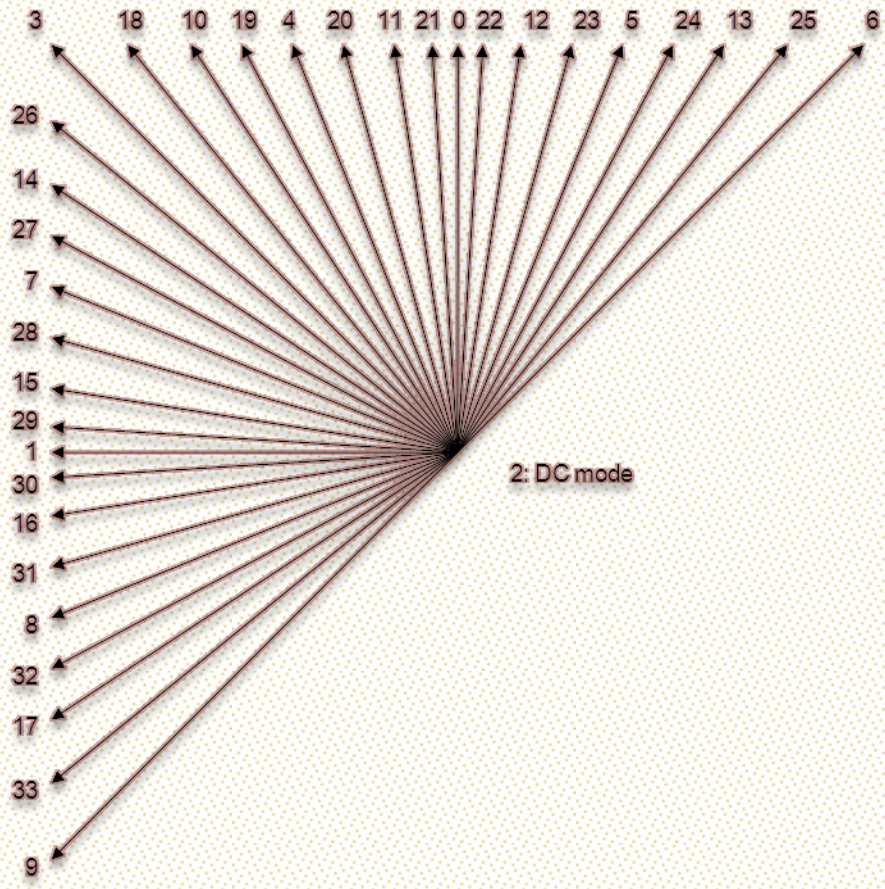


Luisiane Cristel dos Santos Fernandes

Implementação da Predição Intra HEVC em Processadores Multicore

Setembro de 2012

· U C ·

UNIVERSIDADE DE COIMBRA

FACULDADE DE CIÊNCIAS E TECNOLOGIA DA UNIVERSIDADE COIMBRA

Mestrado Em Engenharia Electrotécnica e de Computadores

Implementação da Predição Intra HEVC em Processadores Multicore

---

Luisiane Cristel Dos Santos Fernandes

Membros júri:

Presidente: Rita Cristina Girão Coelho da Silva

Orientador: Luís Alberto da Silva Cruz

Vogal: Marco Alexandre Cravo Gomes

Setembro de 2012

Aos meus pais e às minhas irmãs.

Resumo

Esta dissertação possui como foco principal o estudo do algoritmo do novo *standard* dos codificadores de vídeo, HEVC, tendo como centro da investigação a parte referente aos modelos de predição, mais precisamente o modelo de predição *Intra*.

Foram estudados e implementados métodos de paralelização a nível dos dados, por forma a analisar os benefícios dessa implementação aplicados à zona destinada ao cálculo da predição *Intra* presente no algoritmo do HEVC. Esses estudos destinam-se à investigação e análise da aplicação do codificador em processadores *Multicore*.

Tal como o mundo se deixou encantar pela "caixa mágica", a televisão, desde o ano 1923, em que as imagens nela transmitidas passaram por uma série de transformações e evoluções tais como a imagem a cores e chegando à recente imagem *High Definition*, também agora o desenvolvimento dos computadores e das novas plataformas móveis de comunicação se tornou o novo foco de interesse. Impulsionou uma mudança de paradigma no modo de visualização dos vídeos, sendo mais comum a visualização dos mesmos nos computadores ou *smartphones* do que nos tradicionais aparelhos televisivos.

Esta evolução acarreta custos associados com o aumento de informação que é necessário transmitir e processar. Neste contexto surge o conceito de codificador de vídeo, cujo principal objectivo é a obtenção de uma qualidade de vídeo cada vez melhor, trazendo ao mesmo tempo uma melhoria relativamente aos custos de transmissão, com a redução dos débitos associados.

O avanço dos codificadores e da compressão dos vídeos tornou-se assim um factor essencial no acompanhamento do desenvolvimento das novas plataformas e tecnologias de visualização vídeos digitais.

Palavras-chave:

Codificador de vídeo, *High Definition*, processadores Multicore, HEVC, Predição *Intra*.

Abstract

This work has focused primarily on the study of the new algorithm of standard video encoders, HEVC, with the center of the research regarding the prediction models, specifically the *Intra* Prediction model.

Methods of data-level parallelization were studied and implemented, in order to analyze the benefits of this implementation applied to the area for the calculation of *Intra* prediction algorithm in HEVC. These studies are aimed at research and analysis of the application of this encoder on multicore processors.

Just like the world has been enchanted by the "magic box", television, since the year 1923 where the images transmitted by it suffered profound changes and developments, such as appearance of the color image and getting to the recent High Definition image, also now the development of computers and new mobile platforms of communication become the new focus of interest. It spurred a paradigm shift in the means by which we watch vídeos, where it has become as common, viewing videos on computers or smartphones as in the traditional televisions.

This trend carries costs associated with the increase of information to be transmitted and processed. In this context, appears the concept of the video encoder. The main objective of the encoder is to obtain a better video quality, while bringing improvement on the transmission costs with the reduction in bandwidth associated.

The progress of video encoders and video compression thus became a key factor in the development of new display platforms and technologies of digital videos.

Keywords:

Video Encoder, High Definition, multicore processors, HEVC, *Intra* Prediction.

“Se a aparência e a essência das coisas coincidissem, a ciência seria desnecessária”,

Karl Marx

Índice

Lista de Figuras

Lista de Tabelas

Tabela de Abreviações

CPU	*Central Processing Units*
CU	*Coding Unit*
DLP	*Data Level Parallelism*
HEVC	*High Eficiency Video Coding*
ILP	*Instruction Level Parallelism*
JCT-VC	*Joint Collaborative Team on Video Coding*
LCU	*Largest Coding Unit*
MPEG	*Moving Picture Experts Group*
MISD	*Multiple Instruction Single Data*
MIMD	*Multiple Instruction Multiple Data*
PU	*Prediction Unit*
SATD	*Sum of Absolute Transform Differences*
SISD	*Single Instruction Single Data*
SIMD	*Single Instruction Multiple Data*
SMP	*Simetric Multiprocessing*
SMT	*Simetric Multithreading*
RAM	*Random Access Memory*
RD	*Rate – Distortion*
RDO	*Rate-Distortion Optimization*
RDOQ	*Rate Distortion Optimized Quantization*
RMD	*Rough Mode Decision*
TLP	*Thread Level Parallelism*
TU	*Transform Unit*
VCEG	*Video Coding Experts Group*

1. Introdução

Com o surgimento e evolução da tecnologia *High Definition* no mundo actual dos vídeos digitais, são cada vez mais visíveis os esforços para o desenvolvimento de um codificador de vídeo que possibilite ultrapassar as capacidades físicas impostas pela evolução tecnológica dos dispositivos de captura, transmissão e visualização de vídeos.

Foi assim que, em Janeiro de 2010, o JCT-VC (*Joint Collaborative Team on Video Coding*) deu início ao chamado *call for proposals* (CfPs), de modo a que fosse possível a implementação de um codificador que conseguisse atingir as mesmas qualidades de imagens obtidas pelo codificador anterior (H264/AVC), mas em metade do *bitrate* [1].

No entanto, apesar das vastas melhorias introduzidas pelo HEVC (*High Efficiency Video Coding),* este novo codificador tem como desvantagem um aumento significativo da complexidade da codificação, sendo assim necessário a investigação e desenvolvimento de algoritmos melhores de modo a reduzir essa complexidade [2].

Em [3] foram propostas e avaliadas várias estratégias de paralelização para o codificador HEVC. Uma das estratégias propostas foi o paralelismo a nível dos dados, consistindo assim na aplicação de um mesmo programa ou instrução a fracções de dados diferentes. Foi ainda constatado que o paralelismo ao nível dos dados num codificador de vídeo pode ser aplicado a diferentes níveis dos dados, tal como em tramas, LCUs *(Largest Coding Units*), CUs *(Coding Units)* e PUs *(Prediction Units).*

De modo a introduzir a implementação do paralelismo a nível dos dados nos blocos, foram estudados métodos da programação paralela em processadores *Multicore* no contexto de ambiente Windows, Win32.

Ao longo da análise do contexto de programação paralela, deparei-me com várias opções de implementação, entre as quais a utilização de bibliotecas e interfaces pré-definidas para suporte da programação paralela em Windows (OpenMP – *Open Multiprocessing* e TPL – *Task Parallel Library*), assim como a utilização de vários métodos de criação e manipulação de threads em c++ ambiente Win32.

