计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.200600033

基于 GPU 的特征脸算法优化研究

李 繁[1] 严 星[2] 张晓宇[1]
1 新疆财经大学网络与实验教学中心 乌鲁木齐 830012
2 新疆财经大学信息管理学院 乌鲁木齐 830012

摘 要 特征脸算法是基于脸部表征的常用人脸辨识方法之一。当训练数据量较大时,不管是训练还是测试模块都非常耗时。基于此,采用 CUDA 并行运算架构实现 GPU 加速特征脸算法。针对 GPU 并行运算的效果取决于硬件规格、算法本身的复杂度和可并行性,以及程序开发者使用 GPU 的并行化方式等因素,文中首先提出在特征脸算法训练阶段的计算平均值、zero mean、正规化特征脸等计算步骤以及测试阶段的投影到特征脸空间、计算欧几里得距离等计算步骤使用 GPU 优化加速;其次在相应计算步骤采用不同的并行化加速方法并做出效能评估。实验结果表明,在人脸训练数据量在 320～1920 的范围内,各计算步骤加速效果明显。与 Intel i7-5960X 相比,GTX1060 显示适配器在训练模块中可达到平均约 71.7 倍的加速效果,在测试模块中可达到平均约 34.1 倍的加速效果。

关键词:人脸辨识;特征脸;GPU 并行运算;旋转运算;核心函数
中图法分类号 TP301

Optimization of GPU-based Eigenface Algorithm

LI Fan[1], YAN Xing[2] and ZHANG Xiao-yu[1]
1 Network & Experimental Teaching Center, Xinjiang University of Finance and Economics, Urumqi 830012, China
2 School of Information Management, Xinjiang University of Finance and Economics, Urumqi 830012, China

Abstract Eigenface algorithm is one of the commonly used face recognition methods based on facial representation. When the amount of training data is large, it is very time-consuming both training and testing modules. Based on this, the CUDA parallel computing architecture is used to implement GPU accelerated eigenface algorithm. The effect of GPU parallel computing depends on the hardware specifications, the complexity and parallelism of the algorithm itself, and the parallelization method used by the program developer to use GPU. Therefore, this paper first proposes the calculation of the average value and zero mean in the training phase of the eigenface algorithm. The calculation steps such as normalizing the eigenface and the calculation steps of the projection to the eigenface space and calculating the Euclidean distance in the test phase are optimized and accelerated by GPU. Secondly, different parallelization acceleration methods are used in the corresponding calculation steps and performance evaluation is made. Experimental results show that in the range of face training data from 320 to 1920, the acceleration effect of each calculation step is obvious. Compared with Intel i7-5960X, the GTX1060 display adapter can achieve an average acceleration effect of about 71.7 times in the training module, and an average acceleration effect of about 34.1 times in the test module.

Keywords Face recognition, Eigenface, GPU parallel computing, Rotary operation, Kernel function

1 引言

在人脸识别系统中,为了提高辨识率,通常采用比较高效的算法,或是通过增加训练数据量和人脸的分辨率,从而获得更多的特征信息。但这两种方法下,更高的辨识率往往伴随着更高的时间复杂度。然而,在许多实际应用场景中,一个能够实时侦测、追踪、辨识的人脸辨识系统是非常必要的。在无法简化运算复杂度的情况下,如果计算机的指令周期无法跟上实时的影像串流,就只能舍弃一部分帧数(Frames),但这样会造成影像信息不足,进而影响系统的辨识率。因此,兼顾辨识率和实时性是实际人脸辨识系统必须考虑的问题。

由于 GPU 的并行运算有助于加速许多应用程序,因此 GPGPU(General-Purpose computing on Graphics Processing Units)被广泛应用。CUDA(Compute Unified Device Architecture)是由 Nvidia 提出的 GPGPU 开发环境,程序设计师可以直接用 C,C++或 Fortran 等来操作 GPU,大幅降低了

到稿日期:2020-06-04 返修日期:2020-08-28 本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
基金项目:国家自然科学基金(41830101);新疆社科基金(17BTQ093);新疆财经大学青年博士基金(2015BS003)
This work was supported by the National Natural Science Foundation of China(41830101), Social Science Foundation of Xinjiang Uygur Autonomous Region(17BTQ093) and Doctoral Research Start-up Fund of Xinjiang University of Finance and Economics(2015BS003).
通信作者:李繁(lifanxj@163.com)

GPGPU 的开发门槛[1]。CUDA 已被应用于很多领域，如在密码学中用来加速 AES 加密算法[2]、在生物医学中用于核磁共振成像（MRI）[3]、在天体物理学中用于并行化 N-body 模拟[4]。

若将 GPGPU 的并行处理能力应用于实时的人脸辨识系统中，则系统即便在处理庞大的人脸数据库、特征信息或复杂的算法时，也不会影响其实时辨识能力。本文的研究目标是利用 CUDA 加速特征脸的运算，提升人脸训练和测试模块的速度。

2 相关工作

2.1 特征脸中特征向量的保留

在主成分分析（PCA）算法中，根据特征值的大小，特征向量能够表示数据间差异值的程度。特征值越大，其对应的特征向量越能够表现出数据间的差异，该类特征值为主要的成分；特征值越小则其对应的特征向量越接近噪声。因此，特征向量并非保留得越多就越好。然而，该保留多少特征向量取决于时间和辨识度间的平衡，保留越少的特征向量就越节省运算时间，但辨识度也会受影响；反之，保留越多的特征向量就越耗时，同时也不能保证辨识率越高。

