基于多核 DSP 架构并行编程技术的研究

吕大鹏[1] 杨珂瑶[1] 王宏伟[1]
LÜ Dapeng YANG Keyao WANG Hongwei

摘 要 为适配嵌入式系统中多核处理器的广泛应用需求，充分发挥多核架构优势以提升系统整体性能，并行编程技术已成为核心支撑手段。文章聚焦多核 DSP 架构，围绕 OpenMP 并行编程技术开展系统性研究。首先，系统阐述 OpenMP 并行编程模型的核心原理、编程规范及运行机制，明确其在多核平台上的并行调度逻辑；其次，结合编译器编译优化行为与 OpenMP 运行时库的工作机制，深入剖析该编程模型在多核 DSP 架构上的底层实现逻辑与适配特性，厘清软硬件协同工作的关键环节；最后，针对多核 DSP 核间数据隔离的固有特性，提出一种基于消息传递的 OpenMP 并行启动机制，有效解决核间协同启动的兼容性问题，为 OpenMP 技术在多核 DSP 平台的高效应用提供技术支撑。

关键词 OpenMP；并行编程；多核 DSP 架构；消息传递；并行启动

doi：10.3969/j.issn.1672-9528.2026.02.015

0 引言

近年来，随着数据规模的加速扩张，单核处理器难以满足日益增长的计算需求，多核处理器逐渐成为嵌入式系统的主流选择。为了更好地发挥多核处理器的性能，并行编程技术应运而生。并行编程技术能够将不同的计算任务分配到各处理器核心上并发执行，让所有处理器核都能参与到计算过程中，大幅缩短了数据处理时间。并行编程技术能够更高效地利用硬件资源，充分发挥多核处理器的性能优势，但并行编程对系统开发人员提出了更高的要求 [1]。

早期的并行编程技术需要开发者了解硬件特性，在具体的编程过程中通过直接操作硬件实现程序的并发执行，这种开发方式不仅效率低且错误率高。为了使开发人员能够更加专注于系统功能实现，避免陷入对多核特性的处理，提高软件开发的效率和正确率，逐渐出现了类别丰富的并行编程抽象模型。这些抽象模型简化了并行编程的开发接口，让开发者无需深入理解底层硬件即可编写并行程序，大大推进了并行编程的应用。其中 OpenMP 并行编程模型因其简单易用、可移植性和可扩展性强等特点，在嵌入式多核处理器领域得到了广泛应用 [2]。

本文主要关注面向多核 DSP 架构的 OpenMP 并行编程模型，选择飞腾 6678 多核处理器作为硬件平台，完成对 OpenMP 并行编程模型的研究。

1. 中国航空工业集团公司西安航空计算技术研究所
陕西西安 710065

1 OpenMP 概述

1.1 OpenMP 简介

OpenMP（open multi-processing）并行编程模型是由 OpenMP Architecture Review Board 牵头提出的一套被广泛接受的适用于多核处理器并行程序设计的方案。OpenMP 基于共享内存方式，通过在程序中添加编译指导指令，将串行的计算过程拆分成并行的计算过程，达到并行编程的目的，从而提高系统的整体计算效率。OpenMP 支持 C、C++、Fortran 等多种编程语言 [3-5]。

OpenMP 提供了一套跨平台的多线程并行编程集合，包括了编译器指令、库函数、环境变量等内容。开发者在编码过程中，不需要编写复杂的并行逻辑，只需要在源程序代码中加入专用的关键字，指明程序并行的策略，同时注意控制多核间的同步、互斥、通信等操作，即可轻松地将串行程序转换为并行程序，实现多核并行加速的目的。

使用 OpenMP 编程模型能够有效地简化软件开发人员在多核处理器上并行编程的编码工作，充分挖掘多核处理器的性能潜力，提升程序性能，优化系统的计算效率。

1.2 OpenMP 的运行时间模型

OpenMP 采用了 fork-join 运行模型，fork 指在串行执行的程序运行过程中创建若干子任务，并将这些子任务独立运行在不同的处理器核上；Join 指等到所有子任务都运行结束后，各子任务同步终止并将控制权移回主任务 [6-7]。OpenMP 主要是通过将串行的计算过程拆分成并行的计算过程这一方式，提高系统整体的计算效率。例如，一段使用 for 循环的代码，若使用 OpenMP 编程模型，则会将 for 循环的循环次数划分为若干部分，派生出对应的多个子任务，每个子任务处理其

