VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2020

Bc. Martin Buchta

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

KÓDOVÁNÍ 4K VIDEA V REÁLNÉM ČASE S TECHNOLOGIÍ NVENC
4K REAL-TIME VIDEO ENCODING USING NVENC TECHNOLOGY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR
Bc. Martin Buchta

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. Petr Kříž

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**

Ústav telekomunikací

Student: Bc. Martin Buchta ***ID:*** 165297

Ročník: 2 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Kódování 4K videa v reálném čase s technologií NVENC

POKYNY PRO VYPRACOVÁNÍ:

Vytvořte aplikaci, která bude s využitím NVENC a Video Codec SDK v reálném čase komprimovat data ze 4K kamer(y). Co nejlépe optimalizujte a proveďte analýzu takového systému - porovnejte implementace kodéru s použitím různých API jako je CUDA, DirectX nebo OpenGL, stanovte limity enkodéru pro Vaši konkrétní hardwarovou konfiguraci (max. počet kódovaných paralelních streamů z kamer apod.), sledujte a diskutujte vliv nastavovaných parametrů enkodéru na výslednou kompresi apod.
Doporučeným vývojovým nástrojem je MS Visual C++, Video Codec SDK.

DOPORUČENÁ LITERATURA:

[1] JOHNSON, B. Professional visual studio 2013: selected readings. 1st edition. Washington: Microsoft Press. Penguin education. ISBN 11-188-3204-3.

[2] BING, B. Next-Generation Video Coding and Streaming. October 2015. 320 pages. ISBN 9781119133339.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 11.8.2020

Vedoucí práce: Ing. Petr Kříž

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

Diplomová práce se zabývá enkódováním 4K videa v reálném čase s použitím technologie NVENC. Obsahuje základní popis nejpoužívanějších video kodeků H.264 a HEVC. Následně byly vysvětleny principy grafických karet a jejich programovatelných jednotek. Součástí práce je analýza řešení volně dostupného Video Codec SDK. Hlavním zaměřením práce je implementace aplikace, která zvládne enkódovat video z více 4K kamer v reálném čase. Dále jsou pro výslednou aplikaci provedeny výkonnostní a kvalitativní testy. Jejich výsledky jsou analyzovány a diskutovány.

KLÍČOVÁ SLOVA

Kódování videa, komprese videa, enkodér, NVENC, H.264, HEVC, CUDA, DirectX

ABSTRACT

Diploma thesis is focused on real-time 4K video encoding using NVENC technology. First chapter describes the most used video codecs H.264 and HEVC. There is an explanation of the principle of graphic cards and their programmable units. Analysis of the solution of open source Video Codec SDK is also part of the thesis. The main focus of the thesis is an implementation of an application which can handle 4K video encoding from multiple cameras in real time. Performance and qualitative tests were performed for application. Results of these tests were analyzed and discussed.

KEYWORDS

Video encoding, video compression, encoder, NVENC, H.264, HEVC, CUDA, DirectX

BUCHTA, Martin. *Kódování 4K videa v reálném čase s technologií NVENC*. Brno, Rok, 86 s. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: Ing. Petr Kříž

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Kódování 4K videa v reálném čase s technologií NVENC“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Petru Kříži za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Brno ....

podpis autora

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic

http://www.six.feec.vutbr.cz

PODĚKOVÁNÍ

Výzkum popsaný v této diplomové práci byl realizován v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

Brno ....

podpis autora

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

OP Výzkum a vývoj
pro inovace

Obsah

Seznam obrázků

Seznam tabulek

Úvod

S vývojem nových technologií se zvyšuje i poptávka po kódování videí s nejvyšší možnou kompresí a zároveň nejmenší ztrátou kvality. Zároveň jsou kladeny požadavky na enkódování a dekódování videosekvencí s vysokým rozlišením, které nelze přenášet jinak než v komprimovaném stavu. Přitom je potřeba, aby kódování a dekódování příliš nezatěžovalo výpočetní výkon, a proto vznikl čip NVENC.

Tato práce je zaměřena na technologii čipu NVENC a kódování do standardů H.264 a H.265. V teoretické části jsou popsány principy enkódování v obou standardech a uvedeny rozdíly mezi nimi. Dále jsou rozebrány grafické karty a právě čip NVENC a Video Codec SDK. Cílem práce je analyzovat již implementované Video Codec SDK, které využívá čipu NVENC, a naimplementovat vlastní jednoduchý enkodér pro kódování videosekvencí ve vysokém rozlišení v reálném čase.

Diplomová práce byla rozčleněna do pěti kapitol. V první kapitole je věnována pozornost kódování videa. Jsou zde rozebrány video kodeky a barevné prostory, které jsou důležitou složkou v kódování videosekvencí. Následně je vysvětleno, z jakého důvodu vznikly standardy vztahující se ke kódování videí a k čemu slouží. Stěžejní částí je podrobná analýza standardů H.264 a H.265, která popisuje jejich princip a nejdůležitější metody.

V druhé kapitole je rozebráno využití grafických karet. Je zde uvedena historie a architektura grafických karet. Tato kapitola se také zaměřuje na části grafické karty, které jsou programovatelné. Jedna z nich je CUDA, která se využívá v souvislosti s Video Codec SDK pro enkódování videosekvencí. V další části jsou popsány principy fungování Video Codec SDK od společnosti Nvidia. Dále je vysvětlena role čipu NVENC a jeho předchůdce NVCUVID.

Třetí kapitola se věnuje analýze dostupného Video Codec SDK. Je zde popsán obecný princip, jak probíhá enkódování v již naimplementovaném a nepozměněném projektu. Dále jsou rozebrány zvlášť projekty využívající k enkódování CUDA, DirectX a OpenGL. Jsou vyzdvihnuty a detailně popsány důležité funkce a metody enkodéru. Na konci kapitoly je návrh implementace jednoduchého enkodéru.

Čtvrtá kapitola popisuje implementaci jednoduchého enkodéru s použitím Video Codec SDK. Jsou zde uvedeny nástroje, které byly k implementaci použity. Dále je kapitola rozčleněna do několika sekcí popisujících přístup k implementaci a úryvky z použitých kódů. V sekcích jsou rozebrány principy rozdělování procesů do více vláken, načítání snímků z kamery a poté proces enkódování videosekvencí. Nad rámec zadání diplomové práce bylo vytvořeno grafické rozhraní, ze kterého lze jednotlivé projekty spouštět a pozorovat vliv na výkon systému.

Poslední kapitola se věnuje testování, které bylo rozděleno na výkonnostní a kvalitativní. U výkonnostních testů se měřily hodnoty výkonu systému v závislosti na pa-

rametrech enkódování. Byla zde měřena doba vyhodnocení části kódu při použití různých projektů a také průměrně zvládané rámce při připojení více kamer. U kvalitativních testů se měřil koeficient komprimačních ztrát. Tyto testy byly prováděny na více hardwarových konfiguracích a v kapitole jsou zpracovány výsledky testů.

1 Kódování videa

Kódování videa je důležité hlavně kvůli kompresi dat s cílem nejmenší ztráty kvality. V této kapitole jsou obecně rozebrány video kodeky, barevné prostory a role standardů. Dále jsou detailně popsány dva standardy, které jsou v této diplomové práci používány.

1.1 Video kodek

Video kodek je zařízení, nebo software, který se používá pro kompresi nebo dekompresi digitálního videa. Slovo kodek je složeno ze dvou slov: kodér a dekodér. Kodér zprostředkovává proces komprimace a dekodér rekonstrukci původní videosekvence. Existují kodeky obsahující obě komponenty, ale vyskytují se i kodeky umožňující pouze jednu výše zmíněnou funkci [1].

1.2 Barevné prostory

Aplikace s digitálním videem jsou v dnešní době závislé na zobrazování barev, a proto je důležité určit si, jakým způsobem tuto informaci bude aplikace zpracovávat. Monochromatický obrázek potřebuje pouze jeden byte na zobrazení jasu, barevný snímek potřebuje na jeden pixel minimálně tři. Metoda, jak reprezentovat informaci o barvě nebo jasu, udává zvolený barevný prostor. Existuje několik barevných prostorů, avšak k této práci byly použity prostory RGB a YCrCb, které jsou rozebrány níže.

1.2.1 YCrCb

Vizuální systém člověka lépe vnímá barvu než jas. V RGB jsou všechny tři složky uchovávány ve stejném rozlišení a právě prostor YCrCb nabízí efektivnější reprezentaci barvy oddělením jasové složky od barevné. Tento prostor se skládá ze tří složek. Y je jasová složka, která může být z RGB prostoru vypočtena váženým průměrem R, G a B podle vzorce 1.1

$$Y = k_r R + k_r G + k_r B, \tag{1.1}$$

kde k jsou váhové faktory.

Informace o barvě je reprezentována jako barevný rozdíl, kde každý barevný posun je výsledkem rozdílu složky jasové a příslušné barvy 1.2.

$$
\begin{aligned}
Cr &= R - Y \\
Cb &= B - Y \\
Cg &= G - Y
\end{aligned}
\tag{1.2}
$$

Kompletní reprezentace barvy je potom udávána jednou jasovou složkou Y a třemi barevnými posuny Cr, Cb a Cg, které vyjadřují rozdíl mezi jasem a intenzitou odpovídající barevné složky. Tudíž k úplnému vyjádření je potřeba čtyř složek, avšak to je víc než u RGB. Součet Cr, Cb, a Cg je však konstanta, takže stačí znát pouze dva barevné posuny a poslední se dopočítá. Výrazná výhoda oproti RGB prostoru je ta, že Cr a Cb mohou být vyjádřeny v menším rozlišení než Y, protože jak bylo zmíněno výše, tak visuální systém člověka je víc citlivý na jas než na barvu. Tímto způsobem se redukuje pamět využitá pro uchování informace o barevnosti bez ztráty kvality [2].

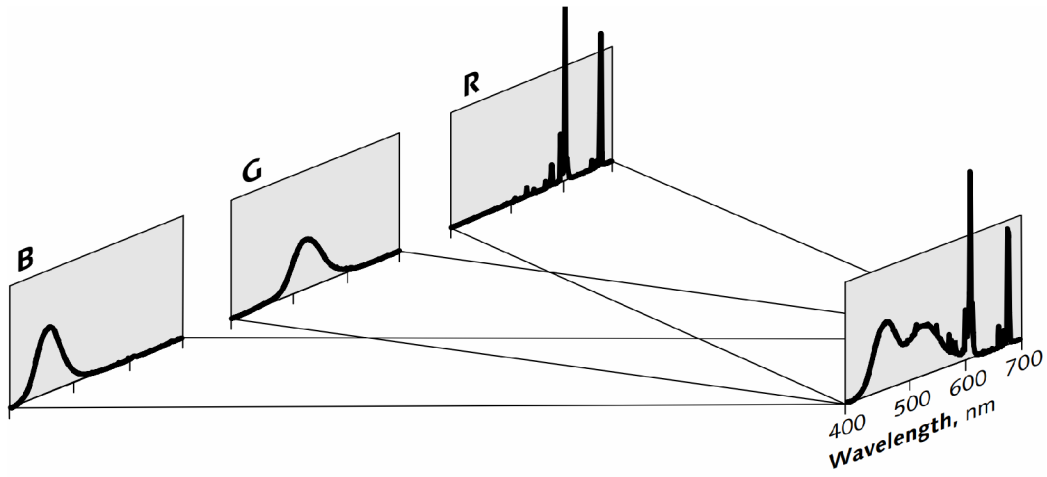
1.2.2 RGB

V RGB barevném prostoru se reprezentuje barevnost pomocí tří hodnot. Každá z hodnot je přiřazená barvám červené, zelené a modré. Výsledná barva vznikne procesem aditivní reprodukce. Je to nejjednodušší způsob pro vytvoření jakékoliv barvy. Zachycení RGB snímku zahrnuje filtrování červené, zelené a modré komponenty, kde pro každou platí jiné parametry. Pixel obrázku má uvedeny tři hodnoty ke každé barevné komponentě podle intenzity barvy v pixelu. Výsledná barva vznikne spojením všech tří složek dohromady. Příklad aditivní reprodukce používané u videa je zobrazen na obrázku 1.1 [3].

1.3 Role standardů

V historii bylo vynalezeno a vydáno mnoho různých technik určených ke kódování videa. Každá z nich poskytuje různé inovativní přístupy k tomuto problému, ale ne všechny mohou být standardizovány. Ty, které byly standardizovány, mají spoustu výhod:

- Zajišťují spolupráci mezi enkodéry a dekodéry.
- Definují normy pro vytváření platforem, které videa využívají a zajišťují konzistenci s videokodeky, audiokodeky a transportními protokoly.
- Jsou velice dobře zdokumentovány a algoritmy, které obsahují, mají jasně stanovená licenční pravidla pro použití.



Obr. 1.1: Na obrázku je popsán proces aditivní reprodukce používané ve videu. Každá složka R, G a B má nezávislou a přímou cestu do výsledného snímku. Spektrum výsledného obrázku je součet spekter každé z komponent [3].

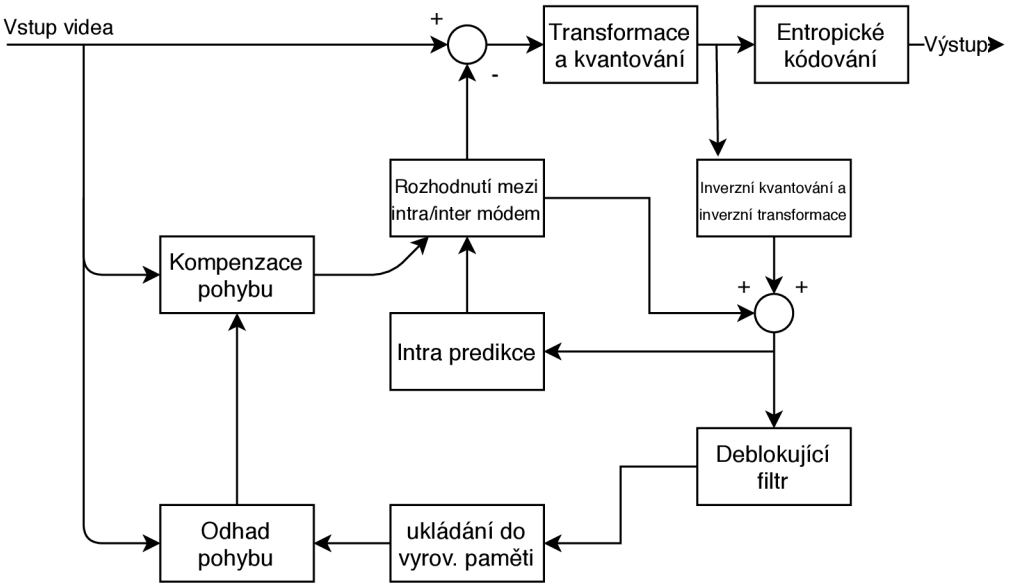
1.4 Kódovací standard H.264

H.264 je jedním z novějších standardů používaný ke kompresi digitálního videa. Je znám i pod názvem MPEG-4 Part 10: Advanced Video Coding. Začaly jej vyvíjet oraganizace ITU-T Video Coding Experts Group (VCEG) a ISO/IEC Moving Picture Experts Group (MPEG). Svou spolupráci tyto organizace zahájily v roce 2000 a spojily se v jednu skupinu nazývanou Joint Collaborative Team on Video Coding (JCT-VC).

Standard byl navržen tak, aby bylo možné ho využívat v širokém spektru aplikací včetně IP a bezdrátových sítí a digitálního kina. Byl vytvořen v roce 2003 na základě předchozího standardu MPEG-2. Oproti svému předchůdci má mnoho výhod. Tou největší je lepší kvalita komprese při stejném datovém toku. Mezi další pozitiva patří zejména flexibilnost při kompresi, různé rozměry makrobloků, přenášení a ukládání videa. Schéma enkodéru lze vidět na obrázku 1.2 [1].

1.4.1 Proces kódování a dekódování

H.264 enkodér zahrnuje proces predikce, transformování jednotlivých bloků a samotné enkódování k vytvoření komprimovaného toku bitů. Dekodér se poté stará o procesy dekódování, inverzní transformace a rekonstrukce k získání původní videosekvence. Před těmito procesy jsou nejdřív data rozdělena na takzvané makrobloky, které typicky odpovídají rozměru 16 x 16 zobrazených pixelů. Není to však podmínkou, protože narozdíl od standardu MPEG-2 může H.264 enkodér dělit data



Obr. 1.2: Na obrázku je schéma enkodéru H.264. [4]

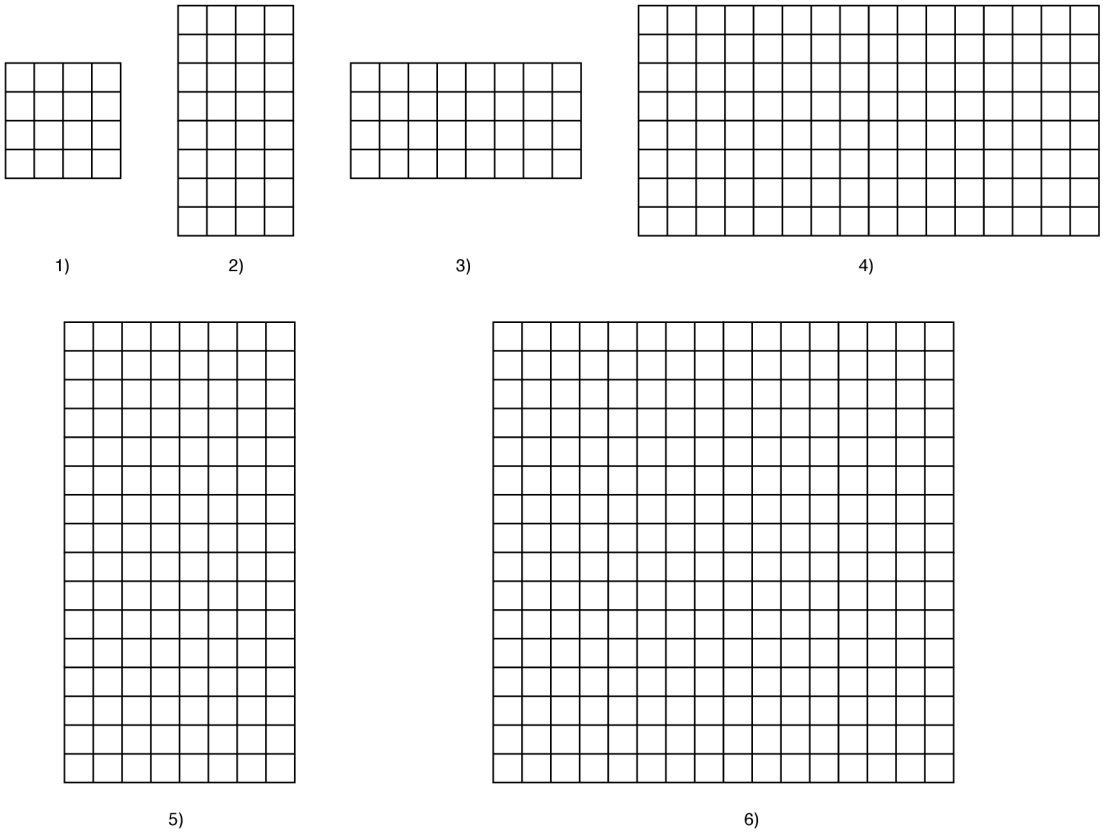
i do jiných rozměrů. Rozdělení je zobrazeno na obrázku 1.3. Tato vlastnost umožňuje větší flexibilitu a také větší efektivitu při kódování videa [5].

Makroblok

Makroblok je základní jednotkou rámce a při videokompresi, kde se používá diskrétní kosinová transformace, je to neméně důležitý pojem. Makroblok většinou sestává z 16 x 16 pixelů. V YCbCr variantě barevného modelu, kde Y představuje jasovou složku a Cb a Cr jsou barevné posuny, je makroblok složen ze čtyř Y bloků, jednoho Cr a jednoho Cb bloku [6]. Může být však reprezentován i jinou charakteristikou:

- 4:2:0 – typický makroblok složený ze 4 bloků Y, 1 bloku Cb a 1 bloku Cr
- 4:2:2 – makroblok složený ze 4 bloků Y, 2 Cb bloků a 2 Cr bloků
- 4:4:4 – video v celé šířce pásma, obsahuje tolik informací, jako v RGB variantě barevného modelu. Každý makroblok obsahuje 4 Y bloky, 4 Cb bloky a 4 Cr bloky.

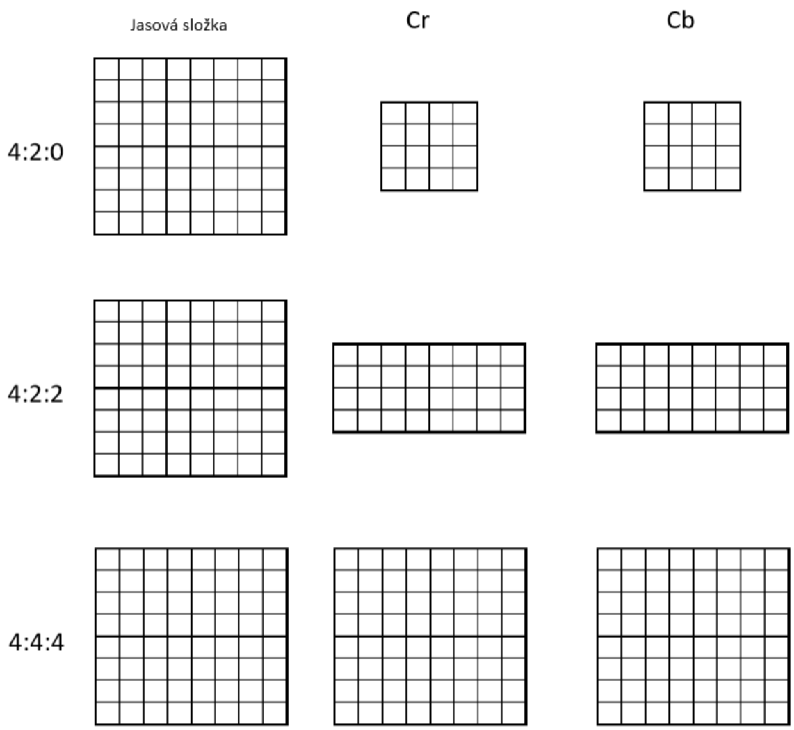
Tímto způsobem charakteristiky (přes jas a barevný posun) je možné v 4:2:0 reprezentaci modelu snížit počet dat na polovinu oproti reprezentaci v plné šířce pásma 4:4:4 nebo RGB modelu. Ukázka všech zmíněných reprezentací makrobloků je znázorněna na obrázku 1.4 [1].



Obr. 1.3: H.264 umožňuje pracovat s různými velikostmi makrobloků. 1) 4 × 4, 2) 4 × 8, 3) 8 × 4, 4) 16 × 8, 5) 8 × 16, 6) 16 × 16, kde každý čtverec představuje jeden zobrazovaný pixel. [8]

1.4.2 Predikce

Aktuální snímek, který se zpracovává je složen z makrobloků. Cílem predikce je snížit redundanci předvídáním dat, která budou aktuálnímu snímku náležet. Toho je možné dosáhnout buď na úrovni aktuálního snímku, nebo na základě již zpracovaných dat v předchozích snímcích. První metoda se nazývá intrapredikce a druhá interpredikce. Enkodér provede predikci současného snímku a uschová si data, která tuto predikci řídí (pohybové vektory, mód predikce, atd.). Po kompresi tato data musí předat enkodéru, aby mohl být snímek zrekonstruován. Metody poskytované standardem H.264 jsou více flexibilní než u předchozích standardů. Je možné provádět predikci na makroblocích o velikosti 16 x 16 nebo 4 x 4 pixelů. Interpredikce používá dynamický rozměr bloků v rozmezí od 4 x 4 po 16 x 16 pixelů [7].



Obr. 1.4: Na obrázku jsou znázorněny 3 reprezentace barevného modelu YCbCr.

Intrapredikce

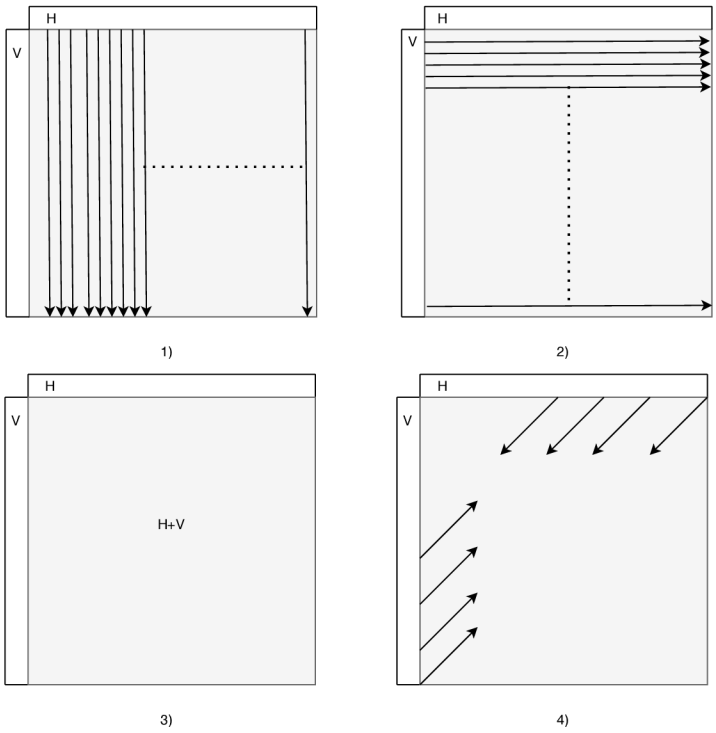
Intrapredikce je metoda, která používá už zpracovaná data v aktuálním snímku a právě zpracovávaný makroblok ve snímku na základě podobnosti podrobí rekonstrukci. Používají se 2 metody intrapredikce.

První z nich je intrapredikce na malém makrobloku, kde výsledek bude přesnější a nesrovnalosti budou menší. Druhou možnou metodou je intrapredikce na makrobloku o větší velikosti např. 16 x 16. V tomto případě bude predikce méně efektivní a výsledný obraz bude mít nedokonalosti. Nicméně volba intrapredikce pro každý 4 x 4 makroblok musí být signalizována dekodéru a tím se zvyšuje velikost výsledného souboru [6]. Existují 4 módy intrapredikce, které lze přiřadit:

- Mód 0 – vertikální
- Mód 1 – horizontální
- Mód 2 – DC (shora a zleva)
- Mód 3 – plane (podle funkce spojené s předchozím módem)

Dohromady je možné zvolit jedno z devíti nastavení pro 16 x 16 makroblok jasové složky a to pro každý 4 x 4 makroblok jasové složky. Jeden mód je přiřazen ke kaž-

dému 4 x 4 makrobloku barevné složky. Tyto módy jsou znázorněny na obrázku 1.5.

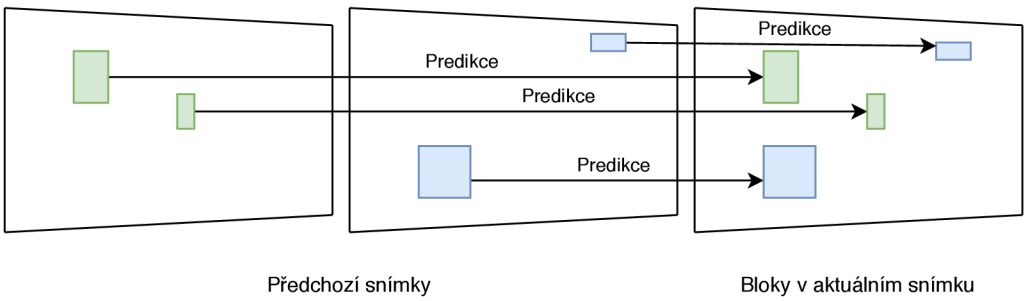


Obr. 1.5: Na obrázku jsou představeny 4 módy používané u intrapredikce: 1) Mód 0 – vertikální, 2) Mód 1 – horizontální, 3) Mód 2 – DC (shora a zleva), 4) Mód 3 – plane (podle funkce spojené s přechozím módem) [8].

Interpredikce

Model interpredikce pracuje s aktuálním snímkem na základě jednoho nebo více předchozích snímků, které už byly zpracovány viz 1.6. Proces predikce se skládá z několika částí. Prvně se musí vybrat oblast, se kterou se bude pracovat. Poté se vygeneruje predikční blok a nakonec pro tento blok bude vypočten rozdíl od originální oblasti ve snímku. U tohoto modelu se může pracovat s makrobloky o velikostech od 4 x 4 až 16 x 16 pixelů. Jak už bylo zmíněno, pracuje se s předchozími snímky, které jsou uloženy v *Decoded Picture Buffer*.