Inicialmente foi alvo de testes da investigação para esta dissertação o método de manipulação de threads em que estas são constantemente criadas no início de cada ciclo, quando necessárias para cálculos e destruídas logo de seguida. No entanto, este método foi posto de lado, porque cedo se verificou um aumento significativo quer na utilização de

recursos como por exemplo a memória, quer no aumento do tempo de execução dispendido na criação e destruição das estruturas respectivas.

Para uma melhor eficiência de execução em *multithread* optou-se pela concentração dos esforços para a construção de um sistema do tipo *threadpool*, constituída por aquilo que é denominado como *worker threads*. Esta denominação deve-se ao facto das *threads* que constituem esse sistema permanecerem num estado constante de execução após serem criadas, sendo manipuladas apenas através de mecanismos de sincronização e de *queles* (filas) de trabalhos a serem executados.

1.1. Objectivos

O principal objectivo desta dissertação foi a implementação de um algoritmo que tirasse o melhor proveito do desempenho do paralelismo de dados em arquitecturas *Multicore*, para o cálculo de um algoritmo inicialmente implementado de forma sequencial.

Esse paralelismo de dados foi explorado ao nível dos blocos das PUs, visando assim uma redução do tempo gasto nos cálculos da predição *Intra* dos mesmos.

1.2. Conteúdo

A contextualização teórica desta dissertação é apresentada nos capítulos 2, 3 e 4, onde se faz uma introdução dos aspectos mais importantes no desenvolvimento da mesma. A contextualização prática é distribuída pelos capítulos 5 e 6.

No capítulo 2, é feita uma introdução aos conceitos da programação em ambientes *Multicore* onde se contempla também os aspectos mais relevantes para a implementação das *threads* em ambiente *Win32*.

No terceiro capítulo, faz-se uma breve descrição do codificador HEVC, começando por falar de una das principais características diferenciativas deste codificador que é a repartição das tramas em árvores de blocos, passando a descrever a estrutura do codificador actual. De seguida é feita uma breve introdução sobre uma das unidades que compõem essa estrutura, o modelo de predição.

O capítulo 4 centraliza-se na descrição do modelo de predição *Intra*, sendo este o foco em análise para a implementação prática. Inicialmente focando as características que são específicas da predição Intra do HEVC, passando depois para a descrição do algoritmo que já se encontra implementado no *Test Model HM4.0*, para a decisão do melhor modo de predição.

No quinto capítulo começa-se por fazer a descrição de como foi implementada a paralelização do algoritmo mencionado no capítulo anterior, expondo os passos adoptados para a concretização da implementação prática desta dissertação.

No capítulo 6 são apresentados os resultados dos tempos de cálculo obtidos, utilizando o código sequencial e o código com o algoritmo de paralelização.

No capítulo 7 são apresentadas as conclusões finais sobre o trabalho desenvolvido.

2. Programação para Processadores Multicores

2.1. Introdução

Segundo a previsão feita por Moore [4], o número de transístores disponíveis em cada circuito integrado (CPU) duplicaria em cada ano, o que implicaria um aumento similar da velocidade dos processadores, *clock speeds*.

No entanto, apesar dos avanços tecnológicos dos processadores terem seguido essa taxa de aumento pressuposta que ficou denominada como Lei de Moore, a evolução na indústria dos transístores encontrou-se perante barreiras tecnológicas que fizeram com que esta lei deixasse de ser válida para os processadores actuais.

Essas barreiras mencionadas anteriormente estão ligadas directamente às dificuldades em aumentar o *clock speed*. Isto devido às várias limitações físicas apresentadas pelos CPU's entre as quais se destacam o aumento do calor e a dificuldade na sua dissipação e o elevado consumo de energia.

De modo a ultrapassar essas barreiras impostas ao aumento do *clock speed* em arquitecturas *single-core,* surgiu a era *Multicore*. Os Processadores *Multicore* são constituídos por mais do que uma unidade de processamento, permitindo deste modo a evolução das capacidades de cálculo dos processadores com a introdução do processamento em paralelo.

Pela Taxonomia proposta e definida por Flynn [5], foi possível classificar as máquinas de arquitecturas paralelas de acordo com o número de fluxos das instruções e dos dados. Através dessa taxonomia as arquitecturas dos computadores foram assim divididas em quatro categorias de acordo com a caracterização desses fluxos:

- **Arquitectura SISD (Single Instruction Single Data)** que se traduz simplesmente na execução sequencial feita por um processador de um conjunto de instruções sobre um conjunto único de dados.
- **Arquitectura SIMD (Single Instruction Multiple Data)** caracterizada pela execução simultânea de uma mesma instrução sob vários conjuntos de dados. Sendo que se caracteriza pelo uso de processadores vectoriais ou matriciais para a execução dessas instruções em paralelo.

- **Arquitectura MISD (Multiple Instruction Single Data)**, onde são executadas várias instruções sob um mesmo conjunto de dados.
- **Arquitectura MIMD (Multiple Instruction Multiple Data),** caracterizada pela execução dos vários fluxos de instruções sob vários conjuntos de dados, envolvendo assim os SMP (Simetric Multiprocessing) e os SMT (Simetric Multithreading).

Assim, com a definição das máquinas de arquitecturas paralelas é permitido a análise e investigação do paralelismo em categorias distintas dessas arquitecturas.

Podemos assim definir a existência de três tipos peculiares de paralelismo[6]:

- Paralelismo a nível das instruções (ILP), caracterizado pela execução simultânea de múltiplas instruções de um mesmo programa.
- Paralelismo a nível dos dados (DLP), representado pela execução simultânea de uma mesma instrução em múltiplos dados.
- Paralelismo ao nível da tarefa (TLP), em que considerando como tarefa uma *thread,* tema esse explorado mais adiante no subcapítulo 2.2.2, este tipo de paralelismo é representado muitas vezes pela execução de uma única aplicação, sendo que essa execução é repartida por múltiplas *threads* em múltiplos processadores.

A implementação do paralelismo ao nível dos dados e ao nível da tarefa, muitas vezes podem ser confundidas. No entanto reside uma diferença fulcral entre elas, o paralelismo a nível da tarefa baseia-se no princípio de que a execução de uma aplicação é divida em etapas, cada uma executada por uma thread num processador diferente. Enquanto no paralelismo a nível dos dados, uma etapa da aplicação é executada pelas *threads* sobre blocos de dados diferentes, de forma independente.

2.2. Programação Paralela

No contexto das ciências de programação, um programa paralelo é definido como programa decomposto em várias linhas de execução (instruções),que são processadas de forma independente umas das outras.

O processamento paralelo de uma aplicação foi introduzido de modo a permitir reduzir o tempo de processamento dos dados, pois traduz-se principalmente na execução de várias partes de um mesmo programa por múltiplos processadores de forma concorrente entre si.

Uma das formas conhecidas de aplicação do paralelismo num programa, é aplicação do paralelismo a nível dos dados, que consiste na execução de uma mesma instrução a blocos

de dados diferentes. Isto é conseguido através da utilização dos processos e threads, pois estes permitem uma melhor manipulação da forma como as instruções são executadas pelo processador.

Segundo [6] uma das melhores soluções de adaptação de paralelismo de códigos sequenciais é a aplicação do *Data Parallel Threads*, que consiste nada mais do que a aplicação de múltiplas threads que executam o seu trabalho sobre um conjunto de dados fornecido.

Essa adaptação é a considerada como foco para a implementação prática desta dissertação. Nos subcapítulos que se seguem, é feita uma introdução sobre os aspectos mais relevantes para a implementação do *Data Parallel Threads* em ambiente Windows.

2.2.1. Processos [7]

Os processos podem ser interpretados como os programas em execução, sendo no entanto a sua estrutura mais ampla do que um simples programa. A constituição dos processos permite que o sistema operacional tenha controlo sobre os acessos ao processador e aos fios de execução.

Um processo normalmente contém não só o programa a ser executado mas também toda a informação necessária para a sua execução, entre as quais se podem destacar como o espaço de endereçamento da memória principal, a área ocupada em disco assim como também o tempo de processador.

2.2.2. Threads [7]

Na execução de um programa as *threads*, consideradas como um processo leve, são responsáveis por executar certas instruções de cálculo de forma autónoma dentro de um mesmo processo.

No início de cada programa, o processo associado a este, executa sempre a *main thread*, que por sua vez pode criar outras *threads* de forma a executar outras funções ligadas ao programa.

As *threads* podem ser confundidas como subprocessos no entanto possuem uma característica fundamental que as distingue dos processos: o facto de compartilharem o mesmo espaço de memória e as variáveis globais.

Uma das grandes vantagens da utilização de *threads* nos programas é a exploração máxima dos múltiplos processadores, permitindo que outras *threads* continuem o seu trabalho

mesmo quando uma *thread* esteja suspensa ou bloqueada à espera que um dos CPUs fique livre.

2.2.3. Problemas com utilização Threads

Na implementação de programas que exploram os benefícios da programação *multithread*, há que ter alguns cuidados.

Uma das situações a ter em conta é a utilização de memória partilhada entre as *threads*, pois pode ocorrer que duas threads tentem aceder à mesma zona de memória levando a conflitos no acesso à mesma.

Outra situação que também pode ocorrer são os chamados *starvation*, que ocorrem quando uma thread de prioridade alta ocupa todos os recursos da CPU, impedindo deste modo que outras threads de baixa prioridade prossigam com a sua execução.

