Design of performance optimized transform and quantization computation blocks for video compression in heterogeneous high performance computing systems.

Čobrnić, Mate

Doctoral thesis / Disertacija

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:296764

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-07-18**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

University of Zagreb
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Mate Čobrnić

DESIGN OF PERFORMANCE OPTIMIZED TRANSFORM AND QUANTIZATION COMPUTATION BLOCKS FOR VIDEO COMPRESSION IN HETEROGENEOUS HIGH PERFORMANCE COMPUTING SYSTEMS

DOCTORAL THESIS

Zagreb, 2020

University of Zagreb
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Mate Čobrnić

DESIGN OF PERFORMANCE OPTIMIZED TRANSFORM AND QUANTIZATION COMPUTATION BLOCKS FOR VIDEO COMPRESSION IN HETEROGENEOUS HIGH PERFORMANCE COMPUTING SYSTEMS

DOCTORAL THESIS

Supervisor:
Professor Mario Kovač, PhD

Zagreb, 2020

Sveučilište u Zagrebu
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Mate Čobrnić

PROJEKTIRANJE TRANSFORMACIJSKIH I KVANTIZACIJSKIH RAČUNSKIH BLOKOVA ZA VIDEOKOMPRESIJU OPTIMIRANIH ZA UČINKOVITO IZVOĐENJE NA HETEROGENIM VIŠEPROCESORSKIM RAČUNALIMA VISOKIH PERFORMANCI

DOKTORSKI RAD

Mentor:
Prof.dr.sc. Mario Kovač

Zagreb, 2020.

Doctoral thesis was made at the University of Zagreb,

Faculty of Electrical Engineering and Computing,

Department of Control and Computer Engineering

Supervisor:

Professor Mario Kovač, PhD

Doctoral thesis contains: 107 pages

Doctoral thesis number: _____

ABOUT THE SUPERVISOR:

Prof. dr. sc. Mario Kovač is a full professor at the Faculty of Electrical Engineering and Computing (FER), University of Zagreb, Croatia. In 1990 and 1991 he received a VLSI and Computer Architecture Scholarship at the University of South Florida, and he subsequently received the Fulbright Award in 1993. He holds several patents including US patents in multimedia systems and architecture domains. In 2008, the Croatian President awarded him with the Medal of Honour "Order of Danica Hrvatska with the image of Ruđer Bošković" for special merit in science. Professor Kovač served as Head of the Dept. of Control and Computer Engineering and Vice Dean for Business Development at FER. He was a member of the supervisory boards of: CARNet, the Croatian Institute of Technology and BICRO - Business Innovation Centre of Croatia. He currently holds several positions: Chief Communications Officer (CCO) at the European Processor Initiative; Expert member of Governing Board as well as Research and Innovation Advisory Group Observer Member at EuroHPC Joint Undertaking; Director HPC Architectures and Applications Research Centre at FER. He is a senior member of the IEEE Computer Society.

O MENTORU:

Prof. dr. sc. Mario Kovač je redovni profesor na Fakultetu elektrotehnike i računarstva (FER), Sveučilište u Zagrebu. 1990. i 1991. dobio je stipendiju za istraživanja VLSI računalnih arhitektura na University of South Florida, Tampa, SAD a potom je 1993. dobio nagradu Fulbright. Autor je nekoliko patenata, uključujući američke patente u području multimedijskih arhitektura i sustava. 2008. godine odlikovan je medaljom "Orden Danice Hrvatske s likom Ruđera Boškovića" za posebne zasluge u znanosti. Profesor Kovač obnašao je dužnost predstojnika Zavoda za automatiku i računalno inženjerstvo i prodekana za poslovanje na FER-u. Bio je član nadzornih odbora: CARNet-a, Hrvatskog tehnološkog instituta i Poslovno-inovacijskog centra Hrvatske-BICRO. Trenutno obnaša nekoliko pozicija: glavni direktor za komunikacije (CCO) Europskog projekta „European Processor Initiative“; Stručni član upravnog odbora, kao i savjetodavne skupine za istraživanje i inovacije EuroHPC; Direktor centra za istraživanje arhitektura i aplikacija za računarstvo visokih performanci na FER-u. Viši je član IEEE Computer Society.

SUMMARY

When analysing Internet traffic today it can be found that digital video content prevails. Its domination will continue to grow in the upcoming years and reach 80% of all traffic by 2021. If converted to Internet video minutes per second, this equals about one million video minutes per second. Providing and supporting improved compression capability is therefore expected from video processing devices. This will relieve the pressure on storage systems and communication networks while creating preconditions for further development of video services. Transform and quantization is one of the most compute-intensive parts of modern hybrid video coding systems. Improving the compression capability of this computation block is achieved using complex algorithms at the expense of increasing implementation complexity. Design requirements for higher throughput, reduced communication latency and low power consumption cannot be accomplished using homogenous systems and heterogeneous multiprocessor high performance systems are imposed as a solution.

This thesis presents an area efficient reusable architecture for the integer discrete cosine transform and quantization and also highly performance optimized kernel designed for execution on a GPU. In the case of hardware architecture, optimization is based on exploiting the symmetry and subset properties of the transform matrix. The proposed multiply-accumulate architecture is fully pipelined. It provides a two-way interface over which the processing system can control the data path of the transform process and receive the feedback information about utilization from the device. The proposed architecture is implemented on the FPGA platform, that achieves a throughput of 815 Msps and can support encoding of a 4K UHD@30 fps video sequence in real-time.

Considering GPU implementation, the performance optimization strategy involved all three aspects of parallel design, exposing as much of the algorithm's intrinsic parallelism as possible, with the exploitation of high throughput memory and efficient instruction usage. It combined efficient mapping of transform blocks to thread blocks and efficient vectorized access patterns to shared memory for all transform sizes. Two different GPUs were used to evaluate the proposed implementation. Speedup factors compared to CPU, cuBLAS and AVX2 implementations are up to 80, 19 and 4 times respectively.

Keywords: Video Coding, High Efficiency Video Coding (HEVC), Integer Discrete Cosine Transform (DCT), Heterogeneous Computing, Hardware Acceleration

Projektiranje transformacijskih i kvantizacijskih računskih blokova za videokompresiju optimiranih za učinkovito izvođenje na heterogenim višeprocesorskim računalima visokih performanci

Digitalni videozapis danas dominira u mnogim industrijama i uslugama više nego ikada prije. Ako se promatraju analiza i predviđanja IP (engl. *Internet Protocol*) podatkovnog prometa u razdoblju od 2017. do 2022., zastupljenost digitalnog videozapisa bit će u rasponu od 80% do 90% ukupnog prometa. Ovakav visoki udio videozapisa posljedica je ne samo zahtjeva korisnika za tom vrstom sadržaja već i uvođenja prijenosa videozapisa vrlo visoke razlučivosti (UHD, engl. *Ultra High Defintion* ili videozapis razlučivosti 4K UHD). Potrebna brzina prijenosa takvog videozapisa je dva puta veća od potrebne brzine prijenosa videozapisa visoke razlučivosti (HD, engl. *High Definition*) i devet puta veća od potrebne brzine prijenosa videozapisa standardne razlučivosti (SD, engl. *Standard Definition*). Predviđa se da će se 2022. 22% od ukupnog IP videoprometa odnositi se na videozapis razlučivosti 4K. Zamjetan je i trend pomaka IP prometa u smjeru mobilnih platformi koji doprinosi rastu svih oblika IP prometa a osobito videoprometu. Predviđanja su da će se do 2022. 29% prometa ostvarivati putem žičanih a preostali dio putem bežičnih mreža. Sve veća dostupnost mobilnih uređaja koji mogu reproducirati videozapise visoke razlučivosti te rast brzina prijenosa mobilnih mreža generiraju dodatnu potražnju za videozapisima visoke kvalitete i visoke razlučivosti.

Postoje različiti oblici internetskog videoprometa: internetski prijenos videozapisa, videozapis na zahtjev, mrežne videoigre te razmjena videozapisa putem dijeljenja datoteka. Ono što je zajedničko postojećim vrstama videozapisa i različitim aplikacijama koje postoje u lancu od generiranja videozapisa do njegove reprodukcije je potreba za video(de)kodiranjem ili video(de)kompresijom. Kodiranje izvode pružatelji usluga dok se dekodiranje obavlja u uređajima korisnika. Obrada videozapisa koja uključuje kompresiju i dekompresiju potrebna je kako bi pohranjivanje i prijenos te vrste podataka bili ekonomični. Naprimjer, za pohranu deset sekundi nekomprimiranog videozapisa razlučivosti 4K, brzine osvježavanja slike 30 fps (engl. *frame per second*, slika u sekundi), standardnog oblika komponentnog videosignala YCbCr i 8-bitne dubine boje potrebno je oko 3,48 GB memorijskog prostora ili preračunato u potrebnu brzinu 356 MB/s. Ako se takav videozapis komprimira koristeći najsuvremeniju normu kompresije HEVC (engl. *High Efficiency Video Coding*) ili H.265 potreban memorijski prostor za pohranu videozapisa je oko 4,34 MB, a potrebna brzina je oko 455 kB/s. Te vrijednosti ovise o karakteristikama videosadržaja i postavkama procesa kodiranja.

Kompresija videosignala temelji se na smanjenju redundancije unutar slike i/ili između slika. Prvi korak u tom postupku je predviđanje unutar slike te predviđanje između slika čiji je rezultat razlika između predviđene i trenutačne slike odnosno pogreška predviđanja. Što je taj korak učinkovitiji, manja je pogreška predviđanja koja se dalje kodira u sljedećim koracima. Prvi sljedeći korak je transformacija u frekvencijsku domenu nakon čega je većina energije signala pogreške sadržana u malom broju komponenti niskih frekvencija. Vrijednosti komponenti niskih frekvencija se nadalje smanjuju nelinearnom kvantizacijom. Taj postupak dovodi do gubitaka informacija i smanjenja vizualne kvalitete. Kvantizacijom je moguće upravljati tako da gubitak kvalitete bude prihvatljiv. Kvantizacija je ireverzibilan postupak. Zadnji korak kompresije je entropijsko kodiranje. Ono smanjuje redundanciju između bitova u slijedu kvantizacijskih razina i vrijednosti sintaksnih elementa.

Kodiranje videozapisa visoke razlučivosti i velike brzine osvježavanja slike zahtjeva veliku računalnu moć i veliku brzinu obrade. Računalni sustavi temeljeni na CPU-u (engl. *Central Processing Unit*) su dosegli gornju granicu brzine računanja po Watu uložene energije i nisu efikasni za takve aplikacije koje koriste velike količine podataka pogodnih za paralelni način obrade. Heterogena višeprocesorska računala visokih performanci su novo i prikladno rješenje za izvođenje takvih aplikacija. Takva računala osim CPU-ova sadrže i druge računalne čvorove kao što su GPU-ovi (engl. *Graphic Processing Units*), DSP-ovi (engl. *Digital Signal Processors*) i namjenske sklopovske akceleratore. Kombinirajući mogućnost paralelne programske izvedbe sa arhitekturama vrlo niske latencije postiže se visoka računalna propusnost uz nisku potrošnju energije. Projektiranje aplikacije za izvođenje na takvim računalima uključuje njenu razdiobu na dijelove pogodne za paralelno izvođenje i visoku propusnost te na dijelove namijenjene serijskom izvođenju, osjetljive na latenciju. Ti dijelovi aplikacije se potom izvode na odgovarajućim računalnim čvorovima, dijelovi pogodni za paralelno izvođenje na GPU-u, DSP-u ili FPGA-u (engl. *Field Programmable Gate Arrays*), a dijelovi pogodni za serijsko izvođenje na CPU-u. U usporedbi s tradicionalnim računalima, izvedba aplikacije na heterogenim računalima postiže tako visoke performance i energetsku učinkovitost. Pri tome je potrebno obratiti pozornost na dva pomoćna procesa koji mogu biti vremenski zahtjevni i značajno utjecati na brzinu izvođenja: prijenos podataka između računalnih čvorova i pokretanje izvođenja dijelova aplikacije na pridijeljenim čvorovima.

Glavni cilj ove doktorske disertacije bio je istražiti programske i sklopovske izvedbe transformacijskih i kvantizacijskih računskih blokova za učinkovito izvođenje na heterogenim višeprocesorskim računalima visokih performanci. Algoritmi za transformaciju i kvantizaciju

su jedan od najsloženijih i podatkovno najzahtjevnijih dijelova algoritama za kompresiju videozapisa. Inovativna i optimirana izvedba tih algoritama je ključna za postizanje visokih performanci u videokodiranju. Cjelobrojna aproksimacija DCT-a (engl. *Discrete Cosine Transform*), najzastupljenija je vrsta transformacije u videokodiranju i kompresiji slike, ujedno propisana u HEVC-u, tema je istraživanja ove disertacije. Kao tipični predstavnici računalnih čvorova unutar heterogenih višeprocesorskih računala visokih performanci uzeti su GPU za programsku izvedbu, a FPGA za sklopovsku izvedbu ovih računskih blokova. Nakon analiza matematičkih modela, značajki transformacije DCT te postojećih izvedbi, predložena je nova sklopovska i nova programska izvedba uz razrađenu metodologiju potvrde učinkovitosti.

DCT se može efikasno izvoditi korištenjem DFT-a (engl. *Discrete Fourier Transform*). DFT se koristi za općenitu spektralnu analizu i nalazi primjenu u mnogim područjima obrade signala. DFT je definiran za kompleksne signale ograničenog trajanja, a DCT za realne signale ograničenog trajanja koji su podskup skupa kompleksnih signala. Značajka DCT-a koja ga čini prikladnijim za kompresiju u odnosu na npr. DFT je da je većina energije signala sadržana u malom broju frekvencijskih komponenti videosignala. Ovo je poželjna karakteristika za kompresijske algoritme. Izvorni signal u vremenskoj ili prostornoj domeni je dakle moguće predstaviti prilično vjerno koristeći mali broj DCT koeficijenata u frekvencijskoj domeni. Preostali koeficijenti se uklanjaju u fazi kvantizacije.

Kompresijska učinkovitost transformacije postiže se sve većim brojem transformacijskih blokova različitih dimenzija čime se oni prilagođavaju prostorno-frekvencijskim karakteristikama ulaznog bloka slike. Učinkovitom izvođenju i povećanju interoperabilnosti doprinio je cjelobrojni DCT. Sklopovska izvedba cjelobrojnog množenja je brža, koristi manje memorijskih resursa, troši manje energije i zauzima manje područje. Učinkovitost programske i sklopovske izvedbe je dodatno naglašena u postupku standardizacije za HEVC normu. Osim energetske kompaktnosti, ortogonalnosti i simetričnosti baznih vektora HEVC je s manjim brojem različitih elemenata transformacijske matrice postigao manje područje potrebno za sklopovsku izvedbu. Ponavljanjem elemenata baznih vektora za sve veličine transformacije omogućena je višestruka upotreba istih množitelja, a jednakim normiranjem baznih vektora izvođenje kvantizacije operacijom skalarnog množenja. Elementi transformacijske matrice koriste 8-bitni zapis, registri za ulazne, transponirane i izlazne podatke te množitelji su 16-bitni, a akumulatori 32-bitni. Sve aritmetičke operacije nad

transformacijskim blokovima je moguće izvesti koristeći procesorske naredbe SIMD (engl. *Single Instruction Multiple Data*).

U sklopu provedenog istraživanja razvijeno je sklopovsko rješenje za cjelobrojni jednodimenzionalni DCT za sve podržane veličine transformacije u HEVC-u s minimizacijom područja sklopovske izvedbe koristeći FPGA tehnologiju te programsko rješenje za transformaciju i kvantizaciju prema HEVC normi optimirano za propusnost podataka pri izvođenju na GPU-u unutar heterogenog višeprocesorskog računalnog sustava.

Zbog različite arhitekture temeljnih logičkih blokova predloženo je sklopovsko rješenje za dva odabrana uređaja FPGA od dva vodeća proizvođača Xilinx i Intel Altera. Pritom su korišteni razvojni alati Vivado IDE te Intel Quartus Prime pri čemu je Vivado IDE bio glavni alat. Metodologija projektiranja usklađena je s proizvođačevom preporučenom metodologijom *UltraFast*. Nakon što su svi moduli sklopovskog rješenja prošli simulacijske testove dizajn je sintetiziran i implementiran. Fizički ostvariv model (engl. *Register Transfer Level Model*) i vremenska ograničenja kao npr. trajanje signala takta su ulazni podaci potrebni za tu fazu razvoja. Budući je smanjivanje područja sklopovske izvedbe postavljeno kao glavni cilj provjerene su predefinirane strategije logičke sinteze i implementacije dostupne u alatima. Za svaku inačicu sklopovskog rješenja zabilježene su količine upotrijebljenih logičkih blokova, registara te potrošnja energije, a proračunata je najveća moguća frekvencija takta i propusnost sklopa. Inačica potvrđena za Xilinxov uređaj je potom sintetizirana i implementirana za Intelov FPGA sklop.

Kako bi isto sklopovlje moglo biti korišteno za sve veličine ulaznih blokova, dizajn je strukturiran u sklopovske staze. Staza koja je projektirana za izračun neparnih elemenata izlaznog vektora za veličinu transformacije N označava se kao *Staza N*. Svaka takva staza instancirana je $N/2$ puta, a osim za veličinu transformacije N korištena je i za sve veće veličine transformacije. Jednu stazu čine ulazni multipleksor, multipleksirano množilo tipa MCM (engl. *Multiple Constant Multiplication*) i akumulator. Ona realizira skalarni umnožak jednog baznog vektora transformacijske matrice i ulaznog vektora pogreške predviđanja. Ovisno o množiteljima koji su izvedeni u MCM-u razlikuje se pet tipova staza. Tijekom $N/2$ ciklusa takta akumulator pribraja umnoške trenutačnoj pohranjenoj vrijednosti. Nakon tog vremena se na izlazu svake staze postavlja rezultat, 16-bitni koeficijent odnosno jedan element izlaznog vektora. Svaka staza sadrži dva konfiguracijska registra. U prvi konfiguracijski registar upisuje se kodna riječ preko koje se upravlja adresnim ulazima svih multipleksora unutar množila

MCM kako bi se nakon tri ciklusa takta izračunao umnožak s jednim od predefiniranih množitelja. Drugi konfiguracijski registar sadrži kodne bitove za upravljanje akumulatorom gdje logičko stanje „0” označava pribrajanje umnoška stanju akumulatora, a logičko stanje „1” oduzimanje umnoška od trenutačnog stanja akumulatora.

Instance multipleksiranog MCM-a dominantni su funkcijski blokovi s obzirom na postotak zauzeća ukupnog područja sklopovske izvedbe. Taj tip MCM-ova je odabran umjesto paralelnog kako bi područje bilo što manje. Njihov nedostatak je veće kašnjenje i smanjena propusnost. Uštede područja sklopovske izvedbe su postignute i u dizajnu ulaznog podsklopa. Parovi elemenata ulaznog vektora koji su 16-bitne vrijednosti slijedno se dovode na njegov ulaz odnosno ulaz cijelog sklopa te se zbroj i razlika njihovih vrijednosti zapisuju u registar čija vrijednost se potom postavlja na ulaze svih staza. Širina ulazne sabirnice je dakle 32 bita. U ulaznom multipleksoru koji se nalazi u svakoj stazi se jedna od te dvije vrijednosti prosljeđuje na izlaz. Kada sklop računa transformaciju veličine N onda ulazni multipleksor u *Stazi N* na svoj izlaz postavlja razliku vrijednosti ulaznih parova, a za sve veće veličine transformacije njihov zbroj.

Na izlazu sklopa se nalazi podsklop za serijalizaciju koji sekvencijalno čita izlaze iz dvije susjedne instance staza te ih postavlja na izlaz sklopa. Za komunikaciju između sklopa i procesorskog sustava projektirana su dva sinkronizacijska kanala. Preko jednog procesor može zaustaviti serijalizaciju što uređaj potvrđuje aktivacijom odgovarajućeg izlaza. Preko drugog kanala sklop signalizira zastoj kada do njega dođe kako procesor ne bi nastavio sa slanjem novih podataka. Do zastoja dolazi kada je vektor transformacijskih koeficijenata postavljen na izlaz bloka staza, a serijalizacija prethodnog vektora još nije gotova. Uz signalizaciju zastoja procesoru izlazni vektor se upisuje u međuspremnik za kašnjenje. Nakon deaktivacije signala zastoja pokreće se novi ciklus transformacije, a vektor iz međuspremnika za kašnjenje se serijalizira.

Predloženo sklopovsko rješenje nadmašuje slična rješenja s obzirom na područje sklopovske izvedbe. Najveća frekvencija rada za transformacijski i kvantizacijski blok je 407,5 MHz. Predložena arhitektura može biti korištena za kodiranje u stvarnom vremenu videosadržaja razlučivosti 4K UHD i brzine osvježavanja slike 30 fps.

Programsko rješenje za transformaciju i kvantizaciju prema HEVC normi optimirano je za najveću moguću propusnost podataka pri izvođenju na GPU-u unutar heterogenog višeprocesorskog računalnog sustava. U procesu razvoja korištena su dva ispitna okruženja

kako bi se potvrdila učinkovitost postupaka optimiranja za GPU-ove sa različitim brojem multiprocesora (SM, engl. *Streaming Multiprocessor*) i napravila usporedba sa ostalim programskim rješenjima koja koriste isti tip GPU-ova. Prvo okruženje sadržavalo je GPU iz obitelji *GeForce*, a drugo okruženje GPU iz obitelji *Tesla*, pri čemu oba imaju arhitekturu Kepler. Programski model CUDA (engl. *Compute Unified Device Architecture*) proizvođača NVIDIA Corp. je upotrijebljen za pisanje programske podrške za korišteni heterogeni računalni sustav. Usporedno s izvedbom za GPU, dodatna izvedba transformacijskog i kvantizacijskog bloka je projektirana za izvođenje na GPU-u uz korištenje NVIDIA-inih funkcija iz biblioteke cuBLAS (engl. *CUDA Basic Linear Algebra Subroutines*). U ispitivanja su uključene i dvije izvedbe za CPU. Prva koristi naredbe za jednostruki tok podataka, a druga vektorske naredbi AVX2 (engl. *Advanced Vector Extensions 2*). U procesu razvoja programskog rješenja primijenjen je iterativni i inkrementalni pristup. Provedena mjerenja su uključivala mjerenje trajanja jezgrene funkcije koja izvodi transformaciju i kvantizaciju svih transformacijskih blokova u jednoj slici te mjerenje ukupnog vremena koje osim trajanja jezgrene funkcije obuhvaća i trajanje prijenosa podataka od CPU-a do GPU-a i od GPU-a do CPU-a. Ukupno vrijeme odgovara vremenu transformacije i kvantizacije jedne slike u videozapisu. S obzirom na postavljene ciljeve, pojedine iteracije su uključivale mjerenja efektivne propusnosti globalne memorije, iskoristivosti skupova dretvi i učinkovitosti pristupa dijeljenoj memoriji. U ispitivanjima su korištene slike videozapisa razlučivosti *DCI 4K* (engl. *Digital Cinema initiatives*) i *8K Full Format* sa sustavom uzorkovanja komponentnog videosignala 4:2:0.

Za povećanje performanci predloženog rješenja u odnosu na druge izvedbe a s obzirom na propusnost podataka ključno je učinkovito korištenje raspoloživih resursa na GPU-u: blokova dretvi i samih dretvi (dvije razine paralelizma), dijeljene memorije pridijeljene svakom bloku, registara u svakoj dretvi te zajedničke globalne memorije. Svaki SM izvodi dretve preko skupova dretvi gdje svaki skup čine 32 dretve. Broj aktivnih blokova dretvi i skupova dretvi po SM-u ovisi o izvedbenoj konfiguraciji jezgrene funkcije, veličini prostora zauzetog u dijeljenoj memoriji i broju korištenih registara. U jezgrenoj funkciji za transformaciju i kvantizaciju obrađuju se velike grupe blokova različitih veličina, a dva matrična množenja predstavljaju računski najzahtjevnije operacije na koje otpada više od 90% vremena izvođenja funkcije. S obzirom na broj bajtova koji jezgrena funkcija dohvaća i pohranjuje u globalnu memoriju i broj aritmetičkih operacija koje koristi, ona može biti obilježena memorijskim pristupom ili aritmetičkim operacijama.

Zbog potrebnog vremena od 50 μs između završetka izvođenja jedne jezgrene funkcije i početka izvođenja druge projektirana je jedinstvena jezgrena funkcija. Koristeći predložak funkcije s dimenzijom bloka kao parametrom predloška, funkcija računa blokove razina i identifikacijski AZB (engl. *All-Zero-Block*) niz, a ulazi su joj blokovi pogreške predviđanja, kvantizacijski parametar i tri faktora skaliranja. Budući se kod matričnog množenja skalarno množe vektori duljine N svi blokovi se spremaju u dijeljenu memoriju jer je trajanje dohvaćanja podataka za red veličine kraće nego kad se operandi nalaze u globalnoj memoriji. U dijeljenu memoriju se kopiraju i blokovi transformacijske matrice smješteni izravno u globalnu memoriju na početku programa. Obrasci pristupa podacima u dijeljenoj memoriji ključna su odrednica performanci predloženog programskog rješenja. Ako se blokovi podataka za oba operanda matričnog množenja izravno kopiraju u dijeljenu memoriju, dolazi do serijalizacije pristupa memorijskim bankama, a trajanje transformacije i kvantizacije jedne slike je pritom četiri puta dulje. Zbog toga se koristi tehnika dopune blokova kako bi se izbjegla serijalizacija pristupa. Budući je propusnost jedne banke dijeljene memorije 64 bita po ciklusu takta podaci su grupirani koristeći tip podataka *short4* a dijeljena memorija rekonfigurirana za 64-bitni način rada kako bi se u jednoj transakciji po memorijskoj banci dohvaćala četiri susjedna elementa u redu bloka. Ovakav obrazac pristupa doveo je do vektorizacije jezgrene funkcije jer je jedna dretva računala četiri kvantizirana transformacijska koeficijenta.

Od izvedbe za GPU-u koja koristi metode iz NVIDA-ine biblioteke cuBLAS očekivalo se ubrzanje u odnosu na vlastito rješenje. Budući cuBLAS metode ne podržavaju cjelobrojne tipove podataka širine veće od 8 bita, korišten je tip podataka *float*. Za svaku od dvije faze matričnog množenja pozvana je metoda za grupno množenje matrica koje su jednako razmaknute u memoriji. Nakon svakog poziva bilo je potrebno pozvati jezgrenu funkciju za zaokruživanje kako bi međurezultati i konačni rezultati bili u skladu s normom HEVC. Takva izvedba se pokazala bržom samo uz korištenje skalarnih tipova podataka. Kada je predloženo programsko rješenje za izvođenje na GPU-u koristilo vektorske tipove podataka *float2* i *short4*, izvedba GPU cuBLAS je bila sporija i u slučaju kada su jezgrene funkcije za zaokruživanje bile izostavljene.

U sklopu projektiranja izvedbe za GPU bilo je potrebno naći optimalan omjer mapiranja broja transformacijskih blokova na blok dretvi i ispitati utjecaj dopune blokova u dijeljenoj memoriji za različite dimenzije transformacije. Trajanje izvođenja najkraće je uz konfigurirani blok dretvi dimenzije 8×32 što je i najmanja moguća veličina bloka dretvi za najveću veličinu

transformacije. Za veće dimenzije bloka dretvi, kada on obrađuje dva, tri ili četiri transformacijska bloka, sinkronizacije izvođenja dretvi i serijalizacije tijekom izračuna AZB indeksa negativno utječu na trajanje izvođenja. Kod mapiranja najmanjih transformacijskih blokova na konfigurirani blok dretvi, unutar njega se obrađuje 64 transformacijska bloka. Za veće omjere mapiranja obrasci pristupa podacima u globalnoj memoriji postaju neučinkoviti. Tehnika petlje sa skokom unutar jezgrene funkcije, koja se koristi u dohvaćanju novih blokova podataka, smanjuje učinkovitost priručne memorije L2 jer se skok povećava za veći omjer mapiranja i podaci se moraju dohvaćati iz globalne memorije. Što se tiče dopune blokova u dijeljenoj memoriji, pokazano je da ona pozitivno utječe na učinkovitost pristupa dijeljenoj memoriji za blokove dimenzija 16 × 16 i 32 × 32, a za manje blokove ne. Razlog tome je veća penalizacija serijalizacije pristupa podacima u stupcima blokova kada nema dopune nego prekomjerno alociranog memorijskog prostora po bankama kada je ima. Za manje veličine blokova ili nema serijalizacije tijekom izvođenja snopa dretvi (4 × 4) ili je ona manje kritična za učinkovitost pristupa od prekomjerno alociranog memorijskog prostora (8 × 8).

Kako bi se ubrzala transformacija i kvantizacija slike iskorišteni su CUDA tokovi koji omogućuju vremensko preklapanje prijenosa podataka i izvođenja jezgrene funkcije. Najveća brzina je postignuta kada su transformacijski blokovi razdijeljeni u osam tokova. Postignuto ubrzanje ovisi o omjeru trajanja prijenosa podataka i trajanja izvođenja jezgre. Što je on bliži jedan, veće je ubrzanje.

Predloženo programsko rješenje postiže prosječno vrijeme transformacije i kvantizacije slike 6,03 ms odnosno 23,94 ms po slici za videozapise razlučivosti *DCI 4K* i *8K Full Format*. Ubrzanja u odnosu na izvedbe za CPU, GPU cuBLAS i AVX2 su redom 80, 19 i 4 puta. S obzirom na vrijeme izvođenja predloženo rješenje nadmašuje sličnu izvedbu transformacijskog i kvantizacijskog bloka 1,22 puta te može biti upotrijebljeno za dekodiranje u stvarnom vremenu.

Unutar ovog doktorskog rada ostvarena su tri izvorna znanstvena doprinosa:

1. Projektiranje programske izvedbe transformacijskih i kvantizacijskih računskih blokova za videokompresiju optimiranih za učinkovito izvođenje na heterogenim višeprocesorskim računalima visokih performanci. Izvedba za GPU postiže ubrzanja do redom 80, 19, 4 i 1,22 puta u odnosu na izvedbe za CPU, cuBLAS, AVX2 i sličnu paralelnu izvedbu za GPU. Predloženo programsko rješenje može se koristiti za

dekodiranje u stvarnom vremenu videozapisa razlučivosti 4K UHD i brzine osvježavanja slike 50 fps.

2. Projektiranje sklopovske izvedbe transformacijskih i kvantizacijskih računskih blokova za videokompresiju optimiranih za učinkovito izvođenje na heterogenim višeprocesorskim računalima visokih performanci. Sklopovska izvedba koristi FPGA tehnologiju i postiže propusnost od 1 Gsps uz korištenje 9,6% manje sklopovskih resursa u usporedbi sa sličnom izvedbom. Predložena arhitektura može se koristiti za dekodiranje u stvarnom vremenu videozapisa razlučivosti 4K UHD i brzine osvježavanja slike 30 fps.
3. Metodologija potvrde učinkovitosti programske i sklopovske izvedbe optimiranih računskih blokova. Za potvrdu učinkovitosti programske izvedbe za GPU mjeri se trajanje izvođenja jezgrene funkcije i trajanje izvođenja transformacije i kvantizacije slike u videozapisu. Dobiveni rezultati uspoređuju se s ostalim izvedbama. U svrhu veće učinkovitosti procesa optimiranja, korištena metrika uključuje i mjerenja efektivne propusnosti globalne memorije, iskoristivosti skupova dretvi i učinkovitosti pristupa dijeljenoj memoriji. Za sklopovsku izvedbu koja koristi FPGA tehnologiju proračunava se propusnost podataka i vrednuje u odnosu na količinu upotrijebljenih sklopovskih resursa koja treba ostati mala. Za obje predložene izvedbe utvrđuje se mogućnost korištenja u sustavima za videokompresiju u stvarnom vremenu.

Ključne riječi: videokodiranje, norma za videokodiranje HEVC, cjelobrojna diskretna kosinusna transformacija, heterogeno računarstvo, sklopovske jezgre za ubrzanje

TABLE OF CONTENTS

1 Introduction 1

1.1 Thesis outline 4

2 Transform coding 5

2.1 Introduction 5

2.2 Discrete cosine transform 6

2.2.1 Derivation and definitions 6

2.2.2 DCT of type II 9

2.2.3 Energy compaction and decorrelation 11

2.3 Quantization 15

2.3.1 Quantizer types 17

2.4 Transform and quantization in HEVC 19

2.4.1 The integer DCT 19

2.4.2 HEVC core transform design 21

2.4.3 Quantization and dequantization 24

3 A performance efficient HEVC transform architecture 25

3.1 Introduction 25

3.2 The mathematical model for hardware implementation 26

3.3 2D integer DCT architecture 28

3.3.1 1D integer DCT architecture 30

3.4 Overview of scientific contributions to the integer DCT architecture 31

3.5 Development environment and methodology 33

3.6 An area efficient and reusable HEVC 1D transform architecture 37

3.7 Implementation and evaluation 41

4 A performance optimized software implementation of the HEVC transform and quantization algorithm 44

4.1 Introduction 44

4.2 HEVC transform and quantization process 45

4.3 Overview of scientific contributions to HEVC transform algorithms on GPUs 46

4.4 Development environment and methodology 47

4.4.1 Effective memory bandwidth 50

4.4.2 Occupancy 51

4.4.3 Shared memory efficiency 51

4.5 Performance engineering for HEVC transform on GPU 52

4.5.1 Indexing transform blocks with thread-blocks and threads 54

4.5.2 Vectorized memory access 57

4.5.3 Efficient vectorized access pattern to shared memory 58

4.6 Iterative implementation and evaluation 62

4.6.1 Merging transform and quantization kernel 62

4.6.2 Benchmarking against NVIDIA's library functions 69

4.6.3 Vectorized memory access with a float2 data type 72

4.6.4 Vectorized memory access with a short4 data type 75

4.6.5 The single access transform matrix 78

4.6.6 Transform block to thread-block efficient mapping 80

4.6.7 Page-locked memory transfer 81

4.6.8 Shared memory padding 83

4.6.9 Overlapping kernel execution and data transfer 84

4.6.10 Performance evaluation on the Workstation environment and comparison with a competing implementation 87

5 Conclusion 92

References 95

LIST OF FIGURES **101**

LIST OF TABLES **104**

Biography **106**

Životopis **107**

1 INTRODUCTION

Today, more than ever before, digital video dominates many industries and services. If observing IP video traffic statistics and forecast for the time period from 2017 to 2022, it is in the range of 80 to 90 percent of total IP traffic [1]. Aside from high customer demand for video content, this high share is caused by introduction of the Ultra-High-Definition (UHD) or 4K video streaming. Its bit rate at about 15 to 18 Mbps, is more than double the High-Definition (HD) video bit rate and nine times more than the Standard-Definition (SD) video bit rate. It is estimated that in 2022 the 4K video content will account for 22 percent of the global IP video traffic. This emerging technology reflects customer requests for high resolution and high quality video content. There exist different forms of IP video content including Internet video, IP Video on Demand (VoD), video conferencing, video-streamed gaming and video files exchanged through file sharing. The very broad range of the forms and applications characterized by different specifications and constraints have one thing in common, video (de)compression. Compression is done by content providers and decompression is carried out at the consumer device. Video compression is a necessary processing step for affordable and efficient storage and transmission. Without compression, it would not be feasible to store and transmit digital video content because of the required huge storage capacity and a very large bandwidth. For example, ten seconds of a YCbCr 4:2:0 8-bit 4K video with a frame rate of 30 fps [2] requires a storage capacity of about 3.48 GB or, when recalculated, a bitrate of 356 MB/s. If such a video is compressed by using the state-of-the-art High Efficiency Video Coding (HEVC) standard[1], storage capacity and bitrate are 4.34 MB and 455 kB/s respectively. In case of a compressed video the output file size or required bandwidth depend on video content and encoding settings.

