

分类号 TP391.4

学号 13069011

U D C 004.93

密级 公开

工学博士学位论文
高性能视觉跟踪关键技术研究

博士生姓名 黄达飞

学科专业 计算机科学与技术

研究方向 计算机系统结构

指导教师 张春元 教授

国防科学技术大学研究生院

二〇一七年四月

Research on Key Techniques for High-Performance Visual Tracking

Candidate: HUANG Dafei

Supervisor: Prof. ZHANG Chunyuan

A dissertation

**Submitted in partial fulfillment of the requirements
for the degree of Doctor of Engineering
in Computer Science and Technology**

Graduate School of National University of Defense Technology

Changsha, Hunan, P. R. China

April 18, 2017

独 创 性 声 明

本人声明所呈交的学位论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表和撰写过的研究成果，也不包含为获得国防科学技术大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文题目：_____ 高性能视觉跟踪关键技术研究 _____

学位论文作者签名：_____ 日期： 年 月 日

学 位 论 文 版 权 使 用 授 权 书

本人完全了解国防科学技术大学有关保留、使用学位论文的规定。本人授权国防科学技术大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档，允许论文被查阅和借阅；可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

(保密学位论文在解密后适用本授权书。)

学位论文题目：_____ 高性能视觉跟踪关键技术研究 _____

学位论文作者签名：_____ 日期： 年 月 日

作者指导教师签名：_____ 日期： 年 月 日

目 录

第一章 绪论	1
1.1 研究背景和意义	1
1.1.1 视觉跟踪	1
1.1.2 高性能视觉跟踪	1
1.2 研究现状	1
1.2.1 经典视觉跟踪算法的相关研究	1
1.2.2 新兴视觉跟踪算法的相关研究	1
1.2.3 高性能跟踪算法实现的相关研究	1
1.3 主要研究内容和创新点	2
1.4 论文结构	2
第二章 使用“目标候选”提高跟踪器的尺度和宽高比适应力	3
2.1 引言	3
2.2 相关研究	5
2.2.1 跟踪器的尺度和宽高比适应力	5
2.2.2 基于相关滤波的跟踪器	6
2.3 使用核化相关滤波器进行视觉跟踪	7
2.4 图像特征整合和鲁棒更新	10
2.5 将“目标候选”嵌入跟踪器中	11
2.6 参数设置	14
2.7 实验结果与分析	17
2.7.1 实验设置	17
2.7.2 尺度和宽高比适应力评测	21
2.7.3 整体性能评测	22
2.8 小结	24
第三章 跟踪器中“目标候选”的作用分析和优化	25
3.1 引言	25
3.2 相关研究	25
3.2.1 目标候选生成器及其在跟踪中的应用	25
3.2.2 图像分割在跟踪中的应用	25
3.3 跟踪器中目标候选生成器的适配	25
3.4 跟踪器中目标候选生成器的优化	25
3.4.1 目标候选生成器 EdgeBoxes	25

3.4.2 使用背景抑制优化 EdgeBoxes	25
3.5 实验评测与分析	25
3.5.1 实验设置	25
3.5.2 跟踪器中目标候选的作用分析	25
3.5.3 跟踪器中目标候选的优化效果评测	26
3.6 小结	26
第四章 基于 OpenCL 的 TLD 算法高性能实现	27
4.1 引言	27
4.2 OpenCL 编程模型	28
4.3 TLD 跟踪算法	31
4.4 Fern 随机森林的高性能实现	34
4.4.1 特征提取的并行化	35
4.4.2 分类过程的并行化	36
4.4.3 与跟踪器重叠执行	37
4.5 最邻近分类器的高性能实现	38
4.5.1 算法流程	39
4.5.2 分类过程的并行化	39
4.6 学习过程的高性能实现	42
4.6.1 重叠率计算和负样本提取的并行化	42
4.6.2 正样本提取的并行化	43
4.7 实验评测与分析	45
4.7.1 Kernel 性能评测与分析	47
4.7.2 整体性能评测与分析	50
4.8 小结	51
第五章 GPU 特定 OpenCL Kernel 程序的性能移植性提升	53
5.1 引言	53
5.2 相关工作	55
5.3 数组访问的线性描述式	56
5.4 基于分析的工作项折叠	58
5.4.1 去除冗余的局部存储数组	59
5.4.2 依赖性分析和同步语句消除	62
5.5 适应体系结构的后继优化	65
5.5.1 向量化	66
5.5.2 局部性重开发	68

5.6 运行时调度	71
5.7 性能评测	73
5.8 小结	76
第六章 总结与展望	79
6.1 工作总结	79
6.2 未来研究方向	79
致谢	81
参考文献	83

表 目 录

表 4.1 特征提取的 Kernel 程序	36
表 4.2 分类过程的 Kernel 程序	37
表 4.3 LK 光流跟踪和 Fern 随机森林分类的重叠执行	38
表 4.4 NCC 计算的 Kernel 程序	41
表 4.5 重叠率计算和负样本提取的 Kernel 程序	43
表 4.6 正样本提取的 Kernel 程序	45
表 4.7 各 Kernel 的时间开销对比	47
表 4.8 各 Kernel 的加速比对比	48
表 5.1 原始的 GPU 特定矩阵乘法 Kernel 程序	57
表 5.2 去除冗余的局部存储数组后的矩阵乘法 Kernel 代码片段	62
表 5.3 工作项折叠后的矩阵乘法 Kernel 代码片段	65
表 5.4 向量化后的矩阵乘法 Kernel 代码片段	68
表 5.5 面向 Sandy Bridge 架构转换后的最终矩阵乘法 Kernel 代码片段	70
表 5.6 面向 Knights Corner 架构转换后的最终矩阵乘法 Kernel 代码片段	71
表 5.7 用于性能评测的 6 个 Kernel 程序	73
表 5.8 与 Intel OpenCL 运行时和 OpenMP 的性能对比	74

图 目 录

图 2.1 尺度和宽高比误差的可视化例子	3
图 2.2 生成“目标候选”的可视化例子	4
图 2.3 KCF 的运动模型和观察模型	8
图 2.4 使用高斯核后的 KCF 运动模型和观察模型	9
图 2.5 本章跟踪器的可视化跟踪过程	12
图 2.6 目标候选辨别和带阻尼更新的可视化例子 (图例见图 2.5)	14
图 2.7 使用不同标准方差的相关滤波器回归目标矩阵	16
图 2.8 本章所使用的 OTB ^[1] 测试集	18
图 2.9 判断视频序列包含“宽高比变化”的过程	19
图 2.10 针对尺度适应力的实验结果	20
图 2.11 针对宽高比适应力的实验结果	21
图 2.12 KCFDP 与 4 个跟踪器的可视化对比	22
图 2.13 完整的 50 个视频序列上的实验结果	23
图 4.1 OpenCL 的平台模型和存储模型 ^[2]	29
图 4.2 OpenCL 的执行模型	29
图 4.3 TLD 跟踪算法的三大部分	31
图 4.4 Fern 随机森林分类的流程	34
图 4.5 特征提取的流程	36
图 4.6 NCC 计算的 OpenCL 索引空间	40
图 4.7 正样本提取流程	44
图 4.8 TLD 算法高性能实现的部分运行结果	46
图 4.9 TLD 算法各部分的时间开销对比	49
图 4.10 整个应用的时间开销对比	50
图 5.1 矩阵乘法 Kernel 中读 \mathbf{A} 、写 \mathbf{AS} 和读 \mathbf{AS} 的数组访问线性描述式	58
图 5.2 通过构建线程循环来进行工作项折叠	59
图 5.3 依赖性分析示意图	64
图 5.4 矩阵乘法 $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ 中对于数组 \mathbf{A} 和 \mathbf{B} 的不同访问顺序	66
图 5.5 转换后的矩阵乘法 Kernel 对于数组 \mathbf{A} 和 \mathbf{B} 的访问顺序	70
图 5.6 GPU 特定 Kernel 的转换和调度执行	72
图 5.7 后继优化中各步骤带来的性能提升	76

第一章 绪论

1.1 研究背景和意义

1.1.1 视觉跟踪

什么是视觉跟踪，用途，价值，困难挑战

1.1.2 高性能视觉跟踪

两方面的高性能

1.1.2.1 高性能视觉跟踪算法

解释含义，及其价值、挑战

1.1.2.2 跟踪算法的高性能实现

解释含义，及其价值、挑战

1.1.2.3 异构计算平台及其编程模型

什么是异构平台（当前火热），介绍 GPU 和 CPU (MIC) 体系结构，介绍几个常用编程模型

1.2 研究现状

1.2.1 经典视觉跟踪算法的相关研究

给出经典跟踪器结构，介绍几大模块的发展

1.2.2 新兴视觉跟踪算法的相关研究

TLD，神经网络的应用，“目标候选”方法的应用

1.2.3 高性能跟踪算法实现的相关研究

1.2.3.1 异构平台下跟踪算法的高性能实现

介绍几个跟踪器的高性能实现（没有使用 OpenCL 实现的），指出缺少可移植性（不能在各种平台下运行）

1.2.3.2 异构平台下的高性能编程模型

介绍几个高性能跟踪器可用的新兴编程模型，指出移植性问题。说明 OpenCL 好处，指出 OpenCL 的性能移植性问题，列出几个解决性能移植性的相关工作

1.3 主要研究内容和创新点

1. 一个将目标候选融入跟踪器的方法
2. 分析目标候选对跟踪精度的影响，从而对通用的目标候选生成方法进行改进
3. 基于 OpenCL 的高性能 TLD 跟踪器实现
4. 解决 OpenCL 程序从 GPU 到 CPU 的性能移植性问题

1.4 论文结构

第二章 使用“目标候选”提高跟踪器的尺度和宽高比适应力

2.1 引言

绪论已经介绍过，视觉跟踪器是视频监控、人机交互、智能导航等应用领域至关重要的基础。从上世纪 80 年代的 KLT 跟踪器^[3, 4]，到最新提出的深度学习跟踪器^[5, 6]，视觉跟踪已经被研究了几十年。但随着应用需求的不断提升，应用场景日趋复杂，很多问题仍然未得到有效解决。跟踪器对目标物体的尺度和宽高比的适应力就是这些问题之一。

跟踪算法的基础研究一般面向最为通用的跟踪器类别，即“短时间、单目标、无模型”跟踪器。“短时间”是一个相对概念，通常意指用于跟踪的是较短的视频序列（通常从几百帧到几千帧），而不是直接进行长时间的在线跟踪。“单目标”即被跟踪的目标物体只有一个，跟踪过程中不考虑其它物体的位置和状态。“无模型”指的是不依赖跟踪目标的任何先验信息，不在跟踪前对目标物体进行任何建模。跟踪算法中最为通用的结果表示方式是“边界框”，即在图像帧中紧密包围目标物体的矩形框。跟踪过程中，目标物体可能发生复杂的运动、旋转、形变等。体现在结果上，就是边界框的位置、尺度和宽高比发生变化。如果跟踪器边界框的尺度和宽高比无法随物体发生变化，那么跟踪结果仅能体现出物体的位置信息，而无法描述其它物体状态。除此之外，不准确的边界框还将导致跟踪过程中不准确的目标 / 背景分割。过多的非目标物体信息可能包含在边界框中，而部分目标物体信息却可能在边界框外。这些误差将在物体描述中积累，积累到一定程度，跟踪器将出现偏移（Drifting），整个跟踪过程很可能就此失败。图2.1就展示了一个尺度和宽高比误差的可视化例子。因此，跟踪器对目标尺度和宽高比的适应力对跟踪精度有着决定性的作用。

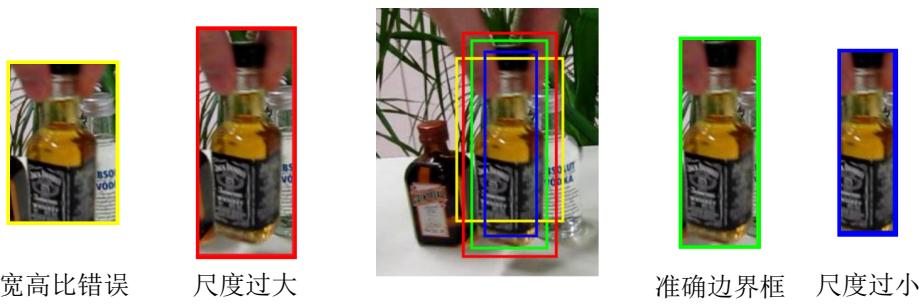


图 2.1 尺度和宽高比误差的可视化例子

为了应对视觉跟踪应用中的诸多挑战，跟踪算法正变得越来越复杂。然而近年提出的基于相关滤波的跟踪器^[7-13]在取得上佳性能的同时，却十分简单和快速。

这些跟踪器的核心部分都是一个具有分辨能力的滤波器，其输出的卷积结果能够显示出输入图像和跟踪目标的相似度。由于频率域的点对点操作等价于时间域（图像处理中为空间域）的卷积操作，上述滤波器能够极其高效地辨识所有循环位移后的输入图像。但是，滤波器的输入必须是固定大小的图像块，因此基于相关滤波的跟踪器天生缺乏对于目标尺度和宽高比变化的适应力。尽管一些能够适应尺度变化的变型^[10, 12, 13]已经出现，但是它们仍局限于预定的尺度采样方式，不够灵活。此外，在本章已知的范围内，除了^[13]以外还没有相关滤波跟踪器能够解决对于目标宽高比的适应性问题。

在物体检测领域，近来具有顶级性能的物体检测系统^[14, 15]均采用了“目标候选”方法来提取可能包含目标物体的候选区域。该类方法可以在没有任何先验知识的情况下，在输入图像中提取任意尺度、宽高比的候选边界框，如图2.2所示。目标候选方法不仅能够避免对大量的边界框进行分类，还能预先滤除大部分错误的边界框，大幅提高检测精度^[16, 17]。在本章中，目标候选生成器 EdgeBoxes^[18]将被融入跟踪器中，以提升跟踪器对尺度和宽高比的适应力。

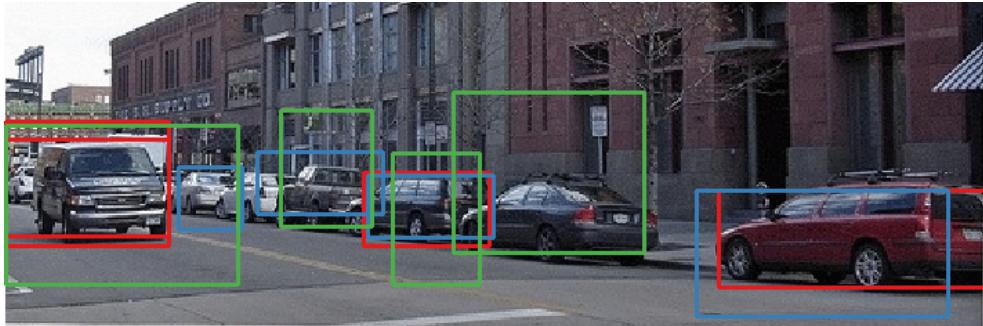


图 2.2 生成“目标候选”的可视化例子

本章选用 EdgeBoxes 的原因在于，在众多目标候选生成器中，它不仅具有顶级的生成精度，而且还具有足以应用于跟踪任务的生成速度（考虑到跟踪任务中的待检测范围远小于物体检测任务）。此外，由于 EdgeBoxes 将物体边界作为生成目标候选的唯一线索，它在跟踪任务中还具有以下优势。首先，它不需要任何预先学习过程，因而适用于通用的视觉跟踪器。其次，作为物体线索的边界，同时也是推测被跟踪目标物体的位置和大小的线索。最后，它包含了大量可调节的参数，使得可以针对跟踪任务进行调优。

本章中，KCF^[9]这一典型的基于相关滤波的跟踪器将作为跟踪器的主体框架。通过对输入滤波器的图像特征进行增强，以及采用更为鲁棒的跟踪器模型更新方法，KCF 的辨别力将变得更加鲁棒和准确，足以对灵活的目标候选进行分类。初步确定目标所在位置后，EdgeBoxes 将被用于在其附近生成目标候选。这些候选边界框经过一个过滤步骤后，将被输入 KCF 进行再次辨别，以提取出最优的目标

候选。最后，通过一个阻尼更新过程，最优候选边界框将被用于确定最终的目标位置、尺度和宽高比。本章最终将得到一个集成了目标候选生成器的全新跟踪器，它在公开的大规模测试集中取得了顶级的精度，并且达到了 20.8 FPS（每秒处理的帧数）的跟踪速度。

本章的内容安排如下：第 2 节介绍与本章紧密相关的现有研究工作；第 3 节介绍本章跟踪器框架的基础——核化相关滤波器 KCF；第 4 节对 KCF 进行优化，以适应对目标候选的分类任务；第 5 节介绍如何将目标候选生成器 EdgeBoxes 高效地融入跟踪框架中；第 6 节将对本章方法进行实验评测和分析；最后在第 7 节进行本章的总结。

2.2 相关研究

2.2.1 跟踪器的尺度和宽高比适应力

根据绪论中对跟踪器各模块的功能分析可以看出，运动模型是决定一个跟踪器的尺度和宽高比适应力的关键。^[19] 基于光流跟踪算法，已被广泛应用于图像对准 (Image Registration)。它通过增量式对齐 (Incremental Alignment) 来计算两帧中目标物体图像块的仿射变换 (Affine Transformation)。由于仿射变换的参数包含 6 个自由度，因此该跟踪器可以感知目标物体的位移、尺度变化和旋转。LSK 跟踪器^[20] 在每一帧中都会预先“猜测”一个目标中心位置并采样候选图像块，然后使用 Mean-Shift 聚类方法优化该猜测，以最大化候选图像块和匹配模板的相似度。由于 Mean-Shift 优化过程会根据多种尺度反复进行，因此 LSK 跟踪器能够成功判断当前目标的尺度大小。上述两个跟踪器中，物体的运动都是通过计算或者优化过程直接得到的，因此其运动模型属于隐式模型。

ASLA^[21] 和 SCM^[22] 均采用仿射变换来描述两帧间的目标运动。与 ^[19] 根据光流跟踪结果直接计算仿射变换不同，这两个跟踪器将仿射变换的 6 个参数用 6 个独立的高斯分布进行建模。然后，跟踪器将通过不断采样仿射变换参数并进行验证的方式，寻找最为合适的目标位置和尺度。这种按照概率分布来采样目标状态参数，然后以目标当前状态更新概率分布的方式，属于典型的点滤波 (Particle Filtering) 运动模型。VTD^[23] 跟踪器会同时考虑目标的位置和尺度变化，并用两个高斯分布来分别建模平滑和突发的目标运动。这两个点滤波运动模型将配合多个不同的观察模型，得出多个跟踪结果，并通过马尔科夫链蒙特卡洛 (Markov Chain Monte Carlo) 方法整合出一个最终结果。

此外，还有部分跟踪器使用了特殊方法来获得尺度和宽高比适应力。为了应对物体非刚性形变带来的大幅度尺度和宽高比变化，HBT^[24] 将图像分割方法加入了跟踪过程。根据霍夫森林 (Hough Forest) 得出的反向映射向量将投票决定目标

物体的中心位置，而中心位置将作为前景点输入图割算法 **Grabcut**，以对目标物体和背景进行分割。由于图像分割得到的是目标物体的细致轮廓，因此 **HBT** 可以准确适应目标的尺度和宽高比变化。但是图像分割计算开销巨大，使得 **HBT** 难以用于实时跟踪场景。**TLD**^[25, 26] 跟踪算法将 **Fern** 随机森林检测器和光流跟踪器进行了整合。**Fern** 随机森林检测器通过滑动窗口的方式搜索整幅帧图像，找出可能包含目标物体的边界框；光流跟踪器则根据多个目标特征点的运动，估计目标的整体运动。这两个模块均能适应目标的尺度和宽高比变化。

2.2.2 基于相关滤波的跟踪器

基于相关滤波的跟踪器通常选择密集采样作为运动模型，即是说，它们通过在图像中密集地采样候选区域并进行辨别，来检测目标物体的当前位置和状态。因此，密集采样的模式（例如采样时边界框的大小、形状是否可变）决定了跟踪器对尺度和宽高比的适应力。**MOSSE** 跟踪器^[7] 将经过随机仿射变换的目标物体图像作为训练集，用于初始化它的相关滤波器。但是在跟踪过程中，采样边界框的尺度和宽高比不再变化，滤波器仅用于检测目标的当前位置。**KCF**^[9] 跟踪器是 **CSK**^[8] 的一个扩展版本，它通过利用图像块中的循环模式进行卷积操作，取得了极高的跟踪效率。**KCF** 还利用“核技巧（Kernel Trick）”来增强了传统的相关滤波器，同时使得滤波器支持多通道的特征。但是，它仍然没有解决尺度的适应力问题。作为 **CSK** 的另一个固定尺度扩展版本，**ACT**^[11] 采用颜色名（Color Naming）作为特征，并使用了特征压缩方法提高跟踪效率。此外，**ACT** 还提出了一个更加鲁棒的模型更新策略以提高精度。

SAMF^[12] 在 **KCF** 的基础上，通过在采样时加入几种预定义的尺度变化来解决 **KCF** 的尺度适应力问题。相关滤波器将对这些不同尺度的采样样本进行辨别，以找出最佳的目标位置和尺度大小。不同于 **SAMF**，**DSST**^[10] 将两个独立的相关滤波器结合在了一起：一个基于 **MOSSE**，仅用于估计目标位置；另一个为一维相关滤波器，仅用于确定目标物体尺度。在每一帧中，**DSST** 首先利用第一个滤波器确定目标位置。然后在该位置处采样不同尺度的图像块组成“尺度金字塔”，并将其输入第二个滤波器中以估计尺度。对比 **SAMF** 和 **DSST**，可以看出 **DSST** 的尺度检测会更加地准确和快速。因为相关滤波器具有辨别力，可以显式地区分出各个尺度的置信度，并且可以在频率域更快地进行计算。但是，这两个基于相关滤波的跟踪器都没有针对目标的宽高比变化进行处理。^[13] 中提出的跟踪器通过使用多个独立的相关滤波器来同时跟踪目标物体的多个部分，并将各个滤波器的输出整合为一个响应图。随后，跟踪器在响应图中按照高斯分布采样不同的位置、尺度和宽高比，并使用一个贝叶斯框架对采样样本进行评价，以确定目标物体的

当前状态。显然，上述跟踪器均依赖于预先定义好的采样模式，因而灵活性严重受限，无法处理突发、快速的尺度和宽高比变化。

STC^[27]是一个例外，它在估计目标尺度时使用的运动模型为隐式模型。严格来讲，STC 并不是基于相关滤波的，但是其推导出的核心算法与相关滤波器十分类似。它对目标和目标周围区域的上下文时空关系进行了重新建模，并将尺度的变化显式地体现在了模型中。因此尺度变化可以被直接计算出来而无需通过采样，但是宽高比的变化仍然无法被感知。

2.3 使用核化相关滤波器进行视觉跟踪

作为本章跟踪器的主体框架和基础，核化相关滤波器 KCF 的主要目标是训练一个线性模型：

$$f(\mathbf{z}) = \langle \mathbf{w}, \mathbf{z} \rangle, \quad (2.1)$$

其中函数值 $f(\mathbf{z})$ 显示了输入图像块 \mathbf{z} 与目标物体的相似程度，即线性模型的响应值。 $\langle \cdot, \cdot \rangle$ 代表内积， \mathbf{w} 为该线性模型的参数矩阵。公式2.1即是 KCF 的观察模型，即判断候选图像块是否包含目标物体的模块。KCF 将上一帧的目标物体边界框进行放大，得到一个上下文窗口，然后根据该窗口提取图像块。该图像块的每一次循环位移都将产生一个候选图像块 \mathbf{z} ，并将被输入观察模型，以计算其中包含目标的概率。显然，若当前帧中目标物体在上下文窗口内，那么目标物体一定位于某一个循环位移后的图像块的中心，该图像块的 $f(\mathbf{z})$ 也应当具有最大值。可视化的例子如图2.3所示。显然，循环位移就是 KCF 的运动模型。由于会对所有循环位移进行辨别，因此它还是一种密集采样运动模型。

求解参数矩阵 \mathbf{w} 的过程，就是求解“岭回归（Ridge Regression）”问题的过程。该求解或者说训练的目标函数为：

$$\min_{\mathbf{w}} \sum_{m,n} (f(\mathbf{x}_{m,n}) - y_{m,n})^2 + \lambda \|\mathbf{w}\|^2. \quad (2.2)$$

$\mathbf{x}_{m,n}$ 是用于训练的图像块， λ 被称作规则化参数 (Regularization Parameter)，用于防止过拟合。 $y_{m,n}$ 是回归目标，即对于输入的 $\mathbf{x}_{m,n}$ 所期望的 $f(\mathbf{x}_{m,n})$ 值。在 KCF 中， \mathbf{x} 代表目标物体图像块，即一个“真值”。 $\mathbf{x}_{m,n}$ 是由 \mathbf{x} 纵向循环位移 $m - 1$ 个像素，再横向循环位移 $n - 1$ 个像素所得到的图像块。显然 $\mathbf{x}_{1,1} = \mathbf{x}$ ，因此其对应的 $y_{1,1}$ 应当具有最大值，即 $y_{1,1} = 1$ 。若将所有的 $y_{m,n}$ 组成一个矩阵 \mathbf{y} ，那么理想情况下， \mathbf{y} 应当符合一个将峰值循环位移到左上角的二维高斯函数。

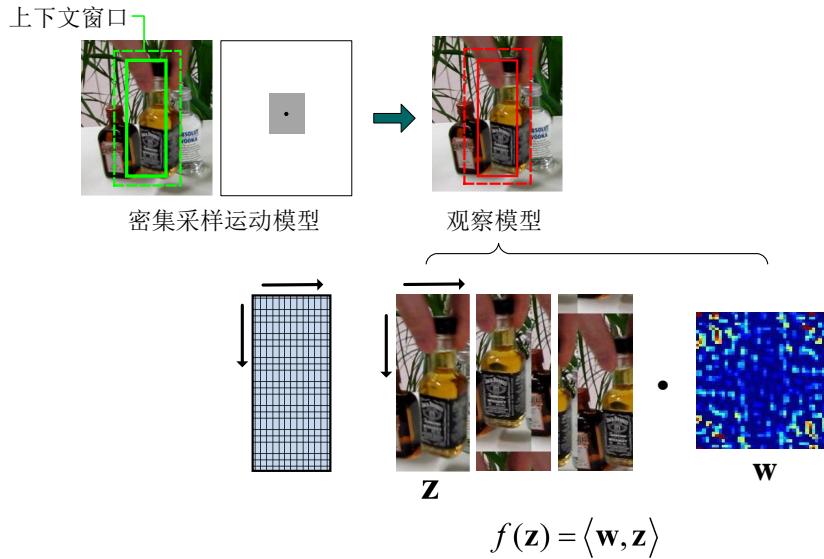


图 2.3 KCF 的运动模型和观察模型

KCF 通过使用高斯核函数，将公式2.1变换到了对偶空间（Dual Space）：

$$f(\mathbf{z}) = \langle \mathbf{w}, \mathbf{z} \rangle = \sum_{m,n} \alpha_{m,n} \langle \mathbf{z}, \mathbf{x}_{m,n} \rangle = \sum_{m,n} \alpha_{m,n} g(\mathbf{z}, \mathbf{x}_{m,n}), \quad (2.3)$$

其中 $g(\cdot, \cdot)$ 代表高斯核函数，它取代了原式中的内积计算。相对应的，KCF 的运动模型和观察模型也变为图2.4所示。计算 $f(\mathbf{z})$ 值的过程，实际上变为了将候选图像块 \mathbf{z} （图中标记为 \mathbf{C} ）和每一个训练图像块 $\mathbf{x}_{m,n}$ 进行对比的过程，参数矩阵 \mathbf{w} 也被新参数 $\alpha_{m,n}$ 所组成的矩阵 $\boldsymbol{\alpha}$ 所替代。而训练图像块集合中，循环位移较小的必然与原目标物体图像块相似，成为了正样本（图中标记为 \mathbf{P} ）；循环位移较大的则成为了负样本（图中标记为 \mathbf{N} ）。

通过利用各个 $\mathbf{x}_{m,n}$ 间的循环关系，以及卷积定理（Convolution Theorem），公式2.2的解为：

$$\hat{\boldsymbol{\alpha}} = \frac{\hat{\mathbf{y}}}{\hat{\mathbf{k}}^{\mathbf{x}_{1,1}\mathbf{x}_{1,1}} + \lambda}. \quad (2.4)$$

上式中的 $\hat{\cdot}$ 代表离散傅里叶变换（DFT）， \mathbf{k} 代表核化相关算子，其定义为：

$$\mathbf{k}^{\mathbf{x}'\mathbf{x}''} = \exp \left(-\frac{1}{\sigma^2} \left(\|\mathbf{x}'\|^2 + \|\mathbf{x}''\|^2 - 2\mathcal{F}^{-1} \left(\sum_c \hat{\mathbf{x}'}_c^* \cdot \hat{\mathbf{x}''}_c \right) \right) \right), \quad (2.5)$$

式中 σ 是高斯核函数的带宽， \mathcal{F}^{-1} 代表离散傅里叶逆变换， $*$ 代表复共轭算子，下标 c 代表图像特征向量的第 c 个通道， \exp 即指数函数。公式2.5中的所有运算操作，包括 \exp ，都是逐元素进行的。公式2.4将负责训练相关滤波器，即在第一帧中用用户标注出的或者检测到的目标物体图像块初始化相关滤波器。

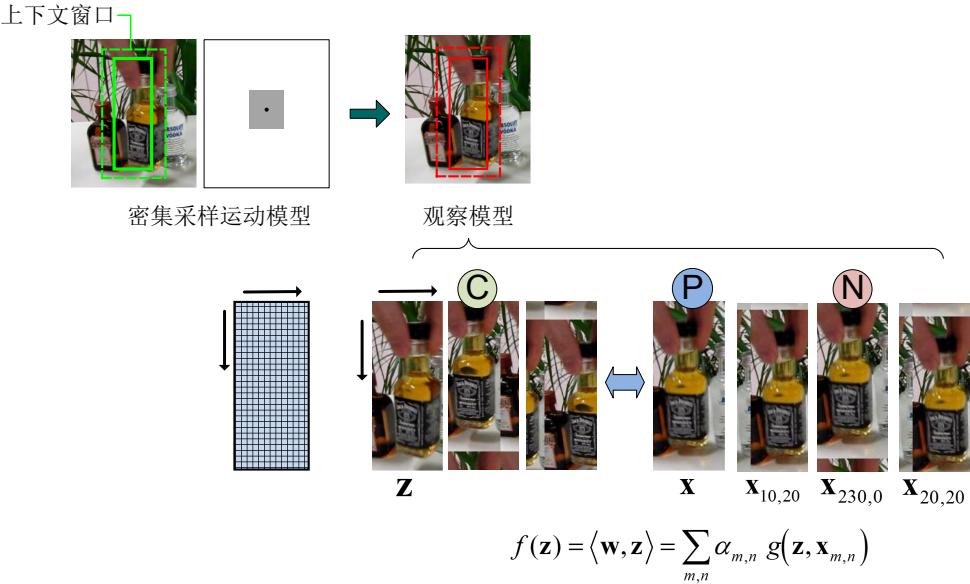


图 2.4 使用高斯核后的 KCF 运动模型和观察模型

同样利用循环关系和卷积定理，KCF 可以同时计算出输入图像块 \mathbf{z} 的所有循环位移所对应的响应值。也就是说，公式2.3可变换为：

$$\hat{\mathbf{f}}(\mathbf{z}) = \hat{\mathbf{k}}^{\bar{\mathbf{x}}\mathbf{z}} \cdot \hat{\boldsymbol{\alpha}}, \quad (2.6)$$

