j, j = \{tid_x, tid_x + dim_{tidx}, tid + 2 * dim_{tidx}\}$ 的距离，如公式 4-4，其中这组候选序列起始位置相同。

$$\begin{cases} \left\{SubDist(T_{i,s}^{len}, T_j)\right\} \\ \text{subject to:} \\ \qquad len = 1 \rightarrow (L - s) \\ \qquad j = \{tid_x, tid_x + dim_{tidx}, tid + 2 * dim_{tidx}\} \end{cases} \tag{4-4}$$

$Block(i, s)$ 中线程 $(tidx,)$ 负责公式 4-4的计算，$Block(i, s)$ 中的所有线程组合承线程块 $Block(i, s)$，负责这组候选序列和所有时间序列 T_j 的距离。多个线程块 $Block(i, s)$ 组成计算网格 $grid$ 完成 w>0 距离计算模块的计算

算法 4.2是关于 $Block(i, s)$ 中线程 $(tidx,)$ 运行过程，下面对于算法进行简单介绍:

Algorithm 4.2 $Block(i, s)$ 中 (tid_x, tid_y) 线程计算 $SubDist(T_{i,s}^{1 \rightarrow (L-s)}, T_j)$

1: **function** KERNEL_COMPUTEDTWSPERBLOCK($T_{i,s}^{1\to(L-s)}, T_j, w$)
2: **for** $j = tid_x; j \le N; t+ = dim_{tidx}$ **do**
3: **for** $p = 0$ to L **do**
4: $odd \leftarrow \{\infty\}$,$even \leftarrow \{\infty\}$,$even[center] \leftarrow 0$
5: **for** $step = 1$ to $longest$ **do**
6: 计算 $tid_y = 0$ 线程的代价矩阵对应元素位置 x_0, y_0
7: **for** $t = tid_y; t \le w + 1; t+ = dim_{tidy}$ **do**
8: $x \leftarrow x_0 + t, y \leftarrow y_0 - t$
9: **if** $x, y \in prunedtwZone$ **then**
10: $even_t = (T_{j,x} - T_{i,y})^2 + \min(even_t, odd_t, odd_{t+1})$
11: **end if**
12: **end for**
13: 将 $even_{center}$ 和对应的 $\mathcal{F}_j$ 比较求出最小值并更新 $\mathcal{F}_j$
14: 计算 $tid_y = 0$ 线程的代价矩阵对应元素位置 x_0, y_0
15: **for** $t = tid; t \le w + 1; t+ = dim_{tid}$ **do**
16: $x \leftarrow x_0 + t, y \leftarrow y_0 - t$
17: **if** $x, y \in prunedtwZone$ **then**
18: $odd_{t+1} = (T_{j,x} - T_{i,y})^2 + \min(odd_{t+1}, even_t, , even_{t+1})$
19: **end if**
20: **end for**
21: **end for**
22: **end for**
23: **end for**
24: **end function**

1.*line* 5循环对应一个 $DTW(T_{i,s}^{len}, T_{j,p}^{len}, w), len = 1 \to longest$ 计算过程；

2.*line* 7-20进行的是通过 dim_{tidy} 个线程完成一次 DTW 并行依赖计算，见图 4.1并行依赖计算；

3.*line* 13进行的 $\min(DTW(S, T_{j,p}^{|S|}, w)), p = 1 \to L - |S| + 1$ 中的一次比较运算。

4.*line* 9中的 *prunedtwZone* 表示代价矩阵限制区域，区域以外的不做更新。

4.1.3　实现细节与性能考虑

前两部分介绍了 Shapelet 并行策略和算法，这一部分就一些实现细节和性能考虑进行了阐述，主要从合并内存访问和存储体冲突两个角度进行介绍：

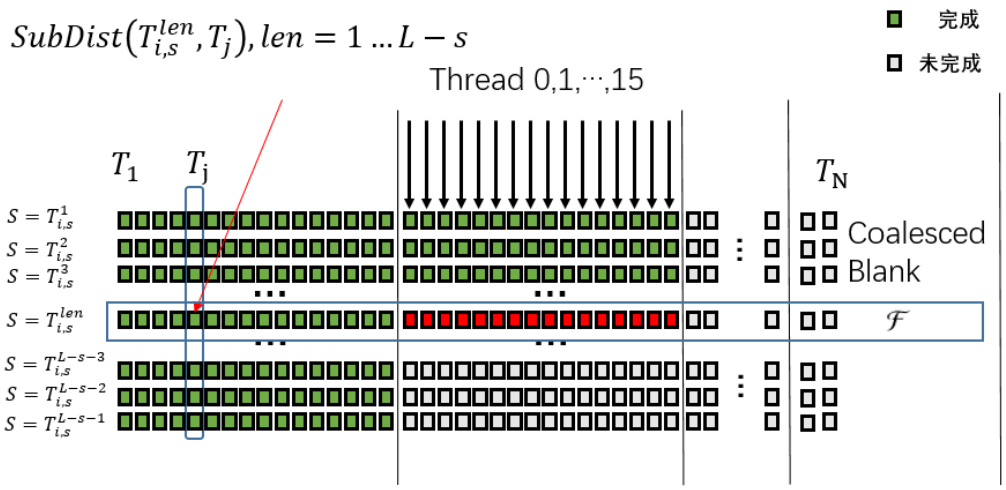

图 4.2　$w > 0$ 距离计算模块中合并内存访问的使用

首先，介绍一下合并内存访问 Coalesced 和基于 DTW 度量的 Shapelet 算法结合。因为在算法 4.2中 *line* 13设计大量的全局内存访存过程，对于一个线程块 Block 来说，需要经历 $O(NL^2)$ 次访存，如果不做任何处理，会造成大量的等待延时，从而影响执行时间。

这里使线程块中的线程负责候选序列和连续的多个时间序列 $T_j, j = m * dim_{tidx} + 1 \rightarrow (m+1) * dim_{tidx}$ 的距离，对于距离 $\mathcal{F}$ 一段连续地址 $m * dim_{tidx} + 1 \rightarrow (m+1) * dim_{tidx}$ 即 $\mathcal{F}_{m*dim_{tidx}+1 \rightarrow (m+1)*dim_{tidx}}$。合并内存访问 Coalesced 可以将这多个线程对于这段连续地址的访问合并成为一个事务，如图 4.2红色部分所示，多个线程正在合并访问 $\mathcal{F}_{17 \rightarrow 32}$，这样可以大幅度降低全局内存访存延时，从而降低执行时间。

然后介绍一下存储体冲突和算法结合使用，算法 4.2中的 *even* 和 *odd* 变量是使用共享内存存储的，*even* 和 *odd* 与多线程配合完成 $DTW(A, B, w)$ 的计算过程。*even* 和 *odd* 在共享内存中占 $2w + 4$ 大小的共享内存，这样容易造成一个存储体被多个线程访问即存储体冲突 Bank-Conflict，当 $w = 6$ 时情况最严重，16 个线程访问同一个存储体，出现 16-way Conflict 现象。这种多个线程访问同一个存储体使得线程访问存储体被大量序列化，从使执行效率大大降低。

我们需要进行存储体冲突的避免操作，这里将连续的两个线程的访问间隔 *stride* 设置为大于实际 $2w + 4$ 的最小奇数，即 $stride = 2w + 5$，可以保证半个线程束 Warp 内的所有线程可以访问到不同存储体。因为 *stride* 为奇数时，和偶数 (16) 没有公约数，在 $stride * 16$ 的地址范围内不可能出现在同一个存储体内，如图 4.3，使用奇数 $stride = 5$, 可以保证 $(5m + n)\%16, m = 0, 1, 2, \cdots, .., n < 5$ 的地址

可以分到 16 个存储体中。

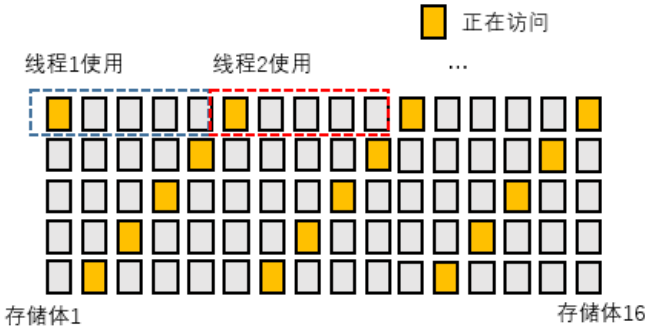

图 4.3　Stride 选择与存储体冲突

4.2　w=0 距离计算模块并行方案

当 $w = 0$ 时，DTW 距离等价于欧氏距离，但是 DTW 距离相比欧氏距离多 $O(L)$ 个比较操作和访存操作（L 为时间序列长度），因此使用 $Euclid(A, B)$ 代替 $DTW(A, B, w)$。本章主要介绍 $w = 0$ 距离计算模块的并行策略、并行算法设计以及实现细节和性能考虑。

4.2.1　并行策略

如主流程图 3.2所示，每个线程需要完成一个候选序列 S 对于数据集 D 中所有时间序列 $T_j, j = 1, 2, \cdots, N$ 计算距离 $SubDist(S, T_j), j = 1, 2, \cdots, N$ 获得 $\mathcal{F}_S$。当某个线程对应的候选序列 S 为 $S = T_{i,s}^{len}$，$SubDist(T_{i,s}^{len}, T_j)$ 包含 $O(L)$ 个欧氏距离 $Euclid(T_{i,s}^{len}, T_{j,p}^{len}), p = 1, 2, \cdots, L - len + 1$ 的计算过程并在取其中的最小值，如图 4.4中的线程 s 及其计算结果。

欧氏距离的时间复杂度 $O(L)$，如公式 4-5，但是线程之间的距离计算存在一定的重复，另一线程的欧式距离 $Euclid(T_{i,s+1}^{len}, T_{j,p+1}^{len})$ 可以在 $Euclid(T_{i,s}^{len}, T_{j,p}^{len})$ 的基础上以 $O(1)$ 的时间复杂度完成，如公式 4-6和图 4.4黄框元素对红框元素的依赖。

$$Euclid(T_{i,s}^{len}, T_{j,p}^{len}) = \sum_{m=0}^{len-1} d(T_{i,s+m}, T_{j,p+m}) \tag{4-5}$$

$$Euclid(T_{i,s+1}^{len}, T_{j,p+1}^{len}) = Euclid(T_{i,s}^{len}, T_{j,p}^{len}) + d(T_{i,s+len}, T_{j,p+len}) - d(T_{i,s}, T_{j,p}) \tag{4-6}$$

