基于上一章中对于HEVC关键技术和CUDA编程相关技术的理解，本章首先简要介绍官方HEVC测试模型（HEVC Test Model，HM）运行流程，分析其中主要的耗时模块和函数。接下来针对其中的帧内预测模块和变换量化两大模块进行进一步研究。针对帧内预测模块，首先分析其整体逻辑，设计合适的并行算法，然后分别在对其中的率失真函数与角度预测函数进行并行优化算法的设计和实现，然后针对整个模块进行并行优化并研究这一算法下的理论并行度。针对量化变换模块，同样首先分析其整体结构，然后针对其中的DCT变换函数、DST变换函数、蝶形快速变换函数分别设计和实现并行优化算法，并计算各个算法的理论并行度。

## HM模块分析

### HM运行流程

HEVC测试模型（HEVC Test Model，HM）是由德国通讯技术研究所（HHI）发布的官方HEVC标准编解码器实现。HM使用C++语言开发，包含了多个工程，通过跟踪和调试HM代码，可以发现HM各个工程和模块之间的关联关系。其中涉及到视频编码的主要为：包含编码器和编码器公共应用函数的TAppCommon，包含了编码器应用函数的TAppEncoder，包含编码器和解码器公用库函数的TLibCommon，包含编码器库函数的TLibEncoder。下面结合HEVC标准和HM的代码实现，以全帧内预测模式为例，简述其运行流程。

（1）HM中对整个编码过程封装了cTAppEncoder类，整个编码流程在其成员函数encode中实现。在encode编码函数中，处理的编码对象为视频序列划分之后的图像组GOP，在HM中通过图像组末尾iPOCLast以及图像组宽度iNumPicRcvd来标识。GOP依次经copressGOP划分后形成slice，经compressSlice划分后形成CTU、经compressCTU划分后形成CU，最后对CU再进行实际的编码操作。

（2）由于HM配置为全帧内模式，因此CU的编码中仅需要进行帧内预测即可，该过程由xCheckRDCostIntra函数完成，HM接下来分别对CU进行亮度预测和色度预测，其中主要为亮度预测，函数estIntraPredLumaQT主要完成亮度预测。按照HEVC标准，帧内预测的流程需要在平面模式、直流模式以及33种角度预测模式种选择“最佳”模式来为CU进一步进行编码。这一过程由xRecurIntraCodingLumaQT实现。对应HEVC标准，该函数中遍历了35种预测模式，对每种模式分别调用角度预测函数predIntraAng和率失真函数xGetHADs，形成候选模式集合，然后再从中选出最佳的编码模式。

（3）完成帧内预测之后，HM根据HEVC标准，将CU划分为TU，进行变化和量化操作。这一操作的入口函数为xIntraCodingTUBlock，其中量化操作的实现为xQuant，变换操作的实现为xT。

（4）最后进行熵编码，并输出为编码码流。

（此处可以加一个流程图）

### HM耗时模块分析

为了明确优化方向和优化重点，需要首先了解和分析HM代码中各个模块的耗时情况，从而找出其中适合优化且有优化价值的部分。此处，对于HM的模块性能分析主要分为两部分，第一部分为，对于不同视频测试序列，测试和分析HM代码中不同模块的耗时情况。第二部分为，针对其中每个模块，分析其中主要功能函数的耗时情况。

对于同的分辨率和帧率，在全帧内模式下，HM代码中各个模块的运行情况如图所示：

（补图-HM代码运行函数时间占比）

可以看出，在不同的测试序列下，各个模块的运行时间占比基本上一致，因此在下面的各模块分别分析中，不再需要针对不同的测试序列一一测试，而只需要使用同一个视频测试序列即可。需要指出的是，由于在HM中，整个帧内编码流程被封装在一个xCheckRDCostIntra函数中，而帧内预测模块和变换量化模块并不是严格上分离的，因此测试中统计的所谓“帧内预测模块”和“变换量化模块”，事实上是指，根据HEVC编码标准，分别完成这两部分功能的函数集合。具体包含的函数和对应功能，以及各自的运行时间见下图：