其中 $\bar{\mathbf{x}}$ 被称作当前目标外观。由于目标物体外观在跟踪中会不断变化，因此公式2.3中的 \mathbf{x} 也会不断更新，并记为 $\bar{\mathbf{x}}$ 。 \mathbf{z} 的所有循环位移的响应值 f 组成了矩阵 \mathbf{f} ，且各个响应值在 \mathbf{f} 中的位置，与循环位移的像素数目相对应，即 $\mathbf{f}[m, n] = f(\mathbf{z}_{m,n})$ 。因此，根据 \mathbf{f} 中最大元素所在的位置，可以找出最接近当前目标外观的 $\mathbf{z}_{m,n}$ ，从而直接推断出目标物体中心在上下文窗口中的位置。公式2.6是 KCF 的核心，将负责在每一帧中检测目标物体的中心位置。而它又是由参数矩阵 $\boldsymbol{\alpha}$ 和当前目标外观 $\bar{\mathbf{x}}$ 所共同决定的，因此这两个矩阵也被称作“KCF 的模型”。

每当在新的一帧中跟踪到目标物体后，KCF 都会更新自己的模型，以适应光照变化、目标旋转、形变等导致的目标外观变化。KCF 的模型更新策略非常简单直接，就是进行线性插值。记当前帧中目标物体图像块为 \mathbf{x}_i ，对于参数矩阵，首先重新训练一个参数矩阵 $\boldsymbol{\alpha}_i$ ，然后进行线性插值：

$$\hat{\boldsymbol{\alpha}}_i = \frac{\hat{\mathbf{y}}}{\hat{\mathbf{k}}^{\mathbf{x}_i \mathbf{x}_i} + \lambda}, \quad \boldsymbol{\alpha} = \eta \boldsymbol{\alpha}_i + (1 - \eta) \boldsymbol{\alpha}. \quad (2.7)$$

对于目标外观，则直接进行线性插值：

$$\bar{\mathbf{x}} = \eta \mathbf{x}_i + (1 - \eta) \bar{\mathbf{x}}. \quad (2.8)$$

2.4 图像特征整合和鲁棒更新

在跟踪过程中加入目标候选生成器后，相关滤波器需要辨别的不仅是循环位移后的图像块，还包括各种尺度大小和宽高比的目标候选。由于候选图像块的灵活性和数量增加，各种干扰因素也将增加，对相关滤波器的精度和鲁棒性也提出了更高的要求。

为了提升精度，本节将首先加强 KCF 的目标描述。KCF 所使用的“梯度直方图（HOG）”特征被扩展为 HOG、亮度和颜色名三者结合的混合特征。这种特征整合的方法在 SAMF^[12] 和 ACT^[11] 中均有使用，但是本节不对特征进行压缩。这三种特征的简要介绍如下：

- **亮度**：要计算亮度，需要先将彩色图像转换为灰度图像。假设图像块按 RGB 三个通道存储，则一个像素的灰度可计算为： $0.2989*R+0.5870*G+0.1140*B$ 。随后，将每个像素的灰度值减去整个图像块的平均灰度值，即得到了各个像素的亮度值。
- **梯度直方图^[28]**：在亮度图的基础上，首先对于每一个像素点，计算其横向和纵向的亮度梯度。然后根据横、纵亮度梯度，计算出每个像素的梯度方向。之后将图像分为 4×4 细胞单元（Cell）组成的网格，并在每一个细胞单元中对 9 个离散的梯度方向进行投票。每个像素点的投票均具有权值，且该权值就是梯度的大小。最后将多个细胞单元组合成较大的区块，并对这些区块中的梯度方向投票进行统计，得出梯度直方图特征。
- **颜色名^[29]**：之所以将这种特征命名为颜色名，是因为它将所有的颜色按照语言学分为了 11 类，即 11 类人类常用的颜色名字。这样的颜色分类更接近于人类的感知，且已经在物体检测^[30] 和动作识别^[31] 领域取得了很好的效果。这里直接利用 [29] 中的方法将 RGB 3 个通道的值向 11 个颜色名通道进行映射，得到颜色名特征。

将这三种特征同时用于目标描述的原因是它们之间有很好的互补性：亮度，顾名思义，描述的是各个像素的明亮程度，但是缺乏对不同颜色的区分度；梯度直方图更着重于描述图像中的轮廓和形状，但是难以区分亮度和颜色；颜色名是当前物体分类常用的颜色特征，有着明显优于传统 RGB 颜色表达的性能。

整合上述三种图像特征的方法比较直接，即对于一个图像块提取出三种特征后，将特征的各通道拼接存储。例如原彩色图像块的每一个像素有 R、G、B 三个通道，故图像块存储为高 \times 宽 \times 3 的三维数组。提取特征并且整合后，由于三种特征共计有 42 个通道，图像块变为高 \times 宽 \times 42 的三维数组。在训练参数矩阵（公式2.4）和评价候选图像块时（公式2.6），只需要按照公式2.5的最后一项，对特

征的各个通道进行分别处理即可。

如2.3节所述，KCF采用简单的线性插值作为模型更新策略。这种更新方法已经被证明是不够理想的^[11]，因为每次模型更新时（公式2.7），仅考虑了当前帧中的目标物体外表图像，而之前帧中的目标外表的贡献，将按照学习速率 η 的指数级函数不断减小。显然，这种更新策略是不够鲁棒的。当跟踪出现偏移，或者目标外表突发剧烈变化时，KCF原来积累的准确模型将被迅速污染，从而很可能导致更大的跟踪偏移。

为了提高KCF的相关滤波器的鲁棒性，本节将把线性插值方法替换为ACT中的鲁棒更新方法。在更新模型的参数矩阵 α 时，假设当前帧为第*i*帧，则从第1帧到当前帧的所有目标物体图像块可记为 $\{\mathbf{x}_j : j = 1, \dots, i\}$ 。更新参数矩阵的过程变为最小化以下目标函数的过程：

$$\begin{aligned} \min_{\alpha} \sum_{j=1}^i \phi_j (\|\mathbf{f}(\mathbf{x}_j) - \mathbf{y}\|^2 + \lambda \langle \alpha, \alpha \rangle) = \\ \min_{\alpha} \sum_{j=1}^i \phi_j (\|\mathcal{F}^{-1}(\hat{\mathbf{k}}^{\mathbf{x}_i \mathbf{x}_j} \cdot \hat{\alpha}) - \mathbf{y}\|^2 + \lambda \langle \alpha, \alpha \rangle). \end{aligned} \quad (2.9)$$

该目标函数同时考虑了当前帧和所有历史帧中的目标物体图像块，其实质为参数矩阵 α 对于所有帧中目标物体外表的平方误差加权和。 ϕ_j 即是权值，这里将其直接设置为学习速率 η 。

为了能够最小化上述目标函数，经过推导（详见[11]），新的参数矩阵更新公式为：

$$\begin{aligned} \hat{\alpha}_N &= \eta \hat{\mathbf{k}}^{\mathbf{x}_i \mathbf{x}_i} \cdot \hat{\mathbf{y}} + (1 - \eta) \hat{\alpha}_N; \\ \hat{\alpha}_D &= \eta \hat{\mathbf{k}}^{\mathbf{x}_i \mathbf{x}_i} \cdot (\hat{\mathbf{k}}^{\mathbf{x}_i \mathbf{x}_i} + \lambda) + (1 - \eta) \hat{\alpha}_D. \end{aligned} \quad (2.10)$$

其中 $\hat{\alpha}_N$ 和 $\hat{\alpha}_D$ 分别为频率域 α 的分子和分母，即 $\hat{\alpha} = \hat{\alpha}_N / \hat{\alpha}_D$ 。值得注意的是，虽然所有历史帧中的目标图像块都在公式2.9中显式地考虑了，但它们隐含在了迭代更新过程（公式2.10）中，因此每次更新 α 时仅需要当前帧中的目标物体图像。而KCF模型的另一部分，目标外观 \bar{x} ，仍然按照公式2.8进行线性插值式的更新。

2.5 将“目标候选”嵌入跟踪器中

原KCF跟踪器在经过特征整合和鲁棒更新的优化后，将足以胜任对目标候选的辨别。因此本节将把目标候选生成器EdgeBoxes嵌入优化后的KCF跟踪框架中，以同时利用目标候选的灵活性和优化后的KCF的准确辨别力。嵌入目标候选生成器后的整个跟踪过程可视化地展示在了图2.5中。

在初始化时，第一帧中，记用户标注的或者目标检测得到的目标物体边界框

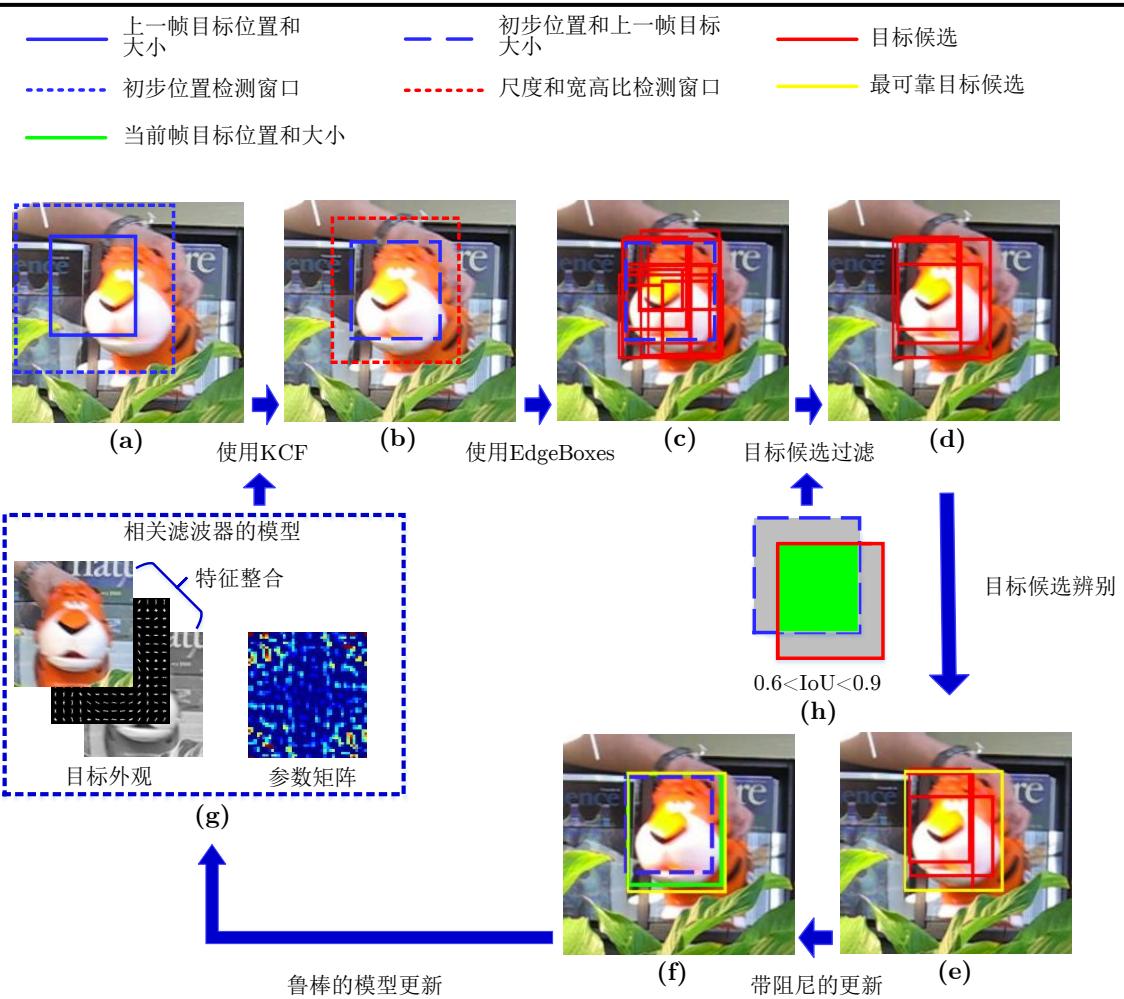


图 2.5 本章跟踪器的可视化跟踪过程

大小为 $w_1 \times h_1$ ，中心位于 \mathbf{l}_1 。将该边界框扩大为 $s^d w_1 \times s^d h_1$ ，即得到了位于 \mathbf{l}_1 的上下文窗口。这里的 s^d 被称为尺度因子，必须大于 1 以包含一定上下文信息，且覆盖下一帧中目标物体可能出现的位置。为了初始化 KCF 的模型，此时将根据该上下文窗口提取出图像块 \mathbf{x}_1 ，并用 \mathbf{x}_1 初始化参数矩阵 $\boldsymbol{\alpha}$ （公式2.4）。至于目标外观 $\bar{\mathbf{x}}$ ，将其直接初始化为 \mathbf{x}_1 。

跟踪过程开始后，每当新的一帧（记为第 i 帧）到来时，首先根据上一帧中的目标位置 \mathbf{l}_{i-1} 和目标大小 $w_{i-1} \times h_{i-1}$ ，设置一个中心位于 \mathbf{l}_{i-1} ，大小为 $s^d w_{i-1} \times s^d h_{i-1}$ 的“初步位置检测窗口”，如图2.5(a) 所示。然后根据该窗口提取图像块 \mathbf{z}^d 。由于跟踪过程中，检测到的目标物体大小会不断变化，因此需要利用双线性插值（Bilinear Interpolation）将 \mathbf{z}^d 缩放到 $s^d w_1 \times s^d h_1$ ，才能利用无尺度适应力的 KCF 相关滤波器。将 KCF 应用于图像块 \mathbf{z}^d 之后（公式2.6），即可根据 \mathbf{f} 中最大元素的位置，判断出新的目标位置 \mathbf{l}_i^d 。该目标位置称作“初步位置”，其对应的 \mathbf{f} 中最大元素值记为 v 。

下一步如图2.5(b)所示，以初步位置 \mathbf{l}_i^d 为中心， $s^e w_{i-1} \times s^e h_{i-1}$ 为大小，构建一个“尺度和宽高比检测窗口”。 s^e 也是一个尺度因子，但是应当设置得比 s^d 小，因为目标物体尺度的变化通常小于其位移。根据尺度和宽高比检测窗口提取图像块 \mathbf{z}^p 后，将其输入目标候选生成器 EdgeBoxes 中。如图2.5(c)所示，EdgeBoxes 将会输出大量的目标候选，即大量可能包含物体的边界框。这些边界框已按照各自所得“分数”进行了排序，而该分数代表的就是边界框内包含物体的置信度。

对于 EdgeBoxes 输出的边界框，本节仅提取其前 N 个，并且对它们进行过滤：计算每一个目标候选边界框与“初步目标物体边界框”的重叠率，如果重叠率大于 0.9 或者小于 0.6，则将该目标候选剔除。这里的初步目标物体边界框位于 \mathbf{l}_i^d ，大小为 $w_{i-1} \times h_{i-1}$ ，即一个位于初步位置，大小等于前一帧中目标大小的边界框。如图2.5(h)所示，重叠率使用 IoU (Intersection over Union) 进行度量，其计算方式为将两个边界框的交集面积除以它们的并集面积。IoU 大于 0.9 的边界框几乎和初步目标物体边界框相同，无需纳入考虑；而 IoU 小于 0.6 的边界框很有可能是错误的目标候选，或者其中包含的是非目标物体，均需要被剔除。

如图2.5(d)所示，通过过滤后的目标候选数目已大幅下降。但是，仍然需要用相关滤波器对这些目标候选进行再次辨别，才能找出图2.5(e)中所示的“最可靠目标候选”。对于每一个经过过滤的目标候选边界框，按尺度因子 s^d 进行扩大，并从当前帧中提取出对应的图像块，记为 \mathbf{p} 。由于 \mathbf{p} 可能具有任意的尺度和宽高比，因此也需要将它缩放到 $s^d w_1 \times s^d h_1$ 后，才能使用相关滤波器进行辨别。辨别一个目标候选图像块和当前目标外观的相似度时，使用下式：

$$f(\mathbf{p}) = \text{sum}(\mathbf{k}^{\bar{x}\mathbf{p}} \cdot \boldsymbol{\alpha}), \quad (2.11)$$

这里的 $\text{sum}(\cdot)$ 代表对矩阵中所有元素的叠加操作， $f()$ 和公式2.3中的 $f()$ 意义相同，是一个代表着 \mathbf{p} 和 \bar{x} 的相似度的标量。由于这里仅需要对目标候选图像块本身进行辨别，而无需辨别它的所有循环位移，因此用公式2.11来代替公式2.6。事实上，公式2.11就是公式2.6转换到空间域的结果。在辨别完所有目标候选之后，本节将找出具有最大 f 值（记为 f_{max} ）的目标候选，并记录其中心位置为 \mathbf{l}_i^p ，大小为 $w_i^p \times h_i^p$ 。

如果 f_{max} 小于 v ，说明最可靠的目标候选边界框的准确性仍然不如“初步目标物体边界框”。此时应放弃所有目标候选，认为初步位置 \mathbf{l}_i^d 就是当前帧中的目标位置 \mathbf{l}_i ，且当前目标物体大小 $w_i \times h_i$ 不变，仍然等于 $w_{i-1} \times h_{i-1}$ 。如果 f_{max} 大于 v ，则通过一个带阻尼的更新过程，将初步目标物体边界框和最可靠目标候选

相结合：

$$\begin{aligned} \mathbf{l}_i &= \mathbf{l}_i^d + \gamma(\mathbf{l}_i^p - \mathbf{l}_i^d); \\ (w_i, h_i) &= (w_{i-1}, h_{i-1}) + \gamma((w_i^p, h_i^p) - (w_{i-1}, h_{i-1})). \end{aligned} \quad (2.12)$$

上式的 γ 为阻尼因子，其作用如图2.5(f)所示。用带阻尼的更新来最终确定目标的位置和大小，可以防止过于敏感的目标状态变化，同时还能降低跟踪错误或是目标候选错误带来的影响，从而让跟踪器更为鲁棒。

在当前帧中完成跟踪后，还需要提取一个新的目标物体图像块来更新 KCF 的参数矩阵 α 和目标外观 \mathbf{x} 。显然，该图像块位于新目标位置 \mathbf{l}_i ，大小为 $s^d w_i \times s^d h_i$ ，可被记为 \mathbf{x}_i 。将 \mathbf{x}_i 输入公式2.10，即可鲁棒地更新 KCF 的参数矩阵 α ；而更新目标外观 \mathbf{x} 则需要根据公式2.8。完成模型更新后，将进入针对下一帧的循环。

除了图2.5所示的可视化过程，本节的跟踪过程还规范化地列在了算法2.1中。更多可视化的目标候选辨别和带阻尼更新的例子可见图2.6。图2.6还清楚地展示出了目标候选方法是如何让本章的跟踪器适应尺度和宽高比变化的。正是由于目标候选的极大灵活性，本章的跟踪器可以轻易地找出被遮挡的目标（图2.6右上），并且还能准确地框出旋转的目标（图2.6左下）。

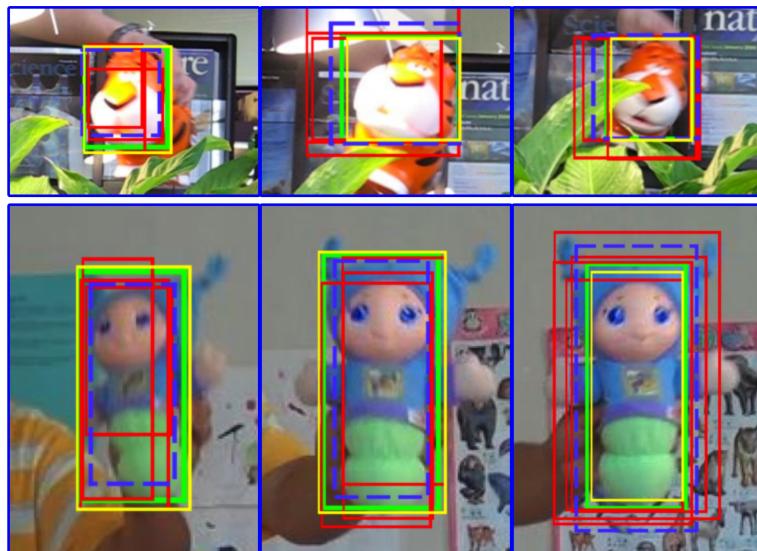


图 2.6 目标候选辨别和带阻尼更新的可视化例子 (图例见图2.5)

2.6 参数设置

从上文中可以看出，本章的跟踪算法引入了大量的参数。这些参数对跟踪器的性能有着极大的影响，因此参数设置也是跟踪算法设计中至关重要的步骤。简单粗暴地进行大范围的参数遍历，寻找能够在测试集上取得最优效果的参数组合，看似合理，但却是不可行的。如果参数数目较多，各参数取值范围较大，若再加

算法 2.1 对于第 i 帧的跟踪过程

输入:

F_i : 图像帧

$(l_{i-1}, w_{i-1}, h_{i-1})$: 在上一帧中紧密包围目标物体的边界框

α, \bar{x} : 之前的模型 (参数矩阵和目标外观)

输出:

(l_i, w_i, h_i) : 在当前帧中紧密包围目标物体的边界框

α, \bar{x} : 更新后的模型

估计初步位置:

- 1: 在 F_i 中, 按照 $(l_{i-1}, s^d w_{i-1}, s^d h_{i-1})$ 提取图像块 \mathbf{z}^d ;
- 2: 利用 \mathbf{z}^d 、 α 和 \bar{x} , 根据公式2.6, 计算出初步位置 l_i^d ;

估计尺度和宽高比:

- 3: 在 F_i 中, 按照 $(l_i^d, s^e w_{i-1}, s^e h_{i-1})$ 提取图像块 \mathbf{z}^p ;
- 4: 将 \mathbf{z}^p 输入 EdgeBoxes, 得到目标候选边界框集合 \mathbf{P} ;
- 5: 根据重叠率对 $\mathbf{P}[1, \dots, N]$ 进行过滤, 得到过滤后的目标候选边界框集合 \mathbf{P}' ;
- 6: 利用 α 和 \bar{x} , 根据公式2.11, 对 \mathbf{P}' 中的每一个边界框进行辨别, 得到具有最大相关滤波响应值的边界框 (l_i^p, w_i^p, h_i^p) ;
- 7: 利用 (l_i^p, w_i^p, h_i^p) 、 (w_{i-1}, h_{i-1}) 和 l_i^d , 根据公式2.12进行带阻尼的更新, 得到 (l_i, w_i, h_i) ;

模型更新:

- 8: 在 F_i 中, 按照 $(l_i, s^d w_i, s^d h_i)$ 提取图像块 \mathbf{x}_i ;
 - 9: 利用 \mathbf{x}_i , 根据公式2.10和公式2.8, 更新 α 和 \bar{x} 。
-

上测试集较大, 那么参数遍历的时间开销将不可接受。更重要的一点是, 针对特定测试集遍历得出的参数设置很可能是过拟合的, 难以适用于其它测试集或者实际场景。因此, 本节将尽量保留原版本 KCF 跟踪器和目标候选生成器 EdgeBoxes 的参数设置。对于少数不适于本章算法的参数设置, 则根据其实际意义进行微调。

在本章跟踪器的原 KCF 部分, 除了两个参数以外, 其它参数设置保持与原版 KCF 一致, 例如尺度因子 $s^d = 2.5$ (2.5节)、高斯核函数带宽 $\sigma = 0.5$ (公式2.5)、规则化参数 $\lambda = 0.0001$ (公式2.4、2.10)。需要调整的两个参数分别为回归目标矩阵 \mathbf{y} 的标准方差 (公式2.4、2.10) 和学习速率 η (公式2.10和公式2.8)。如图2.7所示, 由于回归目标矩阵符合二维高斯函数, 因此它的标准方差设置地越小, \mathbf{y} 的峰值部分就越陡峭, 相关滤波器的辨别规则就会越严格。如果相关滤波所使用的图像特征较为简单, 那么标准方差就不宜设置地太小。原因在于简单特征的不变

性（如旋转不变性、尺度不变性等）普遍较差，若辨别标准过于严格，在目标物体外观发生较大变化时就很可能无法发现目标，导致跟踪失败。然而本章通过使用图像特征整合，获得了不变性较强的混合特征。因此可以将标准方差适当减小，从“ $0.1 \times$ 图像对角线长度的平方根”，减小为“ $0.06 \times$ 图像对角线长度的平方根”。通过减小标准方差，在成功检测到目标的情况下，获得的目标物体边界框将更加准确。同样由于整合后的混合特征具有更强的不变性，学习速率 η 也从 0.02 下调到 0.01，使得跟踪器既能不断适应变化的目标外观，又不易受到目标遮挡和偶发跟踪误差的影响。

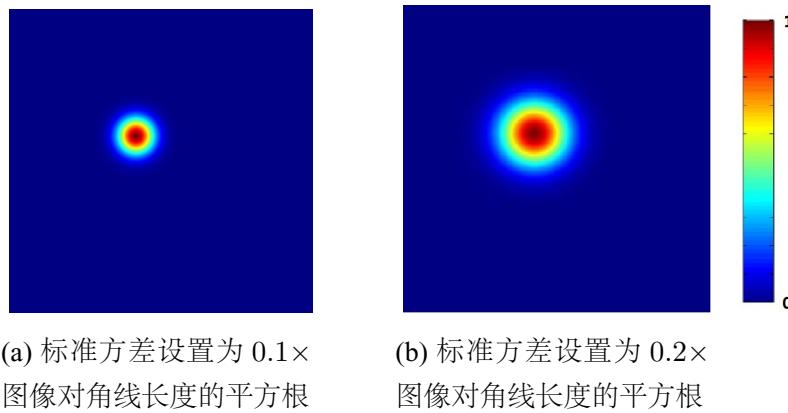


图 2.7 使用不同标准方差的相关滤波器回归目标矩阵

在目标候选生成器部分，新参数 s^e ，即“尺度和宽高比检测窗口”的尺度因子，被设置为 1.4。此处 1.4 为 $\sqrt{2}$ 的近似，它限制 EdgeBoxes 在一个两倍于目标图像块面积的窗口内寻找目标物体。该参数设置即是假设：两帧间目标物体外表的面积最多扩大一倍。

至于 EdgeBoxes 本身，本节仅作简要介绍以便于分析参数设置，更详细的分析和优化将在下一章进行。对于输入的图像，EdgeBoxes 首先利用“结构化边界（Structured Edge）提取器^[32]”进行边界提取，并计算出每个像素的边界响应值（属于物体边界的置信度）。然后它以“滑动窗口”的形式遍历整幅输入图像，并对每一个窗口进行评分。控制滑动窗口的参数有 $stepSize$ 、 $maxAspectRatio$ 和 $minBoxArea$ 等。 $stepSize$ 是滑动窗口时，两个相邻窗口的 IoU，用于控制采样密度。所有窗口的宽高比均被限制在 $1/maxAspectRatio$ 到 $maxAspectRatio$ 之间，而最小窗口的面积被限制为 $minBoxArea$ ，因此这两个参数被用于控制目标候选可能的形状和大小。对于每一个窗口，EdgeBoxes 的评分函数为：

$$h_b = \frac{\sum_{i \in b} w_i m_i}{2(b_w + b_h)^\kappa} - \frac{\sum_{p \in b^{in}} m_p}{2(b_w + b_h)^\kappa}. \quad (2.13)$$

式中, i 代表一个像素, 其边界响应值记为 m_i 。 b 即是待评分的窗口对应的图像块, 其宽度和高度分别为 b_w 和 b_h 。 b^{in} 是 b 的中心部分, 大小为 $b_w/2 \times b_h/2$ 。 $w_i \in [0, 1]$ 是一个权值, 代表着 i 所在的物体边界完全包含在 b 中的可能性。显然 b 的尺度越大, 其中包含的物体边界就越多, 因此 κ 用于惩罚较大的窗口。完成所有窗口的评分后, EdgeBoxes 将对得分高于 $minScore$ 的窗口进行进一步的细化和过滤, 然后作为目标候选输出。

设置 EdgeBoxes 自身的参数时, 本节基本保留其默认设置 (以 0.7 为期望 IoU), 如 $stepSize = 0.65$ 。但是, 参数 $minBoxArea$ 和 $maxAspectRatio$ 将根据上一帧中目标物体边界框的大小, 不断更新:

$$\begin{aligned} minBoxArea &= 0.3 \times w_{i-1} \times h_{i-1}; \\ maxAspectRatio &= 1.5 \times \max\left\{\frac{w_{i-1}}{h_{i-1}}, \frac{h_{i-1}}{w_{i-1}}\right\}. \end{aligned} \quad (2.14)$$

通过用上式控制滑动窗口的大小和形状, EdgeBoxes 生成目标候选的速度将大大加快, 并且避免了大量无用目标候选的产生。此外, 公式2.13中的 κ 从 1.5 轻微下调至 1.4。原因在于, 物体检测任务中的物体可能位于任意位置、具有任意大小; 但是在跟踪任务中, 仅需要找出目标物体, 且目标物体通常占据了输入图像块的大部分面积。因此 κ 需要适度减小, 以减轻对于较大目标候选的惩罚。本节对得分阈值 $minScore$ 的修改较大, 从 0.01 减小到了 0.0005。这是一种较为保守的参数设置, 目的在于确保跟踪目标所在的目标候选不被滤除。通过实际验证, EdgeBoxes 在每一帧中平均生成约 100 个目标候选 (最多时生成了 277 个)。因此进入目标候选过滤的最大边界框数量, 即算法2.1第 5 行的 N , 设置为 200。该参数设置既能保证获取到绝大部分目标候选, 还能将它们用定长矩阵进行存储, 提高计算效率。最后, 作为一个经验参数, 公式2.12中的阻尼因子 γ 设置为 0.7。

2.7 实验结果与分析

2.7.1 实验设置

为了表述方便, 这里将本章提出的, 融合了相关滤波器和目标候选生成器的新跟踪器命名为“KCFDP” (“Kernelized Correlation Filter with Detection Proposals”), 即“加入了目标候选的核化相关滤波器”。KCFDP 的代码主要使用 Matlab 实现, 部分功能如 HOG 特征提取, 直接利用了 PMT^[33] 中的实现。EdgeBoxes 的实现直接采用了其开源版本 (使用 C++ 实现), 并用 Matlab 进行了封装, 以便于嵌入跟踪器中。硬件平台的 CPU 为低功耗处理器 Intel i5-4278U, 2.6 GHz 主频; 内存为 8GB DDR3-1600。操作系统为 MacOS 10.11, 编译器采用 GCC 4.8, Matlab 版本为

2014a。实验将在著名的 OTB (Object Tracking Benchmark) [1, 34] 测试集以及它的两个子集中进行，结果评价标准采用通用的准确率图和成功率图，用于对比的跟踪器共计达到了 34 个。开放的大型测试集、通用的评价标准、以及庞大的对照组将共同保证本章实验的充分性和完备性。

2.7.1.1 测试集和对照组构成

本章的测试集来自著名的 OTB，它包括了 50 个极具挑战性的视频序列（共计 51 个跟踪目标），其概览如图2.8所示。OTB 还为每一个视频序列注明了其主要包含的跟踪障碍，例如“快速运动”、“遮挡”、“尺度变化”、“光照变化”等共计 11 种，且一个视频序列可能包含多种跟踪障碍。因此，将具有相同跟踪障碍的序列组成 OTB 的一个子集，即可用于测试跟踪器对该障碍的适应力和鲁棒性。OTB 提供的跟踪障碍中已有“尺度变化”，且共计有 28 个序列包含该障碍，因此这 28 个序列组成的子集将被用于评测 KCFDP 对于尺度的适应力。