Posun mezi pozicí oblasti v aktuálním snímku a oblastí v předchozím snímku je nazýván pohybový vektor. Může směřovat do pozice začátku referenční oblasti, nebo její poloviny či čtvrtiny. Pozice poloviny a čtrvrtiny jsou určeny interpolací oblasti předchozího snímku. Každý pohybový vektor je vypočten z pohybových vektorů okolních bloků. Predikční blok je možné vypočítat z jedné oblasti referenčního



Obr. 1.6: Na obrázku je ukázka interpredikce, která používá předchozí snímky k vytváření aktuálního snímku.

snímku, potom je označován jako makroblok typu P nebo B. Pokud se jedná o dvě oblasti referenčního obrázku, tak je typu B a má přiřazené pořadí. Existuje i varianta vážené predikce, kde se počítá s dočasnou vzdáleností mezi aktuálním a referenčním snímkem. Pokud je makroblok typu B, potom může být vypočten bez nutnosti přepočtu pohybových vektorů. Ty se převezmou z předchozích snímků [6].

1.4.3 Rekonstrukční filtr

Filtr pro potlačení blokové struktury je použit na každý kódovaný makroblok. Jeho cílem je vyhladit hrany po rozdělení rámce do makrobloků. Filtr je aplikován mezi inverzní transformací a rekonstrukcí obrazu. Snímek po filtraci je použit pro kompenzaci pohybu. Tímto pořadím operací se zvyšuje i velikost komprese, protože vyhlazený rámec je více důvěryhodný než nevyhlazený s hranami. Filtrování se provádí na hrany bloků 4 x 4 v makrobloku. Nejdříve se provede proces na vertikálních hranách jasového bloku zleva doprava, poté na horizontálních hranách shora dolů. Dále se aplikuje filtr na vertikální hrany barevné složky a nakonec na horizontální hrany barevné složky [1].

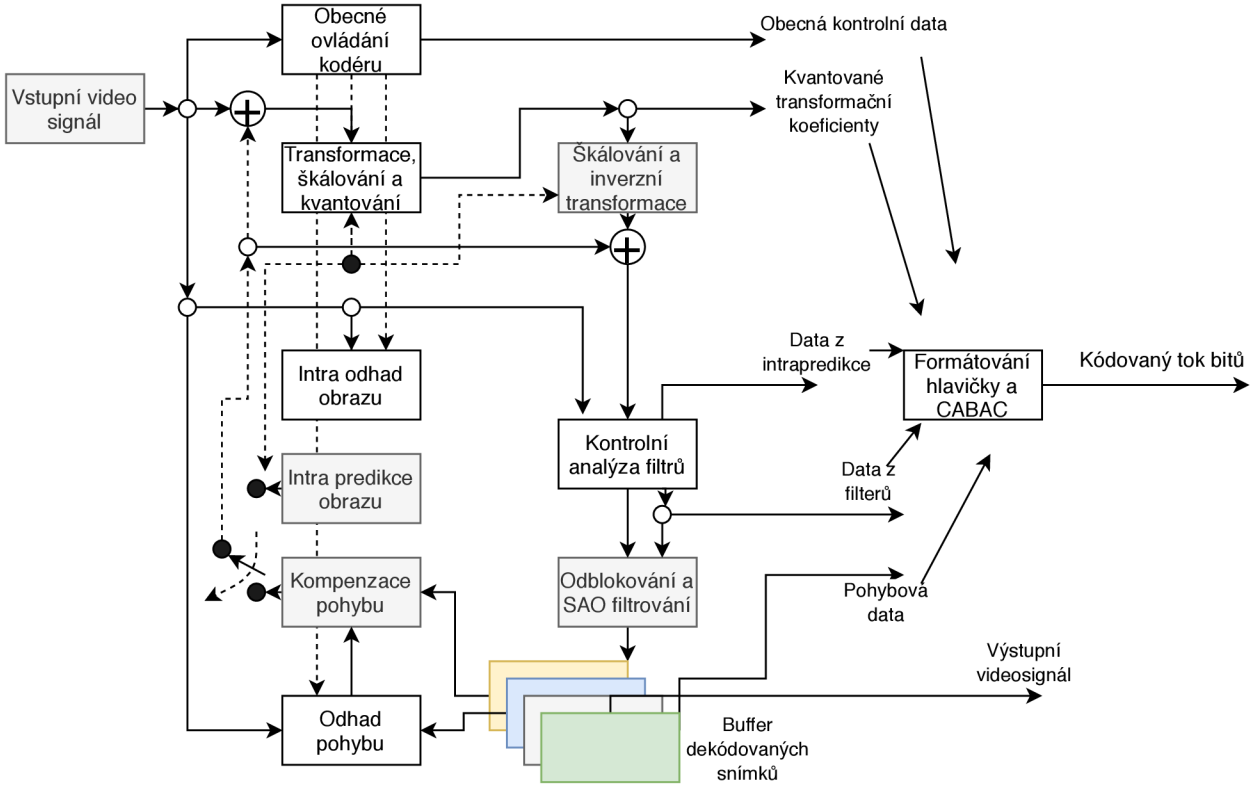
1.5 Kódovací standard H.265

Tento kodek vznikl na základě standardů ITU-T H.264 a jeho předchůdců. Běžně se můžeme setkat také s názvem HEVC [1]. Jako většina dřívějších stadnardů byl i tento vyvinut kvůli potřebě efektivnější komprese videosekvencí pro streamování, komunikaci, videokonference, ukládání médií a televizní vysílání. Standard je navrhnutý tak, aby ho bylo možné použít na širokém množství rozhraní a je flexibilní natolik,

[1] High Efficiency Video Coding

aby bylo možné s videem nakládat jako s daty, která lze přenášet a ukládat na již existující a zároveň na budoucí zařízení.

Kodek H.265 byl vyvinut oraganizacemi ITU-T Video Coding Experts Group (VCEG) a ISO/IEC Moving Picture Experts Group (MPEG). Jeho první verze byla vydána v dubnu roku 2013 výše zmíněnými organizacemi, které se spojily v Joint Collaborative Team on Video Coding (JCT-VC). Od té doby byly vydány ještě čtyři další verze.

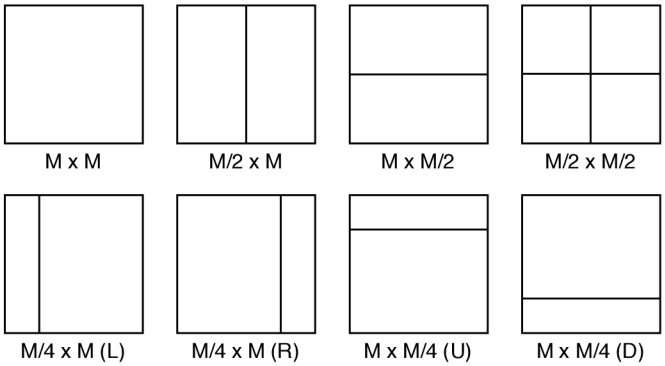


Obr. 1.7: Na obrázku je schéma kodéru standardu H.265. [9]

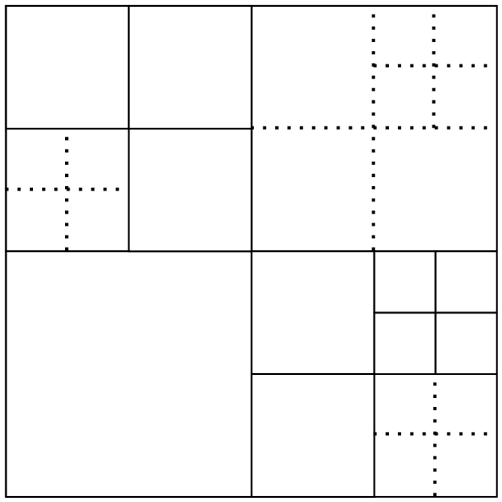
1.5.1 Stromová struktura

HEVC přinesl velkou změnu oproti svým předchůdcům, když začal používat jinou základní stromovou strukturu bloků. Není založena na makroblocích jako H.264, ale na jednotkách CTU (Coding Tree Unit). CTU je složena z jednoho jasového bloku CTB (Coding Tree Block) a dvou odpovídajících barevných CTB. CTB může nabývat velikosti 16 x 16, 32 x 32 a 64 x 64 pixelů. CTB mohou obsahovat jeden nebo více CB (Coding Blocks), které jsou ve tvaru čtverce a jejich minimální velikost je 8 x 8 pro jasovou složku a 4 x 4 pro chrominační složku. Jeden jasový CB a dva odpovídající barevné CB s korespondující syntaxí tvoří CU (Coding Unit). CU obsahují

PU (Prediction Unit) a TU (Transform Unit). PU se skládá, podobně jako u CU, z jasových PB (Prediction Block) a barevných PB viz 1.8. PU je jednotka syntaxe zodpovědná za ukládání informací o intrapredikci a interpredikci. Každá z predikcí obsahuje vlastní rozdílné PU. CU se může skládat až ze čtyř PU. TU je jednotka syntaxe, která má na starosti ukládání transformačních dat. TU jsou opět složeny z TB (Transform Block), které mohou nabývat velikosti 4 x 4, 8 x 8, 16 x 16 nebo 32 x 32 pixelů. Rozdělení CTB do CB lze vidět na obrázku 1.9 [10].



Obr. 1.8: Na obrázku jsou zobrazeny možnosti, jak rozdělit kódovací blok do více predikčních bloků. [10]



Obr. 1.9: Na obrázku je znázorněno rozdělení stromového kódovacího bloku do kódovacích bloků a transformačních bloků. Plné čáry jsou hranice CB a přerušované značí TB. [9]

1.5.2 Predikce

Intrapredikce a interpredikce vychází ze stejného základu jako obsahoval H.264 s rozdílem, že je u intrapredikce rozšířen počet úhlů, které je možné použít.

Intrapredikce

Standard HEVC podporuje 3 základní módy intrapredikce:

- DC (Direct Component)
- Planární
- Úhlový

Úhlový mód intrapredikce má narozdíl od H.264, kde byla možnost pouze osmi směrů, použít až třicet tři směrů. Všechny možné směry jsou znázorněny na obrázku 1.10. Pokud se provádí vertikální dominantní predikce, body nalevo od kódovaného segmentu se nazývají hlavní pole a ty, které jsou nad kódovaným segmentem boční pole. Při horizontální dominantní predikci je hlavní pole místo s body, které jsou nad kódovaným segmentem, a boční pole pro body nacházející se nalevo od segmentu. K celkově třiceti třem módům intrapredikce se ještě přidává možnost použití transformačních bloků, které mohou nabývat velikostí 4 x 4, 8 x 8, 16 x 16 nebo 32 x 32 pixelů. Pokud se jedná o intrapredikci DC módu, tak je k predikci stejnosměrné složky vypočtena střední hodnota pixelů ze sousedních bloků (horního a levého vůči aktuálně predikovanému bloku). Planární mód je výpočetně nejnáročnější ze všech uvedených módů, protože používá dvoudimenzionální lineární interpolaci. U úhlového módu intrapredikce se provádí lineární interpolace bodů ve směru, který je určen úhlem. Jeho velikost je určena úhlem, který svírá směr interpolace s vertikální osou [9].

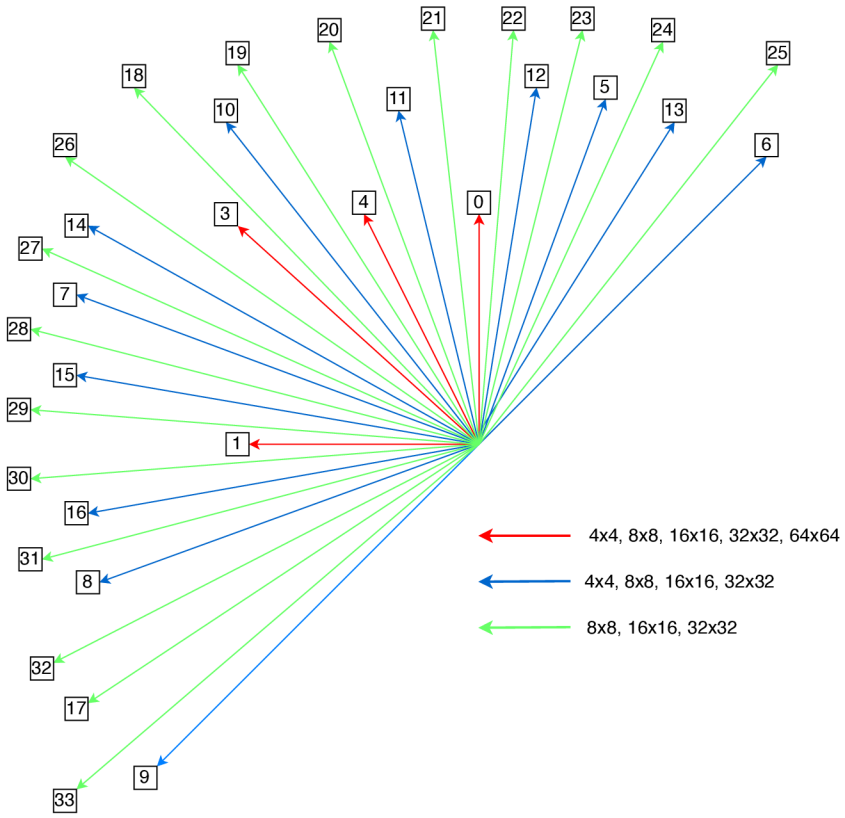
Interpredikce

Pro interpredikci se využívají dva referenční seznamy L0 a L1. Každý z nich má kapacitu šestnáct snímků, kde maximálně osm může být unikátních, takže se některé opakují. To usnadňuje proces predikce ze stejného snímku, ale s různými váhami predikce. Interpredikce probíhá pomocí predikce pohybového vektoru. Je rozdělena na dva módy:

- Merge
- AMVP[2]

Pro každou predikční jednotku se enkodér rozhoduje, jaký z těchto režimů použít a přiřadí mu odpovídající příznak. AMPV používá kódování pohybového vektoru delta a dokáže vyprodukovat jakoukoliv hodnotu pohybového vektoru. HEVC si

[2]Advanced Motion Vector Prediction



Obr. 1.10: Směrové módy intrapredikce. [11]

rozvzorkuje pohybové vektory na 16 x 16 pole, to znamená, že si dekodér alokuje místo pouze pro dva pohybové vektory L0 a L1 pro buffer pohybových vektorů na 16 x 16 pixelů [11].

1.5.3 Entropické kódování

Metodou pro entropické kódování je CABAC[3] na úrovni CTU. Tento princip je velice podobný kódování používaném v H.264, ale byly mu přidána menší vylepšení. Celkově byl zjednodušen inicializační proces a objevuje se asi jen polovina proměnných. Dochází k vyšší datové i přenosové rychlosti a je možné pracovat s konkrétními bloky stromové struktury [11].

1.5.4 Rekonstrukční filtr

HEVC obsahuje možnost použít 2 filtry: *in-loop deblocking* filtr (ILD) a *sample adaptive offset* filtr (SAO). Oba filtry mohou být deaktivovány. ILD filtr je podobný filtru v H.264, zatímco SAO filtr je úplně nový a je aplikován až po ILD filtru.

[3]Context-Adaptive Binary Arithmetic Coding

ILD filtr je nazýván také jako filtr pro potlačení blokové struktury. Jeho úkolem je zjemnit a maskovat hrany, které při enkódování vznikly po rozdělení do bloků na úrovni TU a PU. V H.264 byl tento filtr používán nad 4 x 4 pixely, v HEVC se aplikuje pouze nad 8 x 8 vzorky. Nejdříve jsou potlačeny všechny vertikální hrany, a poté následuje aplikace filtru na všechny horizontální hrany. Filtrace se provádí na veškerých hranách až do minimální velikosti 8 x 8 pixelů.

Po potlačení blokové struktury se aplikuje SAO filtr. Použije se pouze na vzorky, které vyhovují všem požadavkům. Aplikuje se na vrstvě CTB bloků a to pro každý pixel tohoto bloku. SAO filtr je použit na jasovou složku i chrominační složky. Každému CTB bloku přidělí jednu ze čtyř hodnot offsetu, který je v rozmezí od -7 do 7 pro osmibitovou videosekvenci. Enkodér vybere parametry tak, aby se výchozí obraz co nejvíce podobal zdrojovému [11].

1.5.5 Rozdíly mezi H.264 a H.265

S potřebou přesouvat videa na internet a streamovat videa jsou zvyšovány nároky na účinnost komprese, aby zabíraly videosekvence co nejmenší velikosti. Neméně velkým důvodem je i rozšiřování 4K a 8K kamer. Byly provedeny studie, kde HEVC dokázal redukovat objem dat o 52 % lépe na rozlišení 480p a při rozlišení 4K UHD měl o 64 % lepší výsledky než H.264. V závěru má výsledné video na stejné datové velikosti mnohem lepší visuální kvalitu [12].

Velkou změnou je používání CTU[4], které zajišťuje možnost pracovat s jinými velikostmi, než měl makroblok H.264. CTU může utvořit blok o velikosti od 4 x 4 do 64 x 64 pixelů oproti H.264, kde byla maximální velikost 16 x 16 pixelů. Dalším rozdílem je počet úhlových vektorů při intrapredikci, kde HEVC poskytuje až třiceti tři směrů oproti osmi směrům v H.264. V neposlední řadě přišlo HEVC s SAO filtrem, který dopomáhá k lepší rekonstrukci obrazu a v H.264 nebyl implementován.

[4]Coding Tree Units

2 Využití grafických karet

Tato kapitola pojednává o architektuře a využití grafických karet. Osvětlí základní pojmy, jako je GPU [1] a GPGPU [2] z hlediska historie, architektury a funkce.

2.1 GPU

GPU je hardware, jehož hlavní funkcí je rychle provádět grafické výpočty. V dnešní době je nedílnou součástí mnoha zařízení, jako jsou vestavěné systémy, mobilní telefony, počítače a herní zařízení. Jejich architektura je navržena tak, aby bylo GPU schopno zpracovávat velké bloky dat paralelně.

2.1.1 Historie GPU

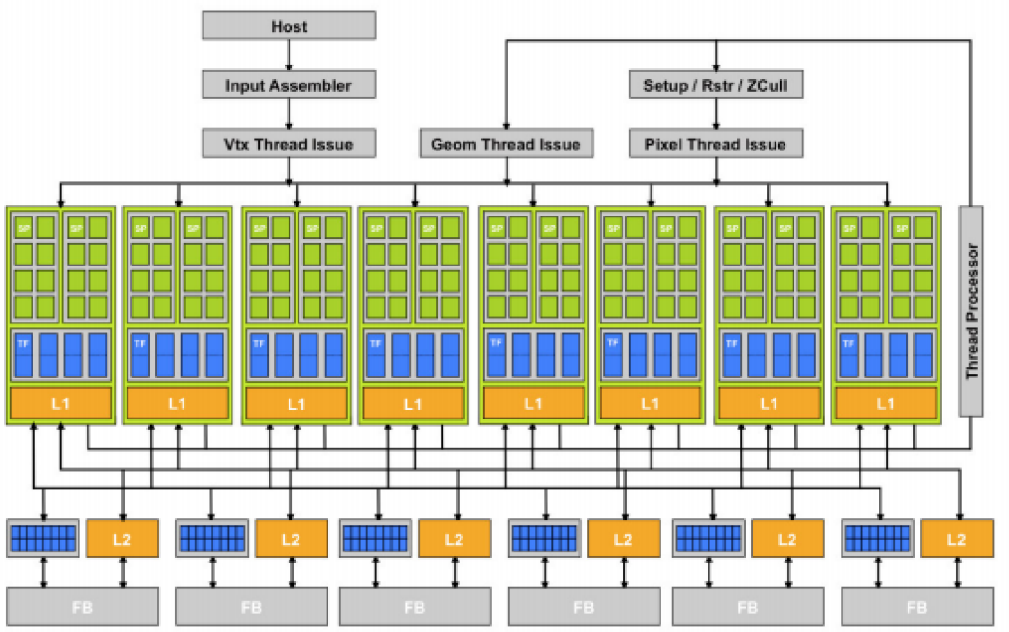
První zmínka o grafických jednotkách byla už v 70. letech minulého století, kde zabezpečovaly paralelní chod grafických a textových aplikací a jejich vykreslování na obrazovku. Postupem času se začaly používat pro vykreslování 2D objektů a v 90. letech byly uzpůsobeny k vykreslování a výpočtům s 3D objekty. K velké popularizaci došlo v roce 1999, kdy společnost Nvidia představila na trh GeForce 256, který je považován za první oficiální GPU. V roce 2001 byla vydána nová grafická karta, která měla plně programovatelný hardware pro transformaci vrcholů a generování fragmentů [13].

2.1.2 Architektura GPU

V dnešní době jsou největšími výrobci grafických adaptérů společnosti Nvidia a ATI. V minulosti právě tyto dvě společnosti vydaly největší počet architektur. Moderní GPU je složen z globálního plánovače, globální paměti, cache textury, multiprocesoru a texturovací jednotky. Architektura je zobrazena na obrázku 2.1. Veškeré výpočty probíhají v procesorech, které jsou sdružovány do multiprocesorů. Tyto výpočtové jednotky jsou optimalizovány pro výpočty SIMD [3], to znamená, že provádí jednu operaci nad mnoha daty. Tento proces lze snadno paralelizovat a data se zpracovávají na co největším počtu jednotek ve stejnou dobu v libovolném pořadí.

Důvodem, proč používat GPU, je prioritně výpočetní výkon. Většina aplikací si pro svůj chod dokáže vystačit s CPU výkonem. Některé aplikace však potřebují pro svůj provoz náročnější výpočty a běžný procesor by nebyl dostatečným řešením.

[1] Graphics Processing Unit
[2] General-Purpose Computation on Graphics Hardware
[3] Single Instruction Multiple Data



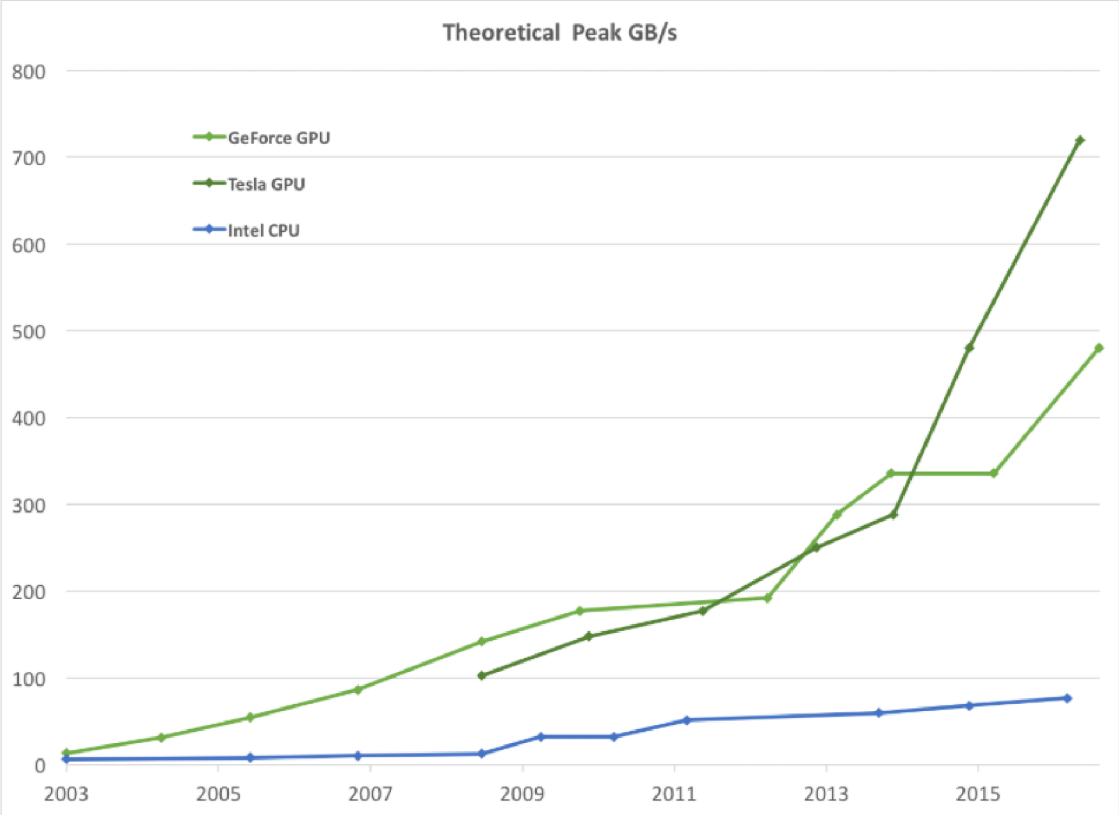
Obr. 2.1: Na obrázku je znázorněna architektura moderního GPU [14].

U těchto aplikací dokáže GPU mnohonásobně snížit výpočetní dobu. Srovnání maximálního toku dat přes paměť RAM[4] mezi CPU a GPU v průběhu času je na obrázku 2.2 [14].

2.2 GPGPU

GPGPU je GPU navrženo jako koprocesor pro speciální účely využití. To tedy znamená, že má programovatelnou jednotku pro paralelizované výpočty. Z historického hlediska začaly vznikat první GPGPU s vydáním GPU Geforce 3 od společnosti Nvidia. Ta přinášela novinku – programovatelný shader. V té době hardwarové komponenty ještě neumožňovaly výpočty složitých algoritmů, proto se používaly hlavně na vertexové a pixelové operace. S příchodem Geforce 8 se stalo programování snazší díky podpoře CUDA[5] a zlepšení shadovacího jazyka, což je programovací jazyk pro programování v shaderech. Mezi ty patří například charekterizace povrchů a objektů. Tento programovací jazyk byl používán nejen pro grafické výpočty. S příchodem Windows Vista přišel DirectX 10, kde se shaderová jádra stanovila jako součást standardů [15].

[4]Random Access Memory
[5]Compute Unified Device Architecture



Obr. 2.2: Na grafu je porovnání množství tekoucích dat přes RAM, počítá se v GB/s. Vyšší hodnoty znamenají vyšší výkon, protože se nemusí čekat na uvolnění paměti. Opět je srovnání několika variant GPU a CPU [16].

2.3 CUDA

CUDA byla představena světu v listopadu roku 2006 společností Nvidia. Je to výpočetní platforma pro obecné účely a programový model, který ovlivňuje výpočetní jádro GPU. Byla navržena tak, aby mohla provádět složité paralelní výpočty efektivněji, než to bylo možné na CPU. Podporuje množství jazyků, mezi které patří C, C++, Fortran nebo DirectCompute. Je podporována operačními systémy Windows, Linux a MacOs v 32bitové i 64bitové verzi. Dříve byly podporovány pouze některé z grafických adaptérů této firmy, avšak později byla rozšířena podpora na všechny grafické karty od společnosti Nvidia [17].

2.3.1 Programátorský model

Vše se odvíjí od programu, který běží na čipu grafického adaptéru nazvaného Kernel. Je to funkce, kterou si programátor deklaruje. Jakmile je zavolána, je provedena pa-

ralelně několika různými vlákny. Vlákna jsou nejmenší a zároveň základní jednotkou pro provádění výpočtů. Jejich režie je takřka nulová a jedno vlákno má velice nízký výkon. Proto se tato vlákna shlukují do větších celků. Pojem warp znamená shluk vláken, která jsou paralelně vykonávaná. Pokud jsou warpy vykonávány v jednom multiprocesoru, potom jsou označovány jako blok vláken. Ten většinou obsahuje 64 až 512 vláken. Tento blok je schopný si mezi svými warpy předávat některá data. Množina bloků vláken, kde běží jeden kernel, je nazýván grid.

Postupně se zjišťuje, která z vláken ještě nejsou přepočítána a ta se posílají k výpočtu na multiprocesor. Pokud warp získá všechny potřebné informace, provede se. Pokud ne, tak čeká na data a mezitím se provádí ty warpy, které už svá potřebná data obdržely. K doručování dat je zapotřebí paměti, která dokáže obsluhovat požadavky paralelně. Paměť musí mít také vysokou propustnost. Díky tomuto principu dekompozice je jasná myšlenka paralelního zpracování výpočtů. Pro jeden kernel je potřeba tisíce vláken. Pro dosažení maximálního výkonu je tedy nutné dosáhnout dobrého poměru zpracovávaných bloků ku počtu multiprocesorů a správné provedení dekompozice [18].