如图 4.4，所有的 $Euclid(T_{i,s+1}^{len}, T_{j,p+1}^{len})$ 在满足 $s \geq 1, p \geq 1$ 的条件下都可以在 $Euclid(T_{i,s}^{len}, T_{j,p}^{len})$ 以 $O(1)$ 的时间内通过线程之间的协作完成。但是

$Euclid(T_{i,0}^{len}, T_{j,p}^{len})$ 和 $Euclid(T_{i,0}^{len}, T_{j,0}^{len})$ 没有可以依赖的计算结果，只能通过原始定义 4-5完成。$Euclid(T_{i,0}^{len}, T_{j,p}^{len}), q = 1, 2, \cdots, L - len$ 和 $Euclid(T_{i,s}^{len}, T_{j,0}^{len}), s = 1, 2, \cdots, L - len$ 共有 $2(L - len)$ 个元素，均摊到每个线程有两个元素的计算量，时间复杂度为 $O(L)$。

对于线程 s 来说，$SubDist(T_{i,s}^{len}, T_j)$ 需要完成 $L - len - 1$ 次 4-6的依赖计算和 2 次 4-5次的原始计算，时间复杂度为 $O(L)$。则线程 s 计算 $SubDist(S, T_j), j = 1, 2, \cdots, N$ 获得 $\mathcal{F}_S$ 时间复杂度为 $O(NL)$，对于 w=0 距离计算模块，时间复杂度为 $O(N^2L^3)$。

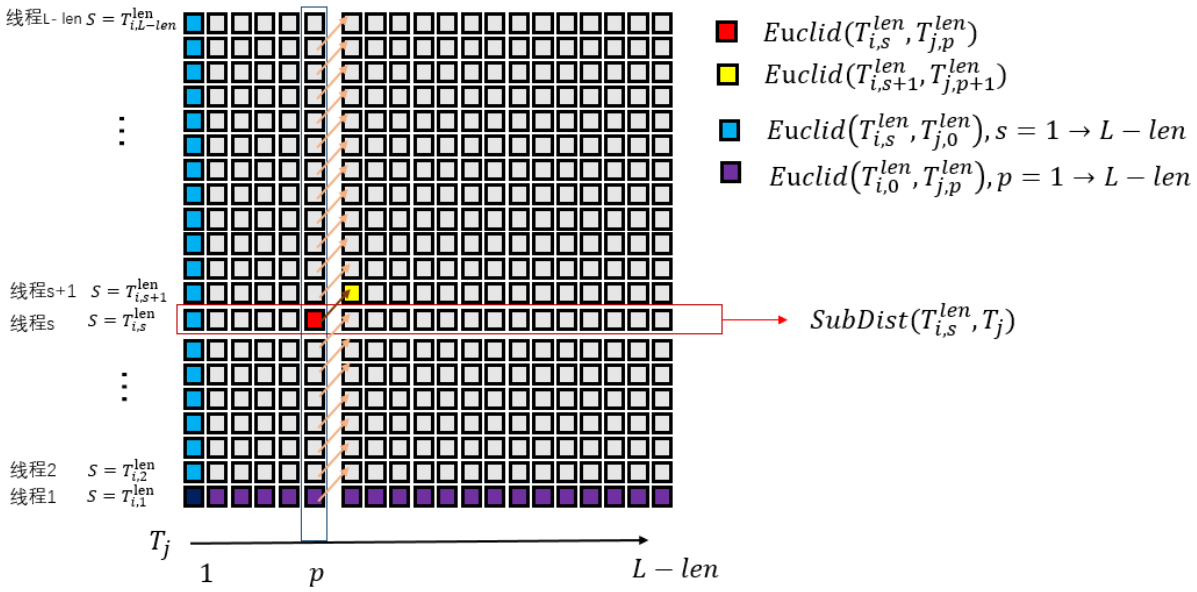

图 4.4　T_i 下长度为 len 的候选序列相对于 T_j 的距离计算 $(w = 0)$

4.2.2　并行算法设计

章节 4.2.1介绍了 $w = 0$ 距离计算的并行原理，本章节将按照原理介绍算法设计部分。

Algorithm 4.3 $Block(i, len)$ 中线程 s 计算 $SubDist(T_{i,s}^{len}, T_j), j = 1, 2, \cdots, L - len$

```
1: function KERNEL_COMPUTEDIST(T_i, len, T_{j=1→N})
2:     for j = 1 to N do
3:         buf0[s] ← 0
4:         buf[s] ← 0
5:         mindis[s] ← 0
6:         for x = 1 to len do
7:             buf0[s] ← buf0[s] + (T_{i,x} − T_{j,s+x})^2
```

8: $buf[s] \leftarrow buf[s] + (T_{i,s+x} - T_{j,x})^2$
9: **end for**
10: $maxdis[s] = buf[s]$
11: **for** $p = 1$ to $L - len + 1$ **do**
12: **if** $1 \neq s$ **then**
13: $buf[s] \leftarrow buf[s-1] + (T_{i,s+len} - T_{j,p+len})^2 - (T_{i,s} - T_{j,p})^2$
14: **end if**
15: **if** $1 = s$ **then**
16: $buf[s] = buf0[p]$
17: **end if**
18: $maxdis[s] \leftarrow min(maxdis[s], buf[s])$
19: **end for**
20: Save $maxdis$ to the jth lines of $matrix$
21: **end for**
22: **end function**

算法 4.3是关于 $w = 0$ 距离计算的并行算法 $Block(i, len)$ 中线程 s 的计算过程，下面关于算法进行介绍：

1.*line* 2的每一个循环对应的都是一个 $SubDist(T_{i,s}^{len}, T_j), j = 1 \rightarrow N$ 计算过程；

2.*line* 3- 9进行的两个欧式距离原始计算；

3.*line* 11循环进行 $L - len$ 次欧氏距离依赖计算。

4.*line* 20是每个 Block 将众多 S 相对 T_j 的距离存储在 $matrix$ 中的一行，$matrix$ 就是需要转置的矩阵。

4.2.3　实现细节与性能考虑

章节 4.2.1和 4.2.2分别介绍了 w=0 距离计算模块的并行原理和算法设计。本章从实现细节和性能考虑角度介绍 w=0 距离计算模块，主要涉及到线程束 Warp 分歧和矩阵转置在并行算法上的应用。

首先，w=0 距离计算模块每个线程执行相同的内容并保持均衡。为了保持各线程之间均衡，原来需要在 *line* 16进行的 $Euclid(T_{i,0}^{len}, T_{j,p}^{len})$ 计算是放在了 *line* 7进行。如果不做这一项优化，线程 0 直接在 *line* 16处直接进行 $Euclid(T_{i,0}^{len}, T_{j,p}^{len})$ 计算，会使线程 0 在整个线程过程中执行 $O(NL)$ 次欧式距离原始计算，使得线程 0 的时间复杂度为 $O(NL^2)$，又因为 Warp 分歧的原因会使线程 0 所在的线程束 Warp 中所有线程都和线程 0 消耗相同的时间，因而执行时间成倍地增加执行时间，当

一个线程块 Block 中只有一个线程束 Warp 的情况下，执行时间会增大 32 倍。

其次，*line* 12和 *line* 15之间需要进行同步。因为 $Euclid(T_{i,s}^{len}, T_{j,p}^{len})$, $s = 1, 2, \cdots, L-len$ 和 $Euclid(T_{i,s}^{len}, T_{j,p+1}^{len})$, $s = 1, 2, \cdots, L-len$ 使用同一段共享内存 buf, $Euclid(T_{i,s+1}^{len}, T_{j,p+1}^{len})$ 都是在 $Euclid(T_{i,s}^{len}, T_{j,p}^{len})$ 基础上计算即 $buf[s]$ 基于 $buf[s-1]$ 计算，所以 $buf[1]$ 的更新必须发生在 $buf[s], s > 1$ 更新之后。

算法 4.3的 *line* 20将多个候选序列相对于时间序列 T_j 的距离存在了对应矩阵 *matrix* 的第 j 行，如图 4.5中 $SubDist(S_m, T_j), m = 1 \rightarrow M$，则某个候选序列 S 对应的 $\mathcal{F}$ 就存在于 *matrix* 中的某一列，如图 4.5中 $\mathcal{F}_{sm}$，$\mathcal{F}$ 对于 GPU 最佳分割点模块不是连续地址，这样不能直接进行最佳分割点计算。

实际上 $\mathcal{F}$ 和 $SubDist(S_m, T_j), m = 1 \rightarrow M$ 处于一个矩阵的行和列，不可能同时存在于连续地址中。为了解决不能同时进行连续地址访问的问题，我们在 $SubDist(S_m, T_j), m = 1 \rightarrow M$ 按行存储和 $\mathcal{F}$ 按行读取之间添加了一个矩阵转置过程，如图 4.5转化过程，这样就能保证 $SubDist(S_m, T_j), m = 1 \rightarrow M$ 和 $\mathcal{F}$ 都能进行连续地址的访问，从而保证了合并存储器的访问。而且矩阵转置可以通过一定的方法使矩阵转置波特率达到全局内存之间数据拷贝的波特率[40]。