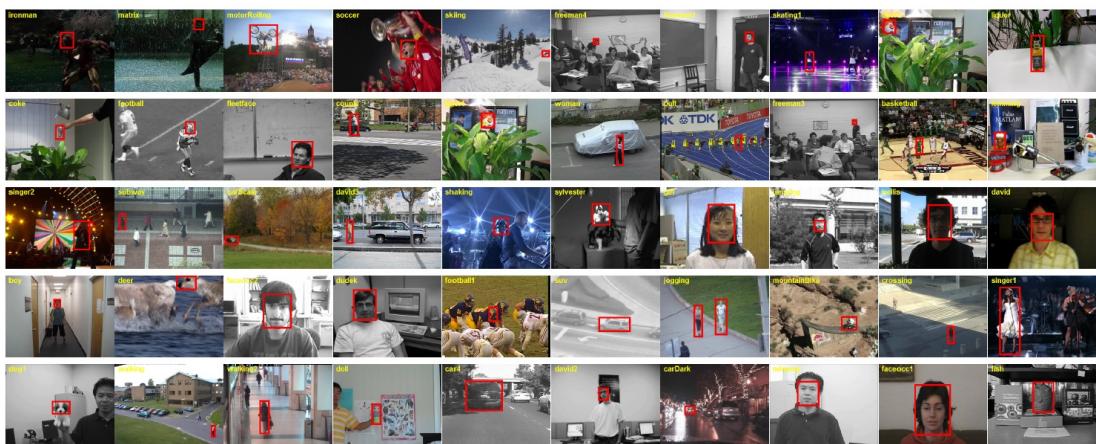


图 2.8 本章所使用的 OTB^[1] 测试集

但是，OTB 并没有标注“宽高比变化”这一障碍，因此本节将根据 OTB 给出的正确跟踪结果 (Ground Truth)，即每一帧中紧密包围目标物体的边界框真值，自行进行提取。判断一个视频序列包含“宽高比变化”障碍的标准如下。如图2.9所示，对于该视频的每一帧，将该帧的目标物体边界框和它的前 30 帧的目标物体边界框一一进行宽高比对比。如果某次对比中，宽高比的比值超过了 $[1/\sqrt{2}, \sqrt{2}]$ 这一区间，则认为当前帧“正在经历宽高比变化”。如果该视频序列有 10% 以上的帧“正在经历宽高比变化”，则认为该序列中包含“宽高比变化”这一跟踪障碍。根据该标准，一共有 14 个序列包含“宽高比变化”障碍，它们是：*Soccer*、*Matrix*、*Ironman*、*Skating1*、*Shaking*、*Couple*、*Girl*、*Walking2*、*Walking*、*Freeman3*、*Freeman4*、*Skiing*、*MotorRolling* 和 *Woman*。这些序列共有 4560 帧，其中 1017 帧“正在经历宽高比变化”。它们组成的子集将被用于评测 KCFDP 对于宽高比变化的适应力。

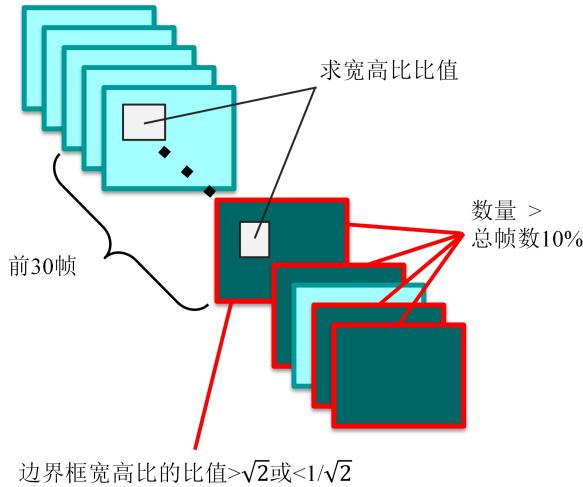


图 2.9 判断视频序列包含“宽高比变化”的过程

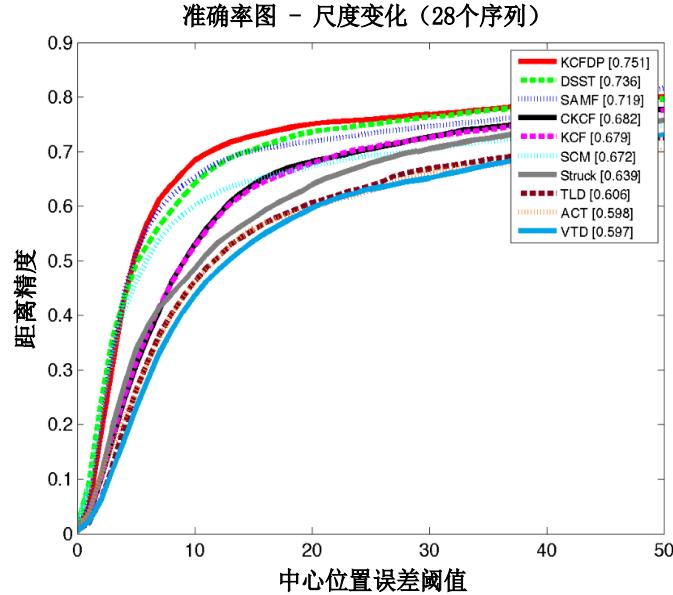
OTB 测试集除了提供了 50 个视频序列和正确的跟踪结果，还额外提供了 29 个跟踪器在它之上的运行结果。这 29 个跟踪器中不乏 SCM^[22]、Struck^[35]、TLD^[25]、VTD^[23] 和 ALSA^[21] 等当前最为先进的跟踪器。本节将直接利用这些跟踪器的运行结果，与 KCFDP 的结果进行对比评测。此外，还有 5 个基于相关滤波的跟踪器被包含在对照组中，即 KCF^[9]、DSST^[10]、ACT^[11]、SAMF^[12] 和 CKCF。其中 CKCF 就是去除了目标候选生成器的 KCFDP，也即是加入了图像特征整合和鲁邦更新的 KCF。综上，对照组中将共有 34 个跟踪器。

2.7.1.2 评价标准

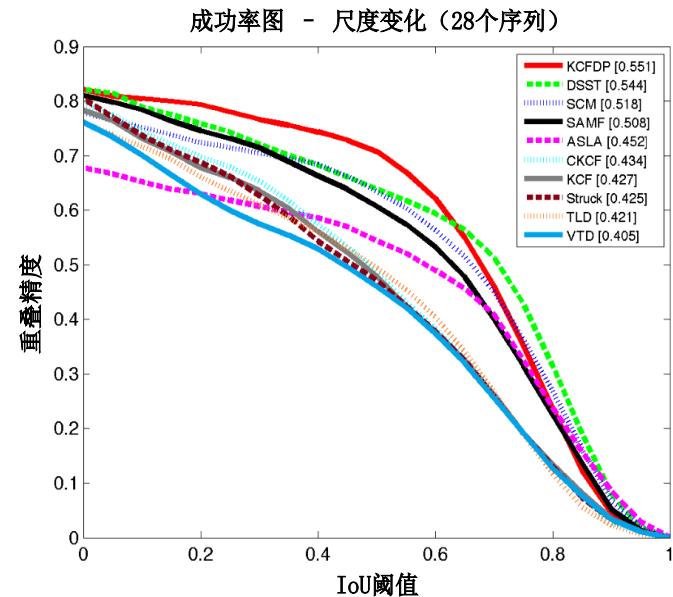
在一些早期的工作中，通常使用平均中心位置误差 (Averaged CLE) 或者平均重叠率 (Averaged IoU) 来评价跟踪器的准确性。平均中心位置误差即是在视频序列的所有帧中，计算跟踪器结果和正确跟踪结果中心间的距离，并进行平均。显然，中心位置误差无法反映出尺度和宽高比是否准确。平均重叠率进了一步，它用跟踪器结果和正确跟踪结果的 IoU 来替代中心位置误差。但是，这两者最大的问题在于，当跟踪器丢失目标时，会对跟踪结果进行随机的惩罚，从而都无法准确反映跟踪器的准确性。

因此，本节将沿用 [1] 中提出的准确性评价标准，即使用准确率图 (Precision Plot) 和成功率图 (Success Plot)。要得到准确率图，需要先计算距离精度 (Distance Precision)。距离精度的定义为：视频序列中中心位置误差小于某个阈值的帧所占的比例。而成功率图则依赖于重叠精度 (Overlap Precision)，其定义为：视频序列中 IoU 大于某个阈值的帧所占的比例。准确率图展示的是，在不同的中心位置误差阈值下 (沿横坐标)，跟踪器在所有序列上的平均距离精度 (沿纵坐标)。类似的，成功率图展示的是在不同的 IoU 阈值下 (沿横坐标)，跟踪器在所有序列上的

平均重叠精度（沿纵坐标）。对于准确率图，跟踪器将按照以 20 像素为阈值时的距离精度进行排名。而对于成功率图，跟踪器将按照“曲线下方面积”（Area Under Curve, AUC）进行排名。

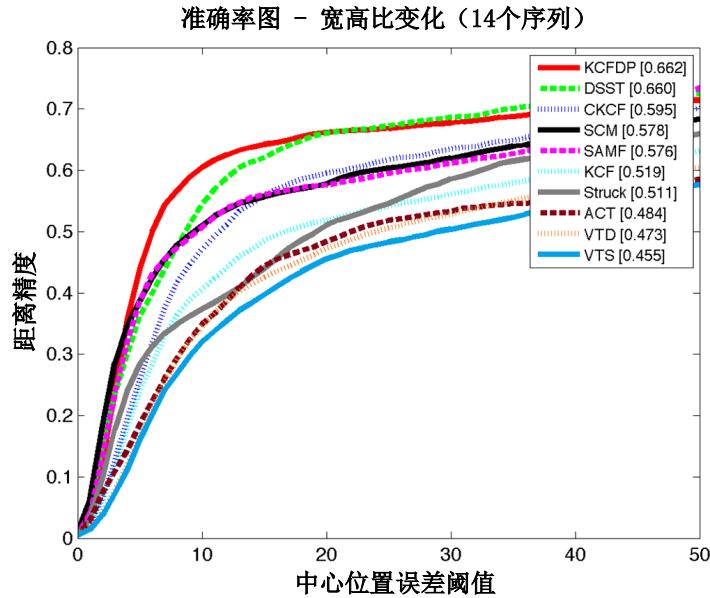


(a) 针对尺度适应力的准确率图

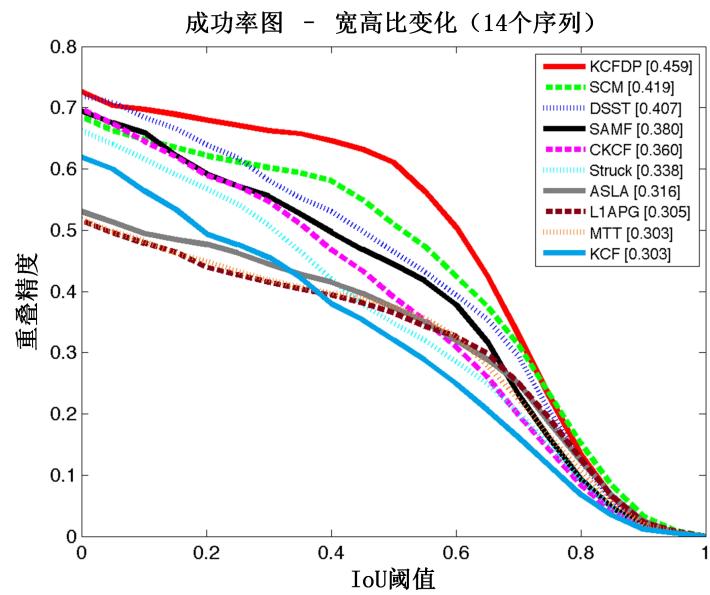


(b) 针对尺度适应力的成功率图

图 2.10 针对尺度适应力的实验结果



(a) 针对宽高比适应力的准确率图



(b) 针对宽高比适应力的成功率图

图 2.11 针对宽高比适应力的实验结果

2.7.2 尺度和宽高比适应力评测

为了评测本章方法的尺度适应力，本节将所有的 34 个跟踪器，加上 KCFDP，应用于包含“尺度变化”这一跟踪障碍的 28 个视频序列上进行跟踪。得到的准确率图和成功率图见图 2.10。同理，为了评测 KCFDP 的宽高比适应力，本节将所有跟踪器应用于上一节提取的，包含“宽高比变化”障碍的 14 个视频序列上，得到

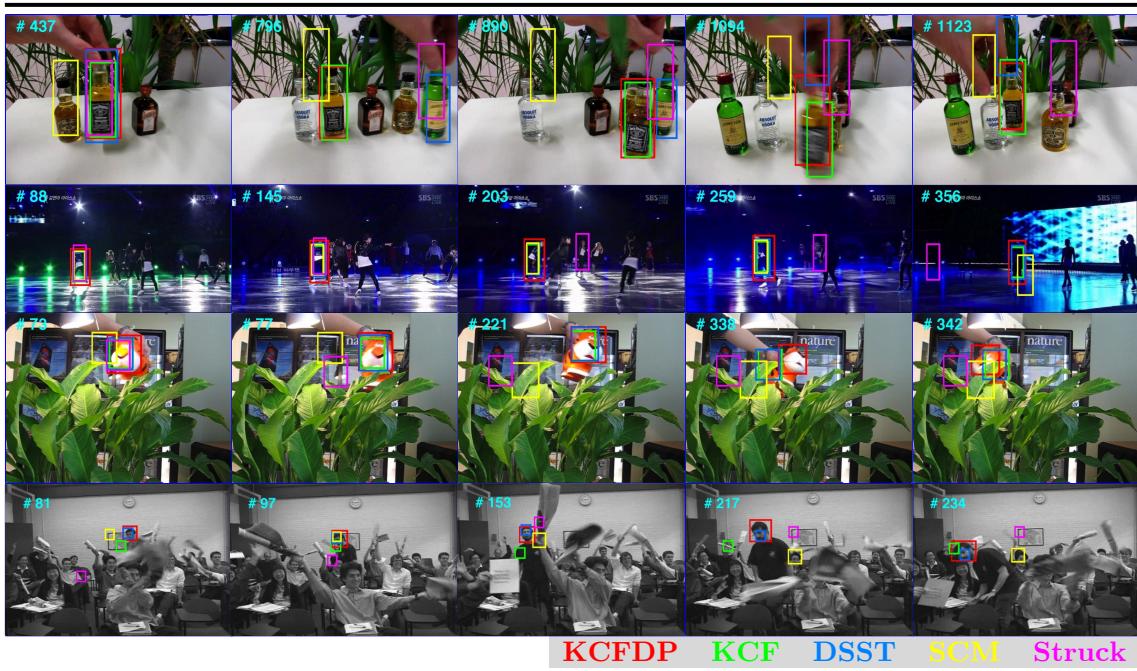


图 2.12 KCFDP 与 4 个跟踪器的可视化对比

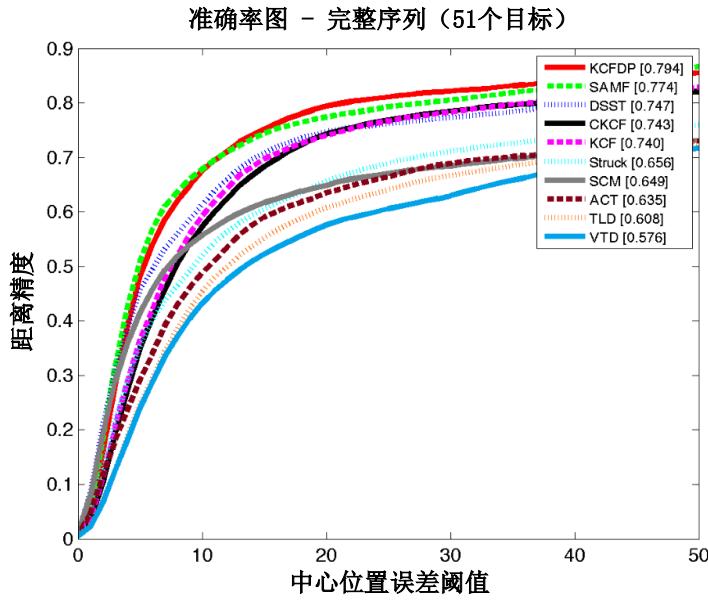
了如图2.11所示的准确率图和成功率图。在图2.10和图2.11中，为了显示清晰，只给出了排名前 10 的跟踪器的对应曲线。各图中，右上角的图例中的排名即是跟踪器的准确性排名，方括号内的数值是排名的依据，即以 20 像素为阈值时的距离精度（准确率图中），或者曲线下方的归一化面积（成功率图中）。

从图2.10和图2.11中可以清晰地看出，无论对于尺度变化还是宽高比变化，本章提出的跟踪方法均具有最佳的适应力。此外，KCFDP 在成功率图中的优势更为明显。这证明了目标候选方法主要提高了跟踪结果和正确结果间的重叠率，即是指，跟踪结果的尺度和宽高比更加准确了。

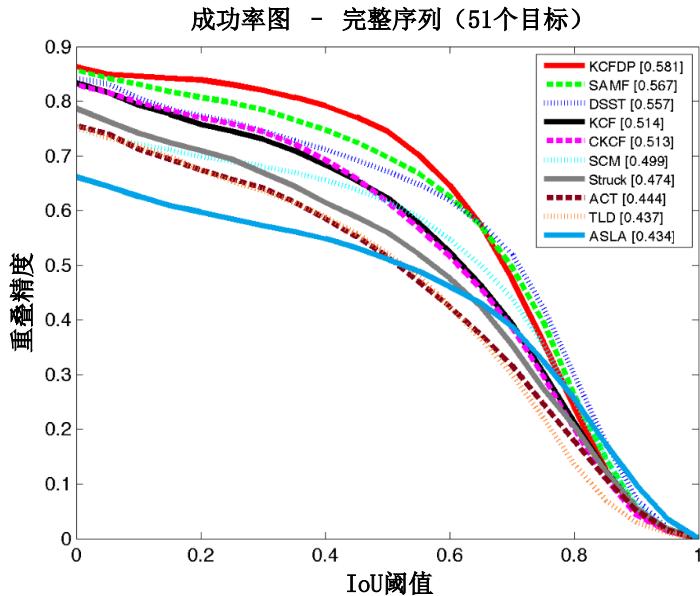
2.7.3 整体性能评测

为了评测 KCFDP 对于各种跟踪障碍的适应力和鲁棒性，以及得出其跟踪效率，本节将在完整的 50 个视频序列（包含 51 个跟踪目标）下对比所有的跟踪器。图2.12展示了在 4 个较为困难的视频序列 (*Liquor*, *Skating1*, *Tiger1* 和 *Freeman4*) 上，排名前 5 的跟踪器的直观可视化对比。可以清楚地看出，KCFDP 在准确地适应尺度和宽高比变化的同时，还能保持很高的跟踪精度。

将所有跟踪器运行于完整的 50 个视频序列上所得的准确率图和成功率图见图2.13。显然，在各种跟踪障碍下，KCFDP 仍然超过了所有其它跟踪器和相关滤波跟踪器变型。更进一步地分析这两节的实验结果，可以看出在图2.10、2.11、2.13 中，KCFDP 与 CKCF 之间都有着明显的性能差距。这充分显示了在跟踪过程中加入目标候选所带来的准确性提升。尽管 CKCF 和 KCF 在“尺度变化”障碍下



(a) 完整的 50 个视频序列上的准确率图



(b) 完整的 50 个视频序列上的成功率图

图 2.13 完整的 50 个视频序列上的实验结果

和整个测试集下的性能区别很小，但是对于“宽高比变化”，CKCF 具有明显的优势。因此可以推断，整合后的混合特征仅在候选图像块出现密集形状变化时才具有较大优势。这也是本章在 KCFDP 中使用混合特征的原因，即用于辨别尺度和宽高比不断变化的目标候选。此外，相比在整个测试集和“尺度变化”子集中，KCFDP 在“宽高比变化”子集下的优势明显最大。原因在于，每个序列都包含着多个跟踪障碍，而一些障碍会导致目标候选生成器失效或者出错，从而部分

抵消掉它带来的性能提升。例如“运动模糊”会导致目标物体的边界变得模糊，严重影响依赖物体边界来生成目标候选的 EdgeBoxes。并且在“尺度变化”子集下，DSST 和 SAMF 等跟踪器也具有特定的尺度适应力优化，故 KCFDP 的相对优势略有降低。

除了很高的跟踪精度以外，KCFDP 的跟踪效率也令人满意，在整个测试集上达到了平均 20.8 FPS 的跟踪速度。考虑到 KCFDP 主要使用较低效的 Matlab 进行实现，且运行于低功耗处理器上，本节认为对实现稍作优化后，就完全可以达到实时跟踪的需求。作为对比，在完全相同的实验环境下，DSST 和 SAMF 这两个基于相关滤波的跟踪器的跟踪速度分别为 28.9 FPS 和 12.0 FPS，但是他们均不具有宽高比适应力。KCFDP 跟踪效率较高的原因在于，相关滤波器实际上仅需要额外辨别很少量的目标候选，且经过调优后的目标候选生成过程也十分高效。

2.8 小结

本章提出了一个在视觉物体跟踪中提高尺度和宽高比适应力的全新方法。通过对一个“物体类别无关”的目标候选生成器进行调优，并配合目标候选过滤过程，本章的方法能够为相关滤波器提供不同尺度和宽高比的候选输入图像块。为了能够准确辨别这些目标候选，本章还利用图像特征整合和更鲁棒的模型更新策略来加强了相关滤波器。得益于目标候选的高度灵活性和相关滤波器的准确辨别力，本章得到的跟踪器在一个大型公开测试集上展现出了很强的鲁棒性和适应力，同时还达到了令人满意的跟踪速度。本章的方法是一个通用方法，在下一章中，该方法将用于把更多的目标候选生成器和跟踪器相结合，以深入研究目标候选在视觉物体跟踪中的作用和潜力。此外，本章并未对目标候选生成算法进行优化，下一章将针对跟踪任务的需求，对目标候选生成器进行深入改进。

第三章 跟踪器中“目标候选”的作用分析和优化

3.1 引言

更具体地介绍目标候选。说明和上一章的关系。分点介绍本章讲什么。

3.2 相关研究

3.2.1 目标候选生成器及其在跟踪中的应用

介绍典型目标候选生成器。

介绍使用了目标候选的典型跟踪器。

3.2.2 图像分割在跟踪中的应用

介绍图像分割和目标候选的关系。

介绍使用了图像分割的典型跟踪器。

3.3 跟踪器中目标候选生成器的适配

ijcv 论文。介绍目的（为了分析目标候选的作用），介绍 5 个目标候选生成器，介绍如何修改它们以嵌入跟踪器中

3.4 跟踪器中目标候选生成器的优化

介绍目的（优化一个目标候选生成器作为“跟踪候选”生成器）

3.4.1 目标候选生成器 EdgeBoxes

翻译并简化 EdgeBoxes 论文的核心部分。

3.4.2 使用背景抑制优化 EdgeBoxes

ijcv 论文

3.5 实验评测与分析

3.5.1 实验设置

主要是参数设置

3.5.2 跟踪器中目标候选的作用分析

ijcv 论文，对比不同目标候选生成器的准确度和速度

3.5.3 跟踪器中目标候选的优化效果评测

3.5.3.1 统计图对比评测

以下均翻译 ijcv 论文

3.5.3.2 数值化对比评测

3.5.3.3 VOT 测试集上的对比

3.5.3.4 参数敏感性分析

3.6 小结

第四章 基于 OpenCL 的 TLD 算法高性能实现

4.1 引言

在前两章中，通过在算法层次上优化当前具有领先精度的跟踪器 KCF，以及适用于视觉跟踪的目标候选生成器 EdgeBoxes，并将两者有效结合起来，取得了令人满意的视觉跟踪性能。但是，随着机器视觉应用对精度和鲁棒性的要求不断提高，跟踪器的结构正日趋复杂，计算负载与日俱增，高质量、高性能的跟踪器实现变得越来越关键。这一点在视觉跟踪领域顶级竞赛 VOT (Visual Object Tracking) 中也得到了印证。作为第一届 VOT 竞赛，VOT2013^[36]选取了 16 个视频序列，每个序列突出一个视觉跟踪中的难点（如遮挡，光照变化等）。首届获胜的跟踪器 PLT^[37] 采用二值化特征作为目标描述，并用查找表的方式实现了线性分类器，因此其 C++ 实现达到了 169.59 FPS 的极高速度。VOT2014^[38] 将视频序列扩充为 25 个，且标记目标物体的矩形框允许旋转，对跟踪器提出了更高难度的挑战。该届获胜的 DSST^[10] 由互相配合的两个相关滤波器构成，一个用于判断目标中心位置，一个用于判断最佳目标大小。结构的简洁性和相关滤波的低计算量使得其 Matlab 版本也能够做到实时处理（24 FPS）。VOT2015^[39] 的视频序列达到了 60 个，且难度大幅增加。为了应对挑战，MDNet^[40] 采用了专门面向视觉跟踪的 6 层卷积神经网络（CNN），并按照高斯分布大量采样不同位置、不同大小的局部图像进行检测。尽管 MDNet 在精度和鲁棒性上取得了第一名，但是由于它的计算量很大，实现（Matlab+MatConvNet）又较为粗糙，仅能达到 1 FPS 的跟踪速度，无法满足实时在线跟踪的要求。

由此可见，在跟踪算法日益复杂的趋势下，为了保证跟踪算法的实用性，或者说为了提高视觉跟踪对计算复杂性的容忍度，研究跟踪算法的高性能实现十分必要。这里的高性能实现，和“高性能计算”属同一范畴，意指在实现过程中针对硬件体系结构特征，充分发挥计算设备 / 平台的计算能力。通过分析体系结构和高性能计算的发展可以看出，并行化无疑是解决高性能实现问题的关键。利用多核并行、向量指令、流水化等并行手段，可以充分发挥通用处理器（CPU）的计算能力。除了 CPU 这一经典计算设备，当前包含两类甚至多类计算设备的异构计算平台正方兴未艾。从 SoC 芯片（如 Qualcomm Snapdragon 835^[41]）到嵌入式平台（如 Nvidia Jetson^[42]），甚至到超级计算机（如天河 -1A^[43]），CPU+GPU 的组合已成为一种潮流。要发挥异构平台的计算潜力，除了充分开发算法中的可并行部分以外，如何同时发挥 CPU 和 GPU 的性能，以及如何控制两者间的协作和数据传输等是高性能实现面临的又一挑战。

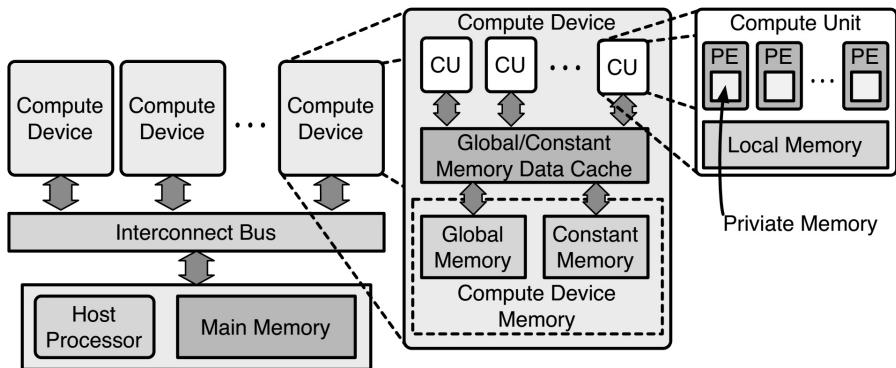
本章将以 TLD^[25, 26] 跟踪算法为例，以 OpenCL^[44] 作为编程模型，阐述视觉跟踪算法在异构平台下的高性能实现所需的关键技术和面临的问题挑战。严格来讲，TLD 算法并非是一个单纯的视觉跟踪算法，而是一个“长时间鲁棒跟踪框架”。它在传统跟踪算法中引入了独立的学习和检测模块，将跟踪 - 学习 - 检测三部分有机结合起来：跟踪模块仅根据前一帧判断目标在当前帧中的位置；学习模块构建一个可靠的目标描述，并不断更新它；检测模块在整个帧图像中找出可能包含目标的区域，用以修正跟踪结果。这样的结构，使得 TLD 比传统跟踪器鲁棒得多，且能适应长时间的目标跟踪。此外，除了 TLD 本身提供的 3 个模块算法，各个模块都可以自由替换为具有类似功能的算法，如将跟踪模块替换为 KCF 跟踪器，将学习模块和检测模块分别替换为 RCNN^[45, 46] 的训练过程和检测过程等。因此，研究 TLD 的高性能实现兼具实用性和理论研究价值，未来应用前景广阔。另一方面，相比其它并行编程模型如 CPU 上的 OpenMP^[47]、GPU 上的 CUDA^[48] 等，OpenCL 是跨平台的，其具备的功能移植性可以使得同一程序（如同一跟踪算法或者同一算法模块）不经修改地在不同异构设备上正确地执行，从而方便地利用异构计算平台下的各种计算设备。因此 OpenCL 是在异构计算平台下进行高性能跟踪算法实现的一个合适选择。

本章内容安排如下：第 2、3 节分别介绍 OpenCL 编程模型和 TLD 跟踪算法流程；第 4 节介绍 TLD 检测模块的第一部分——Fern 随机森林的高性能实现；第 5 节介绍检测模块的第二部分——最邻近（Nearest Neighbor, NN）分类器的高性能实现；第 6 节介绍学习模块中的瓶颈部分的高性能实现；第 7 节对高性能实现的效果进行评测；最后在第 8 节进行本章总结。

4.2 OpenCL 编程模型

当前，异构计算平台，即包含了多种异构计算设备（CPU、GPU、DSP 等）的计算系统，已经非常普及。用传统方法编写高性能程序已较为困难——面向多核 CPU 的编程通常使用共享存储模型，主要开发多核间的线程级并行；而面向 GPU 的编程通常关心复杂的存储层次和数据级并行。巨大的编程方式差异，导致程序员难以用一种代码来开发多种异构设备的计算能力。

OpenCL（Open Computing Language，开放计算语言）是一个专门面向异构系统通用目的的编程模型标准。苹果公司于 2008 年 6 月首次提出，Khronos 工作组于 2010 年 6 月正式发布。它的设计初衷就是面向异构计算平台，提供一个统一的编程模型。程序员们只需编写一个程序，就可以方便地利用异构平台的所有计算资源。为了达到这一目的，OpenCL 屏蔽了硬件设备的体系结构差异，并用四个抽象模型来描述编程方式和其它细节。

图 4.1 OpenCL 的平台模型和存储模型^[2]

- 平台模型 (Platform Model)

平台模型是 OpenCL 的抽象低层硬件结构。通过将硬件结构抽象地更为低层，暴露出更多的硬件细节，从而留出更多的编程和优化灵活性；而抽象性又屏蔽了体系结构差异，保证了程序的可移植性。

如图4.1所示，OpenCL 的平台由宿主处理器（Host Processor）和与之互联的一个或者多个计算设备（Compute Device）构成。每个计算设备又包含一个或者多个计算单元（Compute Unit），而一个计算单元又包含一个或多个处理单元（Processing Element）。处理单元作为最细粒度的计算承载单位，是一个抽象的标量处理器。