2.4 Nvidia Video Codec SDK

Nvidia Video Codec SDK[6] je obsáhlá sada API[7], která zajišťuje hardwarové zrychlení kódování a dekódování videosekvencí. SDK je kompatibilní s operačními systémy Windows a Linux. Obsahuje dvě rozhraní, NVENCODE a NVDECODE. GPU od Nvidia mají vedle CUDA jáder ještě jednu nebo více dekodérů a enkodérů. Ty zajišťují plně akcelerovaný výkon při kódování a dekódování videa. Tyto akcelerátory poskytují pro enkódování a dekódování vyšší rychlosti, než je zpracovávání videa v reálném čase. Tím jsou vhodné především pro transkódovací aplikace. Výhody používání Video Codec SDK jsou:

- Flexibilita – dynamické rozlišení, volba datového toku, volba CABAC[8] nebo CAVLC[9], možnost synchronního i asynchronního volání
- Error concealment[10] – zajištění hladkého průběhu videa a minimalizace ztráty dat
- Kvalita – dvoufázové režimy ke zvýšení kvality, různé předvolby kvality a výkonu

[6] Software Development Kit
[7] Application Programming Interface
[8] Context-adaptive binary arithmetic coding
[9] Context-adaptive variable-length coding
[10] Technika zpracování signálu za záměrem snížit chybovost kódu ztrátovými daty.

GPU ↓	H.264 YUV 4:2:0		H.264 YUV 4:4:4		H.264 LOSSLESS		H.265 YUV 4:2:0		H.265 YUV 4:4:4		H.265 LOSSLESS	
Maximální hodnoty →	**Barva**	**Rozlišení**	**Barva**	**Rozlišení**	**Barva**	**Rozlišení**	**Barva**	**Rozlišení**	**Barva**	**Rozlišení**	**Barva**	**Rozlišení**
Kepler	8-bit	4096 x 4096	X	X	X	X	X	X	X	X	X	X
Maxwell (1. Gen.)	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	X	X	X	X	X	X
Maxwell (2. Gen.)	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	X	X	X	X
Maxwell (3. Gen.)	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096
Pascal	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	10-bit	8192 x 8192	10-bit	8192 x 8192	10-bit	8192 x 8192
Volta	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	10-bit	8192 x 8192	10-bit	8192 x 8192	10-bit	8192 x 8192
Turing	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	10-bit	8192 x 8192	10-bit	8192 x 8192	10-bit	8192 x 8192

Tab. 2.1: V tabulce je vidět podpora NVENC na různých generacích GPU [20].

2.4.1 NVCUVID

Předchůdcem Video Codec SDK byla Nvidia Video Decoder (NVCUVID). Je to API, která poskytuje vývojářům možnost používat hardwarové funkce k dekódování videa na více operačních systémech. Tato API podporuje video formáty MPEG-2, VC-1, H.264 a HEVC. Aplikace, které používají tuto API, jsou schopny dekódovat videa přímo do paměti. API dokáže spolupracovat s CUDA, OpenGL a DirectX. Pro standardní výstup používá formát videa NV12, který je v níže zmiňovaném NVENC používán jako standardní vstup. Tato API již v dnešní době není podporována a je označena za zastaralou [19].

2.4.2 NVENC

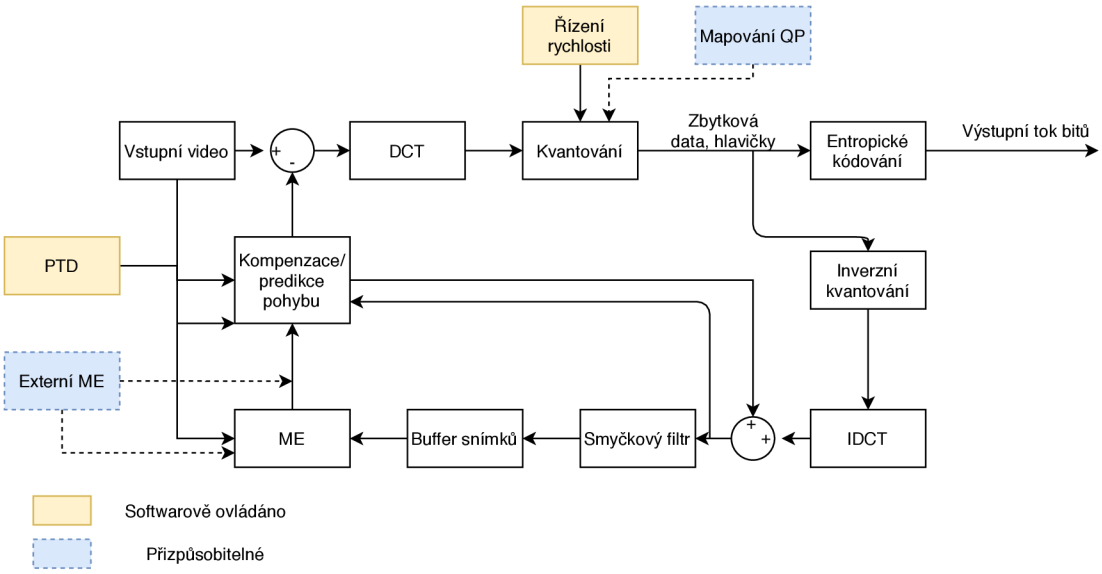
S vydáním první z Kepler generace GPU – GeForce 600 Nvidia představila i NVENC. Jedná se o hardwarový prvek, který zajišťuje plně akcelerované kódování videa. NVENC je nezávislý na grafickém výkonu, tím pádem se GPU může plně věnovat vykonávání jiných operací. Například při nahrávání záznamu z hry se používá NVENC a GPU může použít plnou kapacitu na renderování samotné hry. Přehled podporovaných grafických karet podle generace je v tabulce 2.1.

Nezávislost na GPU není jedinou výhodou NVENC technologie. Mezi další patří:

- Nízká spotřeba z hlediska snížení přenosu dat a fixního hardwaru.
- Nízká odezva – není potřeba přenášet data mezi CPU a GPU.
- Vysoký výkon.
- Vyšší hustota dat na jednom místě.
- Škálovatelnost.
- Jednoduše se programuje pro vlastní využití. Je možné vyvíjet ve více jazycích na operačním systému Linux i Windows. Mezi programovací jazyky patří zejména C, C++, Fortran atd.

Velkým rozdílem mezi grafickými kartami je také počet NVENC čipů, které obsahují. Podle toho existuje i omezení na maximální počet streamů, které lze paralelně vedle sebe využít. Není to ale podmínkou. Například grafické karty GeForce Maxwell a Kepler generace jsou omezeny pouze na 2 streamy, ale Quadra už mají vyšší

maximální počet streamů [21].



Obr. 2.3: Na obrázku je schéma NVENC. [21]

3 Analýza a návrh implementace Video Codec SDK

Cílem diplomové práce je naimplementovat enkodér pro komprimaci 4K videa v reálném čase s použitím NVENC a Video Codec SDK. Co nejlépe aplikaci optimalizovat a analyzovat použití různých API jako je CUDA, DirectX nebo OpenGL. Také je potřeba stanovit limity pro konkrétní hardwarovou konfiguraci a sledovat vliv nastavovaných parametrů na celkové časové i paměťové zátěži vzhledem k výsledné kompresi. Tato kapitola se věnuje podrobnému vysvětlení fungování principů v jednotlivých projektech bez úprav.

3.1 Obecný princip Video Codec SDK

Byly vybrány dva projekty `AppEncCuda` a `AppEncD3D11`, které jsou součástí Video Codec SDK 9.1 jako volně dostupné příklady použití. První z nich využívá k enkódování CUDA a druhý zmíněný provádí enkódování pomocí DirectX. Obě klientské aplikace využívají volání funkcí z NVENCODE API, kterou poskytuje `nvEncodeAPI.dll` pro Windows a `libnvidia-encode.so` pro Linux. Tyto knihovny jsou nainstalovány jako součást ovladače grafické karty. Třetí projekt `AppEncOpenGL` byl pouze analyzován, ale implementace se ho netýkala, protože ho nelze přeložit ve Windows prostředí, ale pouze v operačním systému Linux. Z hlediska překladu projektu pro DirectX, který nelze provést v operačním systému Linux, se autor rozhodl upřednostnit tento projekt nad OpenGL. Obecný postup, jak aplikace provádí proces kódování, je popsán v následujících krocích:

1. Inicializace enkodéru
2. Nastavení parametrů kódování
3. Alokace vstupní a výstupní vyrovnávací paměti
4. Kopírování rámců do vstupní vyrovnávací paměti a čtení výstupního toku bitů
5. Ukončení enkódovací části
6. Dealokace vstupní a výstupní vyrovnávací paměti

Před samotnou inicializací API dojde ke kontrole, zda je verze ovladače grafické karty dostatečná pro spouštěnou verzi Video Codec SDK. Při inicializaci enkodéru se volá metoda `SetInitParams` třídy `NvEncoderInitParam`, která zjistí veškeré parametry, které byly pro enkódování zadány. Ty, které se při spuštění programu nevyplnily, jsou nastaveny na výchozí hodnoty. Potom se na klientu volá funkce `IsCodecH264` a `IsCodecHEVC`, kterými zjistí počet GUIDs[1], které reprezentují požadované kodeky k enkódování video sekvence. Podle toho si pak alokuje vyrovnávací

[1] Globally Unique Identifier

Preset	Nastavení	Zaměření na aplikace
HIGH QUALITY	B rámce, CABAC, 8x8 Transform, všechny intra módy, všechny inter módy, VBR RC, GopLength 30	transkódování s vysokým datovým tokem
HIGH PERFORMANCE	Bez B rámců, CAVLC, P16x16, Intra módy 16x16 a 4x4, VBR, GopLength 30	mnohonásobné transkódování
LOW LATENCY HQ	Bez B rámců, CABAC, všechny intra módy, všechny inter módy, Single frame VBV 2 PASS, nekonečný GOP	cloud gaming, miracast, videokonference
LOW LATENCY HP	Bez B rámců, CABAC, všechny intra módy, všechny inter módy, Single frame VBV 2 PASS, nekonečný GOP, menší vyhledávací rozpětí oproti HQ	cloud gaming, miracast

Tab. 3.1: V tabulce jsou informace o možných presetech, jejich nastavením a využitím.

paměť, aby mohla udržovat informaci o podporovaných GUIDs. Dalším krokem je kontrola, zda zvolený preset je možné použít a zda je podporován grafickým adaptérem. Preset je obecná volba, co se od enkódování očekává, a následně přednastaví množství parametrů s cílem dosáhnout nejlepšího výsledku pro zvolený preset. Informace o možných presetech a jejich nastavení je v tabulce 3.1. Tato tabulka je pouze orientační, protože ne každé GPU podporuje všechny presety.

Dalším krokem je specifikace profilu enkódování pro konkrétní případ. Například kódování videa pro přehrávání na iPhone má jiný profil než video na blue-ray disku. Následně se zjistí jakého formátu je vstupní soubor. Pro inicializaci samotného enkodéru je potřeba volání funkce `CreateEncoder` s validní konfigurací enkodéru.

Jakmile je session inicializována, může klient začít alokovat pamět pro vstupní a výstupní vyrovnávací paměti. Pokud se jedná o enkódování více streamů zároveň, musí se alokovat stejný počet vstupních a výstupních vyrovnávacích pamětí. Proběhne zpracování vstupních souborů a volá se funkce `ConvertRGBToNV12` (pouze u DirectX, Cuda může mít více formátů vstupního souboru), která zajišťuje konverzi původního snímku ve formátu BGR do NV12. V podstatě jde o zarovnání pixelů do formátu NV12 a zapsání do vstupní vyrovnávací paměti. Posléze se snímek předává k enkódování (v každém projektu popsáno zvlášť). V závěru se provede dealokace vyrovnávací paměti, výpis výsledku a zavření cílového souboru.

3.2 AppEncCuda

Aplikace nejdříve otestuje validitu vstupních argumentů a poté inicializuje parametry pomocí funkce `NvEncoderInitParam()`. Všechny základní parametry je nutné předat pomocí přepínačů. Ostatní pokročilé nastavení, jako je volba presetu nebo možnost B rámců, se zadává pomocí pokročilých parametrů, které parser načte jako tokeny. Aby se zajistilo, že API bude využívat ke komprimaci právě CUDA,

musí se změnit parametr `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::deviceType` na `NV_ENC_DEVICE_TYPE_CUDA`. K inicializaci je potřeba vytvořit, najít a ověřit, zda je možné CUDA zařízení využívat a jestli je přítomna správná verze ovladačů. Použití CUDA je možné jak na Linux, tak i na Windows 7 a vyšších verzích. Pokud je nastavení CUDA validní, je možné vytvořit CUDA kontext pomocí volání metody `cuCtxCreate` s argumenty:

- CUcontext* pctx – ukazatel na CUDA kontext
- unsigned int flags – příznak, jak má CUDA zpracovávat přístup z více vláken
- CUdevice dev – aktuální nalezené CUDA zařízení, které bude aplikace využívat

Metoda pro vytvoření kontextu je implementována v CUDA API.

Jakmile jsou všechny parametry nastaveny a kontext inicializován, volá se funkce `void EncodeCuda` s následujícími argumenty:

- int nWidth – šířka rámce
- int nHeight – výška rámce
- NV_ENC_BUFFER_FORMAT eFormat – formát vstupní videosekvence (NV12, YV12, IYUV, YUV444, YUV, ARGB atd.)
- NvEncoderInitParam encodeCLIOptions – informace o vstupních parametrech aplikace
- CUcontext cuContext – CUDA kontext
- std::ifstream &fpIn – ukazatel na vstupní soubor
- std::ofstream &fpOut – ukazatel na výstupní soubor

Nyní se inicializuje samotný enkodér funkcí `InitializeEncoder(EncoderClass &pEnc, NvEncoderInitParam encodeCLIOptions, NV_ENC_BUFFER_FORMAT eFormat)` a nastaví se konfigurace enkodéru. Pokud některý z parametrů nebyl zadán v argumentech aplikace, nastaví se jeho výchozí hodnota. Základní parametry pro komprimaci, jako je vstupní a výstupní soubor, a poté ty nejdůležitější, které budou pro další testování pozměňovány, jsou uvedeny v následujícím seznamu:

- input – cesta ke vstupnímu souboru
- output – cesta k výstupnímu souboru
- codec – je možné použít buď H.264, nebo HEVC (NV_ENC_H264, nebo NV_ENC_HEVC)
- size – rozlišení vstupní videosekvence (např. 1280 x 960)
- bitrate – datový tok
- fps – počet rámců za vteřinu
- rcMode – mód rate control (CONSTQP, VBR, CBR, VBR MINQP, CBR_LOWDELAY_HQ, CBR_HQ, VB_HQ)
- B frames – možnost B rámců
- QP – pouze při módu CONSTQP, značí konstantu, podle které se určuje kvalita endkódování

- preset – volba presetu, možnosti jsou v tabulce 3.1

Před enkódováním je potřeba alokovat vstupní a výstupní buffer. Enkodér postupně načítá rámec po rámci ze vstupní videosekvence pomocí funkce `NVENCSTATUS loadframe(uint8_t *yuvInput[3], HANDLE hInputYUVFile, uint32_t frmIdx, uint32_t width, uint32_t height, uint32_t &numBytesRead)`, která vkládá data rámce do vstupní vyrovnávací paměti. Kontext funkcí se zamkne a dojde na kopírování do CUDA pomocí funkce `memcpy`. Potom dojde na samotné enkódování rámce podle zvolených parametrů pomocí funkce `NvEncEncodeFrame`, která už je součástí NVENC API. Následně se enkódovaný snímek zapíše do dříve zvoleného výstupního souboru. Diagram případu užití je znázorněn na obrázku 3.1 vlevo.

3.3 AppEncD3D11

Nejdříve se zjistí, jestli lze použí DirectX, protože je podporováno pouze operačním systémem Windows 7 a vyššími verzemi. Funkcí `ParseCommandLine_AppEncD3D` se načtou argumenty aplikace, které se zvolí před spuštěním. Zkontroluje se jejich validita a funkce `NvEncoderInitParam` incializuje parametry. Pomocí metody `CreateDXGIFactory1(REFIID riid, _COM_Outptr_ void **ppFactory)` se vytvoří DXGI factory [2], která implementuje metody a funkce pro práci s DirectX. Metodou `D3D11CreateDevice(params)` se vytvoří zařízení, které bude reprezentovat grafický adaptér. Poté se nastaví všechny potřebné parametry pro 2D pole textur, kterými jsou například šířka, výška, velikost pole, formát, příznak přístupu k CPU atd. `CreateTexture2D(const D3D11_TEXTURE2D_DESC *pDesc, const D3D11_SUBRESOURCE_DATA *pInitialData, ID3D11Texture2D **ppTexture2D)` metoda vytvoří 2D pole textur s parametry, které byly v předchozím kroku nadefinovány. Resetuje se konvertor, který převádí snímky z RGB do NV12, pomocí vytvoření nového objektu třídy `RGBToNV12ConverterD3D11(ID3D11Device *pDevice, ID3D11DeviceContext *pContext, int nWidth, int nHeight)`.

Inicializace DirectX a alokace paměti, se kterou bude pracovat, byla dokončena, tak je možné zavolat funkci `void Encode` s argumenty:

- ID3D11Device *pDevice – ukazatel na zařízení DirectX
- ID3D11DeviceContext *pContext – ukazatel na kontext DirectX
- RGBToNV12ConverterD3D11 *pConverter – ukazatel na konverzní funkci z RGB do NV12
- int nWidth – šířka rámce
- int nHeight – výška rámce

[2] DirectX Grpahics Infrastructure

- NvEncoderInitParam encodeCLIOptions – údaje o zadaných vstupních parametrech enkódování
- bool bForceNv12 – příznak, zda je potřeba vstupní snímky konvertovat do formátu NV12
- ID3D11Texture2D *pTexSysMem – ukazatel na 2D pole textur
- std::ifstream &fpBgra – ukazatel na vstupní soubor
- std::ofstream &fpOut) – ukazatel na výstupní soubor

Nyní se provede vytvoření samotného enkodéru `NvEncoderD3D11` s argumenty:

- ID3D11Device* pD3D11Device – ukazatel na zařízení DirectX
- uint32_t nWidth – šířka rámce
- uint32_t nHeight – výška rámce
- NV_ENC_BUFFER_FORMAT eBufferFormat – formát vstupních dat, který může nabývat hodnot NV12, nebo BGRA v závislosti na příznaku *bForceNv12*

Poté se volá funkce `InitializeEncoder`, které se předají v parametrech klientská data, která byla zadána při startu programu, a samotný enkodér. Při inicializaci se volá funkce `SetInitParams`, která nastaví veškeré parametry pro enkódování. Pokud některé z nich nebyly zadány při startu programu, nastaví výchozí hodnoty. Tato funkce zjišťuje také GUID pro určený preset a kodek. Pokud nalezne neshodu v uvedených parametrech, ukončí program s chybovou hláškou "Incorrect parameter".

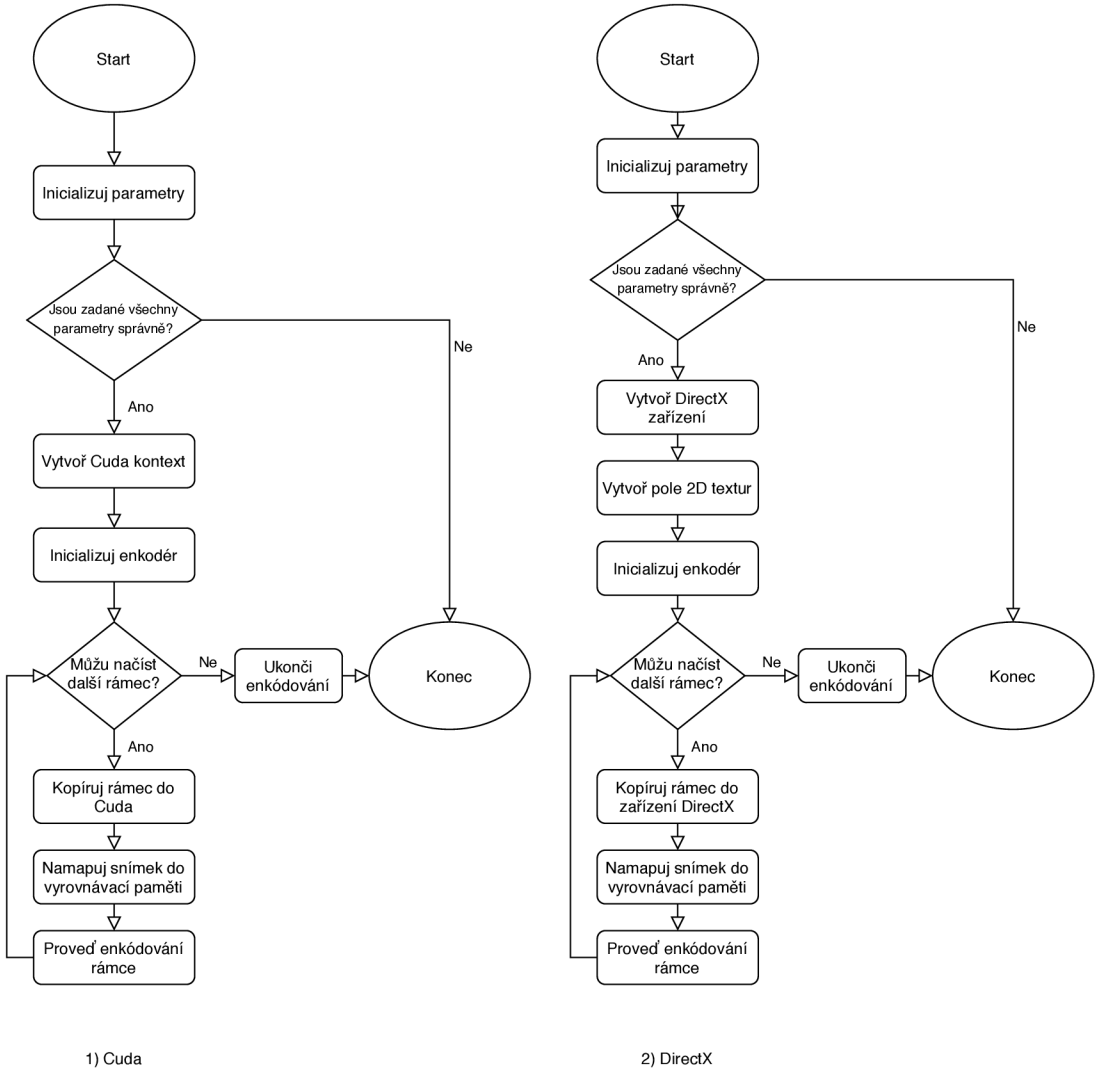
Jakmile je úspěšně inicializovaný enkodér, začne fáze enkódování. Na začátku enkódovací funkce se vytvoří pomocný chytrý ukazatel `std::unique_ptr<uint8_t[]> pHostFrame(new uint8_t[nSize]);`, do kterého se bude vkládat vždy aktuálně kódovaný snímek. Chytrý ukazatel má tu výhodu, že se nemusí manuálně dealokovat paměť, ale ukazatel si hlídá, kdy už není využíván, a následně se dealokuje automaticky. Tím pádem se snižuje pravděpodobnost ztrát paměti. Enkodér zavolá metodu `GetNextInputFrame()`, která alokuje místo ve vyrovnávací paměti a vrací ukazatel na tuto paměť. Snímek se načítá pomocí funkce `std::streamsize ReadInputFrame(const NvEncInputFrame* encoderInputFrame, std::ifstream fpBgra, char *pHostFrame, ID3D11DeviceContext *pContext, RGBToNV12ConverterD3D11 *pConverter, ID3D11Texture2D *pTexSysMem, int nSize, int nHeight, int nWidth, bool bForceNv12)`, jejíž parametry byly popsány v předchozích funkcích. Funkce provede kopírování vstupního snímku ze souboru do vyrovnávací paměti DirectX, poté v závislosti na bForceNv12 provede konverzi snímku na jiný formát.

Po načtení snímku do vyrovnávací paměti si enkodér zavolá metodu `void EncodeFrame`, ve které namapuje vstupní vyrovnávací paměti enkodéru na právě kódovaný snímek a provede proces enkódování. Pokud se proces provede úspěšně, výsledný snímek vloží do paměti, na kterou ukazuje parametr vPacket. Poslední fází v každém cyklu je zapsání enkódovaného rámce do výstupního souboru. To se

provede pomocí funkce `ostream write` s argumenty:

- const char* s – ukazatel na pole znaků
- streamsize n – počet znaků, které se mají vložit

Enkódovací proces končí tím, že ve vstupním souboru už nebudou snímky, takže se nebude rovnat počet předpokládaných znaků reálně načteným znakům. Pokud se tak stane, volá se metoda enkodéru `EndEncode()`, která vyšle příznak o ukončení vstupního souboru a uvolní vyrovnávací paměti rezervované enkodérem. Poté se volá metoda enkodéru `DestroyEncoder()`, která uvolní vyrovnávací paměti rezervované DirectX, deinicializuje se enkodér a zavřou se vstupní a výstupní soubory. Diagram případu užití je znázorněn na obrázku 3.1 vpravo.



Obr. 3.1: Diagramy případů užití projektů AppEncCuda a AppEncD3D11.

3.4 AppEncGL

Jak už bylo zmíněno v úvodu kapitoly, tomuto projektu nebylo věnováno mnoho pozornosti vzhledem k spustitelnosti pouze v operačním systému Linux. Byla provedena pouze analýza projektu pomocí případu užití.

Nejdříve se načtou vstupní argumenty programu zadané uživatelem. Pomocí funkce `GraphicsSetupWindow(contextType)` se vytvoří nový kontext OpenGL. Ověří se validita vstupních parametrů a zda byl úspěšně vytvořen kontext OpenGL. Volá se funkce `void EncodeGL` s následujícími argumenty:

- char *szInFilePath – ukazatel na vstupní videosekvenci
- char *szOutFilePath – ukazatel na výstupní soubor, kam bude probíhat zápis snímků po enkódování
- int nWidth – šířka rámce
- int nHeight – výška rámce
- NV_ENC_BUFFER_FORMAT eFormat – typ vstupního souboru, který může nabývat hodnoty YUV nebo NV12
- NvEncoderInitParam *encodeCLIOptions – vstupní parametry programu

Ve funkci se otevře vstupní a výstupní soubor. Poté se vytvoří nová instance enkodéru podle šablony `NvEncoderGL`. Provede se inicializace parametrů enkodéru a přiřadí se enkodéru pomocí metody `CreateEncoder(initializeParams)`. Pokud se enkodér úspěšně vytvoří a inicializuje, začíná enkódovací fáze, kde se v cyklu načítají snímky ze vstupního souboru pomocí metody enkodéru `GetNextInputFrame()`. Načtený rámec se uloží do vyrovnávací paměti OpenGL textur a provede se jeho enkódování přes metodu `EncodeFrame()`. Výsledný rámec se následně zapíše do výstupního souboru. Pokud už ve vstupní videosekvenci není načten celý snímek, volají se metody `EndEncode()` a `DestroyEncoder()`, které se postarají o dealokaci enkodéru a všech alokovaných vyrovnávacích pamětí. Poslední fází je zavření vstupního a výstupního souboru a ukončení programu.

3.5 Návrh implementace rozšíření Video Codec SDK

SDK nyní umožňuje zpracovávat data ze vstupní videosekvence, avšak cílem diplomové práce je načítat snímky ze 4K kamer. Pro načítání rámců z kamer je potřeba naimplementovat funkce pro konfiguraci a ovládání kamer. Pro používané zařízení mvBlueFox existuje rozhraní mvImpact Acquire SDK, které poskytuje některé užitečné funkce a přístup k těmto typům kamer[3].

[3]Dokumentace k této knihovně lze nalézt na webové stránce: https://www.matrix-vision.com/manuals/SDK_CPP/index.html

Aby bylo možné zpracovávat data ze dvou a více kamer zároveň, je potřeba, aby procesy načítání snímků běžely paralelně. V případě této diplomové práce je potřeba paralelizovat i enkódování snímků. Pro práci s více vlákny byla vybrána knihovna Win 32, která poskytuje API pro *multithreading*. Pro jednu kameru se v závěru budou tvořit dvě vlákna, která budou mít společnou sdílenou paměť. Pro zvolenou hardwarovou konfiguraci, kterou má autor k dispozici, je maximální počet kamer, které dokáže čip NVENC enkódovat paralelně, roven 2, tudíž celkový počet vláken v programu je 4. Dvě vlákna jsou určena pro čtení snímků z kamer a další dvě provádí enkódování načítaných rámců a jejich ukládání do souboru. Po výměně grafické karty v konfiguraci za GPU, která dokáže zpracovávat více než 2 kamery zároveň je vždy počet vláken vláken roven dvojnásobku připojených kamer.

4 Implementace aplikace

V této kapitole bude autor popisovat vlastní implementaci rozšíření Video Codec SDK. V první části se věnuje rozdělení procesů do vláken. V další části je rozebráno načítání snímků z kamer. Třetí sekce pojednává o procesu enkódování a poslední část je věnována implementaci grafického rozhraní.