中的一个部分，这些子任务同时运行，当所有子任务运行完成后，再将结果合并汇总。这段程序的运行过程如图 1 所示，程序开始执行时，只有主核上的主任务在运行。在运行过程中，主核上的主任务 fork 出多个子任务，这些子任务与主任务共同组成一个任务组，并行执行并行区域内的代码。在并行区域执行完毕后，所有子任务会合并（join）回主任务，继续执行后续的串行代码。

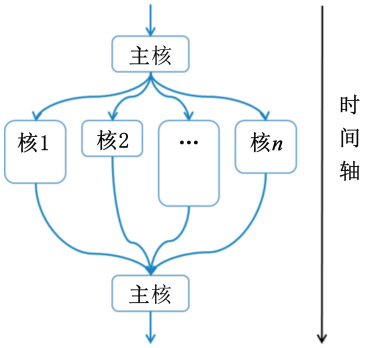

图 1 fork-join 模型

1.3 OpenMP 开发支持

基于 OpenMP 编程模型开发的程序能够正确运行通常需要编译器、OpenMP 运行时库以及操作系统三方面的共同支持。

OpenMP 是通过在原有代码中加入指导编译的语句，告知编译器在代码编译时对原有任务进行拆分，使其能够分散在不同的处理器核上运行，从而实现程序的并行执行。编译器获取程序中插入的 OpenMP 关键字，根据关键字信息明确程序的拆分方式，包括对哪一部分进行多任务拆分、如何进行拆分、fork 出多少个子任务以及最终 join 的时机等。在最终的映像文件中，编译器基于关键字信息自动在合适的代码位置插入用于支持并行运行的扩展代码，扩展代码通常是对 OpenMP 运行时库中接口的调用。

OpenMP 运行时库是 OpenMP 并行编程模型的核心组件，介于硬件和编译器之间的跨平台接口库。OpenMP 运行时库提供了子任务创建、任务调度、同步机制以及环境变量处理等功能接口和相关的数据结构。OpenMP 运行时库与编译器有密切的关联，每一个编译器都有对应的 OpenMP 运行时库，例如 GNU C 编译器的 OpenMP 运行时库为 GUN OMP[8]。通过 OpenMP 运行时库与编译器的搭配使用，完成程序并行运行的支持。

操作系统则主要是为 OpenMP 运行时库的任务创建、管理、调度等方面的接口提供支持。在 GOMP 中，通过 POSIX 接口调用操作系统服务。

2 多核 DSP 架构的 OpenMP 并行编程实现

基于前面章节的分析，可以明确 OpenMP 程序的并行执行主要依赖编译器和其对应的运行时库，通过对 GCC 编译器和对应运行时库的分析，对多核 DSP 架构上 OpenMP 并行编程模型的实现进行研究。

2.1 编译指导语句影响分析

OpenMP 并行编程模型的实现原理是在原有程序中增加编译指导语句，编译器识别编译指导关键字，对程序的执行过程进行拆分，并自动扩充支持并行执行的代码，实现程序的并行处理。现对编译指导语句如何实现这一过程进行分析。

设计一个 OpenMP 并行运行的用例，如图 2 所示代码，该段代码构造了一个循环次数为 100 次的空循环。遵循 OpenMP 并行编程模型，对这段代码进行拆分，在 for 循环开始之前，增加编译指导语句“#pragma omp parallel for private(*i*)”，表示后面的代码要进行并行化处理。

```
int i;
__asm(" nop 9");    /*反汇编标记*/
#pragma omp parallel for private(i)
for (i=0; i < 100; i++)
{
    //do nothing
}    /* end of parallel i loop */

__asm(" nop 9");    /*反汇编标记*/
```