（补图-HM各模块内部时间占比）

基于以上对模块耗时情况以及各模块内部功能函数的分析，便可以进一步确定具体优化内容。

## 帧内预测模块优化

### 帧内预测模块分析

与HEVC标准一致，HM代码中帧内预测也分为亮度预测与色度预测，其流程大致相同，此处以亮度预测函数estIntraPredLumaQT为例，其主要流程如下：

（1）初始化各种参数。

（2）为了减少率失真优化次数，HEVC中默认使用帧内快速搜索算法，将分粗选和细选两个阶段进行。如果不使用快速搜索，将对所有帧内预测模式进行率失真优化。粗选阶段为：遍历35种帧内预测模式使用predIntraAng计算预测值，计算比较哈达玛失真，调用更新候选列表函数xUpdateCandList构建全率失真优化候选列表。全率失真优化候选列表长度numModesForFullRD由块宽度决定。注意在粗选阶段全部使用哈达玛变换函数xGetHADs计算失真，提高速度。细选阶段为：调用getIntraDirPredictor构建MPM列表，加入全率失真优化候选列表中。遍历全率失真优化候选列表，调用xRecurIntraCodingLumaQT进行变换量化重构，计算率失真代价，注意在这里该函数倒数第二个参数是bCheckFirst=true，表示会按照四叉树的方式继续向下划分。比较率失真代价，找到最优模式。

（3）对获得的最优模式再次调用xRecurIntraCodingLumaQT，此时倒数第二个参数设置为false，检测同一模式下，bCheckFirst为true和false的情况下，选出最优模式为最终的亮度帧内预测模式。

（4）收尾工作，记录信息：设置重建块、Cbf、上下文模型、总失真等。

### 哈达玛失真函数并行优化设计与实现

HM中在帧内预测粗选阶段，通过计算哈达玛失真（HAD）来进行初步筛选，这种失真的计算复杂度比率失真要小，因此能够提高初选的速度。其内部分别实现了对4×4矩阵和8×8矩阵的哈达玛（Hadamard）变换以及HAD值的计算。此处以4×4矩阵为例，说明并行优化算法的设计与实现。

算法的输入参数为一维原始图像数组piOrg与一维预测图像数组piCur。这两个图像数组的大小均为16，即包含16个像素值。其中，为了增加变量的复用性，减少运算次数，在哈达玛变换中使用了残差矩阵diff和中间变量矩阵m。首先进行循环，计算出每个像素的残差，即diff[k] = piOrg[k]-piCur[k]。接下来反复对diff和m矩阵进行一系列计算，具体如下，为方便起见，diff矩阵记为d：

最后累加diff矩阵中元素的绝对值abs(diff[i])，得到失真值。

从这一过程中可以发现，diff矩阵与m矩阵的计算的交替进行的，每一次矩阵计算需要依赖于前一次的计算结果，因此不能为每一个thread分配形式相同的计算任务，不便于直接转化为CUDA并行函数。为了能适合CUDA并行程序，需要解除中间矩阵diff与m的数据相关性。此处有两种可行方式，第一种方法（下文称为“代入展开法”）为：直接将之前的中间计算过程代入到最后依次diff矩阵，将其展开，这种方法可以将原本的4次矩阵运算变为一次，且可以省去中间变量m，从而消除数据依赖。经过这种展开之后diff矩阵。具体思路如下：



经过展开，d[0]=d[0]+d[1]+…+d[15]，d[1]、d[2]依次同理可得，因而对CUDA核函数，只需在一个block中开辟4×4个thread，分别计算一个d[i]的值即可，如下图所示：



第二种方法（下文称为“依赖整合法”）为：观察到每一步变换中，矩阵中的任意一项元素都依赖于前一矩阵的两项元素。因此可以分析具体的数据相关性，然后将具有依赖关系的元素放在同一CUDA线程中计算。数据依赖关系以及对应的线程规划如下：



这两种方法的理论并行度、计算复杂度和实际性能有所不同，具体的测试和分析将在第四章详细介绍。

### 模式预测值计算函数并行优化设计与实现

如前文所述，HM中帧内预测值计算函数主要为：首先判断预测模式为平面、直流还是角度预测模式，其中平面模式和直流模式各如前文所述进行处理，不再赘述。对于角度模式，以预测模式为21，预测块尺寸为4×4为例分析具体过程。首先生成预测参考像组集，上方参考像素为p[x][-1],x=-1,…,8，左侧参考像素为p[-1][y],y=1,…,8，记一维参考像素集为ref，其中ref[x]中x=-4,…,8。对于x大于等于0，ref取p[-1+x][-1],x=0,…,8；对于x小于0，ref取p[-1][-1+Round(32×(-1)/(-17))]=p[7][-1]。因此整个参考像素集的产生即如图所示：