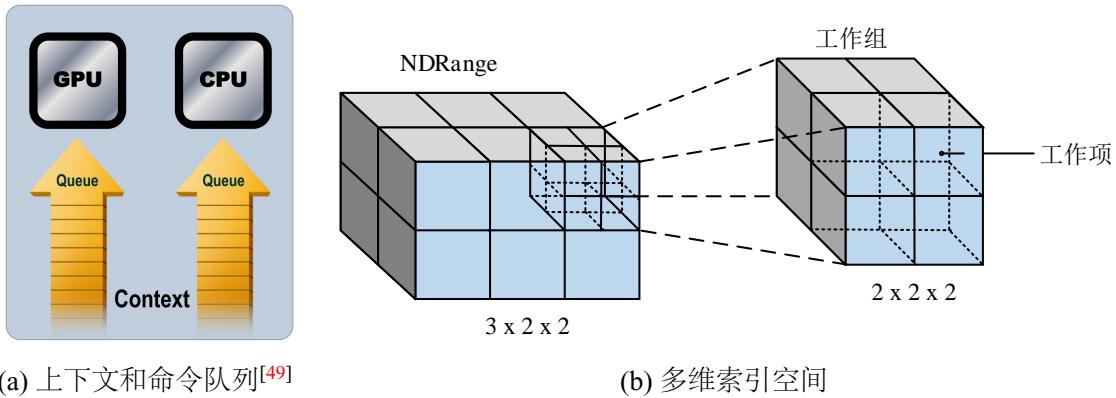


图 4.2 OpenCL 的执行模型

- 执行模型 (Execution Model)

一个 OpenCL 程序包括两大部分——宿主程序（Host Program）和 Kernel 程序（或称内核程序）。

宿主程序用 C 或者 C++ 编写，运行于宿主处理器之上，通过提交命令（Command）的方式来控制计算设备中的处理单元进行计算，或是控制宿主处理器和计算设备之间的数据传输。如图4.2(a)所示，宿主程序会定义一个上下文

(Context)，上下文中包含了可用的计算设备、待执行的 Kernel 程序、存储空间等信息。宿主程序还会为每一个计算设备分配一个或者多个命令队列 (Command Queue，或者 Queue)，并向命令队列中添加命令。随后命令队列就会调度命令进行执行。命令有三种类型：执行 Kernel 程序，数据传输，以及同步。

Kernel 程序实质上是一个函数，用 OpenCL C 语言编写，执行于计算设备之上。宿主程序通过提交一个执行 Kernel 的命令，来启动 Kernel 程序在计算设备上的执行。一个 Kernel 程序有众多的执行实例，对应于一个抽象的多维（一维到三维）索引空间 (NDRange Index Space)。一个 Kernel 的执行实例对应一个工作项 (Work-Item)，它是索引空间中的一个最细粒度点，代表着 Kernel 的一次执行，并根据其在空间中的位置被赋予一个全局索引值 (Global ID)。每个工作项都执行相同的 Kernel 程序，但是由于索引值的不同，各自的执行路径和访问的数据也会不同。多个相邻工作项组成一个工作组 (Work-Group)，是对索引空间的更粗粒度分解。根据在索引空间中的位置，工作组也会被赋予一个组索引值 (Group ID)。而每个工作组内的工作项，还会根据其组内位置被赋予局部索引值 (Local ID)。OpenCL 规定，一个工作组执行于一个计算单元之上，且同一工作组内的工作项是并发执行在计算单元中的处理单元上的。如图4.2(b)所示，为一个三维的索引空间。该空间包含了 $3 \times 2 \times 2$ 个工作组，而每个工作组包含 $2 \times 2 \times 2$ 个工作项。

- **存储模型 (Memory Model)**

OpenCL 的存储模型定义了四个独立的存储空间，分别位于不同的抽象硬件层次之上，且跟执行模型紧密相关。如图4.1所示，全局存储 (Global Memory) 位于计算设备中，任何工作组中的任何工作项都可以读写其中的数据。常量存储 (Constant Memory) 是全局存储的一部分，但是其中数据只能被读出而不能被修改。计算设备还可能提供用以提升全局存储和常量存储性能的高速缓存 (Data Cache)。局部存储 (Local Memory) 通常位于计算单元之中，仅属于某一个工作组，只能被该工作组中的工作项访问，而对其它工作组来说是不可见的。私有存储 (Private Memory) 通常位于处理单元内，是一个工作项所私有的，其它工作项都不可见。

宿主程序可以通过提交数据传输类的命令，来实现宿主处理器的主存 (Main Memory) 和计算设备的全局存储 / 常量存储间的数据传输。

- **程序模型 (Programming Model)**

这里的程序模型，指的是并行程序的实现方式。OpenCL 支持两类并行程序模型，数据并行和任务并行，以及这两类的混合编程。

数据并行是指一个指令序列同时作用于多个存储元素 (数据) 之上，它是通过定义执行模型中的多维索引空间来实现的。所有工作项执行相同的 Kernel 程序，

即同一个指令序列；而各个工作项在索引空间中有着不同的索引值，导致根据索引值的访存会访问到不同的数据。这样一来，同一 Kernel 程序将同时处理不同数据，实现数据并行。

任务并行是指不同指令序列的同时执行。实现任务并行，通常是通过在一个或者多个命令队列中，添加多个 Kernel 执行命令来实现的。

4.3 TLD 跟踪算法

TLD 的命名来自于跟踪 (Tracking) - 学习 (Learning) - 检测 (Detection) 三部分的首字母。将这三部分有机结合，正是 TLD 相比于传统跟踪算法的最大区别。如图4.3所示，跟踪模块负责计算目标物体在连续两帧间的运动，并提供一个目标候选位置。它假设目标物体在连续两帧间的运动幅度较小，且目标总是可见的。因此当目标被遮挡或是离开视野时，跟踪模块通常会失效且无法自行恢复。检测模块将每一帧都看作是独立的，并在整个帧图像中进行搜索检测，从而提供一到多个可能包含目标物体的候选位置。因此当跟踪模块失效时，检测模块可以重启跟踪。学习模块会综合考虑跟踪和检测的结果，以最终确定目标物体的位置。此外，它还会为检测模块提供新的训练数据，以不断提升检测的准确度。

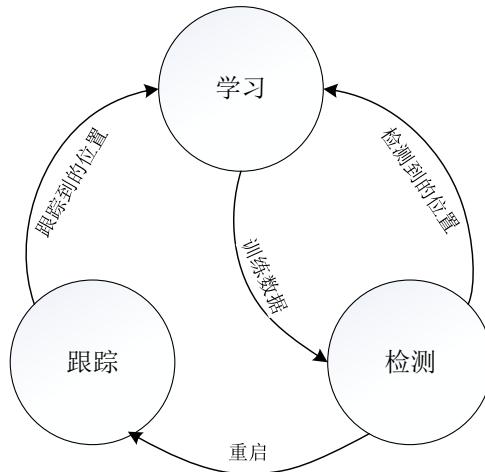


图 4.3 TLD 跟踪算法的三大部分

TLD 的形式化流程如算法4.1所示。更具体地，TLD 算法在逐帧运行之前需进行初始化。初始化的首要任务是构造网格 (Grid)，即大量的边界框 (Bounding Box)。网格通过用不同的尺度和步长来扫描整幅帧图像得到，与当前帧内容无关，仅与帧大小和目标初始尺度相关。逐帧运行过程中，检测模块将检测网格中的每一个边界框，找出其中可能包含目标物体的。

函数 `Tracker()` 的主要部分是 LK (Lucas-Kanade) 光流跟踪^[3, 50]，属于跟踪模块。LK 光流跟踪算法会运行两次，第一次用于估计目标物体从 $frame[i - 1]$ 到

算法 4.1 TLD 跟踪算法 (对于第 i 帧)

已知:

当前帧图像 $frame[i]$,
 前一帧图像 $frame[i - 1]$,
 目标物体在前一帧中的位置 (边界框) $lastBB$;

求:

目标物体在当前帧中的位置 (边界框) $currentBB$;

```

1:  $trackedBB \leftarrow Tracker(frame[i - 1], frame[i], lastBB)$ 
    • 跟踪模块根据目标物体在前一帧中的位置, 计算其在当前帧中的位置, 得到  $trackedBB$ , 失效时  $trackedBB$  为 NULL;
2:  $detectedBBs \leftarrow Detector(frame[i])$ 
    • 检测模块在当前帧中检测出所有可能包含目标物体的边界框, 存入  $detectedBBs$ ;
3:  $clusteredBBs \leftarrow Cluster(detectedBBs)$ 
    • 将  $detectedBBs$  中互相靠近的边界框合并为单个框, 存入  $clusteredBBs$ ;
4: if  $trackedBB \neq \text{NULL}$  then
5:     for each  $clusteredBB$  in  $clusteredBBs$  do
6:         if  $Overlap(clusteredBB, trackedBB) < 0.5$  and  $NNConf(clusteredBB) > NNConf(trackedBB)$  then
7:              $confidentBBs \leftarrow clusteredBB$ 
8:         end if
9:     end for
    • 从  $clusteredBBs$  中选出距离  $trackedBB$  较远, 但是置信度更高的边界框, 存入  $confidentBBs$ ;
10:    if  $Sizeof(confidentBBs) = 1$  then
11:         $currentBB \leftarrow confidentBBs[0]$ 
            • 如果  $confidentBB$  是唯一的, 则认为跟踪模块出错, 重启跟踪模块;
12:    else
13:         $currentBB \leftarrow WeightedAverage(trackedBB, detectedBBs)$ 
            • 否则, 以  $trackedBB$  为主, 将  $trackedBB$  和它附近的  $detectedBBs$  进行加权平均, 得到当前目标位置;
14:    end if
    • 跟踪模块有效时,  $trackedBB$  和  $detectedBBs$  共同决定  $currentBB$ ;
15: else
16:     if  $Sizeof(clusteredBBs) = 1$  then
17:          $currentBB \leftarrow clusteredBBs[0]$ 
18:     end if
            • 跟踪模块失效时, 若存在唯一的  $clusteredBB$ , 则用它来重启跟踪模块;
19: end if
20: if  $trackedBB \neq \text{NULL}$  and  $Validated(trackedBB)$  then
21:      $Learn(currentBB, frame[i])$ 
22: end if
    • 若  $trackedBB$  通过验证, 则学习模块根据  $currentBB$  提取正负样本, 训练检测模块。

```

frame[i] 的运动；第二次逆向运行，根据估计到的 *frame[i]* 中位置，计算物体在 *frame[i - 1]* 中的原位置。通过对比 *lastBB* 和两次 LK 光流计算的结果，可以判断出 LK 光流跟踪是否成功，即跟踪模块是否有效。

函数 *Detector()* 主要包括两部分，Fern 随机森林^[51] 分类和最邻近分类(Nearest Neighbor Classifier)，均属于检测模块。首先，Fern 随机森林将作用于网格中的大量边界框，并得出响应值。该响应值反映了边界框恰好包围目标物体的概率。随后，响应值高于阈值的边界框将通过最邻近分类，计算出置信度 (*NNConf()*)。最邻近分类的模型为一组正负样本图像块，置信度的计算过程为将输入图像块与正负样本逐一进行相似度对比。这里的相似度使用 NCC (Normalized Cross-Correlation，归一化互相关) ^[52] 来度量，而置信度由输入图像块与正样本的相似程度，以及它与负样本的不相似程度共同决定。置信度高于阈值的边界框才能作为 *Detector()* 的输出。函数 *Cluster()* 将 *Detector()* 输出的边界框进行聚类，把互相靠得太近的多个边界框合并为一个，以进一步减少数目。

目标物体在当前帧中的位置最终由学习模块来确定。如果跟踪模块有效，则综合考虑跟踪和检测的结果：若 *clusteredBBS* 中存在且仅存在一个置信度高于 *trackedBB* 的边界框，则认为跟踪模块出错，用检测结果替代跟踪结果；否则，将跟踪结果与它附近的检测结果进行加权平均，作为最终目标位置。如果跟踪模块失效，未避免歧义，当且仅当 *clusteredBBS* 中有唯一边界框时，用来作为最终目标位置。

学习模块还负责决定进行学习的时机，即跟踪结果足够可靠时——跟踪模块有效，且 *trackedBB* 通过了验证。*Validated()* 函数在两种情况下认为验证通过：*trackedBB* 经过最邻近分类得到的置信度足够高；或者在某一历史帧中置信度足够高，且从那时起跟踪模块从未失效或出错。学习过程 *Learn()* 分为两步，提取正负样本和再训练检测模块。提取正负样本前，需要先计算网格中所有边界框与 *currentBB* 的重叠率 (*Overlap()*)。对于 Fern 随机森林，选取重叠率低且响应值较高的边界框作为负样本，即“难反例挖掘 (Hard Negative Mining)”；同时选取重叠率高的边界框，并进行随机旋转缩放，作为正样本。对于最邻近分类器，从 *detectedBBS* 中选取重叠率低于阈值的边界框作为负样本；同时以重叠率最高的边界框作为唯一正样本。再训练 Fern 随机森林的过程，即是更新每一棵随机树的叶节点权值的过程。通过记录落入每一个叶节点的正负样本总数，更新其权值。再训练最邻近分类器的过程，即是验证正负样本是否与当前模型冲突（正样本的置信度太低，或负样本置信度太高）的过程。将冲突的正负样本加入模型中就完成了更新。

通过测试 TLD 算法的官方串行版本 OpenTLD^[53]，以及借鉴 H-TLD^[54] 中的

性能分析，同时考虑到 OpenCL 会带来的额外开销，本章将以 OpenTLD 为基础，对其中的计算密集部分和性能瓶颈部分用 OpenCL 进行高性能实现。这些部分包括 Fern 随机森林分类，最邻近分类，学习过程中的重叠率计算、正负样本提取。LK 光流跟踪部分也是计算密集的，但由于 OpenTLD 所依赖的 OpenCV^[55] 库已提供了 OpenCL 版本的实现，因此这里不再重复实现。至于其它部分，有的计算量太小（如确定最终目标位置时，*clusteredBBs* 中边界框数量通常很少），有的不适用于用 OpenCL 并行化（如再训练过程中，计算量小且分支过于密集）。

4.4 Fern 随机森林的高性能实现

在进行 Fern 随机森林分类之前，OpenTLD 先对网格中的所有边界框进行一次方差过滤，以减少输入随机森林的边界框数量。边界框对应的图像块的灰度值方差被计算出来，若该方差低于初始帧中目标物图像块的方差，那么该边界框将被过滤掉。虽然网格中边界框很多，但由于采用了“积分图（Integral Image）”方法，方差过滤十分高效，计算量较小。

如4.3节所述，即使加入了过滤步骤，Fern 随机森林仍将作用于大量边界框，因此必须十分高效。TLD 算法中，用于分类的特征是基于“像素对比”生成的，即根据边界框提取出图像块，然后在图像块中按照预定模式采样 13 对像素点，并进行灰度值对比。如果用 0 和 1 来表示对比结果，则 13 对像素点的对比结果可用一个二进制向量表示，即 Fern 特征向量。上述过程被称作特征提取，如图4.4中的“特征提取”部分所示。

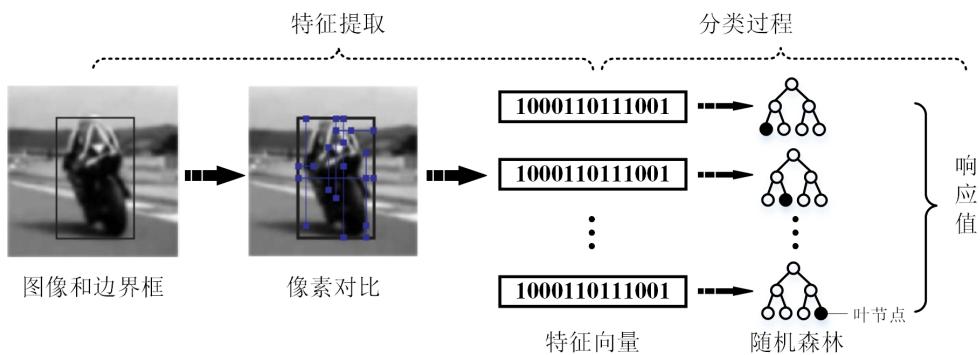


图 4.4 Fern 随机森林分类的流程

由于网格中的边界框有着不同的尺度大小，TLD 为每一个尺度定义了 10 种采样像素点对的模式，每一种采样模式都定义了 13 个点对的不同采样位置。因此，每一个边界框通过特征提取后都会得到 10 个 13 位的 Fern 特征向量。为了对这些特征向量进行分类，TLD 针对每一种采样模式（即每一个 Fern 特征向量）构造一棵随机树，总共 10 棵随机树构成了 Fern 随机森林。随机树的叶节点被赋予

权值，代表着边界框包含目标物体的概率。每个 Fern 特征向量经过对应随机树的分类后，都会落入该随机树的一个叶节点中。将 10 个随机树叶节点的权值进行平均，即得到 Fern 随机森林的响应值。上述过程即是 Fern 随机森林的分类过程，如图4.4中的“分类过程”部分所示。

4.4.1 特征提取的并行化

特征提取过程的输入为当前帧图像、网格、候选边界框索引、以及像素点对的采样模式；输出为所有候选边界框对应的 Fern 特征向量。

在本章的高性能实现中，帧图像的灰度值被存储在一个一维数组中。网格在初始化时就构造好，为一个一维数组，每 5 个相邻元素代表一个边界框，分别是 $\{BBx, BBy, BBw, BBh, scale_id\}$ ，即左上角横、纵坐标、宽、高和尺度索引。因为网格数组在整个跟踪过程中不会变动，因此将它放入 OpenCL 的常量存储中，供多个 Kernel 重复使用，而无需每次都进行分配和释放。候选边界框索引是方差过滤步骤的输出，为一维数组，每个元素 ($BBid$) 都代表着一个候选边界框在网格数组中的位置。在 OpenTLD 中，像素点对的采样模式也是一维数组，每 4 个相邻元素代表一个点对，记为 $\{x_1, y_1, x_2, y_2\}$ ，即两个点在边界框内的坐标。一种采样模式需要 13 个点对，而每一种尺度有 10 种采样模式，考虑所有尺度（最多为 21 种），采样模式数组中共有 $scale_num \times 10 \times 13 \times 4$ 个元素。为了更高效的访存，本章将不同采样点的横、纵坐标连续存储，即将 $\{x_1^1, y_1^1, x_2^1, y_2^1, \dots, x_1^n, y_1^n, x_2^n, y_2^n\}$ 转换为 $\{x_1^1, \dots, x_1^n, y_1^1, \dots, y_1^n, x_2^1, \dots, x_2^n, y_2^1, \dots, y_2^n\}$ 。类似网格数组，采样模式数组也不会变动，因此放入 OpenCL 的常量存储。

特征提取的计算过程为，从候选边界框索引数组中读出边界框索引，根据该索引在网格数组中读出对应边界框的信息（左上角坐标和尺度索引）。根据尺度索引访问采样模式数组，得到所需采样的 10×13 对点在边界框内坐标。将边界框内坐标和边界框左上角坐标叠加，即得到采样点在图像中的坐标。最后根据图像中坐标，进行像素点采样对比，构造 Fern 特征向量。图4.5展示了一次像素点对比，即一个 Fern 向量中的一位的产生过程。

特征提取的 OpenCL 并行实现采用一维的索引空间。一个工作项负责一对采样点的图像中坐标计算、灰度值提取、灰度值对比，即执行一次图4.5所示流程。一个工作组包含 130 个工作项，负责生成一个边界框的所有（10 个）Fern 特征向量。因此一个工作组执行完毕后，每 13 个相邻工作项需要将各自的像素对比结果整合为一个特征向量。最后，工作组的数量应当和候选边界框数量相同。对应的 Kernel 程序核心代码如表4.1所示。第 6~9 行中，由于重构了采样模式数组，局部索引值 (`local_id`) 连续的工作项将访存连续的存储空间，从而实现了高效的合

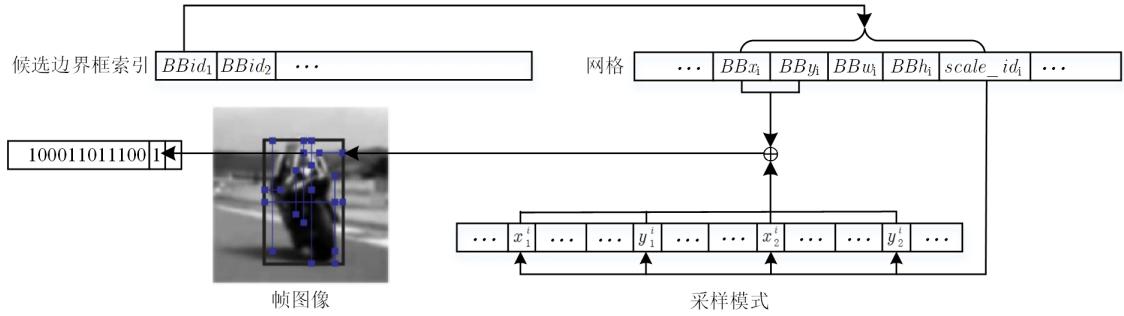


图 4.5 特征提取的流程

表 4.1 特征提取的 Kernel 程序

```

1  __kernel void FernFeatureExtract( __global uchar* img, constant int
2   * grid, __global int* BB_ids, __constant uchar* sample_patterns,
3   __global int* fern_features, ... )
4 {
5   ...
6   BB_id = BB_ids[group_id];
7   scale_id = grid[BB_id*5+4];
8   x1 = sample_patterns[scale_id*130*4*0+local_id] + grid[BB_id
9   *5];
10  y1 = sample_patterns[scale_id*130*4*1+local_id] + grid[BB_id
11  *5+1];
12  x2 = sample_patterns[scale_id*130*4*2+local_id] + grid[BB_id
13  *5];
14  y2 = sample_patterns[scale_id*130*4*3+local_id] + grid[BB_id
15  *5+1];
16  pixel1 = img[y1*img_width+x1];
17  pixel2 = img[y2*img_width+x2];
18
19  local int comp_result[130];
20  comp_result[local_id] = (pixel1>pixel2) << (12-(local_id%13));
21  barrier(CLK_LOCAL_MEM_FENCE);
22
23 }

```

并访存 (Coalesced Memory Access)。第 13 行声明了局部存储数组，用于工作项间像素对比结果的高效共享。最后，由每 13 个工作项的第一个工作项，负责将对比结果整合为 Fern 特征向量，写入全局存储中 (第 18~22 行)。

4.4.2 分类过程的并行化

获得每一个边界框的 10 个 Fern 特征向量后，需要将它们输入随机森林，获得响应值。从表 4.1 的第 20 行可以看出，保存像素对比结果的特征向量实际上是一个 13 位二进制整数，因此高效实现中无需真正建立树结构来逐一判断特征向量的每一元素，而是采用查找表的方式实现。13 位二进制整数一共有 $2^{13} = 8192$ 个

表 4.2 分类过程的 Kernel 程序

```

1 __kernel void FernClassify( __global int* fern_features, __global
2   float* leaf_weights, __global float* fern_responses, ... )
3 {
4   ...
5   tree_id = local_id%10;
6   BB_id = group_id*20 + local_id/20;
7
8   ...
9   local float share_weights[200];
10  fern_feature = fern_features[BB_id*10+tree_id];
11  share_weights[local_id] = leaf_weights[tree_id*8192+
12    fern_feature];
13  barrier(CLK_LOCAL_MEM_FENCE);
14
15  if( ... && local_id % 10 == 0 )
16  {
17    fern_response = share_weights[local_id+0] + share_weights
18      [local_id+1] + ... + share_weights[local_id+9];
19    fern_responses[BB_id] = fern_response;
20  }
21 }
```

可能值，因此可以将随机树实现为 8192 项的表，每一表项存储一个叶节点的权值。由于随机森林有 10 棵树，需分配 10 个这样的表。通过将特征向量看做表索引，对对应表进行查表，即可得到对应树的叶节点权值。对于一个边界框，通过利用它的 10 个特征向量查找对应的 10 个表，再将得到的 10 个权值相加，即可得到响应值。

分类过程的 Kernel 程序核心代码见表4.2。程序的主要输入为 Fern 特征向量数组和叶节点权值数组，输出为每个边界框的响应值。Fern 特征向量数组即是4.4.1节的 Kernel 程序输出，因此该数组可以在全局存储里直接重用。将随机森林所对应的 10 个查找表连接在一起，组成了叶节点权值数组，它按顺序存储着 8192×10 个叶节点权值。由于每次训练会导致叶节点权值更新，因此该数组需要从主存中拷贝。当 OpenCL 程序在 GPU 上运行时，工作组中的工作项数目太少会导致严重性能下降。因此，本实现中的一个工作项负责用一个 Fern 特征向量查表，一个工作组包含 200 个工作项，即负责 20 个边界框的响应值计算。工作项查表的过程就是一个间接访存的过程（表4.2，第 9、10 行）。通过使用局部存储（share_weights，第 8 行），间接访存得到的权值可以被工作组高效共享。因为每 10 个连续工作项计算同一个边界框的 10 个叶节点权值，所以它们中的第一个工作项负责将权值累加，计算出响应值（第 13~17 行）。

4.4.3 与跟踪器重叠执行

LK 光流跟踪和 Fern 随机森林分类是完全独立的，可以重叠执行。表4.3为将光流跟踪和随机森林分类重叠执行的 OpenCL 宿主程序片段。由于在每一帧

表 4.3 LK 光流跟踪和 Fern 随机森林分类的重叠执行

```

1 ...
2 TLDCL.CreateMemBufferFernFeatureExtract ( BB_ids, BB_num,
3   fern_features, BB_num*10 );
4 TLDCL.CreateMemBufferFernClassify ( fern_responses, BB_num );
5 TLDCL.SetKernelArgFernFeatureExtract( );
6 TLDCL.SetKernelArgFernClassify( );
7 TLDCL.WriteBufferFernFeatureExtract( img, img.rows*img.cols );
8 TLDCL.WriteBufferFernClassify( leaf_weights, 8192*10 );
9 TLDCL.EnqueueKernelFernFeatureExtract( );
10 TLDCL.EnqueueKernelFernClassify( );
11 TLDCL.ReadBufferFernFeatureExtract( fern_features, BB_num*10 );
12 TLDCL.ReadBufferFernClassify( fern_responses, BB_num );
13 ...
14 ...
15 TLDCL.SyncCommandQueue( );
16 TLDCL.ReleaseMemObjectFernFeatureExtract( BB_ids, fern_features );
17 TLDCL.ReleaseMemObjectFernClassify( fern_responses );
18 ...

```

中，经过方差过滤后的边界框数目 (BB_num) 是不同的，因此候选边界框索引数组 (BB_ids)、Fern 特征向量数组 (fern_features)、边界框响应值数组 (fern_responses) 的长度也会变化，需要重新创建存储对象，并传输数据 (第 2、3 行)。帧图像 (img) 和叶节点权值数组 (leaf_weights) 的内容会变化，但是长度不变，因此可直接从主存传输 (写入) 到 OpenCL 全局存储 (第 6、7 行)。输入数据准备完毕后，将两个 Kernel 程序 (FernFeatureExtract 和 FernClassify) 的执行命令加入命令队列 (第 8、9 行)，随后再将 Kernel 的执行结果读出到主存 (第 10、11 行)。值得注意的是，无论是数据传输还是 Kernel 执行，上述过程在调用相关 OpenCL API 时均使用非阻塞版本，即是说将命令写入到命令队列后立即返回，而不是等待命令执行完毕。写入命令后，命令队列会自动地调度其中的命令在 OpenCL 计算设备上执行，不再需要干预。得益于这种机制，在传输数据和执行 Kernel 程序时，宿主程序无需等待即可调用 LK 光流跟踪函数进行跟踪 (第 13 行)，从而实现 LK 光流跟踪和 Fern 随机森林分类的重叠执行。

LK 光流跟踪执行完毕后，还需确认命令队列中的命令是否全部执行完毕 (第 15 行)，若未执行完毕，则还需阻塞等待。最后，释放第 2、3 行创建的存储对象 (第 16、17 行)，以便在处理下一帧时重新创建。

4.5 最邻近分类器的高性能实现

在完成 Fern 随机森林分类后，每个候选边界框的响应值都被计算出来。只有响应值高于阈值，且排名前 100 的边界框，才能进入下一步最邻近分类，以计算其包含目标物体的置信度。

4.5.1 算法流程

TLD 的最邻近分类算法较为直接。最邻近分类器维护了两组图像块，即正样本和负样本，它们通常被称作分类器的模型。分类器首先根据待分类边界框在帧图像中提取对应的图像块，然后将该图像块与模型中的图像块进行一一对比，计算它们的相似度。在正样本中，输入图像块必然与某图像块最为相似，记它们的相似度为 $maxP$ 。同理，输入图像块与负样本图像块的最大相似度记为 $maxN$ 。该边界框的置信度即可按下式算出：

$$NNConf = \frac{1 - maxN}{2 - maxP - maxN}. \quad (4.1)$$

显然，一个边界框的置信度，跟对应图像块与正样本的相似度正相关，跟对应图像块与负样本的相似度反相关。

如4.1节中所述，这里的相似度是两个图像块的 NCC (归一化互相关)，其原本定义如下。对于两幅图像，输入图像 \mathbf{f} 和模板图像 \mathbf{t} ，首先将它们的灰度图保存为二维浮点数组，即每个像素点的灰度用一个 $0 \sim 1$ 间的浮点数表示。然后，两幅图像的 NCC 计算公式为：

$$NCC = \frac{\sum_{x,y} \mathbf{f}(x,y)\mathbf{t}(x,y)}{\sqrt{\sum_{x,y} \mathbf{f}(x,y)^2 \sum_{x,y} \mathbf{t}(x,y)^2}} \quad (4.2)$$

其中 (x, y) 代表像素的横纵坐标。TLD 算法对 NCC 进行了细微修改。为了避免光亮度的影响，所有像素的灰度值都会减去所在图像块的平均灰度值。因此，像素点的灰度不再是归一化的，而可能是正数或者负数。为了保证 NCC 的归一化，计算公式修改为：

$$NCC = \frac{1}{2} \left(\frac{\sum_{x,y} \mathbf{f}(x,y)\mathbf{t}(x,y)}{\sqrt{\sum_{x,y} \mathbf{f}(x,y)^2 \sum_{x,y} \mathbf{t}(x,y)^2}} + 1 \right). \quad (4.3)$$

4.5.2 分类过程的并行化

一个边界框的最邻近分类过程分为两个步骤——第一步计算对应图像块与每一个样本图像块的 NCC，第二步找出 $maxP$ 和 $maxN$ 并计算置信度。由于有多个边界框需要分类，第一步中需计算的 NCC 总数目为：边界框数目 \times 样本图像块数目。考虑到每一次 NCC 计算需要遍历两个图像块，第一步的计算量将很大，亟需高性能并行化。而因为边界框数目至多为 100 个，第二步的计算量会较小，且主要为归约操作（提取最大值），因此不适用于用 OpenCL 进行并行化（归约操作需要大量分支和同步）。基于上述考虑，本节仅对第一步，即所有 NCC 的计算，用 OpenCL 并行化实现。第二步仍然放在宿主程序中处理。

OpenTLD 中，最邻近分类器的模型中的图像块大小为 15×15 ，因此根据边界框提取的图像块也会缩放到 15×15 。从公式4.3可以看出，NCC 的计算结果与图像块的像素存储方式和像素访问顺序无关。因此，为了使用 OpenCL 进行并行化，本节把所有边界框对应图像块的灰度值连续地存储在一个一维数组 (patches) 中。这样一来，该数组中每 $15 \times 15 = 225$ 个连续相邻元素就是一个输入图像块。同理，本节也把模型中的正负样本图像块连续地存储在一个一维数组 (p_n_samples) 中，且正样本存储在前，负样本存储在后。这样只需记录正、负样本图像块的数目，就能区分某一元素属于正样本还是负样本。上述两个一维数组即是 NCC 计算的输入。

