VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

AKCELERACE KRYPTOGRAFIE POMOCÍ GPU

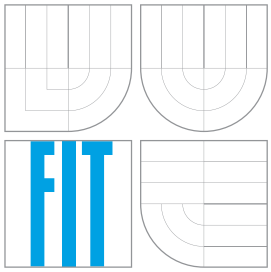
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JOSEF POTĚŠIL

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

AKCELERACE KRYPTOGRAFIE POMOCÍ GPU

CRYPTOGRAPHY ACCELERATION USING GPU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JOSEF POTĚŠIL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETR LAMPA

BRNO 2011

Abstrakt

V této práci bude čtenář nejprve seznámen s vybranými pojmy z oblasti kryptografie. Ve spojení s popisem architektury a programových prostředků pro programování grafických karet (CUDA, OpenCL) byl vybrán algoritmus AES, za účelem jeho akcelerace pomocí GPU. Dále se práce zabývá mapováním aplikačních rozhraní, která existují pro komunikaci se specializovanými krypto-koprocesory v jádrech operačních systémů Linux/BSD (CryptoAPI, OCF) a jejich podporou uvnitř multiplatformní knihovny OpenSSL. Následně práce pojednává o implementačních detailech, dosažených zrychleních a o integraci s OpenSSL. Závěr pojednává o možnostech využití implementovaného algoritmu a krátce o jeho využití přímo v jádrech operačního systému.

Abstract

The reader will be familiar with selected concepts of cryptography consited in this work. AES algorithm was selected in conjunction with the description of architecture and software for programming graphic cards (CUDA, OpenCL), in order to create its GPU-accelerated version. This thesis tries to map APIs for communication with crypto-coprocessors, which exist in kernels of Linux/BSD operating systems (CryptoAPI, OCF). It examines this support in the cross-platform OpenSSL library. Subsequently, the work discusses the implementation details, achieved results and integration with OpenSSL library. The conclusion suggests how the developed application could be used and briefly suggests its usage directly by the operating system kernel.

Klíčová slova

Kryptografie, GPU, CUDA, OpenCL, CryptoAPI, OCF, OpenSSL, AES

Keywords

Cryptography, GPU, CUDA, OpenCL, CryptoAPI, OCF, OpenSSL, AES

Citace

Josef Potěšil: Akcelerace kryptografie pomocí GPU, diplomová práce, Brno, FIT VUT v Brně, 2011

Akcelerace kryptografie pomocí GPU

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana inženýra Petra Lampy.

.....
Josef Potěšil
25. května 2011

Poděkování

Tímto bych chtěl poděkovat vedoucímu své práce za jeho vedení, vstřícnost, usměrňování a zajištění potřebného vybavení.

Obsah

Kapitola 1

Úvod

Moderní grafické karty zaznamenaly v posledních několika letech výrazný nárůst v oblastech výkonnosti, programovatelnosti a možnostech všeobecného použití. Hrubý výkon této masivně paralelní architektury několikanásobně převyšuje současné možnosti klasických procesorů. Je proto užitečné hledat pro tyto čipy další uplatnění.

S rostoucí dostupností internetu, jeho rychlými linkami, migrací velkého množství služeb právě na internet roste potřeba zabezpečení informací přenášených po této síti. Kryptografie tak nabývá na významu nejen pro zabezpečení datových přenosů, ale i pro bezpečné uložení perzistentních dat. Nicméně kryptografické algoritmy nepatří mezi výpočetně nenáročné, proto se začaly před několika lety na trhu objevovat specializované krypto-akcelerátory.

Tato práce se snaží o náhradu těchto specializovaných a relativně drahých zařízení běžně dostupnými grafickými kartami posledních několika generací.

Kapitola 2 seznamuje obecně s pojmy z kryptografie, které jsou používány v dalších kapitolách, zaměřuje se převážně na oblast symetrické kryptografie a také zmiňuje režimy blokových šifer za účelem možnosti jejich paralelizace. Konec kapitoly teoreticky popisuje implementovaný algoritmus AES.

V kapitole 3 jsou zmíněna aktuálně nejpoužívanější aplikačně programová rozhraní, která se používají k programování grafických čipů. Kapitola se zabývá architekturou těchto rozhraní i vzhledem k jeho mapování na grafický hardware. Zmiňuje také několik podmínek, které je vhodné dodržovat za účelem dosažení maximálního výkonu.

Kapitola 4 se snaží mapovat aktuální dostupná rozhraní přítomná nejen v operačních systémech GNU/Linux a FreeBSD ale i samostatných knihovnách. Tato rozhraní slouží k využití specializovaných obvodových krypto-akcelerátorů za účelem integrace s vlastními implementacemi kryptografických algoritmů.

Následuje kapitola 5 zabývající se implementačními detaily algoritmu AES, který byl implementován pomocí CUDA a OpenCL rozhraní. Dále se zmiňuje o implementaci blokových režimů a následnou integrací těchto verzí algoritmu AES s knihovnou OpenSSL.

Další kapitola 6 obsahuje popis testovací metodiky a shrnuje výkonnost implementovaného řešení, porovnává dosaženou rychlost řešení pro grafickou kartu s klasickým řešením prováděným procesorem. Ukazuje rozdíly mezi šifrováním a dešifrováním včetně vlivu blokových režimů na výkon. Snaží se také vysvětlit podmínky dosažení maximálního výkonu.

Závěrečná kapitola 7 shrnuje výsledky této práce, získané poznatky a dosažená výkonnostní zlepšení. Také se zabývá možností integrace s operačním systémem pomocí dostupných rozhraní.

Kapitola 2

Kryptografie

Tato kapitola si bere za cíl seznámit čtenáře s vybranými pojmy z oblasti kryptografie a zabezpečení. Více se zaměřuje na symetrickou kryptografii s výčtem režimů, v jakých mohou symetrické šifry pracovat a nakonec rozebírá implementovanou šifru AES.

2.1 Úvod

Mezi velmi často zaměňovanými pojmy patří kryptografie, kryptoanalýza a kryptologie. Kryptografie většině lidí splývá s pojmem šifrování. Jedná se o transformaci otevřeného textu na šifrovaný (nečitelný) a naopak s využitím utajené informace. To vyhovovalo původní definici, pro dnešní moderní kryptografii by bylo potřeba tuto definici poněkud rozšířit. Kryptoanalýza se zabývá zpětnou transformací zašifrovaného textu na otevřený bez znalosti utajované informace (hesla). Provádí analýzu matematických základů, na nichž jsou algoritmy postaveny a zkoumá jejich potenciální slabiny. Kryptologie je věda zastřešující oba pojmy, kryptografii a kryptoanalýzu.

Cílem kryptografie je zajistit dosažení čtyř základních bezpečnostních cílů:

1. **důvěrnosti** neboli utajení obsahu zprávy,
2. **autentizace** tedy ujištění, že jedinec vydávající se například za autora zprávy je i ve skutečnosti tím, za koho se vydává,
3. **integrity** čili zajištění ochrany proti poškození, úmyslné i neúmyslné manipulaci se zprávou, tím pádem ujištění, že data nebyla nijak modifikována,
4. **nepopiratelnosti** čímž je míněno, že autor zprávy nemůže popřít, že danou zprávu vytvořil/zašifroval/přijal poté, co tak učinil.

2.2 Dělení kryptografie

2.2.1 Historická a moderní kryptografie

Z historického hlediska se kryptografie dělí na historickou a moderní. Toto dělení nastalo přibližně v druhé polovině dvacátého století a rozděluje ji tzv. Kerckhoffův princip. Tento princip pruského důstojníka z 19. století říká: *„Utajení a bezpečnost zašifrovaných dat nesmí záležet na utajení postupu, kterým se šifrují. Naopak, vždy se musí předpokládat, že váš nepřítel zná šifru (algoritmus) do nejmenších detailů. Utajení musí spočívat pouze v klíči*

(např. hesle), které nezná nikdo jiný.“ [16] Nejznámějšími zástupci historických šifer jsou například Caesarova nebo Vigenerova šifra. V případě Caesarovy šifry stačilo pouze vědět, že se jedná o ni a dešifrování pak již bylo snadné. Bezpečnost byla tedy založena na utajení algoritmu. Dá se říci, že všechny moderní šifry důsledně dodržují Kerckhoffův princip a popisy algoritmů včetně matematického aparátu, na nichž šifry staví, jsou veřejně známy.

2.2.2 Asymetrická a symetrická kryptografie

Další dělení kryptografie je pak na symetrickou a asymetrickou. Zásadní rozdíl spočívá v použití klíče určeného k šifrování a dešifrování. Asymetrická kryptografie, často označovaná také jako kryptografie s veřejným klíčem, používá dvojici klíčů, které spolu matematicky souvisí, jeden klíč pro šifrování a druhý pro následné dešifrování. Platí, že jeden klíč je veřejný, tzn. je možno jej veřejně získat/stáhnout. V závislosti na požadovaných bezpečnostních funkcích se liší, který klíč použijeme pro šifrování. Zjednodušeně řečeno, pro zajištění autentizace, integrity a nepopiratelnosti se používá pro šifrování privátní klíč, aby pomocí veřejného klíče bylo možno zprávu dešifrovat a ověřit. Druhý způsob se použije k dosažení důvěrnosti. Aby se daly zajistit všechny cíle, je možné oba způsoby kombinovat. Nejprve tedy provést šifrování jedním způsobem a následně druhým. Je zde jedno, který způsobem budeme šifrovat jako první, oba přístupu jsou z bezpečnostního hlediska rovnocenné, i když častěji se používá varianta nejprve šifrovat privátním klíčem a následně veřejným. Situace bývá přirovnávána k zacházení s dopisem, který se nejprve podepíše a následně vloží do obálky. Typickými zástupci algoritmů jsou RSA, DSA, ElGamal.

Symetrická kryptografie používá jeden sdílený tajný klíč pro účely šifrování a dešifrování. Symetrická kryptografie dokáže v závislosti na režimu zajistit důvěrnost, autentizaci i integritu. Nicméně není schopná zajistit nepopiratelnost, vzhledem k tomu, že klíč je sdílen mezi autorem a protistranou. Výhodou je podstatně vyšší rychlost symetrických šifer (uvádí se 1000–10 000krát vyšší výkon) a menší délka klíčů. Velikost klíčů už nemusí být tak podstatnou výhodou kvůli příchodu asymetrických algoritmů založených na eliptických křivkách. Asi největším problémem při použití symetrických šifer je to, jak bezpečně přenést sdílený klíč protistraně. To je důvod uplatnění asymetrické kryptografie a používání tzv. hybridní kryptografie. Vygenerovaný tajný symetrický klíč si strany navzájem bezpečně vymění použitím asymetrické kryptografie a následný přenos dat probíhá zabezpečen symetrickou šifrou s nižšími nároky na výkon. Proto se tato práce dále soustředí primárně na symetrické šifry, jelikož se používají pro šifrování většího množství dat a grafický čip tak více uplatní svůj výkon. Typickými zástupci algoritmů jsou AES, DES (3DES), Twofish, Blowfish, IDEA, RC4, RC5.

Symetrické algoritmy	*Asymetrické algoritmy založené na diskrétním logaritmu*	*Asymetrické algoritmy založené na eliptických křivkách*
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

Tabulka 2.1: Doporučované délky klíčů poskytující podobné zabezpečení

Dále by bylo vhodné zmínit pojem hašovací funkce. Je to jednosměrná, deterministická

matematická funkce, která přijímá na vstupu data libovolné délky a výstupní hodnota (*hash*) má konstantní délku. Používá se pro účely podpisu, kontroly integrity a výpočtu kontrolního součtu, případně i ve spojení s generátory pseudonáhodných čísel. Typickými zástupci jsou algoritmy rodiny MD (*Message-Digest*) např. MD5, SHA (*Secure Hash Algorithm*). Ideální hašovací funkce by měla být snadno (rychle) spočitatelná, nemělo by být výpočetně zvládnutelné vytvořit zprávu s daným hash, nemělo by být výpočetně zvládnutelné upravit existující zprávu tak, aby se nezměnil hash a nemělo by být výpočetně zvládnutelné nalézt dvě vstupní hodnoty, aby měly stejný hash.

2.2.3 Blokové a proudové šifry

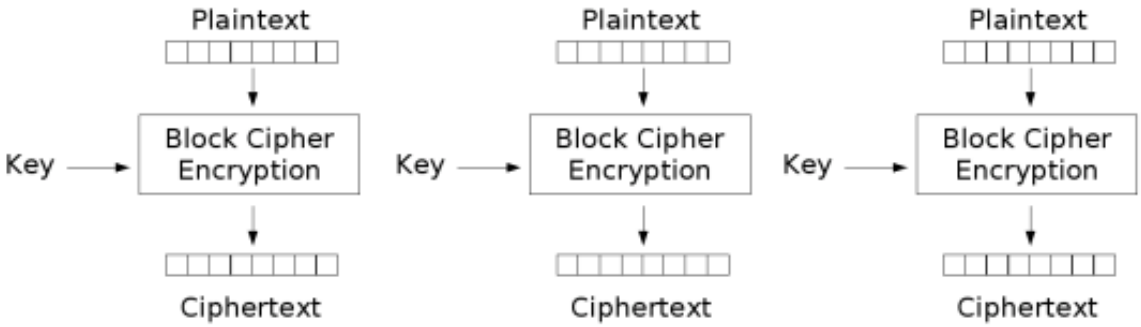
Posledním zde zmiňovaným dělením je rozdělení symetrických šifer na blokové a proudové. V případě proudových šifer jsou vstupní data nějakých způsobem kombinována s druhým, ideálně náhodným nebo alespoň pseudonáhodným, proudem dat. Typickou kombinační funkcí bývá *XOR*. Proudové šifry bývají často rychlejší a vystačí si s jednodušší obvodovou realizací. Naproti tomu kryptoanalýza zde bývá úspěšnější, pokud je chyba v implementaci. Data berou tak jak přicházejí, šifrují nezávisle buď jednotlivé bity, nebo častěji bajty. Blokové šifry přitom pracují s bloky pevné délky. Velikost záleží na konkrétním algoritmu, typicky to bývá 128bitů.

2.3 Režimy blokových šifer

Všechny symetrické blokové šifry umí pracovat s blokem dat určité velikosti. Aby mohly zpracovávat data libovolné délky, musí být nejprve data rozdělena do jednotlivých bloků a v případě potřeby připojeno zarovnání, více viz podkapitola 2.3.5. Existuje několik režimů, které jsou aplikovatelné prakticky na každou blokovou šifru. Režimy byly v minulosti intenzivně studovány za účelem propagování chyb v případě modifikace dat, z čehož se posléze vyvinula potřeba zajištění integrity dat. Dále je třeba zajistit náhodné statistické vlastnosti šifrovaných dat, což by nebylo při jednoduchém použití garantováno.

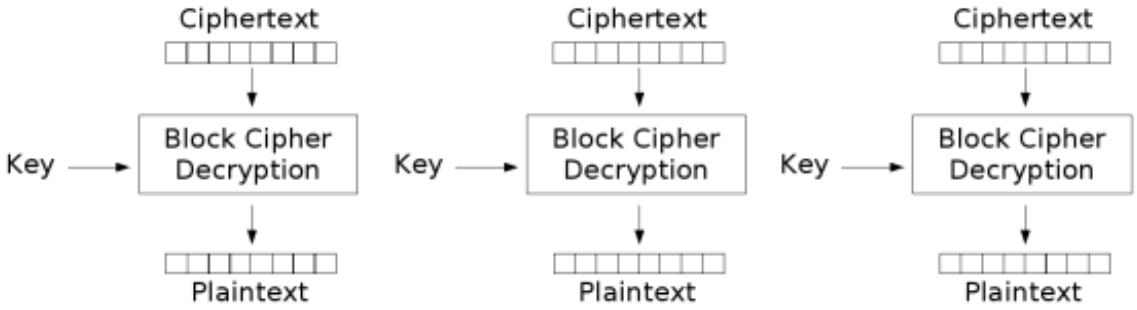
2.3.1 Electronic Code Book

Režim ECB je ten nejjednodušší možný. Data jsou rozdělena do bloků a nezávisle zpracována stejným klíčem, jde tedy o režim, který je přirozeně paralelní a nic nebrání běhu na grafických kartách.



Obrázek 2.1: Šifrování v režimu Electronic Codebook (ECB), převzato z [13].

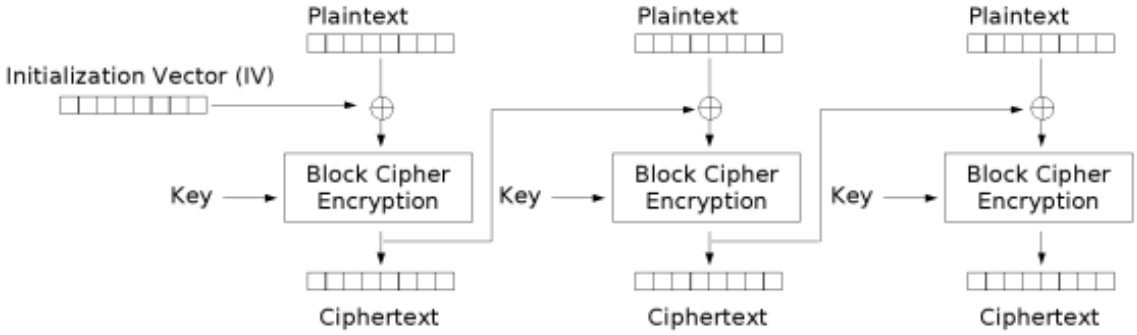
Nevýhodou tohoto režimu je, že stejná vstupní data jsou vždy zašifrována na stejná výstupní data, proto název kódová kniha. Kvůli této vlastnosti není zajištěna náhodnost výstupních dat, režim není dostatečně bezpečný a není tak doporučován v bezpečnostních protokolech, případně jen pro data velikosti několika bloků. Potenciálnímu útočníkovi tak umožňuje provádět slovníkové útoky, opakovat nebo přeskládávat bloky šifrovaných dat.