A inversão de prioridade é um problema que ocorre quando uma tarefa de prioridade mais baixa esteja a bloquear um recurso necessário para a execução de uma tarefa prioridade mais alta.

Outro dos grandes problemas que podem surgir com a utilização de *threads* é a ocorrência dos chamados *deadlocks*. Esses acontecem quando duas ou mais *threads* estão à espera que os recursos necessários para continuarem a sua execução sejam libertados por outras *threads*. Isso leva a que as *threads* permanecem infinitamente à espera que um recurso seja libertado pelo outro.

2.2.4. Sincronização Threads [7] [8]

Num programa baseado em *multithreading*, há que recorrer muitas vezes à utilização de mecanismos de sincronização das *threads*.

Algumas das configurações possíveis de sincronização são a utilização de semáforos, de variáveis do tipo *mutex*, eventos, *critical sections* e de variáveis do tipo *interlocked*.

As ***Critical Sections*** são consideradas formas mais básicas de sincronização em *Win32*. Caracteriza-se por uma porção do código que contém um recurso compartilhado por várias *threads*. Em Win32, para a utilização das ***Critical Sections*** é necessário declarar uma variável do tipo CRITICAL_SECTION, para cada recurso a ser protegido. Essa variável é então inicializada utilizando a função InitializeCriticalSection() e apagada utilizando a função DeleteCriticalSection(). De cada vez que uma thread quer ter acesso ao recurso protegido pela ***critical section*** tem de pedir permissão utilizando a função EnterCriticalSection () e de seguida libertar o recurso usando a função LeaveCriticalSection().

As variáveis do tipo **mutex,** têm funcionalidades semelhantes às *critical sections*. São consideradas como uma *flag* utilizada para o acesso a um determinado recurso.

Podem ser criadas em Win32 utilizando a função CreateMutex() e "libertadas" utilizando a função ReleaseMutex(). Só a uma thread de cada vez é permitida fazer bloquear o mutex.

Os **Semáforos** são tidos como contadores, podendo ser incrementados e decrementados permitindo deste modo que um recurso seja acedido um número finito de vezes.

São criadas em Win32 utilizando a função CreateSemaphore() onde também é definido o número máximo que o contador pode atingir, e são libertados utilizando a função ReleaseSemaphore().

Os **Eventos** são considerados a forma mais flexível de sincronização de objectos em *Win32*. São criados utilizando a função CreateEvent() e controladas pelas funções SetEvent(), que coloca o evento num estado *"signaled"* e ResetEvent() que coloca o mesmo num estado *"nonsignaled"*.

As variáveis do tipo ***Interlocked***, constituem a forma mais simples no mecanismo de sincronização. Garantem que o acesso a uma determinada variável seja sequencial, ou seja, se mais que uma thread tentar aceder a uma mesma variável, elas entram numa espécie de fila de acesso. A variável é incrementada com a função InterlockedIncrement() e decrementada pela função InterlockedDecrement().

2.2.5. Threads em Win32 [7] [8]

Em Win32 a criação das *threads* é feita utilizando a função *_beginthreadex* ().

```
_beginthreadex(void * security,
                unsigned stacksize,
                unsigned (__stdall *funcStart) (void*),
                void *arglist,
                unsigned initFlags,
                unsigned *threadID).
```

Um dos parâmetros mais relevantes da função de criação de *threads* acima referida é o *funcStart* que representa a função a ser utilizada para execução da *thread*. Essa função possui uma característica peculiar para a execução num ambiente de programação orientada a

objectos, tem de ser obrigatoriamente uma função estática, o que leva a que seja necessária uma chamada a uma outra função, pertencente à classe onde se pretende fazer os cálculos. Apos o retorno do *funcStart* é terminado então automaticamente a execução da *thread* correspondente.

Outro parâmetro importante a mencionar é o *arglist*, pois este representa o ponteiro para os argumentos a serem passados como parâmetros para função em execução pela *thread*.

O parâmetro *initFlags* representa o estado da *thread* aquando da criação da mesma, pode ser o valor 0 indicando assim que a *thread* pode começar imediatamente a sua execução ou CREATE_SUSPEND que indica que será necessário utilizar outras funções de manipulação de *threads* para que seja possível iniciar a execução da mesma.

O parâmetro *threadID* corresponde ao ponteiro onde será guardado Id correspondente a *thread* criada.

Após a execução do *_beginthreadex* com êxito, esta retorna um *handle*, que serve de identificador da *thread* perante outras funções, permitindo assim a manipulação da *thread*.

Após a conclusão da execução da *thread* criada, é sempre necessário a utilização da função *CloseHandle* para fechar o *handle* criado anteriormente criado pela *thread* libertando assim o *kernel* do objecto da *thread*.

```
BOOLCloseHandle (HANDLE hObject);
```

Na função onde é feita a chamada para a criação da thread é necessário implementar um mecanismo de espera, de forma a possibilitar que esse fio de execução pare a sua execução à espera que a thread em execução retorne. Esse mecanismo é feito utilizando a chamada à função *WaitForSingleObject()* no caso de ser um único objecto ou então uma chamada à função *WaitForMultipleObjects()* para esperar por mais do que um objecto ao mesmo tempo.

```
DWORD WaitForSingleObject(
    HANDLE hHandle,
    DWORD dwMiliiseconds
    );
```

O parâmetro hHandle representa o *handle* do objecto a aguardar e dwMiliiseconds representa a quantidade de tempo permitida ao fio de execução ficar à espera, no caso do valor do parâmetro ser INFINITE significa que não é desejado que o fio de execução continue por ter havido uma expiração do tempo do retorno do objecto.

No caso de não obtermos o retorno da função de execução da thread, é necessário "forçar" a *thread* a fechar, de modo a libertar a memória por ela alocada. Isso é feito chamando a função *TerminateThread();*

```
BOOL WINAPI TerminateThread(
        HANDLE hThread,  // Handle da thread a terminar
        DWORD dwExitCode);
```

Para a utilização destas e outras funções de manipulação de *threads* em Win32 é necessário incluir no projecto a biblioteca process.h. Para a utilização das variáveis do tipo *handle* e *dword* é necessário incluir a biblioteca windows.h.

3. HEVC (*High Efficiency Video Coding*)

O HEVC emergiu em Janeiro de 2010 sendo concebido pelo que foi denominado o *Joint Collaborative Team on Video Coding* (JCT-VC) [1]. Esta equipa formada pelos maiores especialistas da área codificação de vídeo, tanto do grupo do MPEG (*Moving Picture Experts Group*) como do VCEG (*Video Coding Experts Group*), uniram-se com o objectivo de desenvolver a nova geração do codificador de vídeo.

Essa nova geração de codificadores apareceu com o objectivo principal de melhorar a eficiência da codificação em relação aos codificadores anteriores. Essa melhoria traduz-se na obtenção de uma imagem final com a mesma qualidade, mas tendo gasto metade da taxa de bits [1].

Foi assim que em 2010 o JCT-VC deu início ao chamado *call for proposals* (CfPs), de modo a iniciar o estudo de uma variedade de propostas de implementação de novas funcionalidades. Essas funcionalidades vão sendo assim avaliadas e introduzidas no standard do HEVC para alcançar uma eficiência na compressão significativa em relação ao precedente H264.

Estas funcionalidades implementadas (uma nova estrutura de divisão da trama em árvores recursivas, a utilização de maiores blocos de transformadas, a modificação da função de predição do movimento, etc.) introduziram por sua vez algumas complexidades no codificador HEVC [2].

Apesar de já se encontrar definida a maior parte da estrutura do codificador e as suas características fundamentais, o desenvolvimento do HEVC, ainda não foi concluído. Até à sua conclusão em 2013, ainda serão avaliadas e implementadas outras funcionalidades para maximizar a capacidade e eficiência de compressão e para reduzir ao máximo o nível de complexidade introduzidas [1].

Nos dois subcapítulos seguintes serão descritas algumas dessas novas funcionalidades implementadas no HEVC que o distingue do precedente H264.

3.1. Repartição de uma Trama

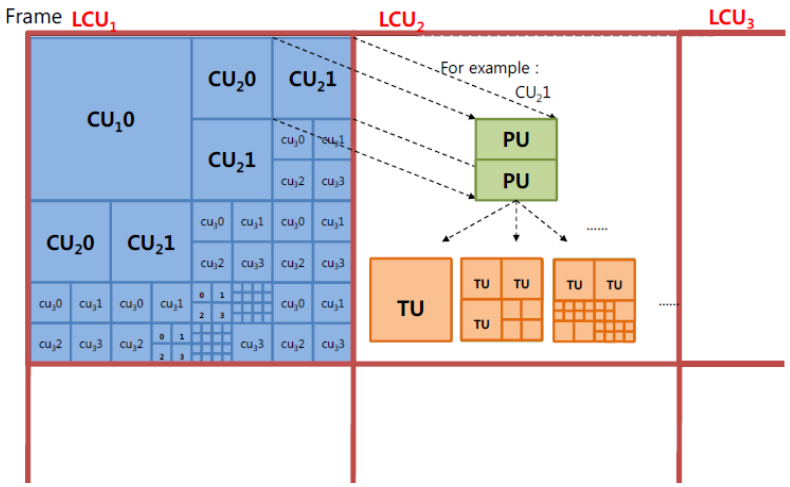
Uma das mais importantes inovações do HEVC é a introdução da divisão do vídeo de entrada do codificador em estruturas em árvores recursivas[1]. Sendo que um sinal de vídeo digital pode ser definido como uma sequência de imagens, dispostas de acordo com uma determinada taxa de amostragem temporal. A essas imagens, que compõem a sequência do sinal de vídeo, dão-se o nome de tramas.