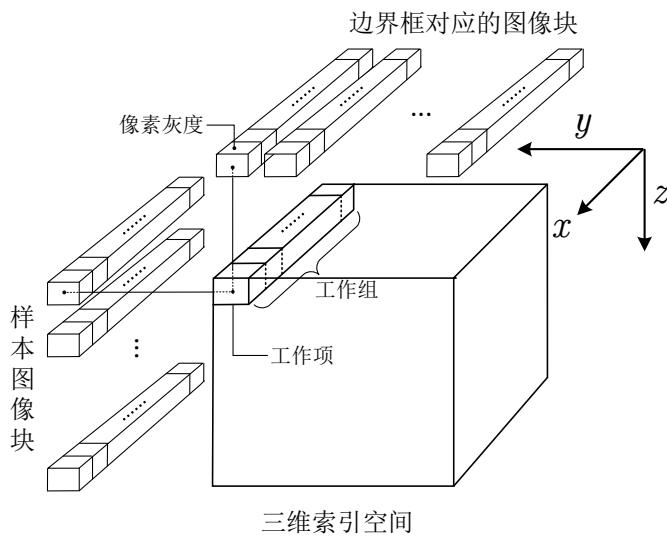


图 4.6 NCC 计算的 OpenCL 索引空间

NCC 计算的 OpenCL 实现使用三维索引空间，如图4.6所示（注：图中的两组图像块是分别连续存储在两个一维数组中的，这里为了显示清晰将各个图像块分开展示）。一个工作组负责计算一个边界框对应图像块和一个样本图像块的 NCC，因此工作组在 $\{x, y, z\}$ 三个维度上的数目设置为 $\{1, \text{边界框数目}, \text{样本图像块数目}\}$ 。每个工作项负责一个像素位置，即计算两个图像块中同一位置处的灰度值平方和乘积，因此一个工作组内的工作项数目为 $\{225, 1, 1\}$ 。故整个三维索引空间的工作项数目是 $\{225, \text{边界框数目}, \text{样本图像块数目}\}$ 。根据公式4.3，将各个像素位置处的计算结果进行累加是不可避免的。为了实现高效的累加操作，本节将采用树形归约。由于树形归约要求元素个数为 2 的次方，因此本节将工作组内工作项数目填充为 $\{256, 1, 1\}$ ，整个索引空间的工作项数目变为 $\{256, \text{边界框数目}, \text{样本图像块数目}\}$ 。

NCC 计算的 Kernel 程序核心代码见表4.4。首先，每个工作项根据各自在索

表 4.4 NCC 计算的 Kernel 程序

```

1  __kernel void NCC( __global float* patches, __global float*
2    p_n_samples, int p_n_sample_num, __global float* nccs )
3  {
4      ...
5      if ( local_X_id < 225 )
6      {
7          patch_pixel = patches[group_Y_id*225+local_X_id];
8          sample_pixel = p_n_samples[group_Z_id*225+local_X_id];
9      }
10     else
11     {
12         patch_pixel = 0;
13         sample_pixel = 0;
14     }
15     __local float patch_squares[256];
16     __local float sample_squares[256];
17     __local float patch_sample_products[256];
18     patch_squares[local_X_id] = patch_pixel * patch_pixel;
19     sample_squares[local_X_id] = sample_pixel * sample_pixel;
20     patch_sample_products[local_X_id] = patch_pixel * sample_pixel;
21     barrier(CLK_LOCAL_MEM_FENCE);
22
23     for( int offset = local_X_size/2; offset > 0; offset >>= 1 )
24     {
25         if ( local_X_id < offset )
26         {
27             patch_squares[local_X_id] = patch_squares[local_X_id+offset]
28             + patch_squares[local_X_id];
29             sample_squares[local_X_id] = sample_squares[local_X_id+
30                 offset] + sample_squares[local_X_id];
31             patch_sample_products[local_X_id] = patch_sample_products[
32                 local_X_id+offset] + patch_sample_products[local_X_id];
33         }
34         barrier(CLK_LOCAL_MEM_FENCE);
35     }
36     if( local_X_id == 0 )
37     {
38         square_sum_product = pow( patch_squares[local_X_id] *
39             sample_squares[local_X_id], -0.5 );
40         nccs[group_Y_id*p_n_sample_num+group_Z_id] = (
41             patch_sample_products[local_X_id]/square_sum_product + 1) *
42             0.5;
43     }
44 }
```

引空间中的位置提取像素灰度值（第 4~13 行）。如果是用于填充的工作项，则灰度值置为 0，以不影响结果正确性。由于位置连续的工作项会访问位置连续的全局存储，这里的访存将是高效的合并访存。第 15~21 行计算出公式4.3 中的 $\mathbf{f}(x, y)\mathbf{t}(x, y)$ 、 $\mathbf{f}(x, y)^2$ 以及 $\mathbf{t}(x, y)^2$ ，并放入局部存储中，以便进行高效累加。23~32 行进行树形归约，每次循环都将两个工作项的计算结果进行叠加，共需 $\log_2 256 = 8$ 次循环。最后，每个工作组的第一个工作项负责利用累加结果，计算出 NCC，并存入全局存储中（第 34~38 行）。在计算出每个边界框对应的图像块和每个样本图像块之间的 NCC 后，所有 NCC 值被传回到宿主程序，由宿主程序

负责第二步的置信度 ($NNConf$) 计算。

4.6 学习过程的高性能实现

如4.3节中所述，当宿主程序整合了检测模块和跟踪模块的结果，确定了当前帧中的目标物体位置 ($currentBB$)，并决定在当前帧进行学习后，将进入 TLD 算法的学习过程。学习过程分为两个步骤。第一步为提取正负样本，即计算出网格中的所有边界框与 $currentBB$ 的重叠率，然后根据重叠率和 Fern 随机森林响应值来选取边界框作为正负样本。第二步为再训练检测模块，即更新 Fern 随机森林的叶节点权值和最邻近分类器的模型。第一步中的网格包含了数量巨大的边界框（例如，对于 320×240 的帧图像将产生 60000 多个边界框），逐个计算重叠率并考察响应值将带来巨大的时间开销。而作为第二步的输入，正、负样本数量通常较少（如训练随机森林的正样本通常为 10 个，负样本为分类出错的边界框，更为稀少），故计算量不大，且多为分支操作。因此，本节仅针对第一步进行基于 OpenCL 的高性能并行化，而第二步仍然由宿主程序负责。

4.6.1 重叠率计算和负样本提取的并行化

这里的重叠率等价于第2、3章中所使用的 IoU (Intersection over Union)，即两个边界框的交集面积除以并集面积。待提取的负样本为重叠率低于阈值，但 Fern 随机森林响应值较高的边界框，即分类出错的边界框。为了提高效率，重叠率的计算和负样本的提取可以在同一个 Kernel 程序中完成，也就是说在计算出重叠率后，若低于阈值，则检查其响应值，判断是否属于负样本。

显然，重叠率计算和负样本提取的输入为网格、当前目标物体位置 ($currentBB$) 和 Fern 随机森林响应值；输出为网格中的所有边界框与 $currentBB$ 的重叠率，以及负样本集合。与特征提取的 Kernel 程序（表4.1和图4.5）不同，本节中每个工作项负责网格中一个边界框的重叠率计算。若网格中的边界框仍然按照 $\{..., BBx_i, BBy_i, BBw_i, BBh_i, scale_id_i, ...\}$ 的形式存储在一维数组中，那么位置连续的工作项在访问同一边界框参数（如 BBx ）时，会访问到间断的存储空间，无法进行合并访存。由于网格是在初始化时构建的，整个跟踪过程中无需更改，因此在构建原网格数组的同时还可以构造一个以不同方式存储边界框的一维数组，其内容为 $\{BBx_1, ..., BBx_n, BBy_1, ..., BBy_n, BBw_1, ..., BBw_n, BBh_1, ..., BBh_n\}$ ，称为重构网格数组。重构网格数组将代替原网格数组作为输入网格，它的构造也只需进行一次，开销可忽略不计。Fern 随机森林的响应值可以重用4.4.2节的结果数组 ($fern_responses$)，但是需要插入被方差过滤掉的边界框响应值（置为 0.0）。由于负样本的数目不确定，且 OpenCL 中不同工作组内的工作项无法同步，

表 4.5 重叠率计算和负样本提取的 Kernel 程序

```

1 __kernel void GetOverlapAndNegativeSample( __constant int* grid_reorg
2   , __global int* currentBB, __global float* fern_responses,
3   __global float* overlaps, __global uchar* negative_tags )
4 {
5   ...
6   int box1x = currentBB[0];
7   int box1y = currentBB[1];
8   int box1w = currentBB[2];
9   int box1h = currentBB[3];
10
11   int box2x = grid_reorg[global_id];
12   int box2y = grid_reorg[global_id+grid_BB_num];
13   int box2w = grid_reorg[global_id+grid_BB_num*2];
14   int box2h = grid_reorg[global_id+grid_BB_num*3];
15   ...
16
17   if ( overlap != 0.0 ) {
18     intersec_x = min(box1x+box1w, box2x+box2w) - max(box1x,
19                 box2x);
20     intersec_y = min(box1y+box1h, box2y+box2h) - max(box1y,
21                 box2y);
22     intersec_area = intersec_x * intersec_y;
23     area1 = box1w*box1h;
24     area2 = box2w*box2h;
25     overlap = intersec_area / (area1+area2-intersec_area);
26   }
27   overlaps[global_id] = overlap;
28
29   if ( global_id < grid_BB_num ) {
30     if ( overlap < low_overlap && fern_responses[global_id]>=
31         high_response )
32       negative_tags[global_id] = 1;
33     else
34       negative_tags[global_id] = 0;
35   }
36 }
```

因此采用标记数组来作为输出的负样本集合。标记数组的长度等于网格中边界框数目，若其中某元素对应的边界框为负样本，则置 1，否则置 0。

重叠率计算和负样本提取的 OpenCL 并行实现较为直接，采用一维索引空间。每一个工作项负责从重构网格数组中提取一个边界框，计算它与 *currentBB* 的重叠率。若重叠率低于阈值，则检查响应值数组中对应的响应值，判断是否是负样本。为了提高 OpenCL 设备的占用率，将 512 个工作项组成一个工作组。因此总的工作项数量需要从网格中边界框数量填充为 512 的倍数。对应的 Kernel 程序核心代码如表 4.5 所示。第 4~12 行提取了用于重叠率计算的边界框，均可以合并访存。第 15~23 行计算出 IoU 并存入全局存储中。最后，25~30 行检查响应值，判断对应边界框是否是负样本。

4.6.2 正样本提取的并行化

OpenTLD 中，学习过程所需的正样本为 10 个具有最高重叠率的边界框，因此提取正样本的过程就是搜索重叠率前 10 名的边界框的过程。虽然该过程是一个

典型的归约过程，计算量小且分支较多，但是由于网格中边界框数量巨大，使用 OpenCL 并行化仍然可以大幅提高对存储带宽的利用，提升整体性能。

为了搜索重叠率前 10 的边界框，本节的 OpenCL Kernel 程序将循环执行 10 次，每次获取一个边界框在网格数组中的索引位置。OpenCL 索引空间设置为一维，其中的工作项总数与网格中的边界框数目相同。每个工作项负责读取一个边界框的重叠率，并与工作组中其它工作项进行对比。1024 个工作项组成一个工作组，工作组内将进行树形归约，得到组内具有最大重叠率的边界框索引，并将该索引写入全局存储中。各个工作组得到的边界框索引将传回宿主程序，由宿主程序再次归约，获得当前具有最大重叠率的边界框。上述过程等价于将重叠率数组分段，每一段包含 1024 个边界框的重叠率。然后在段内和段间进行两步归约，得到最大重叠率。如图4.7所示，Kernel 程序的输入为上一节获得的重叠率数组 (overlaps)，输出为各个工作组 (即各个重叠率数组段) 中具有最大重叠率的边界框索引。宿主程序进行第二次规约后，找出当前具有最大重叠率的边界框并加入正样本中。显然，若不加修改地再次执行 Kernel 程序，仍然会得到同一个边界框。解决该问题的最简单方式就是将重叠率数组中对应的边界框重叠率置为最小值 (0.0)。为了避免在宿主程序中修改重叠率数组，然后再次传输到 OpenCL 计算设备，本节把“当前最大重叠率边界框索引”也作为一个 Kernel 输入参数。Kernel 程序的第一步就是根据该索引，将重叠率数组对应元素置为 0，然后再进行归约。

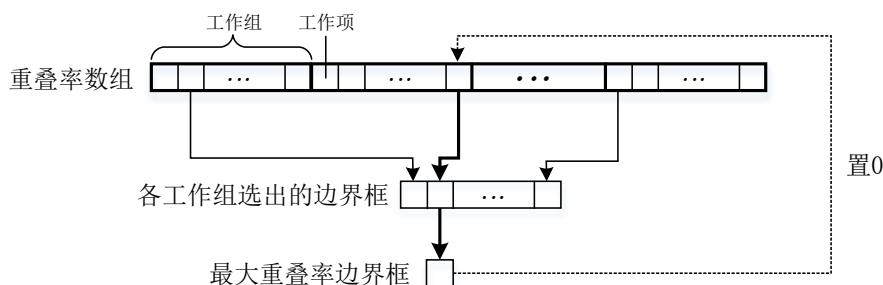


图 4.7 正样本提取流程

正样本提取的 Kernel 程序核心代码如表4.6所示。第 4 行申请一个局部存储数组 `overlaps_local`，用于保存每次重叠率对比后得到的较高重叠率。第 5 行申请的局部存储数组与 `overlaps_local` 对应，用于保存 `overlaps_local` 中每个重叠率对应的边界框索引。第 7~10 行根据输入的“当前最大重叠率边界框索引”，将重叠率数组中的对应元素置为 0.0。值得注意的是，第 10 行的同步语句只作用于各个工作组内的工作项，因此第 8 行对全局存储数组 `overlaps` 的修改可能滞后于某些工作组的归约过程。但是，被置为 0.0 的 `overlaps` 元素只可能被一个工作组用于归约，且负责置 0.0 的工作项正好属于该工作组（由第 7 行保证），

表 4.6 正样本提取的 Kernel 程序

```

1 __kernel void GetPositiveSample( __global float* overlaps, int
2     current_top_BB_id, __global int* top_BB_id_in_group )
3 {
4     ...
5     __local float overlaps_local[1024];
6     __local int BB_ids_local[1024];
7
8     if ( global_id == current_top_BB_id ) {
9         overlaps[current_top_BB_id] = 0.0;
10    }
11    barrier(CLK_LOCAL_MEM_FENCE);
12
13    overlaps_local[local_id] = overlaps[global_id];
14    BB_ids_local[local_id] = global_id;
15    barrier(CLK_LOCAL_MEM_FENCE);
16
17    for ( int offset = local_size/2; offset > 0; offset >>= 1 )
18    {
19        if ( local_id < offset )
20        {
21            if ( overlaps_local[local_id] < overlaps_local[local_id+
22                offset] )
23            {
24                BB_ids_local[local_id] = BB_ids_local[local_id+offset];
25                overlaps_local[local_id] = overlaps_local[local_id+
26                    offset];
27            }
28        }
29        barrier(CLK_LOCAL_MEM_FENCE);
30
31        if ( local_id == 0 ) {
32            top_BB_id_in_group[group_id] = BB_ids_local[0];
33        }
34    }
}

```

故这里只需工作组内同步就可以保证各个工作组的归约结果正确。第 12~14 行将重叠率数组段拷贝到局部存储，并将对应的边界框索引填入 `BB_ids_local`。第 16~27 行是树形归约过程，每一次循环都将并行对比 `overlaps_local` 中的两个元素，并记录较高的重叠率和对应的边界框索引。最后，各个工作组内具有最大重叠率的边界框索引将被写入全局存储中（第 29~31 行）。

全局存储数组 `top_BB_id_in_group` 将被传回宿主程序，由宿主程序进行再次归约，得到当前最大重叠率边界框。该边界框的索引将作为参数之一，再次调用 Kernel 程序，直到获取到 10 个正样本。如 4.6 节开头所述，完成正、负样本的提取后，检测模块的再次训练不适用于用 OpenCL 并行化，将交由宿主程序负责。至此，整个 TLD 算法基于 OpenCL 的高性能实现已介绍完毕。

4.7 实验评测与分析

本章 TLD 算法高性能实现的部分可视化运行结果如图 4.8 所示。图 4.8 中，从上至下，选用的视频序列依次为 *Car*、*Jumping*、*David* 和 *Panda*。序列 *Car* 显示

了 TLD 算法在目标离开视野、遮挡这两类情况下，仍然能够准确跟踪目标。序列 *Jumping* 主要展示了 TLD 对于目标快速移动和运动模糊的适应力。*David* 序列中，光照变化剧烈，TLD 算法展现了很高的鲁棒性。*Panda* 序列主要突出了 TLD 算法对于目标物体非刚性形变的适应能力。

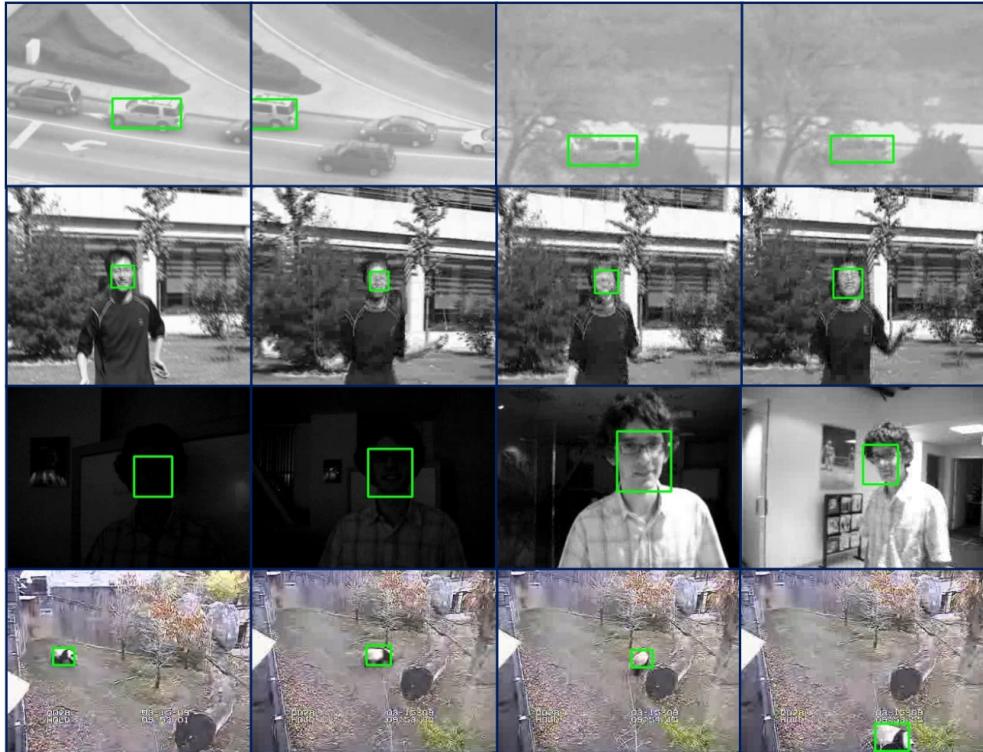


图 4.8 TLD 算法高性能实现的部分运行结果

作为高性能实现的评测，本章的实验将主要针对运行效率。对比基准选用 OpenTLD，输入测试序列选用 *Car*。实验的硬件平台为典型桌面异构计算平台：CPU 为 Intel i7-5500U，采用 Broadwell 架构，双核四线程，2.4GHz，关闭睿频；GPU 为 Nvidia GTX 960M，采用 Maxwell 一代架构，CUDA 计算能力 5.0，包含 640 个流处理器，处理器频率 1096MHz，显存 2GB，带宽 80GB/s；主存为 16GB DDR3-1600 SDRAM，双通道，理论峰值带宽为 25.6GB/s。实验的软件环境采用 Linux 操作系统，发行版本为 Ubuntu 16.04，内核版本为 4.4，编译器为 gcc/g++ 4.8。CPU 的 OpenCL 驱动和运行时采用 Intel OpenCL 1.2 Driver^[56]，所支持的 OpenCL 版本为 2.0；GPU 的 OpenCL 驱动和运行时包含在 CUDA 工具中，CUDA 驱动版本为 8.0，运行时版本为 7.5，所支持的 OpenCL 版本为 1.2。因此，为了保证 OpenCL 实现的可移植性，本章的 OpenCL 代码均按照 OpenCL 1.2 标准编写。整个 TLD 实现还依赖 OpenCV 库，版本为 2.4.13。

4.7.1 Kernel 性能评测与分析

本节首先评测各个 Kernel 的运行效率。使用 OpenCL 进行高性能实现的 Kernel 一共有 5 个，分别是：Fern 特征提取 (FernFeatureExtract)，Fern 随机森林分类过程 (FernClassify)，最邻近分类过程的 NCC 计算 (NCC)，学习过程的重叠率计算和负样本提取 (GetOverlapAndNegativeSample)，以及正样本提取 (GetPositiveSample)。

由于 OpenCL 的移植性，上述 Kernel 可正确运行在多种设备上，且宿主程序可在运行时为各个 Kernel 指定计算设备。通过将 *Car* 作为输入序列，并指定所有 Kernel 在 CPU 上运行，可统计出各个 Kernel 在 CPU 上的时间开销。为了保证时间测量准确，这里利用 OpenCL 命令队列的事件 (Event) 机制来记录 Kernel 的执行时间。该执行时间仅为执行过程所需时间，不包括数据传输、Kernel 启动等额外开销。由于输入的视频序列含有大量帧图像，且 Kernel 一次执行的时间与图像内容有关，本节将 Kernel 在所有帧中的执行时间进行累加，得到各 Kernel 在跟踪过程中的总执行时间，作为其时间开销。整个 TLD 跟踪过程共运行 3 次，各个 Kernel 的最终时间开销为 3 次统计的平均值。同理，使用相同输入序列，且指定所有 Kernel 在 GPU 上运行，可获得各个 Kernel 在 GPU 上的时间开销。作为对比基准，本节还提取了 OpenTLD 中对应于各个 Kernel 的代码段，并同样按照上述方法统计了这些代码段的时间开销。

表 4.7 各 Kernel 的时间开销对比

Kernel 计算设备	FFE ^a	FC ^b	NCC	GONS ^c	GPS ^d
GPU_OCL	502.16	25.08	78.38	15.53	173.01
CPU_OCL	3735.66	281.53	400.39	96.86	990.07
ORG	24925.70	6664.20	7660.12	569.38	17189.30

单位：毫秒

^aFernFeatureExtract

^bFernClassify

^cGetOverlapAndNegativeSample

^dGetPositiveSample

表4.7给出了 5 个 Kernel 在不同设备上的时间开销。其中 GPU_OCL 和 CPU_OCL 分别代表 GPU 和 CPU 上的各 OpenCL Kernel 总执行时间。ORG 代表 OpenTLD 中对应于各 Kernel 的代码段的总执行时间。与表4.7对应的，以 OpenTLD 为基准的加速比如表4.8所示，其中括号内的加速比为 GPU_OCL 相对于 CPU_OCL 的加速比。可以看出，通过本章基于 OpenCL 的并行化实现，无

表 4.8 各 Kernel 的加速比对比

Kernel 计算设备	FFE	FC	NCC	GONS	GPS
GPU_OCL	49.64 (7.44)	265.74 (11.23)	97.73 (5.11)	36.65 (6.24)	99.35 (5.72)
CPU_OCL	6.67	23.67	19.13	5.88	17.36
ORG	1.00	1.00	1.00	1.00	1.00

论在 CPU 还是 GPU 上，各个 Kernel 均取得了令人满意的性能提升。性能提升较低的两个 Kernel 是 GetOverlapAndNegativeSample 和 FernFeatureExtract。在执行 GetOverlapAndNegativeSample 的过程中，绝大部分的边界框由于与 *currentBB* 不相交 (*overlap==0*)，不会进入表4.5的 16~21 行的重叠率计算。因此并行化版本相比原 OpenTLD 版本的优势有所降低。FernFeatureExtract 的计算量很小，仅为像素灰度对比和移位，但访存却较为复杂，需 3 次不规则的间接访存（图4.5，通过边界框索引访问网格数组，通过采样模式和网格数组访问帧图像中两个像素）。因此并行化实现不能充分地利用计算资源和访存带宽。

此外，还可以看出，Kernel 在 GPU 上的执行时间明显短于在 CPU 上，加速比基本在 6 倍左右。GPU 相对于 CPU 加速比最低的 Kernel 为 NCC，原因在于 Kernel 的主要部分为树形归约（表4.4第 23~32 行），而 GPU 体系结构不适于归约操作。归约过程中循环的次数越多，同时参与计算的流处理器越少。直至最后一次循环时，仅有一个工作项负责最后两个元素的累加。GPU 相对于 CPU 加速比最高的 Kernel 为 FernClassify，主要原因在于输入的边界框数量较大，而每个边界框就有 10 个 Fern 特征向量需要分类，导致工作项数量庞大，适于用 GPU 的大量流处理器进行并行处理。除了 CPU 和 GPU 本身的计算能力差别，以及 Kernel 计算负载自身的特点，导致 CPU 和 GPU 上 Kernel 性能差别的另一主要因素就是实现的优化。因为 GPU 体系结构与 OpenCL 平台模型最为相似，本章的 Kernel 在实现时主要考虑了 GPU 平台，而没有针对 CPU 的体系结构特点。这就导致了 OpenCL 的性能移植性问题，下一章将针对该问题进行专门探讨。

使用 OpenCL 进行并行化，在带来计算效率的显著提高的同时，势必会引入额外的时间开销。最为显著的额外开销是数据的重组和传输。例如，在执行 Kernel NCC 之前，需要将所有边界框对应的图像块，和所有正负样本图像块，分别连续地转存在两个一维数组之中。由于这两个一维数组在跟踪过程中长度不断变化，还需要为它们重新创建 OpenCL 存储对象，并传输到计算设备。此外，额外开销还包括命令队列的维护、Kernel 的启动开销等。

为了更加公平地评测本章 TLD 实现的 Kernel 性能，这里考虑所有的额外开销，将 TLD 算法的 3 个部分的执行时间进行对比。这 3 个部分分别是：

- Fern 随机森林分类：包含了 Fern 特征提取和分类过程的两个 Kernel 及前后数据处理、传输；
- 最邻近分类：包含了 NCC 计算 Kernel、置信度计算和数据重组、传输；
- 学习过程中的正负样本提取：包含了重叠率计算和正负样本提取的两个 Kernel，Kernel 循环调用，以及数据传输。

3 个部分中，有的部分包含两个 Kernel。这样划分的原因是，两个 Kernel 间有数据共享或者是生产者—消费者关系，难以确定额外开销属于哪个 Kernel。与之前的对比类似，这里指定 3 个部分中的 Kernel 全部在 GPU 或 CPU 上运行，并统计时间开销。值得注意的是，当 Kernel 在 CPU 上运行时，由于计算设备就是宿主处理器，所以无需真正传输或者拷贝数据，数据传输开销可以忽略。同样地，OpenTLD 中对应于这 3 个部分的代码段也被提取出来，以统计时间开销，作为对比基准。3 部分在不同计算设备上的执行时间对比如图4.9所示。可以看出，考虑所有额外开销后，无论在 CPU 还是 GPU 上，各部分仍然有令人满意的加速。同时，相比单纯的 Kernel 执行时间，加速比出现下降。这说明使用 OpenCL 并行化必然引入的额外开销较多，但是不会抵消并行化带来的巨大性能提升。对于 Fern 随机森林分类，使用 GPU 执行 Kernel 的性能优于使用 CPU。然而对于最邻近分类，GPU 上的性能远低于 CPU。原因主要有两点，一是仅考虑 NCC Kernel 的执行时间，GPU 对于 CPU 有着最低的加速比（表4.8）；二是 GPU 上的 NCC Kernel 需要传输大量数据，即所有边界框对应的图像块和正负样本图像块，此外还需传回所有计算出的 NCC，但 CPU 不需要这些额外的数据传输。由于相同的原因，对于正负样本提取，在 GPU 上执行 Kernel 的性能略低于在 CPU 上执行。

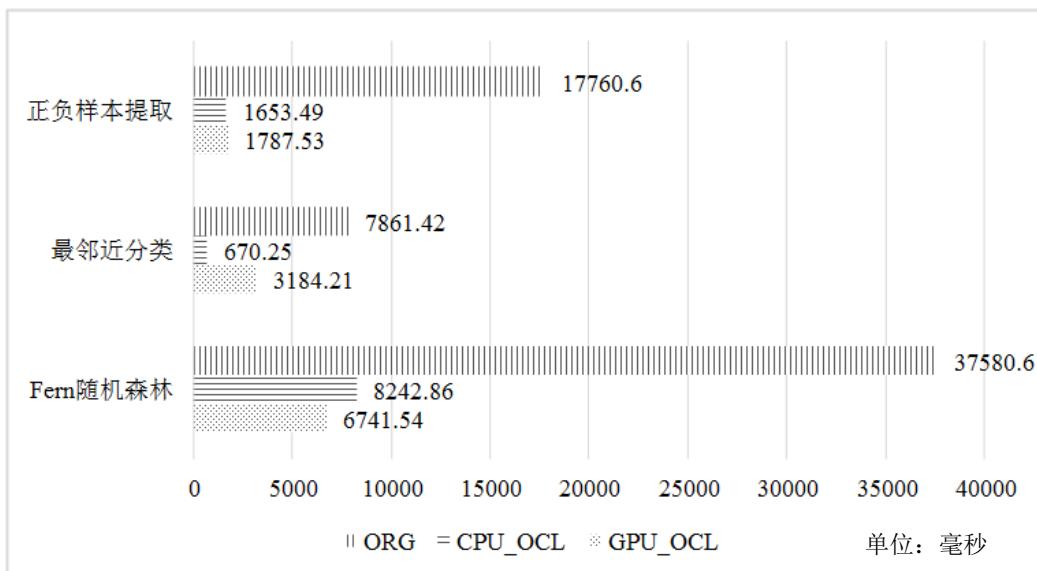


图 4.9 TLD 算法各部分的时间开销对比

4.7.2 整体性能评测与分析

由于本章实现的是一个完整应用，因此本节将评测整个应用的性能。类似上节，本节仍然以 *Car* 作为输入序列，并统计处理完整个序列所需的时间。不同的是，通过上一节对 TLD 算法 3 个部分的性能评测，已经得出了各部分所适用的计算设备；并且作为完整应用，可以如 4.4.3 节所述，将光流跟踪和 Fern 随机森林分类进行重叠执行。因此，这里将评测 4 个版本的整体性能：

- 指定本章实现的所有 Kernel 在 CPU 上执行，光流跟踪在 GPU 上与 Fern 随机森林分类重叠执行（记为 CPU 版本）；
- 指定本章实现的所有 Kernel 在 GPU 上执行，光流跟踪在 CPU 上与 Fern 随机森林分类重叠执行（记为 GPU 版本）；
- 指定 Fern 随机森林分类的两个 Kernel 在 GPU 上执行，最邻近分类和正负样本提取的 3 个 Kernel 在 CPU 上执行，光流跟踪在 CPU 上与 Fern 随机森林分类重叠执行（记为 CPU/GPU 版本）；
- 原始 OpenTLD 版本，作为对比基准（记为 ORG 版本）。