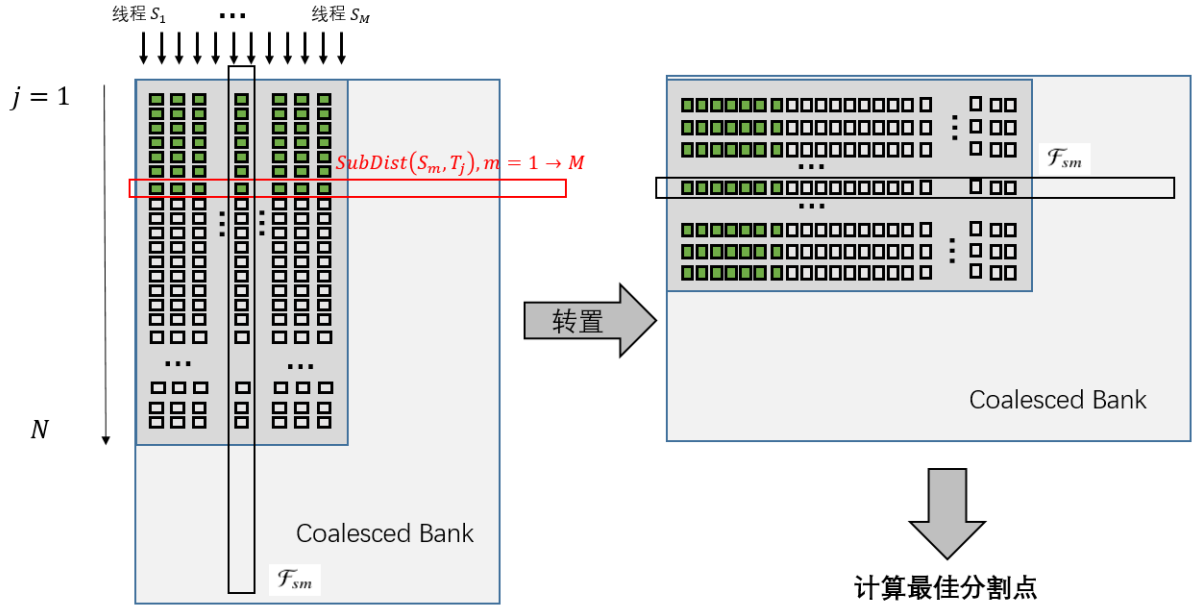

图 4.5　矩阵转置同时满足 w=0 距离计算模块和 GPU 最佳分割点计算模块连续地址访问

这里不能直接使用 CUDA 矩阵转置算法[40]，主要基于两个原因：

一是每个 Block 产生的矩阵大小不一样，主要是因为每个 Block 中线程个数不同，如果统一使用相同的矩阵转置算法会造成计算资源的浪费；

二是 GPU 最佳分割点模块要求 $\mathcal{F}$ 之间是连续存储的，这就要求矩阵之间的

存储不出现空白，但是按照 CUDA 矩阵转置算法调用会出现大量的空白，必须对转置之后的矩阵按行拼接起来，就需要在转置之后再增加一个拷贝调整过程，这样和我们优化效率的初衷不符。

下面是我们如何对于以列形式存储的 $\mathcal{F}$ 变成以行形式存储的方案，需要经过以下步骤：

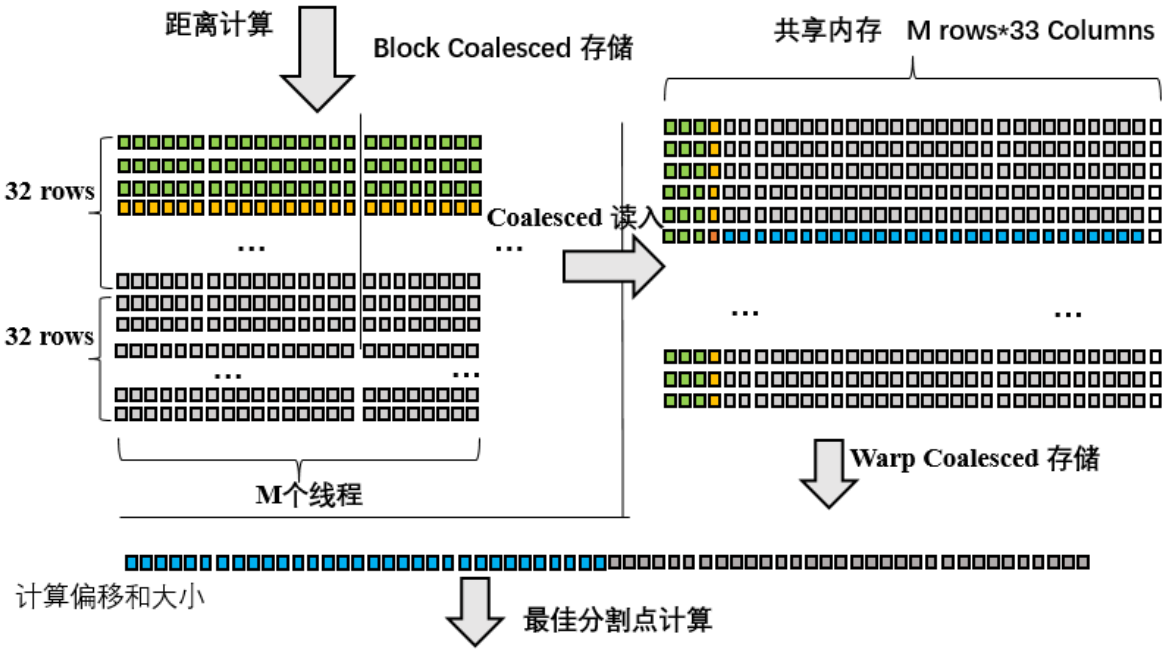

图 4.6　w=0 距离计算模块矩阵转置方法

1. 首先在每个 Block 申请 $M*33$ 的共享内存空间，如图 4.6的红字部分，Block 将原来矩阵中的第一行 $SubDist(S_m, T_1), m = 1, \rightarrow M$ 读入共享内存的第一列（实际上是每 33 地址存一个元素），以此类推，直到存满 32 列共享内存；

2.Block 其中一个 Warp 将共享内存中的第一行存入 S_1 对应 $\mathcal{F}$ 的第一个 128B 空间，另一个 Warp 将第二行存入 S_2 对应 $\mathcal{F}$ 的第一个 128B 空间, 以此类推，对应图 4.6的 Warp Coalesced 过程；

3.1, 2 两步骤存入原始矩阵的前 32 行，按照这样每 32 行进行转置，从而完成整个过程的转置。

上面转置过程中，从全局内存中读入数据和向 $\mathcal{F}$ 某个 128B 地址写入数据都是 Coalesced 过程，将共享内存设置成 33 列是为了避免出现存储体冲突，其中第 33 列元素不存储任何内容。

4.3　GPU 最佳分割点计算模块并行方案

GPU 最佳分割点计算模块是根据距离计算阶段的结果 $\mathcal{F}$，求出一个分割点 $d_{osp(S)}$ 使数据集 D 获得的信息增益 $g(D,(S,d_{osp(S)}))$ 最大。

算法 4.4为最佳分割点计算的原始算法，其中 *line* 2是对于 $\mathcal{F}$ 排序，时间复杂度为 $O(N\log(N))$。Line 4循环是以步进的方式计算获得的最大信息增益以及对应的分割点，时间复杂度为 $O(N)$。因此从候选序列 S 对应的 $\mathcal{F}$ 到最佳分割点计算复杂度为 $O(N\log(N))$，但是对于整个候选序列集合 $SubSet(D)$，时间复杂度度为 $O(N^2L^2\log(N))$。

算法 4.4 最佳分割点计算原始算法

```
1:  function CalcOptimalDividePoint(dist, label, N)
2:      SortByKey(dist, label, key = dist)
3:      g(D,(S,d_th)) ← 0, dividepoint ← 0.0, leftisAOrB ← a
4:      for i = 0 to N − 1 do
5:          计算 lefta, leftb, righta, rightb //O(1)
6:          根据 lefta, leftb, righta, rightb 计算信息增益 temp //O(1)
7:          if temp > g(D,(S,d_th)) then //O(1)
8:              更新 g(D,(S,d_th)), dividepoint, leftisAOrB
9:          end if
10:     end for
11:     return g(D,(S,d_th)), dividepoint, leftisAOrB
12: end function
```

最佳分割点总的时间复杂度为 $O(N^2L^2\log(N))$，仅次于距离计算阶段的 $O(N^2L^3)$(使用重用策略之后)。当数据集的大小 N 增大时，最佳分割点占比越高，从而影响整体执行时间，因此降低最佳分割点计算的时间复杂度。

4.3.1　启发式算法设计

我们需要设计一个算法将最佳分割点计算的时间复杂度从 $O(N\log(N))$ 降到 $O(N)$，在这里，我们使用一种启发式计算分割点的算法来搜索最佳分割点。

图 4.7为子序列 S 相对于数据集 D 中时间序列的距离-类标的集合 $\mathcal{F}$，其中，横轴表示时间序列在数据集中的位置 j，纵轴表示 S 与 j 位置时间序列 T_j 之间的距离 $SubDist(S,T_j)$，不同的颜色表示 T_j 的不同的类标 (A/B)。最佳分割点计算的目的在于寻找一个距离阈值 $d_{th}=d_{osp(S)}$ 使得数据集 D 获得的信息增益 $g(D,(S,d_{th}))$

尽可能大，满足 $g(D,(S,d_{osp(S)}))>g(D,(S,d_{th})),\forall d_{th}\in\mathbb{R}_+$，如图 4.7，并不是对于数据集按照距离 $SubDist(S,T_j)$ 进行排序。但是算法 4.4是对于所有可能产生的信息增益 $g(D,(S,d_{th}))$ 取值都进行了计算，相当于获得了一个全局最优值。

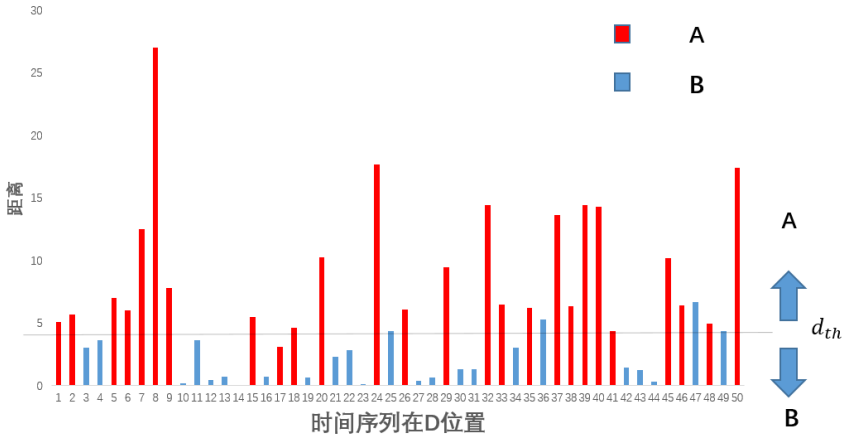

图 4.7　S 对于数据集 D 中时间序列的距离-类标集合,$\mathcal{F}$

相比计算全局最优值，这里提供一种启发式的最优值搜索方式，二分地搜索到一个阈值，使得信息增益最大。首先介绍一下二分搜索地评价指标，根据公式 2-6可知，信息增益 $g(D,(S,d_{th}))$ 最大化等价于条件熵 $H(D|(S,d_{th}))$ 最小化[①]。存在一个点 d_{th} 将数据集 D 分为两个子集 $D_1=\{T,SubDist(S,T)<d_{th}\&T\in D\}$ 和 $D_2=\{T,SubDist(S,T)\geq d_{th}\&T\in D\}$，则条件熵 $H(D|(S,d_{th}))$ 表达式如公式 4-7，为了方便叙述，这里将 $\frac{|D_m|}{|D|}H(D_m),m=1,2$ 定义为 $f(D_m),m=1,2$（对于数据集 D 而言，$|D|$ 为常数）。

$$H(D|(S,d_{th}))=\frac{|D_1|}{|D|}H(D_1)+\frac{|D_2|}{|D|}H(D_2)=f(D_1)+f(D_2) \tag{4-7}$$