Obrázek 2.2: Dešifrování v režimu Electronic Codebook (ECB), převzato z [13].

2.3.2 Cipher-block Chaining

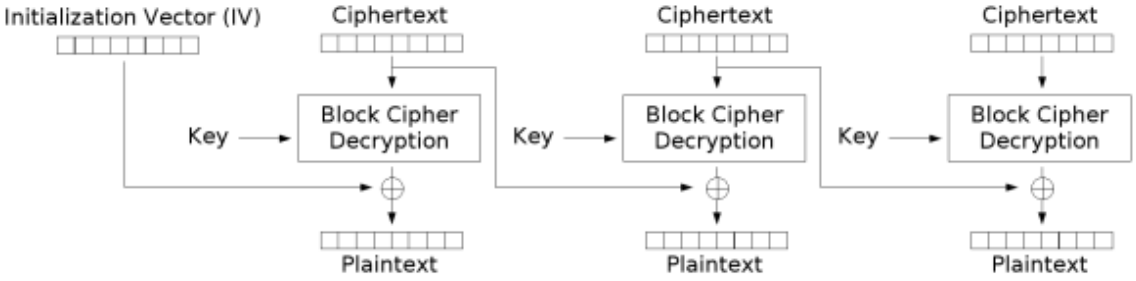
CBC režim je obecně považován za bezpečný, doporučovaný a také často využívaný. Tento režim pro transformaci bloku dat kombinuje výsledek operace na předešlém bloku s aktuálním. Zavádí se zde pojem inicializačního vektoru, viz podkapitola 2.3.4, který slouží pro práci s prvním blokem dat.



Obrázek 2.3: Šifrování v režimu Cipher-block Chaining (CBC), převzato z [13].

Při šifrování prvního bloku dat se zkombinuje tento blok s inicializačním vektorem operací XOR. Následně se data zašifrují, výstupem je první blok zašifrovaných dat. Na ten se opět aplikuje operace XOR s druhým blokem nešifrovaných dat, tento mezivýsledek je zašifrován a výsledek je k dispozici pro třetí šifrování a tak dále. Ve výsledku je tak daný blok dat závislý na výsledku šifrování všech předchozích bloků, proces je tím pádem sekvenční bez možnosti paralelizace. Toto chování má další vlastnost, pokud dojde ať už k náhodné nebo úmyslné změně určitého bloku dat, dojde k chybě při dešifrování tohoto a všech následujících bloků (propagace chyby).

Při dešifrování je první blok dat dešifrován a pak zkombinován operací XOR s iniciali-

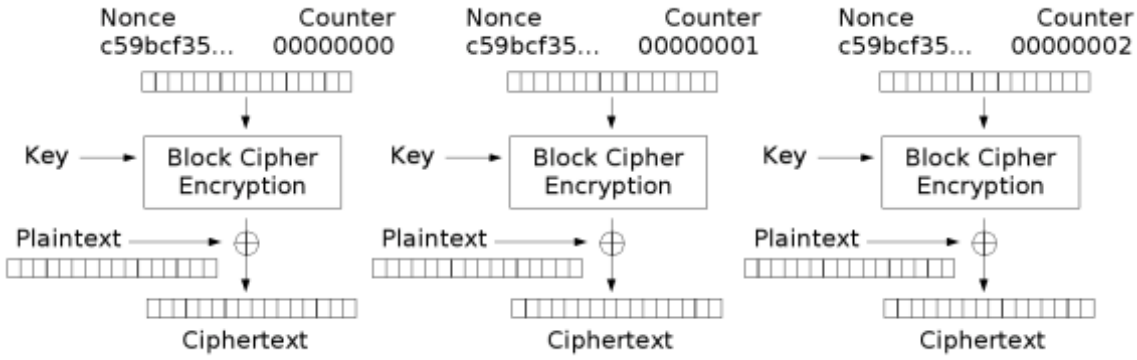


Obrázek 2.4: Šifrování v režimu Cipher-block Chaining (CBC), převzato z [13].

začním vektorem, aby byla obnovena původní data. Následně je každý další blok dešifrován a operací XOR opět obnoven s pomocí předchozího zašifrovaného bloku. Vzhledem k tomu, že operace dešifrování potřebuje aktuální a předchozí šifrovaný blok, tedy dva bloky vstupních dat, je tento režim při dešifrování plně paralelní.

2.3.3 Counter

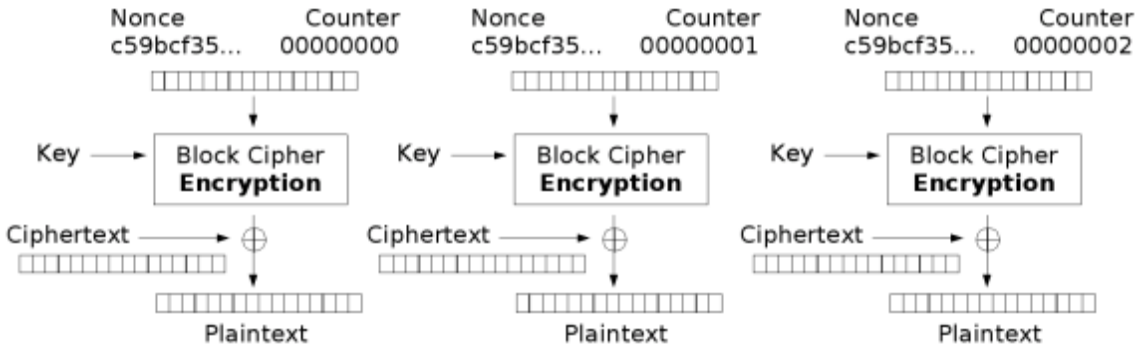
Režim čítače (CTR) v praxi mění blokovou šifru na proudovou, nebude zde třeba data zarovnávat. K chodu tohoto režimu je třeba generovat čísla o velikosti bloku dat dané šifry a následně je tento blok operací XOR zkombinován se vstupními daty. Generované čítače by měly být unikátní nejen v rámci šifrování jen jedné zprávy, ale ideálně i v rámci šifrování daným klíčem.



Obrázek 2.5: Šifrování v režimu Counter (CTR), převzato z [13].

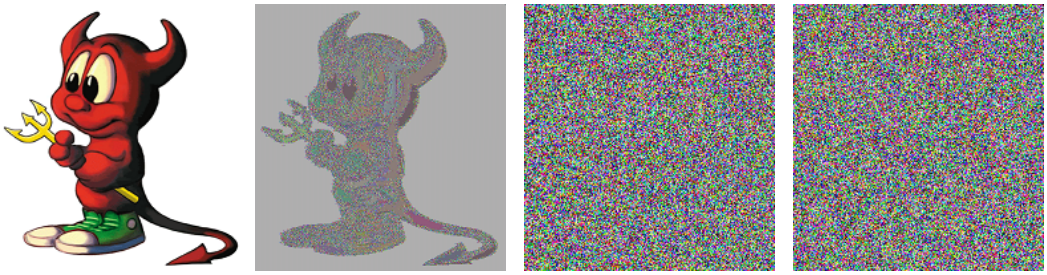
Při operaci šifrování je čítač pro daný blok zašifrován daným algoritmem a výsledek je pomocí operace XOR sloučen se vstupními daty. Chování je tentokrát mírně odlišné při šifrování posledního bloku, kdy se z čítače použije pouze tolik bajtů dat, kolik bude třeba a pouze ta s nejvíce platnými bajty (most significant), nepoužité bajty budou jednoduše zahozeny. Chování je totožné i během dešifrování. Dokonce při dešifrování není ani volán algoritmus v inverzním režimu, tedy dešifrovacím.

Hlavní výhodou tohoto režimu je podpora paralelizace, data tedy mohou být zpracována nezávisle, pokud se dá určit hodnota čítače. Navíc se dají při znalosti počáteční hodnoty čítače a klíče šifrované hodnoty předpočítat a poté, co budou data k dispozici, se jen provede operace XOR a výsledek je hotov.



Obrázek 2.6: Šifrování v režimu Counter (CTR), převzato z [13].

Na obrázku 2.7 je možné pozorovat výsledky šifrování algoritmu AES při použití 256bitového klíče v různých režimech. Je zde patrné, že při použití režimu ECB lze vidět alespoň obrys obrázku s maskotem FreeBSD. Při použití režimů CTR a CBC již nelze poznat žádný náznak původních dat, výsledky připomínají náhodný šum, jak by tomu mělo být.



(a) Beastie (originál) (b) ECB (c) CTR (d) CBC

Obrázek 2.7: Ukázka výsledků šifrování v různých režimech blokové šifry AES-256

2.3.4 Inicializační vektor

Je velmi často zmiňovaný pojem a představuje blok dat pevné velikosti. Ta závisí na aplikaci, která jej ke své činnosti používá. Ve spojení s blokovými šiframi se rovná velikosti bloku. Na inicializační vektor jsou z kryptografického hlediska kladeny požadavky na jeho náhodnost nebo alespoň pseudonáhodnost, aby nebyla ohrožena míra poskytovaného zabezpečení. Pokud by se daly jeho bity predikovat, mohl by si potenciální útočník zvolit vhodný text, ten nechat zašifrovat a zkoušet útok se známým textem. Nároky ale opět záleží na jeho aplikaci, v některých použitích tak jenom stačí, aby se vektor neopakoval. Dále by se také neměl inicializační vektor používat vícekrát pro šifrování se stejným klíčem. Nicméně hodnota inicializačního vektoru se nemusí chránit (šifrovat), jelikož je potřeba pro inverzní režim šifry.

2.3.5 Zarovnání

V závislosti na zvoleném blokovém režimu je potřeba zarovnávat vstupní data na nejbližší násobek velikosti bloku, aby šifra měla kompletní vstup. Nastává však otázka čím tuto me-

zeru vyplnit. Existuje několik metod, i standardizovaných, které jsou doporučovány. Operují na úrovni bitu nebo bajtů. Jeden ze způsobů pracující na úrovni bitů je nastavit první bit výplně na jedničku a zbylé vyplnit nulami. Asi nejznámější způsob vyplňování byl definován standardem PKCS7 (*public-key cryptography standard*). Ten pracuje s celými bajty. Hodnota každého bajtu výplně je stejná a je rovna počtu přidávaných bajtů výplně. Vyvstává ještě jedna otázka. Co když jsou vstupní data již zarovnaná na velikosti bloku? V takovém případě se připojí celý blok s patřičnou hodnotou výplně. Existují však i pokročilejší metody, které toto nedělají a jsou schopny tuto informaci zakódovat do vstupních dat.

2.4 Advanced Encryption Standard

AES je standard pro symetrickou blokovou šifru postavenou na algoritmu *Rinjdeal* [3], který byl schválen americkým úřadem *National Institute of Standards and Technology* roku 2001. AES je úspěšným nástupcem jiného velmi rozšířeného algoritmu *Data Encryption Standard* (DES), jelikož ten byl již zastaralý. NIST tak vyhlásila soutěž o návrh nového šifrovacího algoritmu a návrh belgičanů *Joana Daemena* a *Vincenta Rijmena* byl vybrán jako vítězný.

AES spadá do kategorie substitučních/permutačních šifer. Velikost bloku byla pevně stanovena na 128bitů, přestože šifra Rinjdeal podporuje více velikostí bloku (například 192, 256). Podporované velikosti šifrovacího klíče jsou 128, 192 a 256bitů.

Některé operace jsou definovány na úrovni práce s bajty. Ty přestavují prvky konečného Galoisova pole $GF(2^8)$. Jiné operace zase používají 32bitové hodnoty. Šifra vlastně provádí několik transformačních kol, které převedou čistý vstupní text na žasifrovaný. V závislosti na kombinaci velikosti klíče se liší počet kol algoritmu. Tabulka 2.2 tyto kombinace reflektuje.

	Velikost klíče (násobky 32 bitů)	Velikost bloku (násobky 32 bitů)	Počet kol
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Tabulka 2.2: Počet kol algoritmu Rinjdeal.

Nejprve se musí připravit klíč. Uživatelský klíč se expanduje do takové podoby, aby byl použitelný během jednotlivých kol. Následně se pro oba režimy, standardní i inverzní, volá podle počtu kol několikrát transformační funkce. Data si pro lepší představu představte organizovaná do matice 4x4. Funkce se skládá ze čtyř hlavních kroků:

1. *SubBytes* substituce prvků vstupní matice,
2. *ShiftRows* rotace prvků unitř řádků matice,
3. *MixColumns* násobení sloupců polynomem,
4. *AddRoundKey* přičtení hodnoty expandovaného klíče.

Tyto čtyři kroky tvoří jedno standardní kolo šifry. Poslední kolo se lehce odlišuje. Typická struktura implementace šifry je naznačena níže:

Pseudokód algoritmu AES

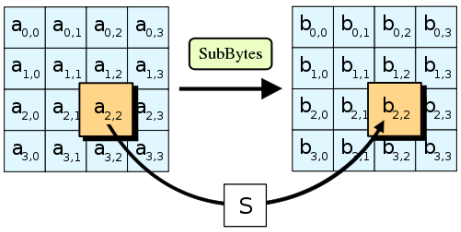
```
void Cipher( unsigned char in[4][4], unsigned char out[4][4] )
{
    unsigned char state[4][4] = in;
    AddRoundKey( state, roundkey );

    for ( int round = 1; round <= ROUNDS; ++round )
    {   // standard rounds
        SubBytes( state );
        ShiftRows( state );
        MixColumns( state );
        AddRoundKey( state, roundkey );
    }
    // final round
    SubBytes( state );
    ShiftRows( state );
    AddRoundKey( state, roundkey );

    out = state;
}
```

2.4.1 SubBytes

V tomto kroku transformace se provádí nelineární substituce jednotlivých bajtů bloku pomocí vyhledávací tabulky S-Box nezávisle na pozici bajtu. Matematický základ vyhledávací tabulky je postaven na substituci inverzním prvkem vzhledem k operaci násobení uvnitř konečného Galoisova pole $GF(2^8)$ s následnou afinní transformací podle vzorce 2.1, kde b představuje substituovaný bajt, b_i je i-tý bit tohoto bajtu a c_i i-tý bit konstanty c, která nabývá hodnot 63 nebo 99.

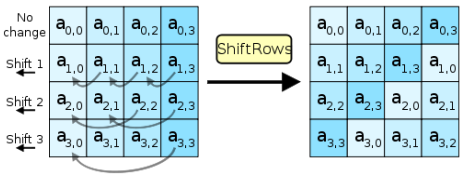


Obrázek 2.8: Převzato z [14]

$$b_i = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus c_i \tag{2.1}$$

2.4.2 ShiftRows

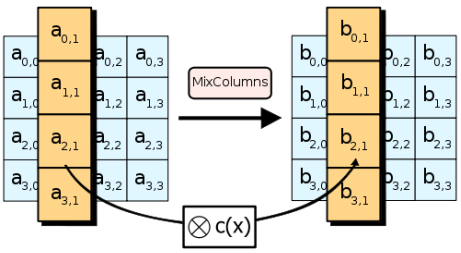
Zde se pracuje s bajty rozdělenými do řádků matice. Cyklicky se posouvají bajty na jednotlivých řádcích o různý počet prvků. Situace s počtem posunutí je v AES ve srovnání s algoritmem Rinjdeal zjednodušena díky užití pouze 128 bitových bloků. První řádek zůstává zachován, druhý se posouvá o jeden prvek, třetí o dva prvky a čtvrtý o tři prvky doleva, v inverzním režimu šifry je rotace na opačnou stranu.



Obrázek 2.9: Převzato z [14]

2.4.3 MixColumns

Nyní se transformují bajty v rámci sloupců matice s daty. Vstupem jsou čtyři bajty sloupce, krok vytvoří čtyři bajty, kde každý bajt vstupu ovlivní všechny bajty výstupu stejnou mírou. Každý sloupec je chápán jako polynom Galoisova pole $GF(2^8)$, který je násoben konstantním polynomem, který je dan rovnicí $c = 3x^3 + x^2 + x + 2$. Následuje operace modulo $x^4 + 1$. Spolu s krokem *ShiftRows* zajišťuje šifře kryptografickou vlastnost zvanou *difuze*. Jejím cílem je dosažení toho, aby vznikla maximálně komplexní závislost bitů výstupních dat na vstupních, aby například při ovlivnění jednoho bitu vstupu se velmi výrazně a také pokud možno nepředvídatelně změnil celý výstup.