文献[5]的实验表明，不是保留越多的特征向量准确度就越高，在保留超过 225 个特征向量后，准确度开始缓慢下降。文献[6]发现，在训练数据时只需保留 20 个特征向量就能达到最高的辨识率，可见保留特征向量的个数和总训练数据量的比例会因为数据库的不同而有所差异。还有研究通过经验分析，找出一个决定合适比例的公式，例如 energy dimension 或 stretching dimension 方法[7-8]。将特征值从大到小排序，energy dimension 方法是保留前 K 个特征向量，确保 K 个特征向量对应的特征值求和除以所有特征值求和的结果大于一定的阈值，典型阈值为 0.9。stretching dimension 方法则是保留前 K 个特征向量，确保第 $K+1$ 个特征向量对应的特征值除以最大的特征值小于一定的阈值，典型阈值为 0.01。具体如式(1)、式(2)所示，其中 λ 为特征值。

$$energy_K=\frac{\sum_{i=1}^{K}\lambda_i}{\sum_{i=1}^{M}\lambda_i}>0.9 \tag{1}$$

$$stretch_{K+1}=\frac{\lambda_{K+1}}{\lambda_1}<0.01 \tag{2}$$

鉴于经排序后的特征值呈现指数递减的趋势，这两种方法所保留的特征向量个数相对于总训练样本数的比例也会随着总训练样本数的增加而呈现指数递减的趋势。本文着重研究 GPU 对特征脸算法的加速效益，因此本文实验中保留的特征向量数固定为总训练样本数的 20%。

2.2 GPU 加速特征脸的相关研究

文献[9]通过 GPU 加速训练数据和测试数据中的投影到特征脸空间这个步骤，在训练时将若干个训练数据并行处理，一个 thread 负责处理一个训练数据，此并行化方法要求每个 thread 都要处理投影到特征脸的投影运算。然而，此并行化方法还有改善的空间，原因是每个 thread 在训练阶段都要负责一个训练数据投影到全部特征脸的运算，负荷非常大。文献[10]除了使用投影到特征脸空间的方法外，还使用 GPU 来加速共变异数和特征脸矩阵的运算，其实验结果表明，每个 thread 负责处理的运算越少并不代表加速的效果会越好。文献[11]提出的加速方法还对平均值、zero mean、正规化特征脸、转置步骤的运算进行了加速，但其只提出 zero mean 和共变异数矩阵这两个步骤的并行化方法。采用 GPU 并行化计算 zero mean 和共变异数矩阵的方法，在计算 zero mean 的步骤中，将 block 数量对应到一个数据的大小，每个 block 中的 thread 为训练数据的总个数，如此一来，输出矩阵的每个元素都能由其中一个 thread 来运算。而在计算共变异数矩阵时，thread 和 block 的数量对应于输出矩阵的宽和高，每个 block 计算输出矩阵的一列元素，其中每个 thread 计算一个输出矩阵的元素。文献[12]将传统 Jacobi 的方法用于 CUDA 并行化加速，通过矩阵的复制找出矩阵上三角元素中最大绝对值的元素，再计算该元素对应的行列索引，利用索引让多个 threads 并行地更新元素。文献[13]基于 CUDA 并行化 Cyclic Jacobi 方法，提出了另外 4 种并行化算法。其实验结果表明，并行化 Cyclic Jacobi 方法比并行化传统 Jacobi 方法有更大的加速效能，这与 Cyclic Jacobi 方法在每次席卷（sweep）中不必找到最大绝对值的元素且一次并行处理每列和每行元素有关。

本文在深入研究上述文献中的并行化方法后，提出了在训练阶段中计算平均值、zero mean、正规化特征脸以及在测试阶段中投影到特征脸空间和计算欧几里得距离这 5 个步骤的并行化方法，实现了 block 中的每个 thread 需要计算多个元素的并行方式。

3 研究方法

本文采用 CUDA 加速特征脸算法的人脸训练模块和测试模块，特征脸算法的具体流程如图 1 所示。

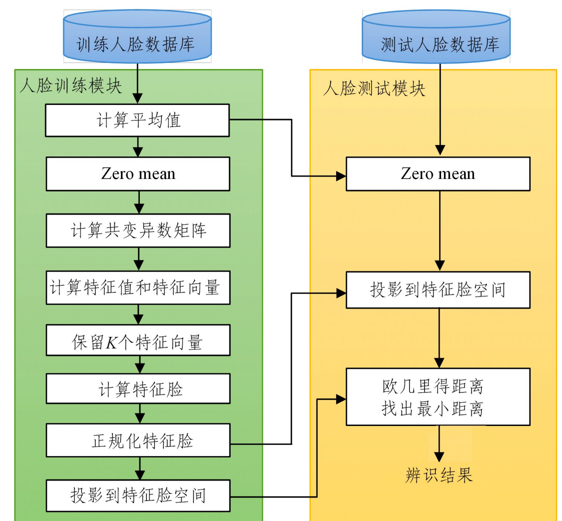

图 1 特征脸的流程图

Fig. 1 Flow chart of eigenface

本节分成训练模块和测试模块两个部分，提出使用 CUDA 并行处理特征脸各个步骤的方法，包括核心函数（kernel function）的输入、输出参数，block 和 thread 的数量配置以及

执行内容。在表达名称上，P(num_pixels)代表一张训练人脸图像的像素总数（宽×高）；M(num_face)为训练人脸图像的总个数；K(num_keepFace)为保留的特征向量个数。在切割成二维子区块的并行化方法中，定义 TILE_DIM 的大小为 32，BLOCK_ROWS 的大小为 8。

3.1 训练模块

训练模块的具体步骤如下。

步骤 1　计算平均值（$\boldsymbol{\Psi}=\frac{1}{M}\sum_{i=1}^{M}\boldsymbol{\Gamma}_i$）。

输入为 M 个一维训练人脸集合（$[\Gamma_1\Gamma_2\Gamma_3\cdots\Gamma_M]^{\mathrm{T}}$）和 M；输出为平均值一维阵列（$[avg_1, avg_2, \cdots, avg_P]$）；thread 数量为 32 的倍数（手动调整）；block 数量为 P。

算法描述如下：

共有 P 个 block，每个 block 负责 M 个训练人脸中其中一个 pixel 的平均值运算。每个 block 中的 threads 负责对 M 个训练人脸中的其中一个 pixel 求和，最后取得平均值。如果 thread 的数量小于 M，则表示一个 thread 要负责多个 pixel 的求和，此过程用 thread 数量（*blockDim. x*）作为索引位移量来达到连续存取存储器的效果，在所有的 threads 都将汇总值存储到共享内存（shared memory）后，用 reduction 做并行求和。reduction 采用完全展开（completely unrolled）的方式[14]，达到最佳连续存取内存空间，求和完成后再除以 M，即获得该 pixel 的平均值。