如图 4.8，这里将距离计算的最大最小值 (d_{low},d_{high}) 作为分割点的搜索区间。假设在区间 (d_{low},d_{high}) 一个分割点 d_{th1}，将数据集 D 分为 D_1 和 D_2。如果 $f(D_1)>f(D_2)$，表示 D_1 的熵更高一点，可能在区间 (d_{low},d_{th1}) 存在一个阈值 d_{th2}，使得条件熵 $H(D|(S,d_{th_2}))$ 更低一些。这里对于 D 重新进行分割，将临近 D_1 在 (d_{th2},d_{th1}) 地元素归入 D_2 数据集内，可能使 $f(D_1)+f(D_2)$ 更小；反之，(d_{th1},d_{high}) 向相反方向寻找阈值，不断循环下去，呈现一个二分的搜索路线，直到分割点的搜索区间为 0 为止。每一个区间对应数据集 D 一个集合，比如 (d_{low},d_{high}) 对应的整个 $\mathcal{F}$，当区间为 0，即集合为 ϕ。任何一个区间对应的集合中可能存在不止一个元素，我

① 条件熵 $H(D|(S,d_{th}))$ 恒大于 0

们需要选择一个点作为分割点来对 D 进行切割，这里选择集合中最后一个元素作为集合切割的阈值，集合初始是无序的，最后一个元素不一定是最大或者最小的元素。

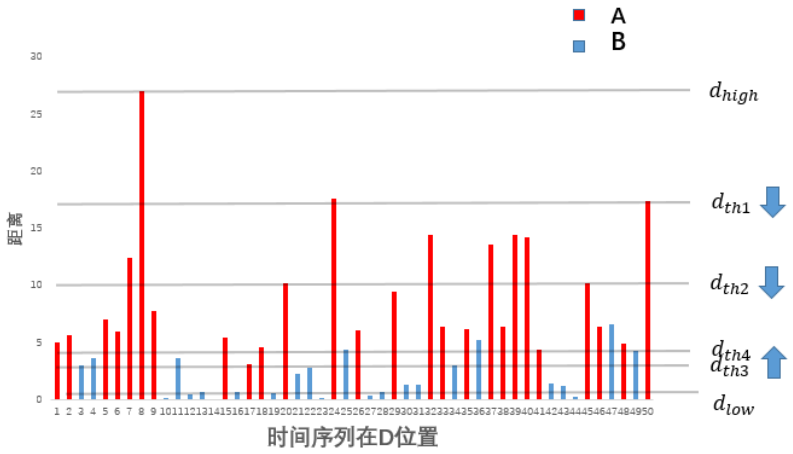

图 4.8　启发式计算最佳分割点

Algorithm 4.5 启发式算法最佳分割点计算算法 *HeuristicOptimalSplitPoint*

function HEURISTICOPTIMALSPLITPOINT($dist, label, N$)
  $left \leftarrow 1, right \leftarrow N$
  while $left < right$ **do**
    $p \leftarrow$ PARTION($dist, label, left, right$)
    $lefta \leftarrow \sum_{i \in BEFORE(p)}(label_i \in A)$
    $leftb \leftarrow \sum_{i \in BEFORE(p)}(label_i \in B)$
    $righta \leftarrow \sum_{i \in BEHIND(p)}(label_i \in A)$
    $rightb \leftarrow \sum_{i \in BEHIND(p)}(label_i \in B)$
    $leftentropy \leftarrow$ ENTROPY($lefta, leftb$)
    $rightentropy \leftarrow$ ENTROPY($righta, rightb$)
    if $leftentropy < rightentropy$ **then**
      $left \leftarrow p + 1$
    else
      $right \leftarrow p - 1$
    end if
    if $lastEntropy < thisEntropy$ **then**
      return $lastEntropy$ 以及对应分割点
    end if
  end while

20: **return** $thisEntropy$ 以及对应分割点
21: **end function**

算法 4.5是启发式求最佳分割点算法：

算法中 $dist$ 中表示候选序列 S 对数据集 D 时间序列的距离数组，即 $\mathcal{F}$ 对应距离部分，$label$ 表示时间序列对应的类标。

line 2为区间初始化，等价于搜索区间 (d_{low}, d_{high})；

line 4是按照区间最后一个元素为切割点进行切割，详见算法 4.6；

line 5- 10为计算 $f(D_1)$ 和 $f(D_2)$ 并在 *line* 11比较并缩小搜索区间，可以在 $O(1)$ 时间复杂度内完成；

line 16表示如果连续两次条件熵呈增大趋势则返回上一次的分割点和条件熵。

算法 4.5可以看出，启发式是使用一种搜索的方式获得最佳分割点，我们可以把这个搜索分割点的过程称为最佳分割点搜索路线。

算法 4.6是启发式最佳分割点计算中需要调用的算法，负责数据集 D 切割成 D_1, D_2 的函数 $PARTION$.

算法 4.6 最佳分割点计算的调用算法

1: **function** PARTION($dist, label, left, right$)
2: $pivot \leftarrow dist[right]$
3: $p \leftarrow left$
4: **for** $q = left$ to $right$ **do**
5: **if** $dist[q] < pivot$ **then**
6: SWAP($dist, p, q$)
7: SWAP($label, p, q$)
8: p ++
9: **end if**
10: **end for**
11: SWAP($dist, p, right$)
12: **return** p
13: **end function**

4.3.2　随机排列 shffule 及其作用

章节 2.1提及 Shapelet 的定义和本质，存在一种模式，数据集中一种时间序列 ($label \in A$) 和这种模式表现出较小的距离，另外一种 ($label \in B$) 则表现出较大的

距离，Shapelet 的本质是寻找这种模式。反之可以认为，与这种模式表现出较小距离的时间序列都存在能够表达这种模式的子序列，并且能够表达这种模式的候选序列不止一个。$S1$、$S2$ 是两个具有高分类能力的候选序列，两者相对于数据集中各时间序列表现的距离很相似。如果使用上述启发式算法，最佳分割点的搜索路线有可能是一致的，最后导致的分类效果是一样的，如图 4.9。

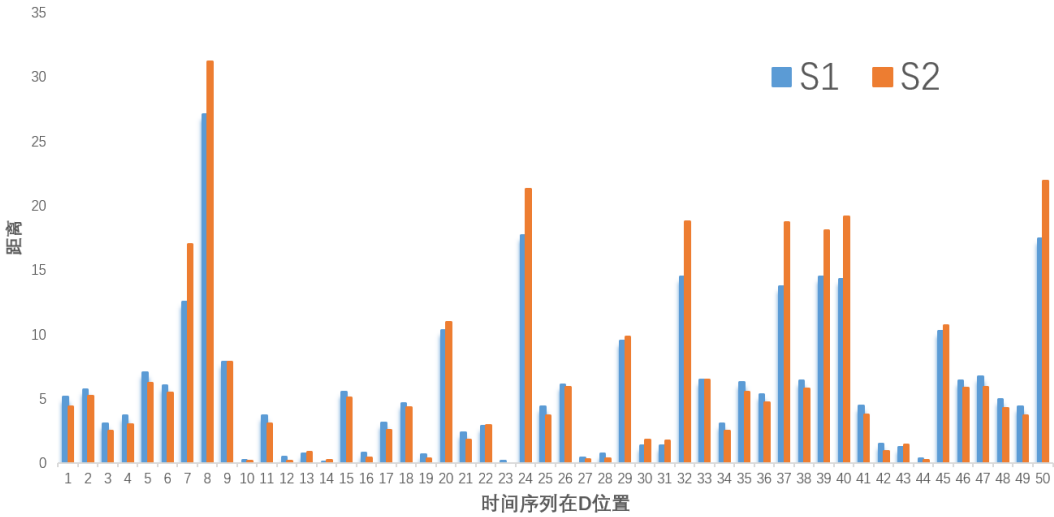

图 4.9　$S1, S2$ 对于数据集 D 中时间序列距离的相似性

启发式算法寻找的不是全局最优值，并不能保证这种唯一的搜索路线就能找最佳分割点。我们需要利用相似性这一特点，使多个候选序列对应的启发式算法尽可能选择不同的最佳分割点搜索路线，目的是多个搜索路线中有某种搜索路线获得的结果和全局最优最接近。如何获得不同最佳分割点搜索路线，关键在于使用哪个元素进行算法 4.6的 *patition*，启发式算法使用集合最后一个元素进行分割，而我们需要局部集合的最后一个元素不是同一个元素来使搜索路线不同，因此在最佳分割点计算之前需要对于 $\mathcal{F}$ 进行随机排列 Shuffle。

为了获得不同的最佳分割点搜索方式，在进行启发式算法搜索最佳分割点之前，对于 $\mathcal{F}$ 进行随机排列 Shuffle。算法 4.7是随机排列算法伪码，本文选择 Kunth shuffle 算法进行随机排列，至于算法的选择原因在章节 4.3.3中结合 GPU 的使用介绍。

4.3.3　实现细节与性能考虑

章节 4.3.1和章节 4.3.2讲述了启发式最佳分割点计算的过程，本章就算法的并行实现细节和性能考虑予以介绍。

章节 4.3.2提及了随机排列 Shuffle 的选择，有多个可以选择的方案：线程束

算法 4.7 对于 $dist$ 和对应的 $label$ 进行 shuffle,$ReservoirShuffle()$

输入:

$dist$: 候选序列 S 对数据集 D 时间序列的距离数组

$label$: 时间序列对应的类标

rng : 一个线程对应的随机数发生器

输出:

$dist$:shuffle 之后的 $dist$

$label$:shuffle 之后的 $label$

```
function ReservoirShuffle(dist, label, N, rng)
    for i = 2 to N do
        j ← rng.random(1, i)
        if i ≠ j then
            swap(dist, i, j)
            swap(label, i, j)
        end if
    end for
    return dist,label
end function
```

Warp 并行随机排列算法、调用 Shuffle device 函数、自实现随机排列函数。其中，线程束 Warp 并行随机排列算法[41] 虽然可以快速的实现一个随机排列算法，但是要求一个 Warp 内所有随机排列 shuffle 结果必须是一致的，不符合章节 4.3.2使用 shuffle 的原因。调用现成的 Shuffle device 函数产生一个随机排列 Index，这里每个线程需要占用 $O(N)$ 的空间，需要较多的 GPU 硬件资源。这里我们选择实现 Kunth shuffle 算法，可以在 $O(N)$ 的时间复杂度下，以 $O(1)$ 的额外空间复杂度完成随机排列 Shuffle，算法按照蓄水池算法的原理，可以使每个元素出现在每个位置的概率为 $O(1/N)$。

算法 4.7需要为每个线程提供不同的随机数发生器 rng，否则，不同线程中使用相同的 rng，最后导致随机数生成的过程是一致，Shuffle 之后的结果依然是一致的。每个线程的随机发生器 rng 需要在进行启发式算法之前随机初始化，并且使用线程 ID 作为随机发生器的初始化参数之一。

最佳分割点计算模块读入 $\mathcal{F}$ 的方法：因为 $\mathcal{F}$ 中的距离在距离模块存储在连续空间中，每个线程读取一个 $\mathcal{F}$ 的距离；$label$ 是通过合并内存访问读取到共享内存中。

4.4　本章小结

本章主要介绍总体方案设计中三处并行算法设计：w>0 距离计算模块并行方案，w=0 距离计算模块并行方案，GPU 最佳分割点计算模块并行方案。w>0 距离计算模块中每个线程负责一系列候选序列和几个时间序列之间的并行距离计算，海量线程组合起来完整整个模块的计算，并且使用合并内存访问和存储体冲突等技术和并行算法结合，加速了并行算法的执行。w=0 距离计算模块通过线程之间协作完成了多个候选序列对于整个数据集中的时间序列并行计算距离，并且通过特定的矩阵转置方法使 w=0 距离计算模块和最佳分割点模块都可以同时进行连续地址访问，大大减少全局内存访存延时。最佳分割点计算模块使用了启发式最佳分割点计算和随机排列的方法完成了最佳分割点的并行计算，两者的结合保证了能够在准确率不降的前提下降低最佳分割点计算的执行时间。

第 5 章　实验设计与分析

本文的目标是验证基于 DTW 距离度量的 Shapelet 并行算法在准确率不降的基础上，获得速度上的提升。本章根据目标设计了一系列针对性的实验，验证 Shapelet 并行算法的有效性，主要包括两部分: 实验设计部分和实验结果分析部分。

5.1　实验设计

实验设计部分首先介绍了运行并行程序的实验环境和数据集，然后针对本文的目标设计了实验方案，最后介绍了我们针对本实验的评价指标。

5.1.1　实验环境

首先介绍本文的实验环境，实验环境分为以下几个部分：

1. 其中硬件环境如表 5.1和表 5.2。

2. 其中软件环境如表 5.3。

3. 开发和调试主要使用 *cuda* 自带的 $cuda-gdb$，$gcc/g++$ 和 *nvcc* 的优化等级都为 $O3$。

表 5.1　硬件环境-主机参数

硬件描述	版本或大小
CPU	i7-7700
内存	16G
主频	3.6GHz

表 5.2　硬件环境-图形处理器参数

硬件描述	版本或大小
GPU 型号	GTX-1080
全局内存	8G
流处理器	20*128
共享内存 (每个 Block)	48K
寄存器 (每个 Block)	64K

表 5.3　软件环境

软件	版本
操作系统	ubuntu 16.04
GUN Make	4.1
gcc/g++	5.4.0
Cuda	8.0
计算能力	6.1
NVCC	8.0.61

5.1.2　实验数据

为了方便和优化工作及并行算法比较，本文使用的 UCR 时间序列分类[42]。本实验选取数据集中的二分类数据作为本实验的实验数据，本文只考虑二分类问题，延展到多分类问题不在本文介绍。表 5.4是关于数据的类标、大小、序列长度等信息的介绍。

表 5.4　实验数据集

数据集名称	类标	训练集大小	测试集大小	时间序列长度
ECGFiveDays	2(1,-1)	23	861	136
Coffee	2(0,1)	28	28	286
Gun-Point	2(1,2)	50	150	150
WormsTwoClass	2(1,2)	77	181	900
ECG200	2(1,-1)	100	100	96
Ham	2(1,2)	109	105	431
Yoga	2(1,2)	300	3000	426
Strawberry	2(1,8)	370	613	235
Wafer	2(1,-1)	1000	6174	152
FordA	2(1,-1)	1320	3601	500

5.1.3　实验方案设计

第三章介绍了并行算法的总体方案以及距离度量选择，因此需要考虑参数 w 对于性能指标的影响，并将本实验和已有优化工作进行指标比较。第四章介绍三个模块的并行算法及优化方法，因此需要对优化方法对对应模块的作用进行评估。基于以上考虑，需要对以下几个方面进行评估：

1.w 发生变化时，对于算法评价指标的影响；

2. 距离计算阶段执行时间与 N, L 变化的关系；

3.$w = 0$ 距离计算模块执行时间与 N, L 变化的关系；

4. 比较最佳分割点计算模块原始算法、启发式最佳分割点计算算法的指标比较。

5. 最佳分割点计算模块执行时间与 N, L 变化的关系；

6. 对于一些重要优化方法，需要指出优化方法对于执行时间的优化效果。

5.1.4　评价方法

本文的主要目的是在保证准确率不降或者微降的基础上尽可能提高执行效率，所以这里只有两个评价指标：准确率和时间。

准确率是将样本的预测 $pred_j, j = 1, 2, \cdots, N$ 和类标 $y_j, j = 1, 2, \cdots, N$ 进行统计，可以表示为 $Aurracy = \sum_{h=1}^{N} 1\left\{pred_j = y_j\right\}/N$, 其中 $1\{\cdot\}$ 为示性函数，表达式为 1{值为真的表达式} = 1。

执行时间的计算和 CPU 统计执行时间略有不同，GPU 统计时间必须等所有 Block 运行完毕之后才能进行统计时间，这里使用 CUDA 的 $nvprof$ 或者 $Events$ 来统计执行时间[41]。

在实际统计执行时间时，没有将申请全局内存和数据集拷入的时间计算在内。因为申请内存和拷贝的时间比较固定，大约 200 ~ 600ms 之间，对于大数据集没有影响，对于小数据集影响很大，为了方便比较，没有将这些时间计入执行时间。

5.2　实验结果分析

实验分析部分首先对于整体执行过程的指标进行比较，然后分别对 w>0 距离计算模块，w=0 距离计算模块，GPU 最佳分割点计算模块的优化方法的效果进行比较。

5.2.1　总体指标比较

表 5.5是**并行执行时间和 CPU 执行时间以及已有优化执行时间的比较**。从表 5.5可以看出，基于 DTW 距离度量的 Shapelet 并行方案相比已有的加速算法有很大的提高，其中相比原始 CPU 算法提高 834 到 3692 倍，平均 1945 倍；相比 Random-Shapelet[18] 提高 21 倍到 51 倍之间，平均 41 倍；相比已有 GPU 并行算法提高 4 倍到 11 倍，平均 6.3 倍。已有优化工作因为各种原因对于 UCR 较大数据集没有涉及，而本文并行算法对于 UCR[42] 中最大的数据集也进行了实验。

下面分析 w **的参数对于准确率/执行时间的影响**,$DTW(A, B, w)$ 距离度量中 w 的参数变化会导致准确率/执行时间的变化，表 5.6中表示不同的数据集中，随着 w

表 5.5　整体执行时间和已有工作的比较

数据集	N	L	CPU(s)	Random-Shapelet,DSAA'15(s)	GPU 并行,ICDM'12(s)	本工作(s)
Coffee	28	286	829.3	8.1^{-}	1.58	0.372
GUN_POINT	50	150	143.6	9.2^{-}	0.65	0.172
ECG200	100	96	151.5	7.6^{-}	1.86	0.155
wafer	1000	152	15471.7^{+}	NA*	299	25.7
FordB	810	500	3013203^{+}	NA*	NA*	816
FordA	1320	500	8002140^{+}	NA*	NA*	2398

-: 使用文献[18] 中采样概率为 1% 的数据

+: 使用 $SubSet(T_1)$ 候选集的执行时间 $*N$ 作为 CPU 执行时间

*: 无对比数据

的变化，准确率/整体执行时间（秒）的变化，其中，$w = 0$ 处使用的是 $Euclid(A, B)$ 的距离度量，表格中是 $w = 0$ 距离计算的执行时间。

表 5.6　评价指标随 w 的变化

数据集	$w = 0(s/\sim)$	$w = 1(s/\sim)$	$w = 2(s/\sim)$	$w = 3(s/\sim)$	$w = 4(s/\sim)$	$w = 5(s/\sim)$
Gun-Point	0.8450/0.205	0.8530/0.837	0.8620/1.098	0.8490/1.343	0.8230/1.576	0.8170/1.781
wafer	0.9528/25.71	0.9530/121.2	0.9531/197.3	0.9529/278.3	0.9531/356.7	0.9532/432.7
ECG200	0.8000/0.592	0.8000/0.951	0.8600/1.249	0.7900/1.489	0.8100/1.758	0.8100/2.000
ECGFiveDays	0.9660/0.136	0.9421/0.235	0.8126/0.347	0.5797/0.409	0.5838/0.528	0.6000/0.607
Coffee	0.8014/0.882	0.7346/1.370	0.6846/1.905	0.6896/2.403	0.6843/2.890	0.7321/3.417
Yoga	0.6437/138.9	0.6397/319.4	0.6462/521.3	0.6442/725.8	0.6387/920.8	0.6427/1124