Concerning trends contributing to the continued growth of global IP traffic, a shift toward mobility is noticeable. In upcoming years an increasing part of the traffic will originate from mobile or portable devices. It is predicted that by 2022 the wired networks will account for 29 percent of IP traffic, and Wi-Fi and mobile networks will account for 71 percent of IP traffic [1]. Video also becomes the dominant medium in the domain of mobile networks. Nearly 61 of the 77 exabytes crossing the mobile network per month will be ascribed to video by 2022 [4]. The new generation of video codecs, the availability of higher resolution mass market

[1] The HEVC reference software HM [3] was selected. Video was encoded using Random Access configuration and default encoding settings.

devices and mobile networks speed improvements enable the increased usage of mobile networks for video applications. High quality video streaming is not feasible with older mobile network infrastructure and devices without higher processing capabilities. When conditions for high quality video streaming are met that increases its popularity among consumers.

The new generation of video codecs provides higher quality at lower bit rates. This accomplishment was made at the cost of a big increase in the computational complexity of video encoding and decoding. Two major standards in this space are HEVC and AV1. Due to wide adoption and legacy issues, video encoding is largely skewed towards Advanced Video Coding (AVC) today, a representative of the previous generation of video codecs. HEVC doubled the compression efficiency compared to AVC [5]. HEVC reference encoder outperforms AV1 in terms of bit rate [6] but its disadvantage is a royalty-based business model. Upcoming years will determine which business model, whether the royalty-free open source software, with AV1 as its representative, or the royalty based one, with Versatile Video Coding (VVC), a successor of HEVC, with a planned release at the end of 2020, will prevail at the market.

Video signal compression relies on redundancy reduction inside and/or between the frames or images. The reduction is accomplished by intraframe and interframe prediction [7] where previous frames are utilized to predict the current one. The predicted and current frame are subtracted and then compressed. The more efficient this process is, there will be less data in the subtracted, prediction error or residual frame for subsequent compression steps. In addition to prediction and differencing, the transformation from time domain to frequency domain is used. The main characteristic of the transformation to frequency domain is that the energy of the signal becomes concentrated in a small number of lower frequencies. For the purpose of further compression, a value range is further reduced through nonlinear quantization. This may lead to losses in the video signal. If those losses are made on a level providing satisfactory visual quality they can be assessed as acceptable. Quantization is the only irreversible processing step and due to losses in it, the entire compression is attributed as lossy compression. The final compression technique used in modern video compression systems is entropy coding. It reduces the redundancy between bits in the sequence of quantized data and syntax element values.

Compared to compression, video signal decompression is a reverse process with reconstructed frames as the process output which can be either stored for future display or

immediately sent to the console output in VoD or a real-time application. Decompression starts with entropy decoding. After dequantization and inverse transform, the residual frame is restored in the time domain. Subsequently, it is added to the reference frame to obtain a reconstructed frame using the same prediction process like on the encoding side.

Video coding of high-resolution and high frame rate videos using new generation video codecs demands a high processing speed and compute capability. Computing systems based on the Central Processing Unit (CPU) have reached the upper boundary in computation speed per Watt and cannot be efficiently utilized for such innovative applications. Video coding applications require processing a large amount of data and exhibit a high level of parallelism. The emerging heterogenous multiprocessor high performance computers can be used as a solution for those challenges. They combine CPUs with non-traditional computing devices such as Graphic Processing Units (GPUs), Digital Signal Processors (DSPs) and specialized hardware (HW) accelerators to achieve high computation throughput at a lower power consumption through exploitation of massive parallelism and low latency architectures. Design for heterogenous computing systems involves portioning of an application to parallel, high throughput parts and latency optimized, serial parts which are then run on appropriate computing nodes. The former are executed using GPUs, DSPs or Field Programmable Gate Arrays (FPGAs), and the latter on CPUs. In such a way heterogenous computing system can achieve a much higher application performance and energy efficiency than the traditional computing systems. The migration to heterogenous computing exposed two overheads that have to be solved to make it completely successful. Initiating a task on a non-CPU computing device, as well as transferring data to and from such devices can be time-intensive and can have a major impact on the overall achievable speed of a parallel application. Moreover, these overheads restrict the sort of work which can be offloaded.

This thesis investigates novel techniques for transform and quantization computation blocks in video compression systems that target heterogeneous multiprocessor high performance computers. Transformation and quantization algorithms are one of the most processing and data demanding parts of the compression algorithm. Innovative and optimized implementation of those algorithms on CPU-GPU systems and using FPGA is a key to high performance video encoding. Finite precision approximation of the Discrete Cosine Transform (DCT), the most widely used transform type in video and image compression, employed in the HEVC standard is a use-case which was addressed in the thesis. As a typical representative of

processing units in heterogeneous systems, the GPU is taken for SW implementation and the FPGA for HW implementation of the computation blocks. After the analysis of theory and properties of DCT transforms in already existing techniques, novel performance optimized implementations are proposed. Detailed validation is conducted to confirm their efficiency and methodology is revealed for each particular proposed implementation.

1.1 Thesis outline

Chapter 2 describes DCT (with DCT type II as its most common variant) and quantization. The relation with DFT that also decomposes finite length discrete-time signal into a sum of scaled and shifted basis functions, but using the harmonically related complex functions instead of real-valued cosine functions, is presented. Emphasis is placed on the properties which allow coding efficiency, and this is demonstrated on the transform and quantization of a video frame. Finally, transform and quantization in the HEVC standard, which is designed considering implementation efficiency and which presents the base of this research is described.

Chapter 3 describes the hardware architecture design for HEVC transform and quantization and presents results of its validation. As an introduction to the author's own design and description of the methodology for efficiency validation, the mathematical model and general architectures of the integer DCT computation block are discussed. This is followed by the evaluation of related hardware architectures that target ASIC or FPGA devices.

Chapter 4 deals with the performance optimized GPU implementation of the HEVC transform and quantization kernel. The chapter begins with a functionality overview of the kernel’s computation subblocks which are ported to the GPU. The development environment and methodology for efficiency validation are described and key design decisions are discussed. The chapter concludes with an analysis and evaluation of implemented optimization techniques, performed in iterations.

Finally, Chapter 5 concludes the thesis, briefly describes the motivation for this work, summarizes the results of the research and suggests directions for further research.

2 TRANSFORM CODING

2.1 Introduction

Since 1988 and the appearance of the H.261 video coding standard, transform coding is the basis for all video coding standards, together with prediction error coding and entropy coding. Video signals contain information in three dimensions. In video encoding schemes they are modelled as spatial and temporal domains. There is a high level of information redundancy in both domains. Neighbouring samples are highly correlated, and the energy tends to be evenly spread across the domains. Taking this into account, discarding data or decreasing the precision would also decrease video quality and irreversibly impair end-user experience.

The main goal of transform coding is to map a set of pixel values from the spatial into the transform domain. The mapping process has to result in a compact form in terms of signal energy. A small number of significant low frequency coefficients in the transform domain will contain most of the energy. Besides the energy compaction, data decorrelation has to be obtained so that those data, which contribute less to video quality, are removed in a later processing. Data removal is made during the quantization stage. Therefore, quantization is usually a part of the transform coding process, although often not explicitly stated. Transform and quantization properties have to be exploited for efficient implementation.

DCT, originally proposed in 1974 [8], is widely used in image and video coding nowadays due to its properties which satisfy all the general requirements placed on the mapping process. Considering strong energy packing and decorrelation properties it is comparable to the Karhunen–Loève transform (KLT) [9] which completely removes the statistical dependence between transform coefficients. KLT's disadvantages are dependence on the input signal and a complex algorithm.

This chapter focuses on the DCT and its derivation integer DCT, which advances DCT's implementation efficiency, ensures device interoperability and minimizes the drift between encoder and decoder implementations. Rather than on mathematical theory, the emphasis is put on properties which are useful both for compression efficiency and for efficient implementation.

2.2 *Discrete cosine transform*

2.2.1 Derivation and definitions

DCT transforms a group of image samples into a group of transform coefficients. It is a reversible operation. An inverse operation to forward DCT (FDCT), the inverse DCT (IDCT), transforms a group of coefficients into a group of image samples. DCT can be applied to a 1D group of data with a 1D group of coefficients as a result or to a 2D group or block of data with a 2D group or block of coefficients. Both arrangements of the DCT are illustrated in Figure 2.1.

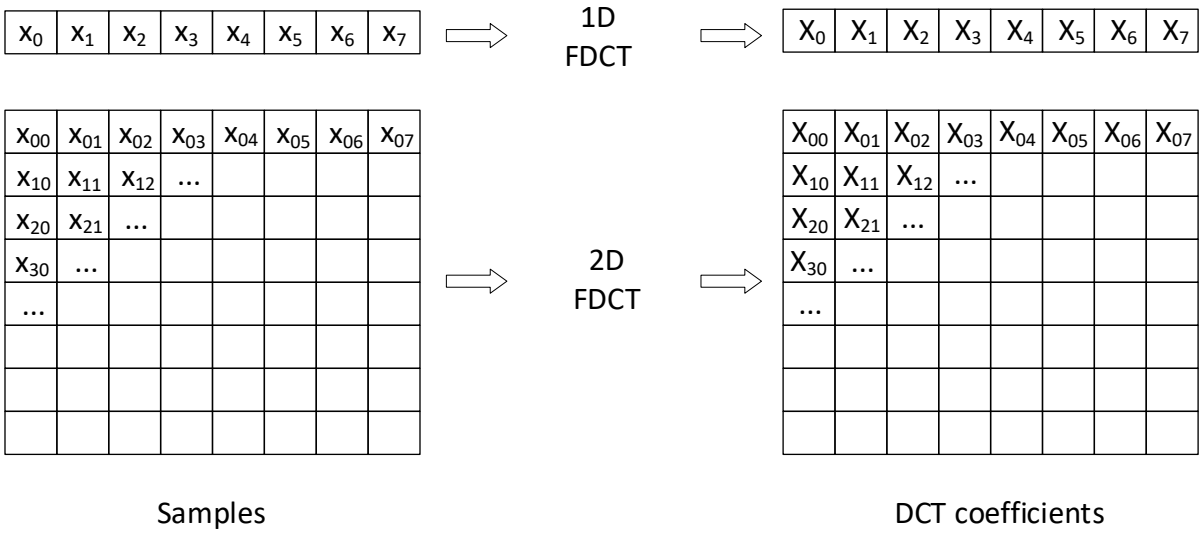


Figure 2.1: 1D and 2D discrete cosine transform

DCT can be derived from the Discrete Fourier Transform (DFT) [10] where frequency coefficients are calculated as follows:

$$X_m = \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi mn}{N}} \tag{2.1}$$

for $m = 0, \dots, N-1$. DFT is widely used for general spectral analysis applications that find their way into a range of fields. It is a linear transform which takes as input a complex signal x_n of length N, and gives as output a complex signal X_m of length N. From (2.1) it can be seen that DFT has a kernel given by:

$$W_{mn}^{N} = e^{-j\frac{2\pi mn}{N}} \tag{2.2}$$

W_{mn}^{N} can be represented as a complex matrix of size $N \times N$. Consequently, the Equation 2.1 could be shown as a matrix multiplication expressed as:

$$\boldsymbol{X} = \boldsymbol{W}\boldsymbol{x} \tag{2.3}$$

Compared to the continuous Fourier transform DFT is applicable for computing applications as a finite number of samples are taken and processed. This means that the signal x_n will be zero outside the domain $\{0, \dots, N-1\}$. Further simplifications can be made considering real discrete-time signals. Complex numbers are used in representing the signal just for mathematical convenience. Real signals are a subset of complex signals. So, when the real signals are analysed with a full complex transformation, it can be expected that the outcome will be restricted as well. One of the restrictions is that the real signals have certain symmetries in the Fourier domain, that the fully complex signals do not have.

To derive DCT from DFT, a new signal denoted as y_n is constructed as follows:

$$y_n = \begin{cases} x_n, & 0 \le n < N \\ x_{2N-n-1}, & N \le n < 2N \end{cases}$$

It can be noticed that y_n is constructed from x_n by adding a mirrored version of x_n to itself as depicted in Figure 2.2. It is even symmetric around the point halfway between and $N-1$ and N.

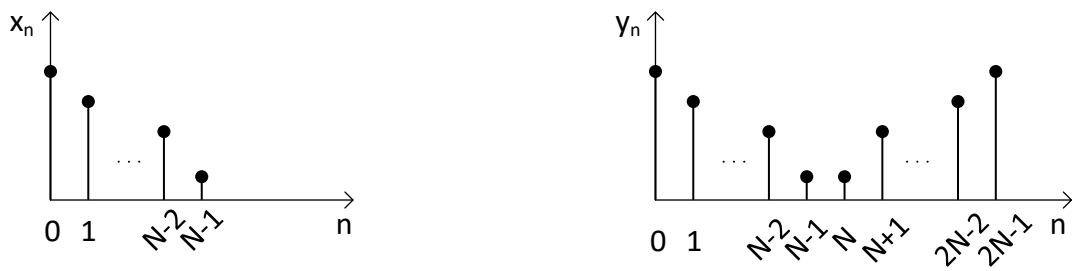


Figure 2.2: Source and constructed signals x_n and y_n

In the next step, $2N$-point DFT of y_n is computed as:

$$Y_m = \sum_{n=0}^{2N-1} y_n e^{-j\frac{2\pi mn}{2N}} \tag{2.4}$$

for $m = 0, \dots, 2N-1$. By rewriting Equation 2.4 as a function of N terms only, the following result can be obtained:

$$Y_m = e^{j\frac{\pi m}{2N}} \sum_{n=0}^{N-1} 2x_n cos\left[\frac{\pi}{2N} m(2n+1)\right] \tag{2.5}$$

for $m = 0, \dots, 2N-1$. If summation part of the expression (2.5):

$$C_m = \sum_{n=0}^{N-1} 2x_n cos\left[\frac{\pi}{2N} m(2n+1)\right] \tag{2.6}$$

is denoted as C_m, Equation 2.5 can be rewritten as:

$$Y_m = e^{j\frac{\pi m}{2N}} C_m \tag{2.7}$$

Factor C_m is defined as N-point DCT. As signal x_n was assumed to be real, it can be seen that its DCT will be real as well. Equation 2.7 presents the relation between DFT of signal y_n and DCT of signal x_n. Mirroring made during the construction of y_n provides a way of calculating C_m from Y_m because of its symmetry. It follows:

$$C_m = e^{-j\frac{\pi m}{2N}} Y_m \tag{2.8}$$

for $m = 0, \dots, N-1$. Overall relations between DCT and DFT including their inverse operations are illustrated in Figure 2.3.

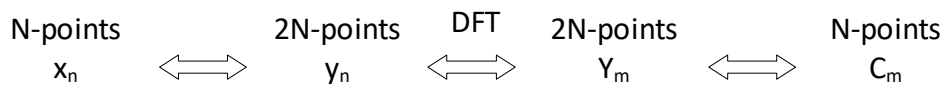


Figure 2.3 Relation between source signal x(n), constructed signal y(n) and their transforms

By exploiting symmetry and using Equation 2.7, Y_m can be derived from C_m in a similar way as in the time domain [11]. The IDCT is given by:

$$x_n = \frac{1}{N} \sum_{m=0}^{N-1} w_m C_m cos\left[\frac{\pi}{2N} m(2n+1)\right] \tag{2.9}$$

for $n = 0, \dots, N-1$ where

$$w_m = \begin{cases} 1/2, & m = 0 \\ 1, & m = 1, 2, \dots, N-1 \end{cases}$$

The pair defined in Equations 2.6 and 2.9 is also referred to as the even symmetrical DCT or DCT of type II. There are three more types of DCT which are determined by choices made when implicitly extending the finite and discrete input signal to make it periodic, and by the point around which the signal is even or odd. These choices relate to signal behaviour at domain boundaries. For cosine transforms, signal is even around the left boundary which is the

opposite of sine transforms where it is odd. Concerning the right boundary, if the signal is even around it then DCT is of type I or II and if odd then of type III or IV. When the signal is even or odd around the last sample in a sequence then DCT is of type I or III, and when around halfway between the last sample and the first extension sample then it is of type II or IV. Extensions mentioned here are the consequence of dualities between the time and transform domains. Discrete (non-continuous) signals in one domain correspond to periodic signals in the other domain.

2.2.2 DCT of type II

Different boundary conditions greatly influence the applications of the transform and lead to uniquely useful properties for the various DCT types. DCT of type II, also denoted as DCT-II, characterized by excellent energy compaction property and the best approximation to KLT, will be in the scope of this thesis and from now on used for all computations in this work.

Same as with DFT, DCT as a linear transform can be expressed in a matrix notation

$$\boldsymbol{X} = \boldsymbol{C}\boldsymbol{x} \tag{2.10}$$

where $\boldsymbol{x}$ is the input signal, $\boldsymbol{X}$ the output signal, both represented as the column vectors of size N, and $\boldsymbol{C}$ is a transform matrix of size $N \times N$. Due to the properties of cosine function with the frequency $\frac{\pi m}{2N}$ and sampling rate n it can be shown that rows of $\boldsymbol{C}$ are orthogonal. Furthermore, if its rows or vectors are normalized, they can be considered as a set of N orthonormal vectors that constitute the basis of a N-dimensional vector space [12]. After normalization, the DCT-II transform coefficients and reconstructed original signal values are given by:

$$X_m = \sqrt{\frac{2}{N}} w_m \sum_{n=0}^{N-1} x_n cos\left[\frac{\pi}{2N} m(2n+1)\right] \tag{2.11}$$

for $m = 0, \dots, N-1$ and

$$x_n = \sqrt{\frac{2}{N}} \sum_{m=0}^{N-1} w_m X_m cos\left[\frac{\pi}{2N} m(2n+1)\right] \tag{2.12}$$

for $n = 0, \dots, N-1$ where in both equations

$$w_m = \begin{cases} 1/\sqrt{2}, & m = 0 \\ 1, & m = 1, 2, \dots, N-1 \end{cases} \tag{2.13}$$

With analogy to an electrical signal, voltage or current, coefficient X_0 is called the DC coefficient and other coefficients are called AC coefficients.

For visualization purposes, the basis vectors of the transform matrix can be represented using pixel frequencies i.e. color changes along a row. For simplicity, scaling factors from Equation 2.13 are neglected and transform coefficients range from -1 to 1. All values from the range are represented as rectangles painted using grayscale normalized to range {-1,1}. Every basis vector is therefore displayed as an array of N rectangles. Graphic representation for the 4×4 transform matrix is shown in Figure 2.4. N calculated transform coefficients from Equation 2.11 are weights for a weighted sum of basis arrays which yields the data vector itself.

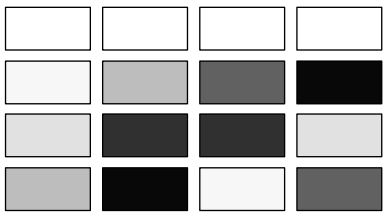


Figure 2.4: Graphic representation of the 1D DCT transform matrix

1D DCT can be used to compress one-dimensional data like an audio signal. When a video signal is considered then the data appear in the two-dimensional structures like a frame or images in the spatial domain or prediction error in both domains. The pixel as a basic information unit in such a data structure resembles all its horizontal neighbours. To make the DCT of such data practical, it is applied to smaller size blocks and in two directions, first to each row of a data block and then to each column of the result. That makes it separable, which is important for the implementation efficiency. 2D DCT for a square-shaped block of size N × N is defined by:

$$X_{m_1 m_2} = \frac{2}{N} w_{m_1} w_{m_2} \sum_{n_1=0}^{N} \sum_{n_2=0}^{N} x_{n_1 n_2} \; cos\left[\frac{\pi}{2N} m_2 (2n_2 + 1)\right] cos\left[\frac{\pi}{2N} m_1 (2n_1 + 1)\right] \tag{2.14}$$

for $m_1, m_2 = 0, \dots, N-1$ and the inverse 2D DCT is defined as:

$$x_{n_1 n_2} = \frac{2}{N} \sum_{m_1=0}^{N} \sum_{m_2=0}^{N} w_{m_1} w_{m_2} X_{m_1 m_2} \; cos\left[\frac{\pi}{2N} m_1 (2n_1 + 1)\right] cos\left[\frac{\pi}{2N} m_2 (2n_2 + 1)\right] \tag{2.15}$$

for $n_1, n_2 = 0, \dots, N-1$, where w_{m_1} and w_{m_2} equal w_m defined in Equation 2.13. Coefficient X_{00} is denoted as the DC coefficient and remaining coefficients are denoted as AC coefficients.

If basis vectors of the 2D transform matrix are visualized using the same approach as for the 1D transform matrix, each square, representing one element of a vector, is rasterized to a $N \times N$ image. Grayscale representation for each image element is obtained from the product of cosine functions which change along two dimensions n_1 and n_2. Figure 2.5 shows the graphic representation of 16 basis squares of the 2D DCT for $N = 4$.

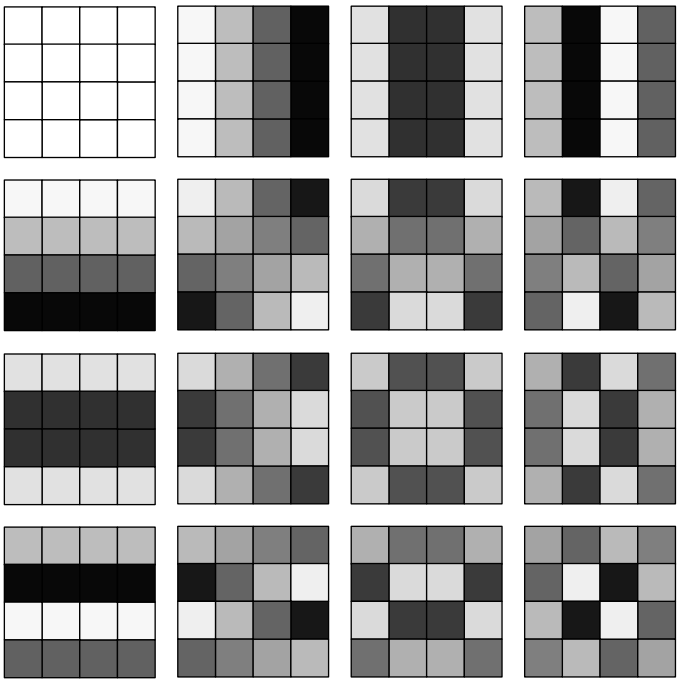
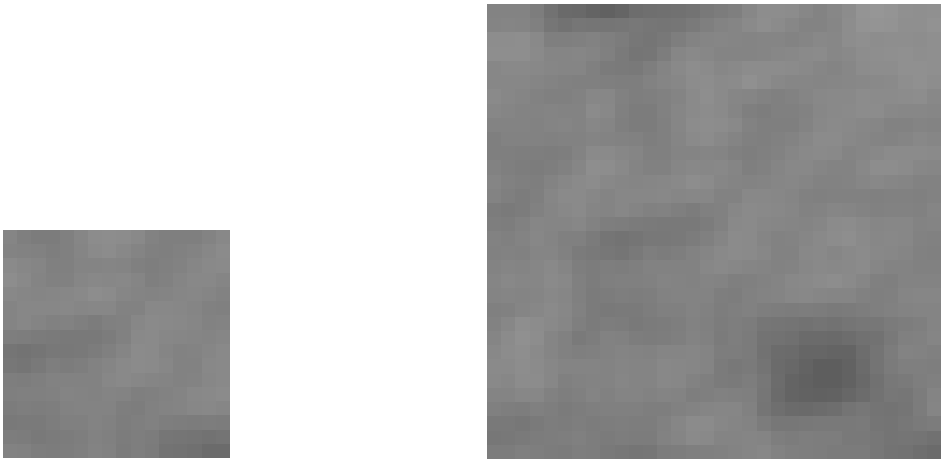


Figure 2.5: Graphic representation of the 2D DCT transform matrix

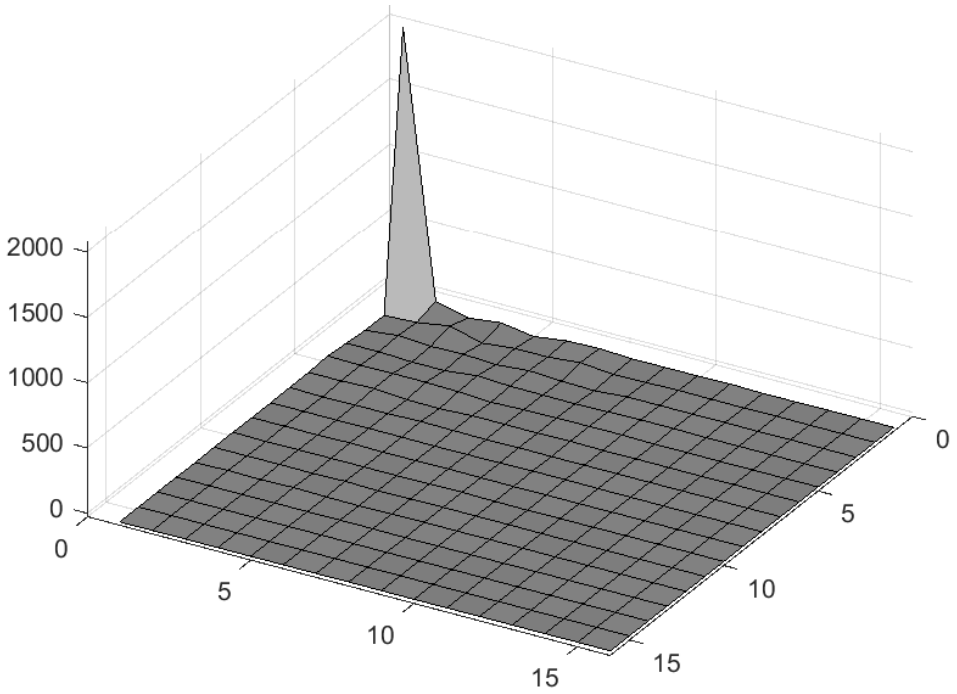
2.2.3 Energy compaction and decorrelation

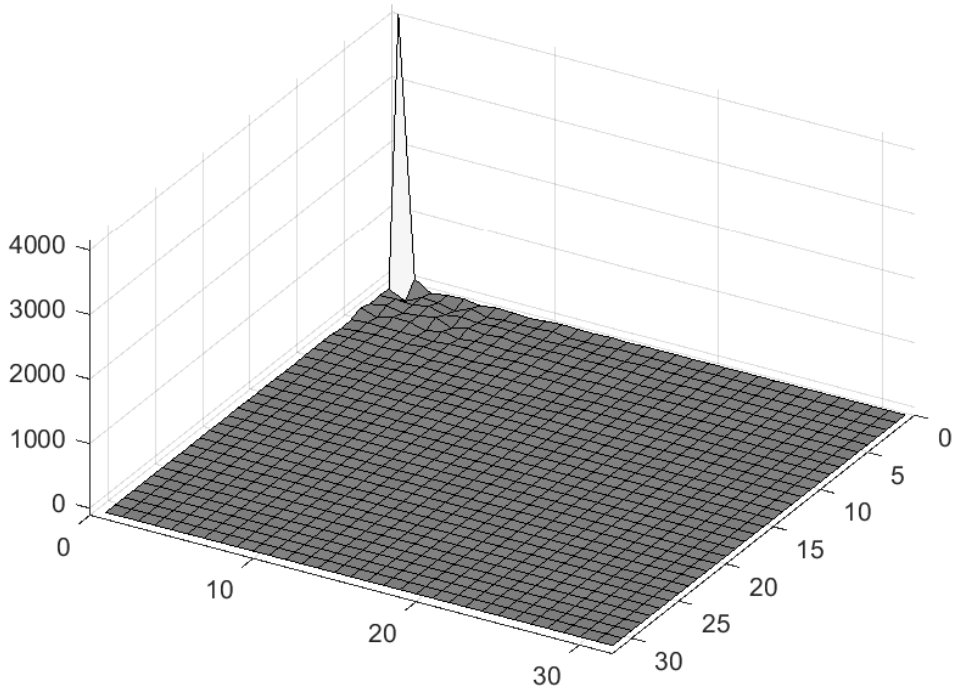
Effective energy compaction and decorrelation are properties that leverage DCT from other transforms and contribute to its wide usage in field of data compression. Scattered signal energy in the spatial domain becomes concentrated in a few coefficients once the signal is transformed. Remaining coefficients will carry very little energy and can be encoded with less fidelity, in a coarse manner compared to the fine encoding of important coefficients. Figure 2.6 illustrates the energy compaction property of the DCT on an example of two blocks with the same centre but of different sizes. The whole frame and its two blocks of sizes 16×16 and 32×32 are shown in Figure 2.6a and Figure 2.6b, while in Figure 2.6c the transform coefficients of the 2D DCT for both blocks are shown.

a

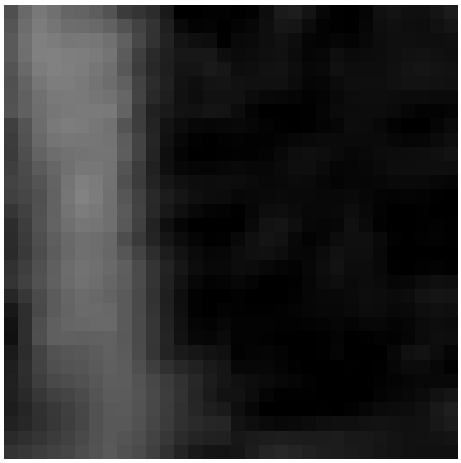


b

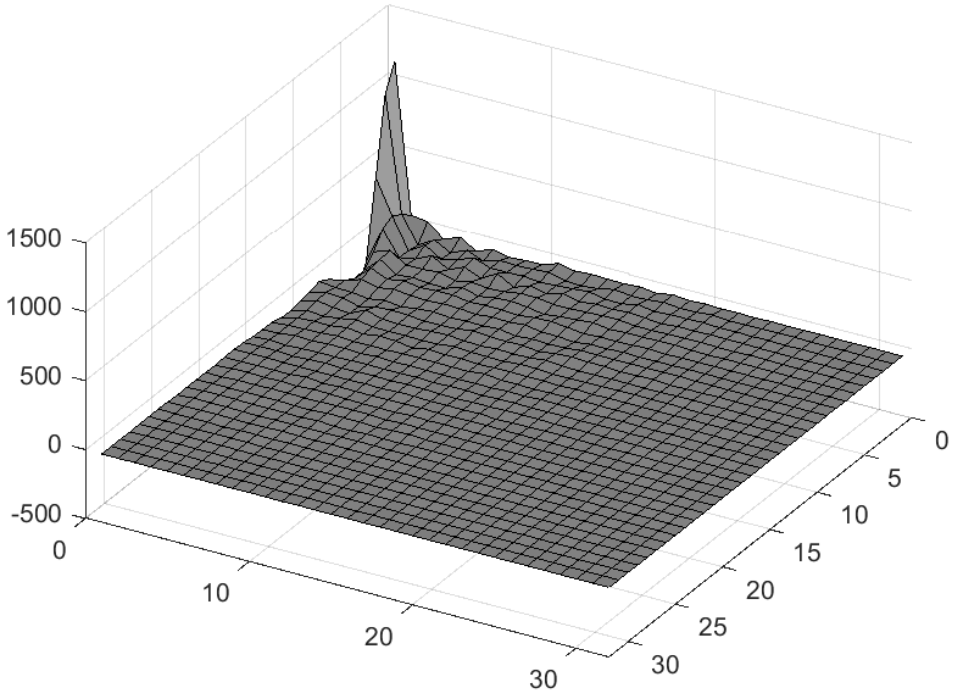

2000
1500
1000
500
0
0
5
10
15
0
5
10
15


4000
3000
2000
1000
0
0
10
20
30
0
5
10
15
20
25
30

c



d



e

Figure 2.6: a) Original 4096 × 1714 frame number 17507 in a video sequence, source [13] b) Frame's two blocks of samples of size 16 × 16 and 32 × 32 having the same centre c) 2D DCT transform of each block d) Difference between blocks at same position in frame number 17507 and 17506 e) 2D DCT transform of block difference

It can be noticed that the pixels in both blocks are highly correlated. In the transform domain, the energy is concentrated (compacted) in the low-frequency part of the spectrum, around the DC coefficient which has the highest value. When moving away from the DC coefficient, towards higher frequency coefficients, their values decrease very quickly exhibiting

successful decorrelation. When block size is bigger DCT is more effective. This performance can be explained with a larger dimension of vector space which is spanned by DCT basis vectors. More frequency components exist, and every pixel or every component of an input vector can be described more precisely using more basis vectors (Figure 2.6c).

In case when the input signal is a prediction error block, which is computed as the difference between a block of pixels at same positions within two neighbouring frames, DCT is less effective. It can be observed by comparing DCT results in Figure 2.6c and Figure 2.6e. A substantial amount of decorrelation is already achieved through block differencing and the transform cannot contribute more to decorrelation like it can for the block of pixels. As a result, heavy quantization will not be effective since it would significantly degrade frame quality. Nevertheless, blocked DCT and subsequent quantization are complementary function blocks to block prediction in video compression systems.

2.3 Quantization

Quantization is the final stage of transform coding. Transform coefficients which were produced during the transform stage are converted into levels. Compared to transform, which is a reversible process, where video data are represented in an another domain, quantization is a lossy, irreversible step. It is important to understand that the energy compaction and decorrelation, obtained with transform, do not involve any data compression. On the other side, quantization discards less important information from the frame and overall video and preserves that which is important. Therefore, this functional block has to be designed considering the characteristics of the input, its range and distribution. Using a dedicated configuration parameter, quantization is usually a primary controlled function inside a video coding system when a desired rate-distortion performance of a system wants to be achieved or exceeded.

Quantization process is depicted in Figure 2.7. Original transform coefficients may have a large number of possible values. That number is limited by implementation features, data types and register sizes. The levels and reconstructed transform coefficients restrain the signal to a discrete set of possible values. The trade-off between coding efficiency and video quality is made through the number of levels which is set as a part of quantizer design. When there are more levels, reconstructed coefficients will be closer to original coefficients, but compression will be low. With less levels on disposal, coefficient precision and video quality decrease but compression gets higher. How the number of levels impacts the blocked transform can be

illustrated when the prediction error block from Figure 2.6d is further quantized and dequantized with both a fine-grained and coarse-grained quantizer. It can be seen in Figure 2.8 that the coarse-grained quantization (with the step size of 32) resulted in a higher percent of block sparsity compared to the fine-grained quantization (with the step size of 4). On the other hand, the former reconstructs the original prediction error more precisely.

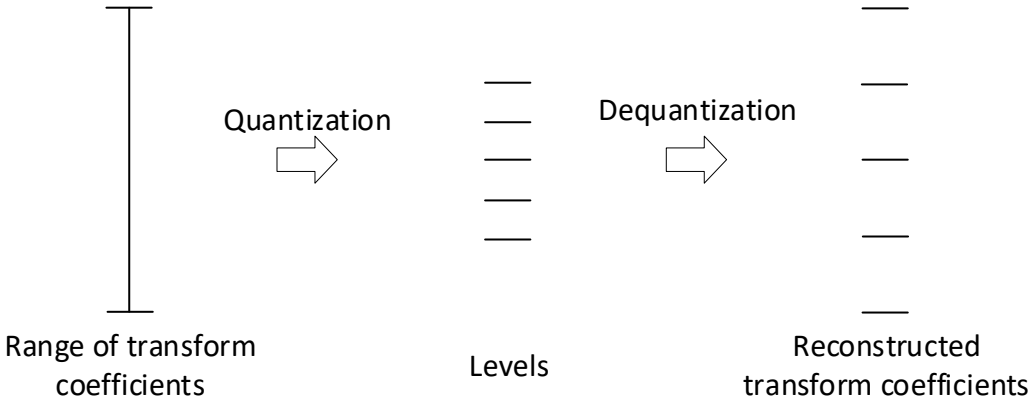


Figure 2.7: Quantization and dequantization

Quantization takes place in a video encoder and dequantization occurs in both the video encoder and decoder. Dequantization in the encoder is part of the frame reconstruction process, which is essential for the subsequent frame predictions. In the decoder, reconstruction aims at frame storage or direct display. Operations performed at the block level end with dequantization. Thereafter the quantized transform coefficients are scanned and included in a stream.