Obrázek 2.10: Převzato z [14]

2.4.4 AddRoundKey

Tento krok je implementačně asi nejjednodušší ze všech a kombinuje klíč s aktuálním stavem dat operací *XOR*. Zde je důležité zmínit, že se nepoužívá uživatelský klíč, ale jeho expandovaná forma. Data klíče jsou tak pro každé kolo jiná. Velikost expandovaného klíče se vypočítá jako počet kol zvýšen o jedničku vynásobená velikostí bloku, například pro 256bitovou verzi šifry má expandovaný klíč velikost 1920 bitů $(128 \cdot (14 + 1) = 1920)$.

2.4.5 Optimalizace šifry

Z předchozího popisu vyplývá, že data jsou v daném kole nejprve substituována, následně posunuta, násobená konstantou a nakonec je přičten klíč. Pokud budeme uvažovat alespoň 32bitovou architekturu, je možno zkombinovat kroky *SubBytes*, *ShiftRows*, *MixColumns* a převést je do formy vyhledávání ve vyhledávacích tabulkách. Všechny zmíněné kroky lze tedy nahradit čtyřmi vyhledávacími tabulkami obsahující 256 prvků velikosti 32bitů (celkem 4KB). Pro jedno standardní kolo se provede 16 vyhledání v tabulkách a výsledky se zkombinují mezi sebou a klíčem opět pomocí operací *XOR*. Paměťové nároky lze ještě zmírnit a to na čtvrtinu. Stačí vyhledávat v jedné tabulce a přidat kód upravující vyhledané hodnoty, prakticky tedy kód generující zbylé tabulky. Pro poslední kolo je potřeba vymyskovat specifické bity, jelikož poslední kolo nezahrnuje krok *MixColumns*.

$$e_j = T_0\left[a_{0,j}\right] \oplus T_1\left[a_{1,j-1}\right] \oplus T_3\left[a_{2,j-2}\right] \oplus T_3\left[a_{3,j-3}\right] \oplus key_j \tag{2.2}$$

Výše uvedená rovnice 2.2 představuje transformaci 32 bitů jednoho standardního kola algoritmu AES, T_{0-3} označují vyhledávací tabulky, proměnná a aktuální data.

Kapitola 3

Aplikační programové rozhraní pro programování GPU

Tato kapitola se zabývá dostupnými rozhraními pro programování grafických karet, aby bylo možno využít výkonu grafických čipů k akceleraci co možná největšího spektra algoritmů. Popisuje architekturu dvou hlavních rozhraní a novinky, které přináší v souvislosti s novými GPU.

3.1 NVIDIA®CUDA™

CUDA je programový balík vyvinutý společností NVIDIA Corporation pro vývoj a běh aplikací, které chtějí využít výkon moderních grafických architektur k akceleraci vhodných paralelních algoritmů. CUDA je akronymem pro anglické *Compute Unified Device Architecture*. Tento balík byl vypuštěn v listopadu roku 2006. Nepoužívá jako dříve grafická rozhraní DirectX ani OpenGL pro přímé programování grafických čipů, přesto s těmito rozhraními umí spolupracovat. Jak název napovídá, není tato technologie určená pouze pro omezené množství produktů, například jen profesionální řadu Quardo, ale pro všechny grafické čipy generace G80 a novějších. Je tedy možné provozovat tyto výpočty i na méně výkonných a běžně dostupných kartách. Celkově se tento balík skládá z několika komponent:

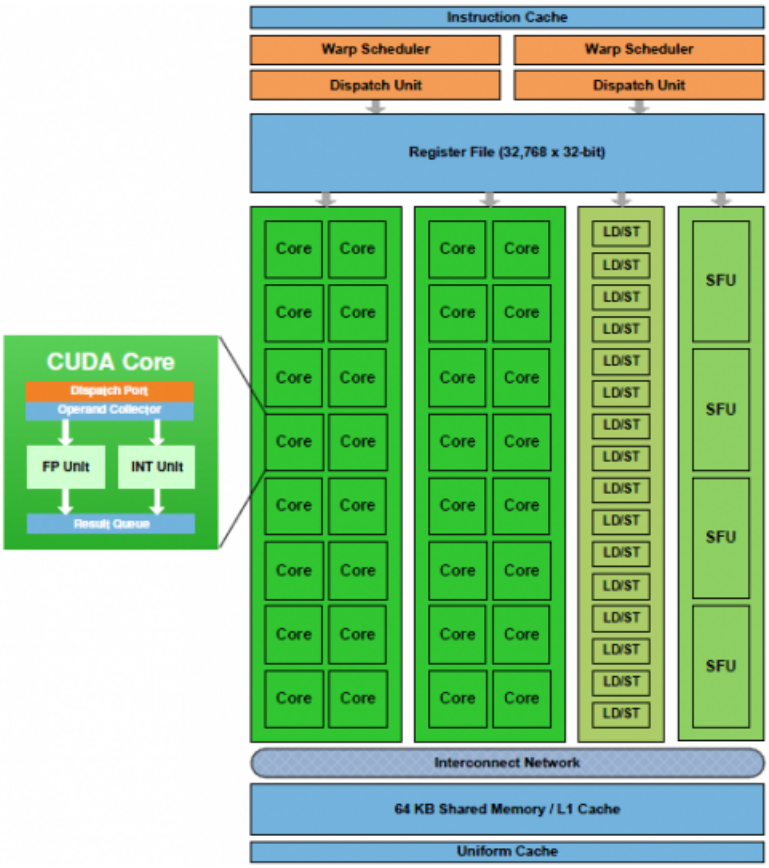
- grafické karty společnosti NVIDIA
- ovladače grafické karty patřičné verze
- CUDA Toolkit obsahuje především překladač *nvcc*. CUDA je jako jazyk odvozena z C/C++ a podporuje snad všechny konstrukce jazyka C, přidává několik specifických konstrukcí a s novějšími verzemi překladače čím dál více vlastností jazyka C++, např. podpora polymorfismu, výchozích parametrů, přetěžování operátorů, jmenných prostorů a šablonového programování. Dále tato sada nástrojů obsahuje několik knihoven:
  - *CUFFT* pro výpočet rychlé Fourierovi transformace
  - *CUBLAS* pro podporu základní lineární algebry (Basic Linear Algebra Subprograms)
  - *CURAND* pro generování pseudonáhodných a kvazi-náhodných čísel
  - *CUSPARSE* pro operace s řídkými maticemi

– knihovnu pro kódování a dekódování videa

- SDK (software development kit) s názornými příklady různých technik a algoritmů

3.1.1 Architektura

Architekturu CUDA balíku můžeme rozdělit na programovou a hardwarovou. Jádro grafického čipu je postaveno okolo pole stream multiprocesorů (SM). Tyto multiprocesory se skládají z několika jednodušších procesorů, CUDA jader (CUDA Core), jejichž počet záleží na generaci GPU. První čipy obsahovaly 8 jader, postupně se zvedají až k dnešním 48. Počet těchto jader je dnes základním výkonnostním faktorem spolu s jejich frekvencí. Na obrázku 3.1 je zobrazena jeho struktura. Každé jednoduché jádro obsahuje jednotku pro celočíselné operace (ALU) a jednu pro operace v plovoucí řádové čárce (FPU). FPU je určena pro jednodušší operace jako sčítání a násobení. Pro složitější operace typu sinus, kosinus, je přítomen menší počet tzv. special function unit (SFU).



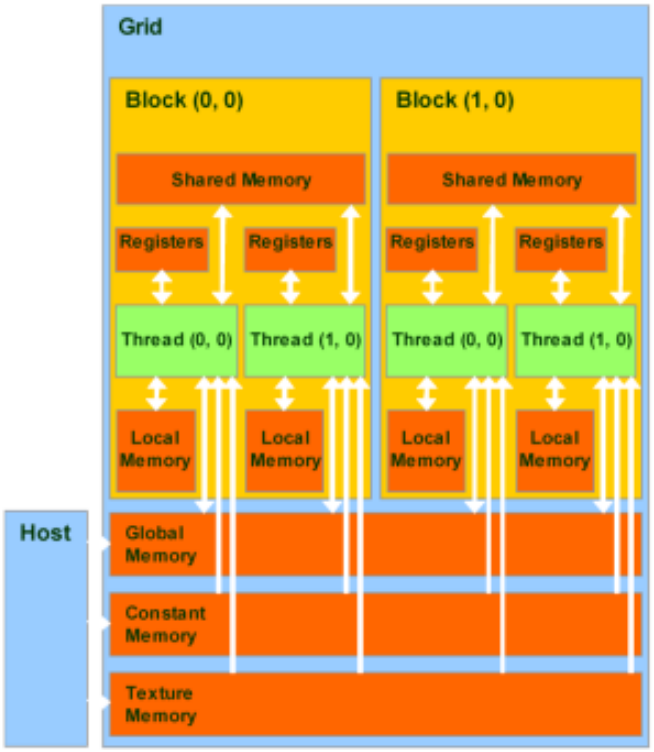
Obrázek 3.1: Struktura multiprocesoru architektury Fermi, převzato z [19]

Z programového hlediska je základní jednotkou vlákno. Vlákna se spouštějí v blocích určených rozměrů a dimenzí (3), celkově však maximálně 1024 vláken u poslední generace. Tento blok vláken se mapuje na jeden multiprocesor a na něm je vykonáván kód programu. Interně se tento blok vláken ještě dělí do tzv. *WARP* bloků po 32 vláknech. Kód vláken se v rámci WARP bloku vykonává současně. V rámci multiprocesoru vytvoření, správa a paralelní vykonání kódu vláken nestojí žádnou režii. Poté, co jeden multiprocesor dokončí

práci jednoho bloku vláken, vezme jednoduše další a jeho kód vykoná. Tato architektura je dobře škálovatelná, jelikož do budoucna stačí pouze zvýšit počet multiprocesorů a dojde k nárůstu výkonu, aniž by se musela upravovat struktura aplikace.

3.1.2 Paměťový model

Každý multiprocesor má přístup k několika typům paměti. Základem je registrové pole. Architektura čipu G80 obsahovala 8192 registrů, každý velikosti 32 bitů. V poslední architektuře se počet registrů zvýšil na čtyřnásobek, tedy 32768. Do registrů se ukládají skalární hodnoty, tedy žádná pole a přístup k nim je privátní pro jedno vlákno. Práce s těmito registry je nejrychlejší možná, latence je pouze jeden takt. Nastane-li situace okamžitého čtení po zápisu, trvá 24 cyklů, než se změna uloží. Tuto latenci je možno překrýt dostatečným počtem aktivních vláken v bloku.



Obrázek 3.2: Paměťová heirarchie architektury CUDA, převzato z [20]

Dále je zde sdílená paměť, která je přístupná a sdílená mezi všemi vlákny uvnitř jednoho bloku vláken a slouží tedy k propagaci informací mezi vlákny. Přístup je také velmi rychlý, nicméně o trochu pomalejší než registry. Velikost této paměti byla 16 KB v dřívějších architekturách, ve Fermi se zvýšila na čtyřnásobek. Sdílená paměť je opatřena šestnácti nebo dvaatřiceti paměťovými bankami pro přístup k 32 bitovým hodnotám. Pokud různá vlákna z poloviny WARP bloku budou číst ze stejné paměťové banky, dojde k serializaci přístupu a tím ke zpomalení. Výjimkou je situace, kdy by všechna vlákna četla ze stejné paměťové banky, pak by došlo k rozhlášení dat bez jakéhokoli konfliktu. Nově je v architektuře Fermi sdílená paměť kombinovaná s L1 vyrovnávací pamětí, sdílejí spolu adresový prostor. Některé aplikace tedy mohou potřebovat více sdílené paměti, jiné budou naopak profitovat z rychlé vyrovnávací paměti. To závisí na tom, kolik sdílené paměti aplikace potřebuje

a jak je možné předpovídat přístup do globální paměti. V architektuře Fermi je nově také L2 vyrovnávací paměť velikosti 768 KB sdílená všemi multiprocesory.

Globální paměť je hlavní pamětí grafické karty typicky o velikosti řádově stovek megabajtů. Přístup ke globální paměti není nijak optimalizován a trvá přibližně 400–600 cyklů. Podobně je na tom lokální paměť každého multiprocesoru. Ta slouží pro ukládání automatických proměnných, které se nevejdou do registrů nebo sdílené paměti. Lokální paměť je mapována v globální paměti, ale přístupná je pouze z jednoho multiprocesoru.

Dalším druhem paměti je konstantní paměť, jako jediná ze všech výše popisovaných je pouze pro čtení. Přístupná je po celou dobu běhu aplikace pro všechny multiprocesory. Jde o kus paměti velikosti 64 KB v globální paměti, který je na rozdíl od všech ostatních opatřen vyrovnávací pamětí. Pokud by vlákna četla ze stejné adresy, bude přístup prakticky okamžitý pro všechna vlákna, pokud by četla z různých adres, přístup bude stejný jako do globální paměti. Pro úplnost je třeba zmínit paměť textur podobných vlastností jako paměť konstant.

3.1.3 Podmínky efektivního běhu na GPU

Tato podkapitola se soustředí na důležité zásady při návrhu a implementaci algoritmů, které mají běžet a akcelerovat výpočet na grafických kartách. Asi samozřejmou podmínkou efektivního běhu na GPU vzhledem k jeho masivně paralelní architektuře je maximálně paralelní návrh samotného algoritmu. Ať už by výkon GPU jakkoli převyšoval klasická CPU, pokud by byl algoritmus ze své podstaty sekvenční, nedokázal by ani zdaleka využít maximálního výkonu a ve výsledku by tak vůbec nemusel zrychlovat výpočet. Datové závislosti mezi jednotlivými prvky zpracovávaných dat by měly být nulové nebo minimální. Následující doporučení jsou spíše implementačního rázu:

- Za každou cenu minimalizovat datové přenosy mezi pamětí procesoru a pamětí grafické karty. Jelikož špičková přenosová rychlost paměti grafických karet daleko přesahuje možnosti sběrnice PCI-Express, je třeba tyto přenosy omezit. V zásadě je proto lepší shlukovat menší datové přenosy do větších balíků dat. Dále pokud algoritmus potřebuje nějaké dodatečné uložiště dat pro meziprodukty, mělo by být alokováno na grafické kartě. Pro dosažení větší přenosové rychlosti je také doporučováno alokovat paměť, která je označena příznakem, aby systém neodkládal paměť například na pevný disk. To přináší ještě další výhodu v podobě možnosti použití asynchronních přenosů dat, což programátorovi poskytne možnost překrytí datových přenosů výpočtem. V takovém případě se naopak vyplatí rozdělit přenosy na menší balíky dat, aby se přenesla první část zpracovávaných dat, spustil se výpočet a paralelně s ním přenos další části dat. V ideálním případě s poslední generací architektury Fermi je možno překrýt přenosy dat v obou směrech, čili současného přenosu dat z CPU na GPU a obráceně. To by mohlo přinést zrychlení i v případě, kdy nejvíce času aplikaci zabere přenos dat nežli výpočet samotný.

- Práce s paměťovou hierarchií je velmi zásadní, může zde dojít k největším propadům výkonu. Klasický vzor v těchto typech aplikací spočívá ve zkopírování všech dat z globální paměti do rychlé sdílené paměti případně do registrů a pracovat s nimi zde. Po dokončení veškeré práce se výsledek zapíše zpět do globální paměti. Dále by měl být přístup do globální paměti zarovnaný, první vlákno by mělo číst první prvek vstupu atd. bez rozestupů i za cenu načtení přebytečných dat.

- Nemělo by docházet k divergenci toku instrukcí mezi vlákny WARP bloku, tzn. při větvení typu if-then-else by měla všechna vlákna vyhodnotit podmínku stejně. Došlo by k rapidnímu poklesu výkonu tím, že by se muselo vykonávání kódu vláken serializovat.
- Přestože je přístup do sdílené paměti multiprocesoru rychlý, mohlo by také docházet k serializaci přístupu kvůli přístupu do stejných bank, jak bylo naznačeno v sekci 3.1.2.
- Je třeba mít dostatečný počet aktivních vláken a jejich počet by měl být násobek velikosti WARP bloku.

3.2 OpenCL

Open Computing Language [18] je první otevřený průmyslový standard pro paralelní programování a obecné výpočty nejen na grafickém hardware. Poskytuje jednotné programovací rozhraní pro vývojáře aplikací běžících na klasických procesorech, grafických kartách, procesoru IBM Cell a jiných paralelních architekturách. OpenCL specifikace byla navržena skupinou The Khronos ^{TM}Group v polovině roku 2008, přesto má své počátky u firmy Apple. Tento návrh byl velmi rychle schválen a koncem téhož roku byla vydána první specifikace. V čase psaní této práce je aktuální verze 1.1. Na této specifikaci se dále podílela většina významných hráčů na poli informačních technologií, např. Intel, AMD, nVidia, IBM, Sun, Samsung, Sony a spousta dalších.

OpenCL si je hodně podobné s rozhraním CUDA. Také specifikuje vrstvu ikdyž více abstraktního hardwarového zařízení, k němu patřičné rozhraní pro ovladač, dále pak rozhraní a jazyk pro programování. Jednoduchost, s jakou byly navrženy, usnadňuje jeho implementaci na různých HW platformách. OpenCL je také nezávislé na operačním systému. Programovací jazyk je podobně jako v případě CUDA založený na jazyku C, konkrétně standardu z roku 1999. Obsahuje standardní hlavičkové soubory, preprocesorové direktivy, navíc obsahuje vestavěné skalární i vektorové datové typy, kvalifikátory adresního prostoru a přístupových práv, kernel funkce, množinu základních funkcí, kterou musí každá implementace mít a množinu speciálních funkcí, rozšíření, které nejsou nutné pro všechny platformy. Jazyk je však omezen o určité konstrukce (např. ukazatele na ukazatele, ukazatel na funkce, rekurzivní funkce).