4.1 Programovací jazyk a použité nástroje

CUDA podporuje více programovacích jazyků, doporučeným pro tuto diplomovou práci byl jazyk C++. Posléze byl i vybrán jako nejvhodnější varianta, protože je k dispozici řada knihoven podporujících volání funkcí CUDA a z hlediska následné optimalizace kódování je ideálním řešením.

Pro tvorbu GUI[1] byla vybrána knihovna Windows Forms, která je součástí .NET Frameworku. Slouží k jednoduchému vytvoření uživatelského prostředí s ovládacími prvky, které jsou používány ve Windows. Ve Windows Forms lze využít jazyky C♯ a Visual Basic, ale v projektu je použit pouze C♯, protože autorovi více vyhovuje.

Jako IDE[2] bylo zvoleno MS Visual Studio 2019. IDE bylo vybráno kvůli doporučení na mnoha fórech a pluginu Nsight, který zjednodušuje procesy debugování, profilování a analýzy. Testovací sada použitá pro spouštění projektů je v tabulce 5.2.

4.2 Rozdělení procesů do vláken

Jak už bylo zmíněno v návrhu implementace, k vytvoření více vláken je použita knihovna WinAPI. Lze vytvořit více vláken zároveň, proto je nejdříve potřeba si zjistit počet připojených kamer pomocí kódu 4.1.

```
1    DeviceManager devMgr;
2    globvar::devCnt = devMgr.deviceCount();
3    if (globvar::devCnt == 0) {
4        cout << "Nebylo nalezeno žádné zařízení!" << endl;
5        getchar();
6        return 0;
7    }
8    else cout << "Počet připojených zařízení: "
9    << globvar::devCnt << endl;
```

Výpis 4.1: Kód znázorňující zjištění počtu připojených kamer k počítači.

[1] Graphical User Interface

[2] Integrated Development Environment – Vývojové prostředí

Nyní je známo kolik vláken můžeme vytvořit, ale tento počet je ještě omezen hardwarovou konfigurací počítače, konkrétně jaká grafická karta je v sestavě. V tomto případě je maximální počet zpracovávaných kamer 2. Nyní se vytvoří odpovídající počet výstupních souborů. Tyto soubory mají název kombinací slova out, čísla kamery a právě používaného kodeku. Nově vytvořený soubor se otevře pro zápis a ukazatel se uloží do vektoru. Nové vlákno se vytváří voláním funkce `CreateThread` s následujícími argumenty:

- LPSECURITY_ATTRIBUTES lpThreadAttributes – pokud není NULL, definuje, kdy může být řízení předáno potomkovi
- SIZE_T dwStackSize – inicializační velikost v bajtech, pokud je argument 0, pro nové vlákno se alokuje výchozí velikost
- LPTHREAD_START_ROUTINE lpStartAddress – reprezentuje výchozí adresu vlákna
- LPVOID lpParameter – ukazatel na parametry přenášené do vlákna
- DWORD dwCreationFlags – příznaky pro vytváření vlákna např. spouští se hned, nebo se uspí a musí se startovat manuálně
- LPDWORD lpThreadId – ukazatel na proměnnou, která vlastní identifikátor vlákna

Do lpStartAddress se většinou přiřazuje funkce, kterou je potřeba v novém vlákně spustit. Ta přijímá jediný argument PVOID lpParams, který je předáván také při vytváření vlákna a má návratový typu DWORD (unsigned long). Další neméně důležitou funkcí WinAPI je `WaitForMultipleObjects`, která čeká, dokud se všechna vytvořená vlákna z hlavního vlákna neukončí, a pak až pokračuje. Pokud je potřeba předat funkci nějaké parametry, musí se vytvořit ukazatel na proměnnou. Pokud je těchto proměnných více, vytvoříme třídu, jejíž objekty ponesou všechny tyto proměnné. V případě této práce se jedná o 2 třídy. První je pro přenos informací o aktuálně zpracovávaném zařízení (kameře) s názvem `ThreadParameter`, který nese informace o kameře a ukazatel na samotné zařízení. Druhou zmíněnou třídou je `NVENCTHREAD_PARAMS`, která nese parametry pro enkódování. Vytvoření parametrů a nového vlákna pro enkódování s funkcí pro čekání na návrat vláken je zobrazeno na kódu 4.2

```
1  std::vector<NVENCTHREAD_PARAMS*> QuadroParams;
2
3  for (int i = 0; i < globvar::devCnt; i++)
4  {
5      QuadroParams.push_back(new NVENCTHREAD_PARAMS (i, eFormat,
6      cuContext, nWidth, nHeight, encodeCLIOptions, outputs[i]
7      ));
8  }
```

```
9
10      vSize = QuadroParams.size();
11      HANDLE* hQuadro = new HANDLE[vSize];
12
13      for (int i = 0; i < vSize; i++)
14      {
15          DWORD dwQuadroId;
16          hQuadro[i] = CreateThread(NULL, 0,
17          NvencThread, (LPVOID)QuadroParams[i], 0, &dwQuadroId);
18      }
19
20      WaitForMultipleObjects(
21      static_cast<DWORD>(vSize), hQuadro, true, INFINITE);
```

Výpis 4.2: Kód znázorňující vytvoření parametrů a vláken pro enkódování v závislosti na počtu připojených kamer.

4.3 Načítání snímků z kamer

Načítání snímků z kamer se provádí pomocí knihovny mvImpact Acquire SDK. Zaprvé je potřeba zjistit počet připojených kamer pomocí kódu 4.1. Vytvoří se manažer zařízení, přes který se bude zařízení nastavovat a ovládat. Voláním funkce `globvar::buffstack_init(globvar::devCnt)` se alokuje místo pro globální proměnné kamery pro počet připojených zařízení. Každé vlákno potřebuje své parametry. Nejdůležitějším parametrem je manažer zařízení, který je objektem třídy `DeviceManager`. Při přidělování parametrů se zároveň zjišťuje velikost pořizovaných rámců. To se provede vytvořením instance `GenICam::ImageFormatControl ifc(devMgr[i])`, kde je v argumentu manažer zařízení na indexu právě zpracovávané kamery. Tento objekt nese informace o výšce a šířce rámců, ale také hloubce, offsetech atd. Potřebná výška a šířka rámce se přiřadí do globálních proměnných a bude se používat i při procesu enkódování.

Před vytvářením vlákna se pro každou kameru zavolá funkce `configureDevice`, které se předá manažer zařízení. Tato funkce má za úkol nastavit potřebné parametry pro načítání snímků. Nastavuje se především formát pixelů a přítomnost alfa kanálu. Po vytvoření vlákna na počáteční adrese funkce `DWORD WINAPI liveThread(LPVOID pData)` se do proměnné přepíše manažer zařízení z předávaného parametru. Nejdříve se zkontruluje, zda je zařízení otevřeno, pokud není, proběhne pokus o otevření pomocí metody `open()`. Pokud zařízení nelze otevřít, vypíše se chybová hláška a ukončí se program. Poté se vytváří objekt, který nese statistiku zařízení a objekt funkčního

rozhraní třídy `FunctionInterface`. Tato třída obsahuje všechny základní funkce pro práci s kamerou. K vytvoření nového funkčního rozhraní je potřeba předat konstruktoru ukazatel na zařízení.

Nyní se vytvoří kontrolní objekty zařízení, kterými se nastaví, aby se získávání snímků a expoziční čas řídily automaticky a nastaví se počet snímků za sekundu, který se zadává v argumentech programu. Spustí se funkce `manuallyStartAcquisitionIfNeeded(pDev, fi)`, která připraví ovladač na zachytávání dat a spustí načítání snímků z kamery. Po spuštění se začnou tvořit požadavky na snímky, které se začnou řadit do fronty. Tento proces je znázorněn v kódu 4.3

```
1  requestNr = fi.imageRequestWaitFor(timeout_ms);
2  if (fi.isRequestNrValid(requestNr)) {
3    pRequest = fi.getRequest(requestNr);
4    if ((pRequest->isOK()) && ((pRequest->infoFrameNr.read()) > 0)) {
5      ++cnt;
6      saveImages(pRequest, pDev);
7    }
8    if (!(pRequest->isOK())) {
9      {
10       LockedScope lockedScope(globvar::g_critSect_print);
11       cout << "Error: " << pRequest->requestResult.readS() << endl;
12       getchar();
13     }
14   }
15   if (fi.isRequestNrValid(lastRequestNr)) {
16     fi.imageRequestUnlock(lastRequestNr);
17   }
18   lastRequestNr = requestNr;
19   fi.imageRequestSingle();
20 }
```

Výpis 4.3: Kód znázorňující načítání snímků z kamery.

Nejdříve se musí vytvořit požadavek na načtení snímku z kamery, to se provádí pomocí funkce `imageRequestSingle()`. První požadavek se tvoří spouštěcí funkcí, která byla zmíněna výše. Na řádku 1 se čeká na zpracování snímku. Této metodě se předává doba, po které bude načtení neúspěšné, protože se požadavek zpracovával příliš dlouho. Na druhém řádku se zkontroluje validita výsledku požadavku. Následně se kontroluje výsledek této validace, jestli je vše v pořádku, výsledek se pošle na uložení. Pokud snímek není v pořádku, vypíše se chybová hláška. Před zapsáním

snímku se musí odemknout poslední načítaný snímek, aby se mohl uzamknout nový rámec.

Další důležitou funkcí je `void saveImages`, jejíž argumenty jsou:

- Request* – ukazatel na výsledek požadavku pro navrácení rámce
- Device* – ukazatel na připojené zařízení

Tato funkce vkládá snímky do vyrovnávací paměti RAM, odkud si je posléze vytahuje enkodér. Tento proces ukládání snímků je zobrazen v zjednodušeném kódu 4.4

```
1  for (int i = 0; i < globvar::devCnt; i++)
2  {
3    if (device == globvar::buffstack.camnumb[i])
4    {
5      size_t alloc_mem = pRequest->imageWidth.read() *
6      pRequest->imageHeight.read() * globvar::channels;
7      unsigned char* pTexSysBuf = (unsigned char*)malloc(alloc_mem);
8      memcpy(pTexSysBuf, pRequest->imageData.read(), alloc_mem);
9      globvar::buffstack.tex_stacks[i].push(pTexSysBuf);
10   }
11 }
```

Výpis 4.4: Kód znázorňující ukládání snímků do fronty v paměti RAM.

Program se ukončí, jakmile uběhne časový limit nastavený uživatelem. Z hlediska implementace se načítání rámců zastaví, pokud je počet již načtených snímků roven součinu délky vstupu a hodnoty počtu rámců za sekundu. Při zastavení programu se nastaví příznak vlákna na ukončeno a volá se funkce `manuallyStopAcquisitionIf Needed (pDev, fi)`, která se postará o zastavení získávání snímků ovladačem kamery.

4.4 Proces enkódování

Pro vlákno, které zařizuje enkódování, je potřeba předat parametry, podle kterých se vytvoří instance enkodéru. Přiřazení parametrů a vytvoření nového vlákna pro enkódování je zobrazeno v kódu 4.2. Po vytvoření vlákna se zavede lokální proměnná, která ponese ukazatel na parametry předávané do funkce. Zbytek kódu funkce se zaobalí do try catch bloku. Nyní se vytvoří objekt enkodéru voláním konstruktoru s argumenty konkrétního zařízení, rozměrů rámce a formátu vstupních dat. Po úspěšném vytvoření enkodéru se provede jeho inicializace. Při inicializaci se nastaví parametry, které byly zadány uživatelem při startu programu. Ostatní, které nebudou

použity pro testování, se nastaví na výchozí hodnotu. Metodou `CreateEncoder` se vytvoří enkodér s nastavenými parametry.

Nyní se začne provádět cyklus, který poběží dokud nebude příznak vlákna nastaven na ukončen. V tomto cyklu běží další cyklus, který kontroluje velikost fronty, na kterou se v druhém vlákně zapisují načítané snímky z kamery. Pokud je tento počet větší než 0, začne se provádět enkódování. Následující proces je zobrazen v kódu 4.5.

```
1  if (nFrame == length) {
2      pNvencThreadParams->terminateThread();
3  }
4  nFrame++;
5  uint8_t* pTexSysBuf = new uint8_t[nSize]();
6  CopyFrameToText3chann(globvar::buffstack.tex_stacks
7  [pNvencThreadParams->count].front(), pTexSysBuf, width, height);
8  unique_ptr<uint8_t[]> pHostFrame(pTexSysBuf);
9
10 const NvEncInputFrame* encoderInputFrame = pEnc->GetNextInputFrame();
11 // v této části závisí na zvoleném projektu a dochází
12 // ke kopírování rámce do zvoleného zařízení
13
14 pEnc->EncodeFrame(vPacket);
15
16 for (std::vector<uint8_t>& packet : vPacket)
17 { std::endl;
18     pNvencThreadParams->fpOut->write(reinterpret_cast<char*>(
19     packet.data()), packet.size());
20 }
21
22 auto lastFrame =
23 globvar::buffstack.tex_stacks[pNvencThreadParams->count].front();
24 globvar::buffstack.tex_stacks[pNvencThreadParams->count].pop();
25 delete[] lastFrame;
```

Výpis 4.5: Kód znázorňující enkódovací proces.

Pokud bude rovnem počet enkódovaných rámců součinu délky vstupu a hodnoty počtu rámců za sekundu, tak se vláknu nastaví příznak ukončeno. Zavolá se funkce `CopyFrameToText3chann` s argumenty:

- unsigned char* frame – ukazatel na vstupní rámec, v tomto případě první snímek ve frontě ukládaných snímků z kamery

- uint8_t* pRGB – ukazatel na výsledný rámec
- int width – šířka rámce
- int height – výška rámce

Tato funkce se postará o rozšíření rámce o prázdný alfa kanál, protože enkodér očekává vstup rámce se čtyřmi kanály. Její tělo lze vidět na kódu 4.6. Z výsledku této funkce se vytvoří chytrý ukazatel, který se o alokaci a dealokaci paměti postará sám. Zavolá se metoda enkodéru `GetNextInputFrame`, která se posune v paměti o velikost rámce a vrací ukazatel na další snímek. Poté se musí nakopírovat data snímku do vstupní vyrovnávací paměti enkodéru. Pak se volá metoda enkodéru `EncodeFrame`, které předáváme ukazatel, kam se po komprimaci nakopírují data výsledného snímku. Pokud v tomto ukazateli jsou nějaká data, proběhne zápis do výstupního souboru. Nakonec se musí provést dealokace paměti původního nekomprimovaného snímku z fronty.

```
1  int channels = 3;
2  int dwStride = width * 4;
3  unsigned char* ptr_in = frame;
4
5  for (int j = 0; j < height; j++)
6  {
7      unsigned char* ptr_in_row = ptr_in + j * width * channels;
8      unsigned char* p4 = pRGB + j * (dwStride);
9
10     for (int i = 0; i < width; i++)
11     {
12         p4[i * 4 + 0] = *(ptr_in_row++);
13         p4[i * 4 + 1] = *(ptr_in_row++);
14         p4[i * 4 + 2] = *(ptr_in_row++);
15         p4[i * 4 + 3] = 0;
16
17     }
18 }
```

Výpis 4.6: Kód znázorňující rozšíření rámce o nulový alfa kanál.

Po ukončení enkódování se volá metoda enkodéru `EndEncode`, která zařídí, aby se všechny fronty enkodéru vyprázdnily, aby mohly být uvolněny. Tato funkce je nezbytná k úplnému uvolnění enkodéru pomocí metody `DestroyEncoder`. Ta uvolní všechny alokované zdroje a zakončí enkódovací session. Ještě před ní se zavře výstupní soubor. Poté se ukončí samotné vlákno.

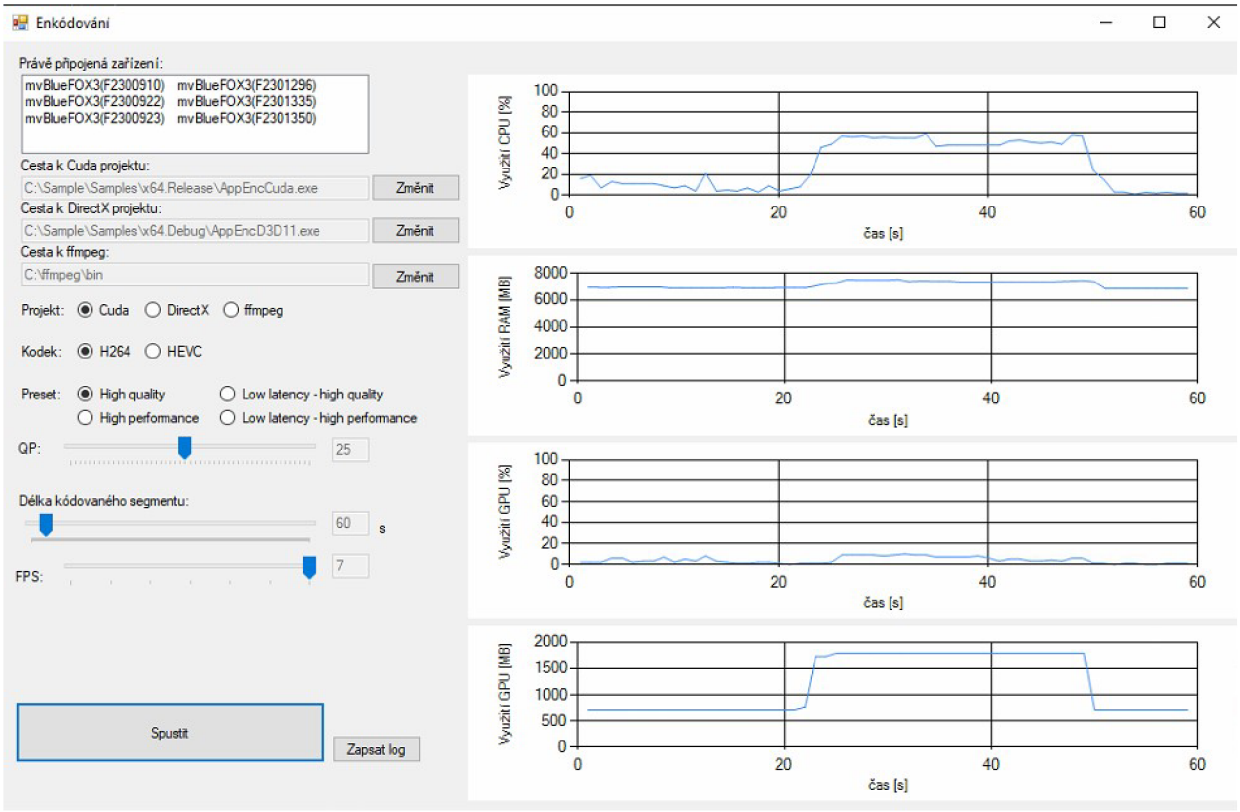
4.5 Implementace grafického rozhraní

Pro implementaci grafického rozhraní bylo využito Windows Forms. Bylo zvoleno, protože se jedná o open-source knihovnu, která je součástí .NET Frameworku. Představuje platformu pro psaní klientských desktopových aplikací. Vzhled aplikace je na obrázku 4.1 Při implementaci diplomové práce byly použity následující prvky:

- Form – Základní zobrazovací prvek. Okno, které zobrazuje všechny ostatní prvky.
- RadioButton – Je možné zvolit jedinou možnost ve skupině prvků radioButton.
- Label – Většinou zobrazuje textový popis, nebo instrukce k dalšímu kontrolnímu prvku, nebo formu.
- TextBox – Zobrazuje hodnotu zadanou uživatelem. Může nabývat masek a zobrazovat hesla, může být pouze pro čtení.
- NumericUpDown – Zobrazuje číselnou hodnotu. Je možné přidávat a ubírat hodnoty šipkami nahoru a dolů.
- ProgressBar – Má funkci nastavení hodnoty z množiny hodnot. Nastavuje se tažením jezdce mezi maximem a minimem.
- ListView – Zobrazuje prvky seznamu, nebo podseznamu. Může zobrazovat na každém řádku jiný prvek, nebo lze vytvořit sloupce.
- Button – Reprezentuje tlačítko, provede akci na klik, nebo stlačení klávesy.
- GroupBox – Primárně slouží pro shromáždění více prvků radioButton, protože v může obsahovat právě jeden zvolený radioButton. Ohraničuje prvky čárou.

Tyto grafické prvky jsou spojeny s řídící částí, kterou lze nastavit v jejich vlastnostech, nebo manuálně naprogramovat. Po spuštění aplikace se provede inicializace všech komponent. Následně se vytváří nová vlákna pro sledování výkonu procesoru, paměti RAM a výkonu GPU. Je potřeba zjistit počet kamer a nejvyšší možnou hodnotu rámců za sekundu. To se provede spuštěním nového procesu, který spustí přes příkazovou řádku aplikaci s příznakem *-maxfps*, která vypíše na výstup název kamery, její sériové číslo a maximální možnou hodnotu rámců za sekundu. Aplikace si tento řádek přečte a vyčte z něj dostupné informace, které použije pro omezení maximální hodnoty ProgressBaru pro nastavování počtu rámců za sekundu. Názvy kamer přiřadí do prvku ListView a zobrazí je jako řádky.

K získání hodnot grafů slouží pro CPU a RAM třída `PerformanceCounter`. K získání hodnot z GPU bylo potřeba doinstalovat nuget balíček NvAPIWrapper. Informace o aktuálním využití nese třída `PhysicalGPU`, která reprezentuje aktuálně zapojenou grafickou kartu. Aktualizace grafů se provádí pomocí metody `UpdateGPUChart`, která přidá do zobrazovaného pole novou hodnotu. Celý proces je zobrazen v kódu 4.7.



Obr. 4.1: Na obrázku je ukázka vytvořeného grafického rozhraní.

```
1  private double[] GPUArray = new double[60];
2  private void UpdateGPUChart()
3  {
4      GPUChart.Series["Series1"].Points.Clear();
5      for (int i = 0; i < GPUArray.Length - 1; ++i)
6      {
7          GPUChart.Series["Series1"].Points.AddY(GPUArray[i]);
8      }
9  }
10
11 private void getPerformanceGPUCounters()
12 {
13     var gpu = PhysicalGPU.GetPhysicalGPUs();
14     while (true)
```

```
15      {
16          var usage = gpu[0].UsageInformation.GPU.Percentage;
17          GPUArray[GPUArray.Length - 1] = usage;
18
19          Array.Copy(GPUArray, 1, GPUArray, 0, GPUArray.Length - 1);
20
21          if (GPUChart.IsHandleCreated)
22          {
23              this.Invoke((MethodInvoker)delegate { UpdateGPUChart(); });
24          }
25          Thread.Sleep(1000);
26      }
27  }
```

Výpis 4.7: Kód znázorňující snímání hodnot využití GPU a aktualizaci grafu.

Uživatel nastaví všechny parametry programu, se kterými chce cílovou aplikaci spustit. Poté stiskne tlačítko spustit, které načte všechny hodnoty ovládacích prvků a začne spouštět aplikaci. To se provede vytvořením nového procesu, ve kterém se otevře příkazová řádka a zadá se tam cesta k souboru podle toho, jaký projekt byl zvolen. Následně se za cestu k souboru připíšou argumenty, které byly sečteny z ovládacích prvků. Proces je na ukázce kódu 4.8. Celý proces je sledován a 3 sekundy před tím, než je spuštěn, se začnou zapisovat všechny načítané hodnoty grafů, které se postupně oddělují středníkem. Po stlačení druhého tlačítka *Zapsat log* se vytvoří soubor s názvy parametrů ve formátu csv, ze kterého lze jednoduše tvořit grafy.

```
1  string genArgs = codec + preset + fps + QP + length;
2  isLogging = true;
3  InitTimer();
4  Thread.Sleep(3000);
5  try
6  {
7      runProg.StartInfo.FileName = pathToFile;
8      runProg.StartInfo.Arguments = genArgs;
9      runProg.StartInfo.CreateNoWindow = true;
10     runProg.Start();
11 }
12 catch (Exception ex)
13 {
14     MessageBox.Show("Nebylo možné spustit program" + ex);
15 }
```

Výpis 4.8: Ukázka kódu vytvoření nového procesu aplikace.

MvBlueFox3	
Model	1100
Varianta	C
Rozlišení	3856 x 2764
MPixel	10.7
Max. snímková frekvence	7.3 Hz

Tab. 5.1: V tabulce jsou uvedeny hardwarové prostředky použité ke spuštění enkodéru.

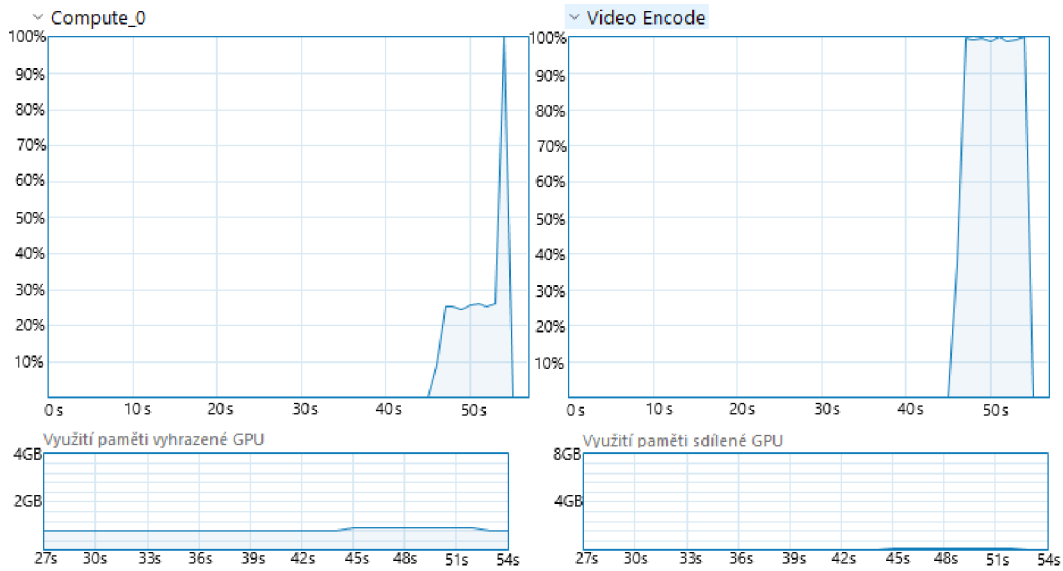
5 Testování

Testování proběhlo na testovací sadě zobrazené v tabulce 5.2. V průběhu testování byly použity kamery mvBlueFOX3. Všechny testy byly spouštěny s grafickou kartou č. 1, pokud u testu není uvedeno jinak. Parametry této kamery jsou zobrazeny v tabulce 5.1. Všechny testy pro dvě kamery byly spouštěny na kamerách s velice podobnou scénou. Při použití více kamer byly zachytávané scény rozdílné. Všechny testy byly spouštěny v debug módu. Testy jsou rozvrženy na výkonnostní a kvalitativní. Mezi výkonnostní patří porovnávání nastavovaných parametrů na vytížení GPU, CPU a RAM. Dále test na průměrnou dobu trvání části kódu a test na maximální vytížení enkodéru v reálném čase. V testech zaměřených na kvalitu se budou pozorovat komprimační ztráty enkódovaného snímku v porovnání s originálem.

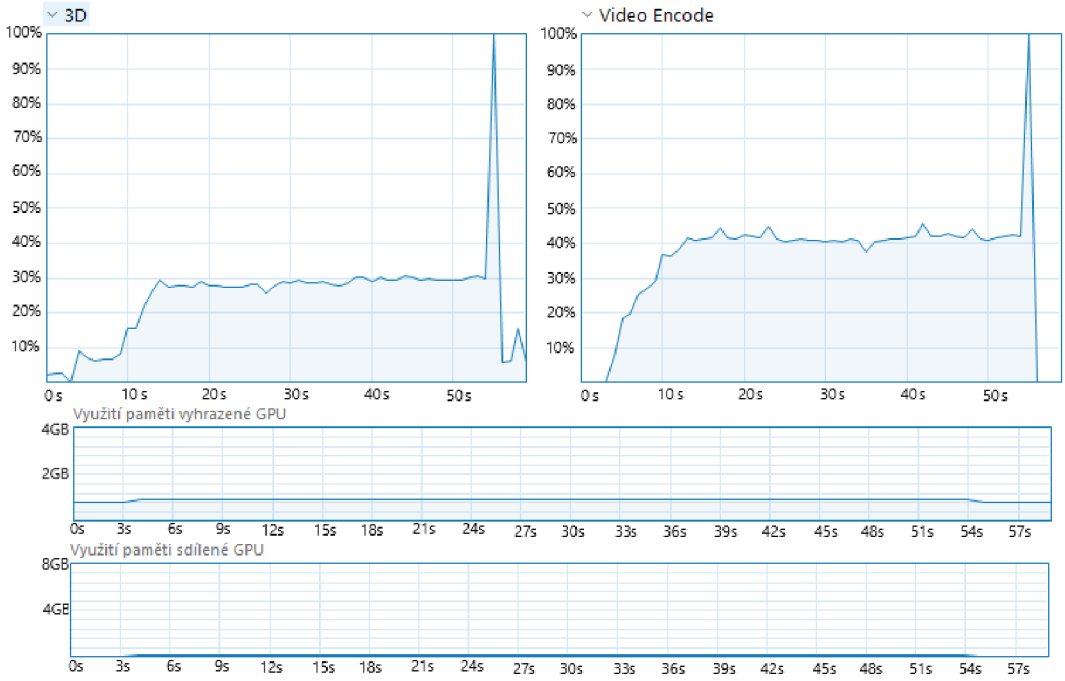
5.1 Porovnání výsledků analýzy jednotlivých projektů

Jakmile mají oba projekty shodné vstupní soubory, je možné porovnat rychlost enkódování. Testování probíhalo na testovací sadě 5.2. Jako vstupní soubory byly použity dvě vstupní sekvence, jejichž parametry jsou zobrazeny v tabulce 5.3. Porovnání výsledků bylo zaznamenáno do tabulky 5.4. V tabulce je zaznamenána hodnota z patnácti spuštění projektu na stejných parametrech a vstupech. Z výsledné sekvence byla vypočtena směrodatná odchylka. Grafy byly pozorovány v aplikaci Správce úloh na odrážce výkon. Na obrázcích 5.1 a 5.2 jsou grafy, které vznikly při enkódování zdrojového souboru s rozlišením 3840x2160 nazvaného *bees*.