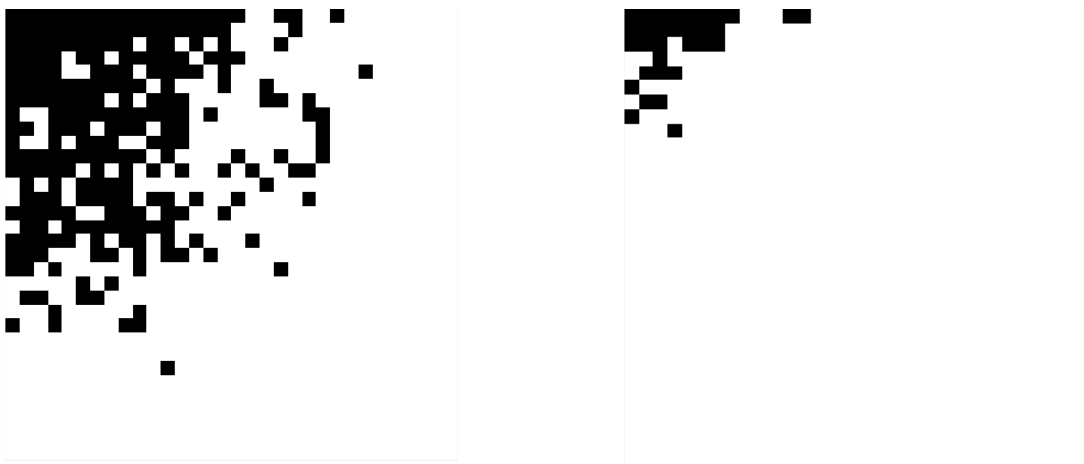


Figure 2.8: Quantized and dequantized transform coefficients with fined-grained quantization (*left*) and coarse-grained quantization (*right*). Nonzero elements are shown with black squares

2.3.1 Quantizer types

There are two major types of quantizers, scalar and vector. Both types are additionally categorized depending on memory usage and input-output characteristics. If quantization of a coefficient is independent of other coefficients, then the quantizer can be categorized as memoryless. Furthermore, depending on the input-output characteristic they can be either symmetric or nonsymmetric.

Scalar quantizers handle input transform coefficients as scalars and convert them to a finite precision representation. They can be uniform or nonuniform. Such classification relates to distances between adjacent boundary values and between adjacent reconstruction values. Choice of a particular quantizer class highly depends on the input signal distribution. A uniform and a common example of a nonuniform quantizer are shown in Figure 2.9.

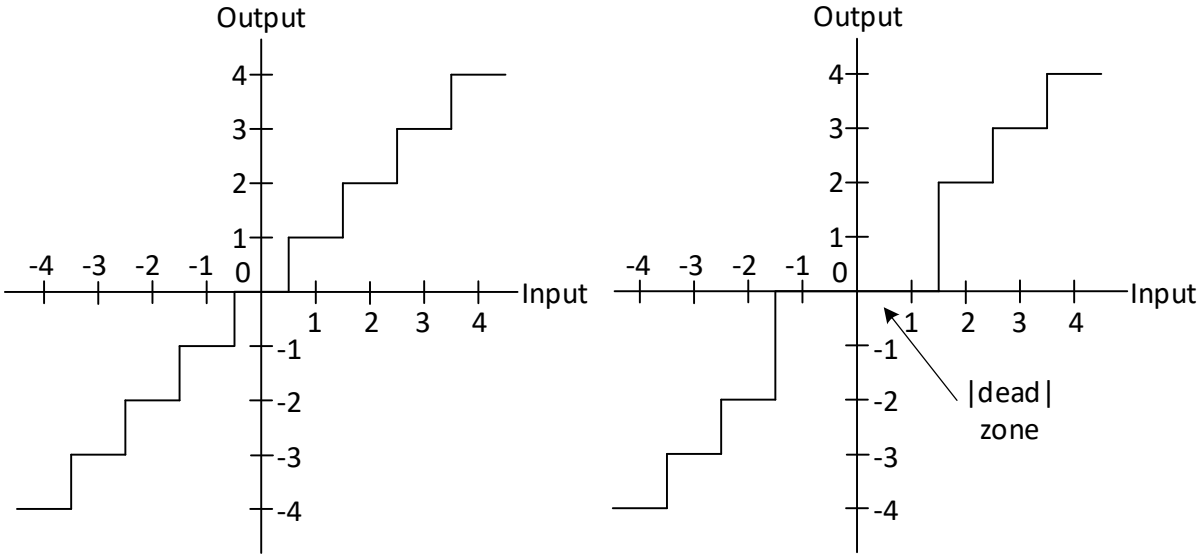


Figure 2.9: Uniform quantizer (*left*) and nonuniform quantizer with a dead zone area (*right*), source [14]

A scalar quantizer can be generally described with the following function:

$$y = Q(x) \tag{2.16}$$

where $Q(\cdot)$ is the quantization function. The K-level scalar quantizer is defined by $k + 1$ decision levels (domain intervals) and k output levels (range intervals). Each interval is referred to as a quantization bin. Quantization error $e(u)$ is defined by:

$$e(u) = Q(u) - u \tag{2.17}$$

This is an essential quantity in specifying the scalar quantizer which determines the level of distortion. The design of the quantizer will tend to minimize the quantization error.

A uniform quantizer has a nonlinear staircase function composed of a set of equally spaced decisions and output levels. Width of the decision interval and the distance between adjacent levels is called *quantization step size*. The choice of a uniform quantizer is appropriate when the input has a uniform distribution. The quantization specified in the modern video coding standards operates on transformed prediction error signals. The residual DCT coefficients are distributed around zero value. The associated coefficient matrix will, therefore, contain a considerable number of values close to zero, either positive or negative, and not many higher values. With such distribution a nonuniform quantizer with the dead zone is a better choice than a uniform quantizer. It will eliminate the near-zero values and keep the higher values.

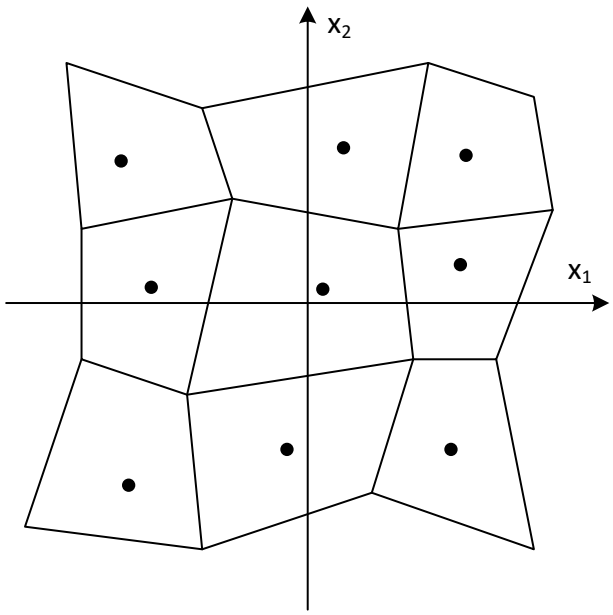


Figure 2.10: Illustration of vector quantization in a two-dimensional vector space, source [15]

In contrast to operating independently on each sample as done in scalar quantization, vector quantization operates on a group of samples. A predetermined set of levels is replaced by a set of vectors. An example of quantization with a two-dimensional vector space is shown in Figure 2.10. The dots inside each quadrilateral represent the reconstruction codeword for all vectors which reside inside it.

Blocks containing transform coefficients are scanned creating vectors of length n which are inputs for the vector quantizer. Quantization is performed in such a way that for every vector $\boldsymbol{X}$, codebook vector representation $\widehat{\boldsymbol{X}}$ is chosen using the closest match criteria. Example for such criteria can be finding the minimum Mean Squared Error (MSE) between vector $\boldsymbol{X}$ and all codebook vectors $\widehat{\boldsymbol{X}}_i$. The information which will be coded into the video stream is the codebook vector index i. On the receiver side, it will be used to decode the vector. The size of the codebook affects coding efficiency and reconstruction quality.

2.4 Transform and quantization in HEVC

Transform and quantization evolved over the years. The driving force behind the improvements was the coding efficiency and efficient implementation for the newest computing architectures. With each new video coding standard, the complexity of the transform computation blocks increased. Allowing different transform block sizes to adapt to varying space-frequency characteristics of the input signal or changing transform basis function to achieve better signal modelling led to this increase. Advances in computing architectures in terms of processing power, which were especially important for high demanding applications like video coding in the real-time, allowed an increase in algorithm complexity.

In subsequent sections transform and quantization computation blocks in HEVC are presented. Emphasis is placed on decisions made during their design process, and the transform and quantization features which contributed to coding efficiency and facilitated the efficient implementation for new and parallel architectures. This will be the basis for the development of scientific contributions in the next chapters, targeting the heterogeneous multiprocessor high-performance computers.

2.4.1 The integer DCT

The original DCT (Equation 2.11) used in digital image, video and audio applications, operates on integer signals and yields real-valued transform coefficients which are subject to subsequent quantization. The cosine function, which is the core function for calculations of transform coefficients, is a source of real numbers. Previous video coding standards H.261, MPEG-1, H.262/MPEG-2 and H.263 specified the 8-point transform with infinite precision. Since this specification is not practical for implementation in digital computers, manufacturers of video codecs had more freedom when designing a DCT computation block. DCT software

or hardware implementations by different manufacturers used different finite-length number representations and rounding. That introduced a drift between encoders and decoders of different manufacturers. To prevent error propagation a block-level periodic intra refresh and accuracy performance test is required. Representing DCT coefficients with finite-length floating-point numbers, even if it would be subject to standardization, inevitably involves round-off errors. These errors distort the DCT's property of orthonormality and therefore compression performance. As a consequence, transformation is not completely lossless anymore.

Considering implementation in hardware, operations with floating-point data are slow, too much memory is being allocated and power consumption is high. Compared to integer multiplication, floating-point multiplication consumes much more time and power leading to larger and more expensive devices. To override these issues and reach faster realization, floating-point factors are scaled up and rounded to integer values. Original coefficients of a DCT matrix are scaled up by a parameter α and approximated by a near integer number:

$$T_N(\alpha, C_x) = round(\alpha \cdot C_x) = T_N(\alpha) \tag{2.18}$$

Parameter α tends to be large to minimize the round-off error and preserve the desirable properties of the DCT matrix. The main problem with a large parameter α is an increase in the dynamic range. Consequently, larger architectures (32 or 64-bit) are needed for implementation.

The DCT with finite-precision approximation of basis vectors is significant for the research community since it simplifies the design of low-power, low-cost compression systems and makes them efficient. This is especially beneficial for battery supplied devices in wireless, satellite or portable computing applications. Various techniques of integer approximation provide the ability to balance between coding efficiency and efficient implementation. Multiplication which consumed a lot of resources and was a performance bottleneck, when made with floating-point data, can be implemented with binary additions and shifts, and it can yield a better performance.

H.264/AVC, H.265/HEVC's predecessor, was the first video coding standard that specified a transform in integer operations. As a trade-off between transform precision and savings in arithmetic bit-width, the scaling parameter α was set to 2.5. The rows of the core transform matrix were orthogonal but didn't have equal norms. This implied the usage of quantization matrices as equal frequency-weighting is expected.

2.4.2 HEVC core transform design

HEVC was being developed at the time when high and ultra-high definition videos were becoming more and more popular. For transform coding of large frame areas with a uniform sample intensity, larger block sizes are supported. The standard [16] specifies transform of sizes 4 × 4, 8 × 8, 16 × 16 and 32 × 32. Integer DCT is used for all transform sizes, prediction modes, and colour components except for the 4 × 4 luma intra prediction. It was found that the discrete sine transform compacts the residual energy more efficiently than the DCT [17].

To obtain the core transform matrix, scaling parameter α was set to $2^{6+M/2}$ where $M = \log_2 N$ and N is the order of the transform matrix. With this setting, each transform matrix element can be represented with 8 bits including the sign bit. Approximation of the integer DCT matrix elements was carried out by prioritizing the real-value DCT properties. Those which reduce the number of arithmetic operations and implementation costs were fully preserved. After scaling, the resulting coefficients were hand-tuned to reach the optimal balance between the orthogonality feature, closeness to the original DCT and equal norm. The 32 × 32 forward transform matrix, symmetric to the right half for even basis vectors and anti-symmetric for the odd ones, with embedded transform matrices for other transform sizes, is shown in Figure 2.11. The embedding property provides reusability of the same hardware architecture for all specified transform sizes, and the small number of unique elements in the matrix, 31 of them in case of the largest transform matrix, means fewer multipliers for such a hardware implementation. Although the orthogonality property is slightly compromised with the approximation process, the transpose of the HEVC transform matrix is defined as its inverse.

As the HEVC transform matrix is the result of upscaling and customized rounding, downscaling is required after each 1D integer transform. To ensure efficient implementation, scale factors are set to a power of two so that they can be implemented as a right shift. Additionally, values of scale factors were calculated and specified in the standard so that the signal dynamic range after each transform can be represented with 16 bits including the sign bit. Adding offset value to minimize the rounding error is specified as an operation before a right shift. The transform and quantization process, with forward and inverse operations is shown in Figure 2.12.

64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64
-90	-90	-88	-85	-82	-78	-73	-67	-61	-54	-46	-38	-31	-22	-13	-4	4	13	22	31	38	46	54	61	67	73	78	82	85	88	90	90
90	87	80	70	57	43	25	9	-9	-25	-43	-57	-70	-80	-87	-90	-90	-87	-80	-70	-57	-43	-25	-9	9	25	43	57	70	80	87	90
-90	-82	-67	-46	-22	4	31	54	73	85	90	88	78	61	38	13	-13	-38	-61	-78	-88	-90	-85	-73	-54	-31	-4	22	46	67	82	90
89	75	50	18	-18	-50	-75	-89	-89	-75	-50	-18	18	50	75	89	89	75	50	18	-18	-50	-75	-89	-89	-75	-50	-18	18	50	75	89
-88	-67	-31	13	54	82	90	78	46	4	-38	-73	-90	-85	-61	-22	22	61	85	90	73	38	-4	-46	-78	-90	-82	-54	-13	31	67	88
87	57	9	-43	-80	-90	-70	-25	25	70	90	80	43	-9	-57	-87	-87	-57	-9	43	80	90	70	25	-25	-70	-90	-80	-43	9	57	87
-85	-46	13	67	90	73	22	-38	-82	-88	-54	4	61	90	78	31	-31	-78	-90	-61	-4	54	88	82	38	-22	-73	-90	-67	-13	46	85
83	36	-36	-83	-83	-36	36	83	83	36	-36	-83	-83	-36	36	83	83	36	-36	-83	-83	-36	36	83	83	36	-36	-83	-83	-36	36	83
-82	-22	54	90	61	-13	-78	-85	-31	46	90	67	-4	-73	-88	-38	38	88	73	4	-67	-90	-46	31	85	78	13	-61	-90	-54	22	82
80	9	-70	-87	-25	57	90	43	-43	-90	-57	25	87	70	-9	-80	-80	-9	70	87	25	-57	-90	-43	43	90	57	-25	-87	-70	9	80
-78	4	82	73	-13	-85	-67	22	88	61	-31	-90	-54	38	90	46	-46	-90	-38	54	90	31	-61	-88	-22	67	85	13	-73	-82	-4	78
75	-18	-89	-50	50	89	18	-75	-75	18	89	50	-50	-89	-18	75	75	-18	-89	-50	50	89	18	-75	-75	18	89	50	-50	-89	-18	75
-73	31	90	22	-78	-67	38	90	13	-82	-61	46	88	4	-85	-54	54	85	-4	-88	-46	61	82	-13	-90	-38	67	78	-22	-90	-31	73
70	-43	-87	9	90	25	-80	-57	57	80	-25	-90	-9	87	43	-70	-70	43	87	-9	-90	-25	80	57	-57	-80	25	90	9	-87	-43	70
-67	54	78	-38	-85	22	90	-4	-90	-13	88	31	-82	-46	73	61	-61	-73	46	82	-31	-88	13	90	4	-90	-22	85	38	-78	-54	67
64	-64	-64	64	64	-64	-64	64	64	-64	-64	64	64	-64	-64	64	64	-64	-64	64	64	-64	-64	64	64	-64	-64	64	64	-64	-64	64
-61	73	46	-82	-31	88	13	-90	4	90	-22	-85	38	78	-54	-67	67	54	-78	-38	85	22	-90	-4	90	-13	-88	31	82	-46	-73	61
57	-80	-25	90	-9	-87	43	70	-70	-43	87	9	-90	25	80	-57	-57	80	25	-90	9	87	-43	-70	70	43	-87	-9	90	-25	-80	57
-54	85	4	-88	46	61	-82	-13	90	-38	-67	78	22	-90	31	73	-73	-31	90	-22	-78	67	38	-90	13	82	-61	-46	88	-4	-85	54
50	-89	18	75	-75	-18	89	-50	-50	89	-18	-75	75	18	-89	50	50	-89	18	75	-75	-18	89	-50	-50	89	-18	-75	75	18	-89	50
-46	90	-38	-54	90	-31	-61	88	-22	-67	85	-13	-73	82	-4	-78	78	4	-82	73	13	-85	67	22	-88	61	31	-90	54	38	-90	46
43	-90	57	25	-87	70	9	-80	80	-9	-70	87	-25	-57	90	-43	-43	90	-57	-25	87	-70	-9	80	-80	9	70	-87	25	57	-90	43
-38	88	-73	4	67	-90	46	31	-85	78	-13	-61	90	-54	-22	82	-82	22	54	-90	61	13	-78	85	-31	-46	90	-67	-4	73	-88	38
36	-83	83	-36	-36	83	-83	36	36	-83	83	-36	-36	83	-83	36	36	-83	83	-36	-36	83	-83	36	36	-83	83	-36	-36	83	-83	36
-31	78	-90	61	-4	-54	88	-82	38	22	-73	90	-67	13	46	-85	85	-46	-13	67	-90	73	-22	-38	82	-88	54	4	-61	90	-78	31
25	-70	90	-80	43	9	-57	87	-87	57	-9	-43	80	-90	70	-25	-25	70	-90	80	-43	-9	57	-87	87	-57	9	43	-80	90	-70	25
-22	61	-85	90	-73	38	4	-46	78	-90	82	-54	13	31	-67	88	-88	67	-31	-13	54	-82	90	-78	46	-4	-38	73	-90	85	-61	22
18	-50	75	-89	89	-75	50	-18	-18	50	-75	89	-89	75	-50	18	18	-50	75	-89	89	-75	50	-18	-18	50	-75	89	-89	75	-50	18
-13	38	-61	78	-88	90	-85	73	-54	31	-4	-22	46	-67	82	-90	90	-82	67	-46	22	4	-31	54	-73	85	-90	88	-78	61	-38	13
9	-25	43	-57	70	-80	87	-90	90	-87	80	-70	57	-43	25	-9	-9	25	-43	57	-70	80	-87	90	-90	87	-80	70	-57	43	-25	9
-4	13	-22	31	-38	46	-54	61	-67	73	-78	82	-85	88	-90	90	-90	90	-88	85	-82	78	-73	67	-61	54	-46	38	-31	22	-13	4

Figure 2.11: The 32 × 32 matrix specifying the 32-point forward transform. Forward transform matrices for the remaining transform sizes specified in the standard 16x16 (solid border), 8x8 (double border) and 4x4 (gray shading) are embedded

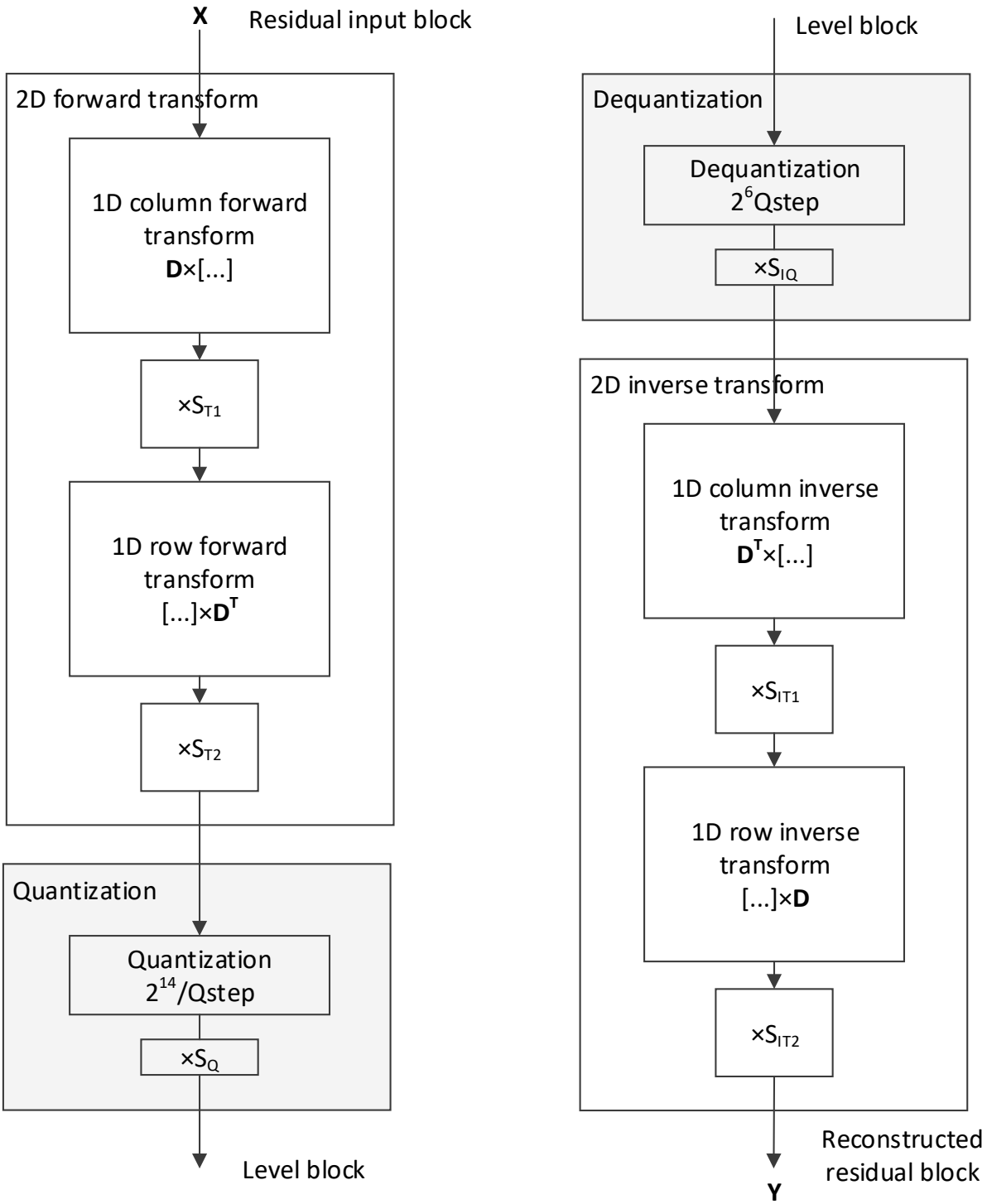


Figure 2.12: The HEVC forward transform and quantization (*left*) and HEVC inverse transform and dequantization (*right*). D is the HEVC transform matrix, the scaled approximation of the DCT matrix. Scaling factors appear after each transform and quantization step to ensure equal norm and specified dynamic range

As inverse transform and dequantization are used in the reconstruction of a referenced frame, which exists in both encoder and decoder, its specification is in the scope of the standard. It is left to the implementer to select or design its own forward transform and quantization.

2.4.3 Quantization and dequantization

Quantizer specified in HEVC is of a uniform scalar type. The relation between the quantization step size $Qstep$ and quantization parameter QP, which can be configured by a user, is given by:

$$Qstep(QP) = G_{QP\%6} \ll \frac{QP}{6} \tag{2.19}$$

where

$$\boldsymbol{G} = [G_0, \dots, G_5\,]^T = \left[2^{-4/6}, 2^{-3/6}, 2^{-2/6}, 2^{-1/6}, 2^0, 2^{1/6}\right]^T \tag{2.20}$$

and QP is an integer value in the range of 0 to 51. When its value increases by 6, $Qstep$ doubles and when it's configured to 4, $Qstep$ becomes 1. Since values from Equation 2.19. are real numbers, they are, similarly to a real-valued transform, scaled up and rounded to the nearest integer. Such upscaling modifies the norm of the residual block and additional multiplication by a scale factor is required to restore the norm. The corresponding factors S_Q and S_{IQ} are indicated in Figure 2.12.

Though $Qstep$ is a divisor value in case of quantization and a multiplicand value for dequantization, the fixed-point approximation is so designed, that upscaling before rounding is carried out in both the quantizer and dequantizer. Related upscaling factors are 2^{14} and 2^6 for the quantizer and dequantizer respectively.

As quantization is a configurable lossy operation, HEVC provides additional tools for this computation block in video coding. Quantization granularity can be differed, depending on frequencies, using quantization matrices. $Qstep$ can be changed dynamically within a picture for the sake of rate control or perceptual quantization. To achieve bitrate savings the QP prediction process is developed at the level of quantization groups, which comprise more coding units (CUs), so that only the delta QP is transmitted if it exists [18].

3 A PERFORMANCE EFFICIENT HEVC TRANSFORM ARCHITECTURE

3.1 Introduction

Due to its properties, useful both for compression efficiency and efficient implementation, the discrete cosine transform (DCT) is widely used in video and image compression. The HEVC standard, with significant improvements in compression performance compared to its predecessors, uses the finite precision approximation of transformation as described in Section 2.4.2. In overall encoding time, transform and quantization takes about 25% for the all-intra and 11% for the random access configurations [19]. To ensure higher compression of the ultra-high definition (UHD) video resolutions the standard introduced additional transform sizes at the cost of processing and implementation complexity. That is why efficient implementation, set as a design goal during the HEVC transform development [20][21][22], significantly impacts the overall performance of the codec.

HEVC transform architectures are mostly based on the parallel butterfly computation structure to obtain maximum sharing of internal operations. Most of the architectures assume that all the elements of a residual vector are transferred to the input in the same cycle. In the case of a video with an 8-bit color depth, 9 bits are required for residual value representation. 288 bits are needed if one residual vector for the biggest transform block size is transmitted at once, or 512 bits if residuals are byte-aligned. Therefore, these architectures raise issues of huge peak access bandwidth and hardware overhead. In modern Systems on Chips (SoCs), especially in their budget versions, hardware resources and communication interface bit widths are limited. Moreover, considering performance requirements of a real HEVC applications, parallel architectures are unnecessary. For example, the percentage of 32 × 32 and 16 × 16 block sizes used for transform is less than 2% and 10% respectively [23]. If the designed hardware outputs 16 transform coefficients in each cycle in parallel, the operating frequency with 4096 × 2048@30fps real-time application is calculated from the following equation:

$$Frame\ width\ \times Frame\ height \times Video\ format \times Frame\ rate$$
$$= Throughput \times Operating\ frequency$$

nearly as low as 7.5 MHz. Such high performance, low-power architecture is obtained at the cost of a high throughput and large hardware overhead.

This chapter is organized as follows. Section 3.2 describes the integer DCT with a parallel butterfly computation structure. Section 3.3 provides an overview of the general architecture of the integer DCT computation block. Research directions and scientific contributions related to the topic of HEVC transform architecture are outlined in Section ., while Section 3.5 describes the development environment and methodology for efficiency validation of performance-optimized hardware implementation. In Section 3.6, an area efficient and reusable HEVC transform architecture for systems with limited resources [24] is presented which can receive two byte-aligned residuals (total of a 32-bit bus width) and output two scaled intermediate 16 bit transform coefficients in one cycle. Improving area efficiency is stressed as the main design goal. Section 3.7 demonstrates and discusses evaluation and comparison results.

3.2 The mathematical model for hardware implementation

The HEVC 2D integer DCT transform of size $N \times N$ ($N = 4, 8, 16, 32$) is given by:

$$\boldsymbol{Y} = \boldsymbol{D} \times \boldsymbol{X} \times \boldsymbol{D}' \tag{3.1}$$

where $\boldsymbol{D}$ is the HEVC transform matrix with constant values, $\boldsymbol{X}$ the residual matrix of size $N \times N$ and $\boldsymbol{D}'$ is a transpose of the transform matrix. When properties of the transpose operator are considered, Equation 3.1 can be rewritten as:

$$\boldsymbol{Y} = (\boldsymbol{D} \times (\boldsymbol{D} \times \boldsymbol{X})')' \tag{3.2}$$

In this equation, it can be noticed that the matrix **D** is twice used in the multiplication. Compared to Equation 3.1 the basis vectors with constant values in both multiplications are multiplicands that can be mapped in the hardware to a single multiply-accumulate block. Its input will be a series of residual vectors when implementing inner multiplication and a series of intermediate vectors, which can be brought to the input of the same block as feedback. This property known as transform separability can be used to design hardware architecture for a 2D integer DCT by utilizing the same 1D integer DCT core and applying transpose to the result matrices.

Since elements of the even basis vectors of the transform matrix are symmetric and the odd basis vectors antisymmetric, the decomposition of the 1D integer DCT matrix multiplication can be made as follows:

$$\begin{bmatrix} Y_0^N \\ Y_2^N \\ \vdots \\ Y_{N-2}^N \end{bmatrix} = \boldsymbol{D}^{\frac{N}{2}} \times \begin{bmatrix} a_0^N \\ a_1^N \\ \vdots \\ a_{\frac{N}{2}-1}^N \end{bmatrix} \tag{3.3}$$

$$\begin{bmatrix} Y_1^N \\ Y_3^N \\ \vdots \\ Y_{N-1}^N \end{bmatrix} = \boldsymbol{B}^{\frac{N}{2}} \times \begin{bmatrix} b_0^N \\ b_1^N \\ \vdots \\ b_{\frac{N}{2}-1}^N \end{bmatrix} \tag{3.4}$$

where

$$a_i^N = X_i^N + X_{N-1-i}^N \tag{3.5}$$

$$b_i^N = X_i^N - X_{N-1-i}^N \tag{3.6}$$

for i = 0, 1, 2, ..., $N/2 - 1$. $\boldsymbol{X}$ is the input residual column vector, $\boldsymbol{Y}$ is the N-point 1D integer DCT of $\boldsymbol{X}$, $\boldsymbol{D}^{N/2}$ is the $N/2$-point HEVC transform matrix and $\boldsymbol{B}^{N/2}$ is the matrix of size $N/2 \times N/2$ defined as:

$$\boldsymbol{B}_{i,j}^{N/2} = \boldsymbol{D}_{2i+1,j}^N \tag{3.7}$$

for i, j = 0, 1, 2, ..., $N/2 - 1$. Matrix $\boldsymbol{D}^{N/2}$ from Equation 3.3 can be further decomposed to $\boldsymbol{D}^{N/4}$ and $\boldsymbol{B}^{N/4}$ in the same way. Decomposition is applicable for all transform sizes supported in the HEVC standard. By observing the elements of the matrix $\boldsymbol{B}^N$ it can be seen that every row contains the same unique values with a different distribution. Reusability as a consequence of decompositions and a small number of unique elements will decrease implementation costs.

The computational complexity calculation shows significant savings for this algorithm, referred to as partial butterfly in [22]. For a straightforward ($N \times N$)-point transform, the number of operations is as follows:

$$O_{mult} = 2N^3$$
$$O_{add} = 2N^2(N-1)$$

whereas using the partial butterfly it is reduced to:

$$O_{mult} = 2N\left(1 + \sum_{k=1}^{\log_2 N} 2^{2k-2}\right)$$

$$O_{add} = 2N\left(\sum_{k=1}^{\log_2 N} 2^{k-1}(2^{k-1}+1)\right)$$

Table 3.1 shows how the number of operations depends on the transform size and algorithm type.

Table 3.1: Number of operations for two transform algorithms

Transform size	Straightforward transform		Partial butterfly algorithm	
	Multiply	Add	Multiply	Add
4	128	96	48	64
8	1024	896	352	448
16	8192	7680	2752	3200
32	65536	63488	21888	23808

3.3 2D integer DCT architecture

Thanks to transform separability described in Section 2.2.2, an ($N \times N$)-point 2D integer DCT can be calculated by the row-column decomposition method in two distinct transform stages. In the first stage an N-point 1D integer DCT is calculated for each column of the input residual matrix of size $N \times N$ to produce an intermediate output matrix of the same size. In the second stage, each row of the intermediate matrix is used for the computation of an N-point 1D integer DCT to produce a 2D integer DCT matrix of size N × N. This matrix contains transform coefficients.

The 2D integer DCT architecture can be implemented using the folded or full-parallel structure. The selected implementation affects performance and determines resource usage. Computation of an ($N \times N$)-point 2D integer DCT using the folded structure is shown in Figure 3.1. The folded structure contains one N-point 1D integer DCT computation block, and N multiplexers, which serve as switches between transform stages and transposition buffer. Data are computed in the DCT computation block and written column-wise to the transposition buffer during the first transform stage and read during the second stage row-wise. Each stage lasts for N cycles. The transform of a new residual block can not start before both transform stages for the current residual block are finished. So, this structure introduces a pipeline latency of $2N$ cycles.

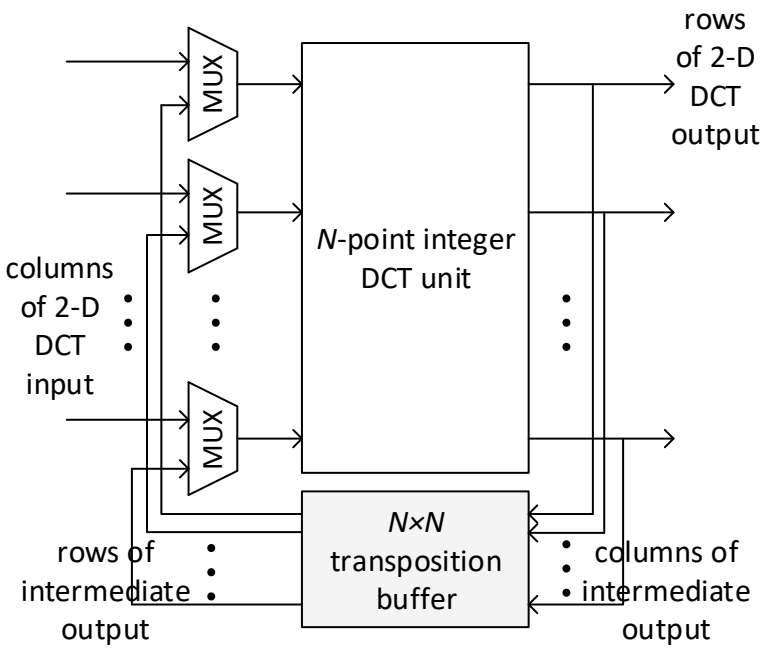


Figure 3.1: The folded structure of a ($N \times N$)-point 2D integer DCT implementation, source [25]

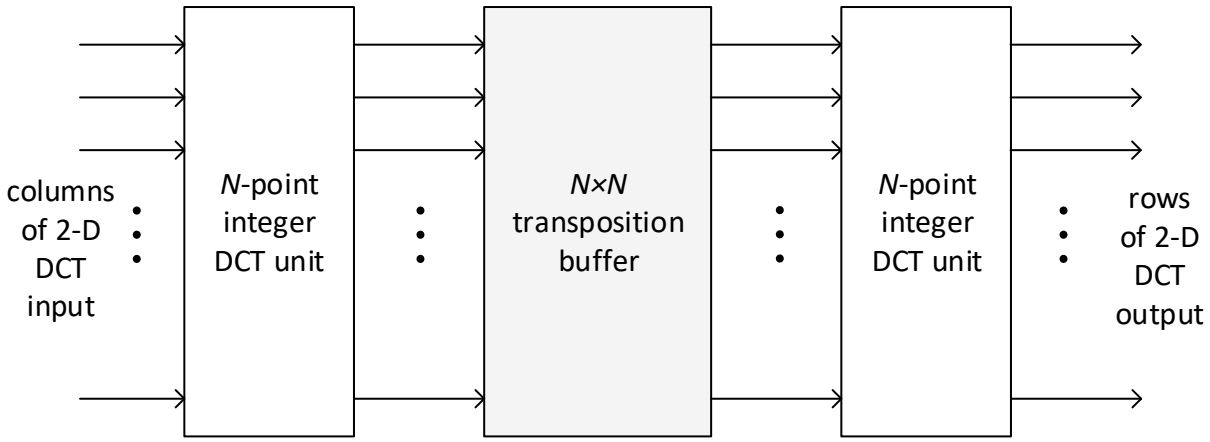


Figure 3.2: The full-parallel structure of a ($N \times N$)-point 2D integer DCT implementation, source [25]

Computation of ($N \times N$)-point 2D integer DCT using a full-parallel structure is shown in Figure 3.2. The full-parallel structure contains two N-point 1D integer DCT computation blocks and a transposition buffer. Compared to the transposition buffer in the folded structure, this transposition buffer has to be able to support alternating column-wise and row-wise read and write operations. During the first transform stage, intermediate transform coefficients are stored column-wise during N cycles to the buffer. In the next N cycles, which belong to the second transform stage, they are read row-wise and transformed sequentially by the second N-point 1D integer DCT computation block. Concurrently, the first N-point 1D integer DCT computation block fills its newly computed intermediate data, now row-wise, to the buffer. So,

in one stage both operations, read and write, are performed in the transposition buffer. In the case of the full-parallel structure, the transposition buffer introduces a pipeline latency of *N* cycles. That period is used to load the buffer at the start of the operation.