不同的实现版本在整个序列中完成跟踪所需的时间如图 4.10 所示。

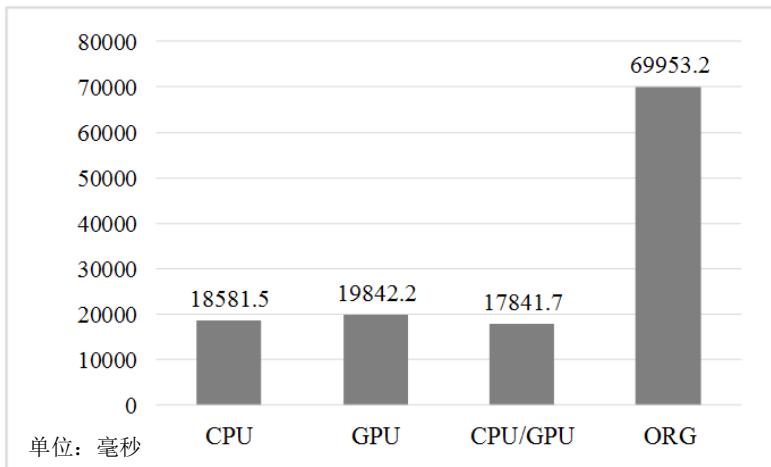


图 4.10 整个应用的时间开销对比

从图 4.10 中可以看出，所有 Kernel 均在 CPU 上执行时的整体性能略高于均在 GPU 上执行时。这与上一节的实验结果一致，即 GPU 在 Fern 随机森林分类上的优势，被最邻近分类和正负样本提取所抵消。CPU/GPU 版本中，各个 Kernel 运行在适合的设备上，取得了最佳的整体性能，因此它将作为本章最终的 TLD 算法高性能实现。相比 OpenTLD，其加速比达到了 3.92 倍，跟踪速度达到了 52.9 FPS，完全满足实时在线跟踪的需求。

4.8 小结

面对跟踪器日益复杂的结构和不断增加的计算负载，高性能的跟踪算法实现变得十分必要。因此，本章在异构计算平台上，基于可移植的并行编程模型 OpenCL，对 TLD 跟踪算法进行了高性能的实现。高性能实现中，本章并行化了 TLD 算法的计算密集部分和瓶颈部分，包括 Fern 随机森林的特征提取和分类过程，最邻近分类的 NCC 计算过程，以及学习过程中的重叠率计算和正负样本提取。此外，还将 Fern 随机森林的处理和 LK 光流跟踪在不同计算设备上进行了重叠执行。通过实验分析，本章发现 OpenCL 并行化所引入的额外开销不可忽视，且存在性能移植性问题，因此高性能实现中，各个 Kernel 程序将运行在各自适合的计算设备上。最终的高性能实现取得了令人满意的整体加速比，完全满足实时处理的需求。但不可否认，本章的高性能实现还远没有达到最佳性能。未来进一步的工作包括：1. 重构代码：通过调整中间数据的存储方式，减少数据重组的时间开销；2. 计算和传输重叠：通过在进行计算的同时传输 Kernel 所需的数据，隐藏数据传输的时间开销；3. 更精细的并行优化：通过针对不同计算设备的体系结构特点，编写更为特定、高效的 Kernel 程序。

第五章 GPU 特定 OpenCL Kernel 程序的性能移植性提升

5.1 引言

上一章已经提到，当前从超级计算机到智能手机，异构计算平台已经广泛应用。而众多异构计算平台中，CPU 和 GPU 仍然是两类最典型的异构计算设备。近来，英特尔推出的“大量集成核”（Many Integrated Core，MIC）协处理器^[57]也是一个日渐受欢迎的选择，例如天河 2 号超级计算机^[58]就曾借助 MIC 排名世界第一。由于共享存储和多指令多数据（MIMD）这两大特点，MIC 仍然属于 CPU 计算设备范畴，可以称作众核 CPU。

异构计算平台的发展，让高性能程序设计面临着更多的挑战。在异构计算平台上编程，最为常见的做法是，为每一个或者每一类计算设备单独编程。譬如说，对于最常见的 CPU-GPU 混合计算平台，主流的编程方式为使用 CUDA 为 GPU 编程，同时使用 OpenMP 为 CPU 编程。这样的“设备特定”的编程方式需要编写大量代码，编程产出率（Productivity）较低，且代码移植性极差，也难以维护。解决这一问题的最理想方案是设计一种统一的编程模型，它使得同一代码可以在所有的异构计算设备上运行，且均能达到良好的性能。

在这样的需求下，OpenCL 应运而生。如上一章介绍的，OpenCL 在设计之初就考虑了跨平台的功能移植性（Functionality Portability），因而采用了抽象的平台模型来代替具体的硬件体系结构。使用 OpenCL 编程的最大便利在于同一代码可以运行在不同的计算设备上，例如上文中的并行 TLD 实现，可以在 CPU 和 GPU 上无障碍地直接编译运行。但是，在实验中已经看出，同样的代码在不同设备上有着不同的执行效率，即 OpenCL 并不能保证性能的移植性（Performance Portability）。这里的性能移植性，可规范地定义为：为某个特定体系结构优化的代码，在其它体系结构的计算设备上可达到的性能。

本章主要讨论一类特定的性能移植性，即 GPU 特定 OpenCL Kernel 程序在 CPU 上的性能移植性。讨论该类移植性的原因有两点。其一，OpenCL 的平台模型很大程度上受经典 GPU 体系结构的启发，因而编程模型被设计成更适于 GPU 体系结构的形式。对于那些明显区别于 GPU 的体系结构，编译器和运行时会以一些代价较高的方式来模拟 GPU 体系结构的某些特征。其二，当前绝大多数 OpenCL 程序是 GPU 特定的，或者说是专门面向 GPU 优化过的。这些程序通常采用大量的线程，适配 GPU 式的轮转指令调度方法，以及尽量利用 GPU 特定的存储层次^[59, 60]。若将这些程序直接运行在 CPU 上，通常无法取得好的性能^[61-63]。例如，GPU 特定的 Kernel 通常会同时创建成千上万个线程，以开发工作组间和工作项间

(工作组内) 的并行性, 但巨大的线程数量对于只有少量重量级核心的 CPU 设备来说是低效的。又例如, GPU 的存储层次和 OpenCL 存储模型有着很好的对应, 但和 CPU 的存储结构有很大不同, 特别是 CPU 并没有可编程的局部存储。

为了提升 GPU 特定 OpenCL Kernel 程序在 CPU 上的性能移植性, 本章将采用代码转换的方法。不同于已有的代码转换方法, 本章的方法不是对原 GPU 特定 Kernel 进行语义上的翻译^[64], 也不是单纯地将原 Kernel 中面向 GPU 的优化部分向 CPU 转换适配^[65, 66], 而是去主动地利用 CPU 特定的体系结构特征, 将原 Kernel 程序转换为适合多核 / 众核 CPU 的形式。本章的方法基于准确的访存分析, 因此生成的代码将去除所有冗余的局部存储数组, 以及它们带来的额外同步开销。此外, 通过从 GPU 特定的 Kernel 中提取并行要素和访存局部性信息, 本章的方法还可以对循环体进行 CPU 特定的优化。本章的目标是一个自动化的源到源 (Source-to-Source) 代码转换工具链, 它以 GPU 特定的 Kernel 程序为输入, 以一个面向 CPU 优化过的函数为输出, 其中的优化手段包括数据局部性优化、向量化、去除局部存储、去除冗余同步等。输出的函数将通过一个调度器在 CPU 上高效并行运行, 而该调度器和本章的代码转换工具将嵌入一个开源的 OpenCL 运行时中, 从而获得一个新的面向 CPU 体系结构的高性能 OpenCL 运行时库。通过用该运行时库替换厂商的原 OpenCL 运行时 (如 Intel 的 [56]), GPU 特定的 Kernel 程序将在 CPU 上获得较好的性能提升。因此, 在异构计算平台上基于 OpenCL 实现高性能跟踪器时, 可避免为 CPU 撰写专门的优化代码, 而仅需面向 GPU 进行实现, 极大提高编程效率并降低代码维护难度。

本章的贡献主要有以下几点:

- 一个全新的工作项折叠方法 (5.4节);

该方法基于一种能够准确体现 Kernel 实际访存模式的线性描述式 (5.3节), 因此能够去除冗余的局部存储数组和同步。通过消除这些局部存储数组所带来的数据拷贝开销和同步开销, Kernel 程序的性能移植性得到很大提升, 且更易于后续的 CPU 特定优化。

- 一个适应 CPU 体系结构的后继优化方法 (5.5节);

该方法作用于工作项折叠后的代码, 通过提取原 GPU 特定 Kernel 中的并行信息和数据局部性信息, 对代码进行循环级 (Loop-Level) 的优化。同时, 该方法还考虑了多核 / 众核 CPU 的体系结构细节, 从而在 CPU 和 MIC 上获得了较好的性能提升。

- 一个 CPU 特定的 OpenCL 运行时 (5.6节)。

该运行时整合了本章的代码转换工具和配套的调度器, 能够先将 GPU 特定的 Kernel 程序进行转换, 然后高效地调度在 CPU 上运行。

此外，[5.2](#)节将介绍与本章相关的已有工作，[5.7](#)节将对本章工作进行实验评测，[5.8](#)节对本章进行总结。

5.2 相关工作

在本章工作之前，已有很多文献研究如何在多核 / 众核 CPU 上提高 OpenCL Kernel 程序的性能，以提升 OpenCL 的性能移植性。这些文献中的方法基本可以分为两类：代码转换方法和自动调优方法。本章的方法属于第一种类别，即直接将 GPU 特定的 Kernel 程序转换为适合在 CPU 上执行的代码。

代码转换类方法中，最为常用的步骤是工作项折叠^[2]（Work-Item Coalescing），又称工作项串行化^[65, 66]（Serialization）。该步骤可以将一个 OpenCL 工作组内的所有工作项合并为一个独立、串行的 CPU 线程，从而得到更粗的线程粒度。经过此步骤，需要并行执行的线程数量大大降低，在 CPU 体系结构上执行 Kernel 程序才变得可行。现有的代码转换方法在进行工作项折叠和提取 SIMD（单指令多数据）并行性时，采用的方案各不相同。Twin Peaks^[64] 所使用的方案简单直接，它利用 `setjmp` 和 `longjmp` 指令来强行把细粒度的工作项合并为单个操作系统线程，然后再在工作项内部进行向量化优化，而忽视了工作项间的数据并行性。分区域串行化（Region Serialization）方法^[65, 66] 通过构造“线程循环（Thread Loop）”来折叠一个工作组内的工作项。对于工作项间的同步，它采用“循环分裂（Loop Fission）”来实现类似功能，即在同步位置处将线程循环分裂为两个循环。此外，它还通过自动向量化技术来将线程循环的多次迭代并行化，开发 SIMD 并行性。Intel 实现的 OpenCL 运行时^[56] 并未开源，仅公开了少量基本方法。它基于分区域串行化，但是直接生成 SIMD 指令，因此在开发工作组内的向量并行性方面非常高效^[67]。值得注意的是，上述方法都没有考虑数据局部性，最终生成的代码总是企图尽量多地执行每一个工作项，以减少工作项间的切换。但这通常会导致间隔、断续的访存模式。解决这一问题的理想方式是将访问同一 Cache 行的工作项段落交叉执行，提高访存效率。为此，Stratton 等人基于 CEAN 表达式^[68]，针对数据的空间局部性进行了改进^[59]。

对于 OpenCL 的局部存储，当前的代码转换方法通常使用一段全局存储（对于 CPU 来说即是主存）来进行模拟，而忽略了 CPU 中 Cache 的存在。至于局部存储带来的同步，Twin Peaks 方法直接使用跳转语句来进行模拟，从而导致极大的开销并且破坏了 Kernel 内的局部性。上述的其它方法则完全依赖循环分裂，从而导致额外的循环控制开销和变量扩展开销。目前，CPU 体系结构下对局部存储和同步的低效处理已经引起了关注。在 [69] 中，工作项折叠的同时会分析局部存储数组是否多余，若多余，则将对局部存储数组的访问转换为对全局存储数组的直

接访问。与此类似，Fang 等人也提出了自动高效地去除局部存储数组的方法^[70]。但是该方法的设计初衷是用于自动调优，因此只是单纯地去除局部存储数组，而没有考虑可能带来的性能影响。此外，他们还构建了一个分析 OpenCL 局部存储对性能影响的工具^[71]，但是该工具并没有被集成到他们的代码转换器^[70]中。

第二类方法，即自动调优^[61, 72]的应用也十分广泛。该类方法通常首先提取对性能有重要影响的参数，然后通过自动调优程序不断尝试各种参数组合，直到达到最佳性能。与代码转换类方法相比，自动调优有很多不可避免的劣势。首先，自动调优需要大量额外编程。除了需要手动编写自动调优程序外，原 OpenCL 代码还必须重构或者重写，以暴露出影响性能的参数，供调优程序修改。其次，自动调优方法通常过于耗时且不够鲁邦。在一种体系结构下极大影响性能的参数，在另一一体系结构下可能无足轻重，但自动调优程序自身无法发现问题，从而导致更大的时间开销。为此，Pennycook 等人提出了一种体系结构无关的方法^[61]，以发掘出在不同体系结构下均能获得可接受的（而不是最优的）性能的参数组合。

以在异构平台上获得最大的性能移植性为目标，Phothilimthana 等人的工作综合了上述两类方法^[73]。他们引入并扩展了 PetaBricks 语言^[74]和相应的编译器，从而能够自动地为多种设备生成 OpenCL 代码。后继的 OpenCL 代码优化过程则主要是基于自动调优的。

5.3 数组访问的线性描述式

准确地识别 OpenCL Kernel 程序对局部存储和全局存储的访存模式是进行高性能代码转换的关键。访存模式通常通过使用编译器前端分析 Kernel 程序，然后构造数组访问描述来获取。但是，绝大部分已有的数组访问描述都是面向嵌套循环的，主要用于自动并行化、私有化等循环级优化。而且，这些描述中通常还引入了一定程度的估计，导致精度不足，无法用于分析 OpenCL 工作项间的依赖性。典型的例子有 Shen 等人的三元组标记^[75]，Paek 等人的线性访存描述子^[76]等。这些描述仅记录了循环体中数组访问的一些基本特征，比如循环变量、循环变量上下界、访存跨度等。由线性等式和线性不等式构成的线性约束系统（Linear Constraint System）也被用于描述数组访问，例如 Trilolet 等人提出的域描述^[77]，Balasundaram 等人的数据访问描述子^[78]，以及新近提出的多面体模型^[79]等。但它们仍然不是面向并行程序的。与本节的线性描述式最为接近的是 Jang 等人的工作^[80]，他们以访存矩阵加偏移向量的形式描述嵌套循环，并用来指导从循环嵌套到数据并行程序的转换。Fang 等人将这一工作应用在了 OpenCL Kernel 上^[71]，构造出了一个自动分析工具。通过将访存矩阵、偏移向量和体系结构参数输入工具，就能分析出局部存储对性能的影响。

本节将提出一个新的数组访问线性描述式。相比上述的描述方法，它更加得精确，但又足够灵活。该描述基于一个重要假设，即绝大多数 GPU 特定 Kernel 程序中，数组访问模式都是线性的。这一假设是通过分析大量 GPU 特定 Kernel 程序得出的，譬如，在 Nvidia GPU Computing SDK^[81] 和 SHOC^[82] 测试集中，只有包含了间接访存（一个数组的访问索引是另一数组的元素）的 Kernel 才可能违反该假设。换句话说，即使本节的线性描述式局限于线性或者直接的存储访问，它仍然可以覆盖绝大多数的 GPU 特定 Kernel 程序。

本节的数组访问线性描述式由下标函数和访问约束共同组成。对于 OpenCL Kernel 程序中的每一个数组访问，都使用一个线性下标函数和一组线性访问约束来进行描述。下标函数的函数值是对应数组的访问索引，表示的是该次访存可能访问到的数组元素位置；每一条访问约束都是一个线性等式或者不等式，根据循环边界和条件语句产生，用以约束数组访问索引的范围。下标函数和访问约束中的变量包括：工作组的组索引值（Group ID）、工作项的局部索引值（Local ID）、包围该次访存的各层循环的循环变量、以及 Kernel 程序的输入参数。如果这些变量的某一赋值满足了所有访问约束，则 Kernel 一定会访问对应数组的下标函数值处的元素。

表 5.1 原始的 GPU 特定矩阵乘法 Kernel 程序

```

__kernel void matrixMul( __global float* C, __global float* A,
    __global float* B, __local float* AS, __local float* BS,
    int uiWA, int uiWB )
{
    1   int aBegin = uiWA * BLOCK_SIZE * Gid.y;
    2   int aEnd   = aBegin + uiWA - 1;
    3   int aStep  = BLOCK_SIZE;
    4   int bBegin = BLOCK_SIZE * Gid.x;
    5   int bStep  = BLOCK_SIZE * uiWB;

    6   float Csub = 0.0f;
    7   for (int a = aBegin, b = bBegin; a <= aEnd;
        a += aStep, b += bStep)
    {
        8      AS[Lid.x + Lid.y * BLOCK_SIZE] = A[a + uiWA * Lid.y + Lid.x];
        9      BS[Lid.x + Lid.y * BLOCK_SIZE] = B[b + uiWB * Lid.y + Lid.x];
        10     barrier(CLK_LOCAL_MEM_FENCE);
        11     for (int k = 0; k < BLOCK_SIZE; ++k)
        12         Csub += AS[k + Lid.y*BLOCK_SIZE] * BS[Lid.x + k*BLOCK_SIZE];
        13     barrier(CLK_LOCAL_MEM_FENCE);
    }
    14    C[(Gid.y*GROUP_SIZE_Y+Lid.y)*GLOBAL_SIZE_X
        + (Gid.x*GROUP_SIZE_X+Lid.x)] = Csub;
}

```

下面以实例来说明本节所采用的数组访问描述式。表5.1为来自 Nvidia GPU Computing SDK 的矩阵乘法 OpenCL Kernel 程序，它面向 GPU 实现了 $C = A \times B$ 。为了清晰易懂，原 Kernel 中的一些变量名被修改，此外，`Lid` 代表工作项的局部

索引值, Gid 代表工作组的组索引值。在 Kernel 的外层循环中(第 7~13 行), 一共有 6 处数组访问: 第 8 行的写 \mathbf{AS} 和读 \mathbf{A} , 第 9 行的写 \mathbf{BS} 和读 \mathbf{B} , 第 12 行的读 \mathbf{AS} 和读 \mathbf{BS} 。作为示例, 对于 \mathbf{AS} 和 \mathbf{A} 的 3 处数组访问的线性描述式如图 5.1 所示。其中, f 代表线性下标函数, $Constraint$ 代表线性访问约束的集合, $Iter_x$ ($x = a, b, k$) 代表循环正规化(Loop Normalization)后的循环变量。对于数组访问 $A[a + uiWA * Lid.y + Lid.x]$, 得到的线性下标函数是 f_A^{read} , 其值覆盖了 Kernel 执行时对数组 \mathbf{A} 的所有可能的访问位置。线性约束集合 $Constraints_A^{read}$ 通过约束 f_A^{read} 中变量的取值, 限制了上述访问位置的范围。对于数组 \mathbf{BS} 和 \mathbf{B} 的访问描述式与图 5.1 非常类似, 因此不再单独列出。

$$\left\{ \begin{array}{l} f_A^{read} = (uiWA \times BLOCK_SIZE \times Gid.y + BLOCK_SIZE \times Iter_a) + uiWA \times Lid.y + Lid.x \\ Constraint_A^{read} = \{Iter_a \geq 0; Iter_a < uiWA/BLOCK_SIZE; Gid.y \geq 0; \\ \quad Gid.y < GLOBAL_SIZE; Lid.x \geq 0; Lid.x < BLOCK_SIZE; Lid.y \geq 0; \\ \quad Lid.y < BLOCK_SIZE\} \end{array} \right.$$

$$\left\{ \begin{array}{l} f_{AS}^{read} = Lid.x + Lid.y \times BLOCK_SIZE \\ Constraint_{AS}^{read} = \{Lid.x \geq 0; Lid.x < BLOCK_SIZE; Lid.y \geq 0; Lid.y < BLOCK_SIZE\} \end{array} \right.$$

$$\left\{ \begin{array}{l} f_{AS}^{read} = Iter_k + Lid.y \times BLOCK_SIZE \\ Constraint_{AS}^{read} = \{Iter_k \geq 0; Iter_k < BLOCK_SIZE; Lid.y \geq 0; Lid.y < BLOCK_SIZE\} \end{array} \right.$$

图 5.1 矩阵乘法 Kernel 中读 \mathbf{A} 、写 \mathbf{AS} 和读 \mathbf{AS} 的数组访问线性描述式

有了本节中提出的数组访问线性描述式的帮助, 对 GPU 特定 OpenCL Kernel 程序的代码转换将分为两步进行: 基于分析的工作项折叠和适应体系结构的后继优化。这两步的目的都是提升 Kernel 程序在多核 / 众核 CPU 上的性能。

5.4 基于分析的工作项折叠

工作项折叠(或者串行化)的目的是加粗线程粒度, 通常会将一个工作组内的所有工作项合并为一个独立的 CPU 线程。标准的折叠方法是构建线程循环嵌套。该循环嵌套的层数和 Kernel 的索引空间维度一致, 各层的循环变量与工作项的局部索引对应, 最内层的循环体就是原 Kernel 程序, 如图 5.2(a)、(b) 所示。

但是构建这样的线程循环并非易事, 面临的最大挑战就是 Kernel 程序中同步语句(如 `barrier()`)的存在。当前最先进的工作项折叠方法均采用循环分裂来解决同步问题。图 5.2(c)、(d) 展示了构建线程循环并使用循环分裂的例子。图 5.2(d) 中, 由于 `barrier()` 语句的存在, 线程循环在同步语句处被分裂为两个线程循环。循环分裂所带来的额外开销主要有两部分, 一是额外的循环控制语句, 二是可能导致的变量扩展(Variable Expansion)。变量扩展指的是, 当分裂出的两个循环的循环体中有共用变量时, 由于执行路径的改变, 需要保存第一个循

环每次迭代后的共用变量值。否则，第二个循环得到的将只有前一循环最后一次迭代得到的共用变量值。如果工作组内的工作项数目较多，变量扩展将导致大量的额外访存开销（变量将无法放入寄存器中）。为此，本节将基于数组访问描述式进行准确的工作项间依赖性分析，将不必要的同步语句预先消除，从而从根本上避免循环分裂。具体细节将在5.4.2节中介绍。

<pre>Kernel_Name(Kernel_Args...) { Kernel_Body... }</pre>	<pre>Kernel_Name(Kernel_Args...) { for(Lid.z=0; Lid.z<GROUP_SIZE_Z; Lid.z++) for(Lid.y=0; Lid.y<GROUP_SIZE_Y; Lid.y++) for(Lid.x=0; Lid.x<GROUP_SIZE_X; Lid.x++) { Kernel_Body... } }</pre>
(a) 原Kernel程序	(b) 工作项折叠后的线程循环
<pre>Kernel_Name(Kernel_Args...) { Kernel_Body_1... barrier(); Kernel_Body_2... }</pre>	<pre>Kernel_Name(Kernel_Args...) { for(Lid.z=0; Lid.z<GROUP_SIZE_Z; Lid.z++) for(Lid.y=0; Lid.y<GROUP_SIZE_Y; Lid.y++) for(Lid.x=0; Lid.x<GROUP_SIZE_X; Lid.x++) { Kernel_Body_1... } for(Lid.z=0; Lid.z<GROUP_SIZE_Z; Lid.z++) for(Lid.y=0; Lid.y<GROUP_SIZE_Y; Lid.y++) for(Lid.x=0; Lid.x<GROUP_SIZE_X; Lid.x++) { Kernel_Body_2... } }</pre>
(c) 原Kernel程序（含同步）	(d) 工作项折叠和循环分裂后的线程循环

图 5.2 通过构建线程循环来进行工作项折叠

工作项折叠过程中，另一极大影响性能的要素就是 OpenCL 局部存储的使用。使用在局部存储中分配的数组是 GPU 特定 Kernel 程序非常常用的性能优化方法。因为 OpenCL 局部存储直接对应着 GPU 的高速片上存储，访问局部存储数组会十分高效。但是，CPU 体系结构中并没有能够与局部存储对应的硬件支持，因此现有的 OpenCL 运行时只能用一段主存空间来模拟局部存储的功能。在这种情况下，如果不加区分地使用局部存储数组，很可能由于额外的数据拷贝和附带的同步，导致性能反而下降。目前已有的工作项折叠方法均没有对局部存储数组进行有效的处理，而本节的一大创新就是在折叠过程中去除冗余的局部存储数组，从而提升 Kernel 程序性能。对局部存储数组是否冗余的相关分析，以及对应的去除过程，都是基于数组访问描述式的，具体将在5.4.1节中进行介绍。

5.4.1 去除冗余的局部存储数组

局部存储数组在 GPU 特定的 OpenCL Kernel 程序中的作用可以分为以下 4 种类型：

-
- 1) **数据缓存**: 为了提升 Kernel 程序的时间和空间局部性, 将经常访问或者将被重复访问的数据缓存在局部存储中, 从而使延迟较长的全局存储访问变为更快速的局部存储访问。
 - 2) **数据重组**: 通过合并访存的方式从全局存储中读出数据, 然后以另一种形式存入局部存储中, 以避免后续对局部存储的访问发生组冲突 (Bank Conflict)。一个典型的例子就是矩阵转置相乘 ($C = A \times A^T$) Kernel^[83] 中, A 按行从全局存储中读出, 之后按列存入局部存储数组。
 - 3) **数据交换**: 将工作项执行的中间结果存入局部存储中, 然后由其它工作项读出。这种作用类型不仅能够在工作项间交换数据, 还能避免多个工作项进行重复的计算。
 - 4) **防止寄存器溢出**: 如果 Kernel 内申请的私有变量过多, 导致执行所有工作项所需的寄存器不足, 那么这些私有变量就会溢出到低速的片外存储 (显存) 内, 导致性能大幅下降。这种作用类型将局部存储看作是私有存储的扩展, 用于存储更多的工作项私有数据。

在 CPU 体系结构下, 第 3) 种作用类型的 OpenCL 局部存储数组也是必需的, 因此在工作项折叠过程中不可去除。具有第 4) 种作用的局部存储数组也不可去除, 因为该局部存储数组是用于存放中间结果的, 一旦去除将无处存放溢出的数据。对于第 2) 种作用类型的局部存储数组, 尽管 CPU 只是用一段主存空间进行模拟, 并且会导致一次额外的数据拷贝, 本节也仍然将其保留。原因在于其中的数据进行过重组, 后续的访存将变得高效, 从而很可能提升最终性能。至于第 1) 种作用类型, 局部存储数组就完全多余了。因为 CPU 体系结构中的多级 Cache 具有完全相同的功能, 所以使用局部存储数组是毫无意义的, 还会导致额外的数据拷贝。要准确去除第 1) 种作用类型的数组, 需要首先通过分析判别局部存储数组的作用类型, 然后将对原数组的访问转换为对全局存储数组的访问。

如果一个局部存储数组不具备第 3) 和第 4) 种作用, 但是具备第 1) 或者第 2) 种 (又或两种皆有之), 那么 Kernel 程序对它的一系列访问必然满足下列过程^[84]:

- 从全局存储数组中读出数据, 再存入该局部存储数组中。
- 可能在工作组内进行一次同步, 以确保之后每个工作项都可以安全地读取到其它工作项写入的数据。
- 从该局部存储数组中读出数据, 并使用数据进行计算。
- 如果访存过程是循环的, 则还需要一次同步, 以避免有的工作项还未利用该局部存储数组中的数据进行计算。

例如表5.1中, AS 和 BS 这两个局部存储数组就是仅用于缓存数据的。对这两个数组的访问完全符合上述过程。如表5.1的第 8、9 行所示, 数组 AS 和 BS 中的

数据直接来自于全局存储数组 \mathbf{A} 和 \mathbf{B} 。而第 10、13 行即是所需的两次同步。

只有当同时满足下列两个条件时，局部存储数组的读访问才能被转换为直接的全局存储数组读访问：

- (1) 存在一个与该局部存储数组读访问相对应的局部存储数组写访问：如果将写访问的描述式中的某些变量替换为读访问中的变量，二者的数组访问描述式（包括下标函数和访问约束）可变得完全相同。
- (2) (1) 中的局部存储数组写访问的数据是来自于全局存储数组的。该条件可以通过检查“定义—使用链（Define-Use Chain）”^[85]来验证，且通常情况下，局部存储数组写访问和对应的全局存储读访问是相邻的。

如果一个局部存储数组具有作用类型 3) 或者 4)，那么对它的读访问将无法满足上述两个条件，因为数组中的元素均来自于中间计算结果，而不是来自全局存储。

将局部存储数组的读访问转换为全局存储数组读访问的过程，就是找出所有满足上述两个条件的局部存储数组读操作，然后将它们用条件 (2) 中的全局存储数组读操作替换。随后，与条件 (1) 中的局部存储数组写访问相关的语句将变为“死代码（Dead Code）”，被编译器自动地去除。一个典型的例子就是图5.1中的那对局部存储数组读写：

$$\left\{ \begin{array}{l} f_{AS}^{write} = Lid.x + Lid.y \times BLOCK_SIZE \\ Constraint_{AS}^{write} = \\ \{Lid.x \geq 0; Lid.x < BLOCK_SIZE; Lid.y \geq 0; \\ Lid.y < BLOCK_SIZE\} \end{array} \right., \quad (5.1)$$

$$\left\{ \begin{array}{l} f_{AS}^{read} = Iter_k + Lid.y \times BLOCK_SIZE \\ Constraint_{AS}^{read} = \\ \{Iter_k \geq 0; Iter_k < BLOCK_SIZE; Lid.y \geq 0; \\ Lid.y < BLOCK_SIZE\} \end{array} \right.. \quad (5.2)$$

如果将公式 (5.1) 中的 $Lid.x$ 用公式 (5.2) 中的 $Iter_k$ 替换，那么上面两个数组访问描述式将变得一样，即满足了条件 (1)。此外，根据表5.1第 8 行，公式 (5.1) 的数据来源是全局存储数组 \mathbf{A} ：

$$\begin{aligned} f_{\mathbf{A}}^{read} = & (uiWA \times BLOCK_SIZE \times Gid.y + \\ & BLOCK_SIZE \times Iter_a) + uiWA \times Lid.y + Lid.x, \end{aligned} \quad (5.3)$$

因此条件 (2) 也满足了。综上，将公式 (5.2) 对应的局部存储数组读访问转换为直接的全局存储数组读访问是可行的。通过将公式 (5.3) 中的 $Lid.x$ 替换为 $Iter_k$ ，

然后用变量替换后对应的读操作代替公式 (5.2) 对应的读操作，即可完成转换：

$$\begin{aligned}
 f_{\mathbf{AS}}^{read} &= Iter_k + Lid.y \times BLOCK_SIZE \\
 \Rightarrow \\
 f_{\mathbf{A}}^{read} &= (uiWA \times BLOCK_SIZE \times Gid.y + \\
 &\quad BLOCK_SIZE \times Iter_a) + uiWA \times Lid.y + Iter_k
 \end{aligned} \tag{5.4}$$

上述的转换条件和方法只考虑了可行性，即可用于所有没有数据交换功能和防止寄存器溢出功能的局部存储数组。但是，对于具有数据重组功能的局部存储数组，上述方法虽然可行但不会带来性能提升。因此这里再附加一个启发式的条件，以确保局部存储数组不具有数据重组的作用：

- (3) 考察条件 (1) 中的局部存储数组写访问，以及 (2) 中的全局存储数组读访问，它们的下标函数中变量 $Lid.x$ 必须具有相同的系数（或者都不含变量 $Lid.x$ ，即系数为 0）。

例如，在公式 (5.1) 和 (5.3) 中，下标函数 $f_{\mathbf{AS}}^{write}$ 和 $f_{\mathbf{A}}^{read}$ 的 $Lid.x$ 变量系数均为 1，并且公式 (5.1) 和公式 (5.2) 覆盖了 Kernel 对局部存储数组 \mathbf{AS} 的所有访问。因此基于条件 (3)，可以确定局部存储数组 \mathbf{AS} 不具有数据重组的作用，去除它将在 CPU 体系结构下带来性能提升。