图 2 代码示意图

“#pragma omp parallel for private(*i*)”是 OpenMP 中常用的编译指导指令，其功能是将 for 循环并行化。其中“#pragma omp”用于告知编译器，在此标识后的代码要进行拆分，转换为并行处理。“parallel”表示开始构建并行运行区域，主任务创建子任务团队，所有子任务共同执行后续的代码。“for”表示将 for 循环处理为并行执行，即 for 循环不同的循环迭代分配给不同的子任务执行。“private(*i*)”表示并行运行区域中的循环变量 *i* 为子任务私有，即每个子任务执行循环时的变量 *i* 是私有的，用于避免不同子任务访问同一变量造成数据竞争。在 OpenMP 中，变量的作用域默认是共享的，即多个子任务可读写同一内存地址，但循环变量 *i* 在串行代码中会被反复修改，例如 *i*++，若并行化时不进行特殊处理，会导致多个子任务同时读写 *i*，引发不可预测的结果，比如部分迭代被跳过或重复执行等。

将编译器对上述代码的编译结果进行反汇编，如图 3 反汇编代码所示。其中 0xc0298d4 和 0xc0298f4 两处的汇编指令“NOP 9”分别与原代码中的反汇编标记对应，则说明上述 OpenMP 程序的并行执行区域位于 0xc0298d4~0xc0298f4 指令之间。通过分析上述汇编指令，可以发现在增加了“#pragma omp parallel for private(*i*)”后，编译器对原有代码自动进行了扩充，插入了 GOMP_parallel_start、__omp_0_main、GOMP_parallel_end 三个接口的调用。其中，GOMP_parallel_start 和 GOMP_parallel_end 两个接口由 OpenMP 运行时库提供，__omp_0_main 接口为编译器自行生成。

```
0c0298c0          main:
0c0298c0   103b1c10          CALLP.S1     __push_rts (PC+121056 = 0x0c0471a0),A3
0c0298c4          $CSRL34:
0c0298c4   07fe6052          ADDK.S2      -832,B15
0c0298c8   00101fdb          MV.L2X       A4,B0
0c0298cc   0200cffe ||       STW.D2T2     B4,*+B15[207]
0c0298d0   0000cefe          STW.D2T2     B0,*+B15[206]
0c0298d4   00010000          NOP          9
0c0298d8   024cfa28          MVK.S1       0xffff99f4,A4
0c0298dc   02060169          MVKH.S1      0xc020000,A4
0c0298e0   1037f413 ||       CALLP.S2     GOMP_parallel_start (PC+114592 = 0x0c045880),B3
0c0298e4       0627 ||       MVK.L2       0,B4
0c0298e6       0726 ||       MVK.L1       0,A6
0c0298e8          $CSRL577:
0c0298e8          $CSRL0:
0c0298e8       0626          MVK.L1       0,A4
0c0298ea       115b ||       CALLP.S2     __omp_0_main (PC+276 = 0x0c0299f4),B3
0c0298ec          $CSRL2:
0c0298ec   1001e012          CALLP.S2     GOMP_parallel_end (PC+3840 = 0x0c02a7e0),B3
0c0298f0          $CSRL4:
0c0298f0   00010000          NOP          9
0c0298f4   100068fe          ADDAN.D2     B15,104,B0
```

图 3 示例代码反汇编示意图

2.2 关键接口分析

在上文中通过反汇编代码看到，编译器在原有代码中插入 GOMP_parallel_start、__omp_0_main、GOMP_parallel_end 三个接口的调用。下面对这三个用于支持并行编程的重要接口进行分析。