Transposition buffer in both structures can be implemented using registers or the dual-port on-chip Static Random Access Memory (SRAM). The former presents a power-efficient solution due to possible clock gating [66]. The full-parallel structure consumes more power than the folded structure since it has two 1D integer DCT computation blocks with approximately the same complexity of the transposition buffer. Due to differences in pipeline latencies, the full-parallel structure can obtain double the throughput of the folded structure.

3.3.1 1D integer DCT architecture

Conventional architecture for *N*-point 1D integer DCT, which is the core computation block for both implementation structures, is shown in Figure 3.3.

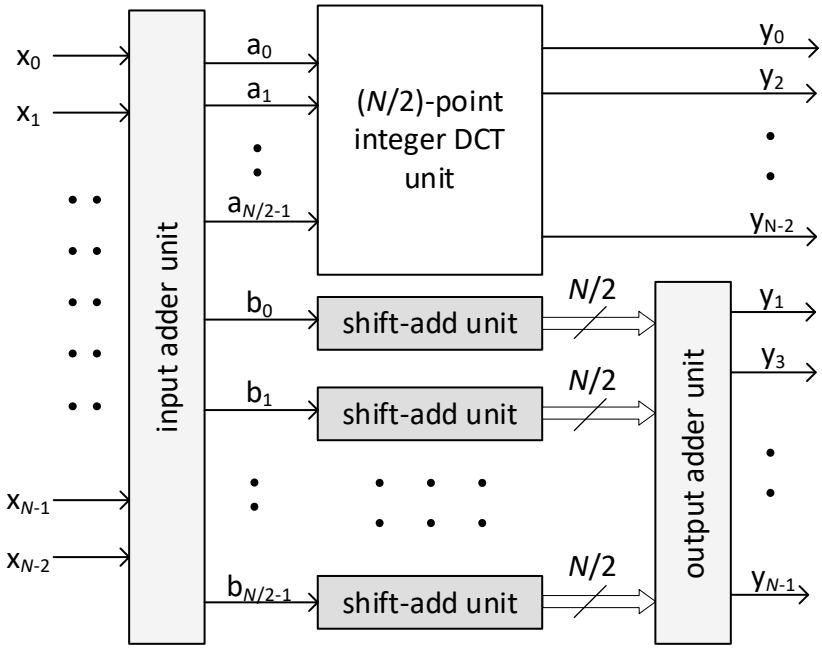


Figure 3.3: Conventional architecture for an 1D integer DCT of lengths N = 8, 16 and 32, source [25]

Inside the input adder unit, symmetric elements of transform block's column vector are both summed and subtracted (Equations 3.5 and 3.6). *N*/2 subtracted values are multiplied with all the elements belonging to the left half of the transform matrix's odd rows. Since absolute values of coefficients, which appear in odd rows of a transform matrix, are the same, with only a difference in their positions in each row, the same circuit is instantiated *N*/2 times. The output adder unit which has the structure of a binary adder tree will sum up all *N*/2 summands from

corresponding shift-add units' outputs to compute transform coefficients in odd rows of an output column vector.

N/2 summation results from the input adder unit are fed to a *N*/2-point 1D integer DCT unit which computes transform coefficients in remaining even rows of the output column vector, using the same units which operate with double fewer elements. The butterfly computation structure of an 8-point 1D integer DCT is shown in Figure 3.4. It can be seen that the 4-point integer transform is embedded in the structure and its results present even transform coefficients of the 8-point integer transform. Generally, the results of smaller block sizes transformations are recursively used for larger block sizes transforms.

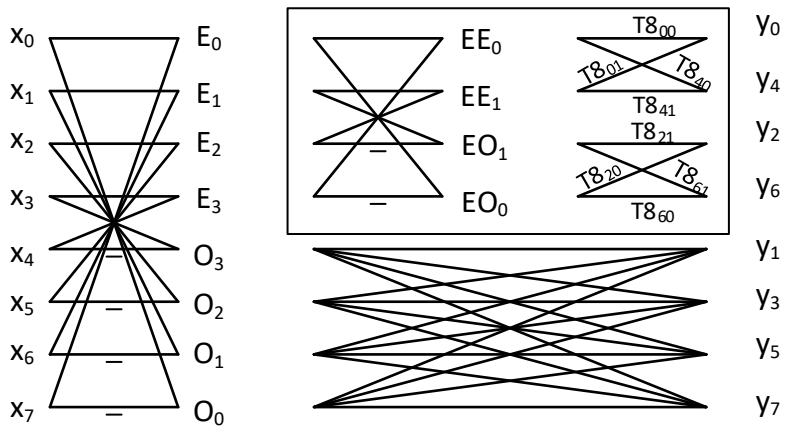


Figure 3.4: The butterfly computation structure of an 8-point 1D DCT

3.4 Overview of scientific contributions to the integer DCT architecture

Architecture improvements in terms of performance increase or obtained resource savings, made from the scientific community, can be classified according to four distinct design approaches:

- Optimization of the shift-add unit
- Pipelined design of the overall circuit
- A unified architecture for the forward and inverse transform
- Resource savings through accuracy decreases

Reusability of the $N/2$-point 1D integer DCT architecture for the implementation of the N-point 1D integer DCT is identified in [26] where $N = 2^M$ and M is an integer. The N-point integer DCT architecture, as already described in Section 3.3.1, consists of a partial butterfly

unit, Multiple-Constant-Multiplication (MCM) units, built from shift and add units, adder units and a $N/2$-point integer DCT unit. Adder units output odd transform coefficients and a recursive $N/2$-point integer DCT unit outputs even transform coefficients of a N-point integer DCT. That architecture was proposed as an alternative to the resource-consuming direct implementation of matrix multiplication in hardware.

Transform matrix decomposition, when calculating even and odd coefficients and hardware reusability was exploited in [25] as well. Additionally, a constant throughput for every transform size was achieved by adding an additional $N/2$-point integer DCT unit. In the same work, the full integer 2D DCT architecture was derived from the integer 1D DCT architecture. That is possible due to the row-column separability of the transform. Two architectures were proposed, folded and full-parallel. Constant throughput was achieved in [27] with the sharing of MCM and adder units for different transform sizes with added multiplexers for the transform size selection as a design overhead. With this technique, additional units were avoided, and better resource utilization was achieved. Usage of MCM units was investigated in [28] and [29] in order to avoid the high latency of the hardware multipliers. In [28] subtractors were not used in the design because they use more hardware resources and have a lower operating frequency. MCM and adder units were designed by using 3-input adders in addition to 2-input adders in order to further decrease resource consumption. In [29] optimization of MCM units was performed considering the critical signal path and signal bit-width. The Hcub algorithm for generating MCM networks with the least possible adders in the critical path was used. Bit-width optimization was conducted on adders' inputs and output inside MCM circuits. The minimum number of bits was selected to represent the adder output value and input bit-widths were aligned with that number. Such a design approach is beneficial to the latency of adder unit and area usage. To decrease design and verification time and improve design reusability, the usage of High-Level Synthesis (HLS) is proposed in [30]. It is a C and C++ based pipelined design in which the even-odd decomposition stages were mapped to the adder tree, Digital Signal Processor (DSP) blocks and accumulator during synthesis respectively. Two architectures were designed for the 8, 16 and 32-point transform, low-cost and high-speed architecture. In a low-cost variant, N residuals are processed in parallel. High-speed architecture has a constant throughput of 32 coefficients per cycle ($32/N$ columns of a residual block are received in a single cycle). Separately, the architecture for a 4-point DCT/DST is designed. Since management and control of residual quadtree partitioning are usually not in the scope of

transform architectures, [31] proposes a flexible input architecture and a supporting configuration encoding scheme. All possible combinations of transform blocks and residuals are mapped to hardware sets using the encoding scheme and ensuring constant throughput. During the HEVC encoding rate-distortion optimization (RDO) every possible combination of prediction blocks is considered, and it is necessary, before a split decision, to carry out both the forward and inverse transform. In [32] two unified pipelined architectures for the forward/inverse integer DCT are proposed. Compared to competing proposals, savings in area usage are not made at the cost of throughput performance. There are also proposals where transform accuracy is reduced to save hardware resources. In [25] resource savings are made through pruning the least significant bits in MCM and adder units. This can be seen as an integration of the scaling step with the integer transform. This affects rate-distortion performance [33] insignificantly but brings a reduction in area and power. In [34] a hardware-oriented implementation of an Arai-based DCT is proposed and sinusoidal multiplication constants are approximated (scaled, rounded and right-shifted) with a configurable number of representation bits. As this number is smaller, gate count and power consumption are lower and the clock frequency is higher.

3.5 Development environment and methodology

Due to availability and prices of integrated development environments, FPGA is selected as the target hardware platform for the validation of the own design of the 1D integer DCT architecture, which is the core computation block in HEVC transform coding. To make an extensive validation, several scientific works, which targeted the same hardware platform, were selected and corresponding results were taken for comparison. Other scientific works whose research objectives relate to the integer DCT or its inverse, no matter which type of VLSI circuit they target, were also considered and analysed to gain insight into design principles and features of modern transform circuits for video coding.

There are two leading manufacturers of FPGA devices, Xilinx and Intel Altera. Both are approximately equally represented in related scientific works. The architecture of logic units, basic building blocks that implement logic functions, and routing architecture of these FPGAs are different. This leads to different performance and utilization results once a design is implemented. The prepared design is therefore synthesized and implemented using an integrated development environment from both manufacturers. As the primary tool, the Vivado

Integrated Development Environment (IDE) was taken. Once the design is verified and implemented in Vivado IDE, the design files are compiled again in Intel Quartus Prime. Table 3.2 describes the targeted device families, and the development tools used for the validation activities.

Table 3.2: Setup of the development environment

Manufacturer	Device family	Development tool	Builds
Xilinx	Virtex 7	Vivado v2016.4 (64-bit)	SW Build 1756540 IP Build 1755317
Intel Altera	Stratix V	Quartus Prime Standard Edition 18.1.0	Build 625

This design is not transferred to a physical FPGA device and the design flow ends with net routing which precedes bitstream generation. Design methodology applied to the FPGA design in this thesis, which is aligned with the manufacturer's *UltraFAST* design methodology guide [35], is shown in Figure 3.5. At the beginning of the process, the application design description is developed in a Hardware Description Language (HDL). In parallel with the development of the Register Transfer Level (RTL) model, a test bench is developed for every submodule to verify its function. After every submodule and top module passed the simulation, the synthesis could be run. There are two types of inputs that have to be available for synthesis. Except for a valid RTL source code, timing constraints have to be defined. This is done in a separate file. For this work such a file contains the single Tool Command Language (Tcl) command [36] as follows:

```
create_clock -period <clock period> -name <clock name> -waveform <clock
edge specification> [get_ports <clock name>]
```

As improving area efficiency is stressed as the main design goal, the synthesis and implementation strategies (Vivado) and the optimization modes (Quartus) are configured accordingly. In case of Vivado this is carried out with a Tcl synthesis command as follows:

```
synth_design -top <top module name> -part <target part> -directive
AreaOptimized_high -mode out_of_context
```

The last two command options define overrides from the `Vivado Synthesis Defaults (Vivado Synthesis 2016)` preconfigured synthesis strategy. In the tool evaluation phase, it has been shown that compared to two other strategies, the `Flow_AreaOptimized_high (Vivado Synthesis 2016)` and `Flow_PerfOptimized_high (Vivado Synthesis 2016)`,

this setting yields the best trade-off between resource utilization, power and timing. Since the physical FPGA device is not the result of the design process, design mode `out_of_context` is configured. It enables the implementation of the module without IO buffers, prevents optimization due to unconnected inputs or outputs, and appropriately adjusts the Design Rule Check (DRC) rules for the design.

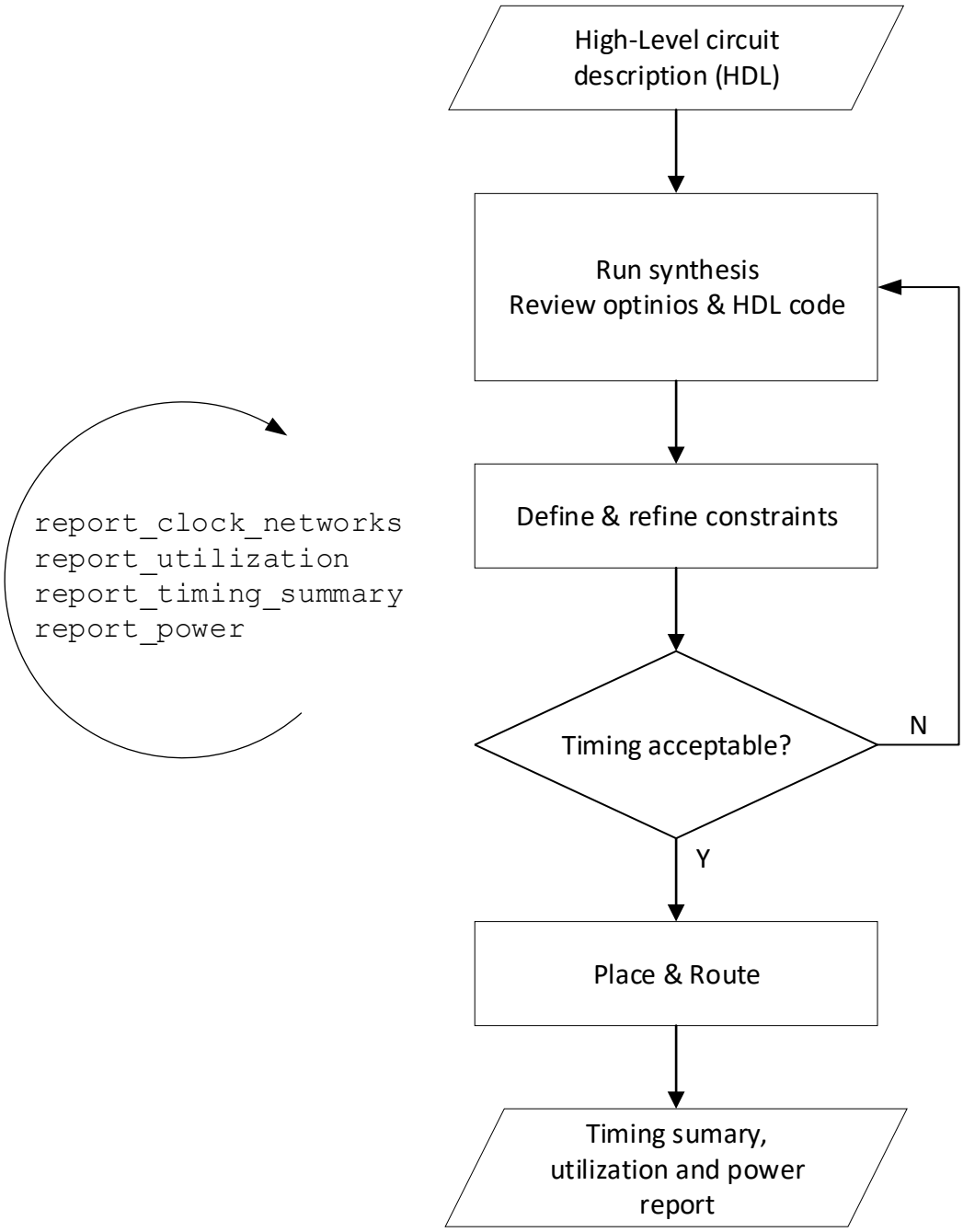


Figure 3.5: Design methodology

The Vivado IDE also offers a set of pre-defined different strategies to use for the implementation run. Seven of them were selected during the evaluation phase and applied to

the implementation of the 32-point 1D integer DCT architecture. The best trade-off between resource utilization and performance was achieved for the `Performance_NetDelay_high` strategy. To provide better understanding of made compromises, results are shown in Table 3.3 for two combinations of synthesis and implementation strategies for the tested design. The clock period was set to 10 ns. The first strategy combination can be characterized as cost-effective and second as performance effective. The latter yields about 1.2 times higher throughput at the cost of 28% more LUTs and 4% more registers. The former is therefore selected as the referent one for all subsequent runs during the design process.

Synthesis strategy	**Implementation strategy**	**LUTs**	**Registers**	**Power (mW)**	**WNS (ns)**
`Vivado Synthesis Defaults`	`Vivado Implementation Defaults, -directive = AreaOptimized_high`	7196	5512	224	2.74
`Flow_PerfOptimized_high`	`Performance_NetDelay_high`	9185	5740	235	3.81

Table 3.3: Implementation results for the two boundary combinations of strategies

In the case of Quartus Prime, the development environment synthesis and implementation strategy selection are combined into the *Optimization mode* selection inside the *Compiler settings* menu. The `Aggressive Area` mode is selected here. It instructs the compiler to target an area minimal solution, even if this reduces the overall timing performance.

By having a verified RTL model and constraint file available and appropriate settings made, the synthesis run [37] can start. The synthesis result is an optimized netlist that is functionally equivalent to the RTL. As an additional result of the synthesis, various reports are generated. Timing closure in this work consists of the design meeting positive timing margin of Worst Negative Slack (WNS). If this timing requirement is not met, a new design iteration starts which includes result analysis, design modification and constraint modification (Figure 3.5).

Results of timing analysis made during the synthesis run contain some estimations of path delays. If timing is not met but the reported WNS negative timing margin has a relatively low value, the process can go ahead. Implementation tools could make this timing margin positive if they allocate the best resources to the failing paths. Implementation [38] contains all steps required to place and route the netlist onto the FPGA device resources while satisfying

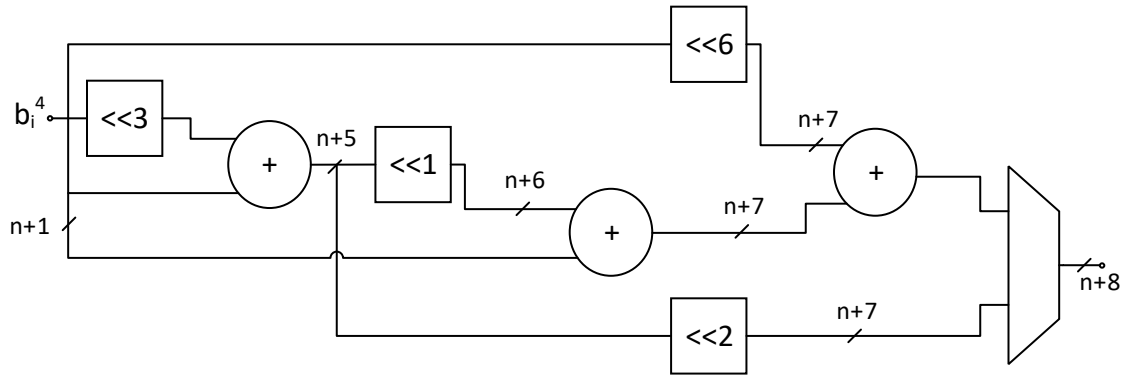
the design's logical and timing constraints. In this design step the logical netlist, output of the synthesis step, is mapped into the physical array of the target device. The implementation comprises logic optimization, placement of logic cells and routing of the connection between cells.

To calculate the maximal operating frequency of the design, the timing summary report is being referred to. At the top of the *Max Delay Paths* section of the report, the signal path with WNS is listed. Its value represents the timing margin for the clock signal. Subtracting this value from the clock time period, specified in the constraint file, yields a minimal time period of the clock which then, in turn, defines the maximum operating frequency. Design throughput can be calculated by multiplying the maximum frequency with the pixel processing rate. Considering this is a pipelined design, pipeline latency is noted.

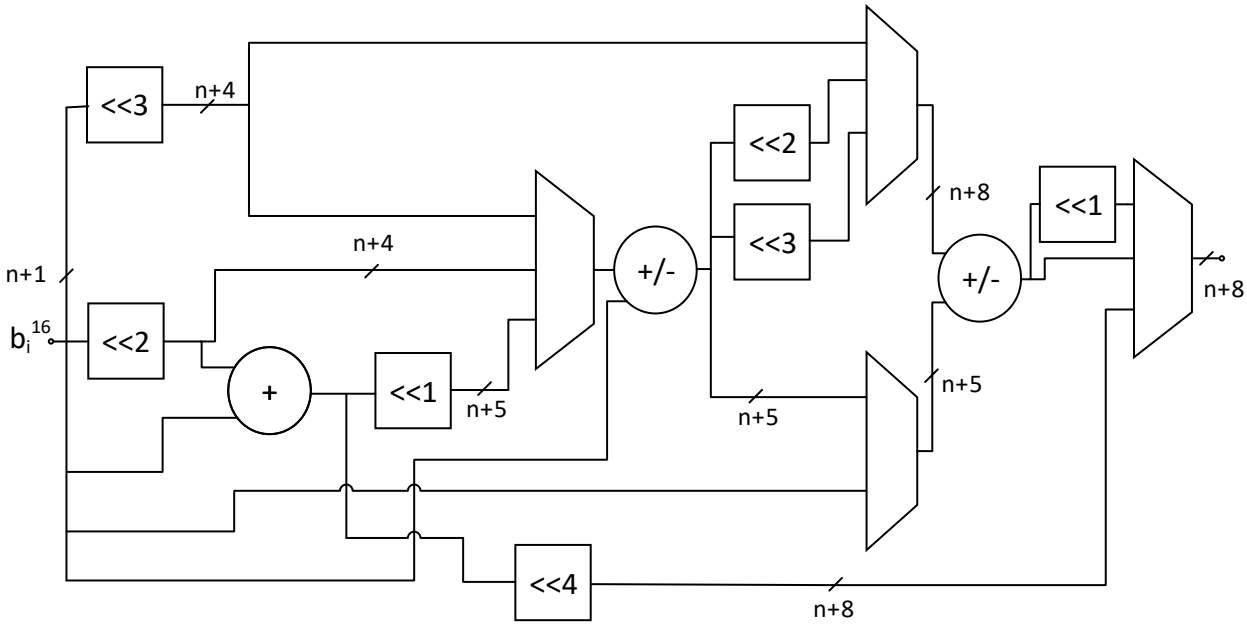
3.6 An area efficient and reusable HEVC 1D transform architecture

To minimize the logic circuit for the input adder unit, elements of the residual vector are transferred sequentially into the transform engine. It can be observed in Equations 3.3 - 3.6 that the optimal grouping of elements would be such that pairs of residuals are transferred in one cycle because they can be immediately processed. For area minimization purposes the input bus-width of 32 bits is selected. Two byte-aligned residuals will be either added or subtracted depending on the position in the result vector which is being calculated. Results of the operation are multiplied with all constants from a matrix $\boldsymbol{D}^N$ column and products are added to their respective accumulated value. Accumulation of products is done during $N/2$ cycles when the resulting vector element is available. Based on the given steps in obtaining the result, the architecture will include a pre-operation unit (PREOP), providing a sum and difference of the residuals, MCM units (MCMs) and accumulation units (ACCUs). Units that build a single lane are shown in Figure 3.7. Parallel MCMs are the preferred solution for systems where the complete residual column vector can be retrieved instantaneously. In the proposed architecture with a 32-bit bus, such a circuit wouldn't be fully utilized and would result in an additional area. Therefore, multiplexed MCMs are used. They are implemented using the algorithm from [39] where a block diagram can be retrieved from a multiplexed multiplier block generator presented in [40]. The block diagrams of the MCMs for transform sizes 4 and 16 are shown in Figure 3.6. Each adder and multiplexer in the circuit is designed with the minimum number of bits that can describe their outputs. For example, in an 4-point 1D DCT, adders are not designed

to have a bit-width of n+8, which is the maximum bit-width increase required to describe the output of the unit. Bit-width extension of the adder is specified to represent its output instead. In the case of multiplexers, the output bit-width matches the bit-width of its largest input. Bit-width extensions are made by wiring, like shifters and save area. The area advantage with multiplexed MCMs is obtained at the cost of increased latency and therefore a decreased throughput.



a



b

Figure 3.6: Multiplexed MCM unit for a) 4-point 1D DCT and b) 16-point 1D DCT

To enable the reusability of the same hardware for different transform sizes, the design is structured into lanes. The lane used for the transform size N is also used for all the larger transform sizes. The difference in the amount of multiply and accumulate cycles depends on

transform size as described above. Compared to the competing hierarchical architectures, which have multiplexers at the input of every N-point integer DCT unit when N = 4, 8 or 16, the proposed lane-based architecture requires multiplexing only once.

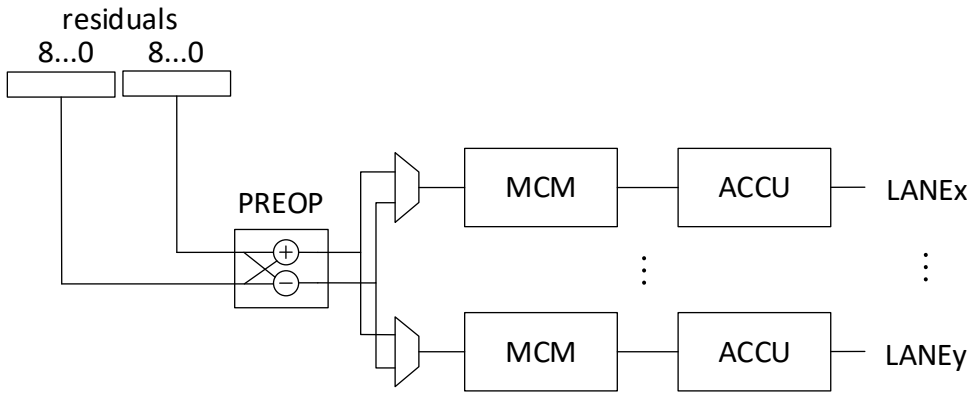


Figure 3.7: General block diagram of the integer DCT lanes

Constant multipliers used in a lane for one transform size are repeated for all transform sizes in which that lane is involved. Which row of the result vector a lane outputs is determined with the value of the MCM configuration register. Every register contains sixteen codes for sixteen multipliers. The code length depends on a number of unique elements in the same row of the transform matrix. The 1-bit code is sufficient to represent two unique multipliers in odd rows for a 4-point transform. In the case of a 32-point transform, a 4-bit code is needed to switch between sixteen different multiplication datapaths. The signs of multipliers are determined with a 16-bit value of the ACCU configuration register. Depending on each bit value, the MCM output will be added or subtracted from the current accumulated value. Instances of a lane differ from each other only in the content of these two registers. For example, LANE8 is used for the 8, 16 and 32-point integer DCT. It has four constant multipliers, which are used in every one of these three applicable transform sizes. LANE2 implements multiplication with a single multiplier sixty-four and therefore doesn't contain the MCM configuration register. A lane can be interpreted as a set of functional blocks that implements the multiplication of a basis vector with a residual vector and outputs the single transform coefficient. Schematic for one lane is shown in Figure 3.8

The serialization block is placed at the output because this architecture receives two residuals per cycle to keep the area usage low. It is controlled by the processing system. The architecture will output two 16-bit coefficients and signal their availability over the dedicated interface. There are two synchronization channels between the device and the processing

system. Over one the system can request a serialization hold and when it becomes effective it is signalized at the output. Over the second channel, the hold mode, as shown in Figure 3.9, is activated when scaled transform coefficients are ready for serialization but the previously started serialization is not yet finished. In that case, the result vector will be moved to the delay buffer and, using an additional output interface, a signal will be sent that the transferring of new residuals has to be stopped and will remain so until the signal is deactivated.

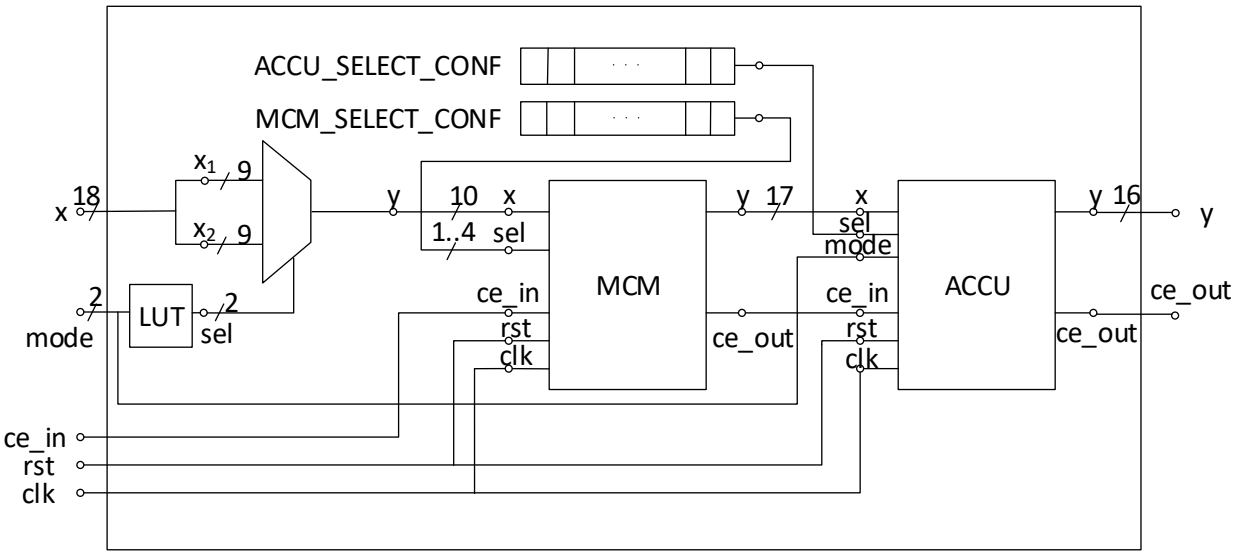


Figure 3.8: Schematic of a lane

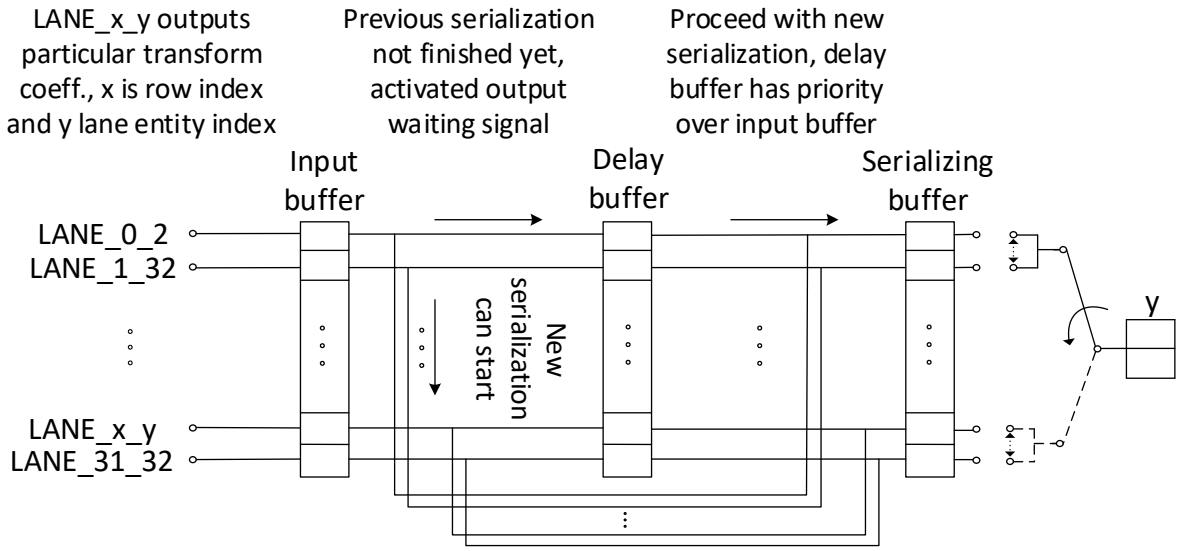


Figure 3.9: Serialization of scaled transform coefficients

The proposed 1D integer DCT architecture, which is designed for systems with limited hardware resources, supports all transform sizes. The proposed lane structure which exploits the subset property of the transform matrix simplifies power management. The interface

responsible for bidirectional communication with the processing system enables the transform to hold and continue without any data loss.

3.7 Implementation and evaluation

In this section, the performance of the proposed architecture is evaluated in terms of area, latency, delay, throughput and power consumption with focus on the area. Implementation and result presentation were done for comparable circuits of leading FPGA manufacturers (Xilinx, Intel Altera). To have an equitable evaluation and comparison with competing parallel integer DCT architectures, two architectures will be implemented, the proposed architecture with 32 bits wide data input and output, and a separate configuration where the serialization unit is omitted.

Table 3.4: Performance results of the proposed integer DCT architecture with output serialization

Transform size	LUTs	Registers	Latency	Frequency (MHz)	Throughput (Gsps)	Total power (mW)
4	313	452	8	741.84	1.48	189
8	1004	1113	10	590.32	1.18	208
16	2862	2493	14	480.77	0.96	251
32	6776	4948	22	407.50	0.81	355

Table 3.5: Performance results of the proposed integer DCT architecture with parallel output

Transform size	LUTs	Registers	Latency	Frequency (MHz)	Throughput (Gsps)	Total power (mW)
4	237	363	7	744.05	1.49	187
8	835	963	9	613.50	1.23	215
16	2367	2198	13	527.15	1.05	247
32	6085	4402	21	500.25	1.00	348

Design is coded in VHDL and synthesized targeting a Virtex 7 device, Xilinx XC7VX330T FFG1157 FPGA with speed grade 3. It is implemented using Vivado IDE according to the process described in Section 3.5. The word length of the input residual is set to 16 bits. The area, the number of used registers, the maximum delay and the total power are

obtained for the baselined design from the implementation reports, and the maximum operating frequency and throughput are calculated. The power analysis which is based on vectorless (probabilistic) power analysis methodology is run with a default device and environment settings and 100 MHz clock frequency. The results of the integer DCT for two device configurations and different transform sizes are presented in Table 3.4 and Table 3.5.

The results show that look-up tables' (LUT) utilization for the N-point transform is about 2.8 times and registers' utilization about 2.2 times the utilization for the $N/2$-point transform for the complete architecture. Power consumption increases by a factor of 1.2 on average. When the serialization unit is omitted there are less employed LUTs and registers and the maximum operating frequency is higher for all transform sizes. By eliminating one functional block in the architecture the path complexity is reduced, which affects the maximum path delay and operating frequency. The latency is contributed by two addends, five or six pipeline stages, depending on the architecture variant, and $N/2$ cycles needed to process one residual vector.