步骤 2　Zero Mean（将每个训练人脸减去平均值），即 $\boldsymbol{\Phi}_i=\boldsymbol{\Gamma}_i-\boldsymbol{\Psi}$。

方法为一个 thread 计算多个 pixels 的差值；输入为 M 个一维训练人脸集合 $[\Gamma_1\Gamma_2\Gamma_3\cdots\Gamma_M]$、平均值一维数组 $[avg_1 avg_2 avg_3\cdots avg_P]$ 和 M；输出为 $\boldsymbol{A}([\Phi_1\Phi_2\Phi_3\cdots\Phi_M])$；thread 数量为 32 的倍数（手动调整）；block 数量为 P。

算法描述如下：

共有 P 个 block，每个 block 负责 M 个训练人脸中的一个 pixel 差值运算。当 threads 的数量小于 M 时，则表示一个 thread 要负责多个 pixels 差值的运算。

步骤 3　计算共变异数矩阵（$\boldsymbol{L}=\frac{1}{M}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$）。

此步骤分成两个核心函数运算，一个是矩阵转置，另一个是矩阵相乘。

(1)将 $\boldsymbol{\Phi}$ 矩阵转置（$[\Phi_1\Phi_2\Phi_3\cdots\Phi_M]^{\mathrm{T}}$）

输入为训练人脸减平均值的矩阵（令 $\boldsymbol{A}=[\Phi_1\Phi_2\Phi_3\cdots\Phi_M]$）、$\boldsymbol{A}$ 的宽度(M)和 $\boldsymbol{A}$ 的高度(P)；输出为 $\boldsymbol{A}$ 的转置矩阵 $\boldsymbol{A}^{\mathrm{T}}$。thread 和 block 参数设置：使用二维的 thread 和 block 参数，thread 的 x 方向大小为 *TILE_DIM*，y 方向大小为 *BLOCK_ROWS*；block 的 x 方向大小为 $\boldsymbol{A}$ 的宽度(M)除以 *TILE_DIM*，y 方向的小大为 $\boldsymbol{A}$ 的高度(P)除以 *TILE_DIM*。将两个参数的结果都取整。

算法描述如下：

将输入和输出矩阵都拆分成二维的 block，每个 block 中的 threads 以列的方式读取输入矩阵的值，然后以列的方式写入到 shared memory。在 shared memory 的最后一行新增一行空白空间，使得 shared memory 在以行的方式读取时不会造成 bank conflict，接着以列的方式写入到输出矩阵就能完成转置操作。其中，每个 thread 处理（TILE_DIM/BLOCK_ROWS）个元素的转置，让一个 thread 处理多个元素，使一个 SM 能同时执行较多的 block。

(2) 计算 $\boldsymbol{L}=\frac{1}{M}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})$

输入为 $\boldsymbol{A}^{\mathrm{T}}([\Phi_1\Phi_2\Phi_3\cdots\Phi_M]^{\mathrm{T}})$、$\boldsymbol{A}([\Phi_1\Phi_2\Phi_3\cdots\Phi_M])$、$\boldsymbol{A}^{\mathrm{T}}$ 的高度(M)、$\boldsymbol{A}^{\mathrm{T}}$ 的宽度(P)和 $\boldsymbol{A}$ 的宽度(M)；输出为 $\boldsymbol{L}(=\frac{1}{M}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}))$。thread 和 block 的参数设置为：

dim3 *dimBlock*(*TILE_DIM*, *TILE_DIM*)
dim3 *dimGrid*
dimGrid. x=(*M*+*TILE_DIM*−1)/*TILE_DIM*
dimGrid. y=(*M*+*TILE_DIM*−1)/*TILE_DIM*

算法描述如下：

共变异数矩阵中 $\boldsymbol{A}^{\mathrm{T}}$ 和 $\boldsymbol{A}$ 矩阵相乘，采用的方法为共享内存的分块矩阵乘法（tiled matrix multiplication with shared memory）。BLOCK_SIZE 大小即为这里定义的 TILE_DIM。此方法将输入和输出矩阵都分割为宽、高相等的子区块（TILE_DIM×TILE_DIM），block 对应到输出矩阵的子区块，thread 对应到两个输入矩阵的子区块。每个 thread 将对应的两个输入矩阵子区块的元素复制到两个 shared memory 后，通过相乘得到子结果，最后对子结果求和汇总得到输出矩阵中子区块的元素。shared memory 可以隐藏更多内存在读取上所造成的大量延迟，因为每个输入矩阵的元素都会因为输入矩阵的宽、高变大而使得被读取的次数变多。例如 $\boldsymbol{A}^{\mathrm{T}}$ 矩阵的每列元素会被读取的次数为 $\boldsymbol{A}$ 的宽度，而 $\boldsymbol{A}$ 的每行元素会被读取的次数为 $\boldsymbol{A}^{\mathrm{T}}$ 的高度。

步骤 4　计算 $\boldsymbol{L}$ 的特征向量和特征值

采用 Cyclic Jacobi 方法计算特征向量和特征值。在矩阵中元素的旋转运算可拆分成列运算和行运算，为了独立地并行化这两个运算，要避免并行运算的列与列、行与行之间的重叠。因此在并行化矩阵旋转前，先用 chess tournament ordering 方法取得不重复的配对编号，总共会产生 $n/2$ 个配对（n 为共变异数矩阵的宽、高）。如果 n 是奇数，则可以在矩阵的边界补零并将 n 值加 1，这并不影响运算的结果。

每次席卷都会计算矩阵中上三角元素的绝对值和，来检测是否达到结束的阈值。若尚未达到结束阈值，则会进行矩阵的旋转运算。因为上三角元素个数总共为($n\times(n-1)/2$)，其中有($n/2$)个配对可并行运算，所以全部的配对排列组合共需要($n-1$)次迭代运算。在每次旋转运算前，除了要计算配对的编号，还得计算要做相乘运算的旋转参数——$\sin(\phi)$ 和 $\tan(\phi/2)$。旋转运算结束后，为了在计算矩阵中上三角元素的绝对值和时能够连续地存取内存空间，需要把矩阵中上三角元素依序写入另一个矩阵中。算法描述中的 $\boldsymbol{B}$ 用于表示旋转前的矩阵，$\boldsymbol{B}'$ 表示旋转后的矩阵，B'_upper 为储存 $\boldsymbol{B}'$ 矩阵中上三角元素的一维数组。