通过去除所有仅具有数据缓存作用的局部存储数组，并且将对于这些数组的访问转换为对全局存储数组的直接访问，工作项折叠后的 Kernel 性能将获得明显提升。表 5.2（代码的行号与表 5.1 一致）展示了去除冗余的局部存储数组后的矩阵乘法代码。其中，局部存储数组 \mathbf{AS} 和 \mathbf{BS} 由于仅作为数据缓存使用，都被去除了。

表 5.2 去除冗余的局部存储数组后的矩阵乘法 Kernel 代码片段

```

8   //Dead Code
9   //Dead Code
10  barrier(CLK_LOCAL_MEM_FENCE);
11  for(int Iter_k=0; Iter_k<BLOCK_SIZE; ++Iter_k)
12    Csub += A[(uiWA*BLOCK_SIZE*Gid.y+BLOCK_SIZE*Iter_a)
13      + uiWA*Lid.y + Iter_k]
     * B[(BLOCK_SIZE*Gid.x+BLOCK_SIZE*uiWB*Iter_b)
     + uiWB*Iter_k + Lid.x];
13  barrier(CLK_LOCAL_MEM_FENCE);

```

5.4.2 依赖性分析和同步语句消除

如果一个工作组中的不同工作项会访问相同的存储位置，那么同步通常是必要的，否则 Kernel 程序的执行结果将是不确定的。在 Kernel 程序中，作为同步点的 `barrier` 语句经常会离局部存储数组的访问非常近，这是因为局部存储是由

工作组内的所有工作项所共享的存储空间。但是，`barrier` 语句的存在并不代表工作项间确实存在依赖性（写后读相关、读后写相关、或写后写相关），而由于依赖所产生的同步才是真正不可消除的。例如，5.4.1节所述的4种局部存储数组作用类型中，只有数据交换才会导致真正的数据依赖，即一个工作项在获取到另一工作项的中间结果前，无法继续执行。具有其它3种作用类型的局部存储数组也可能引入同步语句，但是工作项间并没有真正的依赖性。如果各个工作项直接从全局存储中获取所需的数据，同步语句则不再必要。也即是说，如果 `barrier` 语句是由仅具有数据缓存或数据重组功能的局部存储数组引入的，那么它是可以随着局部存储数组一起去除掉的。

在 CPU 体系结构下，同步语句的作用可能存在两面性。工作项折叠时，如果在同步语句位置进行循环分裂，则一定会导致额外的开销，如循环控制和变量扩展。但是循环分裂的同时也改变了工作组的执行流程，从而也可能带来更好的时间和空间局部性。对于这个两面性问题，本节的解决方案是：在工作项折叠过程中忽略同步语句可能带来的好处，尽可能地去除同步语句。然后在线程循环的后继优化中针对具体的体系结构细节，重新开发数据局部性，具体将在5.5.2节中介绍。

在去除了冗余的局部存储数组之后，就可以开始进行同步语句的消除。但是，单纯地删除所有的 `barrier` 语句是不安全的，因为同步语句可能还服务于其它未被去除的局部存储数组，也可能是由于全局存储数组而导致的同步。为了检查一个同步语句是否可以被安全地删除，需要进行依赖性分析。这里的依赖性分析和经典的依赖性分析有很大的不同，因为它是在不同的工作项间进行的。而在工作项内代码是串行执行的，因此无需考虑工作项内的依赖性，该依赖性也与同步无关。本节依赖性分析的核心思想是：如果两个工作项访问了同一局部存储数组或全局存储数组（其中至少有一个访问是写访问），并且访问的数组区域有重叠，那么这两个工作项间就存在依赖性。

依赖性分析是针对每一条 `barrier` 语句逐次进行的。首先将同步语句也看作是基本块（Basic Block）边界，将 Kernel 代码划分为多个基本块。虽然引入了同步，但是这里基本块的概念和经典编译理论中的概念是一致的，即基本块是一个包含了尽量多的语句的代码段，其中任意一条语句的执行，意味着块内的其它语句也一定会执行。然后需要考察每一对满足下列条件的数组访问：两个数组访问分别来自位于同步语句前、后的基本块；两个数组访问针对同一个局部存储数组或全局存储数组；其中一个数组访问是写访问。如图5.3的上半部分所示，虚线矩形框代表的是不同控制语句导致的基本块划分，箭头指向的是需要考察其中数组访问的两个基本块。图5.3的下半部分强调了对数组访问的考察是针对不同工

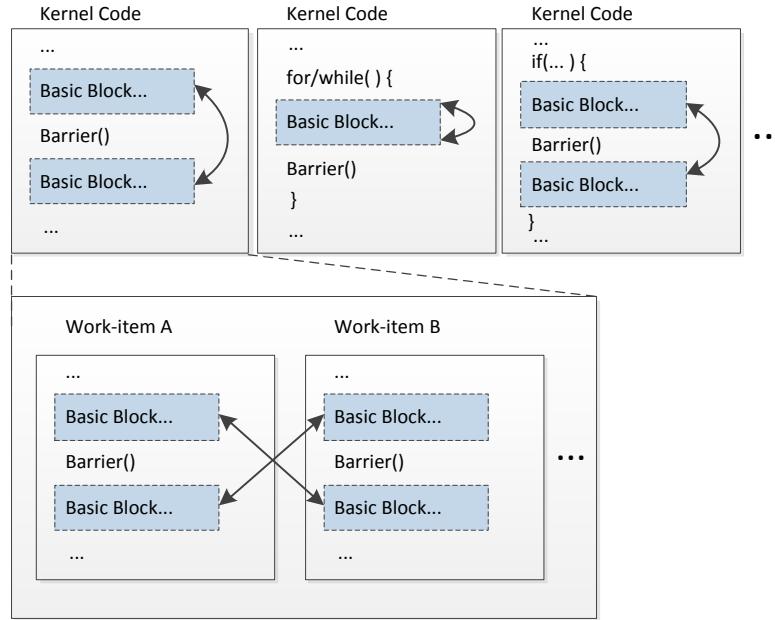


图 5.3 依赖性分析示意图

作项的。考察一对数组访问时，需要将二者对应的数组访问描述式进行联立，构成一个丢番图不等式系统（Diophantine Inequation System）。如果该不等式系统有解，且解中不要求所有的工作项局部索引值相等，那么真正的依赖存在，对应的 `barrier` 语句不可删除。

公式 (5.5) 展示了上述不等式系统的构建过程：

$$\begin{cases} f_1 = \overrightarrow{\mathbf{Coe}_1} \cdot \overrightarrow{\mathbf{Var}_1^T} + Const & \overrightarrow{\mathbf{Var}_1} = (\dots, Lid.z, Lid.y, Lid.x) \\ Constraint_1 \end{cases}$$

$$\begin{cases} f_2 = \overrightarrow{\mathbf{Coe}_2} \cdot \overrightarrow{\mathbf{Var}_2^T} + Const & \overrightarrow{\mathbf{Var}_2} = (\dots, Lid.z', Lid.y', Lid.x') . \\ Constraint_2 \end{cases} \quad (5.5)$$

$$\Rightarrow \begin{cases} f_1 = f_2 \\ Constraint_1 \\ Constraint_2 \end{cases}$$

其中 $\overrightarrow{\mathbf{Coe}}$ 代表变量系数组成的向量， $\overrightarrow{\mathbf{Var}}$ 代表变量组成的向量， $Const$ 代表一个常数。需要注意的是，由于依赖性分析针对的是两个不同的工作项， $\overrightarrow{\mathbf{Var}_1}$ 和 $\overrightarrow{\mathbf{Var}_2}$ 中的工作项局部索引值不再被看作是相同的变量，因此这里进行了换名。如果箭头右侧的不等式系统有解，且解中不要求 $\{Lid.x = Lid.x'; Lid.y = Lid.y'; Lid.z = Lid.z'\}$ ，那么对应的 `barrier` 语句就必须保留。

通过上述的依赖性分析，所有可消除的同步语句都将被删除，继而可使用经典的工作项折叠方法，生成线程循环。对于不可消除的同步语句，仍然采用循环分裂解决。表5.3展示了工作项折叠后的矩阵乘法 Kernel，原 Kernel 中的两个 barrier 语句均被消除了。从表5.3可以看出，同步将不带来任何直接开销。

表 5.3 工作项折叠后的矩阵乘法 Kernel 代码片段

```

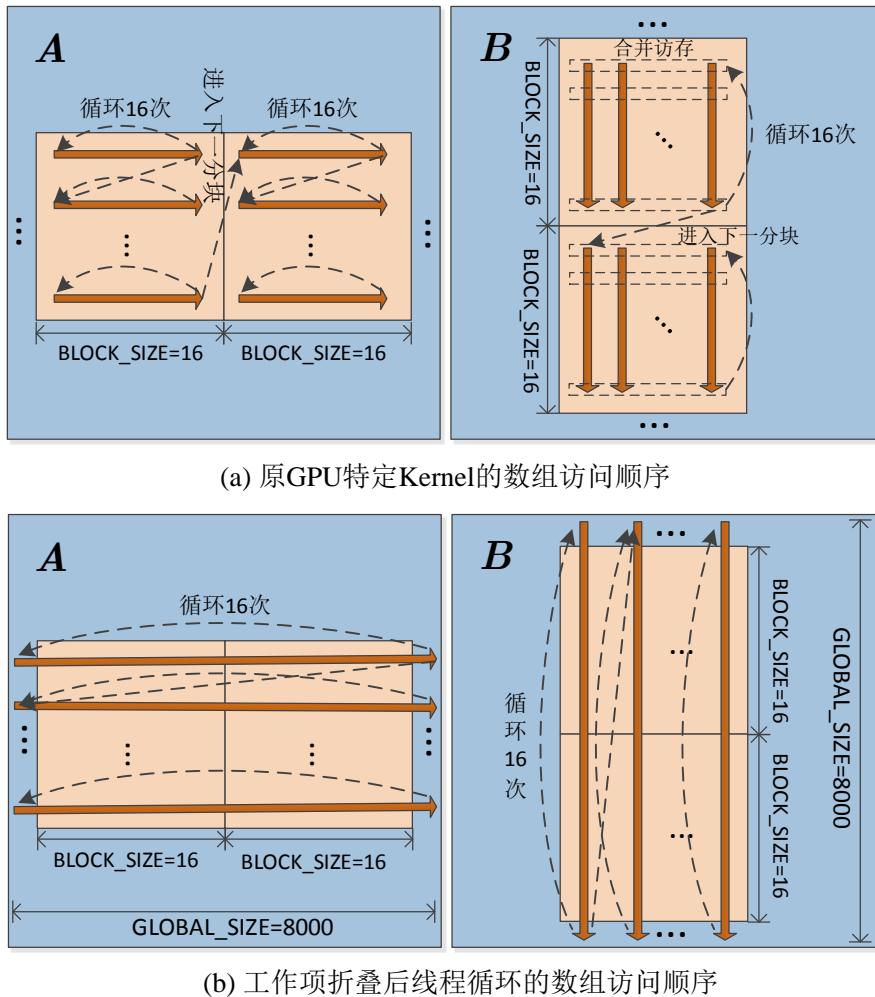
for (int Lid.y=0 ; Lid.y<BLOCK_SIZE; Lid.y++) {
    for (int Lid.x=0 ; Lid.x<BLOCK_SIZE; Lid.x++) {
        ...
        6     float Csub = 0.0f;
        7     for (int Itera=0, Iterb=0; Itera<=uiWA/BLOCK_SIZE;
              Itera++, Iterb++) {
        8         //Dead Code
        9         //Dead Code
        10        //barrier(CLK_LOCAL_MEM_FENCE);
        11        for(int Iterk=0; Iterk<BLOCK_SIZE; ++Iterk)
        12            Csub += A[(uiWA*BLOCK_SIZE*Gid.y+BLOCK_SIZE*Itera)
                      +uiWA*Lid.y+Iterk]
                      * B[(BLOCK_SIZE*Gid.x+BLOCK_SIZE*uiWB*Iterb)
                      +uiWB*Iterk+Lid.x];
        13        //barrier(CLK_LOCAL_MEM_FENCE);
        14    C[(Gid.y*GROUP_SIZE_Y+Lid.y)*GLOBAL_SIZE_X+
          (Gid.x*GROUP_SIZE_X+Lid.x)] = Csub;
    }
}

```

消除同步语句之后，虽然其直接开销也被消除了，但是整个工作组的代码执行顺序也发生了改变。图5.4(a)显示了原 GPU 特定 Kernel 在执行过程中，一个工作组对于全局数组 **A** 和 **B** 的访问顺序。其中粗箭头代表对数组元素的依次访问，黑色虚线箭头代表控制流。由于采用分块矩阵乘法，**A** 的每一行分段会被连续访问 16 次，**B** 的每一分块的各个行将以合并访存的方式被循环访问 16 次。图5.4(b)显示了工作项折叠后，线程循环的访问顺序。矩阵乘法不再是分块的，对数组 **A** 的访问将遍历整个行，对数组 **B** 的访问将遍历整个列。因此，Cache 失效将不可避免，而且 SIMD 并行性无法得到开发。这样的工作项折叠后代码虽然不够高效，但是十分“整齐”，有利于后继优化。本章后继优化的目标不是还原图5.4(a)所示的 GPU 特定的局部性，而是根据目标 CPU 的体系结构细节，重新开发数据局部性和并行性。

5.5 适应体系结构的后继优化

完成工作项的折叠后，线程粒度大大变粗，工作组能够以线程循环的形式作为一个 CPU 线程进行执行。但是此时仍然有两个 CPU 特定的性能要素没有得到利用。一个是工作项间的并行性没有得到开发，导致 CPU 中的 SIMD 单元利用率低下。另一个是线程循环的循环体过长，导致数据局部性不佳。为此，本节将对折叠后的线程循环进行两步优化：向量化和局部性重开发。这两步优化的目的在

图 5.4 矩阵乘法 $C = A \times B$ 中对于数组 A 和 B 的不同访问顺序

于，借鉴原 GPU 特定 Kernel 对于性能的考虑，针对 CPU 特定的体系结构细节提升代码性能。

5.5.1 向量化

对于工作项折叠后的线程循环，*icc* 等高性能编译器能够进行自动向量化，以利用当前多核 / 众核 CPU 中广泛存在的 SIMD 单元。但是，编译器通常只针对循环嵌套的最内层循环进行向量化^[86]，而最内层循环可能无法向量化，或者不是进行向量化的最佳位置。因此本节将不依赖编译器的自动向量化功能，而是显式地进行循环级的向量化优化。这一过程不仅需要提取原 GPU 特定 Kernel 中的并行信息，同时还会考虑 CPU 的体系结构细节。

对于 GPU 来说，访问全局存储数组的最佳模式是顺序、逐个、对齐的访问^[83]。例如，第 k 个工作项访问对齐的全局存储段的第 k 个字。这样的访存模式必然是合并访存，可以最大化地利用 GPU 的显存带宽。GPU 特定 Kernel 程序对于局部

存储数组的访问通常也是顺序、逐个的（无需对齐），目的是避免访存时发生组冲突。此外，GPU 还有一种高效的局部存储访问模式，即令同一 Warp/ 半 Warp 内的工作项访问同一局部存储位置。

由于 CPU 体系结构下，OpenCL 的全局存储和局部存储均对应于主存，因此最适于进行向量化的循环层次就是以 $Lid.x$ 为循环变量的循环。原因在于，该层循环的相邻迭代对应于相邻的工作项。而相邻工作项顺序、逐个的访存模式正好可以转换为向量加载指令（Vector-Load）和向量存储指令（Vector-Store）；相邻工作项访问同一存储位置的访存模式也可以转化为置向量指令（Vector-Set）或向量广播指令（Broadcast）。除了访存操作以外，各种数学操作都可以通过经典的循环向量化方法无障碍地进行向量化，因为 $Lid.x$ 循环在工作项折叠之后，没有任何循环间依赖。

对于工作项折叠后的代码的向量化步骤如下：

- 1) 如果原 GPU 特定 Kernel 程序中包含循环，则进行循环分布（Loop Distribution），使得非线程循环（即循环变量不是工作项索引的循环）成为循环嵌套的最内层循环。该步骤可能需要变量扩展，并会导致更多的循环控制语句。但是相比向量化后的巨大性能提升，这些额外开销是可忽略的。
- 2) 对于每一个循环嵌套，进行循环交换（Loop Interchange），使得 $Lid.x$ 循环成为循环嵌套的最内层循环。这一步骤一定是可进行的，因为线程循环的循环层间没有任何依赖。
- 3) 根据目标 CPU 体系结构的 SIMD 单元宽度，对 $Lid.x$ 循环进行循环分段（Loop Blocking），使得得到的循环嵌套的外层循环变量（记为 $vLid.x$ ）以（SIMD 单元宽度 / 数组元素宽度）为步长。之后即可按经典方法对最内层循环进行向量化。假设用于计算的是单精度浮点数（32 位），对于 Sandy Bridge 架构的 CPU，由于 SIMD 宽度为 256 位，循环步长设为 8。而对于 Knights Corner 架构的 MIC 协处理器，由于 SIMD 宽度为 512 位，循环步长设为 16。此外 Knights Corner 架构还包括了向量乘加单元（FMA），因此其编译器可以自动地将乘法和加法融合。

工作项折叠后的矩阵乘法 Kernel 代码，再经过向量化，变为如表5.4所示。表中假设 CPU 的 SIMD 宽度为 256 位，所示的 $vLid.x$ 是经过了循环正规化的。前缀“vec”代表向量数据类型或者向量操作。可以看出， $Lid.x$ 循环被分段后得到的 3 个循环嵌套中，最内层的循环都已经因为完全向量化而消失了。

表 5.4 向量化后的矩阵乘法 Kernel 代码片段

```

for(int vLid.x=0; vLid.x<BLOCK_SIZE/8; vLid.x++)
    for(int Lid.y=0; Lid.y<BLOCK_SIZE; Lid.y++)
        Csub[Lid.y][vLid.x] = vec_float8(0.0f);

for(int vLid.x=0; vLid.x<BLOCK_SIZE/8; vLid.x++)
    for(int Lid.y=0; Lid.y<BLOCK_SIZE; Lid.y++)
        for(int Itera=0, Iterb=0; Itera<=uiWA/BLOCK_SIZE;
            Itera++, Iterb++)
            for(int Iterk=0; Iterk<BLOCK_SIZE; ++Iterk)
                Csub[Lid.y][vLid.x] =
                    vec_float8_add( Csub[Lid.y][vLid.x],
                    vec_float8_mult(
                        vec_float8_broadcast(A[(uiWA*BLOCK_SIZE
                            *Gid.y+BLOCK_SIZE*Itera)
                            +uiWA*Lid.y+Iterk]), //broadcast
                        vec_float8_load(B+BLOCK_SIZE*Gid.x
                            +BLOCK_SIZE*uiWB*Iterb+
                            uiWB*Iterk+vLid.x*8) //load
                    )
                ); //mult //add

for(int vLid.x=0; vLid.x<BLOCK_SIZE/8; vLid.x++)
    for(int Lid.y=0; Lid.y<BLOCK_SIZE; Lid.y++)
        vec_float8_store(C+(Gid.y*GROUP_SIZE_Y+Lid.y)
            *GLOBAL_SIZE_X+Gid.x*GROUP_SIZE_X
            +vLid.x*8, Csub[Lid.y][vLid.x]);

```

5.5.2 局部性重开发

经过工作项折叠和向量化，转换得到的 Kernel 程序的并行性已经非常适合 CPU 体系结构。但是，数据局部性仍然没有得到改善。大部分的 GPU 特定 Kernel 程序在实现时都考虑了数据局部性，例如将长循环分段等。但是有的 Kernel 程序并没有进行相关优化。例如 [84] 中给出的“基本矩阵乘法 Kernel”，它和上文中讨论的矩阵乘法 Kernel 具有相同的功能，但是没有进行任何局部性优化，因此被 Nvidia 作为一个基准程序来展示局部存储的作用。本节的局部性重开发方法会利用到原 GPU 特定 Kernel 对于局部性的优化，但同时也会尽量适应目标 CPU 体系结构对数据局部性的需求，因此即使是数据局部性很差的 GPU 特定 Kernel，仍然可以通过本节的方法在 CPU 上获得极大性能提升。本节使用（并微调）了经典的循环级优化方法，包括循环交换、循环分段和提升寄存器重用率。具体的数据局部性重开发方法分为以下 3 个步骤。

1) 将非线程循环分段。主要思想是：如果一个循环的循环变量出现在了一个多维数组的最低维（该维度的相邻元素在内存中连续）索引中，且不出现在其它维度的索引中，那么对该循环进行分段通常是有好处的^[87]。对于所有的非线程循环，如果同时满足下面两个条件，则对其进行分段：

- a) 循环体中某个数组访问对应的线性描述式包含了非线程循环的循环变量，且该循环变量的系数足够小；（这里以 2 作为保守的阈值，因为太大的系数会

导致访存模式的步长较大，从而限制循环分段的有效性。)

- b) 对于 a) 中的数组访问，将一次循环迭代的元素访问范围转换为字节数，得到访存覆盖范围足够大。(这里使用一个经验阈值，即 L1 Cache 大小的 1/8，因为如果每次迭代的访存范围很窄，则无需进行循环分段。对于 Sandy Bridge 和 Knights Corner 架构，该阈值均为 4KB。)

如果需要进行循环分段，非线程循环将被转换为两个嵌套的循环，并且内层循环的迭代次数 (Trip Count) 将被设置为工作组最低维的工作项数目，即 $Lid.x$ 的取值范围大小。经典的循环分段包括两个步骤，分段开采和循环交换。但是这里的循环分段只进行分段开采，而不考虑循环交换。循环交换将在下一步骤中，针对所有层次的循环 (包括线程循环) 同时进行。如表5.4的中间部分所示， $Iter_k$ 循环和 $Iter_a/Iter_b$ 循环实际上已经是非线程循环被分段后得到的内、外层循环了，因此没有非线程循环满足上述分段条件。但是，“基本矩阵乘法 Kernel”中的 k 循环会被检测出来并进行分段处理。

2) 循环交换。 这里将沿用 Allen 等人提出的启发式循环交换算法^[87]。对于嵌套的多个循环 $\{L_1, L_2, \dots, L_n\}$ ，首先建立一个启发式函数，用于估计循环嵌套中的数组访问 (包括向量加载和存储) 可能导致的 Cache 失效次数。然后对于每一层循环 L_i ，假设其处于最内层，用启发式函数估计出循环嵌套的 Cache 失效次数 (称作该循环的最内层存储开销)，记为 $C_M(L_i)$ 。最后，各层循环按照 C_M 值的降序，从内到外进行重新排列。

但是，上述循环交换算法在计算 C_M 值时基于一个重要假设，即当前循环的迭代次数是无限大的。而线程循环和分段后的内层非线程循环的迭代次数通常较少，因此上述算法还需要修改：当计算一个循环的最内层存储开销 (即 C_M 值) 时，如果它的迭代次数少于工作组最低维的工作项数目，则不再把它的失效次数乘以它外层循环的迭代次数。上述算法有的时候还会产生多个循环交换方案，例如表5.4中间部分的嵌套循环中， $Iter_a/Iter_b$ 循环应当被置于最外层， $Iter_k$ 循环应当被置于最内层，而 $vLid.x$ 循环和 $Lid.y$ 循环的位置是可以互相交换的。

3) 选择利于向量寄存器重用的循环交换。 单个 CPU 核里的向量寄存器是非常稀有的计算资源。Sandy Bridge 架构中有 16 个，Knights Corner 架构中有 32 个。因此向量寄存器的重用率将对 Kernel 性能产生很大影响。提升寄存器的重用率实际上就是提升 Kernel 的数据时间局部性，故这里仍然利用上一步中计算 C_M 值的启发式函数，并针对时间局部性进行修改：每一个向量操作都被看作是“访存操作”，Cache 大小设置为向量寄存器的数量，Cache 行的长度设置为 1。这样一来，修改后的启发式函数估计出的“Cache 失效次数”实际上是向量寄存器的溢出次数。将该启发式函数应用于上一步得到的所有循环交换方案， C_M 值最小的方案就是

向量寄存器重用率最高的循环交换。再次观察表5.4中间部分的嵌套循环，如果一个CPU核包含16个256位的向量寄存器，那么将 $vLid.x$ 循环置于 $Lid.y$ 循环之外将获得更佳的时间局部性。

经过局部性重开发后的矩阵乘法Kernel代码片段如表5.5所示（仅显示了Kernel中间的循环嵌套），它同时也是将GPU特定Kernel面向Sandy Bridge架构的CPU转换所得到的最终结果。对于数组**A**和**B**的访问顺序也变为图5.5所示。可以看出，原GPU特定Kernel中的合并访存转换为了向量操作，数据局部性也得到了大幅提升。

表5.5 面向Sandy Bridge架构转换后的最终矩阵乘法Kernel代码片段

```

for(int Itera=0, Iterb=0; Itera<uiWA/BLOCK_SIZE;
    Itera++, Iterb++)
    for(int vLid.x=0; vLid.x<BLOCK_SIZE/8; vLid.x++)
        for(int Lid.y=0; Lid.y<BLOCK_SIZE; Lid.y++)
            for(int Iterk=0; Iterk<BLOCK_SIZE; ++Iterk)
                Csub[Lid.y][vLid.x] =
                    vec_float8_add(Csub[Lid.y][vLid.x],
                    vec_float8_mult(
                        vec_float8_broadcast(A[(uiWA*BLOCK_SIZE
                            *Gid.y+BLOCK_SIZE*Itera)
                            +uiWA*Lid.y+Iterk]), //broadcast
                        vec_float8_load(B+BLOCK_SIZE*Gid.x
                            +BLOCK_SIZE*uiWB*Iterb+
                            uiWB*Iterk+vLid.x*8) //load
                    )
                );

```

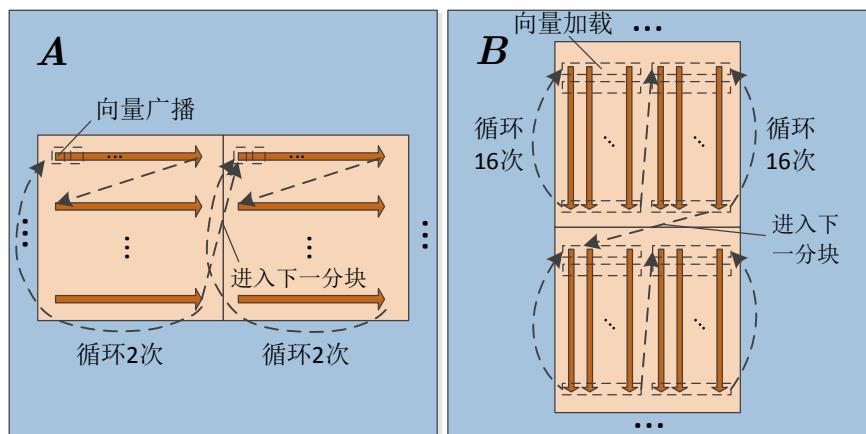


图5.5 转换后的矩阵乘法Kernel对于数组**A**和**B**的访问顺序

表5.6显示了将GPU特定Kernel面向Knights Corner架构的MIC转换所得到的最终结果。其中 $vLid.x$ 循环和 $Lid.y$ 循环的位置可以互换，不影响性能。表5.6和表5.5有区别的原因在于不同的SIMD宽度、是否有FMA单元、以及不同的向量

寄存器数量。

表 5.6 面向 Knights Corner 架构转换后的最终矩阵乘法 Kernel 代码片段

```

for(int vLid.x=0; vLid.x<BLOCK_SIZE/16; vLid.x++)
    for(int Lid.y=0; Lid.y<BLOCK_SIZE; Lid.y++)
        Csub[Lid.y][vLid.x] = vec_float16(0.0f);

for(int Iter_a=0, Iter_b=0; Iter_a<=uiWA/BLOCK_SIZE;
    Iter_a++, Iter_b++)
    for(int vLid.x=0; vLid.x<BLOCK_SIZE/16; vLid.x++)
        //vLid.x loop and Lid.y loop are exchangeable
        for(int Lid.y=0; Lid.y<BLOCK_SIZE; Lid.y++)
            for(int Iter_k=0; Iter_k<BLOCK_SIZE; ++Iter_k)
                Csub[Lid.y][vLid.x] =
                    vec_float16_FMA(
                        vec_float16_broadcast(A[(uiWA*BLOCK_SIZE
                            *Gid.y+BLOCK_SIZE*Iter_a)
                            +uiWA*Lid.y+Iter_k]), //broadcast
                        vec_float16_load(B+BLOCK_SIZE*Gid.x
                            +BLOCK_SIZE*uiWB*Iter_b+uiWB*Iter_k+
                            vLid.x*16]), //load
                    Csub[Lid.y][vLid.x]); //FMA

for(int vLid.x=0; vLid.x<BLOCK_SIZE/16; vLid.x++)
    for(int Lid.y=0; Lid.y<BLOCK_SIZE; Lid.y++)
        vec_float16_store(C+(Gid.y*GROUP_SIZE_Y+Lid.y)
            *GLOBAL_SIZE_X+Gid.x*GROUP_SIZE_X
            +vLid.x*16, Csub[Lid.y][vLid.x]);

```

5.6 运行时调度

上述的整个代码转换过程，包括提取数组访问描述式，消除冗余的局部存储数组和同步语句，构建线程循环，以及后继优化，被实现为一个源到源的代码转换工具链。该工具链基于 Clang^[88] 编译器前端，以及 LLVM^[89] 中间代码转换库。通过工具链的转换，再利用 LLVM 的 CBE 后端，GPU 特定的 OpenCL Kernel 程序被转换为一个可链接的函数，其输入参数为原 Kernel 的全部参数和工作组的组索引值。每一次调用该函数，等价于执行了一个对应的工作组，也就是说，进行运行时调度的粒度为单个工作组。

转换得到的 Kernel 代码不能直接被标准的 OpenCL 运行时调度运行。因此需要一个配套的运行时调度器来调用转换后的函数。该调度器的功能应类似于标准 OpenCL 运行时对工作组的调度。本节实现调度器的主要思想是，将工作组尽可能连续地和均匀地分布到各个 CPU 核上执行，即是说各个 CPU 核分到的工作组数量差别不超过 1。通过调度器，工作组间的并行性被转换为了线程级的并行性。

调度器执行 Kernel 的过程如下：

- 1) 主线程将所有工作组分成多个等量的集合，各个集合被连续地指派到目标多核 / 众核 CPU 的各个逻辑核上。(由于超线程 (Hyper Thread) 技术，Sandy Bridge 架构下一个物理核等价于两个逻辑核，而 Knights Corner 架构下一个物理核等于 4 个逻辑核，但这里忽略其第一个物理核，以用于执行主线程。)
- 2) 主线程为每一个逻辑核创建一个 POSIX 线程，称作工作线程。并通过设置线程的亲和属性 (Affinity) 来指定工作线程在对应逻辑核上运行。
- 3) 各个工作线程根据被指派的工作组，不断地以工作组组索引值为参数调用转换后的函数。
- 4) 主线程等待工作线程的汇聚 (Join)。