3.2.1 Architektura

Běh OpenCL programu se dělí na dvě základní části. Tou první je jádro (kernel) samotného výpočtu prováděné na libovolném OpenCL zařízení, kterých může být v systému několik a klasické aplikační části spouštěné z hostitelského programu, který používá rozhraní ke komunikaci s dostupnými zařízeními, přenosu dat a spuštění samotného výpočtu. Hostitelský program tedy vytváří kontext, v němž jádro běží.

Při spouštění jádra výpočtu se definuje jeho rozměr v třírozměrném prostoru, neboli kolik pracovních jednotek (work-items) bude spolupracovat na řešení problému. Pracovní jednotky tvoří pracovní skupiny (work-groups), jež jsou obdobou bloků vláken z modelu CUDA. Tyto skupiny mají v dané dimenzi svoje unikátní označení. Pracovní jednotky jsou identifikovány buď unikátním označením v rámci pracovní skupiny, nebo globálním identifikátorem v rámci všech pracovních jednotek.

Hostitelský program při vytváření kontextu vybírá podporované OpenCL zařízení, kterých může být několik. To je velká výhoda proti CUDA, jelikož OpenCL může využít i jiná

dostupná zařízení než jen grafické karty. Dále v rámci kontextu vybírá zdrojový objekt, který obsahuje kód výpočtu, funkci, která se bude provádět a vytváří datové objekty se vstupními a výstupními parametry. S kontextem se zachází pomocí příkazových front. Příkazy se uvnitř příkazové fronty vykonávají asynchronně vzhledem k hostitelskému programu. Dále pak se mohou příkazy uvnitř fronty provádět buďto v zadaném pořadí (tak jak byly přidávány do fronty) nebo mimo pořadí (ve skutečnosti jsou příkazy spouštěny v pořadí, jak byly zadány, ale nečeká se na dokončení předchozích volání a jakákoli potřebná synchronizace je v rukou programátora).

OpenCL podporuje jak datově paralelní programovací model tak úlohově paralelní. Datově paralelní model hraje větší roli, rozdělit data na menší nezávislé bloky a provést na nich danou operaci. Vzhledem k adresování pracovních jednotek se dá konstatovat, že existuje mapování 1:1, které jednotce přísluší která data. Nicméně tento model není striktní, jedna pracovní jednotka může zpracovávat více elementů vstupních dat. V kontextu úlohově paralelního modelu se počítá s tím, že se nepoužívá adresování pracovních jednotek, nýbrž vektorové datové typy podporované daným zařízením a zařazením více úloh do příkazové fronty.

3.2.2 Paměťový model

Hierarchie pamětí je prakticky identická s model architektury CUDA někde jen s jiným názvem. Na druhou stranu je tato hierarchie poněkud abstraktnější, jelikož se nemapuje jen na grafické čipy natož ty společnosti NVIDIA. Opět existuje:

- Globální paměť, která je přístupná všem vláknům i hostitelskému systému, umožňuje čtení i zápis všem pracovním jednotkám a podle schopností daného zařízení může být přístup opatřen vyrovnávací pamětí.
- Paměť konstant je oblast paměti inicializovaná hostitelským programem před spuštěním jádra výpočtu (kernel), které již nemá možnost tato data nijak měnit.
- Lokální paměť je přístupná pouze pracovním jednotkám v rámci skupiny, prakticky tak odpovídá rychlé sdílené paměti multiprocesoru architektury CUDA a je tedy určena k propagaci informací mezi pracovními jednotkami v rámci skupiny.
- Privátní paměť, která nemá přímou obdobu v CUDA, je pamětí přístupnou pro čtení i zápis pouze jedné pracovní jednotce. V případě skalárních hodnot by se měly na GPU použít registry multiprocesoru.

Pro práci s pamětí na grafických zařízeních platí stejné podmínky a zásady, které by se měly dodržovat pro dosažení optimální rychlosti jako v podkapitole 3.1.3.

Kapitola 4

Aplikační programové rozhraní pro kryptoprocesory

Tato kapitola pojednává o dostupných rozhraních, která jsou určena ke spolupráci s ovladači speciálních kryptografických zařízení. V kapitole je uvedena stručná historie několika hlavních rozhraní spolu s jejich návrhy.

4.1 The Linux Kernel Cryptographic API

Historie nativního kryptografického rozhraní v Linuxu (dále jen CryptoAPI) [11] sahá do roku 2002. Jako v případě většiny nově vznikajících rozhraní nastala potřeba jednotného kryptografického rozhraní s příchodem specifikace IPsec (*IP Security*). Předchozí implementace specifikace IPsec byla v Linuxu kritizována. Potřebovala podporu kryptografie v jádru, stejně tak rostla všeobecná potřeba tohoto rozhraní.

Přestože je rozhraní spjato s příchodem IPsec specifikace, návrh počítal se všeobecným použitím, stejně tak s možným šifrováním souborů, souborových systémů nebo také s generátorem pseudonáhodných čísel pomocí zařízení **/dev/random**. Požadavky při návrhu byly následující:

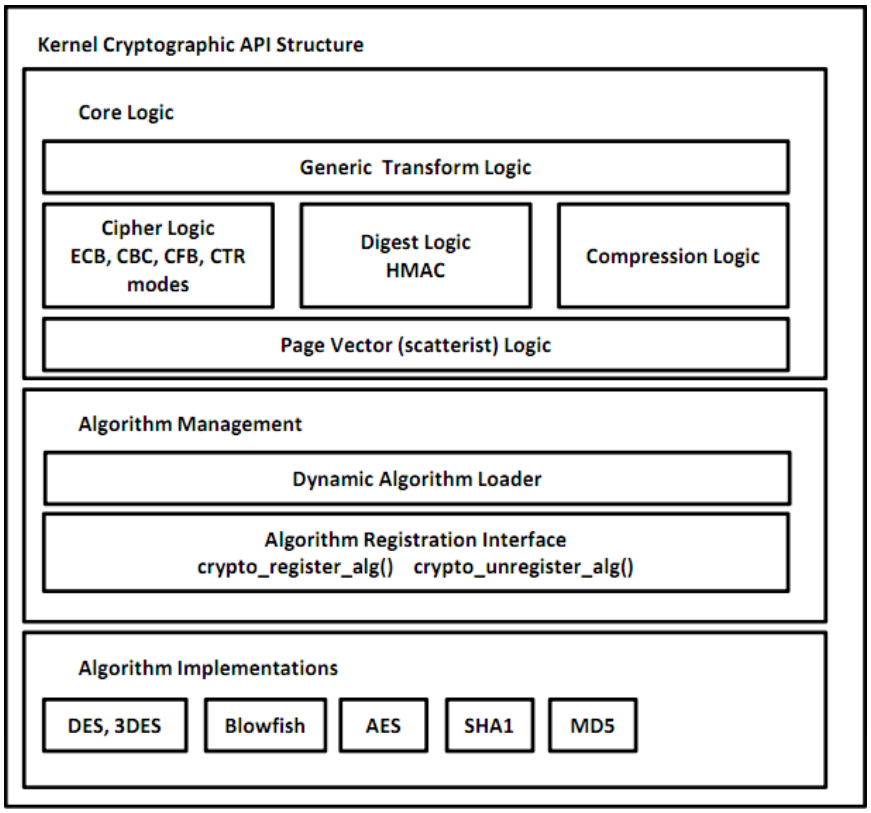
- Možnost provádět transformace na úrovni stránek paměti, což umožňovalo hlubší integraci s dalšími podsystémy jádra (síťový zásobník) a znamenalo také menší režie díky nižší potřebě kopírování dat mezi podsystémy.
- Jednoduchost a flexibilita měla umožňovat dynamické nahrávání nových algoritmů jako modulů jádra.
- V případě existence více implementací stejného algoritmu měla být automaticky vybrána ta výkonnější, např. optimalizovaná ASM verze.
- Podpora asymetrické kryptografie.
- Podpora kryptografických akcelerátorů a síťových karet s podporou IPsec.

CryptoAPI rozlišuje tři druhy algoritmů:

1. Hašovací funkce za účelem podpisu jako např. MD5, SHA1 nebo také HMAC s použitím libovolné hašovací funkce.

2. Šifry byly z počátku výhradně symetrické (AES, 3DES, Blowfish), jelikož asymetrické algoritmy byly časově náročné a nehodily se tak přímo do jádra. Rozlišovaly se tedy jen blokové a proudové šifry a byly nabídnuty různé režimy blokových šifer (ECB, CBC). Nyní jsou již podporovány i asymetrické šifry a to jak v synchronním tak asynchronním řežimu.

3. Kompresní algoritmy se často používaly ve spojení se šifrovacími algoritmy, aby bylo těžší odhalit slabiny nešifrovaného textu a navíc urychlit šifrování, jelikož data po kompresi budou kratší.



Obrázek 4.1: Struktura CryptoAPI

4.2 OpenBSD Cryptography Framework

Projekt OpenBSD Cryptographic Framework, dále jen OCF, byl veřejně prezentován v roce 2003, kdy vyšla publikace s jeho návrhem [6], nicméně práce započaly o tři roky dříve. Cílem bylo nabídnout jednotný a efektivní přístup k zařízením, která se začínala objevovat na trhu a podporovala akceleraci vybraných kryptografických algoritmů. Do té doby se podobná rozhraní v operačních systémech nevyskytovala. OCF je tedy asynchronní služba virtualizační vrstvy, která je implementována přímo v jádru operačního systému. Zajišťuje efektivní a jednotný přístup k funkcionalitě dostupných prostředků jak pro potřeby podsystémů jádra samotného, tak i klientských aplikací pomocí abstraktního rozhraní, jenž odstraňuje nutnost znát detaily o daném zařízení uvnitř aplikace a to vše bez většího dopadu na výkonnost systému.

OCF rozlišuje dva druhy algoritmů, symetrické a asymetrické. Symetrické algoritmy zahrnují např. DES, AES, ale i kompresní algoritmy a hašovací funkce. Tyto algoritmy používají koncept relace, kdy se předpokládá zpracování dat po větších balících a je zde snaha shlukovat malé požadavky do větších bloků dat. Jsou tedy ukládány do fronty pro pozdější zpracování. To na jednu stranu zlepšuje propustnost zařízení za cenu zvýšení latence vyřizování požadavků, což nemusí být vhodné u některých síťových protokolů. V případě asymetrických algoritmů se podobné chování nevyskytuje, nedochází k ukládání do vyrovnávací paměti a všechny operace jsou tím pádem synchronní.

OCF představuje dvě aplikačně programová rozhraní, první pro klientské aplikace a jiné části systému právě pro využití schopností daného zařízení. Druhé rozhraní slouží pro ovladače akcelerátoru k registraci sebe sama uvnitř systému, jaké algoritmy a režimy podporuje, případně další schopnosti zařízení jako např. generování pseudonáhodných čísel. Ovladač je také schopen se nezávisle ze systému odregistrovat a to i na úrovni jednotlivých algoritmů. Tato situace by mohla nastat například při odebrání zařízení ze systému (přídavná karta v PCMCIA slotu). V takovém případě by existující relace byly přesunuty na jiná zařízení. Dále se registrují funkce zpětného volání (*callbacks*), které budou používány OCF k inicializaci zařízení, vytváření a ukončování relací.

Průběh vytvoření a zpracování požadavků vypadá zhruba následovně [7]:

1. Klient si sestaví požadavek na zpracování. Nejprve zavolá funkci *crypto_newsession()* pro vytvoření nové relace, vybere algoritmy, módy operace, klíče, atd.

2. Po vytvoření požadavku a relace volá funkci *crypto_dispatch()*, která vloží požadavek do fronty a upozorní tím jádro.

3. Jádro odstraní požadavek z fronty a zavoláním funkce *crypto_invoke()* předá požadavek patřičnému ovladači.

4. Ovladač zpracuje daný požadavek ať už hardwarově nebo programově.

5. Po dokončení zpracování ovladač zavolá funkci *crypto_done()*, čímž upozorní jádro na dokončení a předá vyřízený požadavek do návratové fronty.

6. Jádro odstraní požadavek z návratové fronty a zavolá zpětnovazební funkci, která byla asociována s tímto požadavkem.

7. Pokud již klient nebude vytvářet další požadavky, pomocí *crypto_freesession()* ukončí relaci.

Důležité je také zmínit, že pokud systém neobsahuje hardwarové akcelerátory některého nebo všech algoritmů, vždy je přítomen programový pseudo-ovladač, jež bude použit v posledním případě. Dále byla celá funkčnost OCF zabalena a zpřístupněna aplikacím jako ovladač zařízení **/dev/crypto**.

Dalšími význačnými rysy jsou podpora rozložení zátěže, migrace relací a řetězení algoritmů. OCF umí využít více kryptografických akcelerátorů v jednom systému a umí mezi ně distribuovat požadavky v závislosti na počtu relací, které dané zařízení obsluhuje. Přestože přítomnost akcelerátorů nemusí nutně vést k rapidnímu zvýšení výkonnosti v dané oblasti, odlehčí se tak hlavnímu CPU a zlepší se alespoň výkon a celková odezva systému a ostatních aplikací. Platí také to, že výkon stoupá s rostoucí velikostí požadavků, proto se ukládají do fronty a velmi malé požadavky je lepší obsluhovat okamžitě programově. Řetězením algoritmů je míněno například šifrování následované kontrolou integrity. Poslední součástí jsou

ovladače pro vybrané akcelerátory, např. značky Broadcom, Hifn.

Návrh tohoto rozhraní se ujal, systém byl přenesen na NetBSD, FreeBSD a následně na Linux. Přenos na FreeBSD nebyl však zcela triviální a narazil na několik problému:

- OCF bylo napsáno pro systém s nepreemptivním jádrem na rozdíl od FreeBSD a bylo tak nutno přepsat jádro OCF.
- Rozdíly ve vstupně/výstupním podsystému znamenaly nutnost přepsat části ovladačů.
- Výkonnostní nedostatky.

Přenos na plně preemptivní jádro FreeBSD byl vyřešen pomocí zamykání datových struktur jádra, úpravou a reorganizací kódu jádra, aby byla zachována konzistence a nedocházelo k uváznutí. Cena použití zámků není ve výsledku příliš velká. Více práce zabral přenos ovladačů zařízení, některé výrobci utajované aspekty daného zařízení musely být získány pomocí reverzního inženýrství. Dále bylo třeba zachovat rozhraní ovladačů, aby mohly být ovladače sdíleny mezi FreeBSD a OpenBSD. Největší práce byla odvedena na odladění výkonnosti. OCF se chlubilo poskytováním přibližně 95% teoretického maxima nabízeného zařízením. Práce prezentující přenos na FreeBSD [7] však uvádí, že nebyla použita příliš vhodná metrika. V průběhu zpracování požadavku docházelo na OpenBSD k několikanásobnému přepínání kontextu mezi vlákny jádra, což bylo na FreeBSD maximálně omezeno použitím programového přerušení nebo okamžitým zpracováním například velmi malých požadavků, které se daly programově rychle zpracovat. V textu výše bylo také naznačeno, že OCF mohlo mít potíže s latencí vyřizování požadavků. Tato situace byla ošetřena přidáním atributu, který určoval, zda daný požadavek má být vyřízen okamžitě nebo může být přidán do fronty. Celkově po úpravách bylo za určitých podmínek dosaženo až dvojnásobného zrychlení na identické konfiguraci počítače. Tabulka 4.1 ukazuje ony výsledky zvýšení výkonnosti implementace ve FreeBSD oproti implementaci OCF v OpenBSD při šifrování algoritmem 3DES.

Velikost operandu	Hifn 7951			Hifn 7811			Broadcom 5822		
	OBSD	FBSD	%	OBSD	FBSD	%	OBSD	FBSD	%
8	1,2	1,9	58	1,2	1,9	58	1,5	3,0	100
16	2,5	3,8	52	2,5	3,8	52	3,1	6,0	94
32	4,6	7,6	65	4,6	7,6	65	6,1	11,7	92
64	9,2	14,8	61	9,1	15,3	68	11,8	22,6	92
128	16,2	23,1	43	18,6	28,5	53	22,7	41,4	82
256	27,3	36,2	33	32,9	49,7	51	43,0	74,9	74
512	41,5	51,2	23	53,9	75,7	40	72,9	128,0	76
1024	54,0	63,1	17	80,4	99,3	24	119,1	205,5	73
2048	65,9	71,3	8	106,5	120,6	13	199,6	282,1	41
4096	71,5	76,4	7	124,1	136,5	10	282,3	346,9	23
8192	76,8	78,4	2	138,9	145,1	4	333,1	380,2	14

Tabulka 4.1: Zrychlení FreeBSD vůči OpenBSD při 3DES šifrování, převzato z [7]

4.3 OCF–Linux

Projekt OCF-Linux [10] staví právě na práci portu OCF na FreeBSD kvůli úpravám, které byly provedeny. Přináší tak plně asynchronní hardwarovou a programovou akceleraci pro GNU/Linux. Poskytuje akceleraci např. projektům OpenSSL, OpenSSH pro algoritmy 3DES, AES, MD5, SHA a dalších včetně zařízení **/dev/random** jako generátoru pseudonáhodných čísel. Projekt se chlubí až sedminásobným zrychlením oproti holému OpenSSL. Tabulka 4.2 převzata z [10] ukazuje zrychlení propustnosti HW akcelerátorů proti čistě programovému řešení (OpenSSL) a OCF bez přítomnosti HW akcelerátorů při šifrování algoritmem AES ve 128 bitovém režimu v módu CBC s různými velikostmi paketu.