(1)计算一维数组元素的绝对值和

输入为上三角元素的总数($n\times(n-1)/2$)和 $\boldsymbol{B}'_upper$

(第一次为 L 中上三角元素的一维数组);输出为绝对值和的一维数组;thread 数量为 32 的倍数(手动调整);block 数量可以手动调整。

算法描述如下:

此步骤纯粹计算所有元素的绝对值和,无需考虑 block 之间的相依性,文献[15]中优化 reduction 的方法为 multiple elements per thread,其让一个 thread 处理多个元素,将 reduction 的并行效率最大化。由于 block 之间无法互相等待,在每个 block 计算完绝对值和之后,用 CPU 将这些子结果汇总起来。

(2)计算旋转运算参数

输入为 $\boldsymbol{B}$ 的宽高(n)、$\boldsymbol{B}$ 和配对的迭代次数($i=0$ to $n-2$);输出为 $\boldsymbol{B}$ 的主对角元素数组、行列配对编号数组、行列配对编号的反函数数组、$\sin(\phi)$ 旋转参数数组和 $\tan(\phi/2)$ 旋转参数数组;thread 数量为 32 的倍数(手动调整);block 数量为($n/2+$thread 数量-1) / thread 数量。

算法描述如下:

通过 blockIdx 和 threadIdx,让每个 thread 都能获得不同的索引值,接着用 chess tournament ordering 方法,使得每次迭代取得不重复的配对编号。根据配对编号可计算出该元素的旋转参数 $\sin(\phi)$ 和 $\tan(\phi/2)$,这里用两个一维数组分别存储旋转参数。为了避免后续做分支处理,这里先对主对角元素数组做旋转(第一次旋转初始化主对角元素为 1)。除了记录配对编号和旋转参数之外,还记录了配对编号之间的反函数,让配对编号具有可逆性,其目的是后续做旋转运算时能连续地存取内存空间。

(3)计算旋转运算

输入为 $\boldsymbol{B}$ 的宽高(n)、$\boldsymbol{B}$、行列配对编号一维数组、行列配对编号反函数一维数组、$\sin(\phi)$ 旋转参数一维数组、$\tan(\phi/2)$ 旋转参数一维数组和特征向量矩阵 $\boldsymbol{V}([v_1 v_2 v_3 \cdots v_M]^{\mathrm{T}})$;输出为 $\boldsymbol{B}'$ 和特征向量矩阵;thread 数量为 32 的倍数(手动调整);block 数量为 $n/2$。

算法描述如下:

共有($n/2$)个配对编号,将 block 数量设为($n/2$),每个 block 处理两列元素旋转运算。thread 的数量可手动调整,当 thread 数量小于矩阵边长时,一个 thread 需要做多个元素的旋转运算。

(4)将 $\boldsymbol{B}'$ 中上三角元素的值复制到 $\boldsymbol{B}'_upper$

输入为 $\boldsymbol{B}'$ 的宽高(n)和 $\boldsymbol{B}'$;输出为 $\boldsymbol{B}'_upper$;thread 数量为 32 的倍数(手动调整);block 数量为($n\times(n-1)/2+$thread 数量-1)/thread 数量。

算法描述如下:

此核心函数是为了在计算 $\boldsymbol{B}'$ 中上三角元素的绝对值和时能够连续存取内存,因此将 $\boldsymbol{B}'$ 中上三角元素(不含主对角元素)的值连续写入一个一维数组中。

步骤 5 保留 K 个特征向量($\boldsymbol{V}=[v_1 v_2 v_3 \cdots v_K]^{\mathrm{T}}$)。

$K=M/5$,这一步骤纯粹取前 K 个特征向量,所占时间比例很低,可以直接用 CPU 完成。

步骤 6 计算特征脸($U^T=VA^T$)。

这一步骤的方法与计算共异数矩阵相同,此处省略说明。

步骤 7 正规化特征脸($\| u_i \|=1$)。

输入为 $\boldsymbol{U}^{\mathrm{T}}([u_1 u_2 u_3 \cdots u_k]^{\mathrm{T}})$,$\boldsymbol{U}^{\mathrm{T}}$ 的宽度(P);输出为经正规化的特征脸矩阵 $\boldsymbol{U}'([u_1' u_2' u_3' \cdots u_K']^{\mathrm{T}})$;thread 数量为 32 的倍数(手动调整);block 数量为 K。

算法描述如下:

若要将 K 个特征脸正规化使得每个 $\| u_i \|=1$,就需要先求出每个特征脸的 pixel 值平方和,然后每个 pixel 值都除以 pixel 值平方和的平方根,结果即为所求。每个 block 都负责一个特征脸的正规化运算,求平方和的方法同步骤 1 中将 pixel 加和的方法。最后在计算每个特征脸的 pixel 除以平方和的平方根时,同样让每个 thread 处理多个 pixel 值的计算。

步骤 8 将 M 个人脸训练图像投影到 K 个特征脸空间。

这一步骤的方法是将 $\boldsymbol{U}'$ 乘上 $\boldsymbol{A}$ 矩阵得到投影后的权重矩阵 $\boldsymbol{\Omega}$。

输入为 $\boldsymbol{A}([\Phi_1 \Phi_2 \Phi_3 \cdots \Phi_M])$、$\boldsymbol{U}'([u_1' u_2' u_3' \cdots u_K']^{\mathrm{T}})$、$\boldsymbol{A}$ 的宽度(M)、$\boldsymbol{U}'$ 的高度(K)和 $\boldsymbol{U}'$ 的宽度(P);输出为权重矩阵 $\boldsymbol{\Omega}$ ($[W_1 W_2 W_3 \cdots W_M]$, $W_i=[w_1 w_2 w_3 \cdots w_K]^{\mathrm{T}}$);thread 数量为 32 的倍数(手动调整);block 数量为($M+$thread 数量-1)/thread 数量。

算法描述如下:

所有 block 的 thread 总数量为 M,每个 thread 计算一笔训练数据投影到 K 个特征脸空间的结果,因此每个 thread 要计算出 K 个输出元素(权重值)。为了在测试阶段可以连续存取内存空间,最后将权重矩阵 $\boldsymbol{\Omega}$ 做转置得到 $\boldsymbol{\Omega}^{\mathrm{T}}$。

3.2 测试模块

经过训练阶段所得到的平均值、特征脸和权重矩阵将被用于测试模块对检测数据做计算,最后找出与数据库中最相近的人脸,下面对每个步骤用 CUDA 做并行化运算的具体过程进行说明。

步骤 1 将人脸检测数据减去平均值后投影到特征脸空间,得到权重数组。

这一步骤并行化的方式是让多个 threads 计算一个特征脸的投影运算。

输入为人脸检测数据一维数组$[t_1 t_2 t_3 \cdots t_P]$、平均值一维数组$[avg_1 avg_2 avg_3 \cdots avg_P]$、$\boldsymbol{U}'([u_1' u_2' u_3' \cdots u_K']^{\mathrm{T}})$和 P;输出为检测数据的一维权重数组$[w_1 w_2 w_3 \cdots w_K]$;thread 数量为 32 的倍数(手动调整);block 数量为 K。

算法描述如下:

每个权重值的运算为人脸检测数据和平均值相减,再乘以 K 特征脸得到 K 个权重值。具体方法为设置 K 个 block,让每个 block 负责一个特征脸的权重值运算,每个 thread 负责多个 pixels 值的运算。先对运算的子结果求和,然后再用 reduction 加上所有的子结果得到权重值。

步骤 2 计算检测数据和各训练人脸权重间的欧几里得距离。

这一步骤和步骤 1 的并行化方式一样,都是处理一维数组和矩阵中某一列的元素间的运算。差别仅在于运算的内容不同。

具体方法为多个 threads 计算一个欧几里得距离;输入为

检测数据的一维权重数组[$w_1 w_2 w_3 \cdots w_K$]、训练的权重矩阵 $\boldsymbol{\Omega}^{\mathrm{T}}$([$W_1 W_2 W_3 \cdots W_M$]$^{\mathrm{T}}$)和 K；输出为欧几里得距离一维数组[$d_1 d_2 d_3 \cdots d_M$]；thread 数量为 32 的倍数(手动调整)；block 数量为 M。

算法描述如下：

与步骤 1 相似，共有 M 个 blocks，每个 block 都负责计算检测数据和其中一个训练权重间的距离。每笔训练数据都有 K 个权重值，若 thread 的数量小于 K，则一个 thread 需负责多个权重值的运算。先将权重的差值取平方后相加，然后用 reduction 对所有的子结果求和后开平方根，即可得到结果。

步骤 3　找出与人脸检测数据最接近的人脸类别($\varepsilon = \min \| \Omega - \Omega_i \|$)。

从欧几里得距离一维数组中找出最小值，此最小值所对应的人脸类别即为最接近的人脸类别。这个步骤的时间复杂度仅为 O(n)，用 GPU 并行化的效益有限，故此步骤用 CPU 来运算。

4　实验仿真与分析

4.1　实验环境与方法

实验环境采用 Windows10 64bit 操作系统下的虚拟机，使用 OpenCV 的 API 对读写图像和基本图像进行处理。硬件部分，CPU 使用 Intel i7-5960X 3.0 GHz。GPU 使用两块不同的显示适配器，一块是可虚拟化的图形卡 Nvidia GRID K1，此 GPU 有 4 个处理核心，使用 GPU 直接存取(Pass-through)的技术，可同时让 4 个虚拟机直接存取其中一个核心；另一块显示适配器使用 Nvidia GTX1060。GPU 的详细硬件规格如表 1 所列。

表 1　GPU 硬件规格

Table 1　GPU hardware specifications

GPU 型号	Nvidia GRID K1	Nvidia GTX1060
核心代号	GK107×4	GP106
处理器架构	Kepler	Pascal
处理器频率/MHz	850	1 506
CUDA 核心	768(192/GPU)	1 280(128cores/SM)
SM Count	1×4	10
内存频率/MHz	891	2 002
内存容量	4 096 MB×4	6 144 MB
内存带宽	28.51 GB/s×4	192.2 GB/s

两种 GPU 的差异在于：GRID K1 是专为虚拟环境提供图形功能所设计的，而 GTX1060 为一般家用型显示适配器。在 GPGPU 方面，要比较不同显示适配器间的指令周期，不能仅观察其 CUDA 的核心数量，处理器架构和频率、Streaming Multiprocessor(SM)数量、内存容量和带宽都对指令周期有所影响。在上述几个方面，GTX1060 都比 GRID K1 优异，因为 GRID K1 本身是为虚拟环境和专业图形所设计的，而运算能力并不强。

实验方法是针对训练模块和测试模块的每个步骤，随着数据库中不同的训练人脸数量，呈现 CPU 与 GPU 之间的时间比较与加速倍率，加速倍率由 CPU 的运行时间除以 GPU 的运行时间而得。数据库使用 LFW 的人脸数据库，其中共有 13 000 多张人脸图像，在实验中，我们取其中的 1 920 张并统一人脸图像的大小为 128×128。训练的人脸图像数量为 320，640，…，1 920。测试阶段的实验数据为测试单张人脸图像的执行时间，实验数据皆为 10 次实验结果的平均值，实验所用的浮点数运算统一为单精度。

4.2　训练模块

本节对比各步骤 GPU 和 CPU 的运行时间和加速倍率。一些实验是将 thread 数量设置为 32 的倍数，在不同的显示适配器资源和训练人脸数量下，最佳 thread 数量可能不同，因此实验数据取最佳的 thread 数量下的实验结果。

在计算平均值(ψ)的步骤中，GPU 的运行时间都包含 device 端复制到 host 端所需的内存传输时间，从图 2 和图 3 可以看出，加速倍率随着训练人脸数量的增加而逐步变高，也能看出 GTX1060 在并行运算方面优于 GRID K1，GTX1060 在此步骤平均比 CPU 快约 80.36 倍。