This architecture supports the processing of 4K ultra high definition (UHD) videos in HEVC standard at 30 frames/s and 4:2:0 YUV format. The minimum operating frequency for this video format is calculated as follows:

$$F_{min} = 2 \times W \times H \times format \times frame\ rate \times \left(\frac{L}{32 \times 32} + \frac{1}{throughput}\right) \qquad (3.8)$$

where $W \times H$ is the resolution of a video sequence, and the format equals 1.5. It is considered that a pair of residuals is transferred every cycle to the input of a 1D integer DCT core and that two transform coefficients can be output per cycle. So, the throughput expressed in Pixels Per Cycle unit (ppc) is 2. Factor two in Equation 3.6 is characteristic for the folded structure of a 2D transform. L is the pipeline latency. To process a 32 × 32 size transform block using a 2D folded structure, built with the proposed 1D architecture, $2L$+1024 cycles are needed. The transpose buffer is not in the scope of this work but is required for a 2D transform according to Equation 3.2. This performance evaluation is not affected if its throughput is not less than two pixels per cycle. The minimum operating frequency for the given format and processing rate is 393 MHz. This condition is satisfied by the proposed architecture. This is a satisfactory result considering the design objectives and compromise on throughput.

The performance of the proposed architecture is compared to related architectures that target the FPGA platform and results are shown in Table 3.6. Since about 90% of transform blocks are of size 4 × 4 or 8 × 8 [41], the 4-point and 8-point integer DCT accelerators are considered. To be able to have a fair comparison with a pipelined design, like the one from [28], the synthesis is made on the same device, Stratix V FPGA 5SGXMABN3F45I4 using Quartus Prime Standard Edition. Logic cells from Stratix V and Virtex 7 device families have a different architecture and therefore the number of used adaptive logic modules (ALMs) in Stratix V can't be directly compared with LUTs in Virtex 7 to assess area efficiency.

Table 3.6: Comparison with related works

Architecture	Technology	Max Freq (MHz)	ALMs / LUTs	Registers	Pixels / Clock Cycle	Throughput (Gsps)
Bolaños-Jojoa [28]	Stratix V	630.12	108	288	4	2.52
Proposed	Stratix V	696.38	92	244	2	1.39
Chatterjee [27]	Virtex 7	183.30	924	-	8	1.46
Proposed	Virtex 7	613.50	835	963	2	1.23

All related designs are made for parallel processing of all elements in a residual vector. The maximum operating frequency for the proposed design is higher than [27] and [28]. The usage of parallel MCMs rather than multiplexed MCMs comes at the cost of the area. That is apparent in the number of utilized ALMs and LUTs. Moreover, these two competing designs are not developed as accelerators but as single DCT modules and therefore do not include synchronization and control signals which are necessary for communication with the processing system. The amount of utilized logic cells is 14.81% lower than in [28] while throughput is decreased 1.81 times. Compared to [27], the hardware cost is lower 9.6% while throughput is lower 1.21 times. Though [27] is not implemented as a pipelined circuit this will implicate the use of flip-flops and the number of used LUTs is not under the impact.

4 A PERFORMANCE OPTIMIZED SOFTWARE IMPLEMENTATION OF THE HEVC TRANSFORM AND QUANTIZATION ALGORITHM

4.1 Introduction

The HEVC standard is devised for a hybrid video coding where transform, scaling, and quantization (TQ) are contained in an important functional block. In modern hybrid video coding systems that block follows the motion-compensated prediction and precedes entropy coding. Regardless of how effectively the preceding prediction process is exercised, there is typically a remaining prediction error residual signal that has to be further processed. As a new feature, when compared to prior standards, HEVC introduced additional transform block (TB) sizes to make encoding of video files of 4K and 8K resolutions more efficient; replaced the real-valued discrete cosine transform (DCT) with the integer DCT to ensure device interoperability and avoid an encoder-decoder mismatch; simplified (de)quantization process by turning it into scalar division (multiplication). Efficient implementation in software exploiting SIMD capabilities and parallel processing were set as design goals during HEVC TQ development [20].

To cope with increased computational complexity, which is especially challenging for the design of real-time applications, advanced computing architectures are required. Heterogenous multiprocessor computing architectures are one possible solution in such cases. There, the application is portioned in such a way that tasks are distributed among coprocessors depending on their specialized processing capabilities. Currently, the most common heterogeneous systems are made of a multicore Central Processing Unit (CPU) and a GPU [42]. Serial portions of applications are run on the former, while data-parallel, compute-intensive portions are offloaded to the latter. Programming paradigms for these two architectures are different, which has to be considered during the application design.

This chapter is organized as follows. Section 4.2 describes types of operations included in the TQ functional block and differences between TQ and its inverse operations. In Section 4.3 an overview of significant scientific contributions related to this topic is given. Section 4.4 describes the development environment and methodology for efficiency validation of performance optimized software implementation for parallel computation. In Section 4.5 one's own, highly parallel HEVC transform and quantization kernel with AZB identification [43] is presented. It is designed for execution on a GPU. Memory bandwidth, instruction throughput,

and on-chip memory allocation were properly balanced to achieve efficient execution and high resource utilization. Different optimization techniques, from overlapping data transfers with computation to fine-tuning arithmetic operation sequences, were incorporated to reach high performance. Section 4.6 provides a comprehensive evaluation of all optimizations made during the iterative and incremental development.

4.2 HEVC transform and quantization process

HEVC TQ processes the TBs coming from the residual quadtree (RQT) structure. The output are the quantized transform coefficients (level). The process consists of three stages: the 2D integer DCT transform, quantization, and all-zero-block (AZB) identification as shown in Figure 4.1.

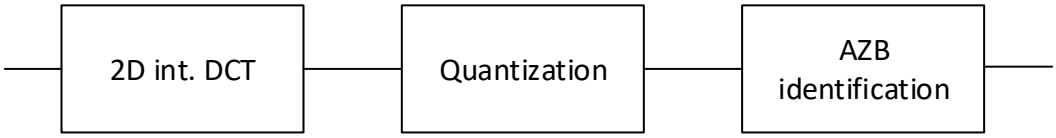


Figure 4.1: General block diagram of the HEVC transform and quantization process

The 2D integer DCT and quantization are described in previous chapters. AZB identification is the last stage in the process. The information if all levels coming out from one TB are zero is used to set the value of the syntax element coded block flag (CBF). This helps to reduce the number of bits to be transmitted.

The decoder is built-in the HEVC encoder since the frame has to be fully decoded to be used as a reference in the estimation and prediction process for subsequent frames. As the first step in the reconstruction of the frame, quantized transform coefficients are dequantized and inverse transform is applied thereafter. The computation complexity of dequantization and inverse transform (DQIT) is equivalent or lower to that of the TQ process. Double matrix multiplication of input blocks with the HEVC transform matrix and its transpose with intermediate scaling, the same number of scalar multiplications and bitwise shifts appear in both functional blocks. AZB identification is part of the TQ process only.

4.3 Overview of scientific contributions to HEVC transform algorithms on GPUs

Previous research in this field focused mainly on CPU+GPU heterogeneous platforms. It tackled motion estimation as a functional block with the highest computational load [44], [45]. A step further was taken with massive parallelization in [46] and all functional blocks of the HEVC decoder, except entropy decoder, were ported to the GPU. Quite the contrary, solutions with TQ acceleration using FPGAs rather than a GPU [47] are much more represented as a research topic.

Regarding migration of the HEVC TQ to the GPU in [48] two tables, one describing transform unit (TU) partitioning and the other QP value storing, together with the mapping algorithm at the CTU level, were proposed to achieve efficient implementation. In [49] authors dealt with a heterogeneous system for a HEVC encoder where motion-compensated prediction processing already resides at the GPU side and additionally the TQ has to be ported there. Parallel TU address list construction and coefficient packing were proposed to achieve high processing speed. Benchmarking the performance of four different HEVC 2D transform kernel designs against its industrial solution, which combine assembly and AVX2 instructions, was carried out in [50] without conducting performance optimizations. In [51] the highly optimized parallel implementation of the HEVC dequantization and inverse transform is presented using the unified programming model for the CPU+GPU heterogeneous system.

Previously mentioned works mainly focus on the integration aspect of GPU implementation of the HEVC TQ and do not reveal in detail the design of kernel function and optimizations which are made to efficiently balance between GPU resources, sum of registers, allocated on-chip memory per thread-block, number of threads per multiprocessor and global memory bandwidth. That is preventing proper validation of their performance gains and gaps. Additionally, it has to be mentioned that fair comparison using only TQ processing time is not feasible. In [48], GPU acceleration is done at the CTU block level and in [49] and [51] it is done at the frame level with transform blocks previously grouped for GPU acceleration. The latter approach allows much better use of GPU parallelism.

The highest workload stage during HEVC TQ is the 2D forward transform which is mathematically realized as a double matrix multiplication. Many guidelines exist with general optimization principles exampled in matrix multiplication and other basic linear algebra

subroutines (BLAS) [52][53][54]. They mainly deal with the multiplication of two large size (one or both dimensions) matrices which are tiled to small size subblocks e.g. 16 × 16 and distributed among the GPU thread-blocks. Values from input matrices are repeatedly loaded in subblocks and the resulting subblock is computed as the sum of products of these subblocks. HEVC TQ operates with batches of small size matrices i.e. TBs where the tiling approach would degrade performance. The efficient mapping of TBs and dot product computations to various components of the GPU subsystem with the adaptation of known performance optimization techniques was set as the main design objective. The final proposal presents a systematic and efficient solution for the multithreaded GPU kernel function for all supported transform sizes in HEVC.

4.4 Development environment and methodology

The heterogeneous processing systems used in this thesis contains the CPU and its memory, hereafter referred to as the host, and GPU and its memory hereafter referred to as the device. Both are connected to motherboard and exchange data over the PCI Express x16 Gen3 high-speed bus (PCI-e). Two different environments, set up to perform all planned evaluations, are shown in Table 4.1.

Table 4.1: Evaluation environments

Environment name	Desktop	Workstation
CPU	Intel Core i5-4570 3.20 GHz	Intel Core i5-3570K 3.40 GHz
Memory	DDR3 8GB 800 MHz	DDR3 8GB 800 MHz
Bus	PCI Express x16 Gen3	
GPU	NVIDIA GeForce GT 640	NVIDIA Tesla K40c
Memory	DDR3 2GB	DDR5 12GB

The Desktop and Workstation environments have similar CPU characteristics, but significant differences in GPU characteristics affect their performances in accelerated computing. GPUs from both environments have the same Kepler architecture but Tesla K40c exhibits a higher performance due to a larger number of streaming multiprocessors (SMs) and faster memory as shown in Table 4.2. For measurements obtained for final benchmarking and comparison with competing implementation in Section 4.6.10 of this chapter, the Workstation environment was used. All other measurements including those associated with the AVX2 implementation, which is not supported in the Workstation environment, were made using the Desktop environment.

In this thesis, the Compute Unified Device Architecture (CUDA) programming model [55] from NVIDIA, one of the leading GPU manufacturers was used to support heterogeneous computation where applications use both the CPU and the GPU. CUDA was integrated into the Microsoft Visual Studio (MSVS) IDE. Components of the toolset were:

- MSVS Community 2017 v15.9.7
- Platform Toolset: Visual Studio 2017 (v141)
- NVIDIA Visual Profiler Ver.: 10.1
- CUDA Compilation tools R10.1, V10.1.105
- CUDA Driver Ver.: 10.1

Except for displaying a timeline of the application's CPU and GPU activity, the Visual Profiler was exploited for a comprehensive application analysis and identification of optimization opportunities.

Table 4.2: GPU comparison in the two environments

GPU model	NVIDIA GeForce GT 640	NVIDIA Tesla K40c
SM count	2	15
Core count	192	192
Core clock (MHz)	902	745
Memory bandwidth (GB/s)	28.51	288.4

Application *DCT_Runner* was launched from the command line with the following arguments:

```
DCT_Runner --gpu --cublas --avx --blockSizes <no. of 4×4 TBs> <no. of 8×8
TBs> <no. of 16×16 TBs> <no. of 32×32 TBs> --qp <QP value> --nTrBuff <no.
of kernel invocations>
```

Four different implementations were supported. Results obtained from the GPU, CUBLAS, and AVX2 implementation are verified against results from the referent CPU implementation. Since the application was developed for benchmarking purposes, every implementation started with an untimed warm-up run before the timed execution was invoked. Kernel function that will be executed on a GPU was invoked ten times and the final measurement value was obtained as an average of all processing times. The application could be reconfigured over several preprocessor macros

```
#define HIGH_RES_TIMER
#define RANDOM_RESIDUALS 1
#define RESIDUAL2THREAD_BLOCK_MAPPING_TYPE 1
      // 0 - 1:1 mapping
      // 1 - (1024/N^2):1 mapping
      // 2 - (2048/N^2):1 mapping
      // 3 - (3072/N^2):1 mapping
      // 4 - (4096/N^2):1 mapping
      // 5 - (512/N^2):1 mapping
#define SMEM_PADDING 1
#define IS_ZERO_MATRIX_IDENTIFICATION 1
#define PINNED_MEM 1
```

High- and low-resolution time measurements were supported in the application. Residual samples could be generated either as random integer values in the range between -255 and 255 or as a linear incremental increase by one, starting with zero-value of the top-left element in every block. As the next step it could be configured how many TBs are processed within one thread-block. A memory padding could eliminate bank conflicts when accessing data stored in shared memory. AZB identification could be bypassed during kernel execution. Data transfers could be selected as pageable or page-locked. By default, data allocations in the host were pageable. To be able to transfer the data from pageable memory, where allocation is made by default, to the device, the CUDA driver must first allocate space in the page-locked memory, copy data there and then transfer them to the device. The copying overhead can be avoided by direct allocation in the pinned memory.

Optimizations were carried out and new features delivered by employing iterative and incremental development. Basic performance measurement in every iteration included overall and kernel processing time per frame as shown in Figure 4.2. Overall processing time per frame

involved the duration of data transfers from the host to the device and back. Data that were sent towards the device were blocks of residual samples from one frame. Once computed, the device returned blocks of levels and AZB identification array where each element identifies if the associated block of levels contains any non-zero level.

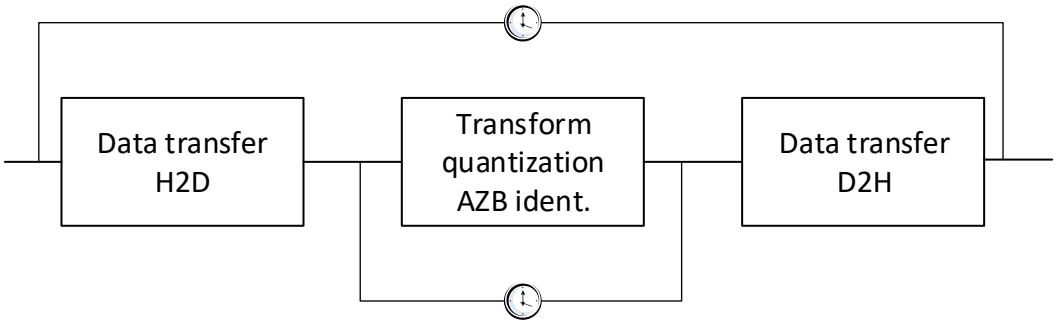


Figure 4.2: Time measurement

Except for the time measurement, performance metrics included the device global memory bandwidth, occupancy and shared memory efficiency, if that was required by the objectives set in an iteration.

4.4.1 Effective memory bandwidth

Two bandwidths are calculated for evaluation purposes, the theoretical peak bandwidth and the effective memory bandwidth. The former is determined by a memory clock rate, bus width, and transfer type. The corresponding value is labeled as BW_{Theo}. The latter is calculated as a ratio between the number of memory accesses in bytes and their overall duration, as expressed in equation:

$$BW_{Eff} = \frac{(R_B + W_B)}{t} \tag{4.1}$$

where R_B is the number of bytes read per kernel, W_B is the number of bytes written per kernel and t is the elapsed time. In general, the best achievable bandwidth of GPU memory is usually in the range of 75% to 85% of the theoretical value. Since time is measured at the host side, first before the kernel call and second with the first command after the kernel was executed, kernel start and stop overheads affected bandwidth. Additionally, it is important to note that the lower BW_{Eff} value doesn't necessarily indicate inefficient memory usage. The more computations in the kernel, the more time is spent performing arithmetic operations, and the value in the denominator in Equation 4.1 increases. In case the kernel execution is mainly occupied by memory accesses and not by computation it will be *bandwidth bound*. Then the

bandwidth is the most important metric to measure and optimize. In another scenario, the kernel can be *computation bound* and then computational throughput is expressed as:

$$OPs\ per\ second_{Eff} = \frac{Total\ number\ of\ operations}{t} \tag{4.2}$$

and is more relevant for performance evaluation. In practice, kernels are mainly bandwidth bound.

4.4.2 Occupancy

Occupancy [56] is an NVIDIA's metric that describes how effectively the GPU hardware is kept busy or from how many warps the SM can choose to execute. It is the ratio of the number of active or resident warps per multiprocessor to the maximum of possible active or resident warps. Each SM on the device has a set of registers and shared memory space available for use by CUDA threads. Both resources are shared and allocated among the thread-blocks executed on an SM. Resource usage indicated through the number of configured threads, bytes allocated in shared memory and used registers per thread-block, affects the occupancy.

Maximizing the occupancy can improve latency cover during global memory accesses. Nevertheless, its higher value does not necessarily suggest higher performance. If the kernel is not bandwidth bound then increasing occupancy will not inevitably increase performance. If thread-blocks, active in a kernel grid, are bottlenecked by computation and not by global memory accesses, then increasing occupancy may have no effect. In this work, experiments were the preferable method to understand how design changes affect the measured processing times.

4.4.3 Shared memory efficiency

Shared memory efficiency is defined by NVIDIA as the ratio of requested shared memory throughput to required shared memory throughput, and it is expressed as a percentage. Shared memory has an overall 32 banks and supports 32- and 64-bit addressing modes. Each bank has a bandwidth of 64 bits per clock cycle. Banks can be accessed simultaneously. If more than one thread in a warp accesses data in the same bank, not belonging to the 64-word aligned segments, accesses cannot be served simultaneously but have to be serialized. This performance degradation of shared memory access is called a bank conflict. The referenced metric describes how many excessive memory accesses happened due to bank conflicts.

Depending on measurement objectives in each iteration, support to different block distributions was implemented accordingly. Work started with separately implemented kernels for a (32 × 32)-point 2D integer DCT and quantization. These types of the kernel have the highest computational complexity. The number of multiplications for a 2D transform is $2N^3$ and the number of additions $2N^2(N-1)$. The kernel for a (4 × 4)-point 2D integer DCT was implemented as next when mapping from TB structure at the host side to the thread-block structure at the device side and transform matrix access modes had been investigated. Most of the design decisions could be met using these two transform sizes. Transforms for the remaining transform sizes 8 × 8 and 16 × 16 had to be implemented when the impact of shared memory padding on bank conflicts had been investigated. When not avoided, bank conflicts cause decrement of shared memory effective bandwidth.

4.5 Performance engineering for HEVC transform on GPU

GPUs are powerful arithmetic engines suited to running thousands of threads in parallel. To obtain the best performance from a GPU and to achieve suitable low video latency, GPU processing is done at the frame level or frame's horizontal segment level. Compared with pure CPU implementation, heterogeneous accelerator-based architecture could potentially suffer from a large overhead in data transfer between the host and the device. As shown in Section 0 and 4.6.10, that overhead could reach up to 54% of overall processing time in case of a mid-end GPU and 87% in case of a high-end GPU.

Residual data blocks are copied asynchronously in groups from host to device as shown in Figure 4.3. One group includes all the blocks of the same size. Data transfers are optimized according to [57]. Once computed, blocks of quantized coefficients and their zero markings are returned to the host. The device receives data into its global memory and transfers them from there to the SMs. Transform matrices of all sizes are written to the global memory in advance. To reach the full SM utilization and decrease the thread creation and destruction cost, the grid-stride loop technique [58] was employed for kernel design. Access to the device global memory is done in a coalesced way to maximize effective memory bandwidth. A thread-block accesses a group of residual blocks, block by block in the raster scan order where the data belonging to a residual block are accessed row by row.

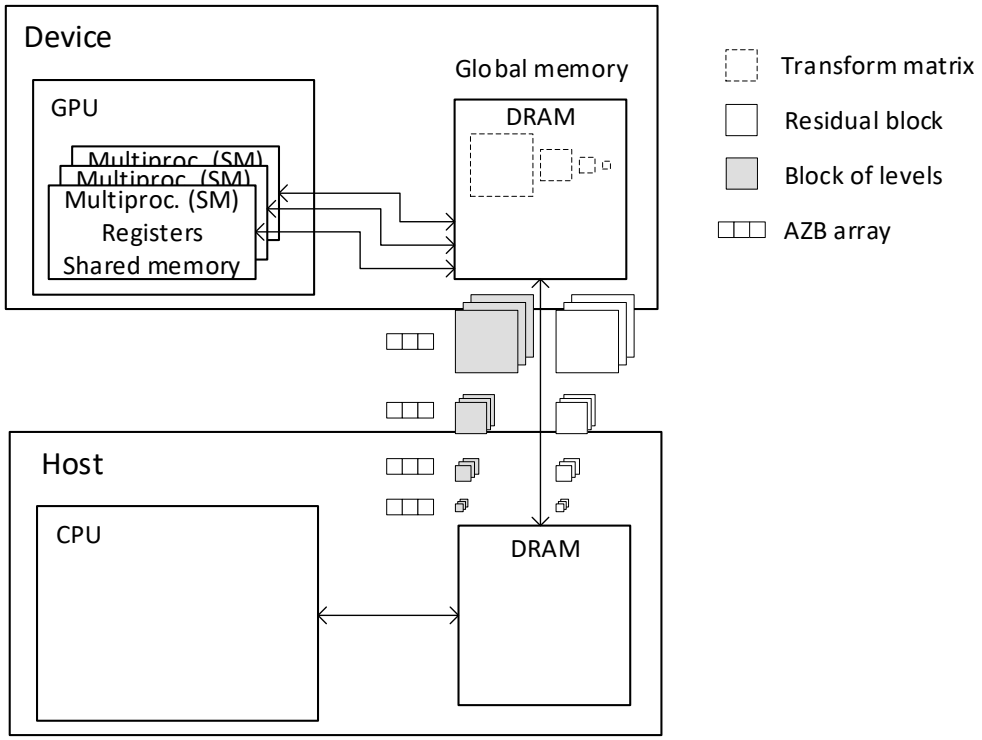


Figure 4.3: Data transfers between the host and the device

The residual image is dynamically partitioned into TBs whose size adapts to spatial and frequency characteristics of the corresponding image area. Transform operations performed with TBs of different sizes are different and TBs are grouped before the transfer to the device. Gathering the same sized TBs can be done by resuming work from [59]. The prediction stage can be modified in such a way that residual values are stored in a memory location for a particular block size. The index of the addressing information for the block is written into the TU data structure.

When computing the HEVC transform, the partial butterfly algorithm significantly reduces the number of arithmetic operations as shown in Section 3.2. This is beneficial for FPGA implementation as additional operations increase the area usage and the latency. If the partial butterfly algorithm would be implemented on a GPU then the data-dependent conditional branching in the kernel code would be necessary. Different arithmetic operations have to be executed to compute a one-half of the transform coefficients compared to another half. Moreover, this conditional branching is recursively applied to one-half of the computed coefficients. In case of branching code, the threads within a warp diverge and the warp consecutively executes each branch. In this way thread utilization and computational throughput decrease. Taking this into account the straightforward matrix multiplication is used.

Since matrix-matrix multiplications are the most compute-intensive part of the process, the shared memory, located on the chip, is used to hold the input data to reduce global memory accesses. If the latencies for these two types of memory are compared, then the shared memory latency is approximately 100 times lower. To achieve high bandwidth, it consists of memory banks. Each bank provides a bandwidth of 64 bits per clock cycle. Buffered data, byte-aligned residuals and intermediate transform coefficients are each 16-bits wide. The shared memory access pattern which enables maximum throughput of shared memory will be presented in the next sections.

Shared memory is additionally exploited as the single access point for AZB identification. Mapping ratio for mapping from residual blocks to thread-blocks can be described with the expression $n{:}\,1$ $(n \in \mathbb{N})$. Therefore, an intermediate array of Booleans of length n is allocated in the memory to identify AZBs. Group of threads in a thread-block that computes levels in a TB will initiate a write request to the same corresponding array element if the non-zero level was identified among transform coefficients which are computed by that thread. As the last step, each array element is tested by a single thread in the group to set a related array element in the output AZB array in global memory. To ensure correct results of values in both arrays, the threads are synchronized two times. The first time after the initialization of arrays in shared and global memory and the second time after the thread wrote a corresponding value to the matching residual block's array element. If the AZB identification stage is skipped in the process kernel, the processing time is shortened 4% or 2% depending on the used GPU type.

4.5.1 Indexing transform blocks with thread-blocks and threads

There are two levels of parallelism in the GPU. In a multithreaded program, kernel function, which is to be parallelized, is partitioned into thread-blocks and in turn, thread-blocks are partitioned into threads. This programming model has a grid at the top of the hierarchy. Thus, a kernel is launched as a grid of thread-blocks of threads. Grid and thread-block partitioning can be made in one, two or three dimensions. This provides a natural and intuitive way to model different data structures like vectors, matrices or volumes. All elements in a data structure can be accessed using built-in variables `gridDim`, `blockDim`, `blockIdx` and `threadIdx`. They provide access to each thread that executes the kernel. An example of a two-

dimensional split of a grid and thread-block with appropriate indexing is shown in Figure 4.4. Associated kernel invocation for the illustrated grid is as follows:

```
dim3 blocksPerGrid(2,3);
dim3 threadsPerBlock(4,2);
kernel<<<blocksPerGrid, threadsPerBlock>>>(…)
```

Parameters inside triple brackets syntax are called execution configuration. Each of the forty eight threads configured in this example executes a `kernel` function.

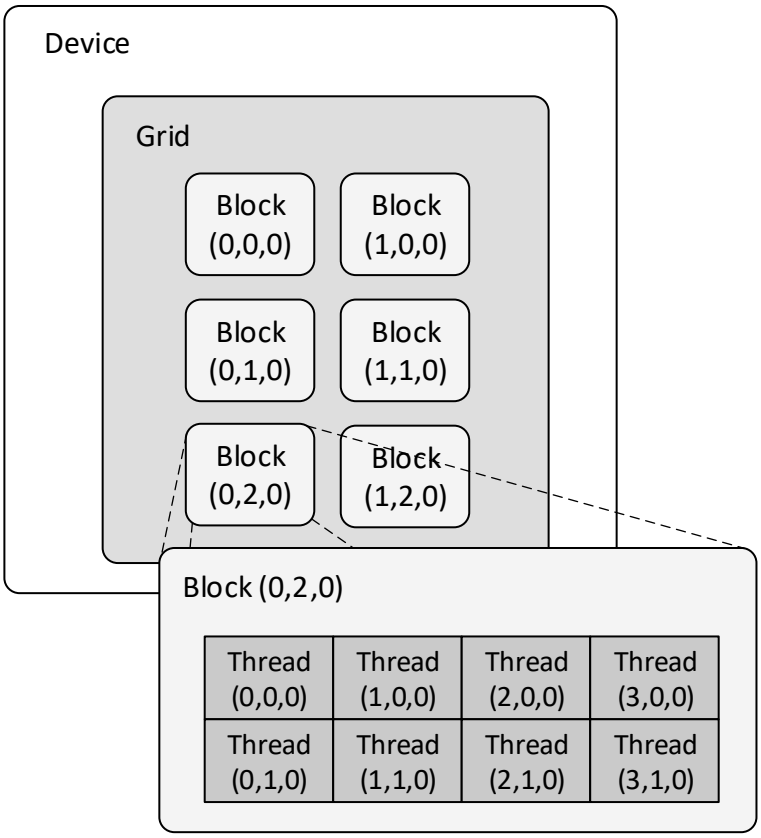


Figure 4.4: Grid of thread-blocks

For performance optimized HEVC TQ with maximized parallel execution, the maximum utilization of available device resources has to be achieved. The GPU physical limits relate to numbers of threads, thread-blocks, available registers, allocatable size of shared memory per SM. The SM schedules and executes threads in warps, groups of 32 parallel threads. The number of active thread-blocks and warps depends on the amount of shared memory and registers used by the kernel but is constrained by their physical limits. The occupancy performance metric will show if a good balance between different resources was reached.

In this work kernel operates with four different matrix sizes. If one thread-block handles one TB and every thread compute one element in a TB then the maximum number of threads per thread-block (1024) is utilized for the largest size TB. An increasing number of TBs that would be handled by a single thread-block would not be feasible. If 1:1 mapping from TB data to thread-block is applied for all transform sizes and data modeled with a one-dimensional grid and two-dimensional thread-blocks, then indexing a batch of TBs with thread-blocks and threads would be as follows:

```
sA[threadIdx.y][threadIdx.x] = A[blockDim.x * blockDim.y * blockIdx.x +
threadIdx.y * N + threadIdx.x]
```

where `sA` is a shared memory copy of a two-dimensional array `A`, initially received from the host and stored to the global device memory. `N` is the order of a square matrix.

Contrary to a 32 × 32-point transform, for other transform sizes one thread-block can process more TBs. If same data modeling within a kernel grid would be kept the indexing would be as follows:

```
sA[threadIdx.y][threadIdx.x] = A[blockDim.x * blockDim.y * blockIdx.x + N *
N *(threadIdx.y /N * (32 / N) + threadIdx.x / N) + threadIdx.y % N * N +
threadIdx.x % N]
```

For the presented data model, the mapping ratio can be generally expressed as $(1024/N^2)$: 1. An example of the mapping of TBs of size 4 × 4 is shown in Figure 4.5. Residual data within a TB are stored in row-major order. When performing matrix multiplication each thread retrieves data from the corresponding row in the first matrix and the corresponding column in the second matrix.

Mapping one residual block to one thread-block is a straightforward approach when setting execution configuration. Considering the maximum number of resident thread-blocks per SM, which is sixteen for the Kepler architecture, such mapping would cause very low occupancy for TBs of size 4 × 4. If one thread would compute one level sample that yields that one SM processes only 16 TBs. In that case 256 threads would be utilized which is significantly less than 2048, the maximum number of resident threads per SM. For TBs of size 32 × 32, this mapping ratio wouldn't be an issue since one thread-block would contain 1024 threads. Low occupancy would cause performance degradation since the kernel is bandwidth bound. An arithmetic calculation on the stream processor has a latency ten to twenty of cycles while the global memory access can take up hundreds of cycles. For the computation of a quantized transform coefficient, $4N$ bytes are loaded and 2 bytes are stored to any of two memories during

each transform stage. The overall number of executed multiply-add operations is $3N$, N of which are performed during each transform stage and N during quantization. Thus, arithmetic intensity equals $3N/(8N+4)$. This value is lower than the ratio of the theoretical instruction throughput (57.73 GInstr/s for 32-bit integer instructions) to the theoretical memory throughput (28.51 GB/s) for the used mid-end GPU. This makes kernel memory bound [60].

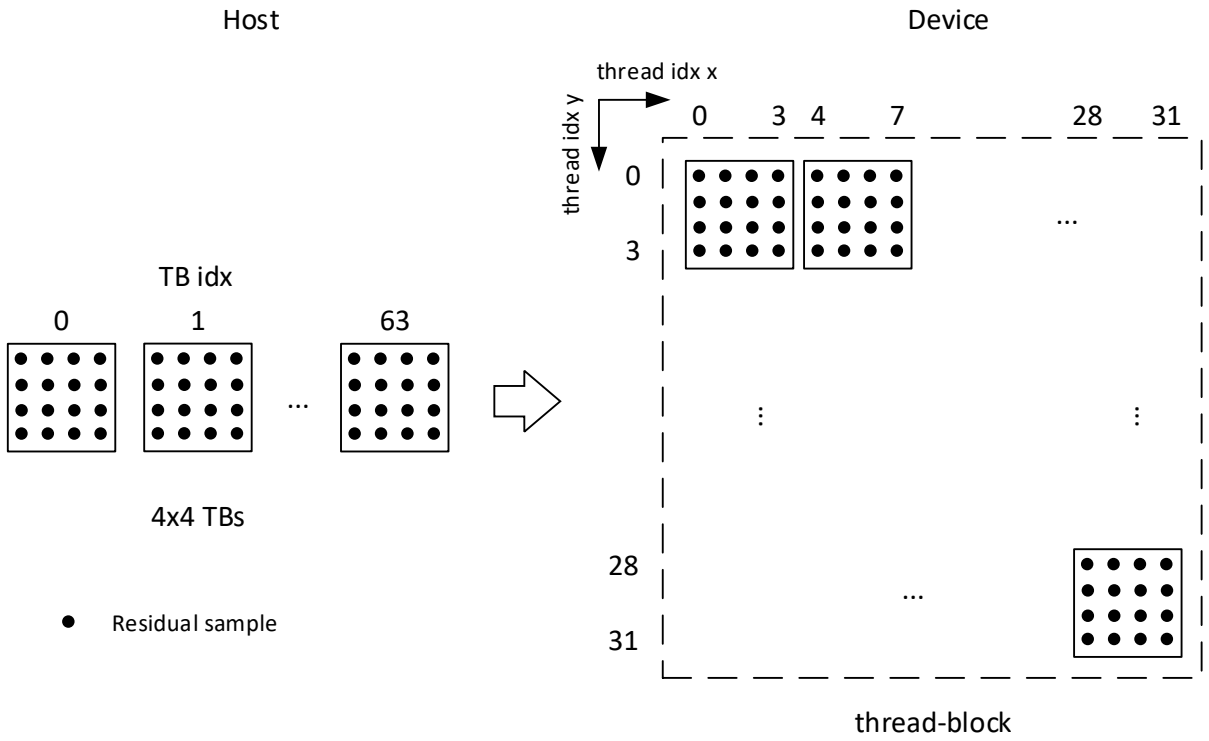


Figure 4.5: 64:1 residual to thread-block mapping for 4 × 4 TBs

4.5.2 Vectorized memory access

The kernel for transform and quantization with AZB identification is bandwidth bound as shown in the previous section. Effective global memory bandwidth is determined with the number of accessed bytes per kernel during its execution time. That number exceeds the number of executed operations. Since residual data are represented using short, int or 32-bit float data types, load and store instructions will operate with 32-bit data. The performance of these operations can be improved by using the vectorized load and store instructions [61]. Those instructions operate with 64- or 128-bit data. The expected gains using vectorized instructions are the reduced total number of instructions, reduced latency, improved bandwidth utilization and kernel shift towards computation bound.