从表 5.6可以看出随着 w 在 $0 \rightarrow |S|$ 变化时，执行时间呈线性变化，而大部分数据的准确率承呈先升后降，如 GUN_POINT 数据，从 $w = 0$ 的 0.8450 到 $w = 2$ 处取得最大值 0.862，之后准确率逐渐下降。有的数据准确率的最大值就在 $w = 0$ 处，如 ECGFiveDay。有两个原因可以解释这件事，一方面，一些时间序列存在固定的模式匹配，不具有伸缩性质，而这些模式就是进行时间序列分类的关键；另一方面，$DTW(A, B, w)$ 具有对于时间序列平移、伸缩来进行时间序列之间的相似性匹配，但是 Shapelet 算法中候选序列 S 和时间序列 T_j 是以滑动窗口的方式进行距离比较 $Dist(S, T_{j,s}^{|S|}), s = 1, 2, \cdots, L - |S| + 1$ 也具有进行时间序列平移匹配的性质，这样使得 $DTW(A, B, w)$ 平移性质没有得到发挥。

这里我们可以得到一个结论：采用 $DTW(A, B, w)$ 距离度量保证分类准确率大于等于 $Dist(A, B)$ 距离度量的准确率。

并行方案中不同阶段的执行时间及占比，并行方案分为三个阶段：距离计算阶段、最佳分割点计算阶段、候选序列筛选阶段。如表 5.7分别是各个数据集在三个阶段的执行时间及占比。其中，由于数据太多的原因，表 5.7中对于每个数据集只列举了部分 w 下的各阶段执行时间及占比，表中的其他是指除了距离计算阶段和最佳分割点计算阶段之外的其他阶段（包含候选序列筛选计算在内）。

表 5.7　不同阶段执行时间在不同数据集下的表现

数据集	w	总时间 (s)	距离计算 (s)	最佳分割点 (s)	其他 (s)
GUN_POINT	2	1.099	1.039(0.946)	0.044(0.04)	0.015(0.014)
wafer	0	25.717	15.362(0.597)	10.335(0.402)	0.02(0.001)
ECG	4	1.758	1.654(0.941)	0.084(0.048)	0.02(0.011)
ECGFiveDays	2	0.347	0.326(0.938)	0.006(0.016)	0.016(0.045)
Coffee	4	2.89	2.845(0.984)	0.033(0.011)	0.013(0.004)
Ham	3	102.763	101.009(0.983)	1.73(0.017)	0.024(0.000)
Strawberry	1	72.351	68.704(0.95)	3.62(0.050)	0.027(0.000)
Yoga	0	138.949	130.438(0.939)	8.482(0.061)	0.029(0.000)

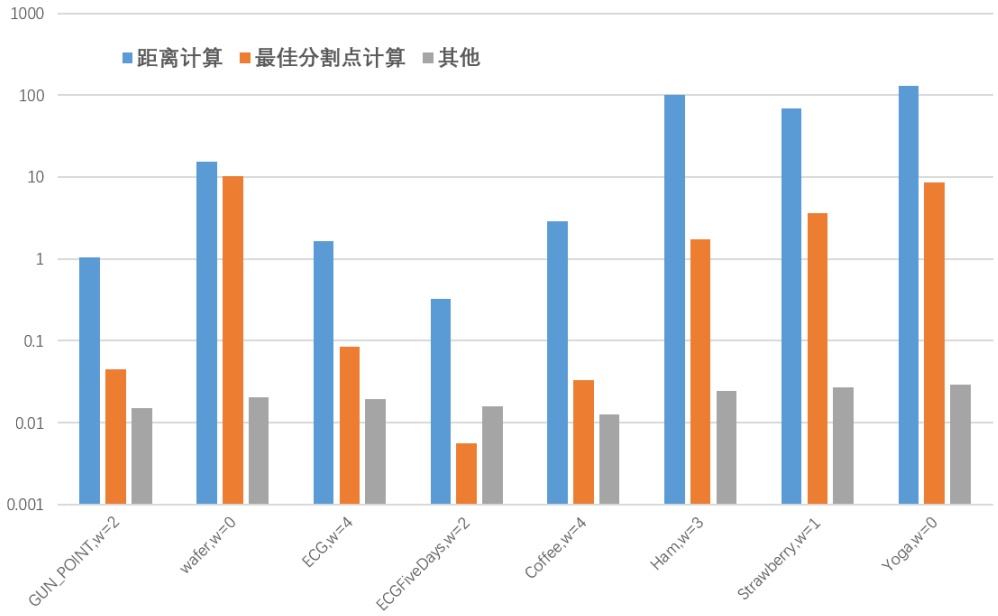

图 5.1　不同阶段执行时间在不同数据集下的表现

图 5.1是表 5.7的图示效果，其中纵轴执行时间采用基数为 10 的对数刻度。从表 5.7和 5.1可以看出距离计算占比最大，其次是最佳分割点计算阶段，包含候选

序列筛选阶段在内的其他阶段占比不超过 1%。而且 N 很大，L 比较小的情况下，距离计算阶段和最佳分割点计算阶段的执行时间比较接近。

各阶段执行时间与 w 的关系，表 5.8是使用数据集 wafer 在不同的 w 计算下各阶段执行时间。使用相同数据集的情况下随着 w 的变化，最佳分割点计算阶段和其他阶段的执行时间基本不变。而距离计算阶段执行时间随着 w 增大不断变化，从图 5.2可以基本呈线性关系，执行时间占比不断增高。

表 5.8　各阶段执行时间与 w 的关系 (wafer 数据集)

w	总时间 (s)	距离计算 (s)	最佳分割点 (s)	其他 (s)
$w = 0$	25.7168	15.3619	10.3345	0.0204
$w = 1$	121.239	110.860	10.3572	0.0218
$w = 2$	197.362	186.967	10.3738	0.0212
$w = 3$	278.350	267.948	10.3812	0.0208
$w = 4$	356.729	346.335	10.3731	0.0209
$w = 5$	432.687	422.306	10.3594	0.0216

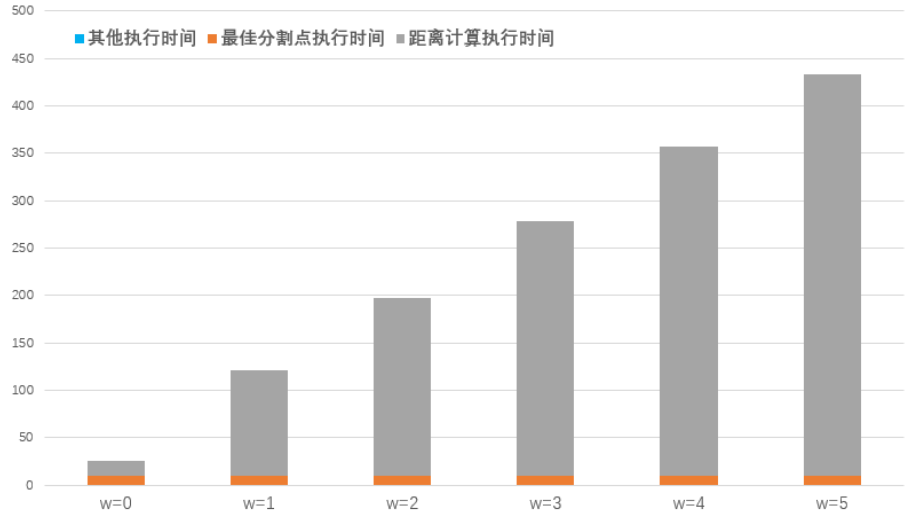

图 5.2　各阶段执行时间与 w 的关系 (wafer 数据集)

5.2.2　w>0 距离计算模块时间分析

在章节 4.1.3中，w>0 距离计算模块主要使用了 Coalesced 合并内存访问和 Bank-Conflict 存储体冲突两个 Cuda 优化技术和算法结合，本章节主要看一下这些优化技术对于 w>0 距离计算模块执行时间的影响。

用合并内存访问对于总体执行时间的影响，表 5.9表示关于在 w>0 距离计算模块是否使用 Coalesced 对于总体执行时间的影响。

表 5.9　是否 Coalesced 的总体执行时间在不同数据集下的表现

数据集	w	Coalesced(ms)	No Coalesced (ms)	时间差百分比
GUN_POINT	w=4	1575.89	1716.1	0.082
GUN_POINT	w=1	837.496	904.68	0.074
ECGFiveDays	w=4	528.163	586.40	0.099
Coffee	w=5	3417.50	3654.5	0.065
Coffee	w=0	882.365	1112.9	0.207
Ham	w=3	102763	104029	0.012

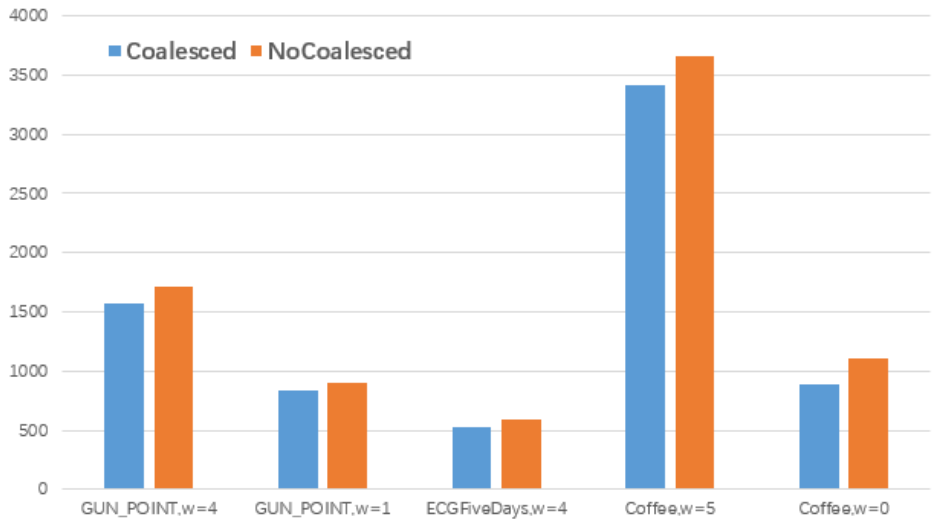

图 5.3　是否 Coalesced 的总体执行时间在不同数据集下的表现

从表 5.9和图 5.3可以看出，使用合并内存访问比普通的多线程连续地址访问效率要高，执行时间要少 1 ~ 20% 之间。

存储体冲突对于执行时间的影响，表 5.10表示是否存在存储体冲突对于距离计算阶段执行时间的影响，这里使用 wafer 数据集对于不同 w 进行测试。其中，$w = 0$ 下的执行时间是基于 $DTW(A, B, w = 0)$ 距离计算的，stride 是半个 Warp 内连续两个线程执行相同的操作的地址间隔。发现存在存储体冲突的执行时间普遍比没有存储体冲突执行时间长，而且在 $stride = 8, 14, 16$ 处尤为明显。存在存储体冲突的距离计算中：当 $w = 2$ 时，两个线程之间的 stride 为 $2*(w+2) = 8$，整个“8-way Bank Conflict”即每 8 线程访问同一个 Bank 的不同地址；而 $w = 5, stride = 14$ 处出