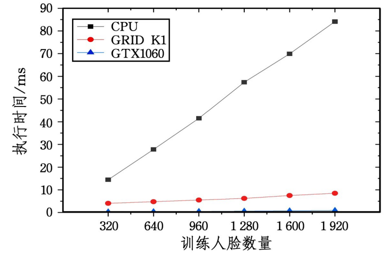

图 2　计算平均值(ψ)的时间比较

Fig. 2　Time comparison of calculating mean(ψ)

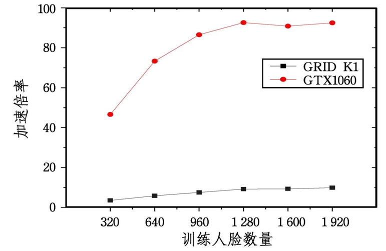

图 3　计算平均值(ψ)的加速倍率

Fig. 3　Speedup ratio of calculating mean(ψ)

计算 zero mean(ϕ)的步骤为一个 thread 计算多个 pixels 的差值，固定 thread 的数量为 32 的倍数则是让一个 thread 处理多个元素 d 运算。实验数据在最佳的 thread 数量下取得。图 4 给出了此步骤的加速倍率，可以看出 GTX1060 在此步骤平均比 CPU 快约 45.57 倍。

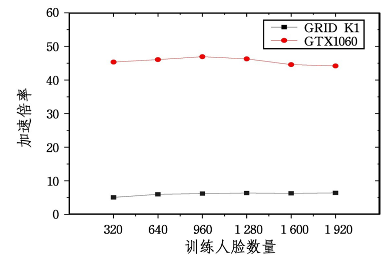

图 4　计算 zero mean(ϕ)的加速倍率

Fig. 4　Speedup ratio of calculating zero mean(ϕ)

图 5 和图 6 给出了计算 ϕ 转置的时间比较和加速倍率。GPU 在此步骤用 shared memory 避免不连续存取所带来的时间延迟，可以看出 GTX1060 在此步骤平均比 CPU 快了约 290.58 倍。

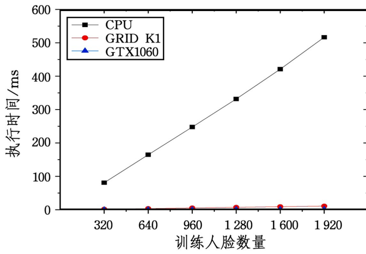

图 5 计算 ϕ 转置的时间比较

Fig. 5 Time comparison of calculating ϕ transpose

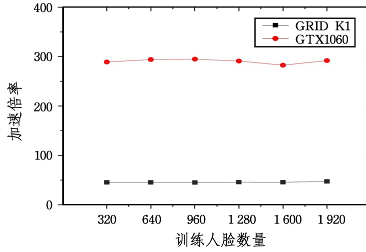

图 6 计算 ϕ 转置的加速倍率

Fig. 6 Speedup ratio of calculating ϕ transpose

在计算共变异数矩阵(**L**)的步骤中，采用 tiled matrix multiplication 方法，将输入和输出矩阵切割成 32×32 的子区块来处理。图 7 给出了使用 tiled matrix multiplication 方法的加速倍率，GTX1060 在此步骤平均比 CPU 快约746.74 倍。

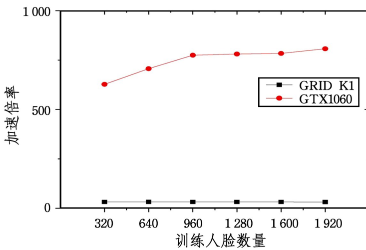

图 7 计算共变异数矩阵(**L**)的加速倍率

Fig. 7 Speedup ratio of calculating covariance matrix(**L**)

图 8 和图 9 给出了计算特征向量的时间比较和加速倍率。在特征脸训练中，计算特征向量是其中一个较为耗时的步骤。从图 9 可以看出，GPU 的加速倍率并不如其他步骤显著。虽然已经将 Cyclic Jacobi 方法的一次席卷中的运算并行化，但在每次迭代中还是需要(n-1)次旋转运算。相较于单纯的矩阵四则运算，计算特征向量是较复杂的算法，因此，其运算加速成效没有其他步骤显著。GTX1060 在此步骤平均比 CPU 快约 38.55 倍。

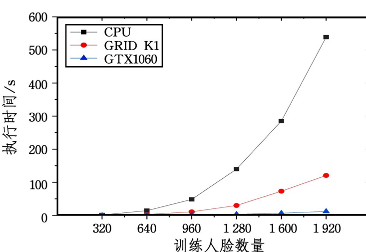

图 8 计算特征向量的时间比较

Fig. 8 Time comparison of calculating feature vector

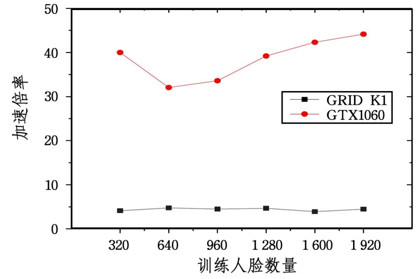

图 9 计算特征向量的加速倍率

Fig. 9 Speedup ratio of calculating feature vector

图 10 给出了计算特征脸(U)的加速倍率。此步骤和计算共变异数矩阵相同，也是矩阵的相乘运算。结果表明，在 GRID K1 和 GTX1060 的执行中，tiled matrix multiplication 的方法效能非常好，GTX1060 平均比 CPU 快约 822.07 倍。

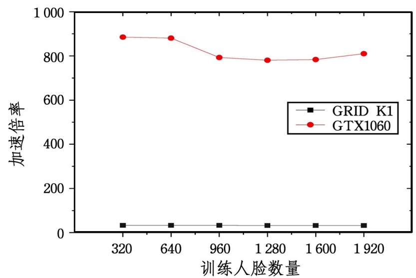

图 10 计算特征脸(U)的加速倍率

Fig. 10 Speedup ratio of calculating Eigenface(U)

图 11 和图 12 给出了计算正规化特征脸($\| u_i \| = 1$)的时间比较和加速倍率。可以看出，GTX1060 在此步骤平均比 CPU 快约 110.89 倍。