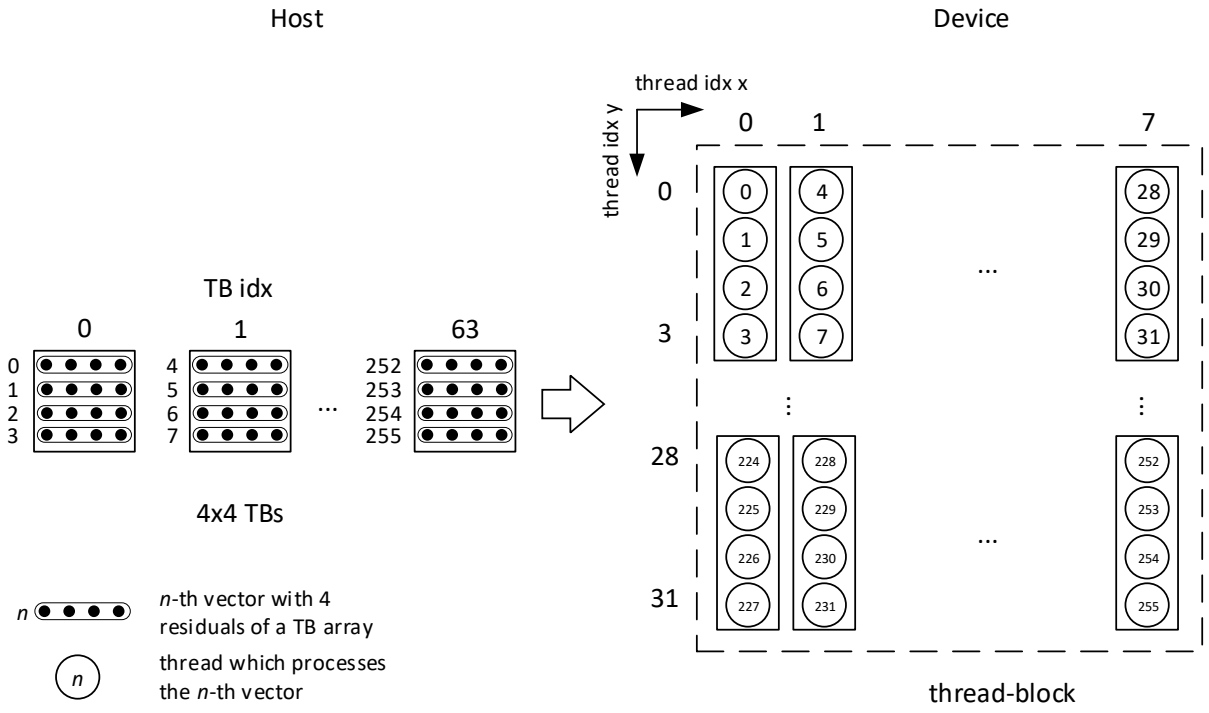


Figure 4.6: 64:1 residual to thread-block mapping with vectorized memory access for 4 × 4 TBs

By employing vectorized memory access, more mapping options are applicable and can be tested. If one thread handles one vector that computes four matrix elements, then four TBs of the largest size can be assigned to one thread-block. An example of residual to thread-block mapping with a vectorized kernel for 4 × 4 TBs using mapping ratio 64:1, as in Figure 4.5, is shown in Figure 4.6. It can be noted that fewer threads are contained in a thread-block compared to scalar memory access. Thus, their utilization increases. The disadvantage of vectorized access is the increased use of registers. This is an acceptable compromise since the use of registers is not excessive. They are used for intermediate results of dot products, stride constant, sign handling during quantization and zero-level flag in AZB identification. The biggest performance improvement is expected for shared memory accesses.

4.5.3 Efficient vectorized access pattern to shared memory

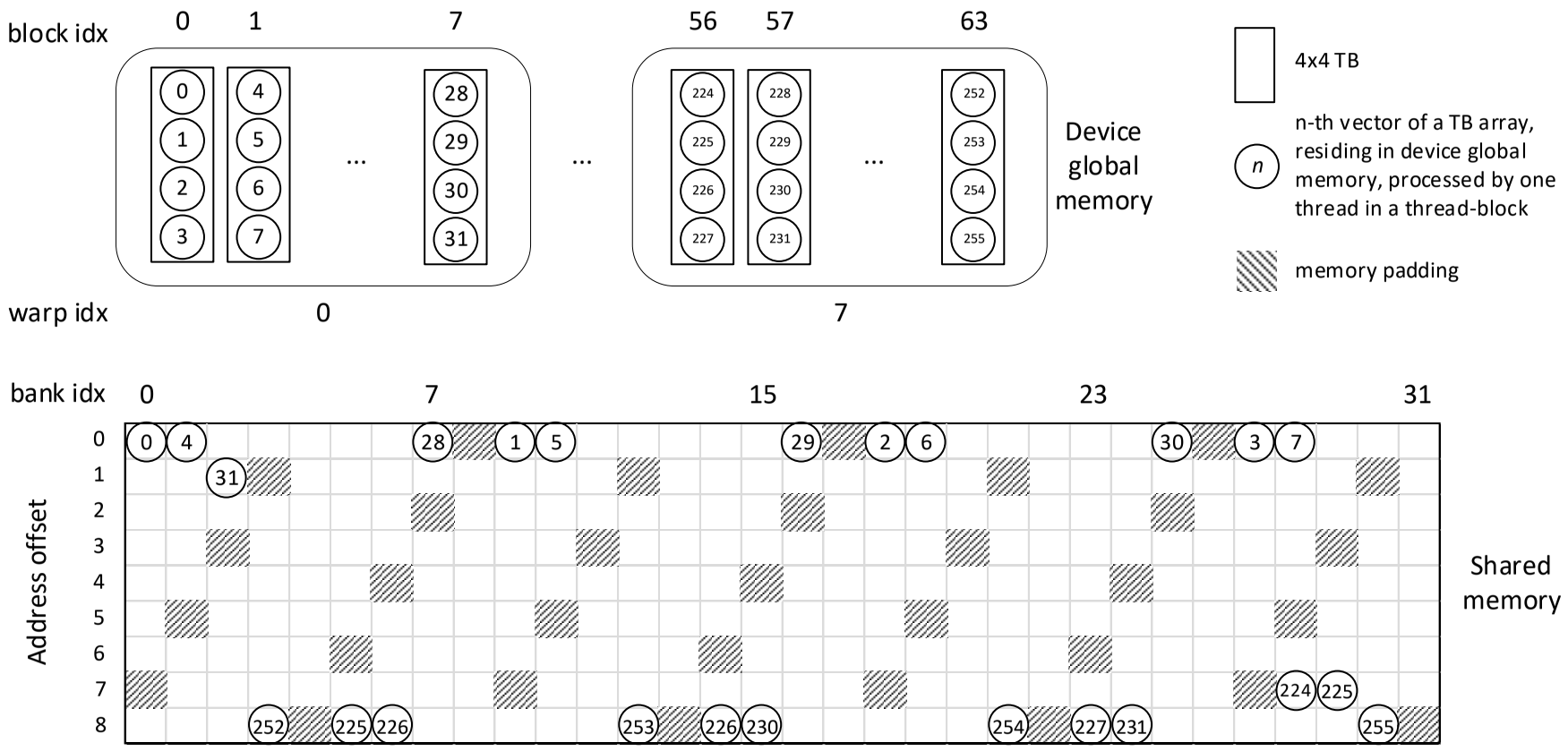
For efficient communication with the memory, vectorized memory access is used and four data per bank per transaction are retrieved from the memory. Bank size is reconfigured to 64 bits so that in a thread-block successive 64-bit vectors are assigned to successive banks. By using 64-bit vectors, an adjustment to the bus width of the shared memory bank is accomplished. To achieve overall high memory bandwidth, an appropriate access pattern for

all TB sizes has to be created. In the 2D transform application, every thread will execute a dot product of a matching row in a multiplicand block with a matching column in a multiplier block. Reading row data is performed as reading along the banks and as such is naturally parallel. In the case of the widest row, which appears for 32 × 32 TB, row data will reside in eight different banks. Data load can be done within one transaction.

Loading of column data within a warp with an access pattern that results in reading from the same bank causes bank conflicts. With an array width of 8, which is the least common multiple for all array widths in vectorized access (1, 2, 4, 8), and regular access pattern bank conflict would happen for transform sizes larger than four. For that particular transform size memory locations, belonging not only to column data from one TB but to a complete row of TBs in a thread-block, always map to different memory banks. For example, in the case of transform size 8 × 8, 32 memory locations in separate banks are sufficient to store only half of one row of TBs in a thread-block.

Memory requests to one bank are split into as many requests as there are requested 64-bit words in that bank. To prevent bank conflicts, the shared memory 2D array is padded with an additional column. The additional column will cause data to shift right to the new bank. But this technique is not efficient for all transform sizes. Access pattern to the padded shared memory array for one thread-block for 4 × 4 TBs and 32 × 32 TBs is shown in Figure 4.7. Padded array locations are marked with striped squares. In the case of a 4-point 1D transform, threads which process eight consecutive TBs compose one warp. As can be seen, when padding is used, some data which are processed by the first warp are spilled to the next memory location in the already used banks. Such data couldn't be accessed in the same transaction as the data from the upper memory location. Without padding, the first warp would access memory location 0 in all 32 banks in the same cycle. A similar thing happens for 8 × 8 TBs where warps would retrieve the necessary data from shared memory in three requests when padding is applied instead of two requests when padding is omitted. For 16 × 16 and 32 × 32 TBs, there are four and eight serialized memory requests to column data respectively when there is no padding. With padding, those accesses are conflict-free. For the latter transform size that is shown in Figure 4.7b.

a


block idx
0
1
7
56
57
63
0
1
2
3
4
5
6
7
28
29
30
31
224
225
226
227
228
229
230
231
252
253
254
255
...
Device global memory
warp idx
0
7
4x4 TB
n-th vector of a TB array, residing in device global memory, processed by one thread in a thread-block
memory padding
bank idx
0
7
15
23
31
Address offset
Shared memory

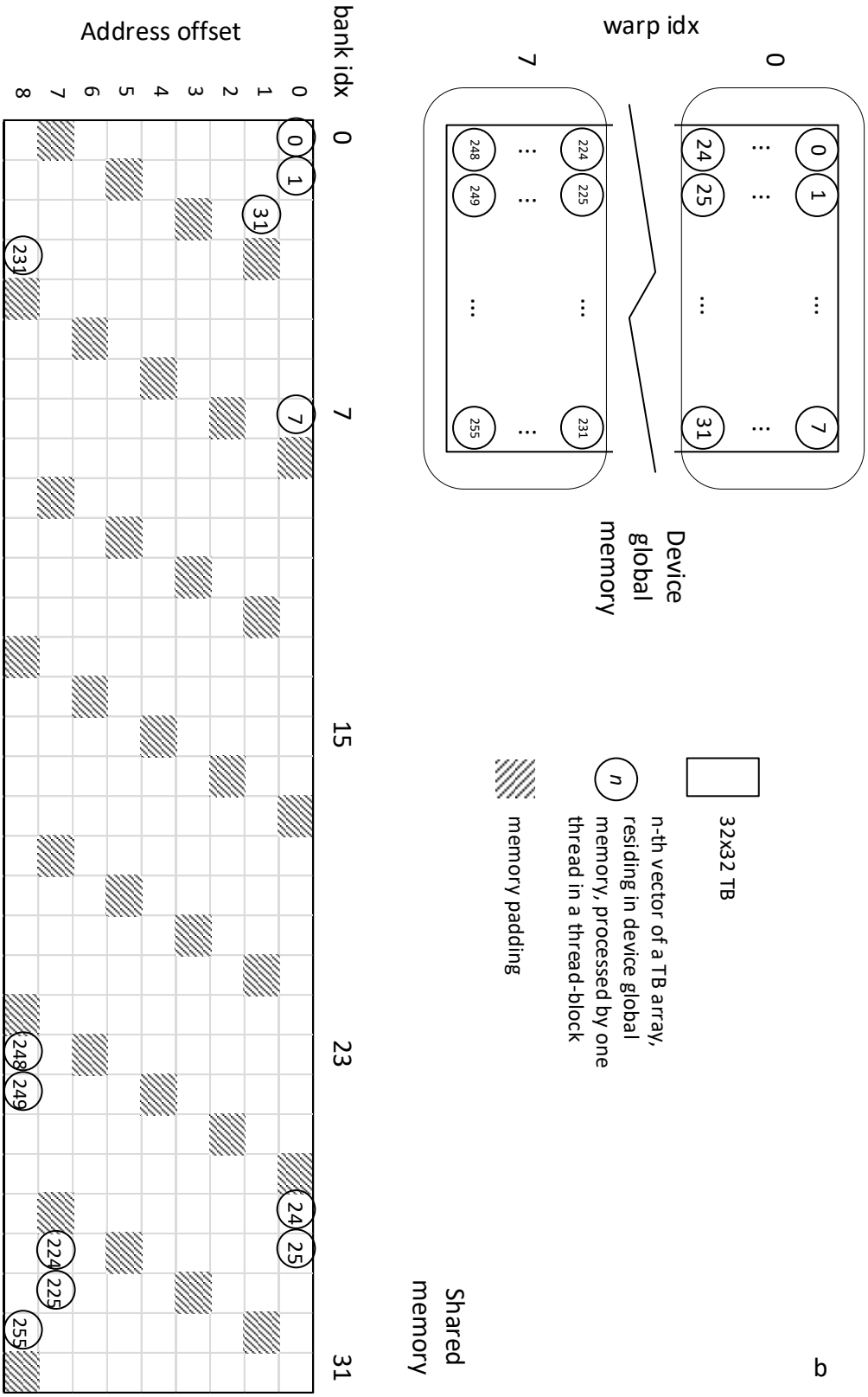


Figure 4.7: Padded shared memory allocated for the short4 data type with the access pattern for 4 × 4 (a) and 32 × 32 (b) transform blocks

4.6 Iterative implementation and evaluation

In this section, performance optimization techniques from previous sections, intended for highly parallel computation on GPUs, are implemented. Implementation is done in an iterative way by increasing implementation complexity. Work starts with combining the number and the scope of kernel functions related to existing stages in the HEVC transform and quantization process. A comparison with the NVIDIA's library methods follows to identify gains and gaps of the basic algorithm. Known CUDA basic and advanced optimization methods are tailored to meet joint characteristics of computations, executed for all transform sizes. Thereafter the optimal ratio for mapping TBs to thread-blocks is found and efficient transfer of data between the host and device is investigated. Lastly, an additional level of parallelization is obtained through overlapping data transfers with kernel execution using CUDA streams. A comparison with the related efficient parallelization of the HEVC DQIT kernel is carried out to place the results of this performance engineering in the context of other scientific contributions.

The performance of the proposed parallel implementation is always evaluated in terms of processing time per frame and compared with the HEVC TQ implementation on other computing platforms. Other performance metrics are included depending on the objectives set for every iteration. Objectives are presented at the beginning of every section. Thereafter measurement results are given and discussed.

The implementation framework is described in Section 4.4. TBs with random values were generated at the host side, transferred to the device and processed there. Quantized transform coefficients and the AZB identification array are sent back as process outputs to the host. The number of blocks matches two video resolutions: the Digital Cinema Initiatives (DCI) 4K and 8K Full Format. This number considers the 4:2:0 chroma sampling format and the same block sizes for all three-color components. Since frame resolutions for the two mentioned standards are 4096 × 2160 and 8192 × 4320, the number of blocks equals 12960 and 51840 respectively. The time required to process TQ in a frame is referred to as frame processing time.

4.6.1 Merging transform and quantization kernel

This section explores how execution configurations affect the performance of the transform and quantization kernel. Variants with separate kernels for transform stages and quantization are examined as well as a unique kernel which includes all operations made on a residual block.

Baseline application includes the first transform stage and operates with 32 × 32 TBs. The kernel comprises of two 1D integer transforms and scaling. A batch of residual blocks, the transform matrix, and its inverse are loaded from global memory and stored into the shared memory. Every thread-block has its own copy of transform matrices. No optimization is carried out. Execution configuration specifies a one-dimensional grid where the number of thread-blocks equals the number of residual blocks and the two-dimensional thread-block with 32 threads in each dimension.

Prior to baselining, an attempt was made with the kernel performing only one 1D integer DCT transform. Thus, transforming one residual block involved two kernel launches, each loading corresponding data blocks from global memory. Except for the overhead in memory accesses, there was also the high overhead cost associated with the kernel launch. The measured time between two kernel launches was about 50 μs, which is ten times more compared to a single kernel execution. This long time can be explained with the exchange of the program control between the host and device. After the GPU had finished the first kernel launch and first transform stage program control is returned to the CPU and then given to the GPU again for the second kernel.

Table 4.3: Comparison of the CPU, AVX2 and GPU implementation of the HEVC transform for 32 × 32 transform blocks and effective memory bandwidth for the GPU kernel

No. of 32 × 32 transform blocks	Processing time [ms]			BW_{Eff}	
	CPU	AVX2	GPU	Absolute [GB/s]	Relative %
12960	544.32	105.66	41.36	1.28	5

Measured performance for the baselined application is shown in Table 4.3. Compared to the CPU and AVX implementation, the GPU implementation exhibits speed ups 13.16 and 2.55 respectively. Theoretical global memory bandwidth for a mid-end GPU is computed as 25.92 GB/s. This value is lower than the one reported in the manual since the memory doesn't operate on maximum frequency. Effective memory bandwidth is 5% since every thread spends most time computing dot product where data are retrieved from the shared memory.

The attempt was made to exploit the device constant memory as a storage location for transform matrices. Kernel execution time was 288.47 ms, seven times slower compared to the baseline. The constant memory is not an appropriate solution for this application since all

threads do not retrieve the same data. In matrix multiplication, every result coefficient of indexes *i* and *j* is computed using row data of index *i* and column data of index *j*.

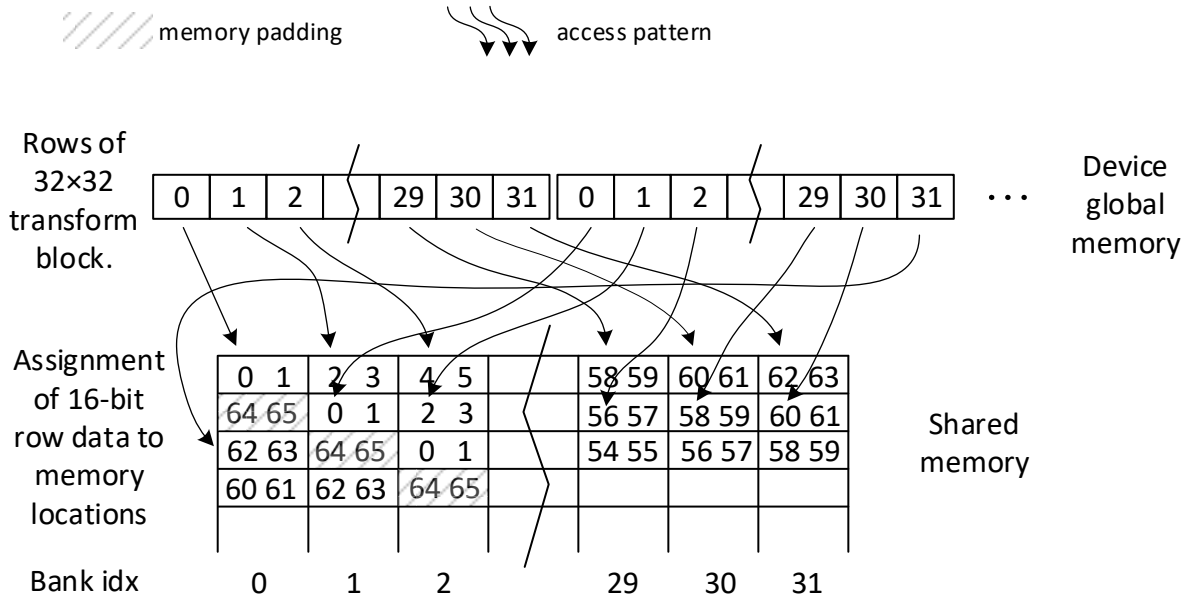


Figure 4.8: Shared memory access pattern with padding

In the first transform stage, thread retrieves row elements of the transform matrix and columns of the residual block to compute the dot product. Reading column data of 32 × 32 transform matrix from shared memory infers reading from the same memory bank and the occurrence of bank conflicts. To prevent this, memory padding is applied to eliminate bank conflicts. As all input data can be represented with a 16-bit signed integer, all matrices are declared with twice as many elements to reach an alignment with the memory bank's bus width. Declaration example is as follows:

```
__shared__ int16_t sA[32 * (64 + 2)];
```

where summand `2` inside the parenthesis denotes a padded number of 16-bit words. Access pattern with memory padding is shown in Figure 4.8. The following code line abstracts the data retrieval of block `A` from global memory and its store to shared memory as block `sA`.

```
sA[2 * threadIdx.y * (N + 1) + 2 * threadIdx.x] = A[blockDim.x * blockDim.y
* blockIdx.x + threadIdx.y * N + threadIdx.x];
```

Matrix elements are stored in the shared memory so that half of the 32-bit word remains empty. For subsequent accesses to shared memory, threads access half of the word in every bank.

In the second transform stage, the column data of inverse transform matrix are loaded. To prevent bank conflicts when accessing the second matrix, shared memory is not allocated for the transpose of the transform matrix and that matrix is not used in the process. Accessing column data of the transposed transform matrix of index *i* corresponds to accessing row data of the regular transform matrix of the same index. Table 4.4 shows the results of application benchmarking after the described optimization measures are implemented.

Table 4.4: Comparison of the CPU, AVX2 and GPU implementation of the HEVC transform for 32 × 32 transform blocks and effective memory bandwidth for the GPU kernel

No. of 32 × 32 transform blocks	Processing time [ms]			BW_{Eff}	
	CPU	AVX2	GPU	Absolute [GB/s]	Relative %
10	0.44	0.15	0.10	0.6	2
100	4.41	0.89	0.41	1.52	6
1000	44.11	7.99	3.25	1.89	7
12960	555.20	109.67	38.40	2.08	8
51840	2203.47	440.47	152.76	2.09	8

It can be noticed that the effective memory bandwidth tends towards 8% as the number of TBs grows. This is due to the decreasing impact of overhead time associated with a kernel launch. For referent implementation related to the DCI 4K video resolution the speed-up compared to CPU and AVX2 implementations is 14.48 and 2.86 respectively.

To validate the overhead related to the data transfer from the host to device and back, additional time measurement is involved in the evaluation. Except for the kernel execution time, the overall GPU processing time is measured (see Figure 4.2). Furthermore, when making comparisons with other implementations, the overall GPU processing time will be considered.

In this phase, the separate kernel function for quantization and AZB identification is included in the application. All one-dimensional arrays which represented both TBs and transform matrices are replaced with two-dimensional arrays. With such a formal change, the kernel execution time is decreased 1.02 times. To confirm the efficiency of the shared memory padding, padding is omitted from declarations of variables. Declaration example for block `sA` looks like as follows:

```
__shared__ int16_t sA[32 * 64];
```

As a result, kernel execution time is 3.95 times slower. Profiling tool reports a warning related to the shared memory access pattern. It suggests that one transaction would be sufficient to retrieve column elements, instead of the existing 16 transactions. Since 64-bit segments can be accessed per bank per cycle, 2 block elements per cycle are accessed. The reported shared memory efficiency is as low as 5.4%.

Table 4.5: Comparison of the CPU, AVX2 and GPU implementation of the HEVC transform for 32 × 32 transform blocks. The duration of data transfers is included in measurements for the GPU implementation.

No. of 32 × 32 transform blocks	QP	Frame processing time [ms]				Speed-up GPU overall over AVX2	Memory copy share [%]
		CPU	AVX2	GPU overall	GPU w/o data transfer		
12960	22	578.07	111.04	68.53	45.35	1.62	34
	27	585.22	111.50	69.33	45.21	1.61	35
	32	592.13	112.34	69.73	45.37	1.61	35
	37	606.27	112.82	69.82	45.52	1.62	35
51840	22	2332.06	447.61	272.76	178.88	1.64	34
	27	2343.35	453.61	273.35	179.71	1.66	34
	32	2390.99	447.21	272.41	179.17	1.64	34
	37	2418.51	448.32	272.38	179.56	1.65	34

For a better understanding of global memory, effective bandwidth and relations between memory bound and compute bound kernels, the transform code is commented out. The only remained computation was the arithmetic right shift of input and the result was written to the output block. Measured bandwidth is 15.128 GB/s or 58% relative to the theoretical bandwidth. When the kernel doesn't include operations that involve shared memory access, achievable bandwidth is typically in the range of 75% to 85% of the theoretical value. These measurements reported the effective memory bandwidth, and conclusions related to arithmetic intensity in

Section 4.5.1 show that kernel performance is bound by the effective shared memory bandwidth. This is thus a performance limiter and will be tackled in the next iterations.

With the application built upon two kernels, one for the transform and another for quantization and AZB identification, performance is measured once more, and results are given in Table 4.5 and Table 4.6.

The former shows frame processing times for three different implementations. All three include quantization and AZB identification. The overall GPU time covers data transfers and two kernel executions. It can be seen that the separate quantization kernel degraded the speed-up from 2.86 to 1.62 for the referent video resolution. The data transfer takes about one-third of the overall GPU processing time. In the quantization kernel, the QP value determines the index of the array, through which quantization multiplicand is retrieved, and the number of places to shift to the right. These scalar operations do not have an impact on measured time.

Table 4.6: Kernel execution time and effective memory bandwidth in the GPU implementation of the HEVC transform and quantization kernels for 32 × 32 transform blocks

No. of 32 × 32 transform blocks	QP	Kernel execution time [ms]		BW_{Eff}			
		2D transform	Quantization and AZB ident.	2D transform		Quant. And AZB ident	
				Absolute [GB/s]	Relative [%]	Absolute [GB/s]	Relative [%]
12960	22	37.90	7.45	2.10	8	10.69	41
	27	37.78	7.42	2.12	8	10.74	41
	32	37.86	7.51	2.10	8	10.60	41
	37	38.04	7.47	2.09	8	10.65	41
51840	22	149.53	29.35	2.13	8	10.85	42
	27	149.84	29.88	2.13	8	10.66	41
	32	149.81	29.35	2.13	8	10.85	42
	37	149.90	29.67	2.12	8	10.74	41

The latter presents a further analysis of the transform and quantization kernels. Compared to the quantization kernel, the transformation kernel takes about 5 times more time as it operates with more data and has intermediate access to the shared memory. The conclusion about less computation and shared memory accesses can be also drawn from measured effective memory bandwidths for two kernels.

Based on findings from previous experiments, a unique kernel for transform, quantization and AZB identification is constructed. Aside from avoiding one kernel launch, saving is also achieved in data transfer. Intermediate transform coefficients, obtained after the first transform stage, do not have to be transferred to the device anymore.

Table 4.7: Processing times and effective bandwidth in the GPU implementation for 32×32 transform blocks

No. of 32 × 32 transform blocks	QP	Frame processing time [ms]		BW_{Eff}		Data transfer	
		GPU overall	GPU w/o data transfer	Absolute [GB/s]	Relative [%]	Duration [ms]	Share in overall proc. time [%]
12960	22	56.51	39.14	0.25	1	17.40	31
	27	56.75	39.24	0.25	1	17.50	31
	32	56.44	38.82	0.26	1	17.60	31
	37	56.99	39.22	0.25	1	17.80	31
51840	22	227.30	155.21	0.26	1	72.10	32
	27	224.98	154.78	0.26	1	70.20	31
	32	229.78	154.89	0,26	1	74.90	33
	37	227.70	155.06	0.26	1	72.60	32

The quantization and AZB identification part of the kernel was also subjected to optimization. The intermediate two-dimensional array of the Boolean data type, allocated in shared memory for a thread-block, is replaced with the Boolean register in every thread. The array size was equal to the number of elements in TB and every element indicated if the matching level is zero. Quantization is a scalar operation and usage of shared memory is

excluded, except for the Boolean variable which indicated if a block of levels is non-zero. All threads in a TB accessed that same variable.

The iteration ended with the redesign of the kernel using the grid-stride loop method. Aside from easier debugging, scalability and thread reuse this method also provides full SM utilization over a number of configured thread-block. Performance results of an application with the unique TQ kernel are shown in Table 4.7. The relation between processing times for different implementations is additionally visualized in Figure 4.9.

The unique TQ kernel and made optimizations decreased the frame processing time for the GPU implementation. Compared to the AVX2 and CPU, the speed-ups are 2 and 10 times respectively. Effective memory bandwidth dropped to 1%. Such a low value indicated a very low percentage of operations with global memory and suggested that all further optimizations should address computation and shared memory accesses rather than global memory accesses. Data transfer share in overall computation decreased but is still above 30%.

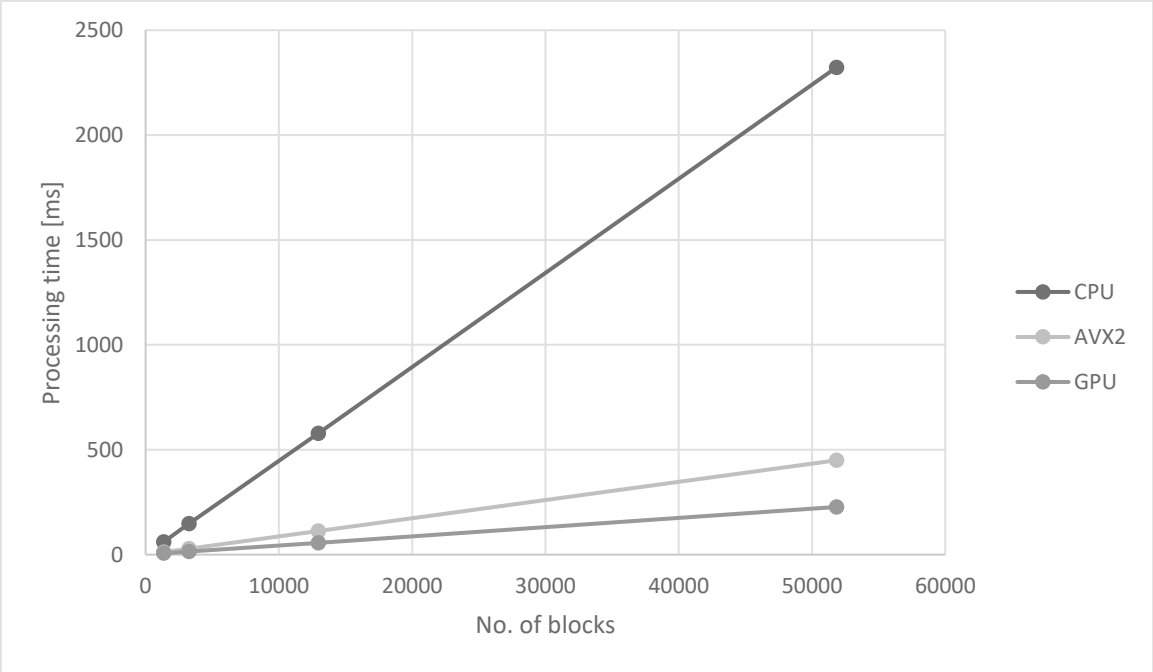


Figure 4.9: Comparison of frame processing times for different TQ implementations for 32 × 32 transform blocks

4.6.2 Benchmarking against NVIDIA's library functions

In this section, it is analyzed if there are library functions provided by the system manufacturer NVIDIA which can replace and enhance parts or whole HEVC TQ kernel. The CUDA basic linear algebra subroutine (cuBLAS) library [62] is used for this purpose. The

application is extended with the additional implementation named CUBLAS. Before calling the appropriate cuBLAS Application Programming Interface (API), data that are processed have to be transferred to the GPU memory. After the API is executed, the results have to be transferred back to the host. The cuBLAS functions are divided into three different levels depending on the type of operands: the BLAS1 functions perform scalar and vector-based operations, BLAS2 matrix-vector and BLAS3 matrix-matrix operations.

Before searching for appropriate library functions the existing kernel is analyzed to determine the timeshare of every computation stage in overall frame processing time i.e. identify computation critical stages. Three tests are carried out for three stages: the transform, quantization and AZB identification. A unique TQ kernel is reworked for every test so that two stages outside the test scope are commented out. Loading a residual block into the thread-block and allocation in shared memory, necessary for a particular test, is kept in every test so that the sum of results can't be 100%. After that, time measurements are collected and calculated portions are as follows:

- Transform 91%
- Quantization 10.6%
- AZB identification 12.4%

Since kernels spend most of the time on transformation, finding a library function for the 2D integer transform which is more efficient than one's own solution is the objective of this iteration. Quantization and AZB identification are excluded from analysis and thus commented out in all implementations. The GPU and CUBLAS implementations are in focus of this iteration, the CPU is kept as a referent and AVX2 is neither adjusted nor executed. It is included in the comparison and executed again during optimization based on the vectorized memory access.

The 2D integer transform is comprised of two 1D integer transforms followed by scaling. As cuBLAS doesn't support integer data sizes larger than 8-bit, the data type for all blocks and intermediate values is changed to a 32-bit floating-point. The 16-bit floating-point data are not used to avoid a compromise on accuracy. The data type change made scaling based on a right-shift obsolete. Scaling is therefore implemented as a scalar multiplication. The results obtained from the cuBLAS API in CUBLAS implementation are rounded in a separate kernel by using the command `floor(x + 0.5)` to obtain equal results for all implementations.

Preliminary or so-called warm-up calls of the transform function are added to every implementation. It has been shown that the execution time ratio between the GPU and CUBLAS implementation is 1:5.92 when there is no preliminary call and 1:1.04 when there is one.

When transforming TB data in a frame, a batch of square-shaped matrices with different values is multiplied with the matrix of predefined values. Two different cuBLAS APIs are taken into consideration for further analysis; `cublasSgemmBatched(…)` and `cublasSgemmStridedBatched(…)`. The latter considers that each batch of matrices, multiplier, multiplicand or product batch contains matrices that are stored consecutively in the device memory. Thus, it has three stride parameters that define address offsets between adjacent blocks in every batch and three pointers to the first instance of the batch. The former allows a more flexible storage of operands and results. There are no stride parameters, but rather three arrays of pointers where each pointer in an array points to each instance of the batch. Both APIs take and output matrices stored in column-major formats. Rearrangement from the row-major format, applicable for other implementations, to column-major format is made together with the device global memory allocation. Results are rearranged back to the row-major format before allocated space is freed.

Table 4.8: Comparison of the CPU, GPU and CUBLAS implementation of a 2D integer transform with scaling for 32 × 32 transform blocks

No. of 32 × 32 transform blocks	Processing time [ms]					Speed-up GPU over CUBLAS w/o data transfer
	CPU	GPU overall	GPU w/o data transfer	CUBLAS overall	CUBLAS w/o data transfer	
12960	563.57	63.99	38.89	1315.31	39.06	1.00
51840	2253.92	246.68	155.03	5468.81	156.32	1.01

A comparison of the GPU, CUBLAS, and referent CPU implementation of a 2D integer transform with scaling using `cublasSgemmBatched(…)` is showed in Table 4.8. Results show that the transform on a GPU based on one's own design is slightly faster than the CUBLAS based implementation. The poor performance of the CUBLAS implementation is caused by calls to the rounding kernel whose executions take about 48% of the processing time without the data transfer. When the rounding kernel is left out from the GPU and CUBLAS

implementation, the CUBLAS implementation is 1.38 times faster. Though it should be noted that the result is not HEVC compliant.

The performance of the data transfer is much worse for the CUBLAS implementation since data are transferred block by block using an array of pointers. Such transfer implies low bandwidth utilization of the PCI Express interface. The CUBLAS implementation built on `cublasSgemmStridedBatched(…)` uses a single pointer and stride parameter to access blocks within a batch and allows for an efficient data transfer. It is showed in Table 4.9 that processing with that cuBLAS API obtains better performance. The duration of the data transfer in both GPU implementations is about the same now. Although the rounding kernel still slows down the CUBLAS implementation, a more efficient API execution contributes to the overall speed-up.

Table 4.9: Comparison of the CPU, GPU and CUBLAS implementation of a 2D integer transform with scaling for 32 × 32 transform blocks

No. of 32 × 32 transform blocks	Processing time [ms]					Speed-up GPU over CUBLAS w/o data transfer
	CPU	GPU overall	GPU w/o data transfer	CUBLAS overall	CUBLAS w/o data transfer	
12960	554.01	57.49	34.44	52.85	30.10	0.87
51840	2229.39	233.00	136.52	212.16	116.75	0.86

The HEVC incompliant application, which includes two matrix-matrix multiplications and two scalings, and excludes result rounding, is repeated once more. It is shown that the CUBLAS implementation outperforms the GPU implementation 1.73 times in terms of overall processing time.

The results obtained in this iteration have driven optimization efforts towards further analysis and improvement of one's own transform implementation inside the TQ kernel.

4.6.3 Vectorized memory access with a float2 data type

The starting point for optimizations in this iteration was a low shared memory efficiency which equals 50%. The shared memory bus enables the transfer of 64-bits per bank in one cycle. With the bank size configured to four bytes, only half of that amount is transferred per bank, in both row and column access. The design objective is therefore to make the access to shared

memory more efficient. The quantization stage is activated again. AZB identification stage remained inactive until the design of the most complex part of the kernel was not finalized.

To better use the available bandwidth, data types of block variables in the TQ kernel are changed from float to float2. Vector loads and stores now replace scalar loads and stores. Declaration example for block `sA` in shared memory looks like

```
__shared__ float2 sA[32][16 + 1];
```

where the block `sA` is padded so that it has an extra column, which is a remedy for bank conflicts.