Zařízení	16 B	64 B	256 B	1024 B	2048 B
čistě SW řešení	4310.24k	4665.73k	4769.11k	4824.06k	4866.05k
OCF se SW ovladačem	634.60k	1757.17k	3105.63k	3837.95k	4087.81k
Intel Xscale IXP425	315.41k	1178.61k	3892.92k	8756.07k	10856.65k
Safenet 1141	202.33k	788.55k	2907.59k	8517.74k	12694.32k
Hifn7956	299.59k	1166.92k	3992.46k	10735.10k	14707.49k

Tabulka 4.2: Výkon šifrování algoritmem AES 128b, CBC

4.4 CryptoDev for Linux

Další projekt který se nechal inspirovat OCF a jeho portem z FreeBSD [9]. Zachovává rozhraní a sémantiku OCF pro práci se zařízením **/dev/crypto**, ale kvůli velkým rozdílům v architektuře a kódu Linux a BSD je ovladač zařízení napsán zcela od počátku. Dalším rozdílem je použití nativního CryptoAPI přítomného v Linuxu pro implementaci. Projekt je dostupný jako samostatný modul jádra, pro jeho použití je tedy nutné použít patch na jádro, zkompilovat jej a povolit při konfiguraci jádra. Dále je také dostupný patch, který zpřístupní funkcionalitu se všemi výhodami a nevýhodami pro knihovnu OpenSSL.

4.5 OpenSSL

Knihovna OpenSSL se snaží o vývoj robustní, plně vybavené Open Source sady nástrojů, která implementuje protokoly **Secure Socket Layer** (SSL v2/v3), **Transport Layer Security** (TLS v1) a také širokou škálu kryptografických algoritmů pro všeobecné použití [17].

S verzí 0.9.6, která vyšla roku 2000, byla představena koncepce kryptografických modulů v podobě tzv. *ENGINE* objektů. Tyto objekty se chovají jako kontejnery pro implementace různých algoritmů a umožňují nahrávání těchto objektů za běhu aplikace. Díky nim je možno si zaregistrovat vlastní implementace algoritmů a využít je tak nejen v rámci vlastních aplikací, nýbrž i v rámci celé knihovny a systému.

Koncept *ENGINE* objektů podporuje statickou nebo dynamickou integraci s OpenSSL knihovnou. Statickou verzi je potřeba přeložit přímo se zdrojovými kódy knihovny a zahrnout při sestavování. Knihovna obsahuje několik zabudovaných statických modulů pro podporou akcelerátorů např. firem Cluster Labs nebo IBM. Další možností je dynamická verze, kdy si programátor vytvoří dynamickou knihovnu (.dll, .so) implementující požadované rozhraní a funkcionalitu. O knihovně je ještě třeba dát vědět samotné OpenSSL

knihovně globálně formou konfiguračního souboru, případně vestavěným aplikacím pomocí parametrů programu. V případě použití uživatelských aplikací a nepřítomnosti konfiguračního souboru knihovna definuje rozhraní pro nahrávání modulů případně výběr pouze specifických algoritmů daného akcelerátoru.

OpenSSL dělí kryptografické algoritmy, které mohou být podporovány hardwarovými akcelerátory, na šifrovací (převážně symetrické), hašovací a potom konkrétní metody pro vybrané asymetrické šifry (RSA, DSA), generátor pseudonáhodných čísel a asymetrických klíčů. Pro tyto skupiny algoritmů je možno registrovat vlastní implementace.

Kapitola 5

Implementace

V této kapitole bude čtenář seznámen s implementačními detaily samotného algoritmu AES, implementovaných blokových režimů a specifickými aspekty CUDA a OpenCL implementace. Na závěr se kapitola zabývá integrací GPU verze do knihovny OpenSSL, aby mohla využívat grafický akcelerátor k šifrování AES.

5.1 Algoritmus

Implementace algoritmu AES je založena na implementaci dostupné v knihovně OpenSSL. Algoritmus nemá klasickou strukturu jako je naznačena v kapitole 2.4, ale využívá ke zrychlení čtyři vyhledávací tabulky s 256 hodnotami velikosti 32bitů podle rovnice 2.2. Na rozdíl od OpenSSL verze bylo potřeba hodnoty v tabulkách přeuspořádat, konkrétněji reverzovat pořadí bajtů. To bylo potřeba kvůli datovému typu použitému pro uchování 128 bitů vstupu. Během přetypování docházelo k přeuspořádání dat a výsledky šifrování neodpovídaly referenčním řešením. Nedocházelo sice k potížím s následným dešifrováním GPU verzí algoritmu, ale jakákoli jiná verze by nebyla schopna data dešifrovat.

Umístění vyhledávacích tabulek v paměti grafické karty je zásadní z hlediska výkonnosti. Výchozím umístěním je paměť konstant. Jelikož přístup do tabulek je prakticky náhodný, každé vlákno může číst různá místa paměti, což paměti konstant z hlediska výkonnosti nevyhovuje, jsou před začátkem samotného šifrování tabulky překopírovány do rychlé sdílené paměti. Je to prakticky jediná rozumná volba, přestože i zde dochází podle práce [5] přibližně v 35% případů ke konfliktům přístupu do bank sdílené paměti. Výchozí umístění v paměti konstant je zde proto, aby nemuselo docházet před každým spuštěním šifrování ke kopírování tabulek do globální paměti a následně do sdílené. Platnost dat v paměti konstant je po dobu běhu celé aplikace, což zcela vyhovuje.

Dalším implementačním rozhodnutím je počet vláken šifrující jeden 128bitový blok dat. AES zde nabízí poměrně jednoduchou paralelizaci. Data je možno rozdělit do čtyř bloků po 32 bitech a každý zpracovávat jedním vláknem. Jednodušší přístup je zpracovávat celý vstupní blok jedním vláknem a paralelizaci řešit až na úrovni blokových režimů. Oba přístupy mají své výhody. Jemnější paralelismus, kdy jeden blok vláken zpracovávají čtyři vlákna, je výkonnější, pokud se šifruje menší množství vstupních dat. Nevýhodou je nutnost sdílet mezivýsledky mezi vlákny, což implikuje použití sdílené paměti. Hrubší paralelismus na úrovni blokových režimů je výkonnější pro větší vstupní data a jelikož vlákna nepotřebují sdílet mezivýsledky, mohou být umístěny v registrech. Rozdíl výkonnosti mezi registry a sdílenou pamětí je specifický pro rozhraní CUDA, proto je více informací na toto téma

uvedeno níže v sekci 5.3.

S mírou paralelismu dále souvisí také to, kolik vláken bude spuštěno na jednom multiprocesoru a kolik budou jedno respektive čtyři vlákna zpracovávat bloků dat. Pro menší vstupní data bude lepší nižší počet bloků zpracovávaných jedním vláknem, aby byla vytvořena práce pro více multiprocesorů a ta následně provedena paralelně. Pro větší množství dat je lepší šifrovat více bloků, aby se například odstranila režie spojená s kopírováním vyhledávacích tabulek do sdílené paměti. Konkrétnější hodnoty konfigurace spouštění naleznete v kapitole 6.

Další součástí algoritmu AES je také tvorba klíče pro všechna kola algoritmu. Proces expanze uživatelova klíče je iterační, čistě sekvenční algoritmus a proto by nemělo smysl přenášet tuto implementaci na grafickou kartu. Příprava klíče probíhá na CPU, výsledek je následně zkopírován do paměti grafické karty. Volba umístění klíče v paměti grafické karty padla na paměť konstant obou použitých API pro programování GPU. Další alternativou byla rychlá sdílená paměť nebo paměť textur v případě rozhraní CUDA. Paměť konstant má dostatečnou velikost, všechna vlákna přistupují v daném kroku ke stejnému bloku paměti, je možno použít rozhlašování a přístup je tedy velmi rychlý, stejně rychlý jako v případě sdílené paměti. Při použití rychlé sdílené paměti nedocházelo k žádnému urychlení, spíše naopak, jelikož bylo potřeba data nejprve zkopírovat do sdílené paměti. Kopírování klíče ani nemohou provádět všechna vlákna a docházelo k mírné divergenci toku instrukcí jednotlivých vláken, čili lehkému snížení výkonu. Následný přístup pak byl stejně rychlý a projevila se tak pouze režie spojená s kopírováním. Navíc množství sdílené paměti je dost omezené. Použití paměti textur v případě CUDA rozraní také nepřineslo žádné pozitiva, navíc je specifická pouze pro CUDA rozhraní a proto zůstala volba na paměti konstant.

Korektnost algoritmu byla testována s využitím příkladů a ukázek ze standardu [1]. Jsou zde uvedeny příklady mezivýsledků po jednotlivých kolech algoritmu pro šifrování i dešifrování. Implementovaná verze těmto ukázkám odpovídala. Dále byly vytvořeny zašifrované testovací soubory pomocí OpenSSL verze vytvořené na CPU. Dešifrování GPU verzí pak vracelo korektní výstupy. Samozřejmě toto fungovalo i obráceně, šifrování na GPU a dešifrování na CPU.

5.2 Blokové režimy

Implementace blokových režimů šifry byly poměrně přímočaré. Implementovány byly režimy Electronic Code Book (ECB), Counter (CTR) a dešifrování v režimu Cipher-Block Chaining (CBC). Šifrování v režimu CBC je sekvenční záležitost, proto se spoléháno na čistě programové řešení prováděné procesorem. Více o režimech viz podkapitola 2.3. ECB režim je vlastně implicitní. Jak jsou data umístěna v paměti, tak se šifrují. Pro režim CTR je potřeba předat aplikaci výchozí hodnotu čítače. Ta je předávána hodnotou jako konstanta a překladač by měl zajistit její efektivní umístění v paměti konstant. Toto umístění je ideální, jelikož všechna vlákna čtou stejnou hodnotu. Nejdříve je do registru uložena výchozí hodnota čítače, pak je tato hodnota patřičně zvýšena a data následně šifrována.

Nejproblematičtější je dešifrování v režimu CBC. Pro jedno dešifrování je třeba načíst dvě vstupní hodnoty, místo jedné. Kvůli tomu je potřeba pro dešifrování 256 bloků načíst 257 bloků. V případě, kdy jedno vlákno zpracovává více vstupních bloků, jsou přednačtena všechna další vstupní dat, čili dalších 256 bloků, aby nedocházelo k divergenci mezi vlákny. Přednačtená data se při dalším šifrování znovu využijí. Pro zjednodušení kódu je nejprve před vstupní data připojen inicializační vektor. Aby nedocházelo k redundantním přístupům do globální paměti pro vstupní data, jsou nejprve bloky uloženy do sdílené paměti včetně

přednačtených. Kvůli tomu je zde, na rozdíl od jiných režimů, potřeba dodatečné explicitní synchronizace vláken. Následně se data dešifrují a zkombinují s předchozí hodnotou.

5.3 CUDA

Tato podkapitola se zabývá aspekty implementace specifickými pro rozhraní CUDA. V předchozím textu byl naznačen problém rozdílu ve výkonnosti mezi registry a sdílenou pamětí. Materiály společnosti NVIDIA (např. [12]) obecně uvádějí, že rychlost přístupu do sdílené paměti je bez paměťových konfliktů stejně rychlá jako práce s registry. To však vyvrací práce [15]. Ta tvrdí, že přístup do sdílené paměti je na starších architekturách grafických čipů až třikrát pomalejší než práce s registry, v případě architektury *Fermi* je přístup dokonce až šestkrát pomalejší. Proto je lepší omezit i práci se sdílenou pamětí a maximalizovat využití registrů.

Dalším aspektem specifickým pro prostředí CUDA je asynchronní rozhraní využívané především pro překrytí výpočtu a kopírování dat. To bylo krátce zmíněno v části 3.1.3. Je ho zde využito. První výhodou je to, že operace nejsou blokující, procesor se z volání funkcí spouštějících výpočet nebo datové přenosy na grafické kartě okamžitě vrací, nečeká se na jejich dokončení. Je tak možno spustit jednu operaci, během jejího provádění naplánovat několik nebo všechny další operace a pak jen počkat na jejich dokončení. Tím se odstraní alespoň nepatrné prodlevy mezi voláním jednotlivých funkcí, což se projeví hlavně při šifrování menších vstupních dat. Pro dosažení překrytí výpočtu a kopírování dat je potřeba vstupní data rozdělit do menších bloků a každý samostatně šifrovat, respektive dešifrovat s ohledem na daný režim blokové šifry. To, na jak velké bloky vstupní data rozdělit, záleží na celkové velikosti šifrovaných dat. Obecně platí, že čím je velikost bloku větší, tím efektivněji a rychleji probíhá kopírování po sběrnici PCI-Express. Nejprve se vytvoří několik CUDA stream objektů (konkrétně čtyři) typu *cudaStream_t*, což je prováděno v inicializační funkci. Pro ně se následně naplánují potřebné operace, jejichž provádění se spustí později. Operace naplánované v rámci stejného CUDA stream objektu jsou prováděny ve stejném pořadí, jak byly naplánovány, chová se tedy jako fronta. Jelikož jsem neměl k dispozici grafickou kartu, která by podporovala překrytí datových přenosů, na které bych mohl kód otestovat, zabýval jsem se převážně překrytím datových přenosů s výpočtem. Přibližný pseudokód použitý pro překrytí je následující:

```
void cipher( unsigned char* dst, const unsigned char* src )
{   // variable declaration and initialization, ...
    copy_memory_to_gpu( dev_src, src, current_size, current_stream );
    for (int i = 0; i < nreps; ++i)
    {
        next_steam = (current_stream + 1) % STREAM_COUNT;

        call_aes_kernel( dev_dst, dev_src, current_steam );
        copy_memory_to_gpu( dev_src, src + offset,
            next_size, next_stream );
        copy_memory_from_gpu( dst + offset, dev_dst,
            current_size, current_stream);

        current_stream = next_stream;
    }   // ...
}
```

Nejprve se zkopíruje část vstupních dat dané velikosti do globální paměti na grafické kartě pro aktuální CUDA stream. Následně se v cyklu určí další stream. Naplánuje se spuštění šifrování na GPU pro předem zkopírovaná data. Následně se naplánuje kopírování vstupních dat pro další stream do globální paměti a kopírování výsledku šifrování pro aktuální stream z globální paměti. To se opakuje v cyklu pro další dvojici stream objektů.

Aby bylo možné asynchronní rozhraní plně využít, je potřeba alokovat paměť na CPU pro vstupní a výstupní data tak, aby nebylo možné odkládat stránky s daty například na pevný disk, tzv. *page-locked* paměť. Výrazně se tím také zrychlí přenosová rychlost po sběrnici PCI-Express mezi pamětí grafické karty a CPU. NVIDIA pro alokaci nabízí funkci *cudaHostAlloc*. Pro usnadnění práce s poli a hlavně vektory jsem implementoval alokátor paměti tak, aby se dal použít se standardním kontejnerem std::vector jazyka C++. V případě, že se použije asynchronní rozhraní s klasicky alokovanou pamětí, nedojde k žádné formě překrytí, ale vše funguje jako by se použil jeden CUDA stream objekt. Výkon bude poté nepatrně nižší, protože se budou po sběrnici kopírovat menší bloky dat.

5.4 OpenCL

Rozhraní OpenCL je velmi podobné rozhraní CUDA. Jelikož vývoj a testování probíhalo na grafických kartách společnosti NVIDIA, platí většina specifických aspektů z podkapitoly 5.3 i pro toto rozhraní. Obě rozhraní poskytují podporu pro asynchronní rozhraní, v OpenCL se pouze použije místo objektů *cudaStream_t* více objektů typu *cl_command_queue* a *cl_kernel*. Opět platí, že stránky paměti se nesmí odkládat. Pro vytváření takovéto paměti se použije funkce *clCreateBuffer* s parametrem *CL_MEM_ALLOC_HOST_PTR*. Zde záleží na konkrétní dodávané implementaci této funkce v rámci OpenCL, jestli opravdu alokuje paměť označenou jako *page-locked*.

5.5 OpenSSL

Další součástí této práce byla integrace grafickým čipem akcelerované verze algoritmu AES s knihovnou OpenSSL. Pro integraci je třeba využít *ENGINE* objektů a implementovat rozhraní, které knihovna definuje. Toto rozhraní lze rozdělit na dvě části.

- První se zabývá vytvořením datové struktury typu *EVP_CIPHER*, která nese konkrétní specifikace dané symetrické šifry, tedy AES. Jedná se například o název algoritmu, jeho konkrétní variantu, tedy velikost šifrovacího klíče, konkrétní blokový režim, adresu funkce, která se stará o úpravu uživatelského klíče na expandovaný. Dále ještě přidá adresu funkce, která bude spouštět šifrování pro předaná vstupní a výstupní data.