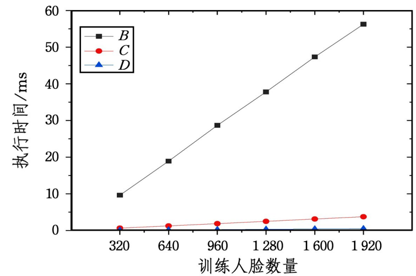

图 11 计算正规化的时间比较

Fig. 11 Time comparison of calculation normalization

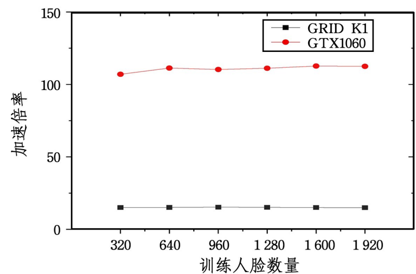

图 12 计算正规化的加速倍率

Fig. 12 Speedup ratio of calculation normalization

图 13 给出了计算训练人脸投影到特征脸空间(Ω)的加速倍率。此步骤也是两个矩阵相乘，与计算共变异数矩阵的方法相同。GTX1060 在此步骤平均比 CPU 快约 823.54 倍。

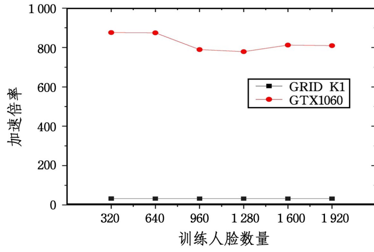

图 13　计算投影到特征脸空间的加速倍率

Fig. 13　Speedup ratio of calculating map to eigenfaces

图 14 给出了全部训练模块的加速倍率。曲线递减是因为计算特征向量的步骤所占用总时间的比例随着训练人脸数量的增多而增大，然而计算特征向量的步骤的加速倍率较其他步骤更低。实验结果表明，GTX1060 在训练模块的整体时间平均比 CPU 快约 71.7 倍。

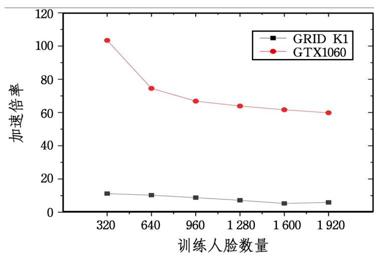

图 14　全部训练模块的加速倍率

Fig. 14　Speedup ratio of all training modules

训练模块中 4 个最耗时的步骤中，计算共变异数矩阵、特征脸、投影到特征脸空间的步骤均在采用 tiled matrix multiplication 的并行化方法时最佳，平均分别加速了 746.74 倍、822.07 倍、823.54 倍。然而，因为计算特征向量的运算较为复杂，此步骤平均只加速了 38.55 倍。

4.3　测试模块

测试模块的实验数据为测试单张人脸所需的执行时间。为了加快 GPU 的运算效率，这里将计算 zero mean 与投影到特征脸空间这两个步骤的时间加在一起进行比较。图 15 和图 16 给出了计算 zero mean(ϕ)和投影到特征脸空间(Ω)的时间比较和加速倍率。与训练模块一样，测试模块也有投影到特征脸空间的步骤，但不同的是，这里只需要将一个测试数据投影到特征脸空间，因此使用的并行化方法也不同。设置 K 个 block，每个 block 都对应一个特征脸的投影，一个 block 中有多个 threads 并行处理同一个特征脸的投影。实验结果表明，GTX1060 平均比 CPU 快约 98.56 倍。

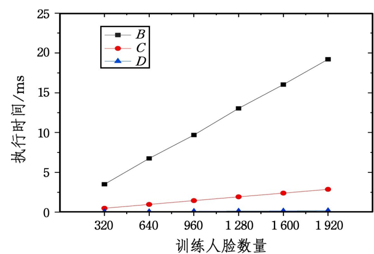

图 15　计算 zero mean(ϕ)和投影到特征脸空间(Ω)的时间比较

Fig. 15　Time comparison of calculating zero mean(ϕ) and map to Eigenface space(Ω)

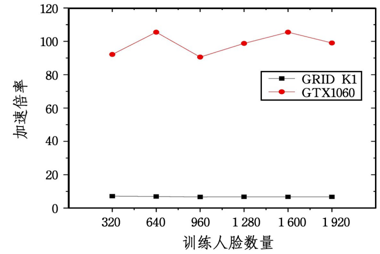

图 16　计算 zero mean(ϕ)和投影到特征脸空间(Ω)的加速倍率

Fig. 16　Speedup ratio of calculating zero mean(ϕ) and map to Eigenface space(Ω)

图 17 和图 18 给出了计算检测数据和训练数据间的欧几里得距离以及找出最小距离(ε)的时间比较和加速倍率。block 的数量为 M，固定 thread 的数量为 32 的倍数，每个 block 都计算一个检测数据与一个训练数据的距离，其中有多个 threads 并行处理。

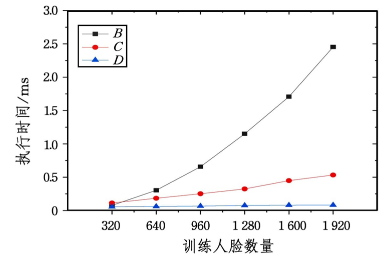

图 17　计算最小欧几里得距离(ε)的时间比较

Fig. 17　Time comparison of calculating minimum Euclidean distance(ε)

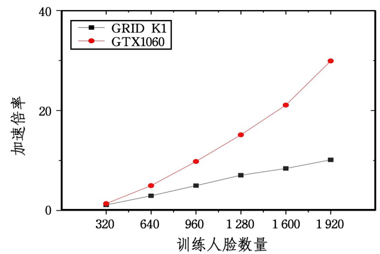

图 18　计算最小欧几里得距离(ε)的加速倍率

Fig. 18　Speedup ratio of calculating minimum Euclidean distance(ε)