Na obrázku 5.1 jsou vidět grafy porovnání výpočetního výkonu GPU a alokace paměti pro projekt AppEncCuda. Na pravé straně nahoře je graf zobrazující vytížení čipu NVENC. Vlevo nahoře je graf znázorňující využití výpočetního výkonu CUDA. Další dva grafy se věnují alokaci paměti. První znázorňuje alokaci paměti vyhrazené GPU (buffery) a druhý využití sdílené paměti GPU.



Obr. 5.1: Na obrázku jsou grafy využití výpočetního výkonu CUDA a graf znázorňující vytížení čipu NVENC.



Obr. 5.2: Na obrázku jsou grafy využití výpočetního výkonu DirectX a graf znázorňující vytížení čipu NVENC.

Na obrázku 5.2 jsou grafy porovnání výpočetního výkonu GPU a alokace paměti pro projekt AppEncD3D11. Na pravé straně nahoře je graf zobrazující vytížení čipu

Testovací sada	
Operační systém	Windows 10 Education, 64bit
Procesor	Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz
RAM	16 GB DDR4, 3000 MHz
Grafická karta 1	NVIDIA GeForce GTX 1050 Ti, Pascal, 4 GB GDDR5
Grafická karta 2	NVIDIA Quadro P2000, 5 GB GDDR5
Disk	SSD disk Intel 600p M.2 256GB

Tab. 5.2: V tabulce jsou uvedeny hardwarové prostředky použité ke spuštění enkodéru.

Název	**Rozlišení**	**FPS**	**Popis**
bunny	1280x720	30	animovaná pohádka, dynamické změny, změny prostoru
bees	3840x2160	24	záběr na včelí roj, pohyb pouze včel

Tab. 5.3: V tabulce jsou některé atributy a popis zdrojových videosekvencí před komprimací.

NVENC. Vlevo nahoře je graf znázorňující výpočetní výkon DirectX. Další dva grafy se věnují alokaci paměti. První znázorňuje alokaci paměti vyhrazené GPU (buffery) a druhý sdílené paměti GPU.

Na základě tabulky a grafů lze vidět, že projekt, kde byla využívána CUDA, provedl enkódování v kratším čase než projekt, který využíval DirectX. Každý z projektů z této testované starší verze Video Codec SDK musel mít jiný formát vstupních dat. Pro projekt využívající CUDA je vstupní formát YUV. Pro projekt využívající DirectX byl vstupním formátem adresář s BMP snímky. To má za důsledek, že velikost vstupních dat pro DirectX je skoro dvojnásobná oproti CUDA. To objasňuje i celkovou dobu trvání enkódování, protože projekt využívající DirectX musel načítat vstupní data složitějším způsobem. Každý snímek musel načítat zvlášť ze souboru, a proto byl čip NVENC využíván ze 40 % celkového výkonu. Projekt využívající CUDA pouze na začátku enkódování otevřel soubor a celou dobu se v něm posouval a četl data. To vysvětluje využití čipu NVENC na maximální hodnotu. Alokace pamětí byla pro CUDA i DirectX poměrně podobná stejně jako velikost výsledných souborů.

Video	**bunny**		**bees**	
Rozlišení	1280x920		3840x2160	
Počáteční velikost pro CUDA	5 765 850 kB		8 723 700 kB	
Kodek	H.264	H.265	H.264	H.265
Koncová velikost bitového streamu	13 610 kB	14 259 kB	29 195 kB	35 111 kB
Výsledný čas kódování	5 873,62 $\pm$ 5, 86 ms	5 511,9 $\pm$ 5, 62 ms	7 436,55 $\pm$ 13 ms	7 357,05 $\pm$ 74, 5 ms
Průměrný čas pro jeden frame	1,83 $\pm$ 0 ms	1,29 $\pm$ 0, 12 ms	10,94 $\pm$ 0, 24 ms	10,56 $\pm$ 0, 29 ms
Počáteční velikost pro DirectX	10 912 645 kB		16 654 214 kB	
Kodek	H.264	H.265	H.264	H.265
Koncová velikost bitového streamu	14 511 kB	15 294 kB	31 005 kB	36 215 kB
Výsledný čas kódování	35 504 $\pm$ 557 ms	35 441 $\pm$ 354, 7 ms	51 546,33 $\pm$ 530, 4 ms	51 784 $\pm$ 197 ms
Průměrný čas pro jeden frame	8,31 $\pm$ 0, 129 ms	8,3 $\pm$ 0, 07 ms	71,79 $\pm$ 0, 74 ms	72,12 $\pm$ 0, 274 ms

Tab. 5.4: V tabulce jsou zobrazeny výsledky enkódování videí v obou projektech a za použití standardů H.264 s porovnáním v H.265.

5.2 Výkonnostní testy

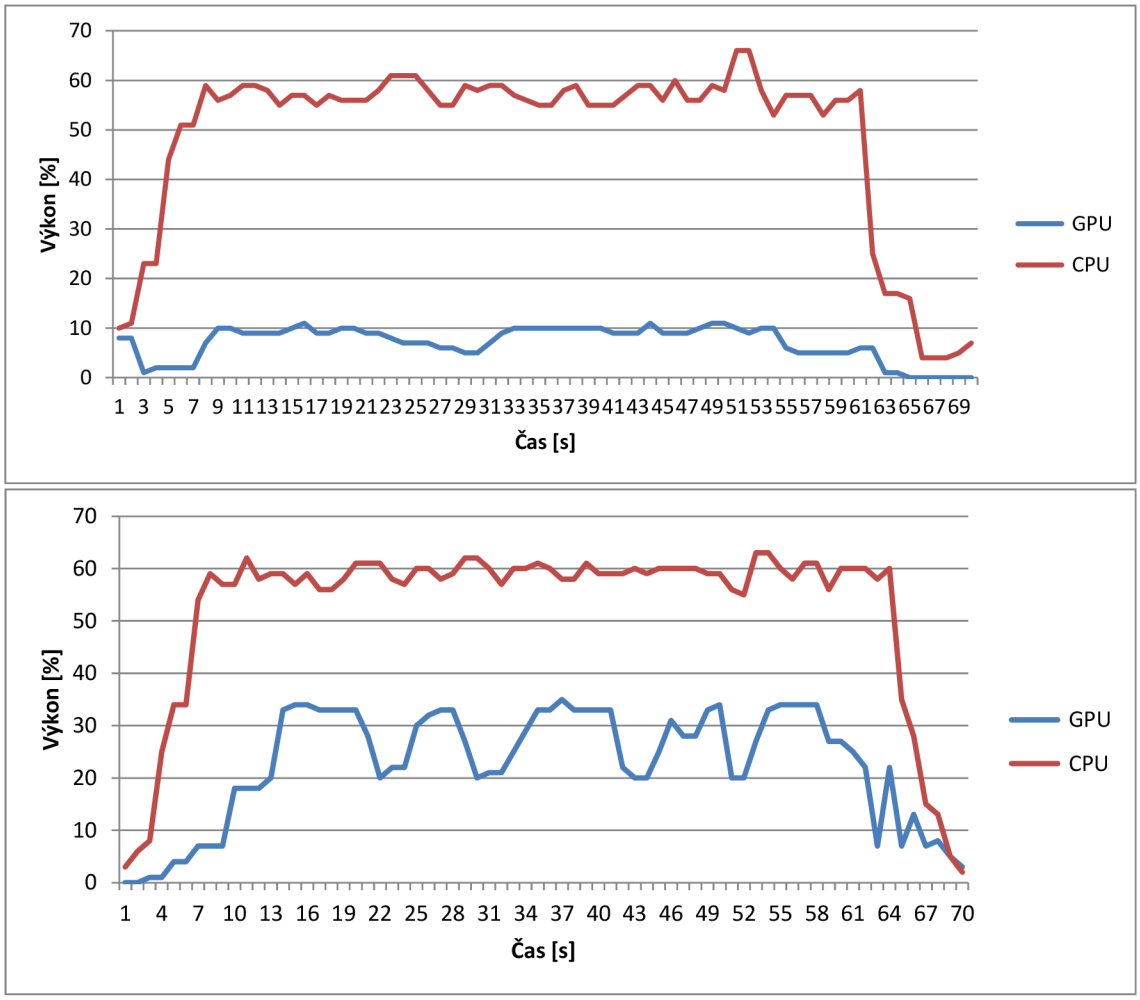
Výkonnostní testy z hlediska praxe slouží k hledání limitů aplikace a jak její běh působí na systém. Nejčastěji jsou měřeny veličiny propustnost, paměť, čas odezvy, výkon CPU. V testech navržených pro tuto diplomovou práci jsou nejdůležitějšími aspekty výkon CPU, výkon GPU, využití paměti RAM a GPU.

5.2.1 Vytížení CPU, GPU a RAM

Pro sledování výkonu CPU, GPU a RAM byly použity funkce pro snímání aktuálního vytížení. Kód znázorňující získávání těchto dat a jejich import do grafu je na ukázce 4.7. Pro načítání hodnot CPU a RAM byly použity funkce třídy *PerformanceCounter*. Pro získávání hodnot GPU byl použit balíček *NvAPIWrapper*. Po spuštění aplikace s nastavenými parametry se začnou načítat získané hodnoty do lokální proměnné. Tyto hodnoty jsou oddělovány středníkem a po poslední hodnotě v řadě je text odřádkován. Dodržením těchto pravidel lze snadno docílit jednoduchého logování. Po stisku tlačítka pro zápis logu se vygeneruje soubor s názvem, který je složen ze zadaných parametrů oddělených podtržítkem a koncovkou csv. Při otevření tohoto souboru v aplikaci Excel od společnosti Microsoft se vytvoří 4 sloupce a odpovídající počet řádků s hodnotami. Od hodnot paměti GPU a RAM byly odečteny počáteční hodnoty paměti, které využíval systém bez spuštěné aplikace. Hodnoty výkonu CPU a GPU byly před spuštěním aplikace v řádu menších jednotek, a proto byly tyto hodnoty ponechány. Testování proběhlo na několika kombinacích parametrů.

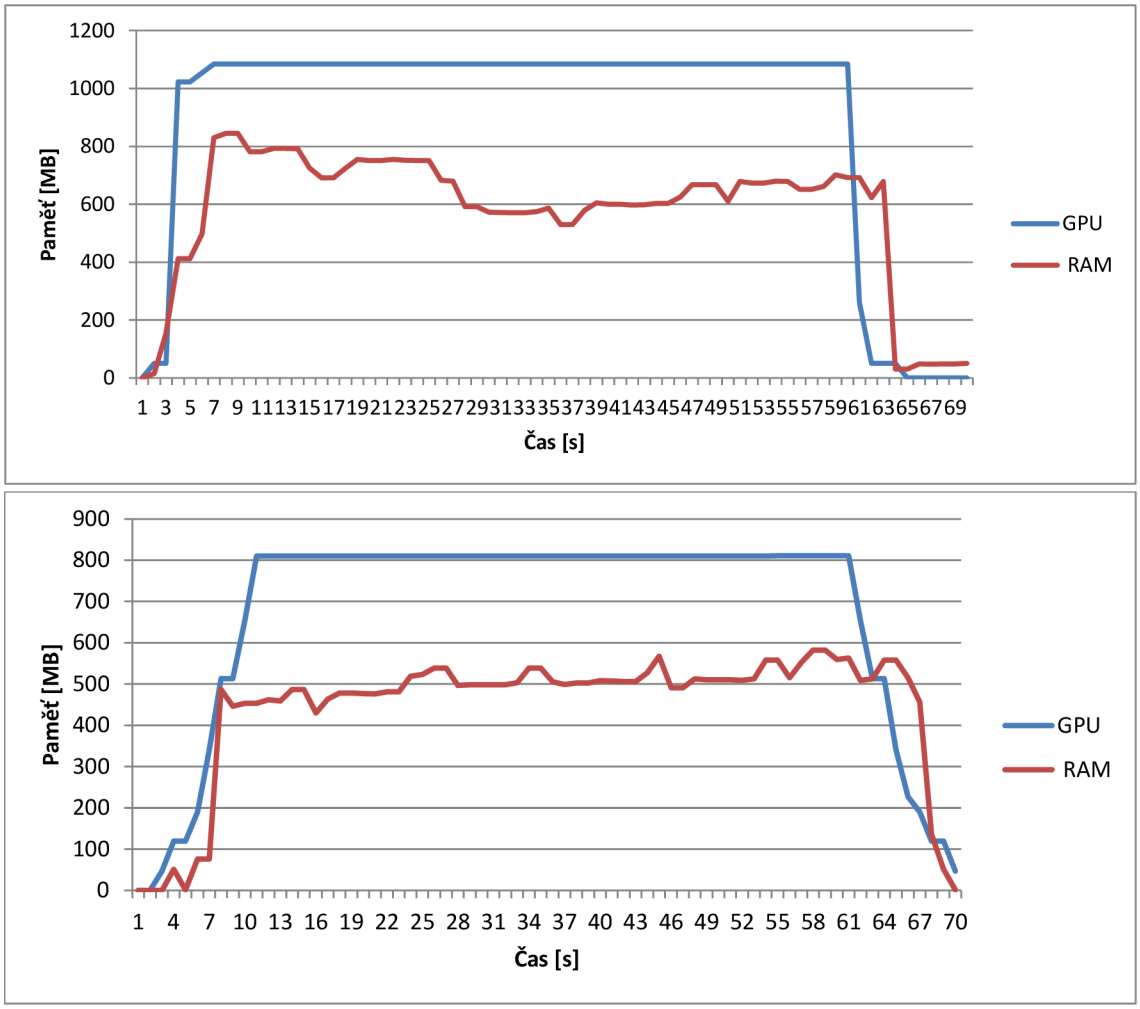
Sledování výkonu systému při enkódování v reálném čase v závislosti na presetu

V tomto testu byl sledován výkon systému při enkódování v závislosti na parametru preset. Tabulka dostupných presetů s jejich popisem je uvedena zde 3.1. V testu byla spouštěna aplikace využívající CUDA a DirectX s kodekem H.264. V závěru bylo provedeno srovnání s výsledky presetu *High Performance* pro kodek HEVC. Na obrázku 5.3 jsou grafy srovnání výkonu systému při enkódování s využitím CUDA (nahoře) a DirectX (dole) pro preset *High Performance*. Grafy pro srovnání využité paměti jsou na obrázku 5.4. Srovnání ostatních presetů s kodekem H.264 jsou v příloze.

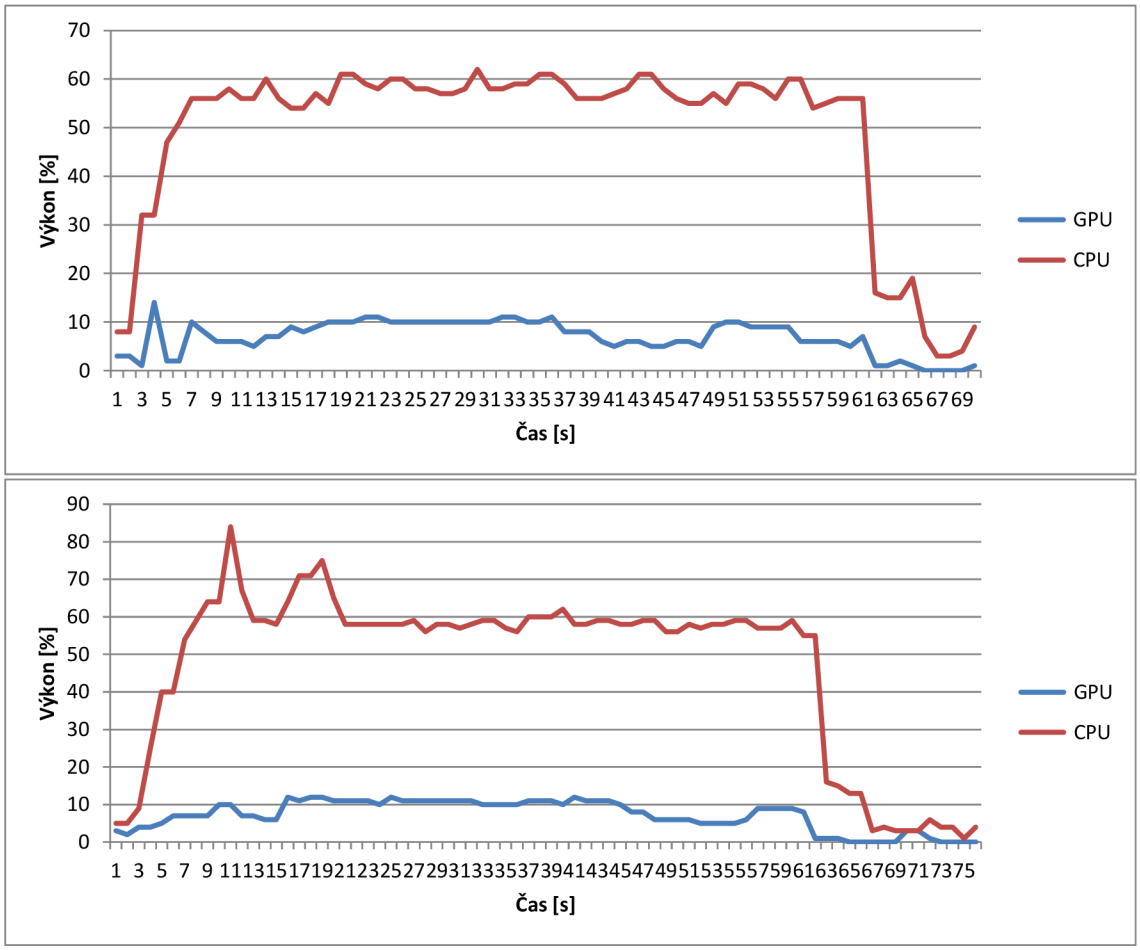


Obr. 5.3: Na obrázku jsou vidět grafy výkonu CPU a GPU při enkódování do H.264. Nahoře je graf pro CUDA a dole graf pro DirectX. Graf je vázán k presetu *High Performance*.

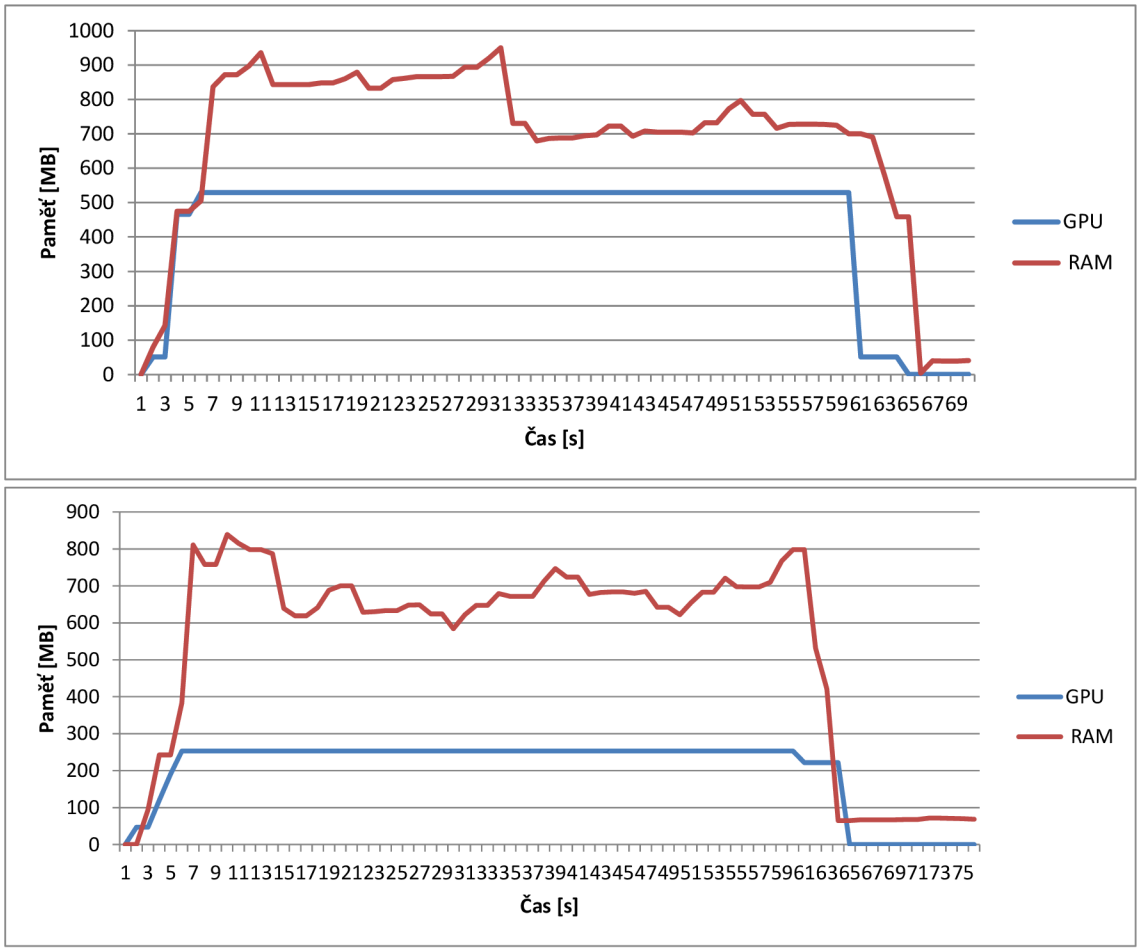
Spotřeba výkonu a paměti na enkódování se v H.264 a HEVC příliš neliší, a proto byly vytvořeny pro kodek HEVC jen další čtyři grafy pro jediný preset *High Performance* a jsou na obrázcích 5.5 a 5.6.



Obr. 5.4: Na obrázku jsou vidět grafy paměti RAM a GPU při enkódování do H.264. Nahoře je graf pro CUDA a dole graf pro DirectX. Graf je vázán k presetu *High Performance*.



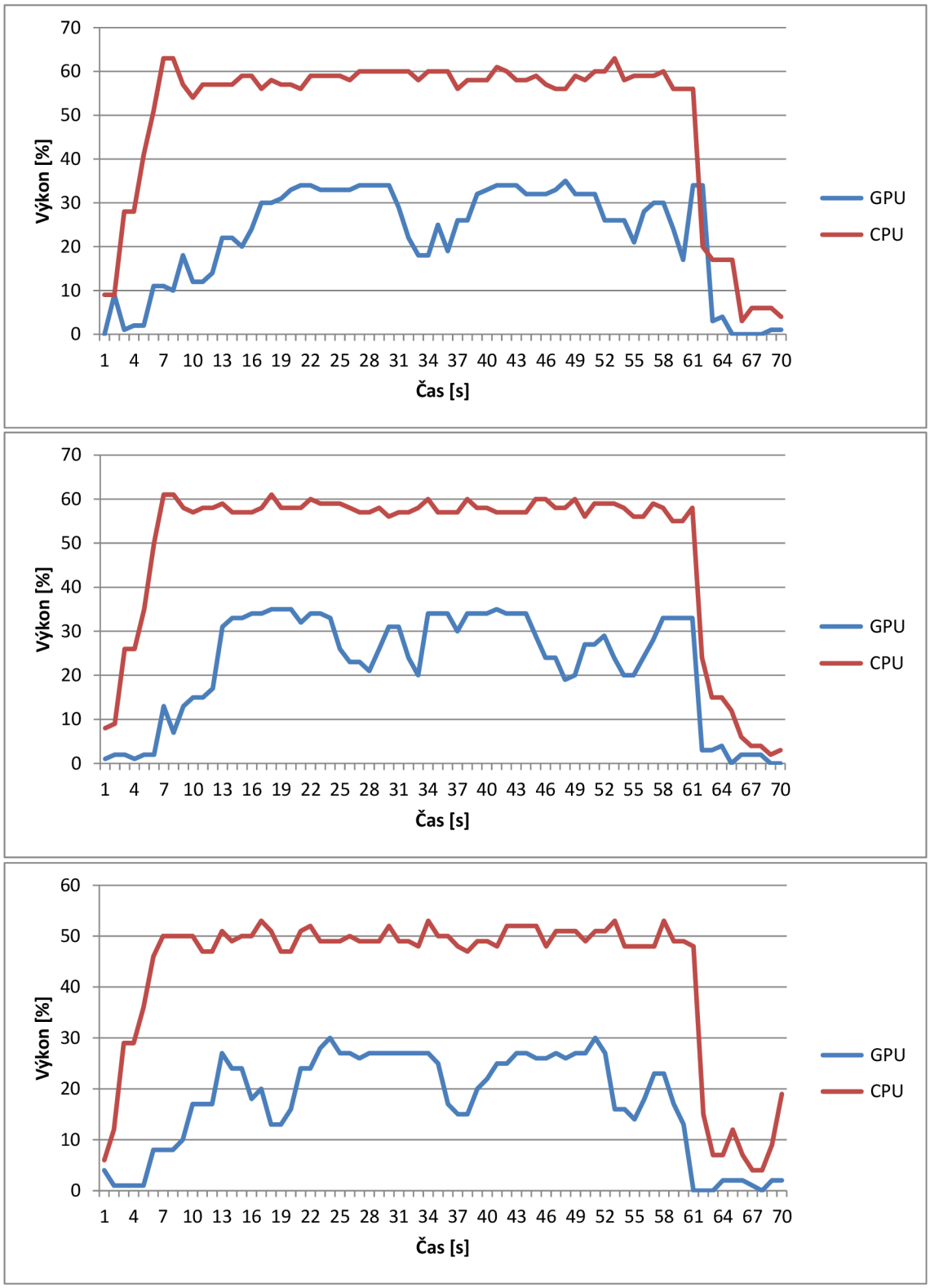
Obr. 5.5: Na obrázku jsou vidět grafy výkonu CPU a GPU při enkódování do HEVC. Nahoře je graf pro CUDA a dole graf pro DirectX. Graf je vázán k presetu *High Performance*.



Obr. 5.6: Na obrázku jsou vidět grafy paměti RAM a GPU při enkódování do HEVC. Nahoře je graf pro CUDA a dole graf pro DirectX. Graf je vázán k presetu *High Performance*.

Sledování výkonu systému při enkódování v reálném čase v závislosti na parametru QP

Parametr QP určuje kvalitu kódování. Čím vyšší hodnota QP, tím je soubor menší a výsledná kvalita je nižší. Pro tento test byly vstupní parametry nastaveny na projekt, který k enkódování využívá DirectX a kodek byl nastaven na H.264. Měřily se tři hodnoty QP: 1, 22 a 50. Na obrázcích jsou seřazeny grafy podle nastaveného QP, kde nahoře je QP = 1, uprostřed QP = 22 a dole QP = 50. V příloze jsou grafy pro srovnání na CUDA i DirectX s nastavenými mezními hodnotami QP 1 a 50.



Obr. 5.7: Na obrázku jsou vidět grafy výkonu CPU a GPU při enkódování v závislosi na parametru QP. Nahoře je graf pro QP = 1, uprostřed QP = 22 a dole QP = 50.

5.2.2 Průměrná doba vyhodnocení části kódu

Pro tento test autor zvolil část kódu, kde se kopírují data snímku do vyrovnávací paměti konkrétního zařízení (DirectX a CUDA) a proces enkódování. Tato část byla vybrána, protože se očekává, že právě zde bude největší možný rozdíl časů zpracování.