（1）__omp_0_main

__omp_0_main 接口的反汇编代码如图 4 所示，可以看出整个 __omp_0_main 分为两大部分，前半部分为编译器自动生成的代码，后半部分为示例代码中的循环体。这里重点分析由编译器自动生成的代码。__omp_0_main 接口中自动生成的代码部分主要包含了 omp_get_thread_num、omp_get_num_threads 以及 __divi 三个接口的调用，如图 4 中的框 1、2、4。

```
          __omp_0_main:
          $CSRL35:
01be14f6          STW.D2T2     B3,*B15--[16]
    0426          MVK.L1       0,A0
    ac45 ||       STW.D2T1     A4,*B15[1]
e8001000          .fphead      n, 1, W, BU, nobr, nosat, 1000000b
10385813    1     CALLP.S2     omp_get_thread_num (PC+115392 = 0x0c045cc0),B3
    cc05 ||       STW.D2T1     A0,*B15[2]
          $CSRL36:
    acc5          STW.D2T1     A4,*B15[5]
10385012 ||   2   CALLP.S2     omp_get_num_threads (PC+115328 = 0x0c045c80),B3
          $CSRL38:
    2426          MVK.L1       1,A0
    0427          MVK.L2       0,B0
    3047          MV.L2X       A0,B1
    fc85 ||       STW.D2T2     B0,*B15[7]
    ec81          ADD.L2       B1,-1,B0
    fc9d ||       LDW.D2T2     *B15[7],B1
    ad05          STW.D2T1     A0,*B15[9]
    8072    3     MVK.S1       100,A0
ef400508          .fphead      n, 1, W, BU, nobr, nosat, 1111010b
    8d05          STW.D2T1     A0,*B15[8]
    9046 ||       MV.L1X       B0,A4
10292c12 ||   4   CALLP.S2     __divi (PC+84320 = 0x0c03e380),B3
          $CSRL40:
    ed45          STW.D2T1     A4,*B15[11]
    dccd          LDW.D2T2     *B15[6],B4
10292c12 ||       CALLP.S2     __divi (PC+84320 = 0x0c03e380),B3
eSa008c0          .fphead      n, 1, W, BU, nobr, nosat, 0101101b
          $CSRL42:
```

图 4 omp_0_main 反汇编示意图

omp_get_thread_num 用于在并行区域内获取当前的子任务 ID。omp_get_num_threads 用于获取当前并行区域内的子任务数量。这两个接口是编写 OpenMP 并行程序的基础接口，在 OpenMP 并行程序执行的过程中，并行区域通过获取子任务 ID 和数量，实现对子任务的管理和分配，调节负载均衡，控制数据访问。

分析图 4 的反汇编代码，梳理出 __omp_0_main 接口的核心处理流程，如图 5 所示，主要分为 4 个步骤：

①创建子任务团队。调用 omp_get_thread_num 获取当前子任务的 ID，调用 omp_get_num_threads 获取当前并行区域内的总并行子任务数量。在 OpenMP 并行编程模型中，为并行区域创建的任务数量有多种配置方式，开发者根据具体需要进行设置，确保每个子任务处理的工作量均衡；

②循环迭代分配。根据循环总次数和创建的子任务数，将迭代空间划分为多个块并行执行。调用 __divi 函数，用总循环次数除以并行子任务数量，计算每个块需要循环的次数。总循环次数即原始代码中 for 循环的次数，在此用例中为 100；

③计算当前的任务 ID 并行执行的范围。OpenMP 将 for 循环的总次数划分为 *N* 个块，每个块对应一个任务，每个任务负责一个数组块的范围。在此用例中，总次数为 100，假设总并行任务数量为 10，则 for 循环会被划分 10 个块，每个块负责 10 次循环。那么 ID 为 0 的子任务负责第 0 次到第 9 次这 10 次循环，ID 为 1 的子任务负责第 10 次到第 19 次这 10 次循环，以此类推；

④按照前面步骤获取的信息，执行循环体。

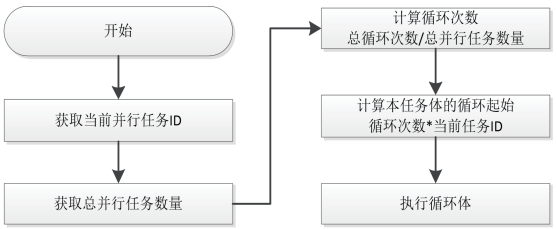

图 5 编译器自动插入部分的处理逻辑

（2）GOMP_parallel_start

GOMP_parallel_start 是 GNU OpenMP（GOMP）运行时库中的底层函数，负责并行执行环境的初始化、子任务创建和启动，是 OpenMP 编程模型中并行区域的入口函数。调用 GOMP_parallel_start 接口用于构造 OpenMP 并行运行环境。