- Druhá část se stará o zpřístupnění konkrétních informací o podporovaných variantách šifer do *ENGINE* objektu, je tedy potřeba zajistit jeho vytvoření, inicializaci a registraci. Po vytvoření objektu se volá inicializační funkce, která nastaví řetězec s názvem a identifikátorem *ENGINE* objektu. Dále přidá adresu inicializační funkce, pokud je jí potřeba (např. pro inicializaci globálních proměnných), adresu úklidové funkce a ještě adresu funkce, která propojí dříve vytvořené objekty typu *EVP_CIPHER* konkrétních variant šifry s identifikátory používanými knihovnou OpenSSL. Nakonec je třeba přidat nově vytvořený a inicializovaný objekt do seznamu všech dostupných *ENGINE* objektů, které bude knihovna aplikacím nabízet.

Knihovna v současné verzi (1.0.0.d) umožňuje akcelerovat algoritmus AES pouze v režimech ECB a CBC. Nabízí ještě možnost registrovat režimy *Cipher feedback (CFB)* a *Output feedback (OFB)*, ty však nebyly implementovány ani v práci rozebírány. Bohužel zde chybí podpora pro režim čítače. Podle neoficiálních informací dostupných na různých fórech se tato podpora ani nechystá.

Dále je možno podporu přeložit a staticky sestavit přímo se zdrojovými kódy OpenSSL nebo zkompilovat jako samostatnou dynamickou knihovnu a pomocí konfiguračního souboru dynamicky nahrávat. Konfigurační soubor je uveden zde:

```
openssl_conf = openssl_init

[ openssl_init ]
engines = engine_section

[ engine_section ]
cuda_engine = cuda_engine_section

[ cuda_engine_section ]
engine_id = CUDA_AES                    # Override default name
dynamic_path = /usr/lib/libcudaAES.so # Load engine
default_algorithms = ALL                # Supply all default algorithms
init = 1                                # Initialize the ENGINE
```

5.6 AES-NI

Pro následné srovnání výkonu grafického čipu s výkonem procesorů, byla implementována ještě jedna varianta algoritmu AES využívající rozšiřující instrukční sadu *AES-NI*, kterou nabízejí vybrané procesory společnosti Intel posledních generací. Podporu mezi na trhu dostupnými procesory nabízejí procesory postavené na mikroarchitektuře *Nehalem* a *Sandy Bridge*. Společnost AMD slibuje podporu této instrukční sady s příchodem mikroarchitektury *Bulldozer*. Jedná se o sadu několika instrukcí, které významným způsobem akcelerují šifrování ve srovnání s klasickým řešením. Následuje jejich stručný přehled:

Instrukce	*Popis*
AESENC	Provádí jedno standardní šifrovací kolo algoritmu.
AESENCLAST	Provádí poslední šifrovací kolo algoritmu.
AESDEC	Provádí jedno standardní dešifrovací kolo algoritmu.
AESDECLAST	Provádí poslední dešifrovací kolo algoritmu.
AESKEYGENASSIST	Používá se vytváření šifrovací klíče.
AESIMC	Provádí konverzi šifrovací klíče na dešifrovací.

Tabulka 5.1: Intel®Advanced Encryption Standard (AES) New Instructions (AES-NI)

Implementace je založena na ukázkových příkladech a *Intel AESNI Sample* knihovně [2]. Akcelerovány jsou stejné režimy jako v případě GPU verze, navíc ještě podporuje akceleraci šifrování v režimu CBC.

5.7 Testovací aplikace

V rámci testování algoritmu a blokových režimů byly vytvořeny několik verzí aplikace. Jedna využívá CUDA rozhraní pro šifrování, druhá OpenCL, další OpenSSL a poslední verze používá k akceleraci AES-NI instrukcí nových procesorů Intel. Všechny verze mají stejné rozhraní:

```
./<aes> -e|d
    -m ECB|CTR|CBC
    -l 128|192|256
    -p <password>
    -i <input_file> -o <output_file>
```

Nejprve pomocí parametru *-e* nebo *-d* je nutno zvolit, zda je požadováno šifrování nebo dešifrování. Pomocí *-m* lze zvolit blokový režim šifry. K dispozici jsou ECB, CTR a CBC. Parametr -l nastavuje velikost šifrovacího klíče 128, 192 nebo 256 bitů. Parametr *-p* přijímá uživatelský klíč. Ten může být libovolně dlouhý. Z něj je následně vypočítán hash algoritmem SHA256 a ten je teprve použit jako šifrovací klíč, tedy pouze odpovídající část. Poslední dva parametry *-i -o* pak označují názvy vstupních a výstupních souborů.

5.8 Vývojové prostředí

Vývoj aplikací probíhal primárně pod operačním systémem GNU/Linux a to pro instalace, které jsou standardně dostupné na fakultních počítačích. Pro překlad a sestavení je použit systém GNU Make. V závěru vývoje byla používána poslední dostupná verze balíku CUDA Toolkit, verze 4.0 RC2, včetně dodávané OpenCL implementace.

Kapitola 6

Výkonnostní testy

Cílem této kapitoly je shrnout všechny možné aspekty týkající se výkonnosti implementace algoritmu AES na GPU. Ať už rozdíly mezi různými režimy šifrování, velikostmi klíče (počet bitů) a dalšími kritérii jako rozdělení dat do bloků přenášených po PCI-Express sběrnici nebo volbou parametrů spouštějících výpočet na grafické kartě. Nejprve kapitola obsahuje úvodní informace o testovacích sestavách a metodice, čili jakým způsobem byla rychlost měřena a nakonec tabulky se samotnými výsledky.

6.1 Testovací sestavy

Testovacích sestav použitých pro měření bylo několik. Nejvýkonnější byla sestava č. 1, obsahovala grafickou kartu architektury *Fermi* a nabízela tak nejvíce možností díky podpoře nejnovějších technologií, které NVIDIA má. Tato sestava také obsahuje procesor s podporou instrukční sady *AES-NI*.

Hardware pcl117-00	
Procesor	Intel Core i5-2500@3.3GHz, 4 jádra, 6MB cache L3
Paměť	8 GB
Grafická karta	GeForce 465 GTX 1024 MB
Počet CUDA jader	$11 \cdot 32 = 352$
Frekvence CUDA jader	1 215 MHZ
CUDA Capability	2.0
PCI-Express	2.0 (max. 8GB/s)
Software	
Operační systém	CentOS 5.5
Ovladač GK	270.41.06

Tabulka 6.1: Testovací sestava č.1

V tabulce 6.2 je referenční počítač, který je dostupný v CVT Fakulty informačních technologií v místnosti O203. Měření probíhala na této sestavě pod operačních systémem GNU/Linux. Většina výsledků byla měřena na ní a počítač sloužil jako hlavní sestava pro měření výkonnosti v této kapitole.

Poslední sestava uvedená v tabulce byla přidána kvůli její výkonnostně slabší grafické kartě, která je i starší generace a nepodporuje příliš velkou funkcionalitu. Mohla by demonstrovat přibližný výkon například v přenosných počítačích.

Hardware pco203-01	
Procesor	Intel Core 2 Duo E8200@2.66GHz, 3MB cache L2
Paměť	2 GB
Grafická karta	GeForce 285 GTX 2048 MB
Počet CUDA jader	$30 \cdot 8 = 240$
Frekvence CUDA jader	1500 MHZ
CUDA Capability	1.3
PCI-Express	1.1 (max. 4GB/s)
Software	
Operační systém	CentOS 5.5
Ovladač GK	270.41.06

Tabulka 6.2: Testovací sestava č.2

Hardware	
Procesor	Intel Core 2 Quad Q9550@2.83GHz, 6MB cache L2
Paměť	8 GB
Grafická karta	GeForce 9500 GT 512 MB
Počet CUDA jader	$4 \cdot 8 = 32$
Frekvence CUDA jader	1 750 MHZ
CUDA Capability	1.1
PCI-Express	
Software	
Operační systém	Fedora 14
Ovladač GK	270.41.06

Tabulka 6.3: Testovací sestava č.3

6.2 Testovací metodika

V této kapitole se čtenář dozví několik faktů týkajících se testovací metodiky a to především, jakými způsoby byla měřena rychlost implementace, pro jak velká data byla rychlost měřena a způsob vyhodnocení a interpretace výsledků.

Pro měření rychlosti na platformě NVIDIA CUDA ™byl použit touto společností doporučovaný způsob měření času pomocí událostí a objektů typu *CudaEvent_t*. Tento způsob je preferovanější než v předchozích verzích používané rozhraní pomocí časovačů (*timer*) a funkcemi jako např. *cutStartTimer()*. Užití událostí by mělo být podstatně přesnější, zachycený čas je měřen v milisekundách a udávaná přenost je polovina microsekundy, viz [12]. Na CPU bylo využito rozhraní *OpenMP* pro měření rychlosti s využitím funkce *omp_get_wtime()*. Tato funkce byla zvolena kvůli jednoduchosti jejího použití a dostupnosti. Prostředí *OpenMP* bylo navíc použito pro snadnou paralelizaci šifrování na procesoru.

Vstupními daty se staly soubory od velikosti 16 kilobajtů rostoucí geometricky až po 256 megabajtů. Data byla předem načtena ze souboru do paměti a následně byl spuštěn test. Všechna měření byla prováděna na jinak nezatíženém počítači, pro danou velikost vstupu bylo měření provedeno pětkrát. Jako výsledek byla zvolena hodnota mediánu, protože se občas stalo, že některé měření se výrazně odchýlilo od ostatních výsledků.

Na grafických kartách byly měřeny tři hodnoty, které se vyskytují v tabulkách:

- **Kernel** je doba samotného výpočtu, tedy pouze šifrování respektive dešifrování bez jakýchkoli datových přenosů po sběrnici PCI-Express mezi procesorem a grafickou kartou.
- **Total** znamená celkovou dobu výpočtu, kde je zahrnuta jak výše zmíněná doba samotného výpočtu, tak přenosy vstupních a výstupních dat mezi grafickou kartu a procesorem. Dále je připočtena doba potřebná pro alokaci paměti na grafické kartě pro vstupní i výstupní data. Při měření se tedy nejprve nakopírovala všechna data z hlavní paměti na grafickou kartu. Po dokončení kompletního přenosu bylo spuštěno samotné šifrování. Následoval přenos výstupních dat do hlavní paměti a až po jeho dokončení byla určena doba ukončení měření. Nicméně v této době není zahrnuta doba nutná k přenosu klíče určeného k šifrování ani inicializačního vektoru, pokud byl potřeba.
- **Async** je asynchronní doba výpočtu měří také celkovou dobu výpočtu. Rozdíl je v tom, že se zde využívá pro zrychlení asynchronního rozhraní, které bylo popsáno v podkapitole 5.3. V této variantě jsou data rozdělena na několik menších bloků a je zde snaha o překrytí doby samotného výpočtu s datovými přenosy nebo překrytí datových přenosů směrem do grafické karty a z ní. Konkurentní datové přenosy jsou však dostupné pouze na malém množství grafických karet. Zde velmi záleží na velikosti bloku přenášených dat a výkon tak limituje i sběrnice PCI-Express, proto je důležité nalezení optimální velikosti bloku v závislosti na velikosti vstupu.

Jen pro připomenutí opakuji, že pro korektní funkčnost asynchronního rozhraní je potřeba, aby vstupní a výstupní paměť byla označena tak, aby bylo zakázáno její odkládání na disk (*page-locked*). To výrazně zrychluje datové přenosy. Aby byly zajištěny rovnocenné podmínky pro procesor i grafickou kartu, byla pro obě varianty použita stejným způsobem alokovaná paměť, tedy *page-locked*, což výrazně urychlilo i CPU variantu.

Rychlost výpočtu na CPU zahrnuje pouze samotné šifrování, jelikož nejsou potřeba žádné dodatečné datové přenosy. Tyto údaje se tedy vyskytují v tabulkách, kde je měřen absolutní čas. Uváděnou jednotkou času ve většině tabulek je milisekunda. Za těmito tabulkami následují grafy vycházející z časů předchozí tabulky a graficky znázorňují dosažené zrychlení.

6.3 Výsledky

Kapitola celkově obsahuje pouze výběr ze všech provedených testů, kvůli jejich prostorové náročnosti. Nejprve se bude tato podkapitola zabývat pouze implementací na klasickém procesoru bez jakékoli akcelerace pro porovnání různých variant na CPU. Poté bude následovat odpovídající porovnání na grafické kartě. Následně práce naznačí výkon při použití *AES-NI* instrukcí a pokračuje hledáním optimální konfigurace spouštění výpočtu na GPU v kombinaci s hledáním velikosti bloků pro rozdělení zpracovávaných dat kvůli optimálnímu přenosu dat mezi CPU a GPU. Optimální konfigurací spouštění je míněno hlavně to, kolik bloků vláken se bude spouštět na grafickém procesoru. S tím také souvisí, kolik bloků vstupních dat[1] bude zpracováno jedním vláknem.

[1] zde je blok míněn v kontextu algoritmu AES a režimů blokových šifer, jeden blok = 128 bitů

6.3.1 Výkon CPU

Jako první je uvedena tabulka 6.4 měřící rychlost šifrování a dešifrování na procesoru pro různé režimy blokových šifer. Jedná se o implementace AES z knihovny OpenSSL včetně režimů činnosti. Zde je důležité si všimnout rozdílů v časech mezi šifrováním a dešifrováním. U režimu CTR rychlost dešifrování uvedena není, protože ta je už z principu stejná, viz 2.3.3. Výkon byl měřen na sestavě č.2.

Velikost bloku	Šifrování		Dešifrování		ECB vs. CTR (%)	ECB vs. ECB (%)
	CTR (ms)	ECB (ms)	CBC (ms)			
16 KB	0,31	0,25	0,35	0,331	22,40	36,58
32 KB	0,59	0,53	0,70	0,668	12,41	32,42
64 KB	1,18	1,05	1,39	1,337	12,98	32,60
128 KB	2,37	2,08	2,77	2,684	14,03	33,11
256 KB	4,75	4,12	5,73	5,305	15,53	39,09
512 KB	9,50	7,91	11,06	10,581	20,21	39,92
1 MB	18,94	15,77	22,21	21,157	20,14	40,86
2 MB	38,18	31,59	44,57	42,795	20,85	41,06
4 MB	75,93	66,74	91,38	84,825	13,77	36,91
8 MB	152,03	125,82	185,47	168,886	20,84	47,41
16 MB	304,90	261,03	353,54	336,278	16,81	35,44
32 MB	607,76	508,41	706,38	680,340	19,54	38,94
64 MB	1220,18	1008,88	1414,28	1355,364	20,94	40,18
128 MB	2421,96	2020,26	2902,92	2715,530	19,88	43,69
256 MB	5074,18	4022,54	5823,24	5426,246	26,14	44,77

Tabulka 6.4: Rozdíly v rychlostech šifrování a dešifrování CPU na 256 bitech.

Je zde vidět poměrně markantní rozdíl, kdy doba potřebná k dešifrování je podstatně delší. Zde si nejsem přesně jist, proč je dešifrování o tolik náročnější, nejspíše je to dáno nutností použít o jednu vyhledávací tabulku navíc ve srovnání s šifrováním. Šifrování v režimu ECB je nejrychlejší ze srovnávaných variant, není zde žádná dodatečná režie způsobená režimem činnosti. Dešifrování ve stejném režimu je zhruba o 40% náročnější, což ukazuje poslední sloupec tabulky. V předposledním sloupečku je znázorněno, že šifrování v režimu CTR je výkonnostně přibližně někde mezi šifrováním a dešifrováním v režimu ECB, rozdíl je přibližně 20%. Proto je také pro většinu dalších tabulek srovnávající výkon grafického čipu s procesorem použit CTR režim, aby odrážel průměrné výkonnostní srovnání.

Na další tabulce 6.5 je nastíněn nárůst požadovaného výkonu při zvětšujícím se počtu bitů klíče použitého k šifrování v režimu CTR. Výsledek přibližně odpovídá tomu, že pro silnější šifrování je nutný větší počet kol, konkrétně o dvě a čtyři kola, což dělá teoreticky 20% a 40%. Podle tabulky je reálný nárůst přibližně 14% a 28%.

Poslední tabulkou, která se ještě zabývá čistě neakcelerovanými výkony procesoru, je tabulka 6.6. Je zde uvedena proto, aby byl demonstrován výkon vícejádrových procesorů. Dále v textu budou uváděny tabulky, kde se bude srovnávat výkon grafických čipů s výkonem jednovláknové implementace na procesoru. Nicméně čtenář by měl mít vždy na paměti, že implementace vhodné pro grafický čip půjdou stejně dobře paralelizovat i na vícejádrových procesorech. Z výsledků v tabulce je patrné, že výkon škáluje poměrně dobře až pro vyšší velikosti vstupních dat a proto nebudou v dalších srovnávacích tabulkách uváděna srovnání s těmito variantami.