No HEVC foi apresentada a proposta de dividir essas tramas em unidades de codificação de grande dimensão que contêm no seu interior várias sub-partições. Essas unidades são conhecidas como *Largest Coding Units* (LCUs).

As LCUs são blocos de uma trama de tamanho NxN que contêm tanto os dados correspondentes à luminância e à crominância dos pixéis análogos ao bloco. O tamanho máximo possível para o bloco LCU correspondente aos dados luminância é de 64x64.

Essas LCUs são posteriormente divididas em unidades de codificação, CUs (*Coding Units*) que correspondem às unidades mais básicas de codificação *Inter*/*Intra*, cujo tamanho pode ir até o tamanho da LCU correspondente ou ser dividida recursivamente em 4 Cus com mesma dimensão podendo então atingir o tamanho mínimo 8x8 [1]. Nessas unidades são feitos todos os processamentos que dizem respeito à predição e quantização dos pixéis.

Na figura a seguir é representado o particionamento de uma trama em LCUs e posteriormente nas estruturas derivadas desta.



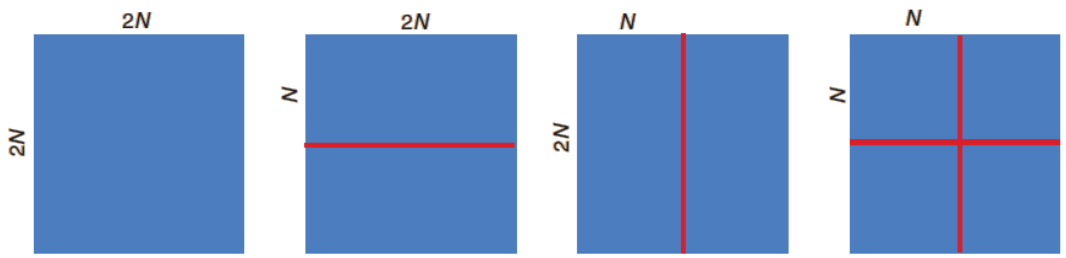
1 Repartição de uma trama nas suas estruturas inferiores [9].

Cada CU é dividida em unidades menores que correspondem às unidades básicas para o cálculo da predição, chamadas de unidades de predição, PUs (*Prediction Units*). As PUs

correspondem na realidade às unidades sobre as quais é feito todo o processamento das predições *Inter*/*Intra*.

Depois de serem escolhidos os modelos de predição (discutidos no subcapítulo 3.3), são então seleccionados os respectivos tamanhos em que cada CU será subdividida, sendo que essa divisão neste caso é não recursiva (ao contrário da divisão de LCUs em CUs) [1].

As CUs ao serem divididas em PUs *Intra* podem adoptar 4 configurações diferentes, que se encontram ilustradas na figura 2 tais como: 2Nx2N, sendo que essa configuração corresponde ao tamanho total da CU, 2NxN que corresponde a duas áreas cada uma com a largura total da CU e com metade da altura da CU; Nx2N, que corresponde a 2 áreas, cada uma com metade da largura da CU e com a altura total; ou então a configuração NxN que corresponde a quatro áreas, em que cada uma tem metade da altura e metade da largura da CU [1].

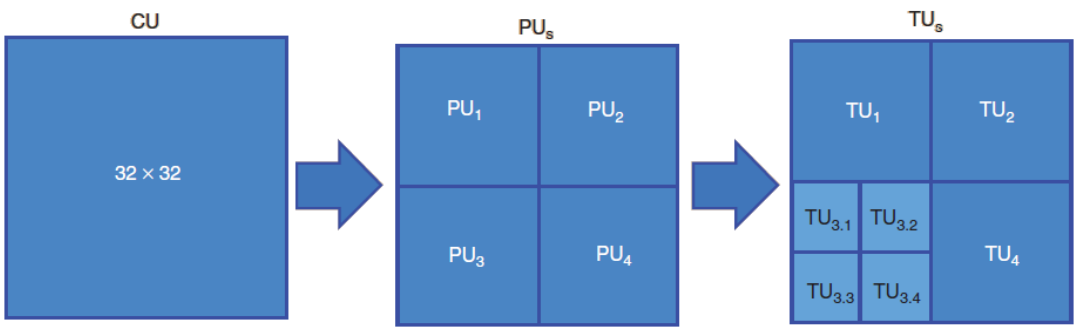


2 Divisão das CUs nos diversos tamanhos de PUs *Intra* [1]

Após esse processamento, são aplicadas então a transformação e quantização dos pixéis correspondentes à trama. Nesse ponto deparamo-nos então com as unidades de transformação, TUs (*Transform Units*) sobre as quais são aplicadas essas operações.

Cada PU é então dividida em unidades menores, TUs, onde depois é aplicado o processamento de dados relativamente à transformação e quantização [1].

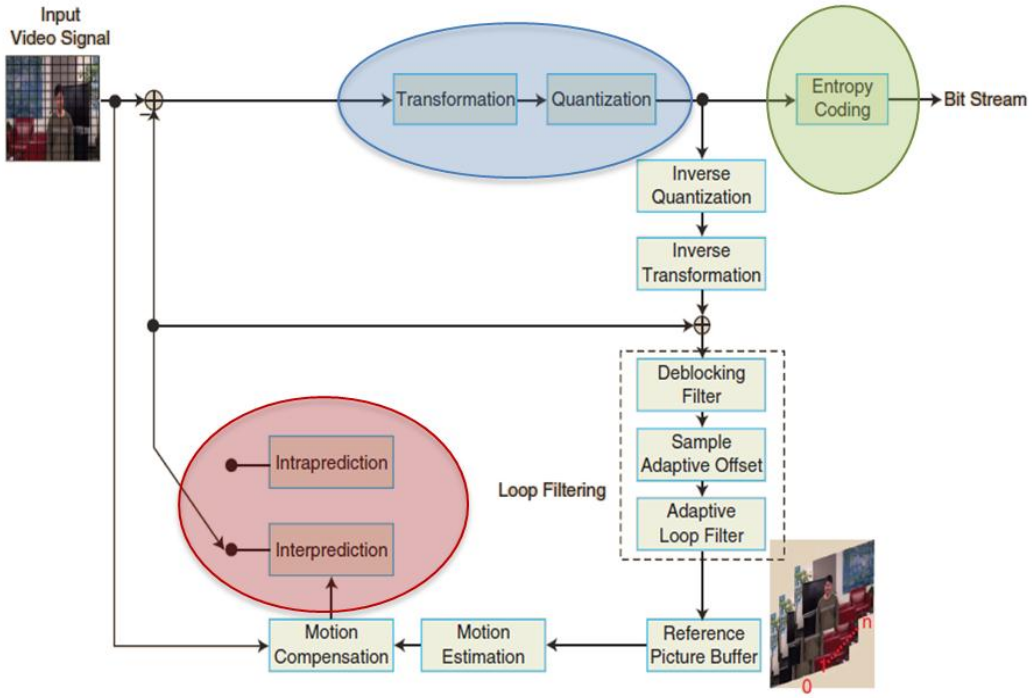
A interligação entre essas unidades é demonstrada na figura a seguir.



3 Exemplo da divisão de uma CU 32x32 em PUs e Tus [1].

3.2. Estrutura do HEVC Test Model (HM)

A arquitectura do codificador HEVC como se pode verificar na figura 4 assemelha-se muito à dos prévios *standards* dos codificadores vídeos existentes, tais como a estrutura do precedente H264/AVC.



4 Diagrama Blocos do Codificador de Vídeo HEVC [1].

A constituição de um codificador de vídeo desmembra-se em 3 unidades funcionais principais. Essas unidades são os modelos de predição, o modelo espacial e o codificador de entropia [10]. Como se pode observar pela figura 4 essas unidades também se encontram presentes no codificador de vídeo HEVC, sendo que o modelo espacial encontra-se assinalado a azul, os modelos de predição a vermelho e o codificador de entropia a verde.

O modelo de predição é constituído pelos modelos de predição *Intra* e predição *Inter*. Estes têm como entrada um sinal de vídeo que consiste numa sequência de tramas e como saída uma outra sequência de tramas, denominadas como sendo residuais. Este modelo tem como principal objectivo a redução da redundância de informação presente numa imagem.

As tramas residuais por sua vez consistem na entrada do modelo espacial e são constituídas principalmente pelo resíduo, este resultante da subtracção da predição feita no modelo anterior pela trama actual, assim como também por um conjunto de parâmetros contendo as características da predição feita.

O modelo espacial recebe como entrada a trama residual, onde será então aplicado o cálculo dos coeficientes de transformação assim como o de quantização.

Do codificador de entropia, tendo como parâmetros de entrada tanto os dados derivados do modelo de predição, como os coeficientes previamente calculados pelo modelo espacial, resulta então numa *bitstream* comprimida, com o propósito de ser efectuada a transmissão ou armazenamento do sinal de vídeo [10].

3.3. Modelos Predição do HEVC

Como foi mencionado no subcapítulo 3.2 os modelos de predição pertencem a uma das três unidades funcionais presentes no codificador HEVC, e tal como nos *standards* anteriores de codificação de vídeo, esse modelo é distribuído por duas configurações predição:

- Modelo Predição Temporal;
- Modelo Predição Espacial.

No modelo de predição temporal, a predição da trama actual é feita pelos valores obtidos de outras duas tramas vizinhas, denominadas como tramas de referência. Sendo que na sua forma mais simples de predição essas tramas de referência correspondem à trama anterior à actual e à posterior à mesma.