Data processing, matrix multiplication, scaling and quantization are reworked according to newly introduced vector data types. The number of threads in the y dimension remains unchanged and the number of threads in the x dimension is twice less than the matrix order as each thread processes two matrix elements. Mapping data, belonging to the top row and the most left column, from one TB to shared memory is shown in Figure 4.10. Those residual elements are processed by a top-left thread in a thread-block. An important feature of this access pattern is that, besides avoiding bank conflicts, 64-bits instead of 32-bits of data are accessed in one cycle per memory bank. It is expected that this will significantly increase the data rate.

Table 4.10: Comparison of the CPU, vectorized GPU and CUBLAS implementation of transform and quantization for 32 × 32 transform blocks

No. of 32 × 32 transform blocks	Frame processing time [ms]					Speed-up GPU over CUBLAS w/o data transfer
	CPU	GPU overall	GPU w/o data transfer	CUBLAS overall	CUBLAS w/o data transfer	
12960	626.94	44.19	18.31	57.50	31.15	1.70
51840	2512.88	180.47	75.34	221.29	119.98	1.59

Optimizations based on vectorization of the TQ kernel result in performance improvements as shown in Table 4.10. Compared to the previous iteration, the vectorized memory access and computation shorten the kernel execution time almost two times. This is an expected achievement since the kernel is mostly occupied with access to shared memory, whose throughput is increased twice. The GPU implementation outperforms the CUBLAS implementation, whether they are HEVC compliant or exclude rounding of scaled transformed coefficients and quantization. The latter yields a speed-up of 1.3 times for the DCI 4K video

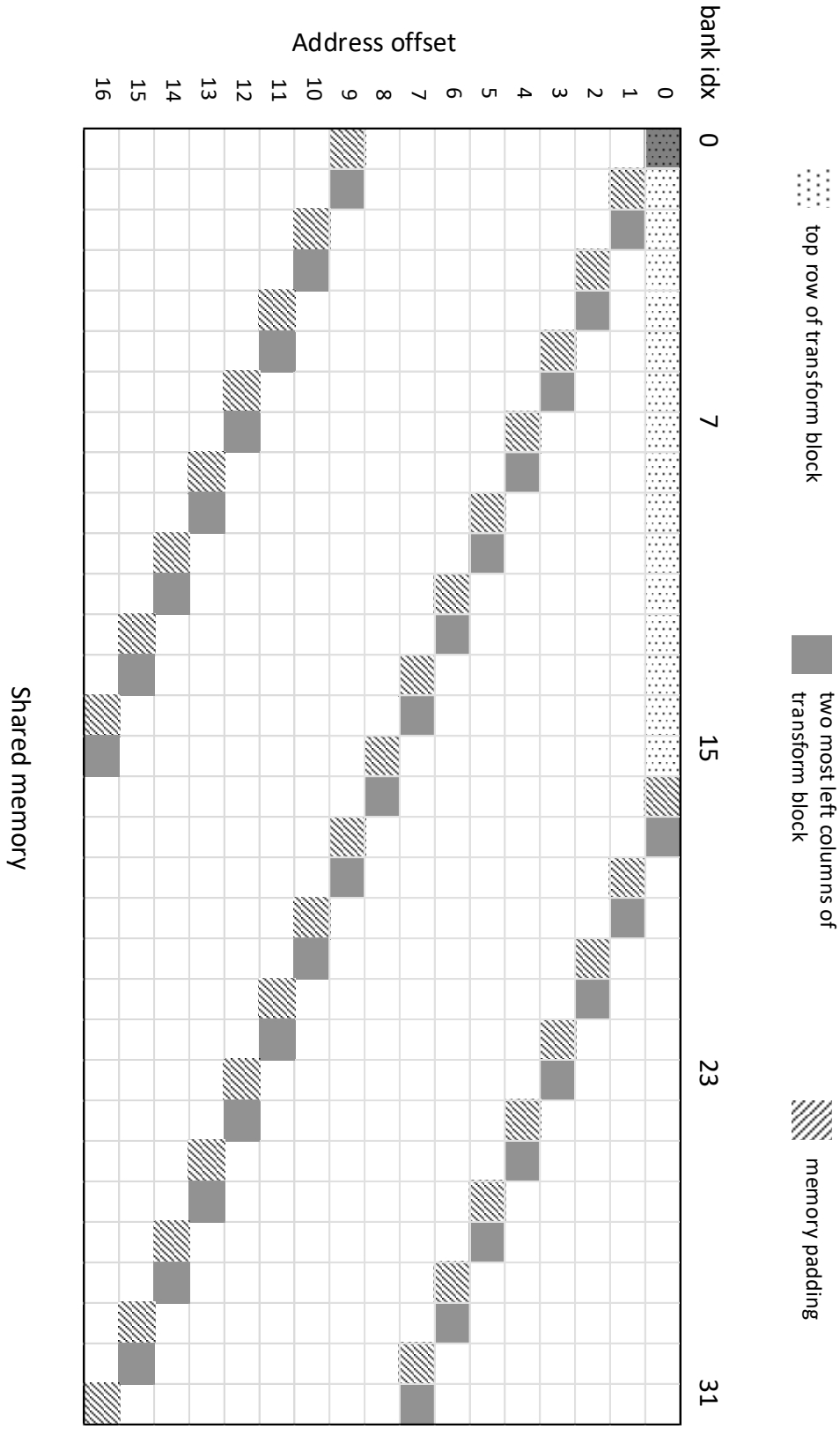


Figure 4.10: Distribution of data structured in shared memory using float2 data type and processed by the top left thread in a thread-block for 32 × 32 transform block

format. The profiling tool reported a shared efficiency of 97.9%. The kernel remained memory bound but compute utilization increased from 35% to 55%, and memory utilization dropped from 85% to 65%.

Vectorization didn't have any impact on the data transfer and its duration remained the same as in the previous iteration. Now with less processing time needed for TQ kernel in the GPU implementation, and thanks to vectorized memory access and computation, the data transfer share in overall processing time was increased to a substantial 59%.

4.6.4 Vectorized memory access with a short4 data type

In the previous section, a significant performance increase was achieved with the vectorized memory access and computation. This was made with a float2 vector data type. That type was selected to make a performance comparison between the GPU and CUBLAS implementations on equal terms. In this iteration, blocks will be declared to be of the integer data type since HEVC specifies integer transform and quantization. In the case of the CUBLAS implementation, a float2 data type will be kept and results will be rounded to get the same results as with other implementations. The AVX2 implementation is activated again with the change of output data type from int to short inside the quantization function.

As HEVC uses 16-bit data representation before and after each transform stage, a short4 vector data type can be used to get the maximum bandwidth of a single bank in the shared memory. Declaration example for block `sA` in shared memory looks like:

```
__shared__ short4 sA[32][8 + 1];
```

A thread-block that computes levels of a TB changes its x dimension to eight and y dimension remains thirty-two. All computations related to the two-component vectors are reworked for four-component vectors. Every thread now computes four outputs inside a TB row. Mapping pattern for a one-dimensional array of size 1024, containing residuals from one 32 × 32 TB, is shown in Figure 4.7b. The array is first transferred to the device global memory. At the beginning of kernel execution, it is stored to shared memory and its data are retrieved from there without bank conflict using the highest possible throughput.

A comparison of frame processing times for different implementations is shown in Table 4.11. Data transfers for the GPU implementation takes twice less time than for the CUBLAS implementation since the datum is represented with two times fewer bits. The speed-up of GPU kernel execution over CUBLAS kernel is 1.47 times for the referent video

resolution. When comparing the overall processing time, and not only kernel execution, the GPU implementations with a short4 data type is 1.31 times faster than with a float2 data type. The decrease in the speed-up compared to a GPU kernel with the float2 data type (Table 4.10) can be explained with computational throughput that is higher for the floating-point than for integer data [63].

Table 4.11: Comparison of the CPU, AVX2, GPU and CUBLAS implementations of transform and quantization for 32 × 32 transform blocks

No. of 32 × 32 transform blocks	Frame processing time [ms]						Speed-up GPU over CUBLAS w/o data transfer
	CPU	AVX2	GPU overall	GPU w/o data transfer	CUBLAS overall	CUBLAS w/o data transfer	
12960	828.19	113.12	33.63	21.14	55.42	30.99	1.47
51840	3318.97	454.04	133.22	83.93	221.63	119.50	1.42

After multiple optimization measures are implemented, the remaining transform sizes and associated TB dimensions have to be included in the analysis. The first kernel is extended to support a TQ of 4 × 4 TBs. The extension is also made in the CPU, GPU and CUBLAS implementations. With support for boundary transform sizes, performance issues can be identified and addressed for optimizations and conclusions can be made related to the overall performance. Up to now, one thread-block computed the levels and AZB flags for one 32 × 32 TB. If one kernel with the same execution configuration for all transform sizes is kept and one to one mapping from TB to thread-block remains effective across transform sizes, different threads will have to follow different execution paths. In this case, threads of the same warp diverge, execution paths are serialized, and instruction throughput and performance decrease. When the kernel is invoked for a small transform size there would be more threads that are configured, consume resources but do not do any computation, and warp divergency would decrease performance even more.

The impact of different mapping ratios on performance is investigated in Section 0. The transform matrix is left out from that investigation. Its instantiation in shared memory is separately discussed in Section 4.6.5. In the remaining part of this section, it is considered that the execution configuration is constant for all transform sizes and that mapping ensures that all

threads perform computation. This means that one thread-block in a kernel calls processes $1024/N^2$ TBs and that the same number of transform matrices are instantiated in shared memory. N in this expression denotes transform sizes.

Indexing TB arrays with thread-blocks and threads is modified to enable mapping dependent on the transform size. If indexing arrays with four elements per thread and usage of grid-stride loop technique are considered, the kernel code looks as follows:

```
int index = blockDim.x * blockDim.y * blockIdx.x + (threadIdx.y / N * (32 /
N) + threadIdx.x / (N / 4)) * N * N / 4 + threadIdx.y % N * (N / 4) +
threadIdx.x % (N / 4);
int stride = gridDim.x * blockDim.x * blockDim.y;

for (int k = index; k < nElements / 4; k += stride)
{
      sA[threadIdx.y][threadIdx.x] = A[k];

      //Transform and quantization
}
```

In this code snippet, `A` is a pointer to the batch of TBs in the device global memory, `sA` is a pointer to a sequence of TBs in shared memory, processed within a thread-block and `nElements` is the total number of residual samples. Kernel dependency on transform size is implemented using a function template where the transform size is passed as an argument to the template. In terms of kernel execution time, this implementation outperforms the implementation where transform size is an additional function parameter.

After application supported 4 × 4 TB, tests are repeated for both supported block sizes. Results are about the same for upper boundary TBs like in Table 4.11, so in Table 4.12 only results for lower boundary TBs are shown.

Table 4.12: Comparison of the CPU, GPU and CUBLAS implementations of transform and quantization for 4 × 4 transform blocks

No. of 4 × 4 transform blocks	Processing time [ms]					Speed-up GPU over CUBLAS w/o data transfer
	CPU	GPU overall	GPU w/o data transfer	CUBLAS overall	CUBLAS w/o data transfer	
829440	272.42	17.47	4.79	721.16	694.87	145.16
3317760	1088.36	71.84	19.09	2881.11	2779.36	145.56

The number of TBs considers that all color components of a frame for video formats DCI 4K and 8K Full Format are split to 4 × 4 TBs. Compared to results for the GPU implementation of TQ for 32 × 32 TBs, kernel execution time is significantly reduced from 21.14 ms to 4.79 ms. As the kernel became computation bound when two-component vectors were replaced with four-components vectors (95% compute utilization, 25% memory utilization), the performance started to depend heavily on computation complexity which is 4^3 times lower for the lower boundary size of TBs. The duration of the data transfer remained unchanged, so it occupied 73% of overall processing time. The profiling tool reported a comparable occupancy for two kernels with a value above 86%. Both kernels also have 100% global load and store efficiencies. Shared efficiency decreased, 97.6% was obtained for TQ of 32 × 32 TBs and now it is 73.8%. This performance gap was further investigated in the next two iterations. The lower shared efficiency is caused by memory padding which is not beneficial for the lower transform sizes as explained in Section 4.5.3. An additional optimization measure is not to apply mapping to the transform matrix. It could be sufficient to use one instance of it rather than a separate instance in shared memory for every TB in a thread-block.

Performance of the CUBLAS implementation for 4 × 4 TBs degraded considerably; it is even lower than the performance of the CPU implementation. Analysis using a profiling tool showed that for the 1D integer transform kernels were invoked thirteen times altogether. The first kernel, invoked twelve times, contains 65535 thread-blocks and the second, invoked once, 43020 thread-blocks. Thread-blocks in both kernels contain 128 threads. The overall number of thread-blocks in these two kernels equals the number of TBs but the number of threads indicates an excessive number of configured threads as there are only sixteen coefficients that have to be computed. Additionally, both kernels are characterized by a low global load and store efficiency, 33.3% and 50% respectively, a shared efficiency of 25.9% and low occupancy of 24.2%.

4.6.5 The single access transform matrix

In this section shared memory access patterns for the transform matrix will be analyzed. In the previous section, it was also considered that there are as many instances of the transform matrix in the shared memory as there are TBs mapped to one thread-block. In the case that only one instance of the transform matrix would exist, which all the threads in a thread-block would access when performing matrix multiplication, less memory space would be used. But it is

questionable whether this would allow faster access to the memory since address offsets are not the same for both arrays involved in the matrix multiplication. This iteration, therefore, aims to find a performance efficient access to predefined transform matrix data.

For the implementation of one instance of the transform matrix per thread-block, dynamic allocation in shared memory is necessary. Since the size of the allocated memory depends on transform size the variable `sDCT` is declared using the variable template (supported by C++14 compiler) as follows:

```
template <int N>
extern __shared__ short4 sDCT[][N / 4 + 1];
```

Indices for two-dimensional transform matrices are reworked since all TBs processed within one thread-block multiply one instance of the transform matrix. The following code snippet abstracts the instantiation of the 4 × 4 transform matrix in the shared memory.

```
if ((threadIdx.y < 4) && (threadIdx.x < 1))
      {
      sDCT<N>[threadIdx.y][threadIdx.x] =
reinterpret_cast<short4*>(d_D4)[threadIdx.y % N * (N / 4) + threadIdx.x %
(N / 4)];
      }
```

A conditional branch using an if statement has to be defined in the kernel for every transform size separately. In this code `d_D4` is the initial allocation of transform matrix in the device global memory. For the purpose of the vectorized access `d_D4` is recast to the short4 data type.

Table 4.13: Comparison of the CPU, GPU and CUBLAS implementations of transform and quantization for 4 × 4 transform blocks

No. of 4 × 4 transform blocks	Processing time [ms]					Speed-up GPU over CUBLAS w/o data transfer
	CPU	GPU overall	GPU w/o data transfer	CUBLAS overall	CUBLAS w/o data transfer	
829440	278.08	17.76	4.81	720.34	694.88	144.40
3317760	1094.86	69.87	18.77	2883.57	2779.34	148.07

The results for the single access transform matrix are shown in Table 4.13. Speed-up of 1.02 is achieved by using a single access instead of multiple instantiations. This improvement can be explained with fewer memory transactions, as the retrieved data can be reused among

the threads within a warp. Read accesses from threads in a warp, belonging to different TBs, to a single core transform array result in a performance efficient broadcast mechanism. The improvement is not as big since shared memory broadcasting is rewarded much less compared to avoiding bank conflicts. Shared efficiency increased from 73.8%, measured in the previous iteration, to 79.8%. Global load and store efficiency remained the same. Occupancy increased from 86.6% to 98.6%. Due to savings in allocated shared memory space (6.75 kB previously, 4.562 kB now) occupancy increased but this is not of big importance since the kernel is now compute bound.

4.6.6 Transform block to thread-block efficient mapping

In previous iterations, TB to thread-block mapping was set to $1024/N^2$: 1. With such a configuration, one thread-block processes one 32 × 32 TB, four sixteen 16 × 16 TB, sixteen 8 × 8 TB and sixty-four 4 × 4 TB. When these figures are scaled-up by four, full thread-block exploitation is achieved. As same kernel with configured execution configuration is invoked for all transform sizes, the mapping ratio is determined by the thread-block size and cannot be set separately for a particular transform size. The design objective in this iteration is to find a performance efficient mapping ratio. Data transfer is not considered. Besides the GPU no other implementation is executed. Resource consumption and occupancy will be tracked and discussed for every mapping configuration under test where the kernel execution time is set as a decisive parameter for performance efficiency.

Table 4.14: Processing times for different block mappings for 4 × 4 transform blocks

No. of 4 × 4 transform blocks	Mapping ratio	Kernel execution time [ms]	Registers per thread	Shared memory usage per block [bytes]	Occupancy [%]
829440	1:1	49.55	28	112	25
	64:1	5.02	32	4160	100
	128:1	5.60	32	8320	100
	192:1	7.59	31	12480	75
	256:1	8.71	32	16640	100

Table 4.14 shows processing time for several mapping ratios for 4 × 4 TBs including two boundary cases; when only one TB is processed within a thread-block and when the maximum thread-block size is configured. Configured number of threads depends on the mapping ratio. Fox example, as four threads process one TB, there are 512 threads configured for 128:1 block mapping.

Low occupancy for 1:1 block mapping is caused by the low utilization of thread blocks. Though the maximum number of thread-blocks resides on every SM, there is only one warp allocated per thread-block. Moreover, this one warp is not fully utilized. Only four of its threads are active. In the case of 64:1 block mapping, higher thread-block utilization yields higher performance, with a kernel speed-up of 9.8 times. As the mapping ratio further increases, so does the kernel execution time. This behavior is caused by the progressive reduction of global memory throughput. As more TBs are mapped to a thread-block, the stride of the kernel loop is larger and fewer memory transactions can be served from the L2 cache load. The L2 cache is shared by all SMs and used to cache accesses to global memory. This kernel is computation-bound and therefore the lower occupancy, in case of 192:1 block mapping, is not penalized with an additionally longer execution time. The block mapping of the ratio 64:1 with TB and thread indexing is illustrated in Figure 4.6. This thread-block size will be employed for all transform sizes in the rest of the chapter.

4.6.7 Page-locked memory transfer

Up to now, data allocations at the host side were pageable. Pageable host memory cannot be accessed from the GPU directly. Thus, when data were transferred from the host to the device they had to be copied to page-locked memory, where the GPU driver allocated necessary memory space, and then transferred data to the device memory. The described process is represented with the following code:

```
short *residual32 = new short[residual32Size];
short *d_residual32 = NULL;

cudaMalloc(&d_residual32, residual32Size * sizeof(short));
cudaMemcpy(d_residual32, residual32, residual32Size * sizeof(short),
cudaMemcpyHostToDevice));
```

where `residual32` and `d_residual32` are pointers to arrays of size `residual32Size` in the host and device memory respectively. Aforementioned intermediate allocation and copy to page-locked memory is part of the `cudaMemcpy` command execution. The cost of transfer

between pageable and page-locked arrays can be prevented with following modifications of the previous code:

```
cudaHostAlloc((void **)&residual32, residual32Size * sizeof(short), 0);
short *d_residual32 = NULL;

cudaMalloc(&d_residual32, residual32Size * sizeof(short));
cudaMemcpyAsync(d_residual32, residual32, residual32Size * sizeof(short),
cudaMemcpyHostToDevice));
```

With `cudaHostAlloc`, the CUDA API arrays are allocated directly in the page-locked memory. The last parameter of the API denotes a stream identifier. From page-locked memory, data are transferred across the PCI-e bus to the device memory with `cudaMemcpyAsync`, which is the opposite of the `cudaMemcpy` non-blocking CUDA API and the CPU continues with the execution of the next command. Synchronization using `cudaStreamSynchronize(0)` has to follow to distinguish the time between the data transfer from the kernel launch at the host side where the time is being measured.

Processing times changed using the page-locked data transfer for the GPU implementation and their values for the two boundary transform sizes are shown in Table 4.15. If the referent DCI 4K resolution is observed, the data transfer duration is shortened about 2.5 times (in comparison with values in Table 4.11 and Table 4.13) and its share dropped to 17% and 49% of the overall GPU processing time for 4 × 4 TBs and 32 × 32 TBs respectively.

Table 4.15: Comparison of the CPU and GPU implementations of transform and quantization for 4 × 4 and 32 × 32 transform blocks based on page-locked data transfers

No. of transform blocks		Processing time [ms]			Data transfer share in overall proc. time [%]	Data transfer speed-up
32 × 32	4 × 4	CPU	GPU overall	GPU w/o data transfer		
12960	0	776.41	26.92	22.34	17	2.58
51840	0	3134.82	109.93	88.73	19	2.14
0	829440	253.90	9.26	4.68	49	2.45
0	3317760	1009.70	40.31	18.74	54	2.09

4.6.8 Shared memory padding

In Section 4.5.3 it is described how memory padding affects shared memory performance for different transform sizes. It is shown that padding is not appropriate for smaller transform sizes and as such, it shouldn't be applied in those cases. In this section, those design hypotheses are confirmed with measurements. Only the GPU implementation is under test. For every transform size, two tests are conducted. In one, arrays associated with the first and second block involved in matrix multiplication and intermediate block product are padded so that they have an extra column. In the other the padding is omitted.

The impact on the kernel execution time for different transform sizes is shown in Table 4.16. As shown in the table, padding has to be applied for the upper half of transform sizes, whereas it has to be avoided for the lower half. Its absence most affects the shared memory efficiency in the case of the largest transform size. This happens because all threads within the warp access 32 elements from shared memory and if a column of data is accessed that results in 16-way bank conflict. In the case of the smallest transform size, each thread accesses four elements and in the worst case a 2-way bank conflict occurs.

Table 4.16: Execution times and shared memory efficiency with or without the shared memory array padding

Transform block size	Number of transform blocks (DCI 4K)	Shared memory padding	Overall processing time [ms]	Kernel execution time [ms]	Shared memory efficiency [%]
4 × 4	829440	No	9.69	4.99	81
		Yes	10.01	5.15	67.5
8 × 8	207360	No	12.59	7.89	90.2
		Yes	12.67	8.07	83.4
16 × 16	51840	No	17.92	13.26	69.9
		Yes	17.49	12.87	91
32 × 32	12960	No	30.04	25.37	26.7
		Yes	27.69	22.93	95.3

After optimization measures have been separately analyzed and implemented into the kernel and data transfer, the relation between different implementations is visualized in Figure 4.11 for the most complex transform size. This is done in order to have an overview of the performance improvement compared to the end of the first iteration (Section 4.6.1 and Figure 4.9). It can be seen that the GPU implementation outperforms others. Thanks to the optimization steps, progress has been made compared to the initial TQ processing time of 56.51 ms and is now 26.92 ms. In the last two iterations, a new level of parallelism, overlapping data transfers with the kernel execution, is investigated.

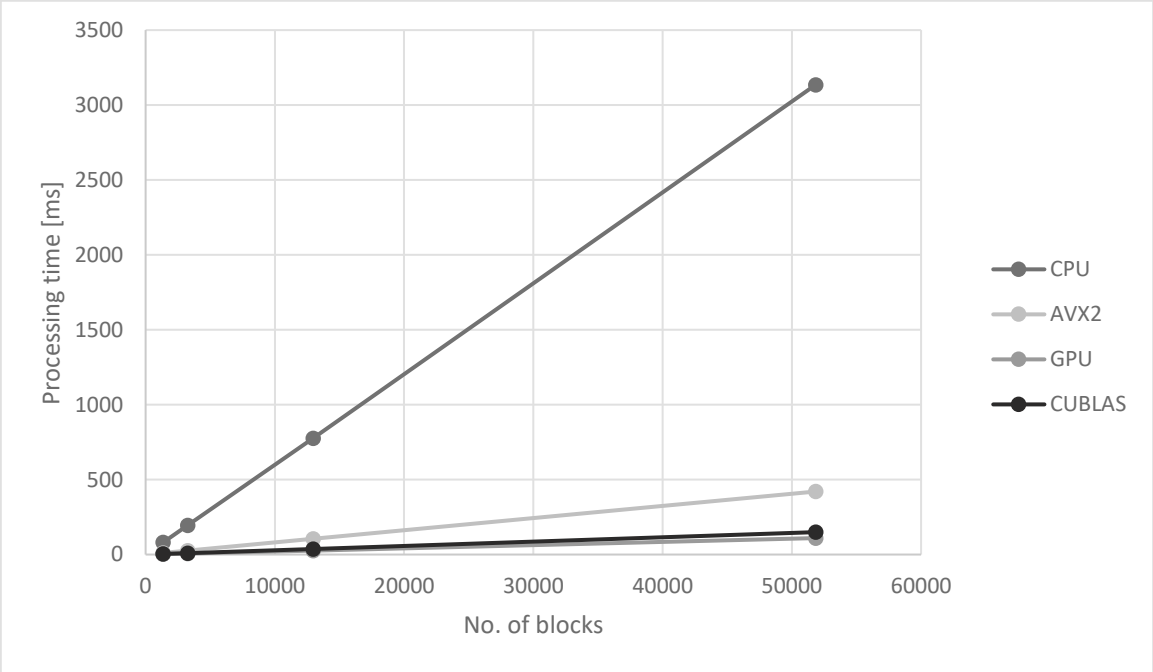


Figure 4.11: Comparison of frame processing times for different TQ implementations for 32 × 32 transform blocks after optimizations

4.6.9 Overlapping kernel execution and data transfer

In previous sections, efficient memory transfers and computation on the device were discussed. A memory transfer consists of two operations; transfer from the host to the device and transfer from the device to the host. In this section, the discussion is shifted towards the overlapping of all three operations or independent tasks, two memory transfers and computation on the device, for an even more efficient GPU implementation.

The application controls the concurrent tasks through *streams*. A stream is a sequence of commands that execute in the order in which they are invoked by the host code. Usage of the default or “null-stream” is implied when no stream is specified in the CUDA API call. The

default stream is synchronizing with respect to operations on the device; operations in the default stream do not start before ongoing operations in other streams finish and operations in other streams cannot start while the ongoing operation in the default stream is being executed. Thus, neither the data transfers nor kernel can be assigned to the default stream. Otherwise, these tasks would follow each other in time and their concurrent execution would not be accomplished. Remaining preconditions which have to be fulfilled to enable the overlapping are data stored directly to the page-locked memory and the device hardware capable of that. The latter is fulfilled for nearly all devices with a compute capability 1.1 and above.

Only the GPU implementation is in the scope of this iteration. All tests are conducted with the referent number of TBs related to the DCI 4K video format. They are broken up into chunks, which can be interpreted as horizontal frame segments and each chunk is assigned to its own stream. The objective of the section is to find the frame segmentation for which the highest performance in terms of frame processing time is achieved.

There are two ways to implement the TQ of frame segments using streams. The common feature for both is that all three tasks related to a segment are issued to the same stream according to their dependencies; the host-to-device transfer is followed by kernel execution which in turn is followed by the device-to-host transfer. So, there are so many streams as there are frame segments. In the first way, all three tasks related to a segment are first issued to one stream. Thereafter the same is done for the next segment. The characteristic of the second way is to issue tasks of the same type for different segments to different streams first. Device-to-host is thus issued for all segments first. Thereafter the kernels are issued to their streams.

The efficiency of a particular approach to frame segmentation depends on the GPU architecture [64]. Some architectures contain only one copy engine and therefore only one transfer at a time can be executed. For such devices, the first approach doesn't provide the desired overlapping. Host-to-device transfers for the following segments in the order of processing are blocked by device-to-host transfer for the segment which is currently processed. It is issued to the copy engine earlier and has to be processed earlier. Kernels, belonging to pending segments that depend on the related host-to-device transfers, cannot start in parallel with the device-to-host transfer from previous segments. Hence not only are the operations issued to a stream serialized, which they must be due to interdependence, but also the streams are serialized. So, there is no overlap here.

In the case of the second approach, the device-to-host tasks are not queued up into the copy engine before host-to-device tasks so they do not block them and in turn, they do not block the invocation of kernels which depend on their accomplishment. The GPU from the Desktop environment has an architecture with a single copy engine. The diagram in Figure 4.12 illustrates the execution of the TQ application on the Desktop environment for the two described work decompositions. Each approach to horizontal frame segmentation is represented with its own application version. Four streams are used for this purpose. The overall frame processing time is 29.24 ms for the first version and 24.33 ms for the second version. Time-saving is achieved through double overlapping which occurred for the second version. Kernel execution associated with $stream\ n$ is overlapped with the host-to-device transfers for all other streams and kernel executions associated with those other streams are overlapped with the device-to-host transfer from their preceding stream. This version is used for the next test where it is evaluated how a different number of CUDA streams affects performance. Separate tests are made for the frame split to all 32 × 32 TBs and 4 × 4 TBs to check if the optimal number of streams depends on TB distribution. The results are shown in Figure 4.13.

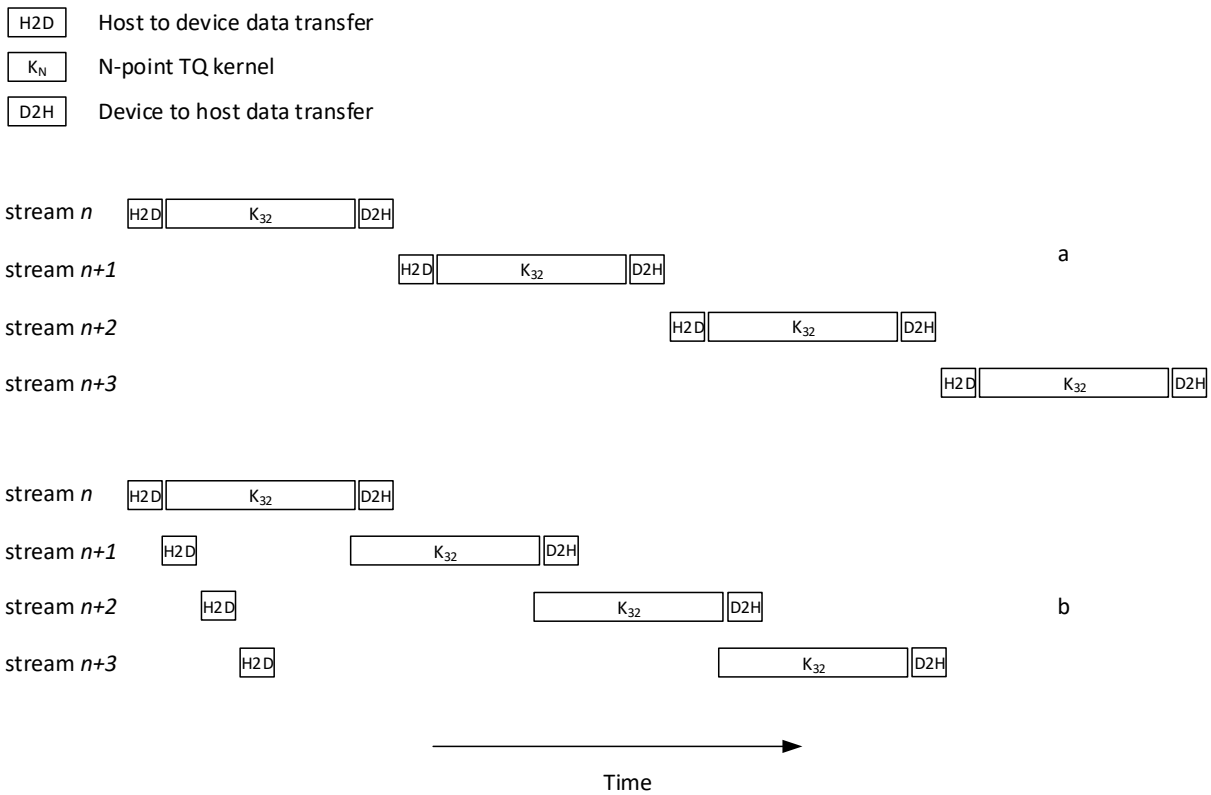


Figure 4.12: Execution timelines for two versions of the TQ application, implemented on the GPU. In version a) all operations related to one segment are issued and then the next segment is processed in the same way

and in version b) host-to device transfers for all segments are issued first, followed by all kernels and all device-to-host transfers.

When only one stream is used then the results obtained in previous sections with the default stream are repeated. Processing time is reduced 8% and 22% for 32 × 32 TBs and 4 × 4 TBs respectively when the two streams are specified. Further increments in the number of streams result in smaller reductions in processing times. Lowest processing times 23.73 ms and 6.08 ms are achieved when eight streams are used. Further increments in the number of streams keep the processing time at about the same value. When twenty streams are used, processing times are 23.91 ms and 6.53 ms and with sixty streams specified they are 25.73 ms and 7.35 ms. When comparing the minimum values of processing times for two transform sizes with their initial values it can be noticed that the reduction amount depends on the data transfer share in overall processing time for the GPU implementation (see Table 4.15). Kernels for different array chunks are issued to different streams and their execution cannot overlap. As the number of streams increases data work is partitioned to more data transfer and kernel operations where only data transfers overlap with the kernels. The performance gain achieved through the overlapping depends on the ratio between the data transfer duration and the duration of kernel execution. Closer this value is to one, the higher the speedup is.

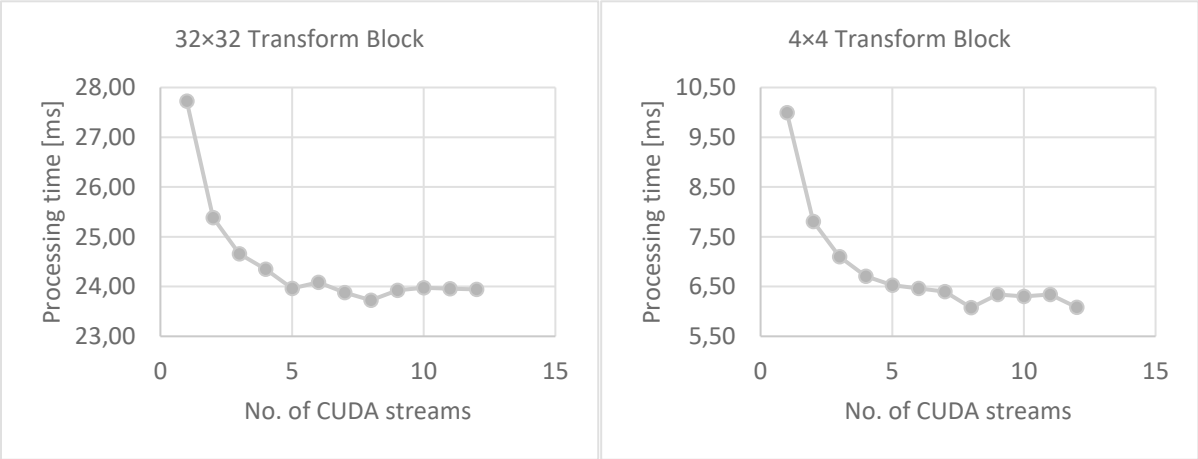


Figure 4.13: Impact of the number of CUDA streams on performance

4.6.10 Performance evaluation on the Workstation environment and comparison with a competing implementation

In the last section, performance is evaluated for the Workstation environment. It is analyzed how a high-end GPU of the same, Kepler architecture affects frame processing times. CPU and CUBLAS implementations and both executions, with and without overlapping are in

the scope of the section. The AVX2 instruction set is not supported on the Workstation's CPU and is left outside the scope. Besides the frame split to 4 × 4 and 32 × 32 TBs, the real-value block distribution, which emulates the frame split of real video sequences, is included in the test. These three distributions are named 4 × 4, 32 × 32 and are real-valued in the tests. Real-value block distribution refers to TB shares reported in [23]. For transform sizes 4 × 4 to 32 × 32 they are 58%, 32%, 8% and 2% respectively. When these values are represented through the number of TBs in the referent DCI 4K frame it corresponds to 108840, 60050, 15012 and 3754. To confirm the scientific contribution a performance comparison is made with a parallel implementation on a GPU from [51].