在 TI-C6XX 架构上，使用 A4 寄存器传递第一个参数，从图 3 可知，GOMP_parallel_start 接口的第一个参数被赋值为 0xC0299F4（图 3 中框 1 所示），而 __omp_0_main 的地址正是 0xC0299F4（图 3 中框 3 所示），由此得知，GOMP_parallel_start 接口的第一个参数为指向 __omp_0_main 的函数指针。因此，在 TI C6XX 架构中，编译器在遇到 OpenMP 的 parallel 关键字后，将会自动插入并行启动接口 GOMP_parallel_start 的调用，并将需并行执行的循环体封装为函数接口，作为参数传递给 GOMP_parallel_start。

（3）GOMP_parallel_end

GOMP_parallel_end 接口也是 GNU OpenMP（GOMP）运行时库中的底层函数，负责子任务同步、资源回收、子任务删除等功能，是并行区域的出口函数。调用 GOMP_parallel_end 接口用于将 OpenMP 运行时环境恢复到进入并行区域前的状态。

GOMP_parallel_end 接口要与 GOMP_parallel_start 接口成对出现。因此，当编译器遇到 OpenMP 的 parallel 关键字后，会自动在原程序中增加 GOMP_parallel_start 和 GOMP_parallel_end 接口的调用。

3 基于消息的并行启动设计

3.1 设计方案

DSP（digital signal processor）处理器是一类专门为实时数字信号处理设计的处理器，主要面向信号处理类的应用场景，DSP 处理器的架构设计与通用处理器存在较大差异。以 TI C6XX 架构为例，其与通用处理器最显著的差异为硬件不维护多个处理器核之间的核内数据 CACHE 的一致性，即不同处理器核在数据上完全隔离，这一特性使得传统的 SMP 模式不能直接运行在 TI C6XX 架构上 [9-10]。

同时，由于无法保证多个处理器核间数据的一致性，若直接使用 OpenMP 编程模型进行并行化编程，会因为多核间数据不一致造成运行结果不正确。造成多核间数据不一致的主要原因是无法保证并行区域的开始位置和结束位置的数据是最新数据，如果能够精确管理并行区域的启动位置和结束位置的数据，则能够保证 OpenMP 并行运行结果的正确性。

为了解决这一问题，设计了一个基于消息的并行启动管理方案。该方案的核心思想是在主核运行主程序，在从核运行并行子任务，程序执行到并行区域时，主核向从核发送消息，通知从核开始并行执行。完成并行执行后，从核给主核发消息通知主核，主核上的主程序汇集各处理器核的子任务运行结果，完成并行计算。整个并行区域的执行过程通过主从核之间互相发送消息保证程序执行时使用的数据为最新数据，从而保证计算结果的准确性。该设计采用主从异构的方式支持 OpenMP 模型的并行编程，能够对并行区域的运行时机进行管理，从而实现对 OpenMP 并行运算的支持。

基于前面章节的分析得知，OpenMP 并行编程模型是通过在程序中自动扩充和插入运行时库接口的调用，实现并行运行的支持。因此，可以通过修改被自动插入的运行时库接口，实现基于消息的并行启动管理方案。该方案主要管理并行区域执行的开始位置和结束位置，因此主要涉及运行时库提供的启动接口和结束接口，即 GOMP_parallel_start 和 GOMP_parallel_end。在这两个接口的合适位置增加消息发送的操作，实现对并行区域执行时机的管理。

其中，GOMP_parallel_start 主要参数为：

（1）void(* fptr)(void*)，即编译器生成的并行部分的函数指针；

（2）void* data，传递给并行部分的参数；

（3）int tCnt，需要创建的并行子任务数。

前两个用于向从核提供并行所需的数据，是消息的主体；后一个用于主核控制发出消息的数量。

为了便于消息的管理，降低消息使用过程中由于动态申请、多核互斥等带来的系统开销，可以采用存储空间与管理逻辑相隔离的设计思路，将一组消息空间映射到多个逻辑队列中，再将整个消息空间映射到硬件的共享存储空间，从而为每个处理器核提供了独立的消息队列，其结构如图 6 所示。