Este modelo de predição, também denominado como predição *Inter*, utiliza as correlações temporais das tramas de referência para fazer a estimação e compensação do movimento dos pixéis, explorando assim a redundância existente entre tramas vizinhas.

O modelo de predição espacial baseia-se na predição das amostras da trama actual através de amostras anteriormente calculadas para a mesma trama. Esse modelo é também conhecido como predição *Intra* e explora a redundância existente entre os pixéis de uma mesma trama, e será o foco de estudo detalhado no capítulo a seguir.

4. Predição Intra

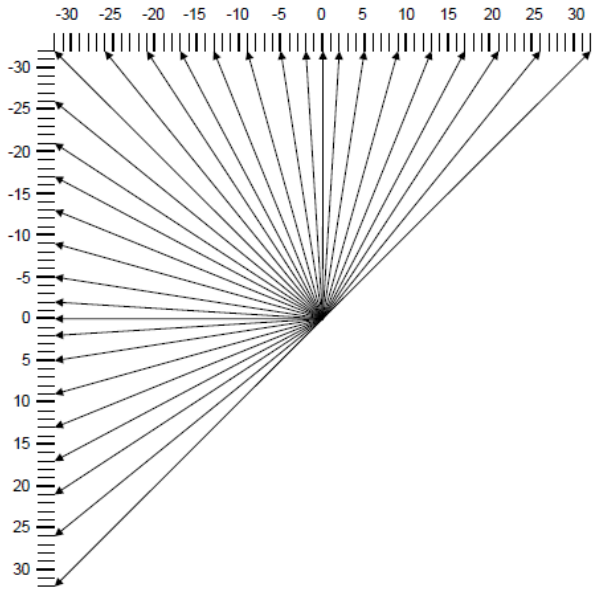
A predição *Intra* caracteriza-se principalmente pela utilização dos valores atribuídos aos pixéis previamente codificados nas PUs vizinhas à PU a ser predita de uma mesma trama.

Apoia-se no princípio de que as amostras dos blocos próximos (PUs) ao bloco actualmente a ser processado dentro de uma mesma trama, possuírem valores altamente correlacionados com o mesmo.

4.1. Predição Intra no HEVC (modelo testes HM4.0)

O aumento verificado na eficiência do codificador HEVC em relação aos seus precedentes deve-se em grande parte ao aumento do número de direcções de predição Intra, possibilitando assim uma reconstrução com maior exactidão de diferentes tipos de estruturas [11].

No modelo de testes HM4.0 são definidos um total de 35 modos para o cálculo da predição *Intra* para as componentes *Luma* de cada PU, que se repartem pelos 33 modos de direcção de predição angulares, apresentados na figura 5, mais os modos Planar e DC [12].



5 - As 33 direcções disponíveis na predição *Intra* [12].

No caso da predição angular vertical os ângulos/direcções são escolhidos de acordo com o deslocamento dos pixéis da linha inferior à PU e da linha de pixéis de referência acima da PU. No caso da predição horizontal esses ângulos são escolhidos de acordo com o

deslocamento dos pixéis da coluna mais à direita da PU e da linha de pixéis de referência à esquerda da PU [12].

O modo de predição *Intra* planar é utilizado principalmente para a predição das regiões de imagens mais atenuadas pois têm a capacidade de acompanhar as variações graduais do valor dos pixéis.

Apesar da predição *Intra* no HEVC suportar até 35 direcções para que seleccione a melhor direcção de predição, tem a desvantagem de tornar o codificador muito complexo do ponto vista computacional, caso todas essas direcções forem utilizadas no processo RDO (*Rate-Distortion Optimization*) para todas as PUs de uma trama [2].

Deste modo foram implementados algoritmos nos modelos de testes que permitem reduzir essa complexidade como por exemplo a limitação do número total de modos suportados por cada PU para os respectivos cálculos da predição.

Assim para cada PU em cálculo, o número total de modos de direcções de cálculo da predição possíveis é dependente do tamanho já previamente atribuído à PU na fase de repartição da trama em blocos.

A tabela seguinte demonstra como é feita essa relação entre o tamanho da PU e o número de direcções suportadas.

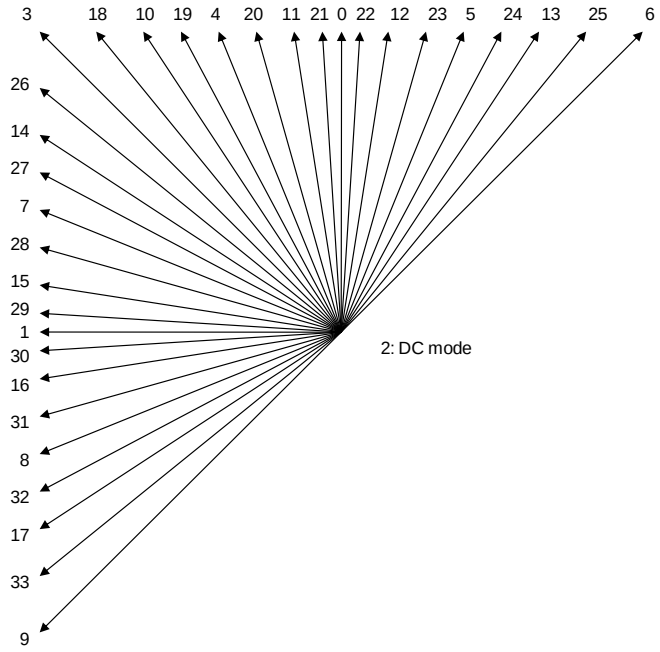
1 Numero direcções predição Intra de acordo com o tamanho da PU [13].

Tamanho da PU	Número modos Intra
4	18 (17 +1 planar)
8	35 (34 +1 planar)
16	35 (34 +1 planar)
32	35 (34 +1 planar)
64	4 (3 +1 planar)

Pela análise da tabela verifica-se que para os tamanhos dos blocos das PUs 4x4 e 64x64, o número de modos de direcção é reduzido em comparação com os outros. Isto deve-se ao facto de se ter verificado que, nos blocos 4x4, a pouca melhoria na qualidade de predição introduzida pela utilização de mais direcções, não compensava o aumento da taxa de bits por estes utilizados. Para o caso dos blocos 64x64 é utilizado um número muito pequeno de modos de direcção, pois estes blocos estão associados a áreas extremamente suaves e a utilização de direcções extras não era justificada [11].

Para estes tamanhos de PUs em que o número de modos permitidos são inferiores ao valor máximo de modos (35), as direcções dos primeiros N são dadas de acordo com o

mapeamento entre a direcção predição *Intra*, que se encontram ilustradas na figura 5, e o número do modo predição *Intra* especificada [14]. Esse mapeamento é feito de acordo com a figura seguinte:



6 Mapeamento entre as direcções de predição *Intra* e o modo de predição *Intra* [14].

Na prática no *software* HM4.0, após serem feitas as escolhas dos modos suportados para cada PU, é aplicado um método de escolha por forma a seleccionar o melhor modo de entre os candidatos para a predição *Intra*. Este método constituído por três fases encontra-se descrito em maior pormenor no capítulo 4.2.

Pela aplicação deste processo de decisão dividido em três fases, verifica-se assim uma optimização do código apresentando assim uma redução parcialmente da complexidade na codificação Intra. No entanto apesar de se verificar uma redução, a complexidade continua bastante alta devido à procura extensiva nos modos [2].

4.2. Descrição do *Intra Prediction Mode Decision* no HM4.0

Neste subcapítulo é feita uma breve descrição de como está organizada a Decisão do Modo de Predição *Intra* no HM 4.0, com principal foco nas funções responsáveis pelo cálculo da predição *Intra* da PU e pelo cálculo dos custos finais relativamente a cada direcção de predição *Intra* da PU. Essas funções são feitas através de um cálculo sequencial e é objectivo desta dissertação a paralelização destes cálculos.

Como mencionado no subcapítulo anterior, por forma a reduzir a complexidade a nível computacional do codificador, é adoptado no modelo de testes HM4.0, um processo de decisão dos modos de direcção dividida por três fases.