利用工具链转换 GPU 特定 Kernel 的过程，以及转换后的代码通过调度器运行的过程都展示在了图5.6中。

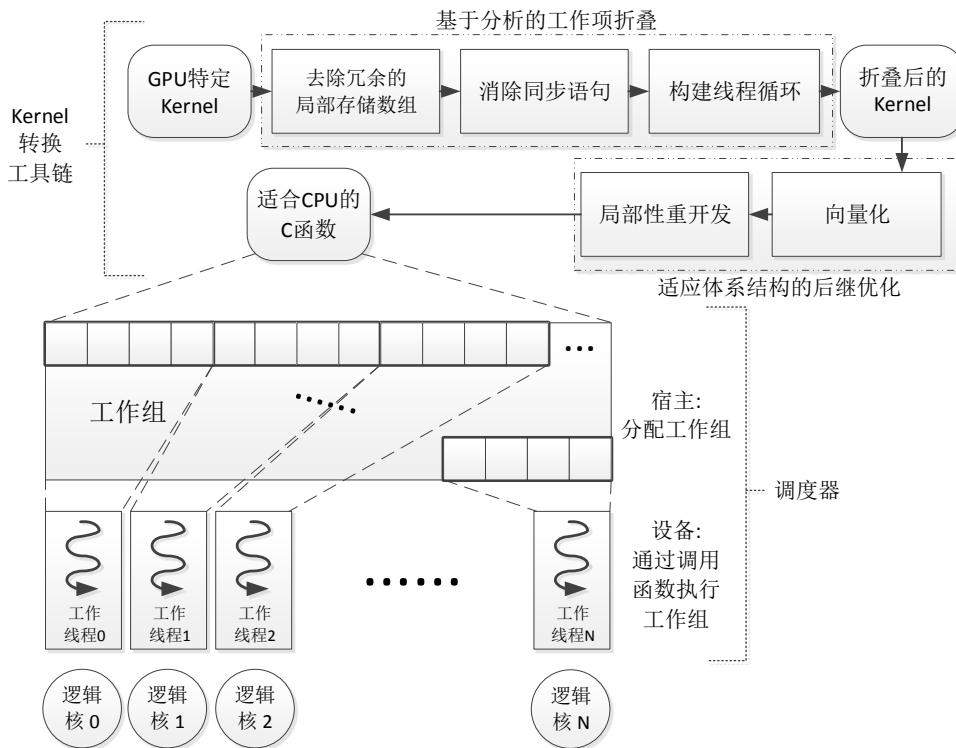


图 5.6 GPU 特定 Kernel 的转换和调度执行

为了能够运行包括宿主程序和 Kernel 程序的完整 OpenCL 程序，本章的代码转换工具链和运行时调度器被集成到一个开源的 OpenCL 实现——FreeOCL^[90] 中。原 FreeOCL 中的代码生成模块和 Kernel 调度模块被本章的代码转换工具链和调度器分别替换。其它用于实现 OpenCL 宿主程序 API 的模块保持不变。对于 CPU 和 MIC，均使用 Intel C++ Compiler v13.0.0 编译转换后的 Kernel 代码。在面向 Sandy Bridge 架构的 CPU 进行编译时，主要的编译选项为 -O3 -xHost。而

对于 Knights Corner 架构的 MIC，将在编译前需要为转换后的 Kernel 代码加上“offload”指导语句，使得整个 Kernel 程序可以在 MIC 上执行。值得注意的是，由于本节在 MIC 上执行 Kernel 时使用的是卸载（Offload）模式，MIC 的最后一个物理核将被保留，不可用于计算。至此，包含 GPU 特定 Kernel 的 OpenCL 程序，将能够借助本章的新 OpenCL 运行时，在多核 / 众核 CPU 上高效地运行。

5.7 性能评测

本节将在两个硬件平台上进行实验和性能评测：(1) 两个 Intel Xeon E5-2650 8 核 CPU，共有 16 个物理核，作为经典的多核 CPU 平台；(2) 一个 Intel Xeon Phi 5110p MIC 协处理器，共有 60 个物理核，作为新兴的众核 CPU 平台。Xeon E5 CPU 同时也作为宿主处理器，其上的操作系统为 Red Hat Enterprise Linux 6.2，内核版本 2.6.32-220。本章提出的加入了 Kernel 转换工具链和运行时调度器的新 OpenCL 运行时（记作 OurOCL），将与 Intel 官方提供的 OpenCL 运行时（记作 IntelOCL）进行对比。Intel 的官方 OpenCL 运行时来自 Intel SDK for OpenCL Applications 2013^[91]。

一共 6 个 GPU 特定的 OpenCL Kernel 程序组成了本节的测试集。如表 5.7 所示，前 5 个 Kernel 程序均针对 GPU 体系结构进行了高性能优化，其中 Stencil2D 来自 SHOC^[82]，其它 4 个来自 Nvidia GPU Computing SDK。第 6 个 Kernel 程序 NaiveMatrixMul 来自 [84]，即 5.5.2 节中介绍过的“基本矩阵乘法 Kernel”。它的数据局部性没有得到任何优化，将作为本节评测的基准。

表 5.7 用于性能评测的 6 个 Kernel 程序

Kernel 名	数据规模	工作组大小	功能介绍
oclMatrixMul	8000×8000	16×16	分块矩阵乘法
oclFDTD3d	$320 \times 320 \times 320$, Radius=16, Timestep=5	32×32	有限差分时域演化，三维模板计算
Stencil2D	4096×4096 , 1000 次迭代	16×16	标准的二维 9 点模板计算
oclDCT8x8	10240×10240	32×2	8×8 的离散余弦变换 (DCT)
oclNbody	327680	256	327680 个个体的引力模拟
*NaiveMatrixMul	8000×8000	16×16	未使用分块和局部存储的矩阵乘法

在 Intel 的产品平台上，IntelOCL 通常是最为高效的 OpenCL 运行时。一个 Kernel 依靠 IntelOCL 获得的性能通常也是最优的，即代表了“官方提供”的性能移植性。因此通过对比同一 Kernel 在 OurOCL 和 IntelOCL 下的性能，即可评价两个运行时所提供的性能移植性。上述的 GPU 特定 Kernel 程序在 OurOCL 下运行时，会先经过 Kernel 转换工具链的转换，然后被调度器调度运行。同样的 Kernel 将在

IntelOCL 下再次运行，并对比两次运行所需的时间。当运行 Kernel 程序时，本节仅记录 Kernel 执行的时间，而忽略其它开销。通过 Kernel 的执行时间计算出的相对性能如表 5.8 所示，表中的所有相对性能都是相对于 CPU+IntelOCL 的性能比值。此外，Kernel 的绝对执行时间也显示在了括号里。可以看出，在多核 CPU 平台上，GPU 特定 Kernel 程序在 OurOCL 下的平均性能达到了 IntelOCL 下的 3.24 倍（不包括 NaiveMatrixMul）。而在众核 MIC 平台上，该平均性能比也达到了 2.00 倍。

表 5.8 与 Intel OpenCL 运行时和 OpenMP 的性能对比

Kernel	相对性能 (执行时间)		
	CPU+IntelOCL	CPU+OurOCL	CPU+OMP
oclMatrixMul	1 (23.30 s)	3.02 (7.71 s)	0.37 (62.98 s)
oclFDTD3d	1 (0.80 s)	6.15 (0.13 s)	2.16 (0.37 s)
Stencil2D	1 (20.65 s)	2.53 (8.16 s)	1.16 (17.80 s)
oclDCT8x8	1 (75.96 ms)	3.42 (22.20 ms)	2.27 (33.46 ms)
oclNbody	1 (10.63 s)	1.20 (8.82 s)	0.74 (14.36 s)
*NaiveMatrixMul	1 (258.16 s)	33.44 (7.72 s)	4.10 (62.98 s)
Kernel	MIC+IntelOCL	MIC+OurOCL	MIC+OMP
oclMatrixMul	1.94 (12.03 s)	3.93 (5.93 s)	3.74 (6.23 s)
oclFDTD3d	2.22 (0.36 s)	5.71 (0.14 s)	4.21 (0.19 s)
Stencil2D	1.83 (11.26 s)	2.42 (8.53 s)	1.95 (10.59 s)
oclDCT8x8	1.43 (53.22 ms)	4.18 (18.19 ms)	4.52 (16.81 ms)
oclNbody	1.13 (9.44 s)	1.24 (8.59 s)	1.38 (7.70 s)
*NaiveMatrixMul	4.55 (56.73 s)	43.76 (5.90 s)	41.44 (6.23 s)

通过开发工作组间和工作项间的并行性，IntelOCL 对于多处理器核和核内 SIMD 单元的利用非常高效。但通过实验发现，它处理同步的开销处在分区域方法和 Twin Peaks 方法之间^[59]。因此，OurOCL 相比 IntelOCL 的性能提升应当主要来源于去除局部存储和同步，部分来源于对数据局部性的重开发。Kernel 程序 oclNbody 在两个硬件平台上的性能提升都是最小的，原因在于它的计算是最为密集的，局部存储和同步导致的开销只占了执行时间的很小一部分。两个模板计算 Kernel (oclFDTD3d 和 Stencil2D) 在 MIC 上的性能提升远低于在 CPU 上。这是由于它们是高度访存密集的，只有少部分时间用于计算，而大部分时间用于访存，因此 MIC 难以体现出并行计算能力的优势。同样是由于访存过于密集，并且没有针对 MIC 的环状 Cache 互联进行优化，这两个 Kernel 在 MIC 上的性能 (MIC+OurOCL) 甚至低于它们在 CPU 上的性能 (CPU+OurOCL)。此外，因为多种开销的消除和数据局部性的大幅提升，NaiveMatrixMul 在两个平台上均获得了

可观的性能提升。

为了探究 OurOCL 可以将 Kernel 性能提升到何种程度，表5.8也给出了各 Kernel 对应的 OpenMP 版本的相对性能。这里的 OpenMP 版本来自于各 Kernel 的宿主程序中（用于检查 Kernel 执行结果的正确性），通过加入合适的 OpenMP 指导语句进行实现。此外，在 MIC 上执行 OpenMP 程序时使用的是本地模式（Native Mode）。上述的 OpenMP 版本并不是未优化的。一些 Kernel 的宿主程序，如 oclDCT8x8，已经进行了 CPU 特定的局部性优化。并且由于没有了 OpenCL 的语义限制，*icc* 编译器也可以更加自由地进行多种自动优化。从性能对比还可以看出，相比 CPU，编译器为 MIC 进行的优化更为激进和有效。因此，oclDCT8x8 在 MIC+OMP 下的性能略高于它在 MIC+OurOCL 下的性能。尽管 oclNbody 的 OpenMP 版本并没有像 oclDCT8x8 那样的专门优化，它在 MIC 下的性能仍然高于 MIC+OurOCL。目前其原因还尚不清晰，本节估计，这是因为 oclNbody 有着非常简单的且利于编译优化的访存模式，同时 MIC 上 OpenMP 的调度方式要优于 OurOCL 的静态调度器。总的来讲，无论在 CPU 还是 MIC 上，基于 OurOCL 的 OpenCL Kernel 性能接近于甚至优于 OpenMP 版本的性能。这足以说明本章的代码转换方法和调度器可以减轻 GPU 特定 Kernel 中的负面性能要素，并将性能提升到接近于适度优化的 CPU 特定程序。

图5.7显示了适应体系结构的后继优化中每一步带来的性能提升。图中的纵坐标轴表示的是对应于各自原 GPU 特定 Kernel 的相对性能。由于后继优化中的循环交换步骤只对 CPU 下的 oclMatrixMul 和 NaiveMatrixMul 产生了多个候选交换方案，因此只有上图中的这两个 Kernel 才显示出最后一步优化。其它的 Kernel 程序只有一种循环交换方案，因此略去了最后一步。从图中可以看出，有的 Kernel 在折叠后由于数据局部性差以及并行性未被开发，性能急剧下降（oclMatrixMul、oclFDTD3d、oclNbody），但是本章的后继优化将会激进地提取并行性并重开发局部性。对于有的 Kernel 来说（Stencil2D、oclDCT8x8），去除局部存储和同步带来的性能提升抵消掉了折叠带来的局部性降低，因此在折叠后有着和原 GPU 特定 Kernel 接近或者更好的性能。而本章的后继优化仍然能进一步提升这些 Kernel 的性能。

根据图5.7，向量化为 oclMatrixMul 和 oclFDTD3d 带来了巨大的性能提升，一些提升甚至超过了 CPU 的 SIMD 宽度。原因在于向量化之前需要进行循环分布和交换，它们也会改变数据局部性，因此两种性能提升叠加在了一起。但是，向量化对于 oclNbody 来说效果不明显，这是因为 oclNbody 中个体的速度和位置被紧密地存储在一起，即按“结构的数组”（Array of Structure）方式进行存储。这将严重阻碍向量化的顺利进行。NaiveMatrixMul 的性能提升曲线和 oclMatrixMul 几乎

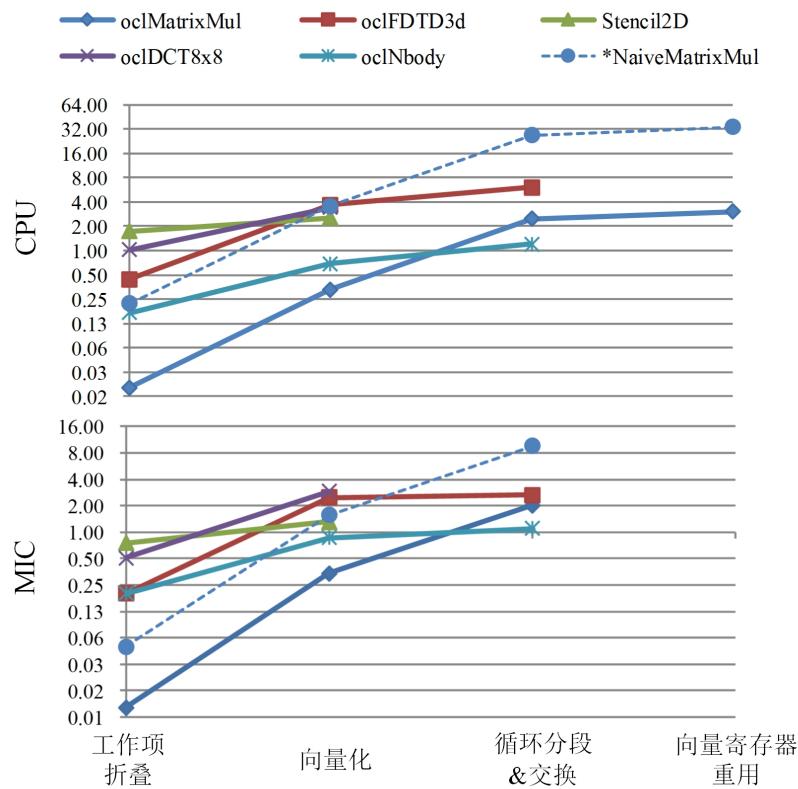


图 5.7 后继优化中各步骤带来的性能提升

一样，这是由于两者在工作项折叠后有着一样的控制流。NaiveMatrixMul 需要进行循环分段，而在循环分段后，两者的代码将变得相同。

5.8 小结

上一章中，通过在异构平台上基于 OpenCL 实现高性能 TLD 算法，OpenCL 的性能移植性问题被暴露出来。为此，本章进行了深入的探讨和解决。本章提出了一套新的代码转换方法，该方法能够有效地提升 GPU 特定 Kernel 程序在多核 / 众核 CPU 上的性能移植性。本章的方法解决了现有方法普遍忽略的两个重要问题：一是在 CPU 上使用局部存储数组可能对性能带来负面影响，即额外的数据拷贝和同步开销；二是忽视或者盲目继承 GPU 特定 Kernel 中的数据局部性特征，可能带来性能损失。借助于本章新提出的数组访问描述式，在工作项折叠过程中，所有的冗余局部存储数组和对应的同步都将被消除。后继优化过程中，本章不仅从原 GPU 特定 Kernel 中提取并行性和局部性特征，还会考虑目标 CPU 的体系结构细节，以进一步提升 Kernel 程序在 CPU 上的性能。实验显示，无论对于经典的多核 CPU，还是新兴的众核协处理器 MIC，包含本章方法的新 OpenCL 运行时都能取得超越 Intel 官方运行时的性能。借助本章的工作成果，在异构平台上实现高

性能跟踪器时，可专注于面向 GPU 的优化，或是重用已有的 GPU 特定代码，极大提高编程效率。本章的方法仍然存在诸多不足，例如无法支持间接数组访问、静态调度算法不够高效、没有考虑工作组间的并行优化等。未来将针对这些不足进行进一步的研究。

第六章 总结与展望

6.1 工作总结

6.2 未来研究方向

致 谢

参考文献

- [1] Wu Y, Lim J, Yang M-H. Online Object Tracking: A Benchmark [C]. In IEEE Conference on Computer Vision and Pattern Recognition. 2013: 2411–2418.
- [2] Lee J, Kim J, Seo S, et al. An OpenCL Framework for Heterogeneous Multicores with Local Memory [C]. In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. 2010: 193–204.
- [3] Lucas B D, Kanade T. An Iterative Image Registration Technique with an Application to Stereo Vision [C]. In Proceedings of the International Joint Conference on Artificial Intelligence. 1981: 674–679.
- [4] Tomasi C, Kanade T. Detection and Tracking of Point Features, CMU-CS-91-132 [R/OL]. 1991. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.131.5899&rep=rep1&type=pdf>.
- [5] Wang N, Yeung D Y. Learning a Deep Compact Image Representation for Visual Tracking [C]. In Annual Conference on Neural Information Processing System. 2013: 809–817.
- [6] Li H, Li Y, Porikli F. DeepTrack: Learning Discriminative Feature Representations by Convolutional Neural Networks for Visual Tracking [C]. In British Machine Vision Conference. 2014: 1–12.
- [7] Bolme D S, Beveridge J R, Draper B A, et al. Visual Object Tracking using Adaptive Correlation Filters [C]. In IEEE Conference on Computer Vision and Pattern Recognition. 2010: 2544–2550.
- [8] Henriques J F, Caseiro R, Martins P, et al. Exploiting the Circulant Structure of Tracking-by-Detection with Kernels [C]. In European Conference on Computer Vision. 2012: 702–715.
- [9] Henriques J F, Caseiro R, Martins P, et al. High-Speed Tracking with Kernelized Correlation Filters [J]. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2015.
- [10] Danelljan M, Häger G, Felsberg M. Accurate Scale Estimation for Robust Visual Tracking [C]. In British Machine Vision Conference. 2014.
- [11] Danelljan M, Shahbaz Khan F, Felsberg M, et al. Adaptive Color Attributes for Real-Time Visual Tracking [C]. In IEEE Conference on Computer Vision and Pattern Recognition. 2014: 1090–1097.

- [12] Li Y, Zhu J. A Scale Adaptive Kernel Correlation Filter Tracker with Feature Integration [C]. In European Conference on Computer Vision Workshop. 2014: 254–265.
 - [13] Liu T, Wang G, Yang Q. Real-Time Part-Based Visual Tracking via Adaptive Correlation Filters [C]. In IEEE Conference on Computer Vision and Pattern Recognition. 2015: 4902–4912.
 - [14] Girshick R, Donahue J, Darrell T, et al. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation [C]. In IEEE Conference on Computer Vision and Pattern Recognition. 2014: 580–587.
 - [15] He K, Zhang X, Ren S, et al. Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition [C]. In European Conference on Computer Vision. 2014: 346–361.
 - [16] Hosang J, Benenson R, Dollár P, et al. What Makes for Effective Detection Proposals? [J]. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2015.
 - [17] Hosang J, Benenson R, Schiele B. How Good are Detection Proposals, Really? [C]. In BMVC. 2014.
 - [18] Zitnick C L, Dollár P. Edge Boxes: Locating Object Proposals from Edges [C]. In European Conference on Computer Vision. 2014: 391–405.
 - [19] Baker S, Matthews I. Lucas-Kanade 20 Years on: A Unifying Framework [J]. International Journal of Computer Vision. 2004, 56 (3): 221–255.
 - [20] Liu B, Huang J, Yang L, et al. Robust Tracking Using Local Sparse Appearance Model and K-Selection [C]. In IEEE Conference on Computer Vision and Pattern Recognition. 2011: 1313–1320.
 - [21] Jia X, Lu H, Yang M-H. Visual Tracking via Adaptive Structural Local Sparse Appearance Model [C]. In IEEE Conference on Computer Vision and Pattern Recognition. 2012: 1822–1829.
 - [22] Zhong W, Lu H, Yang M-H. Robust Object Tracking via Sparsity-Based Collaborative Model [C]. In IEEE Conference on Computer Vision and Pattern Recognition. 2012: 1838–1845.
 - [23] Kwon J, Lee K M. Visual Tracking Decomposition [C]. In IEEE Conference on Computer Vision and Pattern Recognition. 2010: 1269–1276.
 - [24] Godec M, Roth P M, Bischof H. Hough-Based Tracking of Non-Rigid Objects [C]. In International Conference on Computer Vision. 2011: 81–88.
-

-
- [25] Kalal Z, Matas J, Mikolajczyk K. P-N Learning: Bootstrapping Binary Classifiers by Structural Constraints [C]. In IEEE Computer Society Conference on Computer Vision and Pattern Recognition. 2010: 49–56.
- [26] Kalal Z, Mikolajczyk K, Matas J. Tracking-Learning-Detection [J]. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2012, 34 (7): 1409–1422.
- [27] Zhang K, Zhang L, Zhang D, et al. Fast Visual Tracking via Dense Spatio-Temporal Context Learning [C]. In European Conference on Computer Vision. 2014: 127–141.
- [28] Dalal N, Triggs B. Histograms of Oriented Gradients for Human Detection [C]. In IEEE Conference on Computer Vision and Pattern Recognition. 2005: 886–893.
- [29] Van de Weijer J, Schmid C, Verbeek J, et al. Learning Color Names for Real-World Applications [J]. IEEE Transactions on Image Processing. 2009, 18 (7): 1512–1523.
- [30] Khan F S, Anwer R M, Van De Weijer J, et al. Color Attributes for Object Detection [C]. In IEEE Computer Society Conference on Computer Vision and Pattern Recognition. 2012: 3306–3313.
- [31] Khan F S, Rao M A, Van De Weijer J, et al. Coloring Action Recognition in Still Images [J]. International Journal of Computer Vision. 2013, 105 (3): 205–221.
- [32] Dollár P, Zitnick C L. Structured Forests for Fast Edge Detection [C]. In International Conference on Computer Vision. 2013: 1841–1848.
- [33] Dollár P. Piotr's Computer Vision Matlab Toolbox (PMT). <http://vision.ucsd.edu/~pdollar/toolbox/doc/index.html>.
- [34] Wu Y, Lim J, Yang M-H. Visual Tracker Benchmark v1.0. 2015. http://cvlab.hanyang.ac.kr/tracker_benchmark/benchmark_v10.html.
- [35] Hare S, Saffari A, Torr P H S. Struck: Structured Output Tracking with Kernels [C]. In International Conference on Computer Vision. 2011: 263–270.
- [36] Kristan M, Pflugfelder R, Leonardis A, et al. The Visual Object Tracking VOT2013 Challenge Results [C]. In IEEE International Conference on Computer Vision Workshop. 2013: 98–111.
- [37] Maresca M E, Petrosino A. The Matrioska Tracking Algorithm on LTDT2014 Dataset [C]. In IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops. 2014: 720–725.
- [38] Kristan M, Pflugfelder R, Leonardis A, et al. The Visual Object Tracking VOT2014 Challenge Results [C]. In Lecture Notes in Computer Science. 2015: 191–217.

-
- [39] Kristan M, Pflugfelder R, Leonardis A, et al. The Visual Object Tracking VOT2015 Challenge Results [C]. In IEEE International Conference on Computer Vision Workshop. 2015: 1–27.
- [40] Nam H, Han B. Learning Multi-Domain Convolutional Neural Networks for Visual Tracking [C/OL]. In arXiv preprint arXiv:1510.07945. 2015: 4293–4302. <http://arxiv.org/abs/1510.07945>.
- [41] Snapdragon 835 mobile platform. 2017. <https://www.qualcomm.com/products/snapdragon/processors/835>.
- [42] Stone J E, Hallock M J, Phillips J C, et al. Evaluation of Emerging Energy-Efficient Heterogeneous Computing Platforms for Biomolecular and Cellular Simulation Workloads [C]. In IEEE 28th International Parallel and Distributed Processing Symposium Workshops. 2016: 89–100.
- [43] TOP500. TOP500 lists: November 2010. 2010. <https://www.top500.org/lists/2010/11/highlights/>.
- [44] Munshi A. The OpenCL Specification [M/OL]. 2011. <http://www.khronos.org/opencl>.
- [45] Girshick R. Fast R-CNN [C]. In Proceedings of the IEEE International Conference on Computer Vision. 2015: 1440–1448.
- [46] Ren S, He K, Girshick R, et al. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks [C]. In Conference on Neural Information Processing Systems. 2015: 1–10.
- [47] Open Multi-Processing. 2015. <http://www.openmp.org/>.
- [48] NVIDIA Corporation. CUDA C Programming Guide [M/OL]. 2017. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.
- [49] Mattson T, Bordoloi U. OpenCL: an Introduction for HPC Programmers. 2011. <https://indico.cern.ch/event/138427/sessions/11397/attachments/116552/165428/OpenCL-intro-ISC11.pdf>.
- [50] Shi J, Tomasi C. Good Features to Track [C]. In Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. 1994: 593–600.
- [51] Özysal M, Fua P, Lepetit V. Fast Keypoint Recognition in Ten Lines of Code [C]. In Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. 2007.
- [52] Lewis J. Fast Normalized Cross-Correlation [J]. Vision Interface. 1995, 10: 120–123.
- [53] Kalal Z. OpenTLD. 2011. <https://github.com/zk00006/OpenTLD>.
- [54] Gurcan I, Temizel A. Heterogeneous CPU-GPU Tracking-Learning-Detection (H-TLD) for Real-Time Object Tracking [J]. Journal of Real-Time Image Processing. 2015: 1–15.

- [55] OpenCV Library. 2016. <http://opencv.org/>.
- [56] OpenCL Drivers and Runtimes for Intel Architecture. 2016. <https://software.intel.com/en-us/articles/opencl-drivers>.
- [57] Intel Corporation. Intel Xeon Phi Coprocessor x100 Product Family, 328209 004EN [R]. 2015.
- [58] TOP500. TOP500 lists: November 2014. 2014. <http://top500.org/lists/2014/11/>.
- [59] Stratton J A, Kim H, Jablin T B, et al. Performance Portability in Accelerated Parallel Kernels, IMPACT-13-01 [R]. 2013.
- [60] Baskaran M M, Bondhugula U, Krishnamoorthy S, et al. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs [C]. In Proceedings of the 22nd annual international conference on Supercomputing. 2008: 225–234.
- [61] Pennycook S J, Hammond S D, Wright S A, et al. An Investigation of the Performance Portability of OpenCL [J]. Journal of Parallel and Distributed Computing. 2013, 73 (11): 1439–1450.
- [62] Rul S, Vandierendonck H, D’Haene J, et al. An Experimental Study on Performance Portability of OpenCL Kernels [C]. In Proceedings of the 2010 Symposium on Application Accelerators in High Performance Computing. 2010.
- [63] Dong H, Ghosh D, Zafar F, et al. Cross-Platform OpenCL Code and Performance Portability for CPU and GPU Architectures Investigated with a Climate and Weather Physics Model [C]. In Proceedings of the 41st International Conference on Parallel Processing Workshops. 2012: 126–134.
- [64] Gummaraju J, Morichetti L, Houston M, et al. Twin Peaks: a Software Platform for Heterogeneous Computing on General-Purpose and Graphics Processors [C]. In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. 2010: 205–216.
- [65] Stratton J A, Stone S S, Hwu W M W. MCUDA: An Effective Implementation of CUDA Kernels for Multi-Core CPUs [C]. In Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing. 2008: 16–30.
- [66] Stratton J A, Grover V, Marathe J, et al. Efficient Compilation of Fine-Grained SPMD Threaded Programs for Multicore CPUs [C]. In Proceedings of the 8th annual IEEE/ACM International Symposium on Code Generation and Optimization. 2010: 111–119.
- [67] Intel Corporation. Intel SDK for OpenCL Applications XE 2013 Optimization Guide. 2013.
- [68] Intel Corporation. CEAN Language Extension and Programming Model [R/OL]. https://software.intel.com/sites/default/files/fd/c5/CEAN_Model.docx.

-
- [69] Huang D, Wen M, Xun C, et al. Automated Transformation of GPU-Specific OpenCL Kernels Targeting Performance Portability on Multi-Core/Many-Core CPUs [C]. In Proceedings of Euro-Par 2014 & Lecture Notes in Computer Science. 2014: 210–221.
- [70] Fang J, Sips H, Jaaskelainen P, et al. Grover: Looking for Performance Improvement by Disabling Local Memory Usage in OpenCL Kernels [C]. In Proceedings of the 43rd International Conference on Parallel Processing. Minneapolis, USA, 2014: 162–171.
- [71] Fang J, Sips H, Varbanescu A L. Aristotle: A Performance Impact Indicator for the OpenCL Kernels using Local Memory [J]. Scientific Programming. 2014, 22 (3): 239–257.
- [72] Du P, Weber R, Luszczek P, et al. From CUDA to OpenCL: Towards a Performance-Portable Solution for Multi-Platform GPU Programming [J]. Parallel Computing. 2012, 38 (8): 391–407.
- [73] Phothilimthana P M, Ansel J, Ragan-Kelley J, et al. Portable Performance on Heterogeneous Architectures [C]. In Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems. 2013: 431–444.
- [74] Ansel J, Chan C, Wong Y L, et al. PetaBricks: A Language and Compiler for Algorithmic Choice [C/OL]. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2009: 38–49.
<http://portal.acm.org/citation.cfm?doid=1543135.1542481%5Cncode.google.com/p/petabricks/>.
- [75] Shen Z, Li Z, Yew P. An Empirical Study of Fortran Programs for Parallelizing Compilers [J]. IEEE Transactions on Parallel and Distributed Systems. 1990, 1 (3): 356–364.
- [76] Paek Y, Hoeftlinger J, Padua D. Efficient and Precise Array Access Analysis [J]. ACM Transactions on Programming Languages and Systems. 2002, 24 (1): 65–109.
- [77] Triolet R, Irigoin F, Feautrier P. Direct Parallelization of Call Statements [C]. In ACM SIGPLAN Notices. 1986: 176–185.
- [78] Balasundaram V, Kennedy K. A Technique for Summarizing Data Access and its Use in Parallelism Enhancing Transformations [C]. In Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation. 1989: 41–53.
- [79] Bastoul C. Code Generation in the Polyhedral Model is Easier than You Think [C]. In Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques. 2004: 7–16.
- [80] Jang B, Schaa D, Mistry P, et al. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures [J]. IEEE Transactions on Parallel and Distributed Systems. 2011, 22 (1): 105–118.
- [81] NVIDIA Corporation. NVIDIA GPU Computing SDK 4.2. 2013. <http://developer.nvidia.com/gpu-computing-sdk>.

- [82] Danalis A, Marin G, McCurdy C, et al. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite [C]. In Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. 2010: 63–74.
- [83] Nvidia Corporation. OpenCL Best Practices Guide. 2011.
- [84] Nvidia Corporation. OpenCL Programming Guide for the CUDA Architecture. 2011.
- [85] Steven S M. Advanced Compiler Design and Implementation [M]. Morgan Kaufmann, 1997.
- [86] Intel Corporation. A Guide to Vectorization with Intel C++ Compilers. 2012.
- [87] Allen R, Kennedy K. Optimizing Compilers for Modern Architectures: A Dependence-Based Approach [M]. Morgan Kaufmann San Francisco, 2002.
- [88] LLVM Team and others. Clang: A C Language Family Frontend for LLVM. 2012. <http://clang.llvm.org/>.
- [89] Lattner C, Adve V. The LLVM Compiler Framework and Infrastructure Tutorial [M]. In Languages and Compilers for High Performance Computing. Springer, 2005: 15–16.
- [90] FreeOCL. FreeOCL: Multi-Platform Implementation of OpenCL 1.2 Targeting CPUs. 2012. <https://code.google.com/p/freeocl>.
- [91] Intel Corporation. Intel SDK for OpenCL Applications. 2013. <https://software.intel.com/en-us/intel-opencl>.