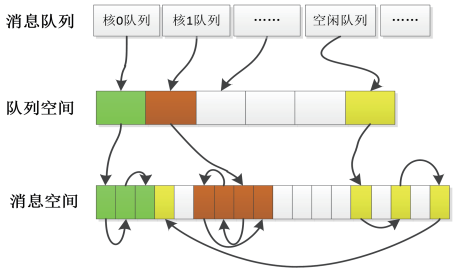

图 6 队列结构示意图

3.2 运行流程

通过对 OpenMP 运行时库接口的修改，在启动接口和结束接口中增加基于消息的主从并行机制，对并行区域执行的启动时机和结束时机进行管理，保证了并行区域启动位置和结束位置的数据一致性，从而确保了 OpenMP 并行计算的正确性。整个运行过程如图 7 所示，具体步骤为：（1）主核进行系统初始化，并加载应用执行；（2）从核完成简单初始化后，进入查询等待状态，持续查询对应消息队列是否有新消息进入，一旦获取到新消息，即跳转执行消息所携带的并行体执行；（3）当应用中出现并行指令时，通过 GOMP_parallel_start 接口向对应的从核发送消息，从核接收到消息后开始执行，即为并行子任务开始执行；（4）主核上的程序等待所有从核上的子任务执行完成后，汇聚结果完成运算。

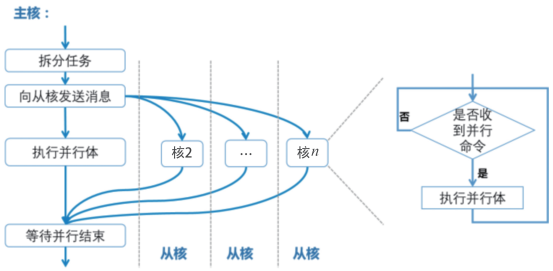

图 7 基于消息的主从并行逻辑示意图

4 小结与未来展望

本文对 OpenMP 并行编程模型的实现原理进行分析，着重讨论了 OpenMP 编程模型的并行实现方式和关键接口，并基于多核 DSP 架构及其编译器，对 OpenMP 程序运行时的编译器行为、OpenMP 运行时库支持以及整个运行过程进行了研究。最后提出了一种基于消息的主从并行启动设计，解决了多核 DSP 架构上多个处理器核间数据不一致导致 OpenMP 程序运行结果不正确的问题。该方案已经在飞腾 6678N 和某操作系统上，采用 8 个处理器核进行验证，在没有依赖的情况下，采用 OpenMP 进行并行处理后，最高加速比可达到 5。但方案中对于并行区域的数据区切分采用了完全平均的方式，且数据位于每个处理器核的私有 cache 中处理。如果切分位置没有按处理器的 cache 线对齐，且在 cache 使能的情

况下，会造成后完成的处理器核数据影响相邻处理器核数据的情况出现。目前此种情况仅能依赖编程人员通过仔细的数据结构设计才能避免，后续可作为进一步工作的切入点。

参考文献：

[1] 胡博文 . 面向异构嵌入式实时系统的并行任务编程框架设计与实现 [D]. 成都 : 电子科技大学 ,2024.

[2] 罗秋明 , 明仲 , 刘刚 , 等 . OpenMP 编译原理及实现技术 [M]. 北京 : 清华大学出版社 ,2012.

[3]OpenMP Architecture Review Board. OpenMP Application Programming Interface. Version 5.0[R/OL].(2018-11-04)[2025-12-30].https://www.booksofall.com/openmp-application-programming-interface-standard-version-5-0/.

[4]CAPPELLO F, ETIEMBLE D. MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks[C/OL]//SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing.Piscataway:IEEE,2000[2025-12-30].https://ieeexplore.ieee.org/abstract/document/1592725.DOI:10.1109/SC.2000.10001.