K testu byla využita knihovna *std::chrono*[1], která slouží k sledování času s různými mírami přesnosti. Pro zaznamenání času, kdy se část kódu začala provádět, slouží třída *high_resolution_clock*, která reprezentuje hodiny s nejmenší možnou periodou tikání. Pro výpočet doby trvání kódu byla použita knihovna *duration*. Objekt této třídy si udržuje informaci o délce jednoho tiku hodin a počet tiků. Knihovna poskytuje i funkci pro převod tohoto objektu na zvolené časové jednotky typu int. Provedení testu lze vidět na kódu 5.1.

```
1  vector<float> values;
2  while (!pNvencThreadParams->terminated()){
3          ...
4          auto start = high_resolution_clock::now();
5          // měřená část kódu
6          auto stop = high_resolution_clock::now();
7          auto duration = duration_cast<microseconds>(stop - start);
8          if (nFrame >= 0 && nFrame <= 500)
9              values.push_back(duration.count());
10         ...
11 }
12 float sum = 0, mean = 0, variance = 0, standardDeviation = 0;
13 for (int i = 0; i < values.size(); ++i)
14 {
15     sum += values[i];
16 }
17 mean = sum / values.size();
18 for (i = 0; i < values.size(); ++i)
19     standardDeviation += pow(values[i] - mean, 2);
```

Výpis 5.1: Kód znázorňující test na výpočet doby provádění kódu.

[1]Dokumentace ke knihovně je dostupná na adrese: https://en.cppreference.com/w/cpp/chrono

Projekt	CUDA				DirectX			
Kodek	HEVC		H.264		HEVC		H.264	
Kamera	1	2	1	2	1	2	1	2
FPS	7	7	7	7	7	7	7	7
Měřená doba	8006 μs	11727 μs	8059 μs	12519 μs	4970 μs	6188 μs	4418 μs	6672 μs

Tab. 5.5: V tabulce jsou uvedeny výsledky testu měření průměrné doby vyhodnocení části kódu.

Projekt	CUDA				DirectX			
Kodek	HEVC		H.264		HEVC		H.264	
Kamera	1	2	1	2	1	2	1	2
FPS	7	7	7	7	7	7	7	7
Průměrná doba	10746 μs	10778 μs	10876 μs	10747 μs	7235 μs	6936 μs	7309 μs	6946 μs
Směrodatná odchylka	4202 μs	3412 μs	4314 μs	4316 μs	4220 μs	4241 μs	4388 μs	4671 μs

Tab. 5.6: V tabulce jsou uvedeny výsledky retestu měření průměrné doby vyhodnocení části kódu.

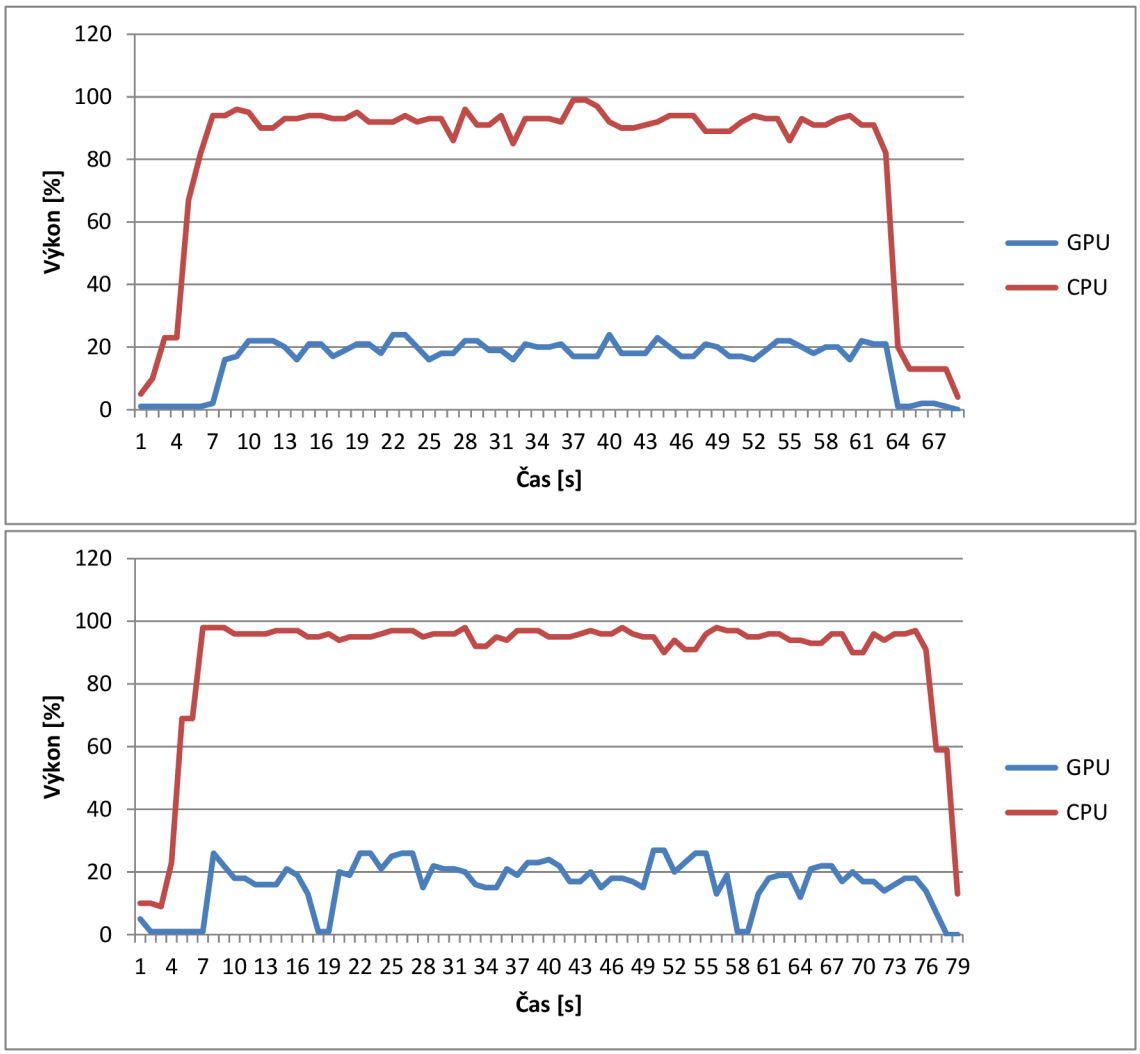
Pro tento test byl stanoven počet vzorků na hodnotu 500, ze které se udělá aritmetický průměr. Vzhledem k tomu, že měřená část kódu je uvnitř cyklu, který se provádí pro každý snímek, není problém takové množství vzorků zaznamenat. Každá měřená doba se ukládá do vektoru. Na konci cyklu je potřeba spočítat výslednou průměrnou dobu měřeného kódu. K tomu slouží další cyklus, ve kterém se sečtou všechny uložené hodnoty a výsledná doba se vydělí počtem vzorků. Výsledky tohoto měření jsou vidět v tabulce 5.5.

V tabulce lze vidět rozdíly hodnot mezi kamerou 1 a kamerou 2. Při enkódování paralelně na dvou kamerách by měl být však čas zpracování jednoho rámce na obou kamerách podobný. Po hlubším zkoumání bylo zjištěno, že jedna kamera byla rozostřená a tudíž doba při kódování byla menší. Proto bylo provedeno retestování na dvou zaostřených kamerách zachycujících téměř totožnou scénu. K hodnotám byla vypočítána směrodatná odchylka. Výsledky jsou zachyceny v tabulce 5.6.

5.2.3 Počet připojených kamer

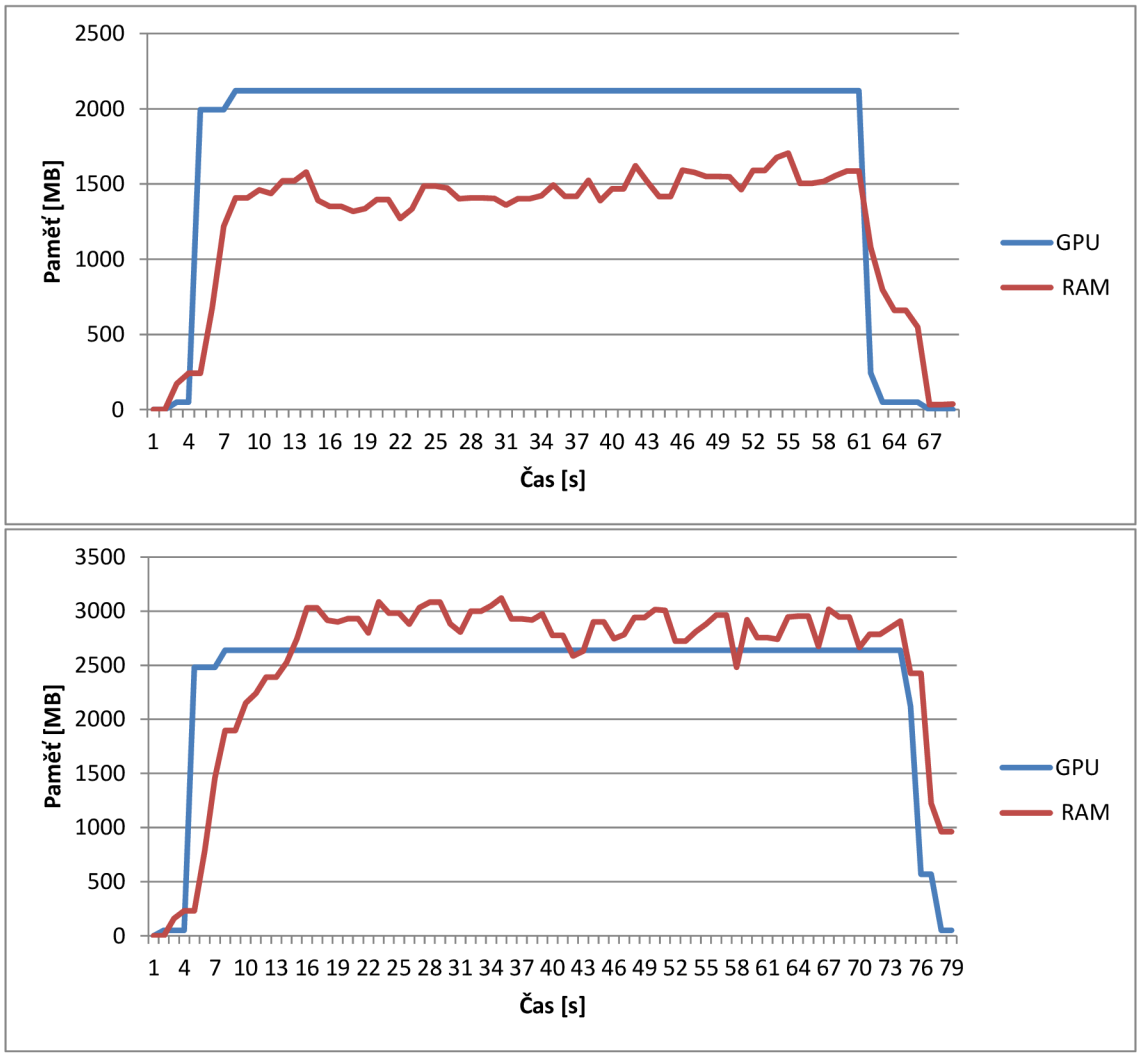
V následujícím testu bude měřen počet kamer, ze kterých dokáže enkodér v reálném čase kódovat snímky. Poté se v tomto testu budou nastavovat rámce za sekundu na hodnotu, při které zvládne enkódovat snímky z šesti kamer bez ztrát. Tento test byl spouštěn na projektu, který k enkódování využívá CUDA s grafickou kartou č. 2. Parametry aplikace byly QP = 22, fps = 7.0 a preset byl nastaven na *High Performance*. Test probíhal tak, že se postupně připojovaly kamery a zjišťovalo se

zatížení systému. V předchozích testech bylo zjištěno, že dvě kamery zvládne enkodér bez problému, a proto byl počáteční počet kamer tři. Grafy pro výkon systému se třemi a šesti kamerami jsou vidět na obrázcích v příloze. Grafy výkonu systému mezi čtyřmi a pěti kamerami je na obrázku 5.8. Grafy pro srovnání využité paměti jsou na obrázku 5.9.



Obr. 5.8: Na obrázku jsou vidět grafy výkonu CPU a GPU při enkódování s využitím CUDA do kodeku H.264. Nahoře je graf pro zapojené čtyři kamery a dole graf pro pět kamer. Graf je vázán k presetu *High Performance*.

Nyní bude proveden test pro průměrně zvládané rámce za sekundu při připojení šesti kamer. Grafy znázorňující zátěž systému při nastavení dvou a tří rámců za sekundu jsou zobrazeny v příloze.

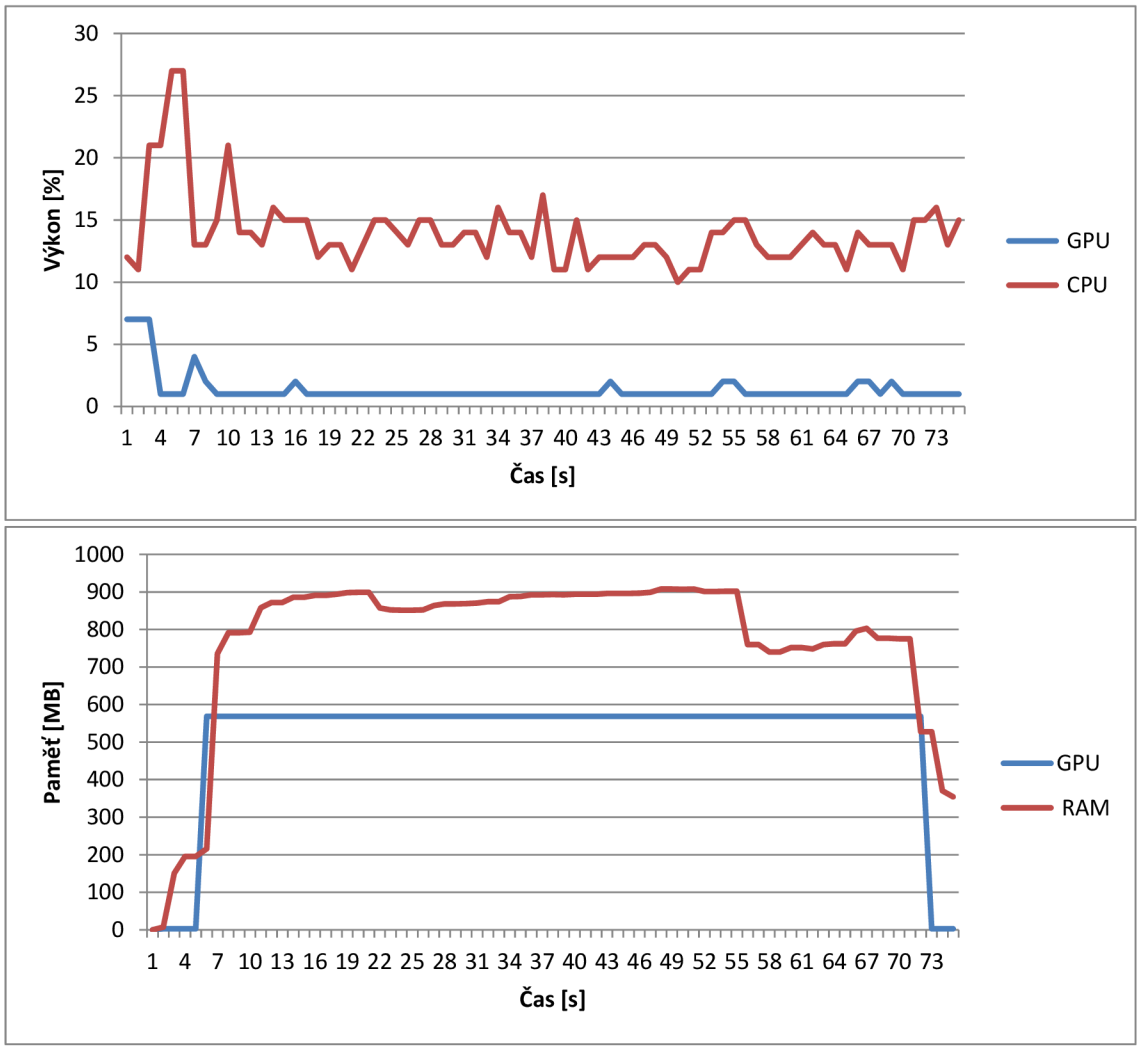


Obr. 5.9: Na obrázku jsou vidět grafy paměti RAM a GPU při enkódování s využitím CUDA do kodeku H.264. Nahoře je graf pro zapojené čtyři kamery a dole graf pro pět kamer. Graf je vázán k presetu *High Performance*.

5.2.4 Srovnání zátěže systému při enkódování aplikací FFmepg

Test se zaměřuje na měření zátěže systému při enkódování aplikací FFmpeg[2]. Aplikace poskytuje enkódování snímků pomocí čipu NVENC z kamery v reálném čase. Možnost použít kódování přes aplikaci FFmpeg je možné i z grafického rozhraní výsledné aplikace. U této volby je možné měnit kodek, fps a délku kódovaného segmentu. Na obrázku 5.10 lze vidět grafy zátěže systému při enkódování jedné kamery.

[2]Dokumentace je k dispozici na adrese: https://ffmpeg.org/documentation.html



Obr. 5.10: Na obrázku jsou vidět grafy zátěže systému při enkódování s využitím aplikace FFmpeg do kodeku H.264 se zapojenou jednou kamerou. Nahoře je graf pro vytížení CPU a GPU a dole graf alokace paměti RAM a GPU.

5.3 Kvalitativní testy

Mezi kvalitativní testy při enkódování obrazu jsou nejpoužívanější metody měření PSNR[3] a SSIM[4]. Pro tuto práci byla zvolena metoda měření PSNR.

5.3.1 Komprimační ztráty

Pro test na komprimační ztráty obrazu byla zvolena metoda měření PSNR. V této metodě se porovnává rámec zdrojového (nekomprimovaného) snímku se stejným již enkódovaným snímkem. Pro tento výpočet je možné využít různé programy, ale

[3]Peak signal-to-noise ratio
[4]Structural similarity index measure

v rámci tohoto testu autor vytvořil funkci pro vyhodnocení PSNR v C++ s použitím knihovny OpenCV. Funkce pro výpočet je na ukázce kódu 5.2.

```
1  double getPSNR(const Mat& source, const Mat& test)
2  {
3      Mat result;
4      absdiff(source, test, result);
5      result.convertTo(result, CV_32F);
6      result = result.mul(result);
7
8      Scalar scalar = sum(result);
9
10     double sse = scalar.val[0] + scalar.val[1] + scalar.val[2];
11
12     if (sse <= 1e-10)
13         return 0;
14     else
15     {
16         double mse = sse/(double)(source.channels()*source.total());
17         double psnr = 10.0 * log10((255 * 255) / mse);
18         return psnr;
19     }
20 }
```

Výpis 5.2: Kód znázorňující funkci pro výpočet PSNR.

Testován byl projekt DirectX, kde se měnila hodnota QP, která udává úroveň kvality kódování. Výsledná hodnota PSNR je v jednotkách dB. Typické hodnoty jsou od 30dB do 50dB, kde vyšší hodnota znamená lepší výsledek. V tabulce 5.7 jsou zaznamenané hodnoty testování.

5.4 Diskuse výsledků

V rámci výkonnostních testů se měřil výkon systému při enkódování v závislosti na nastavovaných parametrech enkódování. Při testu měření výkonu při změnách presetů lze vidět, že není podstatný rozdíl v nastavení presetu na výkonu CPU, ten je ve všech měření téměř na stejné hodnotě okolo 60 %. Velké rozdíly jsou naopak při srovnání výkonu GPU při enkódování využívající CUDA a DirectX. Vytížení GPU je při kódování s pomocí CUDA stále na hodnotě blížící se 10 %, ale při použití DirectX kolísá mezi 20 % a 30 %. To je spojeno s enkódováním do kodeku H.264, protože při HEVC je výkon GPU ustálen také na 10 %.

Projekt	DirectX	
Kodek	HEVC	H.264
QP = 1	39.05 dB	37.12 dB
QP = 15	36.12 dB	35.82 dB
QP = 22	35.17 dB	33.72 dB
QP = 30	34.49 dB	33.83 dB
QP = 40	33.43 dB	33.45 dB
QP = 50	32.82 dB	32.80 dB

Tab. 5.7: V tabulce jsou uvedeny výsledky měření hodnoty PSNR v závislosti na hodnotě QP.

Z hlediska alokace paměti je na grafech vidět, že si pro různé presety enkodér alokuje stejné množství paměti. Při porovnání enkódování pomocí CUDA a DirectX je znatelné, že pro DirectX si alokuje GPU méně paměti než pro CUDA. Hodnota pro jednu kameru při enkódování CUDA je 550 MB a pro DirectX 400 MB. Při enkódování s kodekem HEVC se alokovaná velikost paměti GPU zmenšuje při použití DirectX na cca 270 MB pro jednu kameru a 125 MB pro CUDA. Alokace paměti RAM nabývá hodnot od 500 MB do 1000 MB pro dvě kamery, kde nejnižší hodnotu využívá enkódování při použití DirectX do H.264 s presetem *High Performance* a nejvyšší hodnoty dosahuje enkódování s použitím CUDA do H.264 s presetem *Lossless High Quality*.

V dalším testu byl měřen výkon systému v závislosti na parametru QP. Z grafů výsledků lze vidět, že výkon CPU a GPU při testu na QP = 1 a QP = 22 dosahoval podobných hodnot. Při nastavení parametru QP na hodnotu 50 se vytížení CPU a GPU snížilo cca o 10 %. Z hlediska alokace paměti je z grafů viditelné, že změna parametru QP nemá vliv na alokaci paměti GPU ani RAM. Při porovnání grafů pro kodek HEVC mezi DirectX a CUDA byl rozdíl v paměti, který odpovídá alokaci paměti v předchozím testu (alokovaná pamět odpovídá alokaci pro jednu kameru). Při srovnání grafů výkonu se při enkódování za pomocí CUDA snižuje procento využití GPU na cca 10 %, zatímco při využití DirectX kolísá výkon mezi 20 % a 35 %.

V rámci dalšího testu byl měřen čas provedení části kódu, který kopíroval rámec do vyrovnávací paměti zařízení a poté provedl enkódování. Tento test byl proveden pro porovnání mezi CUDA a DirectX. Při prvním testování vycházely různé hodnoty pro první a druhou kameru, protože první kamera byla rozostřená. Proto bylo provedeno testování znovu na dvou kamerách, které zachytávaly podobnou scénu. Při druhém testování byla ještě dopočítávána směrodatná odchylka. Z tabulky 5.6

je viditelné, že měřená část kódu se prováděla rychleji při enkódování s využitím DirectX.

Další test se zabýval měřením výkonu systému při enkódování s různým počtem připojených kamer. Kvůli tomuto testu musela být vyměněna grafická karta, která dokáže zpracovávat více kamer. Úkolem testu bylo najít hraniční počet připojených kamer, které dokáže enkodér paralelně zpracovávat při nastavení 7 rámců za sekundu. Bylo zjištěno, že největší slabinou hardwarové konfigurace je procesor, který při připojených pěti kamerách dosahoval 100 %, takže se začaly snímky z fronty zahazovat. Výsledkem testu je, že maximální počet kamer, které hardwarová konfigurace dokáže zpracovávat v reálném čase při nastavení 7 rámců za sekundu je čtyři.

Následoval test na průměrně zvládané rámce za sekundu pro enkódování snímků ze šesti kamer zároveň. Při tomto testu bylo provedeno spuštění aplikace s hodnotami parametru fps = 2 a fps = 3. Bylo zjištěno, že při nastavení parametru fps = 3 nedokázaly na použité hardwarové konfiguraci enkódovat všechny snímky a v průběhu enkódování byly zahazovány. Při hodnotě parametru fps = 2 už dokázal enkodér zpracovávat všechny rámce ze šesti kamer.

Další provedené testování porovnávalo enkódování pomocí čipu NVENC s použitím aplikace FFmpeg a pomocí Video Codec SDK. Bylo zjištěno, že při použití aplikace FFmpeg dosahují hodnoty výkonu CPU nižších hodnot než u implementované aplikace. Hodnota využití výkonu GPU je téměř shodná s implementovanou aplikací při použití CUDA. Z hlediska alokace paměti GPU si FFmpeg alokuje pro jednu kameru cca 580 MB. To je v porovnání s alokací paměti GPU při použití DirectX mnohem více, ale od CUDA se výrazně neliší.

Poslední test byl zaměřen na kvalitu kódování. Pro kontrolu kvality byla použita metoda měření hodnoty PSNR. Aplikace byla spouštěna s různými hdonotami parametru QP, který určuje kvalitu enkódování. Dobré hodnoty PSNR jsou v rozmezí 31-37 dB a výborné hodnoty jsou vyšší než 37 dB [22]. Výsledky jsou zobrazeny v tabulce 5.7. Jak lze v tabulce vidět, tak v celém rozsahu parametru QP se hodnota PSNR nedostala mimo rozmezí hodnot PSNR označovaných jako dobrý výsledek. V tabulce jde vidět, že hodnota PSNR je nepřímo úměrná parametru QP a čím vyšší je QP, tím je horší kvalita enkódování.

Závěr

Cílem práce bylo vytvořit aplikaci, která bude s využitím čipu NVENC a Video Codec SDK v reálném čase komprimovat data ze 4K kamer. V rámci diplomové práce autor definoval video kodeky, nejpoužívanější barevné prostory a detailně popsal komprimační standardy H.264 a HEVC. Zmínil role standardů a jejich rozdíly. V teoretické části jsou také vysvětleny pojmy GPU, GPGPU a CUDA. Byly zde také popsány stěžejní technologie Video Codec SDK a NVENC.

Součástí práce je analýza již naimplementovaných projektů Video Codec SDK, které využívají k enkódování DirectX, CUDA a OpenGL. Jsou zde popsány důležité funkce a metody jednotlivých projektů a vysvětlen princip zpracování snímků při komprimaci. Projekty byly otestovány a výsledky byly zpracovány a porovnány. Následně autor navrhl aplikaci pro kódování v reálném čase.

Praktická část práce je implementována v programovacím jazyku C++ a využívá technologie WINAPI, OpenCV a mvImpact Acquire SDK. Jsou zde popsány nejdůležitější části implementace aplikace. Autor popsal rozdělování procesů do více vláken za použití technologie WINAPI. Následně je vysvětleno načítání snímků z 4K kamer a jejich ukládání do vyrovnávací paměti. Důležitá část popisu implementace se věnuje procesu enkódování. Aplikace byla úspěšně implementována a nad rámec diplomové práce bylo naimplementováno grafické rozhraní pro spouštění výsledných aplikací a monitorování zátěže enkódování na systém. Grafické rozhraní bylo implementováno v programovacím jazyce C♯.

Pro výslednou aplikaci byly provedeny výkonnostní a kvalitativní testy. V rámci výkonnostních testů bylo provedeno měření výkonu systému při enkódování v zavislosti na nastavovaných parametrech. Data získaných výsledků byla ve všech testech zpracována do dvou grafů. První graf vždy zobrazuje zátěž komprimace na hodnoty CPU a GPU. Druhý graf znázorňuje alokaci paměti RAM a GPU.

První výkonnostní test byl zaměřen na volbu presetu, který definuje pravidla pro komprimaci. Test byl proveden na čtyřech presetech. Další test měřil výkon systému v závislosti na paramtru QP, který značí kvalitu enkódování. Autor provedl testování na průměrnou dobu vyhodnocení části kódu při enkódování na DirectX a CUDA. Pro test experimentující s počtem připojených kamer byla vyměněna grafická karta v hardwarové sestavě a na základě výsledků byly stanoveny limity pro tuto konfiguraci. Naimplementovaná aplikace byla porovnána s aplikací FFmpeg. V kvalitativních testech bylo provedeno měření komprimačních ztrát metodou výpočtu PSNR zdrojové a enkódované videosekvence.

Literatura

[1] *Richardson, I.E. The H.264 Advance Video Compression Standard. Second Edition.* Chichester, UK: John Wiley & Sons, Ltd, 2010. ISBN 9780470516928.

[2] Wikipedia, The Free Encyclopedia YCbCr [online] Dostupné z URL http://en.wikipedia.org/wiki/YCbCr, [cit. 2018-12-01]

[3] Poynton, Charles A. *Digital Video and HDTV: Algorithms and Interfaces. Second Edition.* Morgan Kaufmann, 2012. ISBN: 978-0-12-391926-7

[4] A. Ansari, M & Khan, Imran. (2015). *Performance analysis and evaluation of proposed algorithm for advance options of H.263 and H.264 video codec.*, 371-376. 10.1109/RDCAPE.2015.7281427.