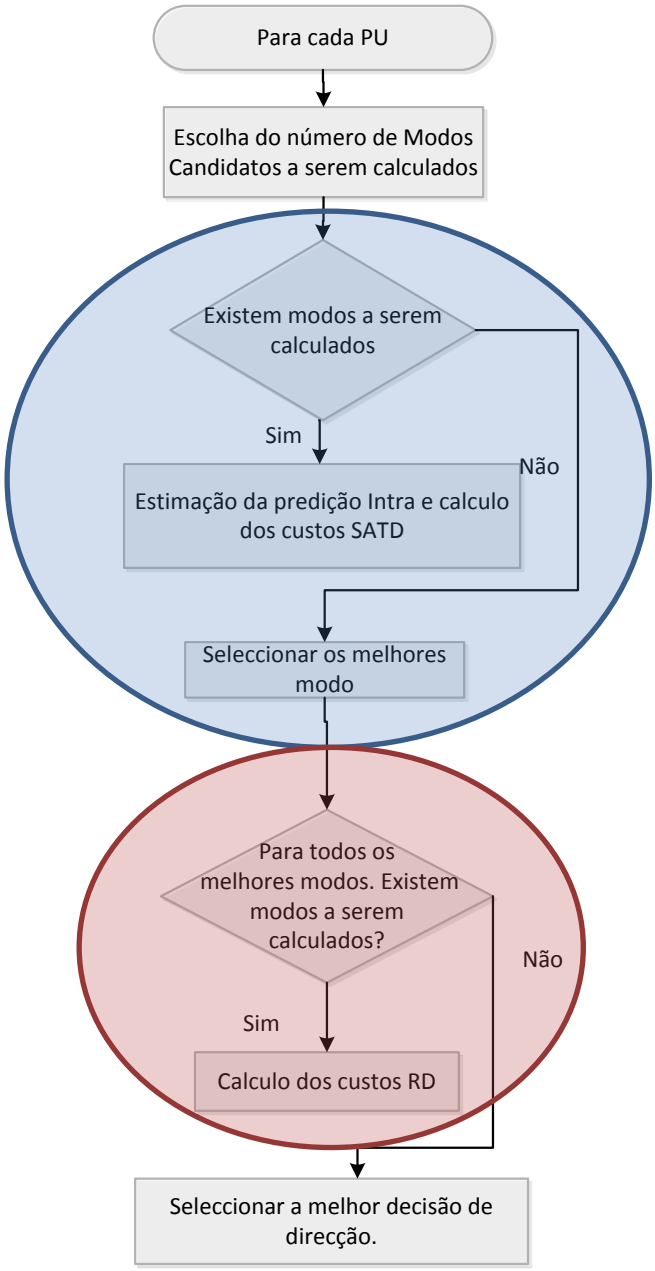
Na fase inicial é aplicada o processo denominado RMD (*Rough Mode Decision*) de modo a seleccionar os melhores candidatos de entre todos os modos possíveis definidos de acordo com a tabela 1. Este processo é baseado na predição residual SATD *(Sum of Absolute Transform Differences)* e nos bits estimados para os modos em cálculo.

Na segunda fase é aplicado então o cálculo computacional RDOQ (*Rate Distortion Optimized Quantization*) sobre o reduzido número de modos candidatos seleccionados previamente na primeira fase, de modo a obter a melhor predição.

Após esse cálculo, temos então a terceira fase onde são aplicadas transformadas recursivas, no modo ideal escolhido através das duas primeiras fases, para a codificação residual obter finalmente o melhor modo de predição [2].

Segundo [18], a procura da direcção óptima não é mais do que a procura da direcção na qual o bloco da PU contém uma variação menor.

Na figura 7 é possível identificar as duas fases que compõem o processo de decisão da melhor direcção de predição do modo, sendo que sombreadas a azul e a vermelho encontram-se representadas a primeira e segunda fase respectivamente.



7 Esquema representativo *Intra Prediction Mode Decision*

Pela análise do esquema podemos verificar que para a aplicação deste processo de decisão, inicialmente é feita a escolha do número de modos candidatos possíveis para predição de acordo com o tamanho de cada PU que compõem a CU em cálculo. Essa escolha do número de modos candidatos possíveis é feita de acordo com a tabela 1 acima referida.

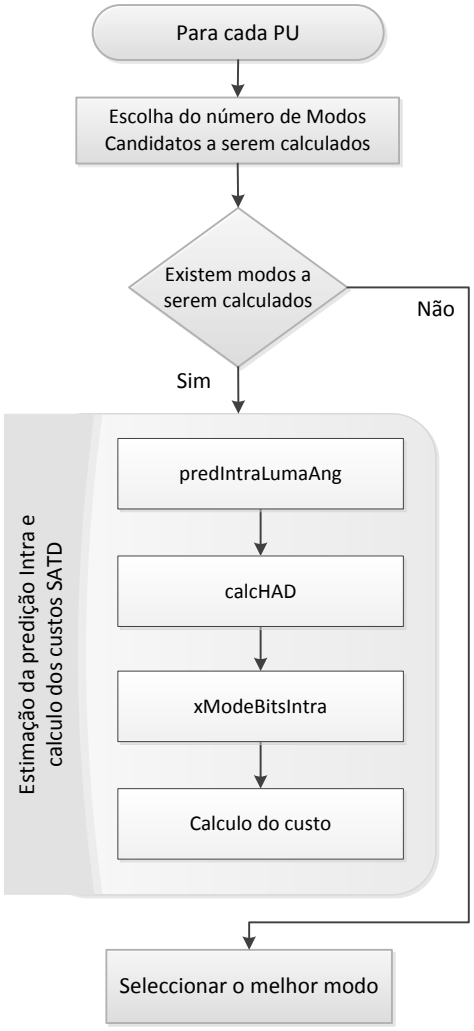
De seguida e a partir da lista de modos candidatos são executados então o cálculos correspondentes à primeira fase acima referida. Nesta fase é feito o cálculo da predição *Intra* para as direcções da lista dos modos candidatos e também o cálculo dos custos finais

associados a cada direcção. Com esses custos é feita a escolha do subconjunto dos melhores modos de predição, constituído então pelos modos que obtiveram menor valor de custo SATD *(Sum of Absolute Transform Differences)*[12].

Após a selecção desse subconjunto dos melhores modos de predição, entra-se então na segunda fase do processo, onde é aplicada a este subconjunto o cálculo dos custos RD (*Rate – Distortion)* de modo a seleccionar de entre todos os melhores modos aquele com menor custo RD, que se após passagem pela terceira fase obter-se-á então a melhor decisão de direcção para a predição *Intra*.

O foco principal desta tese encontra-se nas funções responsáveis pela selecção do subconjunto dos melhores modos de predição da tabela dos modos candidatos, ou seja a primeira fase do *Intra Prediction Mode Decision*.

Como se pode verificar na figura 7 o cálculo foi organizado através de uma estrutura do código sequencial.



8 Esquema representativo do cálculo da predição *Intra* e dos custos segundo cada modo de forma sequencial

Para cada modo disponível, é então calculado a predição *Intra* pela função `pred*Intra*LumaAng`, onde são calculadas a predição *Intra* Planar e a predição *Intra* Angular. O cálculo da predição angular é então feito baseado na direcção de predição indicada pelo índice do modo. Essa predição é então guardada no *buffer* correspondente ao pixéis preditos.

De seguida é utilizada a função `calcHAD` para o cálculo do *Hadamard Transform* onde é na realidade feito todo o cálculo dos valores SATD de cada PU para cada modo. Os valores SATD correspondem à soma absoluta da diferença das transformadas Hadamard entre os pixéis originais da PU e os pixéis preditos para a mesma PU [14].

A função `xModeBits*Intra*` tem por objectivo o cálculo do número de bits correspondentes a cada modo.

O custo total para cada modo é então obtido pela soma dos valores obtidos pelo SATD com os valores dos bits de cada modo.

Após o cálculo desse custo é feita então a decisão do melhor modo da predição *Intra* para cada PU.

5. Implementação Prática

Neste capítulo é exposto como foi implementada a programação paralela no procedimento dos cálculos da predição *Intra Luma* e da escolha do melhor modo de predição *Intra*.

5.1. Paralelização da selecção *Intra Prediction Mode.*

De modo a implementar a paralelização no cálculo dos custos de cada modo de predição para cada PU descrito no subcapítulo 4.2, foi adoptado um sistema tipo "*threadpoll*" em que as *threads* criadas executam o seu trabalho sob as funções responsáveis pela selecção do subconjunto dos melhores modos de predição.

O algoritmo desenvolvido visa a modificação dos cálculos executados numa estrutura sequencial, que se encontra descrito na figura 8, de forma a que estes sejam executados por múltiplas threads em vários CPUs.

Para a concretização desta implementação foram criadas:

- A estrutura *Othreads*, para armazenar as variáveis necessárias, essa estrutura será explicada ao pormenor no subcapítulo 5.2;
- A função *createall()*, onde serão implementadas a criação das *threads* e criação dos eventos necessários para manipulação das *threads.*
- A função *destroyall()*, onde serão implementados os mecanismos de destruição das *threads* e dos eventos.

As funções de criação e destruição das threads e eventos serão explicados no subcapítulo 5.3.

5.2. Criação da estrutura das variáveis

Para armazenamento dos dados a serem utilizados, tanto nos cálculos da predição *Intra* em cada direcção, como para o armazenamento dos custos com SATD correspondentes a cada modo, foi criada uma estrutura *ThreadStruct Othreads* com o tamanho máximo de direcções possíveis de cálculo.

Nessa estrutura de dados encontram-se todos as variáveis e *buffers* necessárias para os cálculos acima mencionados no capítulo 4.2. Essas variáveis e buffers são:

- `UInt tmodo`, que correspondente ao modo em cálculo
- `TComDataCU* tpcCU`, *buffer* que contém todos os dados relativos à CU em cálculo;
- `UInt tuiWidth`, largura da CU em cálculo;
- `UInt tuiHeight`, altura da CU em cálculo;

- `Bool tbAboveAvail`, variável booleana que indica se a linha de pixéis acima da CU em cálculo está disponível;
- `Bool tbLeftAvail`, variável booleana que indica se a linha de pixéis à esquerda da CU em cálculo está disponível;
- `Pel *tpiOrg`, ponteiro correspondente ao *buffer* dos pixéis de origem da PU em cálculo.
- `Pel *tpiPred`, ponteiro correspondente ao *buffer* dos pixéis preditos da PU em cálculo
- `UInt tuiPartPffSet`
- `UInt tuiDepth`, profundidade utilizada para a divisão da CU em PUs
- `UInt tuiInitTrDepth`
- `Double tcost`, variável onde será guardada os custos relativos a cada modo aplicado a PU em cálculo. Esta variável é inicializada com valor -1 de modo a indicar às *threads* que o trabalho sobre o índice da estrutura respectivo à variável ainda não foi efectuado nenhum tipo de cálculo.