Table 4.17: Processing time comparison of the CPU and GPU implementations of transform and quantization for three transform block distributions on the Workstation environment

Transform block distribution	Video resolution	Frame processing time [ms]				
		CPU	GPU overall	GPU w/o data transfer	CUBLAS overall	CUBLAS w/o data transfer
32 × 32	DCI 4K	708.11	8.88	3.66	21.48	8.96
	8K Full Format	2832.67	36.19	14.01	76.38	35.84
Real-valued	DCI 4K	421.32	8.51	2.30	48.95	38.55
4 × 4	DCI 4K	246.26	6.03	0.79	113.84	101.75
	8K Full Format	979.30	23.94	2.96	447.65	407.12

Performance results for the GPU implementation without overlapping are shown in Table 4.17. and the calculated speed-ups and data transfer shares are shown in Table 4.18. Data transfers for both GPU implementations on the Workstation environment require the same time as on the Desktop environment since both environments have the same PCI-e bus type. Kernel execution times are shortened for two boundary transform sizes applied to the referent video format from 22.34 ms and 4.68 ms to 3.66 ms and 0.79 ms, which yields speed-ups of 6.10 and 5.93 respectively. These accelerations are achieved due to the 7.5 times larger amount of SMs. Since the kernel is compute bound now, the difference in the device global memory bandwidth in two environments doesn't have much of an impact. If the increase in the processing power

is multiplied with the ratio of core clock speeds (see Table 4.2), a factor of 6.2 is calculated. This value is close to the accelerations which were measured in the empirical analysis.

Table 4.18: Performance evaluation of the GPU implementation of transform and quantization for three transform block distributions on the Workstation environment

Transform block distribution	Video resolution	Speed-up GPU over CPU	Speed-up GPU over CUBLAS w/o data transfer	GPU data transfer share [%]
32 × 32	DCI 4K	79.75	2.45	59
	8K Full Format	78.28	2.56	61
Real-valued	DCI 4K	49.51	16.77	73
4 × 4	DCI 4K	40.85	128.68	87
	8K Full Format	40.90	137.61	88

The CUBLAS implementation exhibits the same performance issues as on the Desktop environment (see Table 4.11 and Table 4.13). Performance is still much lower compared to the GPU implementation when the frame is only split to 4 × 4 TBs. The acceleration considering an environment switch from the Desktop to Workstation is higher compared to the GPU implementation but only for 4 × 4 TBs and equals 6.82. For 32 × 32 TBs the acceleration equals 3.20 which is less than the expected value, that should be close to 6. Kernel invoked by the `cublasSgemmStridedBatched(…)` CUDA API is memory bound as one TB is mapped to one thread-block. When kernel performance reports from profiling tools are compared it can be noticed that memory utilization on the Desktop environment is around 85% whereas it is 35% on the Workstation environment. Considering the significantly higher memory bandwidth on Tesla K40c compared to GeForce GT 640 this indicates an inefficient global memory access.

Similar to Section 4.6.8 and Figure 4.11, the results for all implementations are visualized for the most complex transform size in Figure 4.14. A high-end GPU increased the performance gap between the CPU implementation and implementations on the GPU, which was expected considering its computing power. The GPU implementation again proved to be a better solution than the CUBLAS implementation in terms of processing time.

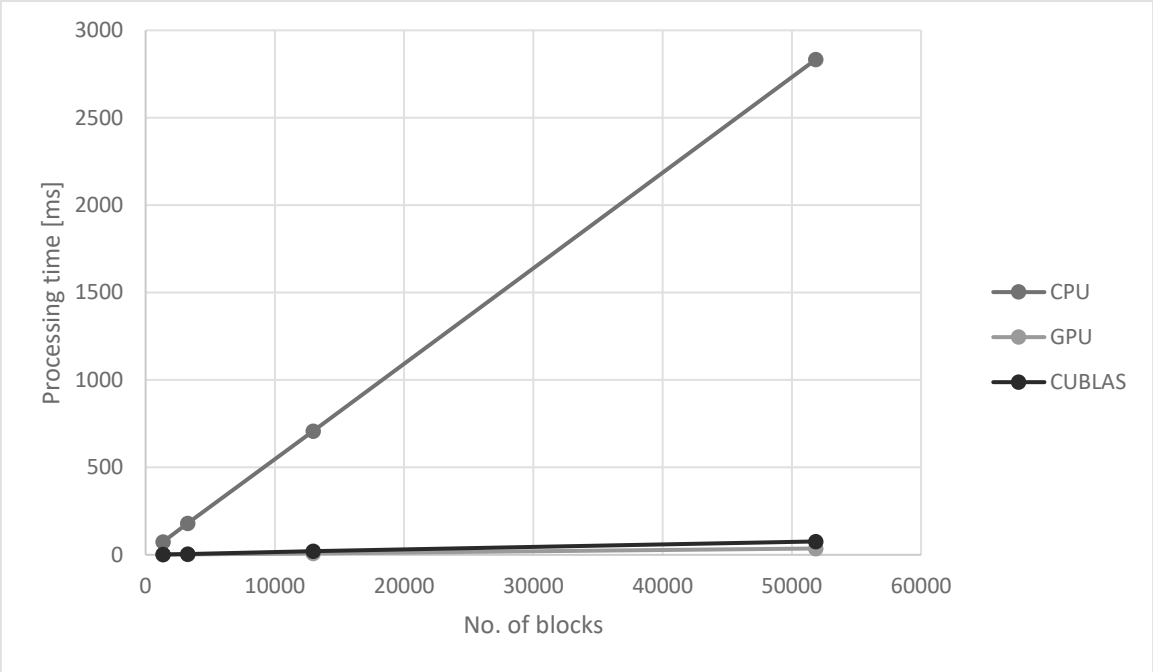


Figure 4.14: Comparison of frame processing times for different TQ implementations on the Workstation environment for 32 × 32 transform blocks

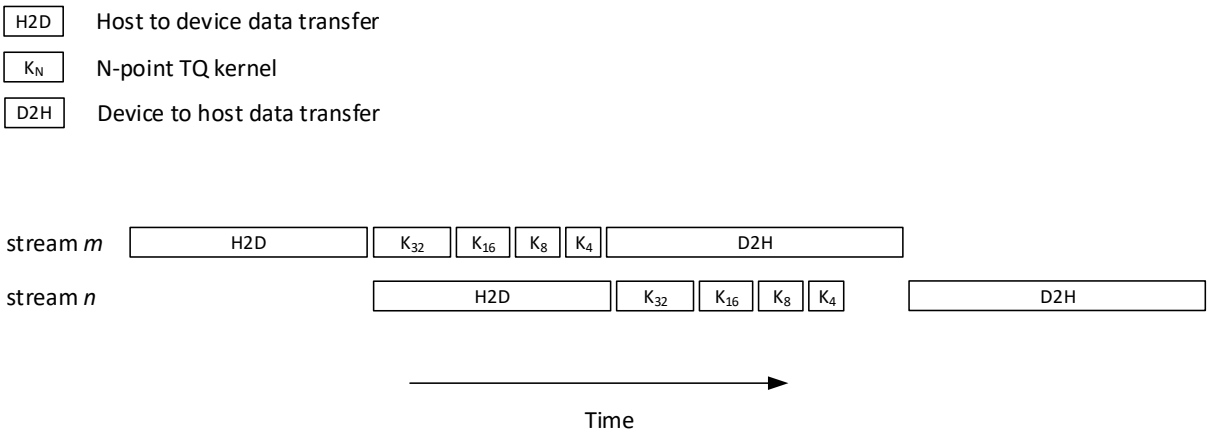


Figure 4.15: Overlapping TQ kernel execution and data transfers by using two CUDA streams

For a performance comparison with a competing work, the parallel implementation on GPU from [51] for the 3840 × 2160 frame resolution is selected. Though that proposal brings a parallel, fully HEVC compliant, implementation of DQIT, the share of control logic for bypassing TQ, handling the CBF and the transform skip in processing time can be assessed as neglectable. Moreover, the proposed implementation contains the AZB identification as an additional processing stage. As written before, the computational complexity for TQ and its inverse is the equivalent or lower. The OpenCL implementation used in the related work doesn't show a worse performance than CUDA [65]. To be able to have a fair comparison, kernel

execution is overlapped with data transfers in the proposed implementation by using CUDA streams. The frame is broken up into two segments with the execution flow shown in Figure 4.15.

Frame processing times for two parallel implementations are shown in Table 4.19. In the case of the proposed implementation, two block distributions are considered, real-valued and 32 × 32, as a worst-case distribution related to performance. Time comparison is made against the best time for a given resolution in the competing proposal. As can be observed, the proposed implementation achieved a speedup as high as 1.22 times.

By observing Figure 4.15 it can be noticed that there is a possibility to further reduce the processing time by overlapping two data transfers with the kernel execution, since these three tasks take a similar time. The test with passing through different numbers of streams and finding minimum processing time is therefore repeated on the Workstation environment. The capability of the TQ kernel for real-time applications is validated as well. The 32 × 32 block distribution for the DCI 4K video frame is considered in the test.

Table 4.19: Comparison with a related implementation

Implementation	Encoder config / TB distribution	Processing time [ms]
De Souza [51]	"DucksTakeOff" B Frame, Random Access	6.56
Proposed	32 × 32	6.17
	real-valued	5.38

It is measured that the best performance is achieved with nine streams. Processing time equals 4.82 ms. The speed-up compared to the configuration with one CUDA stream is 1.84 times. Resulting processing times expose the capability of this implementation to be used in real-time applications. For a frame rate of 50 fps, each frame has to be processed in less than 20 ms. Considering the decoding time distribution for DQIT functions in an all-intra configuration [19], the remaining time would be sufficient for other decoding stages to be done.

5 CONCLUSION

Performance optimized architectures and algorithms of transform and quantization computation blocks for video compression targeted to heterogeneous multiprocessor high performance computers were explored in the scope of research that was carried out for this thesis. Such researches are motivated by the continuous growth of video traffic and video services, especially in the field of high resolution and high-quality video content. Deployment of UHD video content, its representation precision, a higher dynamic range and a wider range of representable colors place heavy demands on video compression, which must be more efficient to lessen the burden that falls on storage and transmission systems. The required compression capability or coding efficiency is achieved with an extended set of coding tools, advanced algorithms and sophisticated computational methods. When this is compounded by the demands of a broad range of real-time video processing applications their need for reduced communication delays, the implementation of algorithms and methods on high-performance computing systems becomes a necessity. Computing systems based on CPUs spend a lot of power on non-computational units, do not provide enough parallelism and cannot be utilized for such innovative applications. Heterogenous multiprocessor high performance computers aim to address these issues. They deliver improvements in power efficiency, achieve a performance gain through parallelism, but also pose new challenges not found in typical homogeneous systems. The application has to be decomposed, tasks are portioned to those computing units that provide the best performance or power-performance ratio for the overall application. This inevitably leads to non-uniformity in system development and programming practices. Overheads in task initiating and data transfer must be considered, and synchronization carried out in a timely manner.

Transform and quantization are very important parts of not only video coding systems but multimedia compression systems in general. Mapping signals from spatial into the transform domain provides energy compaction and decorrelation. Once this is obtained, spectral components contributing less to video quality are removed during quantization. DCT is the most widely used lossy compression method by reason of its properties, which contribute to both compression efficiency and efficient implementation. Driven by a further increase in implementation efficiency and interoperability, modern compression standards adopt integer approximation to the DCT. One such transformation is the one specified in the state-of-the-art video coding standard HEVC which was investigated as a case study in this thesis.

Performance optimization in terms of area, power and reusability for systems with limited resources and low power consumption was a research direction for the heterogeneous system with the accelerated transform and quantization on FPGA. The multiply-accumulate architecture is built on the regular lane design where lanes are reused and shared among all supported transform sizes. Computation of one scaled transform coefficient is made within one lane, which processes the added/subtracted symmetric residual values and involves multiplexed MCM and accumulation with scaling. The result values from 32 lanes are serialized to have the appropriate data format at the output. Additionally, an architecture variant with a parallel output is proposed. Concerning the hardware cost, it outperforms related architectures. The proposed architecture can encode 4K UHD@30fps in real-time.

Another research direction dealt with the performance optimized software implementation for the most common heterogeneous system with a CPU and GPU as the processing unit specialized for highly parallel computation. The performance engineered transform and quantization kernel is presented to be executed on GPU accelerators. All the relevant state-of-the-art techniques related to kernel vectorization, shared memory optimization and overlapping data transfers with computation were investigated and carefully combined to obtain a performance efficient solution across all applicable transform sizes. Experiments were conducted using two different GPUs. Substantial gains are shown against the CPU, cuBLAS and AVX2 implementations, and improvements are also demonstrated against state-of-the-art GPU solutions.

In this thesis, three original scientific contributions were achieved:

1. Design of performance optimized software implementation of transform and quantization computation blocks for video compression targeted to heterogeneous multiprocessor high performance computers. The GPU implementation exhibits speed-ups up to 80, 19, 4 and 1.22 times compared to the CPU, cuBLAS, AVX2 and related highly parallel implementations respectively. The proposed implementation can support the 4K UHD@50fps real-time decoding process.
2. Design of performance optimized hardware implementation of transform and quantization computation blocks for video compression targeted to heterogeneous multiprocessor high performance computers. The FPGA implementation achieves throughput of 1 Gsps, with a 9.6% lower hardware cost compared to the related

implementation. The proposed architecture can be used to decode4K UHD@30fps in real-time.

3. Methodology development for efficiency validation of performance optimized software and hardware computation blocks implementations. For efficiency validation of the performance optimized GPU implementation, the kernel execution time and frame processing time were compared with other implementations and between different development iterations. To identify optimization opportunities, performance metric also included the device global memory bandwidth, occupancy and shared memory efficiency. For the area efficient FPGA implementation, the maximal throughput is computed and evaluated related to the hardware cost which had to stay low. For both proposed implementations, usability for real-time applications is analyzed.

Obtained findings and results provide a fertile ground for further research. Considering a hardware-based kernel the changes to the front-end when using different bus-widths (e.g. 64 bits) and its impact in the scaling of the architecture can be investigated. Subsequently, integration aspects of architecture can be analyzed, and the proposed solution of the problem can be focused in the context of the energy consumption of the accelerator. ASIC device design can help to fully benchmark the performance of this approach.

In the case of the GPU implementation, other optimization techniques can be explored to increase the accelerator's performance. To maximize the memory throughput, texture memory space and texture cache which is optimized for 2D spatial locality can be used. Double buffering by loading data into one shared memory, while operating on another, can be an efficient solution when the kernel would be extended to execute an inverse transform and dequantization.

REFERENCES

[1] Cisco Visual Networking Index: Forecast and Methodology, 2017–2022, available at https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html, (18th Dec. 2019)

[2] Ultra Video Group Test Sequences, available at http://ultravideo.cs.tut.fi/#testsequences, (23rd Mar. 2020)

[3] Joint Collaborative Team on Video Coding Reference Software ver. HM 16.20, available at https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/tags/HM-16.9/, (23rd Mar. 2020)

[4] Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2017–2022, available at https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-738429.html, (18th Dec. 2019)

[5] Sullivan G. J., Ohm J. R., Han W. J., Wiegand T., "Overview of the High Efficiency Video Coding (HEVC) Standard," in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1649-1668, Dec. 2012.

[6] Grois D., Nguyen T., Marpe D., "Performance comparison of AV1, JEM, VP9, and HEVC encoders," in *Proceedings SPIE 10396, Applications of Digital Image Processing XL*, pp. 68-79, San Diego 2017.

[7] Kim B.-G., Goswami K., "Basic Prediction Techniques in *Modern Video Coding Standards*", Springer, 2016.

[8] Ahmed N., Natarajan T., Rao K. R., "Discrete Cosine Transform," in *IEEE Transactions on Computers*, vol. C-23, no. 1, pp. 90-93, Jan. 1974.

[9] Britaniak V., Yip P. C., Rao K. R., "The Karhunen-Loève transform and optimal decorrelation" in *Discrete cosine and sine transforms: General properties, fast algorithms and integer approximation*, Academic Press, 2006.

[10] Sundararajan D., "The Discrete Fourier Transform" in *The Discrete Fourier Transform: Theory, Algorithms and Applications*, World Scientific, 2001.

[11] Miao G. J., Clements M. A., "Discrete Time Transforms" in *Digital Signal Processing and Statistical Classification*, Artech House, 2002.

[12] Salomon D., "Image Compression" in *Data Compression: The Complete reference*, Springer, 2007.

[13] Xiph Video Test Media, available at https://media.xiph.org/tearsofsteel/tearsofsteel-4k-tiff/, (10th Jan.2020)

[14] Richardson I., "Transform Coding" in *Video Codec Design: Developing Image and Video Compression Systems*, Wiley, 2002.

[15] Wang Y., Zhang Y. Q., Ostermann J., "Foundations for Video Coding" in *Video Processing and Communications*, Prentice Hall, 2002.

[16] "ITU-T Rec. H.265 and ISO/IEC 23008-2: High efficiency video coding," ITU-T and ISO/IEC, 2013.

[17] Han J., Saxena A., Melkote V. and Rose K., "Jointly Optimized Spatial Prediction and Block Transform for Video and Image Coding," in *IEEE Transactions on Image Processing*, vol. 21, no. 4, pp. 1874-1884, Apr. 2012.

[18] Sze V., Budagavi M., Sullivan G.J., "HEVC Transform and Quantization" in *High Efficiency Video Coding (HEVC)*, Springer, 2014.

[19] Bossen F., Flynn D., Bross B., Suhring K., "HEVC Complexity and Implementation Analysis," in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1685-1696, Dec. 2012.

[20] Budagavi M., Fuldseth A., Bjøntegaard G., Sze V., Sadafale M., "Core Transform Design in the High Efficiency Video Coding (HEVC) Standard," in *IEEE Journal of Selected Topics in Signal Processing*, vol. 7, no. 6, pp. 1029–1041, Dec. 2013.

[21] Tikekar M., Huang C.-T., Juvekar C., Chandrakasan A., "Core Transform property for Practical Throughput Hardware Design," *7th meeting of the Joint Collaborative Team on Video Coding (JCT-VC)*, Geneva 2011.

[22] Fuldseth A., Bjøntegaard G., Sze V., Budagavi M., "Core transform design for HEVC," in *7th meeting of the Joint Collaborative Team on Video Coding (JCT-VC)*, Geneva 2011.

[23] Hong L., He W., Zhu H., Mao Z., "A cost effective 2-D adaptive block size IDCT architecture for HEVC standard," in *2013 IEEE 56th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 1290-1293., Columbus 2013.

[24] Čobrnić M., Duspara A., Dragić L., Piljić I., Mlinarić H., Kovač M., "An Area Efficient and Reusable HEVC 1D-DCT Hardware Accelerator," in *13th International Conference on Parallel Processing and Applied Mathematics (PPAM)*, pp. 199-208, Bialystok 2019.

[25] Meher P. K., Park S. Y., Mohanty B. K., Lim K. S., Yeo C., "Efficient Integer DCT Architectures for HEVC," in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 24, no. 1, pp. 168-178, Jan. 2014.

[26] Zhao W., Onoye T., Song T., “High-performance multiplierless transform architecture for HEVC,” in *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp.1668–1671., Beijing 2013.

[27] Chatterjee S., Sarawadekar K. P., “A low cost, constant throughput and reusable 8X8 DCT architecture for HEVC," *2016 IEEE 59th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 1-4., Abu Dhabi 2016.

[28] Bolaños-Jojoa J. D., Velasco-Medina J., “Efficient hardware design of N-point 1D-DCT for HEVC," *2015 20th Symposium on Signal Processing, Images and Computer Vision (STSIVA)*, pp. 1-6., Bogota 2015.

[29] Abdelrasoul M., Sayed M. S., Goulart V., “Scalable Integer DCT Architecture for HEVC Encoder,” *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 314-318., Pittsburgh 2016.

[30] Sjövall P., Viitamäki V., Vanne J., Hämäläinen T. D., “High-level synthesis implementation of HEVC 2-D DCT/DST on FPGA,” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1547-1551., New Orleans 2017.

[31] Arayacheeppreecha P., Pumrin S., Supmonchai B., “Flexible input transform architecture for HEVC encoder on FPGA,” *2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, pp. 1-6., Hua Hin 2015.

[32] Abdelrasoul M., Sayed M. S., Goulart V., “Real-time unified architecture for forward/ inverse discrete cosine transform in high efficiency video coding,” in *IET Circuits, Devices & Systems*, vol. 11, no. 4, pp. 381-387, Jan. 2017.

[33] Bjøntegaard G., “Calculation of average PSNR differences between RD-curves,” *16th meeting of the Video Coding Experts Group (VCEG)*, Austin 2001.

[34] Renda G., Masera M., Martina M. and Masera G., “Approximate Arai DCT Architecture for HEVC,” *2017 New Generation of CAS (NGCAS)*, pp. 133-136., Genova 2017.

[35] UltraFast Design Methodology Guide for the Vivado Design Suite (UG949), available at https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug949-vivado-design-methodology.pdf, (11th Jan. 2020)

[36] Vivado Design Suite Tcl Command Reference Guide (UG835), available at https://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_4/ug835-vivado-tcl-commands.pdf, (10th Jan. 2020)

[37] Vivado Design Suite User Guide: Synthesis (UG901), available at https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug901-vivado-synthesis.pdf, (10th Jan. 2020)

[38] Vivado Design Suite User Guide: Implementation (UG904), available at https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug904-vivado-implementation.pdf, (10th Jan.2020)

[39] Tummeltshammer P., Hoe J. C., Puschel M., "Time-Multiplexed Multiple-Constant Multiplication," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 26, no. 9, pp. 1551–1563, Sept. 2007.

[40] Spiral project: Multiplexed Multiplier Block Generator, available at http://spiral.net/hardware/mmcm.html, (10th Dec. 2018)

[41] Hong L., He W., Zhu H., mao Z., "A full-pipelined 2-D IDCT/IDST VLSI architecture with adaptive block-size for HEVC standard," *IEICE Electronics Express*, vol. 10, no. 9, pp. 1–11, 2013.

[42] Liu F., Liang Y., Wang L., "A Survey of the Heterogeneous Computing Platform and Related Technologies," *2016 International Conference on Informatics, Management Engineering and Industrial Application (IMEIA)*, Phuket 2016.

[43] Čobrnić M., Duspara A., Dragić L., Piljić I., Kovač M., "Performance Engineering for HEVC transform and quantization kernel on GPUs," in *Automatika Journal for Control, Measurement, Electronics, Computing and Communications*, vol. 61, no. 3, pp. 325-333, 2020.

[44] Xiao W., Li B., Xu J., Shi G., Wu F., "HEVC Encoding Optimization Using Multicore CPUs and GPUs," in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 11, pp. 1830-1843, Nov. 2015.

[45] Wang F., Zhou D., Goto S., "OpenCL based high-quality HEVC motion estimation on GPU," *2014 IEEE International Conference on Image Processing (ICIP)*, Paris, 2014, pp. 1263-1267.

[46] De Souza D. F., Ilic A., Roma N., Sousa L., "GHEVC: An Efficient HEVC Decoder for Graphics Processing Units," in *IEEE Transactions on Multimedia*, vol. 19, no. 3, pp. 459-474, Mar. 2017.

[47] Mohamed B., Elsayed A., Amin O., Khafagy E., Abdelrasoul M., Shalaby A., sayed M. S., "High-level synthesis hardware implementation and verification of HEVC DCT on

SoC-FPGA," *2017 13th International Computer Engineering Conference (ICENCO)*, pp. 361-365, Cairo 2017.

[48] He L., Goto S., "A high parallel way for processing IQ/IT part of HEVC decoder based on GPU," *2014 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, pp. 211-215, Kuching 2014.

[49] Igarashi H., Takano F., Moriyoshi T., "Highly parallel transformation and quantization for HEVC encoder on GPUs," *2016 Visual Communications and Image Processing (VCIP)*, pp. 1-4, Chengdu 2016.

[50] Masoumi M., Ahmadifar H., "Performance of HEVC discrete cosine and sine transforms on GPU using CUDA," *2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, pp. 0857-0861, Tehran 2017.

[51] De Souza D. F., Roma N., Sousa L., "OpenCL parallelization of the HEVC de-quantization and inverse transform for heterogeneous platforms," *2014 22nd European Signal Processing Conference (EUSIPCO)*, pp. 755-759, Lisbon 2014.

[52] Ryoo S., Rodrigues C. I., Baghsorkhi S. S., Stone S. S., Kirk D. B., Hwu W. W., "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA" *13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP)*, pp. 73-82, New York 2008.

[53] Cui X., Chen Y., Mei H., "Improving Performance of Matrix Multiplication and FFT on GPU," *2009 15th International Conference on Parallel and Distributed Systems*, pp. 42-48, Shenzhen 2009.

[54] Volkov V., Demmel J. W., "Benchmarking GPUs to tune dense linear algebra," *2008 ACM/IEEE Conference on Supercomputing*, pp. 1-11, Austin 2008.

[55] NVIDIA Corp.: CUDA Compute Unified Device Architecture, Programming Guide, Version 10.1.243, available at https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, (19th Aug. 2019)

[56] NVIDIA Corp.: CUDA Occupancy Calculator, Version 10.1.243, available at https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html, (19th Aug. 2019)

[57] Harris M., "How to optimize data transfers in CUDA C/C++," NVIDIA Developer Blog, available at https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/, (19th Aug. 2019)

[58] Harris M., "CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops," NVIDIA Developer Blog, available at: https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/, (19th Aug. 2019)

[59] Piljić I., Dragić L., Duspara A., Čobrnić M., Mlinarić H., Kovač M., "Bolt65 – performance-optimized HEVC HW/SW suite for Just-in-Time video processing," *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 966-970, Opatija 2019.

[60] Micikevicius P., "Performance Optimizations" lecture in *High Performance Computing with CUDA* tutorial, *2011 Supercomputing (SC)*, Seattle 2011., available at: https://www.nvidia.com/docs/IO/116711/sc11-perf-optimization.pdf, (20th Feb. 2020)

[61] Luitjens J., "CUDA Pro Tip: Increase Performance with Vectorized Memory Access," NVIDIA Developer Blog, available at: https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/, (28th Jan. 2020)

[62] NVIDIA Corp.: cuBLAS, Version 10.2.89, available at https://developer.nvidia.com/cublas, (19th Aug. 2019)

[63] NVIDIA Corp.: "Performance Guidelines" in Programming Guide v10.2.89, available at https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions, (5th Feb. 2020.)

[64] Harris M., "How to Overlap Data Transfers in CUDA C/C++," NVIDIA Developer Blog, available at: https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/, (19th Aug. 2019)

[65] Fang J., Varbanescu A. L., Sips H., "A Comprehensive Performance Comparison of CUDA and OpenCL," *2011 International Conference on Parallel Processing*, pp. 216-225, Taipei City 2011.

[66] Zhu J., Liu Z., Wang D., "Fully pipelined DCT/IDCT/Hadamard unified transform architecture for HEVC Codec," *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 677-680, Beijing 2013.

LIST OF FIGURES

Figure 2.1: 1D and 2D discrete cosine transform 6

Figure 2.2: Source and constructed signals x_n and y_n 7

Figure 2.3 Relation between source signal x(n), constructed signal y(n) and their transforms . 8

Figure 2.4: Graphic representation of the 1D DCT transform matrix 10

Figure 2.5: Graphic representation of the 2D DCT transform matrix 11

Figure 2.6: a) Original 4096 × 1714 frame number 17507 in a video sequence, source [13] b) Frame's two blocks of samples of size 16 × 16 and 32 × 32 having the same centre c) 2D DCT transform of each block d) Difference between blocks at same position in frame number 17507 and 17506 e) 2D DCT transform of block difference 14

Figure 2.7: Quantization and dequantization 16

Figure 2.8: Quantized and dequantized transform coefficients with fined-grained quantization (*left*) and coarse-grained quantization (*right*). Nonzero elements are shown with black squares 16

Figure 2.9: Uniform quantizer (*left*) and nonuniform quantizer with a dead zone area (*right*), source [14] 17

Figure 2.10: Illustration of vector quantization in a two-dimensional vector space, source [15] 18

Figure 2.11: The 32 × 32 matrix specifying the 32-point forward transform. Forward transform matrices for the remaining transform sizes specified in the standard 16x16 (solid border), 8x8 (double border) and 4x4 (gray shading) are embedded 22

Figure 2.12: The HEVC forward transform and quantization (*left*) and HEVC inverse transform and dequantization (*right*). D is the HEVC transform matrix, the scaled approximation of the DCT matrix. Scaling factors appear after each transform and quantization step to ensure equal norm and specified dynamic range 23

Figure 3.1: The folded structure of a ($N \times N$)-point 2D integer DCT implementation, source [25] 29

Figure 3.2: The full-parallel structure of a ($N \times N$)-point 2D integer DCT implementation, source [25] 29

Figure 3.3: Conventional architecture for an 1D integer DCT of lengths N = 8, 16 and 32, source [25] 30

Figure 3.4: The butterfly computation structure of an 8-point 1D DCT 31

Figure 3.5: Design methodology 35

Figure 3.6: Multiplexed MCM unit for a) 4-point 1D DCT and b) 16-point 1D DCT 38

Figure 3.7: General block diagram of the integer DCT lanes 39

Figure 3.8: Schematic of a lane 40

Figure 3.9: Serialization of scaled transform coefficients 40

Figure 4.1: General block diagram of the HEVC transform and quantization process 45

Figure 4.2: Time measurement 50

Figure 4.3: Data transfers between the host and the device 53

Figure 4.4: Grid of thread-blocks 55

Figure 4.5: 64:1 residual to thread-block mapping for 4 × 4 TBs 57

Figure 4.6: 64:1 residual to thread-block mapping with vectorized memory access for 4 × 4 TBs 58

Figure 4.7: Padded shared memory allocated for the short4 data type with the access pattern for 4 × 4 (a) and 32 × 32 (b) transform blocks 61

Figure 4.8: Shared memory access pattern with padding 64

Figure 4.9: Comparison of frame processing times for different TQ implementations for 32 × 32 transform blocks 69

Figure 4.10: Distribution of data structured in shared memory using float2 data type and processed by the top left thread in a thread-block for 32 × 32 transform block 74

Figure 4.11: Comparison of frame processing times for different TQ implementations for 32 × 32 transform blocks after optimizations 84

Figure 4.12: Execution timelines for two versions of the TQ application, implemented on the GPU. In version a) all operations related to one segment are issued and then the next segment is processed in the same way and in version b) host-to device transfers for all segments are issued first, followed by all kernels and all device-to-host transfers. 86

Figure 4.13: Impact of the number of CUDA streams on performance 87

Figure 4.14: Comparison of frame processing times for different TQ implementations on the Workstation environment for 32 × 32 transform blocks 90

Figure 4.15: Overlapping TQ kernel execution and data transfers by using two CUDA streams 90

LIST OF TABLES

Table 3.1: Number of operations for two transform algorithms 28

Table 3.2: Setup of the development environment 34

Table 3.3: Implementation results for the two boundary combinations of strategies 36

Table 3.4: Performance results of the proposed integer DCT architecture with output serialization 41

Table 3.5: Performance results of the proposed integer DCT architecture with parallel output 41

Table 3.6: Comparison with related works 43

Table 4.1: Evaluation environments..... 47

Table 4.2: GPU comparison in the two environments 48

Table 4.3: Comparison of the CPU, AVX2 and GPU implementation of the HEVC transform for 32 × 32 transform blocks and effective memory bandwidth for the GPU kernel 63

Table 4.4: Comparison of the CPU, AVX2 and GPU implementation of the HEVC transform for 32 × 32 transform blocks and effective memory bandwidth for the GPU kernel 65

Table 4.5: Comparison of the CPU, AVX2 and GPU implementation of the HEVC transform for 32 × 32 transform blocks. The duration of data transfers is included in measurements for the GPU implementation. 66

Table 4.6: Kernel execution time and effective memory bandwidth in the GPU implementation of the HEVC transform and quantization kernels for 32 × 32 transform blocks 67

Table 4.7: Processing times and effective bandwidth in the GPU implementation for 32×32 transform blocks 68

Table 4.8: Comparison of the CPU, GPU and CUBLAS implementation of a 2D integer transform with scaling for 32 × 32 transform blocks 71

Table 4.9: Comparison of the CPU, GPU and CUBLAS implementation of a 2D integer transform with scaling for 32 × 32 transform blocks 72

Table 4.10: Comparison of the CPU, vectorized GPU and CUBLAS implementation of transform and quantization for 32 × 32 transform blocks 73

Table 4.11: Comparison of the CPU, AVX2, GPU and CUBLAS implementations of transform and quantization for 32 × 32 transform blocks 76

Table 4.12: Comparison of the CPU, GPU and CUBLAS implementations of transform and quantization for 4 × 4 transform blocks 77

Table 4.13: Comparison of the CPU, GPU and CUBLAS implementations of transform and quantization for 4 × 4 transform blocks 79

Table 4.14: Processing times for different block mappings for 4 × 4 transform blocks 80

Table 4.15: Comparison of the CPU and GPU implementations of transform and quantization for 4 × 4 and 32 × 32 transform blocks based on page-locked data transfers 82

Table 4.16: Execution times and shared memory efficiency with or without the shared memory array padding 83

Table 4.17: Processing time comparison of the CPU and GPU implementations of transform and quantization for three transform block distributions on the Workstation environment 88

Table 4.18: Performance evaluation of the GPU implementation of transform and quantization for three transform block distributions on the Workstation environment 89

Table 4.19: Comparison with a related implementation 91

BIOGRAPHY

Mate Čobrnić was born in 1982 in Makarska, Croatia. He graduated in 2006 at the Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia. After graduation, he started working as an embedded software developer in "MLC Electronic", Zagreb. From 2007 to 2011 he worked in Siemens d.d., Zagreb as an embedded software developer. From 2011 to 2014 he was delegated to Siemens Ltd., China where he worked as a project manager. He returned to Siemens d.d. in 2014 and worked there to 2017 as an integration test manager. During his work in the process industry, he received several certifications related to software testing and process automation protocols. In 2008 he started his postgraduate studies under the mentorship of prof.dr.sc. Mario Kovač. His area of expertise includes high-performance computing and algorithm optimization for heterogeneous system architectures with special focus set on exploring multimedia architectures and video coding. He is a member of the HiPEAC organization. He is the author and co-author of several scientific papers published in international conferences and journals.

PUBLISHED PAPERS

1. Čobrnić M., Duspara A., Dragić L., Piljić I., Kovač M., "Performance Engineering for HEVC Transform and Quantization Kernel on GPUs," in Automatika Journal for Control, Measurement, Electronics, Computing and Communications, vol. 61, no. 3, pp. 325-333, 2020.
2. Čobrnić M., Duspara A., Dragić L., Piljić I., Mlinarić H., Kovač M., "An Area Efficient and Reusable HEVC 1D-DCT Hardware Accelerator," in 13th International Conference on Parallel Processing and Applied Mathematics (PPAM), pp. 199-208, Bialystok 2019.
3. Piljić I., Dragić L., Duspara A., Čobrnić M., Mlinarić H., Kovač M., "Bolt65 – performance-optimized HEVC HW/SW suite for Just-in-Time video processing," 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 966-970, Opatija 2019.

ŽIVOTOPIS

Mate Čobrnić rođen je 1982. godine u Makarskoj. Diplomirao je 2006. godine na Fakultetu elektrotehnike i računarstva. Nakon diplome počeo je raditi kao programer ugrađenih računalnih sustava u tvrtki MLC Electronic d.o.o. Od 2007. do 2011. radio je u tvrtki Siemens d.d. kao programer ugrađenih računalnih sustava. Od 2011. do 2014. delegiran je u Siemens Ltd., Kina, gdje je radio kao voditelj projekata. 2014. vratio se u Siemens d.d. i tamo radio do 2017. kao voditelj integracijskih testova na projektu. Tijekom rada u procesnoj industriji, primio je nekoliko certifikata vezanih za testiranje softvera i protokole za automatizaciju procesa. 2008. godine započeo je poslijediplomski studij pod mentorstvom prof.dr.sc. Maria Kovača. Područja znanstvenog interesa su mu računarstvo visokih performanci i optimizacija algoritama za arhitekture heterogenih računalnih sustava s posebnim fokusom na istraživanje multimedijskih arhitektura i videokodiranje. Član je organizacije HiPEAC. Autor je ili koautor više znanstvenih radova objavljenih u međunarodnim konferencijama i časopisima.