[5] Igarta, M. *A STUDY OF MPEG-2 AND H.264 VIDEO CODING* [online] Dostupné z URL https://engineering.purdue.edu/ ace/thesis/igarta/thesis-igarta.pdf, [cit. 2018-11-28]

[6] Béatrice Pesquet-Popescu, Marco Cagnazzo, Frédéric Dufaux, *Academic Press Library in Signal Processing* Elsevier, Volume 5, 2014, Pages 27-92, ISSN 2351-9819, ISBN 9780124201491

[7] Richardson, I.E. *H.264 and MPEG-4 Video Compression*, Chichester, UK: John Wiley & Sons, Ltd, 2003. ISBN 0470848375

[8] Igarta, M. *A STUDY OF MPEG-2 AND H.264 VIDEO CODING* [online] Dostupné z URL https://engineering.purdue.edu/ ace/thesis/igarta/thesis-igarta.pdf, [cit. 2018-11-19]

[9] Gary J. Sullivan *IEEE Transactions on Circuits and Systems for Video Technology* IEEE Press Piscataway, NJ, USA, ISSN: 1051-8215 doi

[10] G. J. Sullivan, J.-R. Ohm, W.-J. Han, T. Wiegand, *Overview of the high efficiency video coding (HEVC) standard.* , IEEE Trans. Circuits Syst. Video Technol., vol. 22, pp. 1648-1667, Dec. 2012.

[11] AKRAMULLAH, S, *Digital video concepts, methods, and metrics: quality, compression, performance, and power trade-off analysis.* 2013, 344 s.

[12] Ana Rodrigues. (2016-06-09). *H.264 vs H.265 — A technical comparison. When will H.265 dominate the market?* [online] Dostupné z URL cit. 2018-11-15]

[13] Margaret Rouse. (2019-08). *GPU (graphics processing unit)* [online] Dostupné z URL https://searchvirtualdesktop.techtarget.com/definition/GPU-graphics-processing-unit, [cit. 2020-04-13]

[14] AntenehAyele, Eskinder & B. Dhok, S. (2012). *Review of Proposed High Efficiency Video Coding (HEVC) Standard.* International Journal of Computer Applications. 59. 1-9. 10.5120/9621-4265.

[15] Oancea, Bogdan Andrei, Tudorel Dragoescu, Raluca. (2014). *GPGPU Computing.* Challenges of the Knowledge Society. 2.

[16] NVIDIA Corporation *NVIDIA VIDEO CODEC SDK* [online] Dostupné z URL cit. 2018-12-12]

[17] Očkay, Miloš Harakal, Marcel Líška, Miroslav. (2008). *Compute Unified Device Architecture (CUDA) GPU programming model and possible integration to the parallel environment.* 3. 64-68.

[18] NVIDIA Corporation, *CUDA C++ PROGRAMMING GUIDE* [online] Dostupné z URL cit. 2019-04-08]

[19] NVIDIA Corporation *NVIDIA VIDEO DECODER (NVCUVID) INTERFACE* [online] Dostupné z URL cit. 2020-08-04]

[20] NVIDIA Corporation *Video Encode and Decode GPU Support Matrix* [online] Dostupné z URL cit. 2018-12-12]

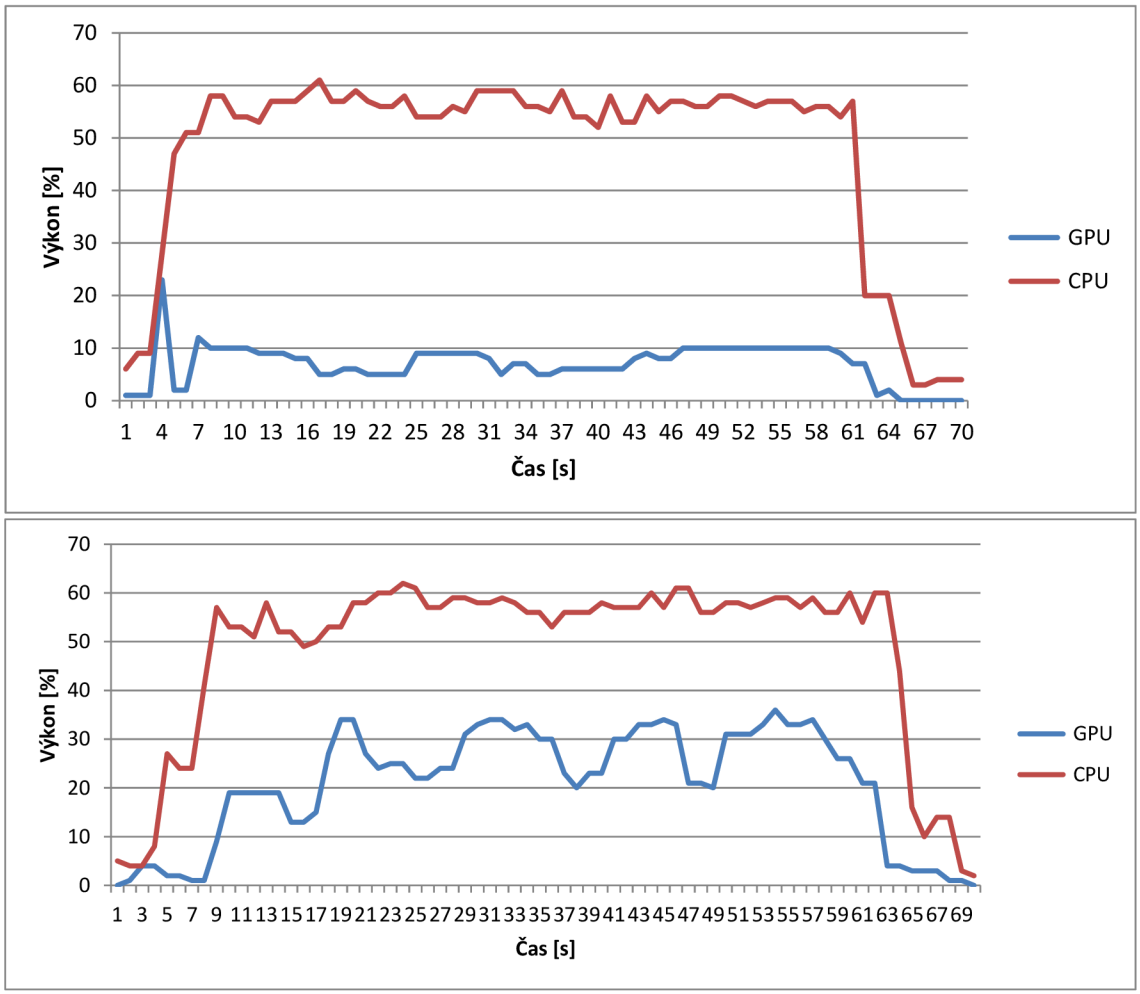
[21] NVIDIA Corporation, *NVENC - NVIDIA HARDWARE VIDEO ENCODER* [online] Dostupné z URL cit. 2018-12-12]

[22] Klaue J., Rathke B., Wolisz A. (2003) EvalVid – A Framework for Video Transmission and Quality Evaluation. In: Kemper P., Sanders W.H. (eds) Computer Performance Evaluation. Modelling Techniques and Tools. TOOLS 2003. Lecture Notes in Computer Science, vol 2794. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-45232-4$_1$6

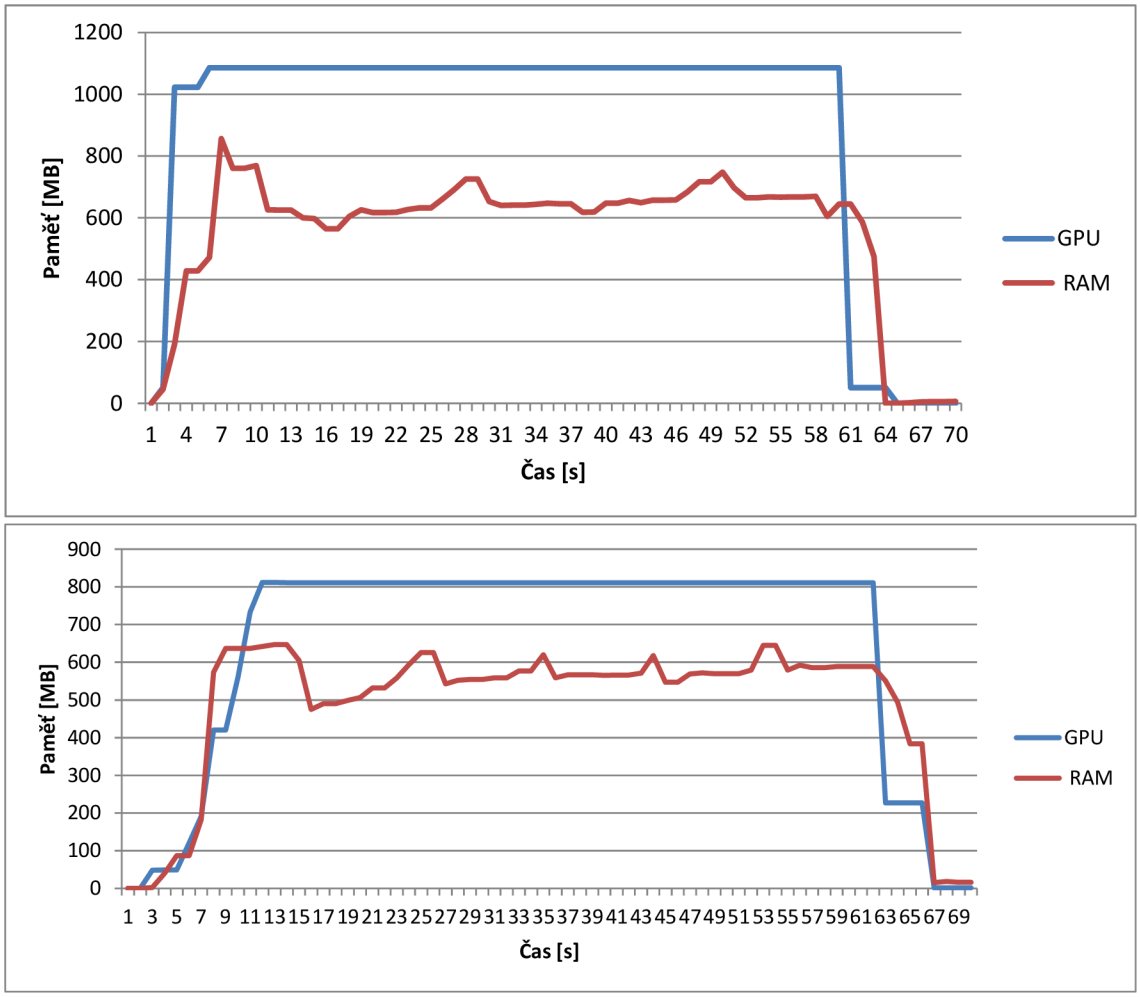
Seznam příloh

A Grafy výsledků testování

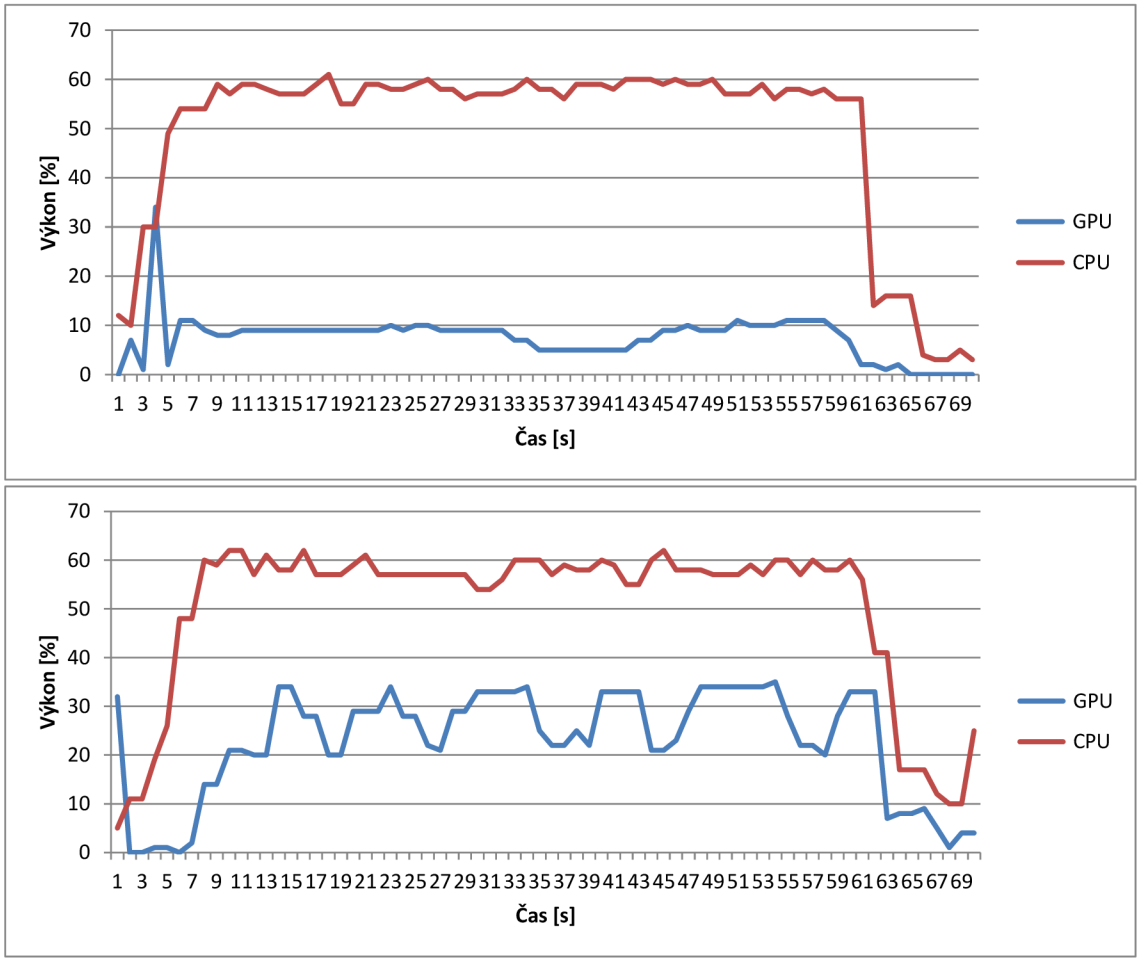
A.1 Grafy vytížení CPU, GPU a RAM v závislosti na presetu



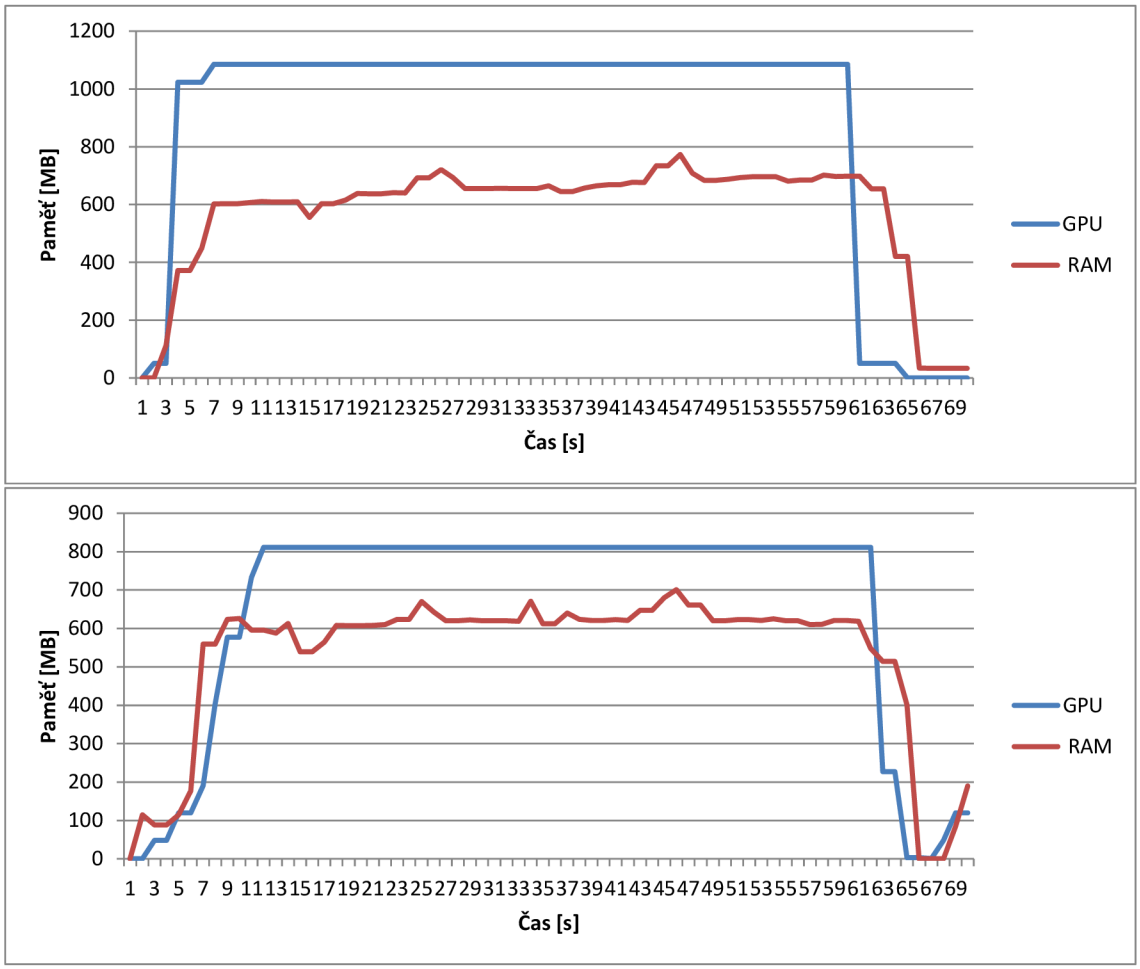
Obr. A.1: Na obrázku jsou vidět grafy výkonu CPU a GPU při enkódování. Nahoře je graf pro CUDA a dole graf pro DirectX. Graf je vázán k presetu *High Quality*.



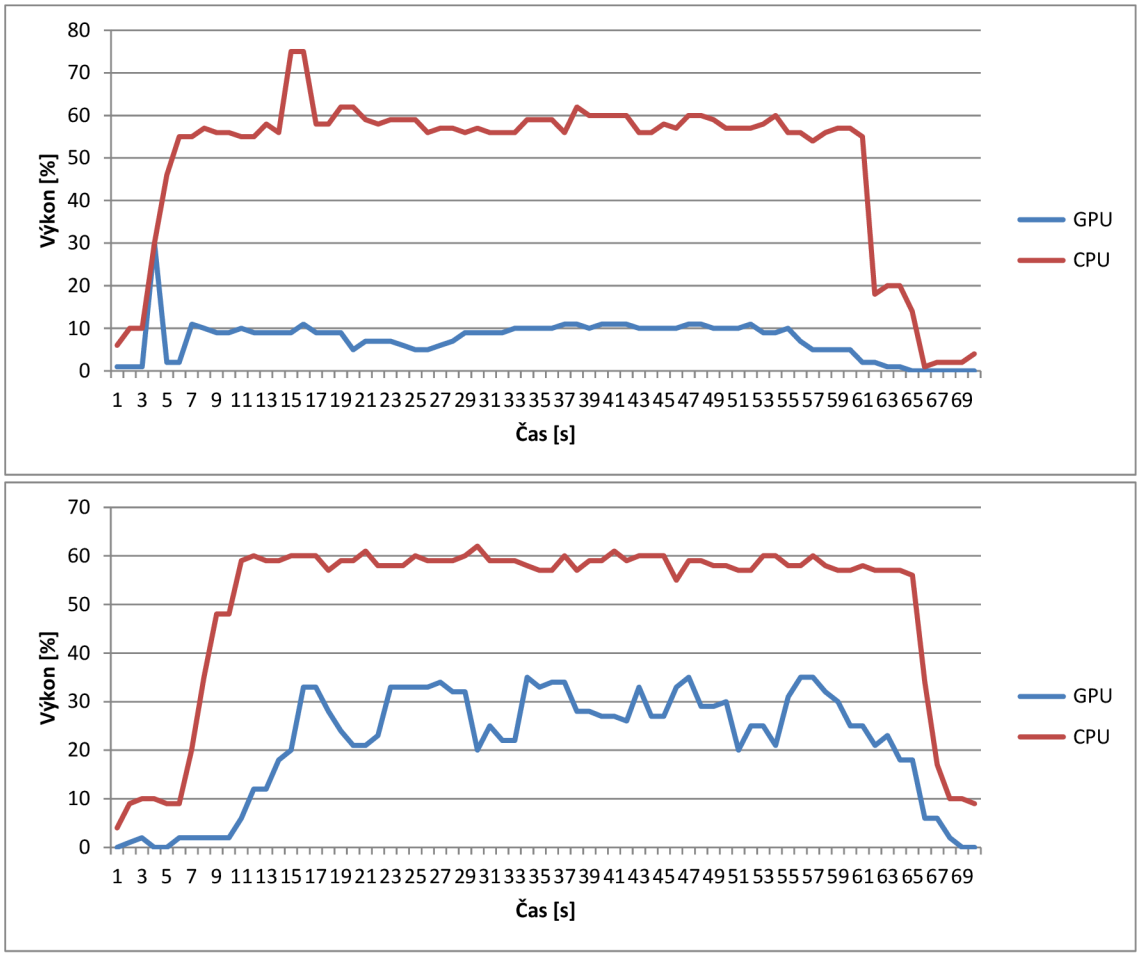
Obr. A.2: Na obrázku jsou vidět grafy paměti RAM a GPU při enkódování. Nahoře je graf pro CUDA a dole graf pro DirectX. Graf je vázán k presetu *High Quality*.



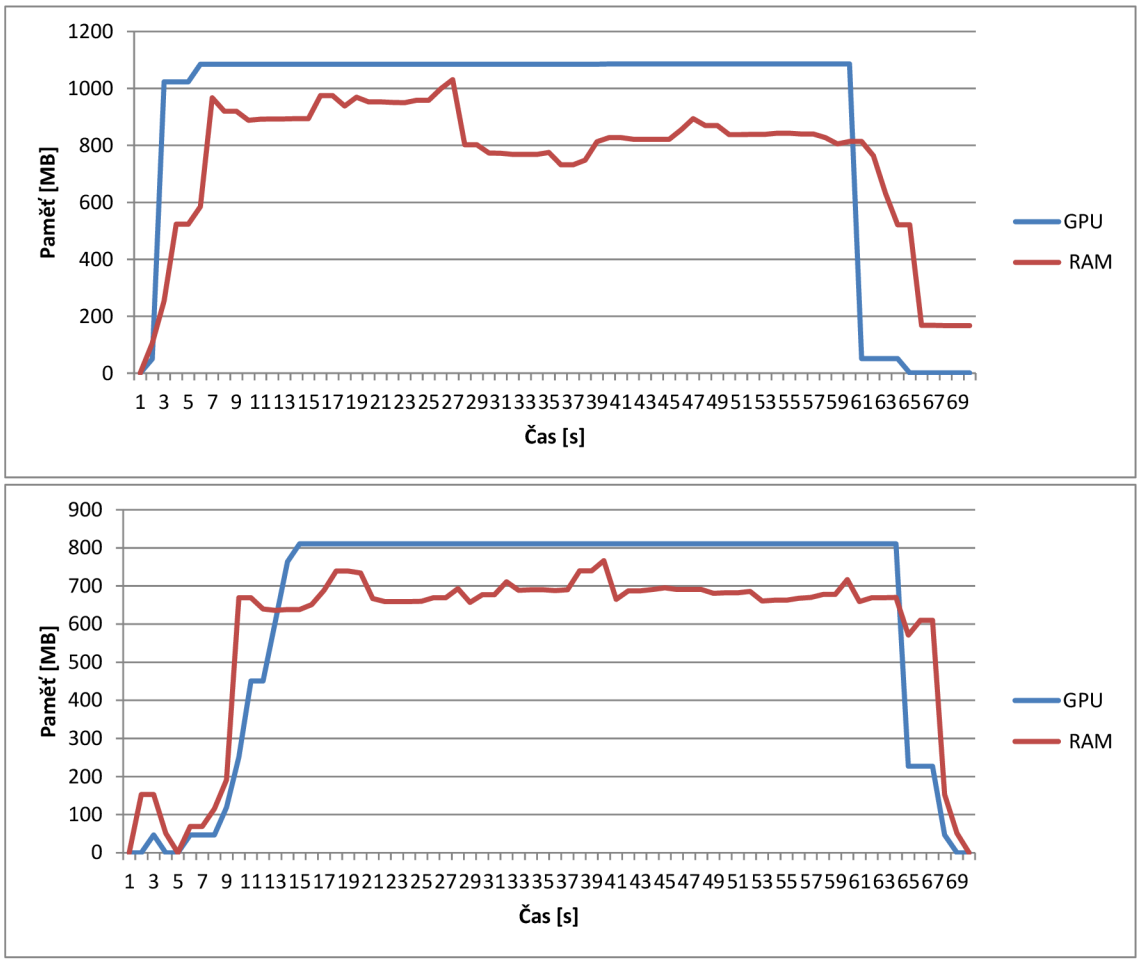
Obr. A.3: Na obrázku jsou vidět grafy výkonu CPU a GPU při enkódování. Nahoře je graf pro CUDA a dole graf pro DirectX. Graf je vázán k presetu *Lossless High Performance*.



Obr. A.4: Na obrázku jsou vidět grafy paměti RAM a GPU při enkódování. Nahoře je graf pro CUDA a dole graf pro DirectX. Graf je vázán k presetu *Lossless High Performance*.

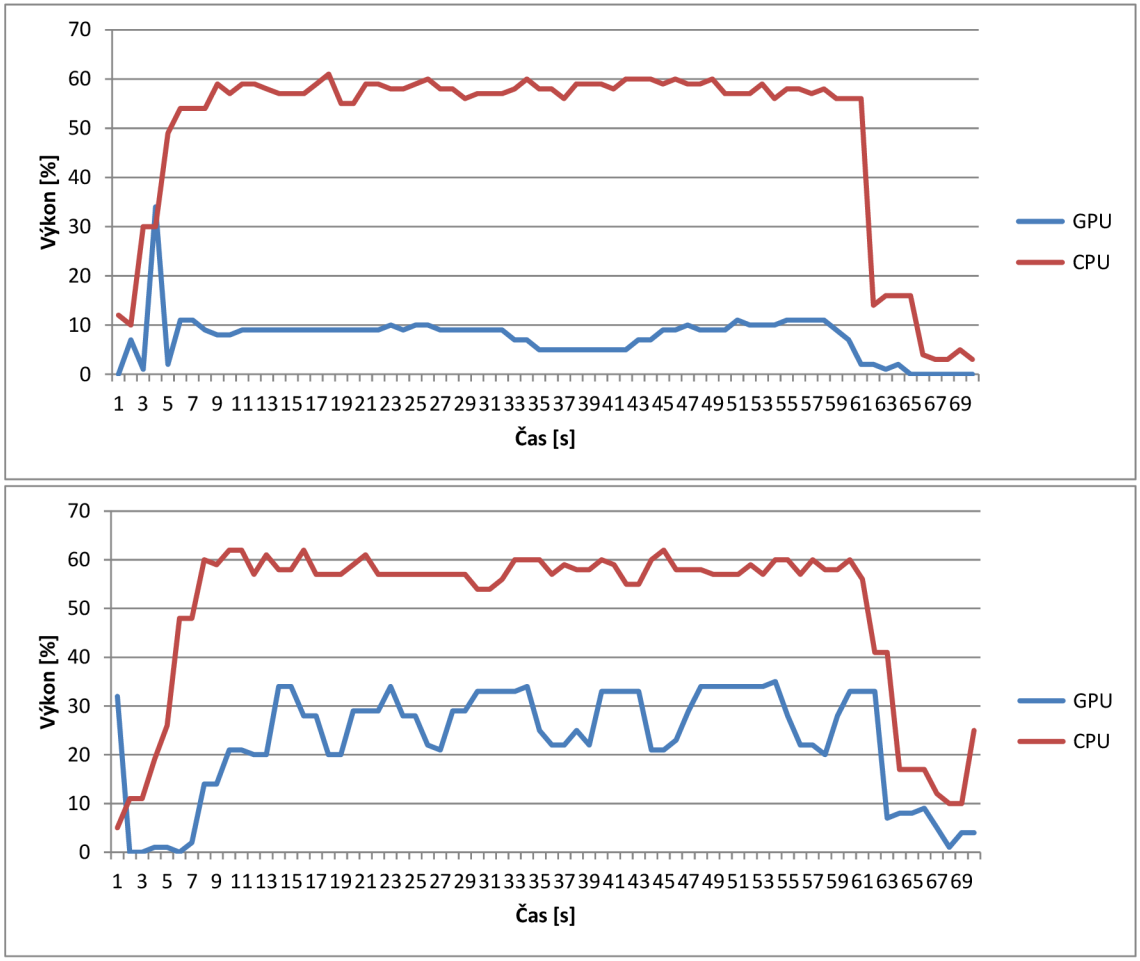


Obr. A.5: Na obrázku jsou vidět grafy výkonu CPU a GPU při enkódování. Nahoře je graf pro CUDA a dole graf pro DirectX. Graf je vázán k presetu *Lossless High Quality*.