5.3. Funções *create*all() e destroyall()

A função *createall()* foi implementada com o objectivo de criar todos os dados relativos a implementação *multithreading* no HEVC, entre elas a criação da estrutura de dados mencionada no subcapítulo 5.2, a criação de todas as *threads*, assim como a criação dos eventos e das *critical sections* utilizadas na sincronização das *threads* com a função *main*.

A função *destroyall()* esta função é chamada no fim da execução de todo o programa e foi implementada com a finalidade de destruir todas as threads e variáveis criadas e alocadas para a implementação do paralelismo.

Ambas as funções utilizam as funções de criação e manipulação de *threads* em Win32, já mencionadas e descritas no subcapítulo 2.2.5.

5.4. Descrição da implementação

Inicialmente foi criada um número fixo de *threads* que foi definido na biblioteca Typedef.h.

Essas *threads* são todas criadas no início quando se excuta a criação de todos os outros dados que dizem respeito ao programa principal sendo assim também destruídas quando todos os dados correspondentes ao programa são destruídos. Foi assim adoptado um sistema tipo

"*threadpoll*" que consiste principalmente na criação das *threads* no início do programa em execução e de deixa-las "trabalhar" ao longo do tempo de vida do programa principal.

Como mencionado no subcapítulo 2.2.4 é necessário a utilização de mecanismos de sincronização dos acessos múltiplos a uma mesma variável feita por *thread.*

Um desses mecanismos adoptados na implementação foi a utilização de *critical sections* de modo a permitir que só uma *thread* de cada vez acesse uma variável compartilhada.

Foram assim criadas duas *criticals sections*, inicializadas na função `createall()`, e utilizadas em variáveis cujo acesso deve ser restrito a uma *thread* de cada vez.

Outro mecanismo adoptado foi o uso de eventos para indicação quer de trabalho presente a ser executado, *array* de eventos *Awake* com tamanho máximo do número de threads criadas, quer para a indicação de que todas as tarefas foram completadas pelas *threads*, o evento *doneEvent*.

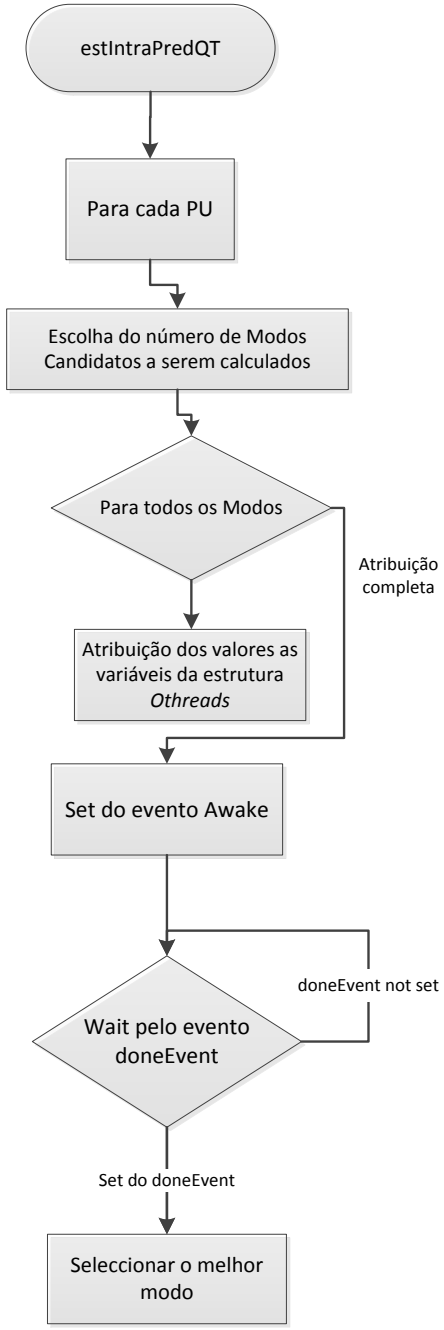
A implementação do paralelismo foi configurada por 2 etapas interligadas entre si. Nos diagramas de blocos apresentados na figura 9 e 10, é possível verificar as modificações implementadas no esquema da figura 7 por forma a adoptar essa nova configuração referente ao paralelismo.

Pela análise da figura 9, corresponde a etapa inicial, podemos verificar que no início dos cálculos para o *Intra Prediction Mode Decision*, após ser feita a escolha dos modos candidatos para predição de acordo com o tamanho de cada PU disponível na CU, foi atribuído para cada índice da estrutura *Othreads* os valores necessários para os cálculos da predição *Intra* e dos custos, correspondentes a cada um desses modos candidatos.

Quando todos esses valores estiverem atribuídos, é assinalada todas as *flags Awake*, que indicam às *threads*, previamente inicializadas que podem começar execução dos cálculos sob a estrutura *Othreads.* Este evento tem a funcionalidade de indicar à respectiva thread, que se encontra em execução na função *ThreadPredFunc*, à espera desse sinal, que pode dar início à execução dos cálculos respectivos.

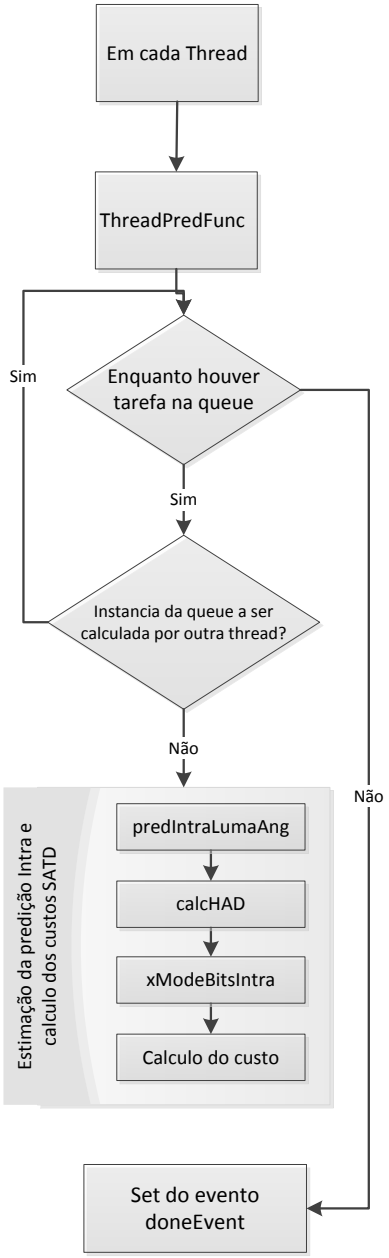
A função *estIntraPredQT* após o *set* de cada índice do *array* de eventos *Awake*, fica a espera que a *flag* do evento *doneEvent*, seja assinalada a indicar que todo o trabalho disponível na *queue* foi concluído, essa espera é realizada através da função `WaitForSingleObject`.

Após a execução de todas as tarefas pelas *threads*, é possível então constituir o subconjunto dos modos de predição candidatos com os valores dos custos correspondentes a cada modo de predição guardados na variável *tcost* da estrutura *Othreads.*



9 Esquema da primeira parte implementação *Multithread* da predição *Intra* e dos custos para cada PU.

Cada *thread* ao receber o sinal enviado pelo evento *Awake* vai prosseguir para execução dos cálculos, isso é retratado no esquema da figura 10, que corresponde à segunda etapa da paralelização.



10 Esquema da segunda parte implementação *Multithread* da predição *Intra* e dos custos para cada PU.

Segundo o ponto de vista da *threadpoll* cada índice da estrutura *Othreads* corresponde a uma tarefa a ser executada, sendo a estrutura toda considerada como uma *queue* de tarefas.

A *thread* em execução, começa então por fazer uma pesquisa sob a estrutura de dados de modo a identificar quais os custos que ainda não foram calculados, ou seja quais as tarefas da *queue* que ainda se encontra por executar. Assim que começa a execução dos cálculos sobre uma dessas tarefas atribui o valor zero à variável *tcost* respectiva, de modo a indicar a qualquer outra *thread* que esta tarefa está a ser calculada.

Enquanto a *flag Awake* estiver assinalada, a *thread* continua a sua execução após o término de uma tarefa, procurando sob a estrutura *Othreads* a próxima tarefa da *queue* que ainda não esteja completa ou indicada como a ser processada.

A execução de cada *thread* é feita até todas as tarefas estarem completas, sendo que, logo que ela termine a sua execução, a thread decrementa a variável *itemsRemaining*.

Essa variável é inicializada com o tamanho máximo de *threads* possíveis e é decrementada cada vez que uma thread termine o seu trabalho. Tem o propósito de fazer o *set* do evento *doneEvent*, quando o valor dessa variável atingir o valor 0.

Após o *set* do evento *doneEvent* a função *estIntraPredQT* que se encontra à espera da sinalização deste evento, continua então a sua execução e escolha dos melhores modos sob os valores dos custos calculados e guardados nas variáveis *tcost*.

6. Resultados experimentais

A implementação do algoritmo descrito para paralelização da selecção *Intra Prediction Mode* desta dissertação foi realizada no *software Test Model HM4.0*.

Para a avaliação das diferenças temporais entre a aplicação do algoritmo de cálculo dos custos em modo sequencial e em modo *multithread* foram utilizadas 4 sequências de vídeos com diferentes resoluções.