接下来继续得到模式索引和权重参数并计算预测值，具体算法前文已说明。将预测值计算函数改写为CUDA核函数的主要目的是为了便于帧内预测模块的整体优化，其内部的优化并没有统一方法，主要为对规模较大的for循环进行改写，此处给出一处为例。串行代码为：

*……*

*for (Int y=0, deltaPos=intraPredAngle; y<height; y++, deltaPos+=intraPredAngle, pDsty+=dstStride)*

*{*

*const Int deltaInt = deltaPos >> 5;*

*const Int deltaFract = deltaPos & (32 - 1);*

*// Do linear filtering*

*const Pel \*pRM=refMain+deltaInt+1;*

*Int lastRefMainPel=\*pRM++;*

*for (Int x=0;x<width;pRM++,x++)*

*{*

*Int thisRefMainPel=\*pRM;*

*pDsty[x+0] = (Pel) ( ((32-deltaFract)\*lastRefMainPel*

*+ deltaFract\*thisRefMainPel +16) >> 5 );*

*lastRefMainPel=thisRefMainPel;*

*}*

*}*

*……*

其中y的循环控制变量height和x的循环控制变量width最大都为64，因此这是一个规模较大的函数，这一过程也可以较容易地改写为并行函数。即配置64个线程块，其中每个线程块包含64个线程，每一个线程块对应1次外层循环，其中地每一个线程对应1次内层循环的计算。代码实现为：

*int bid = blockIdx.x;*

*int tid = threadIdx.x;*

*for (Int y=0, deltaPos=intraPredAngle; y<height; y++, deltaPos+=intraPredAngle, pDsty+=dstStride)*

*{*

*if (y == bid)*

*{*

*……*

*for (Int x=0;x<width;pRM++,x++)*

*{*

*if (x == tid)*

*……*

*}*

*}*

*}*

*…..*

其余部分类似的循环也可以按照相似方法进行改写。

### 帧内预测模块整体并行优化设计与实现

除了对具体功能函数，即哈达玛失真函数和预测值计算函数进行优化之外，整个帧内预测的入口函数也有可以并行优化的空间。根据HM代码，其中候选集粗选的过程，是对于35中预测模式进行遍历进行的，对其中每一种模式都进行“预测值计算-哈达玛失真计算”的操作。根据前文工作，这两个主要的功能函数已经被优化为了CUDA核函数，因此可以考虑对全部35种预测模式进行并行优化。具体做法为：64×64的预测图像最大被划分为64个8×8的预测块，因此在CUDA中至多开辟64×35个block，其中每个block分别并行处理一个PU的一种预测模式，每个block内部再开辟最多8×8个thread，每个thread计算变换矩阵一个元素值，最后再串行累加计算出该模式的失真值。优化后整个“预测值计算-哈达玛失真计算”过程如下：



### 帧内预测优化算法理论并行度分析

在实际的CUDA并行优化中，由于涉及到GPU初始化、内存的申请与数据拷贝、CPU与GPU数据通信等等问题，因此实际的优化效果与预期存在有较大差距。为了对优化效果有更好地评估和分析，通过计算并行算法的最大理论并行度来作为实际优化结果的参考值。

对于哈达玛失真函数，为前文提出的两种方法分别分析并行度。当矩阵大小为4×4，“代入展开方法”开辟16个线程，理论并行度为16， “依赖整合方法”开辟8个线程，理论并行度为8；当矩阵大小为8×8，同理，两种方法理论并行度分别为64和32。