Velikost bloku	128 bitů (ms)	192 bitů (ms)	256 bitů (ms)	Rozdíl v % 128 vs. 192 bitů	Rozdíl v % 128 vs. 256 bitů
16 KB	0,24	0,28	0,32	15,31	32,59
32 KB	0,47	0,54	0,60	14,63	27,19
64 KB	0,94	1,07	1,19	14,17	27,20
128 KB	1,86	2,11	2,38	13,60	27,93
256 KB	3,73	4,21	4,76	12,85	27,57
512 KB	7,45	8,93	9,50	19,88	27,61
1 MB	14,88	16,89	18,95	13,48	27,29
2 MB	30,12	33,88	38,18	12,46	26,75
4 MB	59,18	67,58	75,93	14,19	28,31
8 MB	127,25	135,20	152,04	6,25	19,48
16 MB	238,58	270,30	304,91	13,29	27,80
32 MB	473,83	570,43	607,77	20,39	28,27
64 MB	950,94	1084,64	1220,18	14,06	28,31
128 MB	1915,46	2166,14	2421,96	13,09	26,44
256 MB	3807,00	4320,12	5074,18	13,48	33,29

Tabulka 6.5: Nárůst času při zvětšujícím se počtu bitů klíče, režim CTR.

Velikost bloku	1 vlákno (ms)	2 vlákna (ms)	3 vlákna (ms)	4 vláken (ms)
16 KB	0,29	0,22	0,14	0,13
32 KB	0,58	0,35	0,25	0,22
64 KB	1,13	0,64	0,43	0,36
128 KB	2,63	1,17	0,81	0,64
256 KB	4,44	2,31	2,97	1,20
512 KB	8,88	4,55	3,50	2,32
1 MB	17,66	8,73	5,98	4,68
2 MB	34,47	17,25	11,83	9,02
4 MB	68,76	34,39	23,61	17,98
8 MB	138,01	68,71	47,15	35,87
16 MB	272,50	138,05	94,21	71,64
32 MB	537,17	274,35	188,24	143,15
64 MB	1075,00	555,27	376,35	286,50
128 MB	2143,59	1097,78	753,85	572,85
256 MB	4274,05	2196,59	1504,81	1144,21

Tabulka 6.6: Škálování výkonu dešifrování na vícejádrových procesorech, sestava č.3.

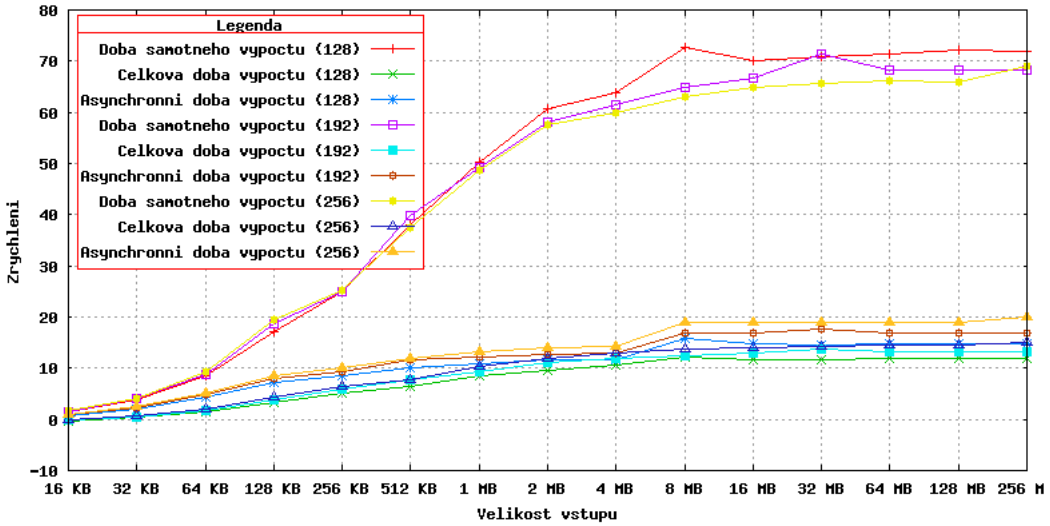
6.3.2 Výkon GPU

Jako první je zde uvedena tabulka 6.7, kde je zachycen pouze výkon samotného šifrování bez započtení přenosů dat mezi hlavní pamětí RAM a GPU. Tabulka je protějškem tabulky 6.4. V porovnání s ní na grafickém čipu nedochází k tak markantnímu propadu výkonu v případě dešifrování jako tomu bylo na CPU. Rozdíl je řádově v desetinách procent, což se dá považovat za statistickou chybu. Naproti tomu jistý rozdíl je zde pozorovatelný v případě šifrování v režimu CTR a ECB. Paradoxně teoreticky výpočetně náročnější režim CTR sedí grafickému čipu nepatrně lépe a podává tak vyšší výkon. Dešifrování v režimu CBC je pro grafický čip pomalejší, jelikož se pracuje se dvěma vstupními bloky. Dá se ale prohlásit, že zvolený režim, v kontrastu s CPU, nemá na výkon tak velký dopad.

Velikost bloku	Šifrování		Dešifrování		ECB vs. CTR (%)	ECB vs. ECB (%)
	CTR (ms)	ECB (ms)		CBC (ms)		
16 KB	0,11715	0,11996	0,11935	0,12979	0,97658	0,99491
32 KB	0,11702	0,11891	0,12006	0,12962	0,98411	1,00967
64 KB	0,11760	0,12006	0,12259	0,12763	0,97951	1,02107
128 KB	0,11830	0,11955	0,12146	0,13154	0,98954	1,01598
256 KB	0,18298	0,18221	0,18298	0,19780	1,00423	1,00423
512 KB	0,24848	0,25062	0,25139	0,27125	0,99146	1,00307
1 MB	0,38479	0,38710	0,39034	0,42207	0,99403	1,00837
2 MB	0,65386	0,66493	0,66954	0,72444	0,98335	1,00693
4 MB	1,24970	1,27950	1,28740	1,38932	0,97671	1,00617
8 MB	2,37659	2,43880	2,44839	2,70198	0,97449	1,00393
16 MB	4,62849	4,75260	4,77540	5,18976	0,97389	1,00480
32 MB	9,13240	9,38640	9,42849	10,21943	0,97294	1,00448
64 MB	18,18499	18,71199	18,78200	21,13508	0,97184	1,00374
128 MB	36,24499	37,29999	37,45300	40,34376	0,97172	1,00410
256 MB	72,37399	74,48499	74,76699	81,46933	0,97166	1,00379

Tabulka 6.7: Rozdíly v rychlostech šifrování a dešifrování GPU na 256 bitech.

Tabulka 6.8 je zaměřena na propad výkonu při použití silnějšího režimu šifrování, čili jak se projeví více kol algoritmu AES. Jedná se o větší protějškem CPU verze z tabulky 6.5. Pokud bychom brali v úvahu pouze rychlost samotného šifrování a pokud bychom zanedbali výsledky pro menší vstupní data, z tabulky vyplývá, že výsledky více odpovídají teorii a propad výkonu je přibližně 19% a 38% při použití 192 a 256 bitů ve srovnání se 128 bitovým režimem. Obrázek 6.1 znázorňuje dosahované zrychlení proti CPU variantě.



Obrázek 6.1: Zrychlení šifrování pro data z tabulky 6.8

Naproti tomu se tento rozdíl začne vytrácet, pokud budeme uvažovat datové přenosy. V případě celkové doby absolutní rozdíl samozřejmě zůstane, v relativním porovnání se podíl doby výpočtu minimalizuje. Při využití asynchronních přenosů dat se tento rozdíl naprosto vytrácí, jelikož datové přenosy zaberou podstatně více času něž doba výpočtu a tu je tedy možné úplně překrýt. Ve výsledku je pak jedno, jestli se použije šifrování na 128 nebo 256

Velikost bloku	CPU	GPU		
	Výpočet (ms)	Kernel (ms)	Total (ms)	Async (ms)
128 bitové šifrování				
16 KB	0,23	0,09609	0,32768	0,13763
32 KB	0,46	0,09580	0,34314	0,15917
64 KB	0,93	0,09782	0,37386	0,17104
128 KB	1,82	0,10320	0,43123	0,22722
256 KB	3,72	0,14349	0,59667	0,38976
512 KB	7,44	0,19026	0,99795	0,67773
1 MB	14,88	0,28963	1,57020	1,24520
2 MB	30,12	0,48890	2,81939	2,37519
4 MB	59,18	0,91312	5,07690	4,67189
8 MB	127,25	1,72799	9,66590	7,61380
16 MB	238,58	3,35389	18,82199	15,20500
32 MB	473,82	6,60379	37,11399	30,38700
64 MB	950,94	13,13899	73,70099	60,72099
128 MB	1915,46	26,18100	146,90000	121,34999
256 MB	3807,00	52,26899	293,37999	242,86000
192 bitové šifrování				
16 KB	0,27	0,10614	0,33461	0,14845
32 KB	0,53	0,10557	0,35465	0,15845
64 KB	1,06	0,10778	0,38207	0,18163
128 KB	2,11	0,10678	0,44163	0,23709
256 KB	4,20	0,16253	0,61360	0,40988
512 KB	8,92	0,21840	1,02849	0,70357
1 MB	16,89	0,33578	1,61979	1,29239
2 MB	33,87	0,57362	2,80989	2,45449
4 MB	67,57	1,08200	5,24929	4,84090
8 MB	135,19	2,05330	9,98920	7,61479
16 MB	270,29	3,99710	19,46900	15,23099
32 MB	570,42	7,87600	38,40699	30,37600
64 MB	1084,64	15,67999	76,22700	60,70700
128 MB	2166,13	31,24399	151,93000	121,43000
256 MB	4320,11	62,38199	303,50000	243,05000
256 bitové šifrování				
16 KB	0,31	0,11731	0,34483	0,15826
32 KB	0,59	0,11498	0,36573	0,16771
64 KB	1,18	0,11533	0,39171	0,19334
128 KB	2,37	0,11549	0,44506	0,24723
256 KB	4,75	0,18226	0,63517	0,42774
512 KB	9,50	0,24626	1,07489	0,73282
1 MB	18,94	0,38090	1,66270	1,33279
2 MB	38,18	0,65302	2,97319	2,54159
4 MB	75,93	1,24560	5,41640	5,00860
8 MB	152,03	2,37029	10,32000	7,61669
16 MB	304,90	4,62629	20,15599	15,20700
32 MB	607,76	9,12829	39,62500	30,46399
64 MB	1220,18	18,17899	78,72899	60,77900
128 MB	2421,96	36,24900	156,93999	121,46999
256 MB	5074,18	72,37900	313,76999	242,83000

Tabulka 6.8: Rozdíly v rychlosti šifrování v režimu CTR na 128, 192, 256 bitech

bitech, jelikož z tohoto pohledu trvají operace stejně dlouho.

6.3.3 Maximální výkon GPU

Jak již bylo naznačeno dříve, dosažení maximálního výkonu závisí na několika faktorech, které spolu navzájem souvisí. Jsou jimi:

- velikost bloku dat přenášených mezi pamětí procesoru a grafické karty po sběrnici PCI-Express při použití asynchronních přenosů,
- výkonnost grafické karty, která je dána především počtem multiprocesorů,
- počet AES bloků (128 bitů) zpracovávaných jedním vláknem.

PCI-Express je dnes prakticky jedinou sběrnicí používanou pro komunikaci procesoru a grafických karet. Její efektivita je dána velikostí dat, které se po ní přenášejí, aby se minimalizovala s tím spojená režie. Maximálního výkonu lze obecně dosáhnout přenosem vstupních dat v jednom celku. Nicméně pro překrytí výpočtu a datových přenosů je potřeba data rozdělit na bloky. Příliš malá velikost znamená velkou režii a příliš velká zase malý počet bloků a nízké překrytí. Velkou roli zde také hraje velikost vstupních dat.

Výkon grafické karty, počet jejich multiprocesorů a počet zpracovávaných dat jedním vláknem spolu úzce souvisí. Obecně platí, že je lepší nechat jedno vlákno zpracovávat více položek. Dojde tak například k lepšímu využití některých zdrojů nebo může dojít k lepšímu promíchání výpočetních instrukcí s datovými instrukcemi. Jak bylo zmíněno v kapitole s implementací, v případě algoritmu AES je potřeba pro lepší výkon přenést vyhledávací tabulky z paměti konstant do sdílené paměti, což je určitá režie.

V závislosti na počtu procesorů grafického čipu se musí dát pozor na to, aby při zvoleném počtu prvků zpracovávaných jedním vláknem neklesl počet bloků vláken spouštěných na multiprocesorech pod určitou mez. Je potřeba mít dostatečný počet bloků vláken a pro ně nachystané co největší množství práce. Cílem je tedy nalézt takový kompromis, aby klesal čas výpočtu a především asynchronní čas.

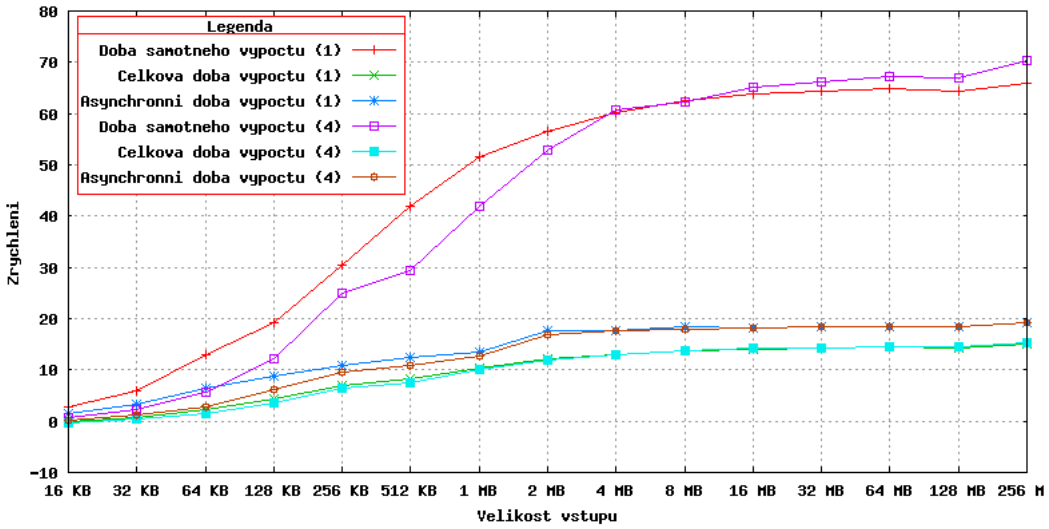
Velikost bloku	CPU	GPU		
	Výpočet (ms)	Kernel (ms)	Total (ms)	Async (ms)
16 KB	0,316	0,08374	0,31541	0,12822
32 KB	0,593	0,08460	0,33088	0,13894
64 KB	1,187	0,08540	0,36165	0,16009
128 KB	2,376	0,11706	0,45119	0,24134
256 KB	4,757	0,15075	0,60655	0,39677
512 KB	9,502	0,22141	1,03610	0,70733
1 MB	18,946	0,36058	1,65129	1,31560
2 MB	38,181	0,66354	2,90160	2,05310
4 MB	75,934	1,24190	5,39829	4,09159
8 MB	152,036	2,39520	10,33799	7,85210
16 MB	304,908	4,69789	20,14699	15,82799
32 MB	607,768	9,31339	39,79599	31,39399
64 MB	1220,180	18,53600	79,09099	62,97200
128 MB	2421,960	36,99199	157,65000	125,29999
256 MB	5074,180	75,77631	314,00999	250,46000

Tabulka 6.9: Šifrování 256 bitů, CTR, 1MB blok dat, vlákno šifruje 1 blok

Velikost bloku	CPU	GPU		
	Výpočet (ms)	Kernel (ms)	Total (ms)	Async (ms)
16 KB	0,316	0,18090	0,41280	0,27242
32 KB	0,593	0,18093	0,42649	0,28259
64 KB	1,187	0,18112	0,45667	0,30482
128 KB	2,376	0,18179	0,51536	0,32591
256 KB	4,757	0,18314	0,63421	0,44701
512 KB	9,502	0,31130	1,11670	0,79366
1 MB	18,946	0,44130	1,72080	1,39159
2 MB	38,181	0,70803	2,95429	2,12970
4 MB	75,934	1,23229	5,39799	4,09269
8 MB	152,036	2,40189	10,34099	8,00249
16 MB	304,908	4,61280	20,07999	15,83200
32 MB	607,768	9,03389	39,53099	31,47400
64 MB	1220,180	17,87800	78,42799	62,89600
128 MB	2421,960	35,67799	156,34000	125,14000
256 MB	5074,180	71,16800	312,10000	250,00999

Tabulka 6.10: Šifrování 256 bitů, CTR, 1MB blok dat, vlákno šifruje 4 bloky

Ukázka problému nalezení kompromisu je demonstrována v tabulkách 6.9 a 6.10. Po PCI-Express sběrnici se přenáší bloky dat velikosti 1 megabajt. Jedno vlákno šifruje jeden nebo čtyři AES bloky (po 16 bajtech). Vláken v bloku je vždy 256. Řekněme, že velikost vstupních dat bude 128 kilobajtů. Při takové konfiguraci se vytvoří práce pro 32 nebo 8 multiprocesorů. Na kartě, která má větší počet multiprocesorů, tak nemusí dojít k plnému využití všech multiprocesorů a výkon klesá. Nejvíce markantní je to pro velmi malá data ve sloupci s časem výpočtu GPU. Konfigurace se čtyřmi prvky na vlákno je více než dvakrát pomalejší v důsledku nízkého využití multiprocesorů grafiky. Karta se obrací pro největší velikost vstupu, kdy je již dostatek práce pro všechny multiprocesory a rychlost výpočtu je lepší. I asynchronní čas je při vyšších velikostech vstupu shodný, protože se výpočet schová za datové přenosy.