现的时“2-way Bank Conflict”，但是由于 $w = 5$ 每个线程访问 Bank 次数比较多所以导致使时间差异比较大；当 $w = 6$ 时，连续两个线程间隔 $2 * (w + 2) = 16$，半个 Warp 中的所有线程访问相同的 Block 中不同的地址出现“16-way Bank-Conflict”，这个在存储体冲突访问中是最严重的，所以时间差异最大。图 5.4是两者执行时间随 w 的变化，最初 $w > 0$ 距离计算模块存在存储体冲突的问题也是通过此图 $w = 6$ 的尖峰异常发现的。

表 5.10　否存在存储体冲突对于距离计算阶段执行时间的影响

w	stride	Bank-Conflict(s)	No-Bank-Conflict(s)	差 (s)	访问 Bank 次数
$w = 0$	4	43.4377	43.5549	0.1172	6.5e5
$w = 1$	6	110.860	122.403	11.543	9.7e5
$w = 2$	8	186.967	205.580	18.613	1.2e6
$w = 3$	10	267.948	274.802	6.8540	1.6e6
$w = 4$	12	346.335	347.883	1.5480	1.9e6
$w = 5$	14	422.306	445.596	23.290	2.3e6
$w = 6$	16	496.314	629.338	133.02	2.6e6
$w = 7$	18	571.834	595.752	23.918	2.9e6

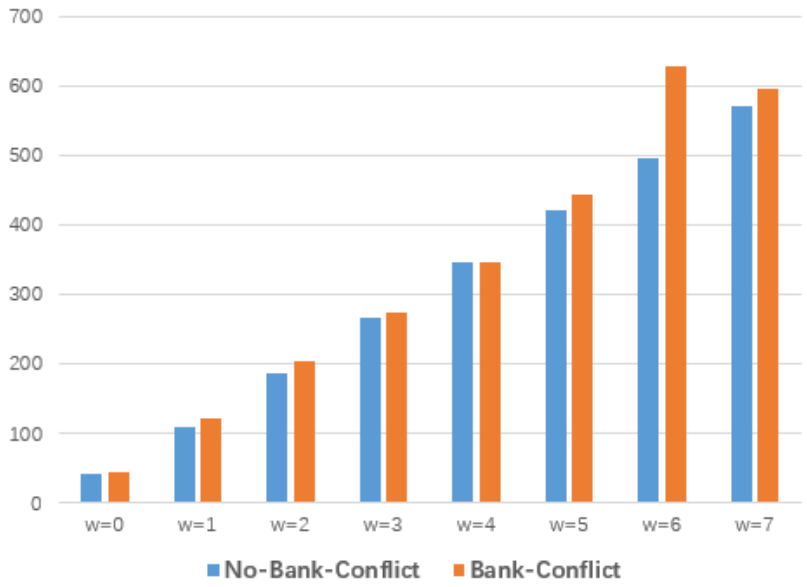

图 5.4　否存在存储体冲突对于距离计算阶段执行时间的影响

5.2.3　w=0 距离计算模块时间分析

在距离计算和最佳分割点计算之间有增加了一个转置过程，因此使用总的执行时间来进行比较。

表 5.11　转置对于 $w = 0$ 时并行总体计算总体执行时间的影响

数据集	实现转置 (ms)	直接存储 (ms)
GUN_POINT	172.816	205.170
ECG200	155.726	172.474
ECGFiveDays	17.8826	20.9651
Coffee	127.688	152.241
Strawberry	11559.5	12946.3
wafer	25716.8	39719.9

从表 5.11可以看出，使用矩阵转置的方法比直接存储执行时间更短，转置过程使距离计算和最佳分割点计算访问内存都可以通过合并内存访问进行，使得延时等待的时间大大减少，而且转置过程本身也是全局内存访问的经典应用，访存等待时间较短。总之，转置过程使 w=0 时 Shapelet 并行计算效率提升很多。

5.2.4　最佳分割点计算模块执行时间分析

表 5.12　最佳分割点计算的启发式算法和传统算法的比较

数据集	N	L	原始最佳分割点算法		启发式	
			时间 (ms)	准确率	时间 (ms)	准确率
GUN_POINT	50	150	275.370	0.849	99.2050	0.845
ECG200	100	96	363.013	0.800	135.543	0.800
ECGFiveDays	23	136	124.928	0.962	5.94600	0.963
Coffee	28	286	525.071	0.806	77.6015	0.811
Strawberry	370	235	8284.36	0.801	3663.46	0.794
Yoga	300	426	24883.4	0.648	8406.19	0.644
Ham	109	431	4668.29	0.614	1744.19	0.629
wafer	1000	152	NA*	0.953	10334.5	0.953

*：需要资源太多，图形处理器不能执行

本章节针对章节 4.3提出的启发式最佳分割点计算以及随机排列 Shuffle 进行

验证。为了获取启发式算法和原始算法的评价指标关系以及使用 Shuffle 对于指标的影响，需要对于原始最佳分割点算法、启发式最佳分割点计算进行多个数据集下准确率和执行时间指标的评价。表 5.12是对于两种算法在不同数据集下的时间/准确率表现，其中时间只是最佳分割点计算阶段的执行时间。

从表 5.12可以看出启发式算法可以在准确率不降或者微降的前提下大幅度提高执行效率，平均提速 2 倍左右 (仅对于最佳分割点计算阶段执行时间)。

5.3　本章小结

本章对于基于 DTW 距离度量的 Shapelet 并行算法在多个时间序列集上进行了验证，首先进行了实验环境、实验环境、实验数据、评价指标的介绍，在执行时间方面和已有的加速算法进行比较；最后对于文中使用的优化技术分别进行了验证，肯定了优化技术对于性能的提高，这有优化技术包括 $w > 0$ 距离计算模块的合并内存访问和存储体冲突，$w = 0$ 距离计算模块的矩阵转置，GPU 最佳分割点计算模块的启发式算法。

第 6 章　总结与展望

6.1　工作总结

时间序列数据遍布金融、互联网、制造等各个行业，越来越多的机器学习算法应用于时间序列数据帮助进行分析和决策，使人们意识到了时间序列的价值，开始投入精力研究时间序列数据挖掘。Shapelet 是具有一种具有很好分类能力的子序列，能够对于没有类标的数据进行分类。但是 Shapelet 的发现过程是很消耗时间和资源的过程。为了提高获得 Shapelet 的速度，本文提出了一种基于 DTW 距离度量的 Shapelet 并行方法，主要工作分为以下几点：

1. 提出了一种计算 Shapelet 的计算框架，根据数据集的不同（实际根据是不同都会根据内存的消耗情况以及 N, L 的大小）来确定合适的方法，能够对于较大的数据集进行处理。并且可以根据 w 参数来选择子序列间相似度计算的弯曲匹配程度，在 $w = 0 \rightarrow |S|$ 区间内选择使准确最高的候选序列。

2. 在距离计算阶段，根据 w 的不同，分别设计了 w>0 距离计算模块和 w=0 距离计算模块，两者都利用了不同候选序列和时间序列的距离之间的关系，来避免重复性计算降低时间复杂度。在设计并行算法时，将并行算法和 CUDA 优化技术结合使在有限的图形处理器资源上进行更多的计算，这些优化技术包括线程通信、合并内存访问、Warp 分歧、存储体冲突、矩阵转置等。

3. 在最佳分割点计算阶段，使用一种启发式最佳分割点计算 + 随机排列的方法在保证准确率不降的基础上来减少最佳分割点计算需要的时间，其中，启发式算法用来降低时间复杂度，随机排列使启发式获得分割点趋近最优值。

4. 在 w>0 距离计算模块、w=0 距离计算模块、最佳分割点计算模块，会估计不同数据集即不同 w, N, L 参数下每个 Block 中共享内存和寄存器的使用情况，然后会根据不同资源使用情况调节 Block 和 Grid 参数，避免单个 Warp 占用资源太多，而影响 SM 上同时执行的 Warp 个数，这样可以使有限的资源上进行更多的计算。

5. 最后对于算法在多个数据集上进行了验证，和深度学习进行了准确率方面的比较，和已有优化算法进行比较效率方面的比较。对于距离计算阶段各个 CUDA 优化技术和算法结合对于优化加速的影响进行了验证。最佳分割点计算时分别使用启发式算法和传统算法在准确率和性能方面进行了比较。

6. 本文主要应用于时间序列分类的二分类问题，适用于其中一类存在某种现象和症状的问题，能够快速获得这类问题的分类器。

6.2　对未来工作的展望

本文中我们提出的基于 DTW 距离度量的 Shapelet 并行方法虽然在性能优化上解决了一定的问题，但是整个算法还是存在很多优化的空间，具有如下面几个方面：

1. 本文中对于距离的计算 $Dist(A, B)$ 是按照时间序列内部的实际值计算的，还可以对 A, B 进行标准化之后再进行计算距离 $Dist(zscore(A), zscore(B))$，来进一步提高准确率指标。但是因为本算法内部使用了重用策略，如果加入标准化之后仍然想使用重用策略，这里就重新考虑标准化之后距离之间的转化。对于 $w = 0$ 时，需要计算 $Dist(zscore(T^{len}_{i,s+1}), zscore(T^{len}_{j,p+1}))$ 和 $Dist(zscore(T^{len}_{i,s}), zscore(T^{len}_{j,p}))$ 两者之间关系；而对于 $w > 0$ 情况来说, 就需要计算 $DTW(zscore(A_{1:m}), zscore(B_{1:m}), w)$ 和 $DTW(zscore(A_{1:m+1}), zscore(B_{1:m+1}), w)$ 两者之间的关系。

2. 对于定制化需求的矩阵转置部分，还存在可以优化的空间，目前的定制化矩阵转置算法每次从 Block 产生的矩阵读取一行数据，当数据集中时间序列长度 L 较小时没有问题，但当 L 较大时，需要较多的共享内存空间，使得每个 SM 中同时执行的 Warp 个数减少从而影响执行效率。对于 L 较大占用较大资源的问题，可以通过将每次读取的一行进行分段读取，前提是满足合并内存访问。

3. 文中提到时间序列数据必须考虑其长尾分布，本文使用数据集的最大的正负样本比例为 1 : 3。应该在正负样本比例更大的数据集中进行试验，以保证 Shapelet 方法对于正负样本不敏感并且能够对长尾时间序列进行正确分类。

4. 对于延时隐藏还可以继续改进，目前显示 Block 和 Grid 的参数都是经过简单比较获得的，不能确定目前的参数是否最优。可以加入占有率的计算和资源分配计算确保目前是能够最大限度利用 GPU 资源。

参考文献

[1] Anandarajan M, Anandarajan A, Srinivasan C A. Business intelligence techniques: a perspective from accounting and finance[M]. [S.l.]: Springer Science & Business Media, 2012

[2] Zhang Q, Wu J, Yang H, et al. Unsupervised feature learning from time series.[C]//IJCAI. [S.l.: s.n.], 2016: 2322–2328.