对于预测值计算函数，取代的最大循环为64×64，因此理论最大并行度为4096。

对于模块整体优化，串行算法在粗选阶段遍历35种预测模式，再对最大为64×64大小的块分割处理，计算每个8×8小块的预测值和哈达玛失真，而在并行优化后，至多同时生成35×64个线程块（blocks），且其中每个线程块又分别产生最大8×8个线程。因此整体的最大理论并行度为143360。可见整体的理论优化效果是较好的。本模块所有优化后的理论并行度汇总如下，其中PIA为帧内角度预测predIntraAng的缩写，此外，虽然前文对哈达玛失真计算给出了两种并行思路，但为简洁起见，下表中仅列出了“带入展开法”的理论并行度，“依赖整合法”的并行度为其1/2：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 函数名称 | 功能描述 | 线程块 | 线程数 | 理论并行度 |
| CUDA\_HAD4 | CUDA 4×4哈达玛失真 | 1 | 16 | 16 |
| CUDA\_HAD8 | CUDA 8×8哈达玛失真 | 1 | 64 | 64 |
| CUDA\_PIA | CUDA 预测值计算 | 1 | （64，64） | 4096 |
| CUDA\_Intra | CUDA 模块整体优化 | 35 | 1 | 35 |
| CUDA\_Intra  +CUDA\_HAD4 | CUDA 模块整体优化  +4×4哈达玛失真 | （35，64） | 16 | 35840 |
| CUDA\_Intra  +CUDA\_HAD8 | CUDA 模块整体优化  +8×8哈达玛失真 | （35，64） | 64 | 143360 |
| CUDA\_Intra  +CUDA\_PIA | CUDA 模块整体优化  +预测值计算 | （35） | （64，64） | 143360 |

## 变换量化模块优化

### 变换量化模块分析

由于HM中的量化函数涉及较多的条件判断，计算复杂度较高，因此在实际优化中仅针对一系列变换函数进行优化，但是在HM中变换与量化的过程在很多入口函数都较为模糊，因此仍然对整个变换量化模块进行分析。入口函数transformNxN的逻辑较为简洁，流程如下：

1. 对残差值进行处理PCM处理，利用周围临近参茶之预测当前残差。
2. 检验是否为trans-quant-bypass模式，若为该模式，则直接使用前一步预测出的残差值。
3. 检验是否为transform-skip模式，若为该模式，则调用跳过模式下的变换函数xTransformSkip。若为否则进行下一步。
4. 调用变换计算的入口函数xT，进行普通变换。
5. 调用量化计算的入口函数xQuant，进行量化。

### 快速DST变换函数并行优化设计与实现

根据前文简述的二维离散正弦变换矩阵，其元素值为正弦函数值，存在无理数，因此在HM中，为了便于存储和计算，使用存放整数矩阵和系数的方式来表示变换矩阵。需要指出的是，为了性能提升，在HM中并没有实现完整的DST变换，而是使用了一种近似的“快速DST变换算法”。

### 蝶形DCT变换函数并行优化设计与实现

在HM中，为了简化矩阵乘法运算，实现了一种蝶形DCT变换partialButterfly，充分利用了矩阵运算过程中间变量的重复性，大大降低了矩阵乘法计算的复杂度。针对不同大小的矩阵，HM中分别实现了4×4、8×8、16×16和32×32矩阵的。为了便于说明蝶形算法实现方法，此处以8×8矩阵的蝶形DCT变换为例，使用常规的矩阵乘法运算需要执行512次乘法运算和448次加法运算，而蝶形变换仅需要64次乘法运算和160次加法运算。HM中该算法实现思路如下：



针对这一算法，从图中可以看出，上下两半部分存在不同的分支情况，因而也就存在不同的计算复杂度的分支。因此在并行优化时需要考虑的主要问题为线程的设置和计算任务的划分。考虑运算过程的最后一步的蝶形加运算，下半部分的每个结果值为4个中间变量的求和。因此在数据依赖关系上，最后的每一个运算结果依赖于上一步的4个中间结果，而进一步向上分析，又依赖于2个原始矩阵中的元素值。为了解决数据相关性问题，应将公共数据部分写入通过\_\_shared\_\_关键字申请的CUDA共享内存，然后为每一线程分配计算量相当的计算任务，当需要使用公用变量时，可以直接从线程块共享的共享内存中读取，从而同时兼顾提高并行度和降低线程闲置与等待时间。