图 19 和图 20 给出了是测试模块所有步骤的总时间比较和加速倍率。其中，时间包含将检测数据从 host 端复制到 device 端和欧几里得距离数组从 device 端复制到 host 端的传输时间。实验结果表明，在训练人脸数量为 1 920 时，GPU 使用 GTX1060 的测试时间为 0.45 ms，相较于 CPU 的 21.66 ms 快了约 50.07 倍，整体平均快了约 34.1 倍。

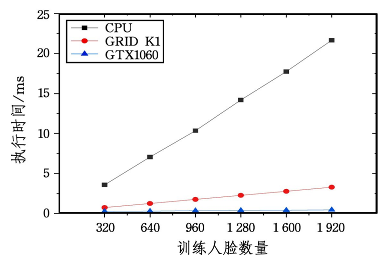

图 19　测试模块所有步骤的时间比较

Fig. 19　Time comparison of total steps of test module

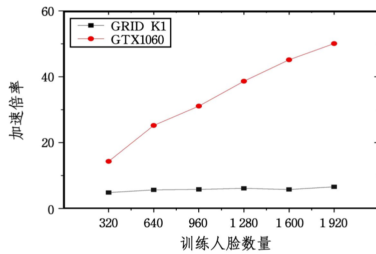

图 20 测试模块所有步骤的加速倍率

Fig. 20 Speedup ratio of total steps of test module

结束语 本文使用 CUDA 并行运算技术来优化加速特征脸算法，对训练模块和测试模块的各个步骤都详细说明了加速的方法。在人脸训练数量为 320～1 920 时，GTX1060 在全部的训练模块中得到了平均约 71.7 倍的加速倍率；而在全部测试模块中得到平均 34.08 倍的加速倍率。在训练模块前 4 个最耗时的步骤中，3 个步骤用 tiled matrix multiplication 方法均可得到较好的加速效果，印证了有效利用 shared memory 可以在矩阵相乘时隐藏大量读取内存的时间。值得进一步探讨的是耗时且难加速的步骤，如计算特征向量的步骤。虽然 Jacobi 方法本身的可并行化程度高，但算法本身的复杂度也比其他步骤更高，相较之下加速成效不明显。是否有其他计算特征向量的算法可以在某种程度上进行并行化，使得加速效率高于 Jacobi 并行化方法，是值得进一步研究的方向。一个全面的人脸识别系统还包含移动对象侦测、人脸侦测、人脸追踪的模块，将这些模块也使用 GPGPU 的运算能力来加速，以提升整体系统的执行效率，是未来值得研究的课题。另一方面，在实际的人脸识别应用中还需考虑辨识率的高低，未来研究可将特征脸与其他算法相结合，在达到更高的辨识率的同时使用 GPU 并行运算的能力以兼顾系统的实时性。

参 考 文 献

[1] MACIEJ B, SKURSKI A, MAREK K, et al. Applications of Ray-Casting in Medical Imaging[J]. Advances in Intelligent Systems & Computing, 2014, 283: 3-14.

[2] JIN X X, DAKU B, KO S B. Improved GPU SIMD control flow efficiency via hybrid warp size mechanism[J]. Microprocessors and Microsystems, 2014, 38(7): 717-729.

[3] LIN Y C, WANG C C, LIN G, et al. A Simple Method to Improve the Quality of Diffusion-Weighted Magnetic Resonance Imaging with Rapid Histologic Correlation in a Murine Model [J]. Molecular Imaging, 2014, 13: 1-8.

[4] WANG W, ZENG X H, WANG F H, et al. Parallel Time-Space Processing Model Based Fast N-body Simulation[J]. Journal of Frontiers of Computer Science & Technology, 2011, 5(11): 63-69.

[5] SAPNA S, ANJALI R, KAMATH S N. Performance Analysis of Parallel Implementation of PCA-based Face Recognition using OpenCL[C]// 2019 4th International Conference on Recent Trends on Electronics, Information, Communication & Technology (RTEICT). IEEE, 2019: 877-881.

[6] DESHPANDE N T, RAVISHANKAR S. Face Detection and Recognition using Viola-Jones algorithm and Fusion of PCA and ANN[J]. Advances in Computational Ences and Technology, 2017, 10(5): 1173-1189.

[7] BANERJEE S, SCHEIRER W, BOWYER K, et al. Fast face image synthesis with minimal training[C]// 2019 IEEE Winter Conference on Applications of Computer Vision (WACV). IEEE, 2019: 2126-2136.

[8] MULGREW, AMY C. A geometric approach to study the relationship between maternal and fetal characteristics and the shape of placental surfaces[J]. Dissertations & Theses Gradworks, 2011, 30(3): 425-436.

[9] WOO Y, YI C, YI Y. Fast PCA-based face recognition on GPUs [C]// IEEE International Conference on Acoustics, Speech & Signal Processing. IEEE, 2013: 2659-2663.

[10] ZHANG D, MABU S, HIRASAWA K. Robust intelligent PCA-based face recognition framework using GNP-fuzzy data mining [J]. IEEJ Transactions on Electrical & Electronic Engineering, 2013, 8(3): 253-262.

[11] FENG C, YU-BO T, MIN Y. Research and Design of Parallel Particle Swarm Optimization Algorithm Based on CUDA[J]. Computer Science, 2014(47): 280-287.

[12] HWANG F N, WEI Z H, HUANG T M, et al. A parallel additive Schwarz preconditioned Jacobi-Davidson algorithm for polynomial eigenvalue problems in quantum dot simulation[J]. Journal of Computational Physics, 2010, 229(8): 2932-2947.

[13] ZHAO T. A Convergence Analysis of the Inexact Simplified Jacobi-Davidson Algorithm for Polynomial Eigenvalue Problems [J]. Journal of Scientific Computing, 2018, 75(3): 1207-1228.

[14] AYRES D L, CUMMINGS M P, BAELE G, et al. BEAGLE 3: Improved performance, scaling, and usability for a high-performance computing library for statistical phylogenetics [J]. Systematic Biology, 2019, 68(6): 1052-1061.

[15] ANDREW R, DINGLE N. Implementing QR Factorization Updating Algorithms on GPUs [J]. Parallel Computing, 2014, 40(7): 161-172.

LI Fan, born in 1974, Ph. D, associate professor. His main research interests include high-performance computing and so on.