[5] 熊波 . 面向 OpenMP 的并行任务调度设计与实现 [D]. 成都 : 电子科技大学 ,2022.

[6] 王子聪 . 基于 OpenMP 的多核 DSP 并行优化方法的研究及应用 [D]. 长沙 : 国防科学技术大学 ,2014.

[7]TROBEC R, SLIVNIK B,BULIĆ P, et al. Introduction to Parallel computing [M]. Berlin:Springer,2018.

[8]GCC Team. GNU OpenMP Runtime Library Implementation[R]. GCC 9.3 Documentation, 2020.

[9]KRITIKOS W V, SCHMIDT A G, FRENCH M, et al. Parallel programming models for multicore DSP systems[J]. CCF transactions on high performance computing, 2020, 2: 382–400.

[10]Texas Instruments. TMS320C6000 DSP Cache User's Guide [R]. Literature Number SPRU862, 2010.

【作者简介】

吕大鹏（1983—），男，陕西西安人，硕士，高级工程师，研究方向：嵌入式系统。

杨珂瑶（1990—），女，陕西西安人，硕士，高级工程师，研究方向：嵌入式系统。

王宏伟（1989—），男，陕西西安人，硕士，高级工程师，研究方向：嵌入式系统。

（收稿日期：2025-07-15 修回日期：2026-01-29）

（上接第 66 页）

由图 5 可知，本文利用结合自注意力机制与深度学习提取了标线逆反识别特征，将 U-Net 模型作为编码器输出双通道分类特征，结合 Softmax 激活函数计算标线逆反射特征参数，生成 Cross Entropy Loss 识别损失函数进行二分类转换与 Dice 描述，计算此时的特征识别权重。在不同可见度场景下，研究方法获取的识别逆反射强度极值与实际值重叠度较高，未出现显著波峰边界差异问题，证明研究方法的识别效果较好，具有高精度鲁棒性。

3 结语

在交通安全设施体系中，高速公路标线逆反智能识别技术发挥着举足轻重的作用。该技术不仅显著提升了夜间、雨雾等低能见度条件下的道路识别能力，还有效降低了标线反光性能衰减带来的安全风险，从而提升道路的养护效果。通过全自动智能检测，该技术能够提供精准的数据支持，为道路管理部门制定科学的养护计划提供有力依据。本次研究将自注意力机制与深度学习技术深度融合，应用于高速公路标线逆反智能识别中，成功提取了高质量标线图像的局部特征，并进行了自注意力特征整合。通过随机裁剪与光变化模拟，进一步提高了识别的综合质量。这一技术的应用，不仅为保证道路通行安全提供了有力保障，也为推动智能交通系统的发展作出了积极贡献，是交通安全设施智能化升级的重要里程碑。

参考文献：

[1] 彭金栓 , 张淋俊 , 周磊 , 等 . 高速公路跟车情景下认知分心影响机制与识别方法 [J]. 交通运输系统工程与信息 , 2025, 25(1): 221-230.

[2] 霍俊妤 , 王雪松 , 刘倩 , 等 . 基于混合模型的地面道路车道线识别和影响因素分析 [J]. 中国公路学报 ,2025,38(3):1-12.

[3] 何银鑫 , 齐华 , 朱运权 , 等 . 基于多损失融合和混洗注意力的车载 LiDAR 点云道路标线提取方法 [J]. 测绘通报 , 2024(8): 135-140.

[4] 胡敏 , 钟炜 , 毛炜青 , 等 . 基于深度学习的超大城市高精度地图关键要素自动提取 [J]. 测绘通报 ,2024(S1):266-270.

[5] 董侨 , 林烨龙 , 王思可 , 等 . 基于目标检测算法与迭代阈值分割的道路标线可视度评估 [J]. 同济大学学报 (自然科学版), 2023, 51(8):1168-1173.

【作者简介】

田广策（1988—），男，内蒙古赤峰人，本科，中级，研究方向：公路工程。

（收稿日期：2025-07-23 修回日期：2026-01-30）