[3] Sprint G, Cook D, Weeks D, et al. Analyzing sensor-based time series data to track changes in physical activity during inpatient rehabilitation[J]. Sensors, 2017, 17(10): 2219.

[4] Chakraborti A, Patriarca M, Santhanam M. Financial time-series analysis: A brief overview[M]// Econophysics of Markets and Business Networks. [S.l.]: Springer, 2007: 51–67

[5] Cammarota C. Time series analysis of data from stress ecg[M]//Applied And Industrial Mathematics In Italy III. [S.l.]: World Scientific, 2010: 156–164

[6] Rangaswamy S. Time series data mining tool[J]. IJRCCT, 2013, 2(10): 1037–1040.

[7] Esling P, Agon C. Time-series data mining[J]. ACM Computing Surveys (CSUR), 2012, 45(1): 12.

[8] Wei L, Keogh E. Semi-supervised time series classification[C]//Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. [S.l.]: ACM, 2006: 748–753.

[9] Xing Z, Pei J, Keogh E. A brief survey on sequence classification[J]. ACM Sigkdd Explorations Newsletter, 2010, 12(1): 40–48.

[10] Fakhrazari A, Vakilzadian H. A survey on time series data mining[M]. [S.l.: s.n.].

[11] Hills J, Lines J, Baranauskas E, et al. Classification of time series by shapelet transformation[J]. Data Mining and Knowledge Discovery, 2014, 28(4): 851–881.

[12] Mueen A, Keogh E, Young N. Logical-shapelets: an expressive primitive for time series classification[C]//Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining. [S.l.]: ACM, 2011: 1154–1162.

[13] Saboia J L M. Autoregressive integrated moving average (arima) models for birth forecasting[J]. Journal of the American Statistical Association, 1977, 72(358): 264–270.

[14] Wang Z, Yan W, Oates T. Time series classification from scratch with deep neural networks: A strong baseline[C]//Neural Networks (IJCNN), 2017 International Joint Conference on. [S.l.]: IEEE, 2017: 1578–1585.

[15] Earnest A, Chen M I, Ng D, et al. Using autoregressive integrated moving average (arima) models to predict and monitor the number of beds occupied during a sars outbreak in a tertiary hospital in singapore[J]. BMC Health Services Research, 2005, 5(1): 36.

[16] Zheng Y, Liu Q, Chen E, et al. Time series classification using multi-channels deep convolutional neural networks[C]//International Conference on Web-Age Information Management. [S.l.]: Springer, 2014: 298–310.

[17] Hou L, Kwok J T, Zurada J M. Efficient learning of timeseries shapelets[C]//Thirtieth AAAI Conference on Artificial Intelligence. [S.l.: s.n.], 2016.

[18] Renard X, Rifqi M, Erray W, et al. Random-shapelet: an algorithm for fast shapelet discovery [C]//Data Science and Advanced Analytics (DSAA), 2015. 36678 2015. IEEE International Conference on. [S.l.]: IEEE, 2015: 1–10.

[19] Yang Y, Deng Q, Shen F, et al. A shapelet learning method for time series classification[C]// Tools with Artificial Intelligence (ICTAI), 2016 IEEE 28th International Conference on. [S.l.]: IEEE, 2016: 423–430.

[20] Furao S, Ogura T, Hasegawa O. An enhanced self-organizing incremental neural network for online unsupervised learning[J]. Neural Networks, 2007, 20(8): 893–903.

[21] Grabocka J, Schilling N, Wistuba M, et al. Learning time-series shapelets[C]//Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. [S.l.]: ACM, 2014: 392–401.

[22] Rakthanmanon T, Keogh E. Fast shapelets: A scalable algorithm for discovering time series shapelets[C]//Proceedings of the 2013 SIAM International Conference on Data Mining. [S.l.]: SIAM, 2013: 668–676.

[23] Lin J, Keogh E, Wei L, et al. Experiencing sax: a novel symbolic representation of time series [J]. Data Mining and knowledge discovery, 2007, 15(2): 107–144.

[24] Chang K W, Deka B, Hwu W W, et al. Efficient pattern-based time series classification on gpu [C]//Data Mining (ICDM), 2012 IEEE 12th International Conference on. [S.l.]: IEEE, 2012: 131–140.

[25] Ye L, Keogh E. Time series shapelets: a new primitive for data mining[C]//Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining. [S.l.]: ACM, 2009: 947–956.

[26] Patidar A K, Agrawal J, Mishra N. Analysis of different similarity measure functions and their impacts on shared nearest neighbor clustering approach[J]. International Journal of Computer Applications, 2012, 40(16).

[27] Ding H, Trajcevski G, Scheuermann P, et al. Querying and mining of time series data: experimental comparison of representations and distance measures[J]. Proceedings of the VLDB Endowment, 2008, 1(2): 1542–1552.

[28] Serra J, Arcos J L. An empirical evaluation of similarity measures for time series classification [J]. Knowledg e-Based Systems, 2014, 67: 305–314.

[29] Müller M. Dynamic time warping[J]. Information retrieval for music and motion, 2007: 69–84.

[30] Sart D, Mueen A, Najjar W, et al. Accelerating dynamic time warping subsequence search with gpus and fpgas[C]//Data Mining (ICDM), 2010 IEEE 10th International Conference on. [S.l.]: IEEE, 2010: 1001–1006.

[31] Salvador S, Chan P. Toward accurate dynamic time warping in linear time and space[J]. Intelligent Data Analysis, 2007, 11(5): 561–580.

[32] Chen L, Özsu M T, Oria V. Robust and fast similarity search for moving object trajectories[C]// Proceedings of the 2005 ACM SIGMOD international conference on Management of data. [S.l.]: ACM, 2005: 491–502.

[33] Harris M. Optimizing cuda[J]. SC07: High Performance Computing With CUDA, 2007.

[34] Nickolls J, Kirk D. Graphics and computing gpus[J]. Computer Organization and Design: The Hardware/Software Interface, DA Patterson and JL Hennessy, 4th ed., Morgan Kaufmann, 2009: A2–A77.

[35] Lippert A. Nvidia gpu architecture for general purpose computing[M]. [S.l.]: April, 2009.

[36] NVidia C. C best practices guide[J]. NVIDIA, Santa Clara, CA, 2012.

[37] Woolley C. Gpu optimization fundamentals[J]. Technical report, Tech. Rep, 2013.

[38] Li J, Chen W, Tian J, et al. Data partitioning strategy of gpu heterogeneous clusters based on learning[J]. International Journal of Grid and Distributed Computing, 2016, 9(9): 403–418.

[39] Chrzeszczyk A, Chrzeszczyk J. Matrix computations on the gpu[J]. CUBLAS and MAGMA by examples. Source:< https://developer. nvidia. com/sites/default/files/akamai/cuda/files/Misc/mygpu. pdf, 2013.

[40] Ruetsch G, Micikevicius P. Optimizing matrix transpose in cuda[J]. Nvidia CUDA SDK Application Note, 2009, 18.

[41] Nvidia C. C programming guide, version 7.5[J]. NVIDIA Corp, 2015.

[42] Chen Y, Keogh E, Hu B, et al. The ucr time series classification archive[M]. [S.l.: s.n.], 2015.

致 谢

雕塑园边丁香初开，学堂路上车流涌动，清芬园里美食香醇，西操跑道汗水依旧，我却要离开学习生活了三年的清华，结束研究生生涯。挥不去的是园子里的点点滴滴，带不走的是留在这里的美好时光。在研究生三年里，有太多的人需要感谢。

首先，需要感谢我的导师邓仰东老师。邓老师知识渊博、涉猎广泛，有时候邓老师一句“你可以看看 XX，可以试试 XX，Somehow Work”就可以帮助解决困扰很久的问题，在 Research 的道路上帮我们指明方向。首先站在学生的角度考虑问题，平时我们需要找邓老师讨论问题时，都是进办公室直接找邓老师讨论，没有提前预约过，邓老师都会放下手头的工作，进行细致的讨论。邓老师经常给我们分享人生经验，言传身教，在生活上也是我们学习的榜样。

其次，感谢实验室的同学们以及已经毕业的学长们，相互提携，共同度过这研究生三年，时光因为有了你们才会更精彩。

最后，感谢家乡父母对我求学生涯的默默支持，伴随着一次次寒迎暑送、车载离愁中完成了研究生生涯。感谢女友郑艳女士的不离不弃，在假期里帮我调整格式，在午夜里陪我打吊瓶，在我心情苦闷的时候陪我去荷塘散心排解，人生难觅一知己，恰为红颜。

同样感谢 GitHub 开源项目 THU-Thesis，为本文的 latex 排版提供便利。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

个人简历、在学期间发表的学术论文与研究成果

个人简历

1989 年 03 月 26 日出生于陕西省临潼区。

2007 年 09 月考入西安交通大学电子科学与技术专业，2011 年 07 月本科毕业并获得工学学士学位。

2011 年 07 月进入艾默生网络能源有限公司工作，并于 2014 年 09 月离职。

2015 年 09 月进入清华大学软件学院攻读硕士学位至今。