通过分析，上半部分（编号为0、1、2、3的原始块）和下半部分（编号为4、5、6、7的原始块）的分支情况，发现计算过程中上半部分的每个块进行3次加法和1次乘法计算，下半部分中每个块进行4次加法和1次乘法运算，上下半区的计算量存在不同，如果统一分配线程可能导致最后同步线程时发生等待和闲置。为避免线程闲置等待，可以将将整个运算过程分拆为3个CUDA核函数执行，保证每一线程计算量相等。因而此处也有两种并行方法。第一种方法（下文称为“单一核函数法”）计算过程的线程分配如下：



根据上图，编号为0~3的线程计算中共执行3次加法运算和1次乘法运算，编号4~7的线程计算中共执行4次加法运算和1次乘法运算。在第一次出现数据相关的整体蝶形加减法计算，每个线程执行的计算过程一致，都是1次加法，因而基本不存在线程等待。在第二次出现数据相关的上半部分的蝶形加减法计算，上半部分的每一线程都执行了2次加法，因而运行时间也基本一致。在第三次出现数据相关的下半部分的蝶形加法计算，下半部分每一个线程执行了1次乘法1次加法，因此线程间基本不存在等待。同时上半部分线程执行了2次加法一次乘法，而最后一次蝶形加法中，上半部分每个线程执行1次加法，下半部分执行3次加法，各线程计算任务基本一致。

第二种方法，也即是将整个变换过程拆分为多个CUDA核函数依次运行（下文称为“多阶段核函数法”）。具体为：核函数1开辟8个线程，分别完成第一步的蝶形变换加减法，核函数2开辟4个线程，每个线程分别完成对上半部分中间变量的计算，核函数3开辟8线程，完成对每个元素值和对应变换系数的乘法计算，核函数4完成对8个结果的计算。具体流程和线程配置情况如下：



### 变换量化优化算法理论并行度分析

对于量化变换模块，主要针对快速离散正弦函数fastForwardDST和4种不同尺寸的蝶形离散余弦变换函数partialButterfly进行了并行优化。

对于离散正弦变换函数，将其中每一个矩阵元素值的计算过程直接展开，矩阵中共有16个元素，理论并行度为16；对于4×4大小的蝶形离散余弦变换函数partialButterfly4（一下简写为PBF），由于蝶形计算只进行一次，所以不需分两种情况讨论，直接展开后理论并行度为16。针对8×8大小的蝶形变换函数PBF8，单一核函数法的理论并行度为64，多阶段核函数的理论并行度在各个阶段不同，最高值也为64。对于16×16的蝶形变换函数PBF16，为了简化核函数内部逻辑，仅对蝶形变换的第一步、乘法计算以及之后的部分蝶形计算过程进行了并行，单一核函数法配置为8线程块，每一线程块16线程，理论并行度为128，多阶段核函数法至多配置16个线程块，每一线程块包含16个线程，理论并行度为256。对于32×32的蝶形变换函数PBF32，与PBF16情况类似，单一和函数法理论并行度为512，多阶段核函数理论并行度最大为1024。汇总如下，表中\_SK为单核函数single-kernal缩写，MS为多阶段multi-stages缩写：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 函数名称 | 功能描述 | 线程块 | 线程数 | 理论并行度 |
| CUDA\_FFDST | CUDA快速离散正弦变换 | 1 | 16 | 16 |
| CUDA\_PBF4 | CUDA 4×4蝶形变换 | 1 | 8 | 8 |
| CUDA\_PBF8\_SK | CUDA 8×8单核函数蝶形变换 | 1 | 64 | 64 |
| CUDA\_PBF8\_MS | CUDA 8×8多核函数蝶形变换 | 1 | 64 | 64 |
| CUDA\_PBF16\_SK | CUDA 16×16单核函数蝶形变换 | 8 | 16 | 128 |
| CUDA\_PBF16\_MS | CUDA 16×16多核函数蝶形变换 | 16 | 16 | 256 |
| CUDA\_PBF32\_SK | CUDA 32×32单核函数蝶形变换 | 16 | 32 | 512 |
| CUDA\_PBF32\_MS | CUDA 32×32多核函数蝶形变换 | 32 | 32 | 1024 |

## CUDA函数模块整合

（内容不足时可以补充）

### HM静态链接库分析

### CUDA模块的添加与编译