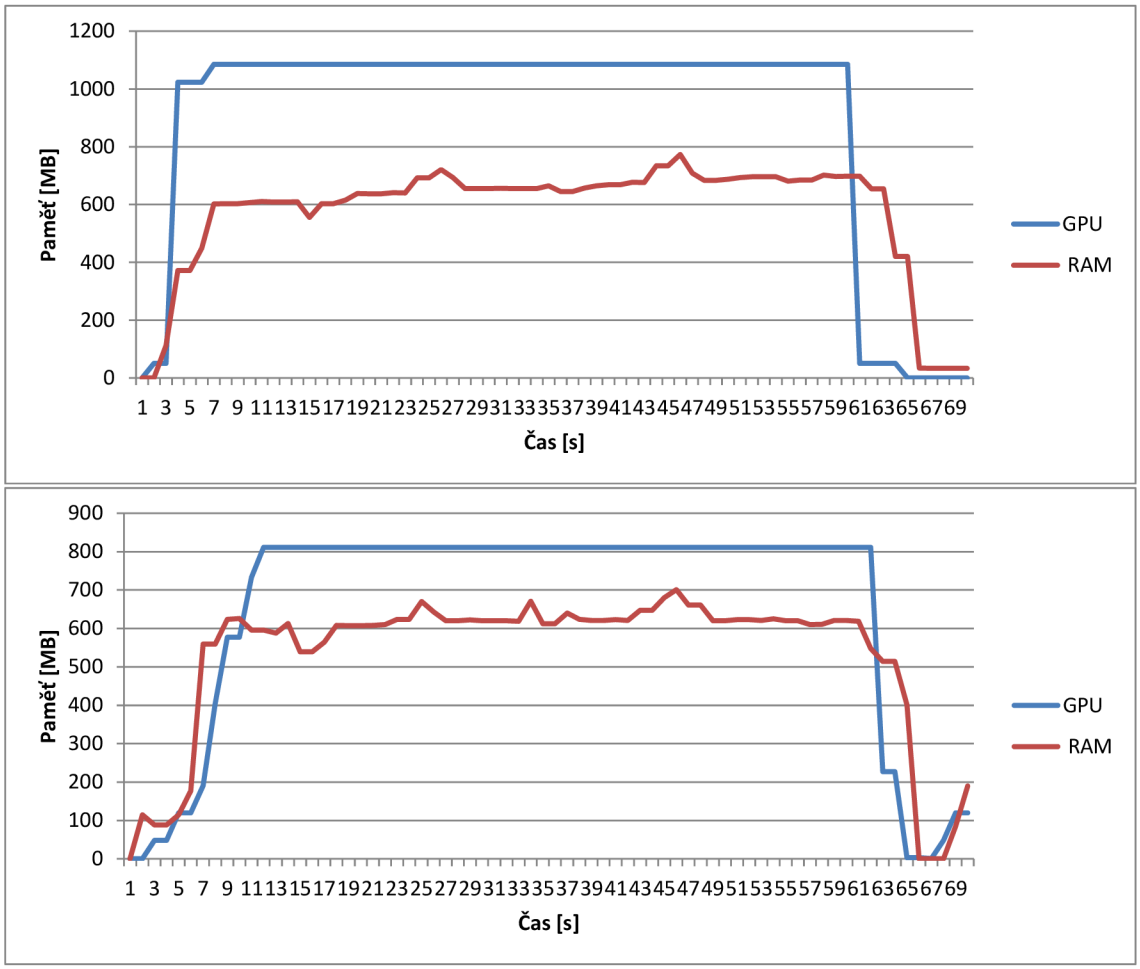


Obr. A.6: Na obrázku jsou vidět grafy paměti RAM a GPU při enkódování. Nahoře je graf pro CUDA a dole graf pro DirectX. Graf je vázán k presetu *Lossless High Quality*.

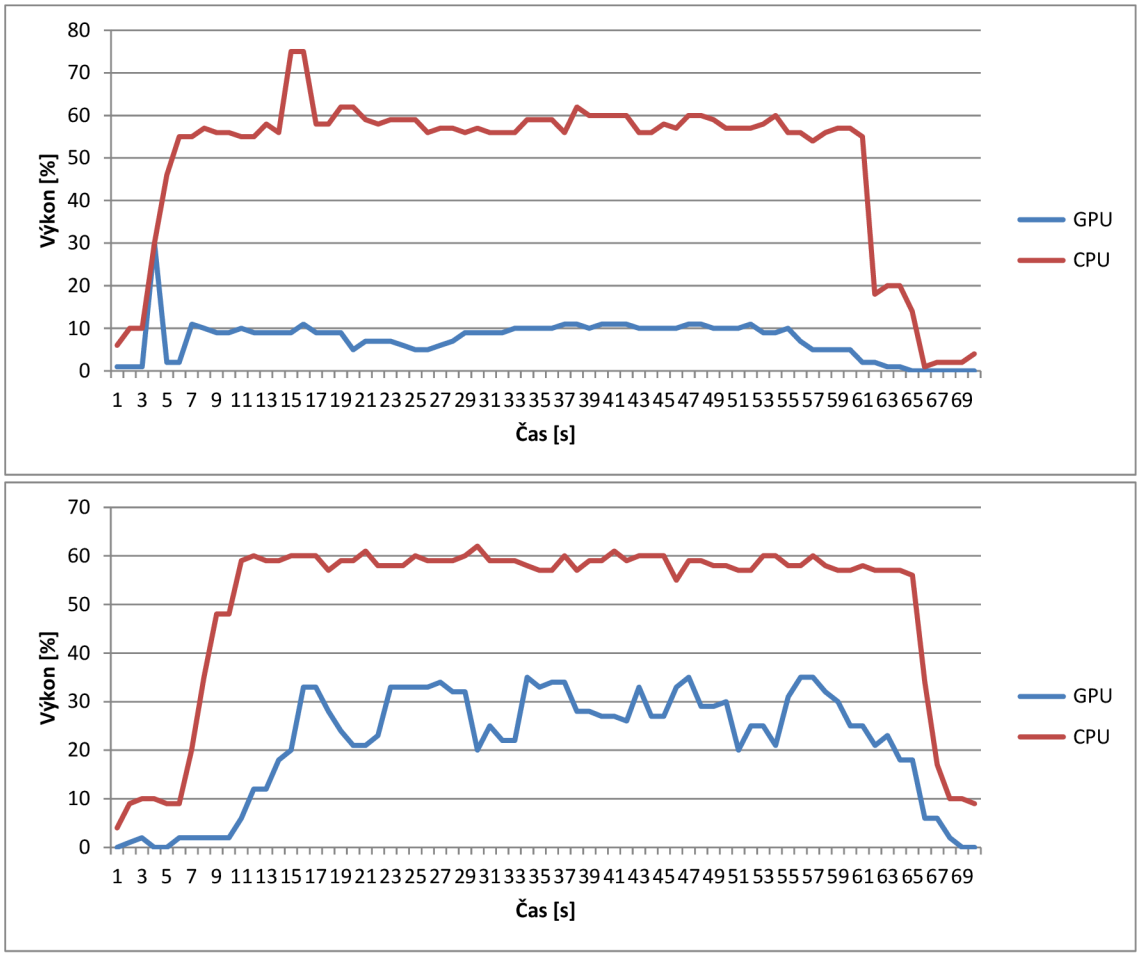
A.2 Grafy vytížení CPU, GPU a RAM v závislosti na parametru QP



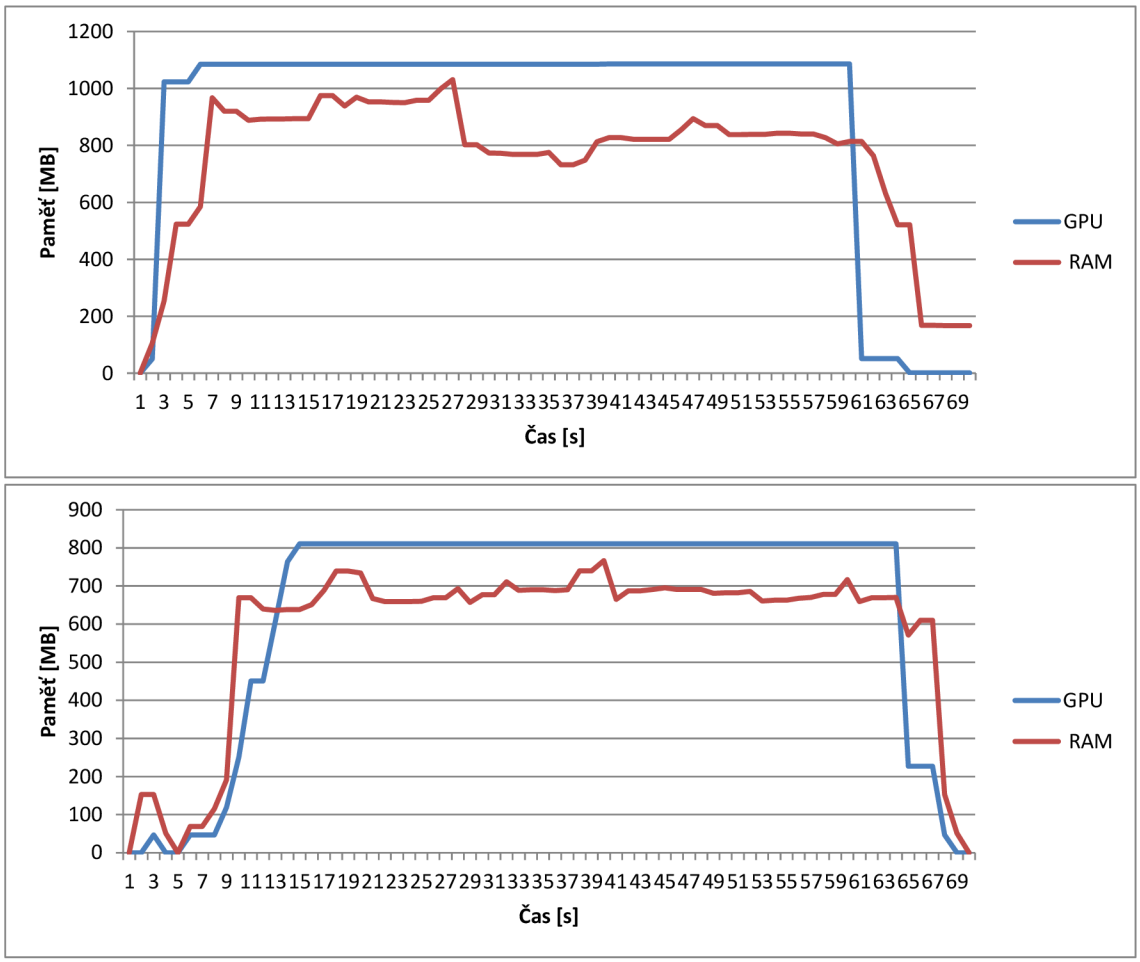
Obr. A.7: Na obrázku jsou vidět grafy výkonu CPU a GPU při enkódování v závislosi na parametru QP. Nahoře je graf pro CUDA a dole pro DirectX. Tento výsledek byl meřen na kodeku HEVC s parametrem QP=1.



Obr. A.8: Na obrázku jsou vidět grafy paměti RAM a GPU při enkódování v závislosi na parametru QP. Nahoře je graf pro CUDA a dole pro DirectX. Tento výsledek byl meřen na kodeku HEVC s parametrem QP=1.

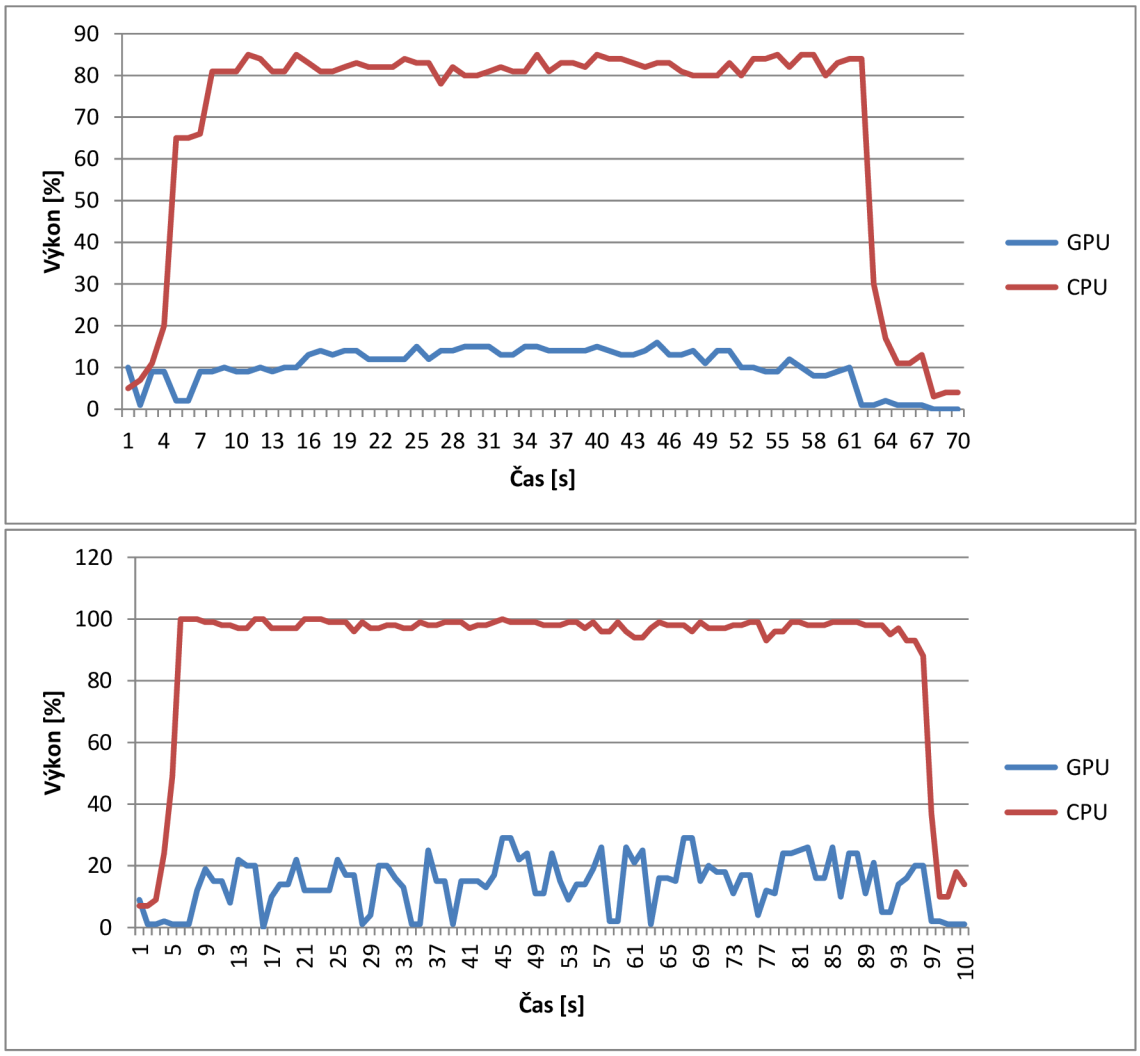


Obr. A.9: Na obrázku jsou vidět grafy výkonu CPU a GPU při enkódování v závislosi na parametru QP. Nahoře je graf pro CUDA a dole pro DirectX. Tento výsledek byl meřen na kodeku HEVC s parametrem QP=50.

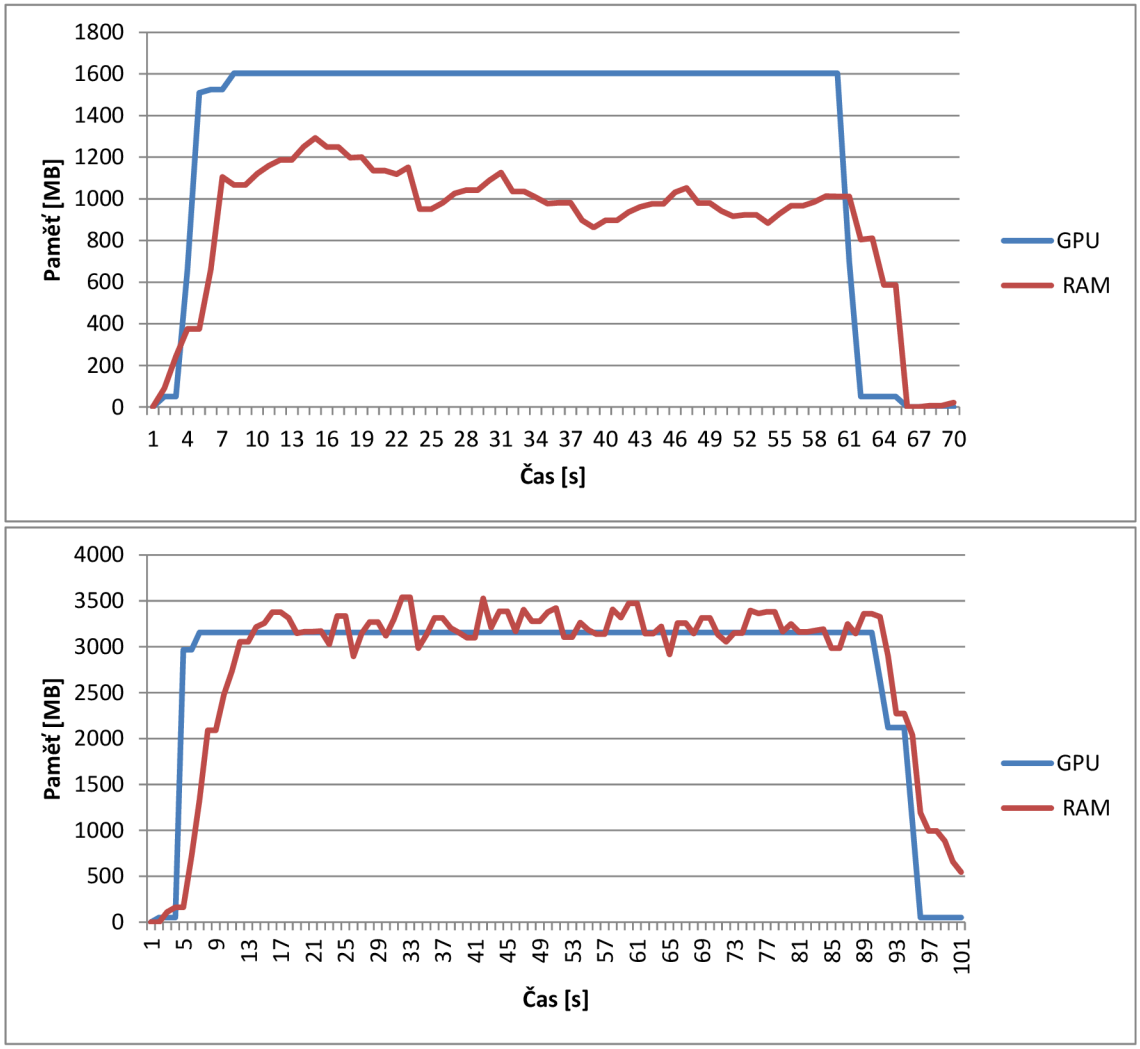


Obr. A.10: Na obrázku jsou vidět grafy paměti RAM a GPU při enkódování v závislosi na parametru QP. Nahoře je graf pro CUDA a dole pro DirectX. Tento výsledek byl meřen na kodeku HEVC s parametrem QP=50.

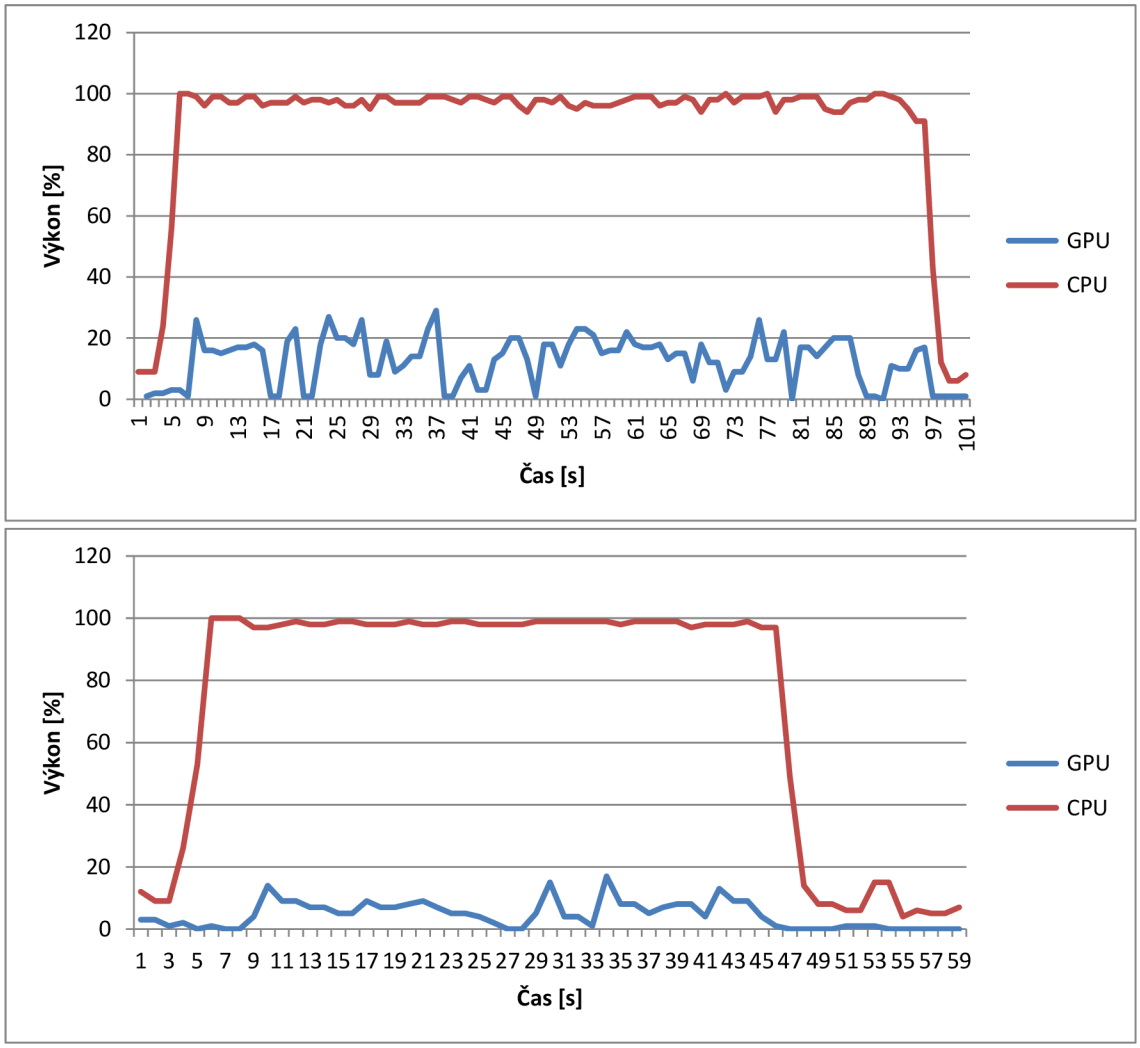
A.3 Grafy vytížení CPU, GPU a RAM v závislosti na počtu připojených kamer



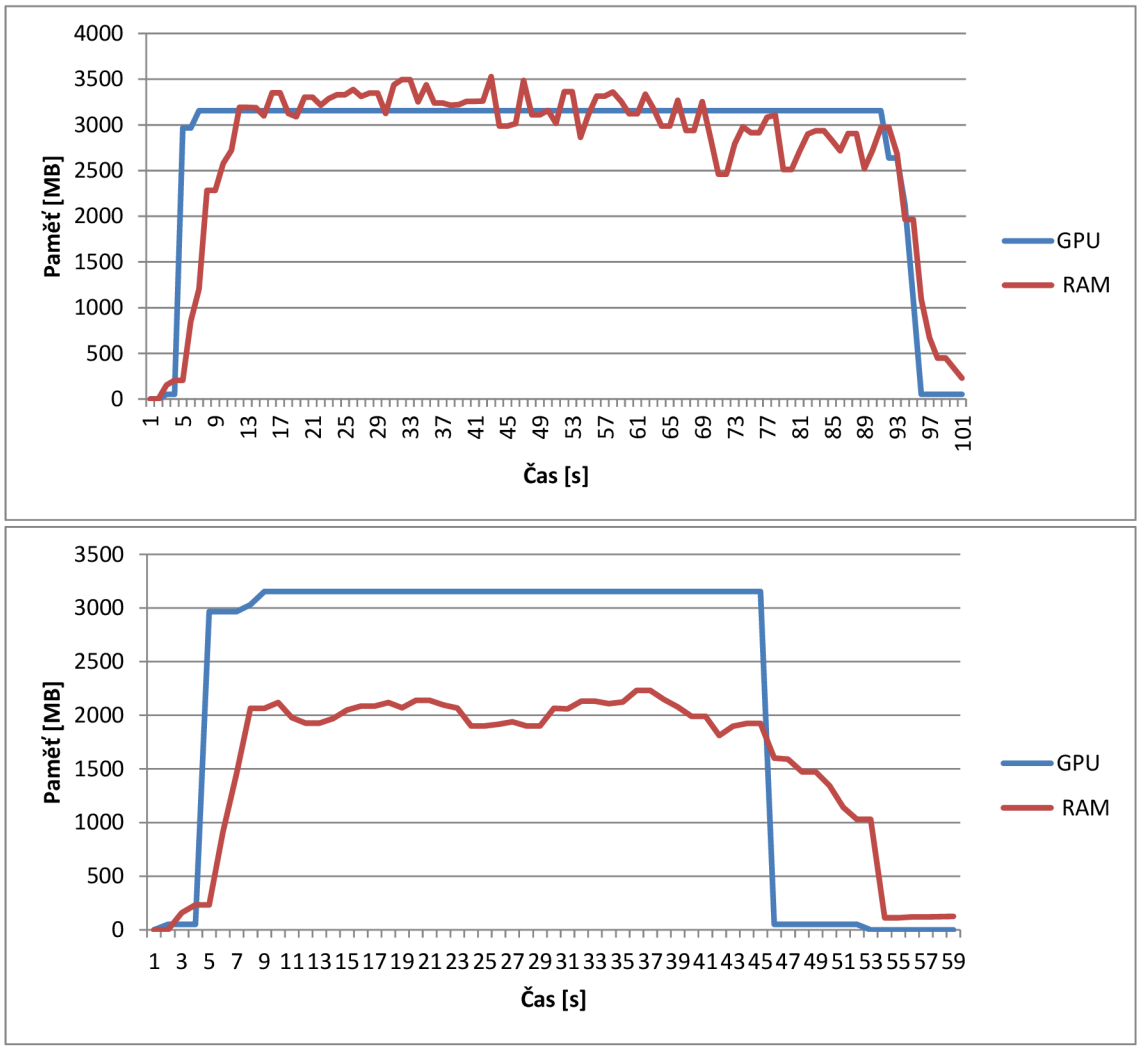
Obr. A.11: Na obrázku jsou vidět grafy výkonu CPU a GPU při enkódování s využitím CUDA do kodeku H.264. Nahoře je graf pro zapojené tři kamery a dole graf pro šest kamer. Graf je vázán k presetu *High Performance*.



Obr. A.12: Na obrázku jsou vidět grafy paměti RAM a GPU při enkódování s využitím CUDA do kodeku H.264. Nahoře je graf pro zapojené tři kamery a dole graf pro šest kamer. Graf je vázán k presetu *High Performance*.



Obr. A.13: Na obrázku jsou vidět grafy výkonu CPU a GPU při enkódování s využitím CUDA do kodeku H.264 při zapojení šesti kamer. Nahoře je graf pro nastavení dva rámce za sekundu a dole graf pro dva rámce za sekundu. Graf je vázán k presetu *High Performance*.



Obr. A.14: Na obrázku jsou vidět grafy paměti RAM a GPU při enkódování s využitím CUDA do kodeku H.264 při zapojení šesti kamer. Nahoře je graf pro nastavení dva rámce za sekundu a dole graf pro dva rámce za sekundu. Graf je vázán k presetu *High Performance*.