A avaliação foi efectuada num computador com processador Intel® Core™ i5 CPU M460 @ 2.53Ghz com 4 GB de memória RAM instalada e com sistema operativo Windows 7.

Após efectuar a recolha dos tempos de cálculo dos custos para cada PU, tanto no algoritmo sequencial como no algoritmo paralelo para 4 threads, foi efectuada a relação desses valores através do cálculo do *Speedup*.

Pela Lei de Amdahl's, o *Speedup* corresponde neste caso ao factor de aceleração de um processo executado em um único processador e a sua execução em processadores [15].

Equação 1

$$S = \frac{Ts}{Tp} = \frac{Tempo\ de\ execução\ em\ série}{Tempo\ de\ execução\ em\ paralelo}$$

Para uma máquina paralela com n processadores, o *speedup* ideal seria n (*speedup* linear) [15].

Nas tabelas seguintes são apresentados os resultados de alguns tempos de execução para primeiras 5 PUs, de cada trama do vídeo. Essas 5 PUs foram escolhidas unicamente para servir de exemplo de comparação do tempo de cálculo em algumas PUs, sendo que essa comparação poderia ser feita em qualquer outro número de PUs.

Apresenta-se na tabela 2 os valores referentes ao código em modo sequencial e os valores com a implementação paralela, assim como o speedup em cada uma dessas PUs.

2 Resultados da comparação dos tempos de execução de 5 PUs com aplicação do algoritmo

Sequência	Número da PU	Ts	Tp	SpeedUP PU
BasketballPass 416x240	1	0,00009	0,000073	1,23287
	2	0,000189	0,000212	0,89150
	3	0,000057	0,000056	1,01785
	4	0,000021	0,000047	0,44680
	5	0,000007	0,000032	0,21875

Sequência	Número da PU	Ts	Tp	SpeedUP PU
PartyScene 832x480	1	0,000177	0,000064	2,76562
	2	0,000189	0,00012	1,57500
	3	0,000058	0,000057	1,01754
	4	0,000022	0,000032	0,68750
	5	0,000008	0,000016	0,50000

Sequência	Número da PU	Ts	Tp	SpeedUP PU
BlowingBubbles 416x240	1	0,00009	0,000103	0,87378
	2	0,000191	0,000159	1,20125
	3	0,000058	0,000092	0,63043
	4	0,000021	0,000074	0,28378
	5	0,000007	0,000058	0,12068

Sequência	Número da PU	Ts	Tp	SpeedUP PU
Kimono 1920x1080	1	0,000105	0,000104	1,00961
	2	0,000188	0,000027	6,96296
	3	0,000057	0,000094	0,60638
	4	0,000021	0,0001	0,21000
	5	0,000007	0,000058	0,12068

A verde encontram-se destacadas as PUs, em que o speedup obtido é positivo, observando-se assim uma melhoria significativa em alguns casos nos tempos de cálculo, principalmente em vídeos de maior resolução.

Para além dos resultados acima apresentados foi ainda efectuado o cálculo do *SpeedUp* do somatório dos tempos de execução para todas as PUs existentes numa trama. Sendo assim a equação do *SpeedUp* é então traduzida de forma a contemplar os tempos de execução total da trama:

Equação 2

$$S_{Frame} = \frac{\sum_{PU} Ts}{\sum_{PU} Tp}$$

Equação 3

$$S_{Frame} = \frac{Somatorio\ do\ Tempo\ de\ execução\ em\ série\ de\ todas\ as\ PU}{Somatorio\ do\ Tempo\ de\ execução\ em\ paralelo\ de\ todas\ as\ PU}$$

Os resultados dos cálculos do *SpeedUp* total da trama, efectuados nas sequências de imagens acima referidas, encontram-se apresentados na tabela seguinte.

3 Resultados do *SpeedUp* obtidos para todas as PUS de 1 frame

Sequência	**Resolução (pixeis)**	**SpeedUp**
Kimono	1920x1080	0,59337
PartyScene	832x480	0,57725
BlowingBubbles	416x240	0,67555
BasketballPass	416x240	0,53993

Os tempos acima referidos foram obtidos somente na parte do código onde foi feita a implementação, de modo que esse tempo em nada é influenciado por outras partes do código. Ou seja, no caso do tempo sequencial e tendo como base explicativa do código a figura 8, o tempo inicial foi retirado após a escolha do número de modos candidatos e o tempo final foi retirado após todos os cálculos da estimação da predição e dos custos serem executados.

No caso do tempo paralelo tendo como base explicativa do código os esquemas das figuras 9 e 10, o tempo inicial foi retirado após o *set* de todas as *flags Awake* e o tempo final foi retirado após a função *estIntraPredQT* ter obtido a sinalização da *flag* do evento *doneEvent*.

7. Conclusão

O trabalho elaborado nesta dissertação abrangeu um estudo sobre a estrutura da nova geração de codificadores de vídeo, assim como também um estudo aprofundado sobre aplicação de algoritmos *multithreading*.

O objectivo desta dissertação foi a análise da implementação de uma das funções mais importantes do modelo do codificador por forma a ser executado num processador *multicore*.

Verificou-se que apesar desta implementação demonstrar melhorias significativas a nível de consumo de recursos de memória e de CPU, o tempo gasto pela execução total das tarefas em modo *multithread* não foi inferior ao tempo da implementação sequencial.

Assim, com o presente trabalho, pretendi, acima de tudo, dar início à investigação e desenvolvimento da implementação de uma estratégia de paralelização para função que diz respeito à decisão do modo da Predição *Intra* HEVC. Através dos testes realizados, foi possível averiguar que em certas PUs o tempo gasto pelas *threads* para completarem as tarefas é inferior ao tempo feito pelo código sequencial como se pode verificar nas tabelas 2 e 3 demonstradas no capítulo 6. No entanto verificou-se também que o somatório calculado de todos os tempos gastos em todas as PUs de uma trama demostrou que a eficiência da implementação do cálculo dos custos em paralelo é negativa.

Uma das principais razões para que esse tempo de cálculo seja superior, deve-se à implementação de mecanismos de sincronização, que no entanto foram necessárias para a correcta implementação e funcionamento das *worker threads* no código HEVC.

Nos trabalhos futuros considero que os estudos deveriam incidir sobre outras técnicas de paralelização para implementação no HEVC, e investigação de uma melhor técnica de sincronização de *threads* de modo a melhorar significativamente o *SpeedUP* dos cálculos para todas as PUs.

Referencias Bibliográficas

[1] M. T. Pourazad, C. Doutre, M. Azimi e P. Nasiopoulos, “HEVC: The New Gold Standard for Video Compression,” *IEEE CONSUMER ELECTRONICS MAGAZINE,* Julho 2012.

[2] Z. M. Hao Zhang, *Fast Intra Prediction for High Eifficiency Video Coding.*

[3] M. A. Mesa, C. C. Chi, T. Schierl e B. Juurlink, *Evaluation of Parallelization Strategies for the Emerging HEVC Standard*

[4] G. E. Moore, "Cramming More Components Onto Integrated Circuits," *Solid-State Circuits Newsletter, IEEE,* pp. 33 -35, 2006.

[5] M. J. Flynn, “Some Computer Organizations and Their Effectiveness,” *Computers, IEEE Transactions on,* Vols. %1 de %2C-21, n.º 9, pp. 948-960, 1972.

[6] AMD, Advanced Micro Devices, *Designing & Optimizing Software for N Cores,* 2006.

[7] J. Beveridge e R. Wiener, Multithreading Applications in Win32, The Complete Guide to Threads, Addison Wesley Longman, Inc., 1997.

[8] Microsoft, “MSDN,” [Online]. Available: http://msdn.microsoft.com/en-us/library/hh156548. [Acedido em 08 2012].

[9] D. Sim, *High Efficiency Video Coding(HEVC),* Kwangwoon University, 2011.

[10] I. E. Richardson, THE H.264 ADVANCED VIDEO COMPRESSION, John Wiley and Sons, Ltd, 2010.

[11] K. U. Jani Lainema, “Angular intra prediction in High Efficiency Video Coding (HEVC),” em *Multimedia Signal Processing (MMSP), 2011 IEEE 13th International Workshop*, 2011.

[12] L. Zhao, L. Zhang, S. Ma e D. Zhao, “Fast mode decision algorithm for intra prediction in HEVC,” em *Visual Communications and Image Processing (VCIP), 2011 IEEE*, 2011.

[13] J. Y. H. L. a. B. J. Jaehwan Kim, “Fast intra mode decision of HEVC based on hierarchical structure,” em *Information, Communications and Signal Processing (ICICS) 2011 8th International Conference on*, 2011.

[14] K. McCann, “HM4: High Efficiency Video Coding (HEVC) Test Model 4 Encoder Description,” Joint Collaborative Team on Video Coding, 2011.

[15] P. L. Coelho, *SISTEMAS DISTRIBUIDOS Programação Paralela.*

[16] R. H. CARVER e K.-C. TAI, MODERN MULTITHREADING, New Jersey: John Wiley & Sons, 2006.

[17] Y. Liu, “H265.net Witness the development of H.265,” 01 12 2010. [Online]. Available: http://www.h265.net/2010/12/analysis-of-coding-tools-in-hevc-test-model-hm-intra-prediction.html. [Acedido em 06 09 2012].

[18] H. M. Y. C. Wei Jiang, “Gradient based fast mode decision algorithm for intra prediction in HEVC,” em *Consumer Electronics, Communications and Networks (CECNet), 2012 2nd International Conference on*, 2012.