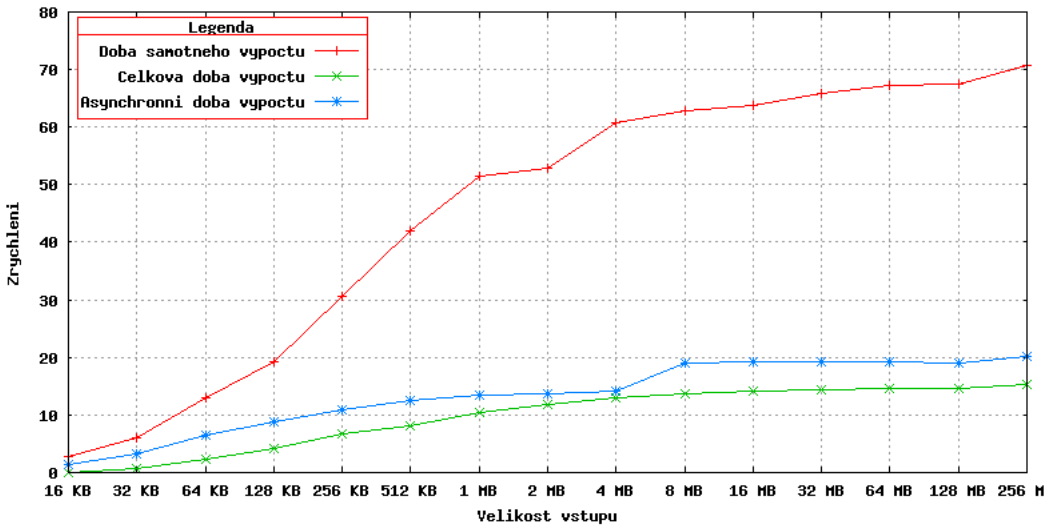
Obrázek 6.2: Zrychlení šifrování pro data z tabulkek 6.9 a 6.10 v porovnání s CPU.

To by však nemuselo platit, pokud by byla velikost bloku například 128kB. V takovém případě by rozdělená data byla zpracována s nízkým využitím GPU i pro větší vstupní data a výsledný asynchronní čas 4prvkové varianty by byl horší až o 20% a byl by i horší, než celková doba zpracování přibližně o polovinu. Dále například při srovnání voleb bloku dat 128kB a 1MB by byl asynchronní čas až dvakrát horší. Závěrem lze konstatovat, že konfigurace 1MB dat přenášených po sběrnici a zpracování jednoho prvku vstupních dat jedním vláknem představuje velmi dobrou kombinaci pro soubory menší velikosti než 2MB. Nicméně i pro větší vstupní data nabízí dobrý výkon.

Při zpracovávání větších objemů dat se vyplatí uvažovat o zpracování 8 prvků jedním vláknem a pro přenos po PCI-Express volit hodnoty 8–16MB. V tabulce 6.11 jsou uvedeny dosažené časy s vybranými nastaveními pro dosažení maximálního výkonu pro všechny velikosti vstupních dat.

Velikost bloku	CPU	GPU		
	Výpočet (ms)	Kernel (ms)	Total (ms)	Async (ms)
16 KB	0,316	0,0837	0,3154	0,1282
32 KB	0,593	0,0846	0,3308	0,1389
64 KB	1,187	0,0854	0,3616	0,1601
128 KB	2,376	0,1170	0,4511	0,2413
256 KB	4,757	0,1507	0,6065	0,3968
512 KB	9,502	0,2214	1,0361	0,7073
1 MB	18,946	0,3605	1,6512	1,3156
2 MB	38,181	0,7081	2,9531	2,5885
4 MB	75,934	1,2325	5,4195	4,9837
8 MB	152,036	2,3799	10,3130	7,6115
16 MB	304,908	4,7009	20,1589	15,1250
32 MB	607,768	9,0818	39,5829	30,1220
64 MB	1220,180	17,8500	78,3979	60,2140
128 MB	2421,960	35,3909	156,0699	120,2900
256 MB	5074,180	70,6899	311,6299	240,5999

Tabulka 6.11: Šifrování 256 bitů, CTR, maximální výkon sestavy č.2

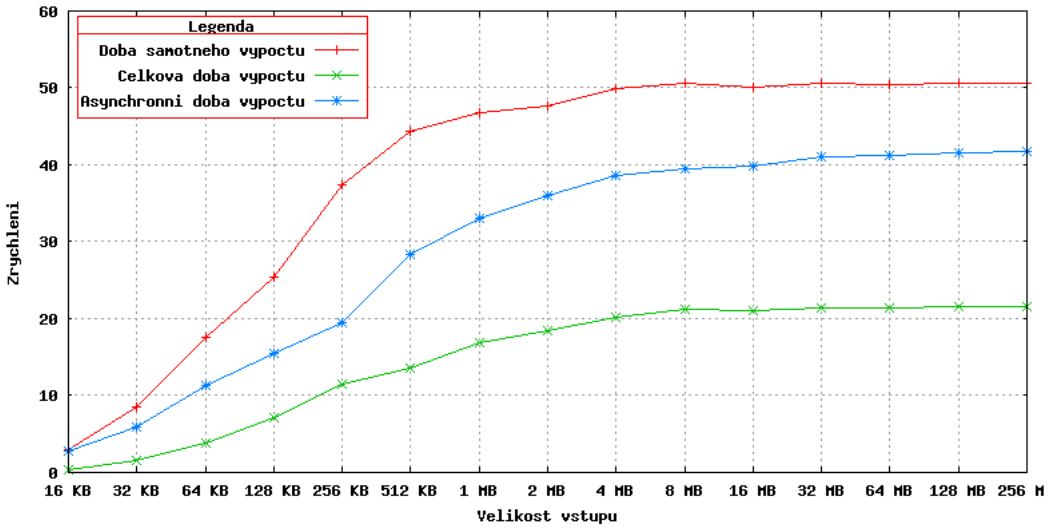


Obrázek 6.3: Zrychlení šifrování ve srovnání s CPU pro data z tabulky 6.11

Situace se dosti mění na testovací sestavě č. 1, která obsahuje silnější procesor, rychlejší sběrnici a grafickou kartu architektury *Fermi*. Karta ve srovnání s druhou sestavou obsahuje menší počet multiprocesorů, zato jeden multiprocesor je tvořen větším počtem CUDA jader. Výsledky měření na této sestavě již nebyly tak závislé na konfiguraci spouštění, tedy kolik prvků bude jedno vlákno zpracovávat, ani na velikosti bloku přenášeného po PCI-Express. To je nejspíše dáno hlavně díky rychlejší sběrnici a změnou architektury multiprocesoru GPU, možná i vylepšenou komunikací grafické karty po sběrnici obecně.

Velikost bloku	CPU	GPU		
	Výpočet (ms)	Kernel (ms)	Total (ms)	Async (ms)
16 KB	0,242	0,05990	0,17946	0,06444
32 KB	0,470	0,04950	0,18013	0,06854
64 KB	0,935	0,05013	0,19037	0,07571
128 KB	1,923	0,07290	0,23462	0,11683
256 KB	3,717	0,09679	0,29914	0,18076
512 KB	7,596	0,16774	0,52451	0,25913
1 MB	14,728	0,30784	0,82755	0,43200
2 MB	28,751	0,59091	1,48364	0,77612
4 MB	57,257	1,12525	2,71178	1,44499
8 MB	114,459	2,21465	5,16608	2,82179
16 MB	223,381	4,37126	10,11650	5,47494
32 MB	448,632	8,68338	20,08210	10,68420
64 MB	891,413	17,31409	39,78679	21,07999
128 MB	1784,109	34,58970	79,26680	41,89090
256 MB	3562,969	69,13209	158,25499	83,51390

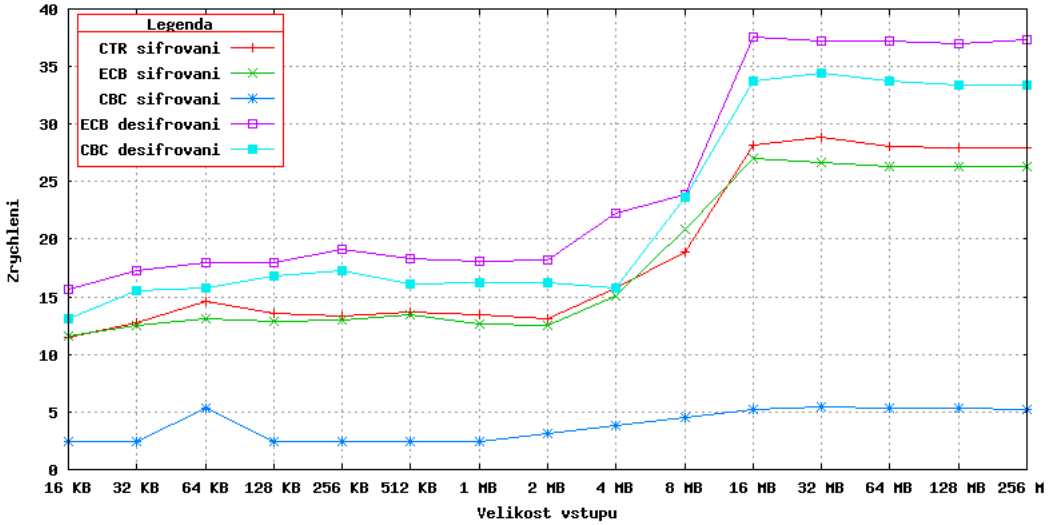
Tabulka 6.12: Šifrování 256 bitů, CTR, maximální výkon sestavy č.1



Obrázek 6.4: Zrychlení šifrování ve srovnání s CPU pro data z tabulky 6.12

6.3.4 AES-NI

Následující obrázek 6.5 ukazuje zrychlení při srovnání OpenSSL implementace s verzí používající AES-NI instrukce. Obě implementace běžely na testovací sestavě č.1. Zrychlení je markantní už od nejmenších velikostí. Pro sekvenční režim CBC je zrychlení také dobré, nicméně všechny paralelní režimy dosahují podstatně lepších výsledků.



Obrázek 6.5: Zrychlení šifrování různých režimů AES-NI verze proti OpenSSL.

6.3.5 Zhodnocení

Závěrem lze konstatovat, že výkon grafické karty sestavy č.2 je při správné konfiguraci parametrů vždy výkonnější i pro menší vstupní data ve srovnání s klasickou implementací. Propustnost systému je přes 1 GB/s pokud bereme v úvahu přenos dat na GPU, samotné šifrování pak 3.57 GB/s. Zrychlení při šifrování je tedy až 70-ti násobné, při započtení datových přenosů až 20-ti násobné na dané konfiguraci. Výkon je do značné míry limitován sběrnicí PCI-Express, jelikož výpočet je kompletně skryt za datovými přenosy a i tak je zde ještě velká rezerva (70ms proti 240ms). Karta by tak mohla teoreticky zvládnout 2krát větší přísun dat.

To se potvrzuje na sestave č.1. Ta má k dispozici rychlejší sběrnici a na rychlosti je to znát. Ve srovnání s klasickou implementací pomocí OpenSSL dosahuje GPU až 43-ti násobného zrychlení se započítanými datovými přenosy, bez nich je zrychlení až 51-ti násobné. Propustnost systému je 3.7 GB/s v případě samotného šifrování, s datovými přenosy 3 GB/s.

Verze využívající AES-NI instrukce dosahují také velmi dobrých časů, hlavně u paralelních režimů. Tato verze je ve srovnání s GPU verzí o něco pomalejší, ale výsledek není až tak výrazný.

Kapitola 7

Závěr

Pro efektivní běh algoritmů na grafických kartách je potřeba mít dostatečné množství nezávislých dat, se kterými je možno dále paralelně pracovat. V kryptografii se kvůli vysoké výpočetní náročnosti asymetrických algoritmů používá hybridní kryptografie, kdy pomocí asymetrické kryptografie dojde k výměně symetrického klíče a ten je následně použit pro rychlejší šifrování většího množství dat. Stejně tak tento princip funguje i při SSL spojení, na což se měla práce zaměřit.

Naproti tomu tato diplomová práce nijak výrazně nepopisuje oblast hašovacích funkcí, protože se jedná většinou o iterační funkce a nenabízejí velké možnosti paralelizace. Na jednu stranu v práci [4] byla uvedena implementace algoritmu MD5, která doznala značného zrychlení. Na druhou stranu MD5 hašovací funkce již delší dobu není považována za bezpečnou, je doporučováno nahradit ji rodinou hašovacích funkcí SHA (Secure Hash Algorithm), tudíž implementace MD5 nebyla v rámci této práce dále rozvíjena ani implementována. Práce [8] se zabývá implementací dvou algoritmů rodiny SHA, ale je zde uvedeno přibližně padesátinásobné zpomalení. Na základě zmíněných výsledků bylo upuštěno i od této varianty a celkově od dalšího prozkoumávání a snah v oblasti hašovacích funkcí.

Kvůli potížím s nalezením hašovacího algoritmu a používání tzv. hybridní kryptografie se tato diplomová práce soustředila hlavně na oblast symetrické kryptografie s primárním zaměřením na algoritmus AES. Ten je v současné době, ale i do budoucna, považován za bezpečný, rychlý, silně doporučovaný a stal se i nejpoužívanějším. Všechny moderní aplikace i nová hardwarová zařízení přicházejí primárně s tímto algoritmem. AES byl implementován pro obě GPU rozhraní, CUDA i OpenCL s popisem v kapitole 5. Míra paralelismu primárně závisí na použitém blokovém režimu, které byly vysvětleny v podkapitole 2.3. Byly implementovány všechny uvedené paralelní režimy, tedy ECB, Counter a dešifrování v režimu CBC. Přestože šifrování v režimu CBC patří mezi nejpoužívanější, režim nebyl implementován na GPU, protože se jedná o sekvenční proces. Ve spojení s paralelními režimy by mohly být dále obdobným způsobem implementovány prakticky jakékoli požadované symetrické blokové šifry.

Výsledky uvedené v kapitole 6 dosahují v závislosti na zvolené hardwarové konfiguraci testovacích počítačů až sedmdesátinásobného zrychlení, pokud bereme v úvahu pouze rychlost samotného šifrování a až dvacetinásobného se započítanými datovými přenosy. Hlavní testovací počítač byl do značné míry limitován sběrnicí PCI-Express, což potvrzují výsledky na jiné testovací sestavě. Na té bylo dosaženo podstatně vetší přenosové rychlosti sběrnice a výsledky tomu odpovídají. Je však třeba brát v úvahu, že srovnání probíhala s jednovláknovou verzí OpenSSL šifrování. Kapitola se také snažila objasnit problematiku optimální

konfigurace algoritmu pro dosažení maximálního výkonu.

Možné využití tedy vidím například pro šifrování souborů nebo celých souborových systémů. Dále ve spojení s SSL kanálem by mohl být výkon grafických karet efektivně využit při nasazení na serverech fungujících jako datová uložiště, například služby typu Rapidshare nebo video servery se záznamy přednášek, kde se pracuje s velkými bloky dat.

Na základě studia kryptografických rozhraní, která existují v operačních systémech, nevidím příliš velké problémy s integrací a využitím GPU pro služby jádra operačního systému GNU/Linux. Zádrhel by mohl nastat snad jen s blokovými režimy, kdy jádro využívá vlastních implementací. Pro efektivní běh by tak bylo nutné donutit rozhraní, aby používalo blokové režimy dodávané s algoritmem. Další možný problém by mohl být s režií vzniklou kopírováním paměti. CryptoAPI však již v návrhu na toto brala ohledy. Práce by mohla, dle mého názoru, bezproblémově pokračovat integrací s CryptoAPI. Na operačních systémech typu BSD je velký problém v tom, že například společnost NVIDIA oficiálně tyto systémy nepodporuje, tudíž nenabízí potřebné ovladače. Existují funkční snahy využívat GPU díky emulaci a spouštění binárních souborů určených pro operační systém GNU/Linux. Zde by bylo nejspíše vhodnější využít procesory s podporou AES-NI instrukční sady, pro kterou již je ovladač pro OCF na systému FreeBSD dostupný.

Literatura

[1] Advanced Encryption Standard. 2001.
URL http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[2] Intel AESNI Sample Library. 2011.
URL http://software.intel.com/en-us/articles/download-the-intel-aesni-sample-library/

[3] Daemen, J.; Rijmen, V.: AES Proposal: Rijndael. 1998.
URL http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf

[4] Hu, G.; Ma, J.; Huang, B.: High Throughput Implementation of MD5 Algorithm on GPU. In *Ubiquitous Information Technologies Applications, 2009. ICUT '09. Proceedings of the 4th International Conference on*, Dec. 2009, ISSN 1976-0035, s. 1 –5, doi:10.1109/ICUT.2009.5405734.
URL http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=05405734

[5] Joppe W. Bos, D. A. O.; Stefan, D.: Fast Implementations of AES on Various Platforms. Cryptology ePrint Archive, Report 2009/501, 2009.
URL http://eprint.iacr.org/2009/501.pdf

[6] Keromytis, A. D.; Wright, J. L.; de Raadt, T.: The Design of the OpenBSD Cryptographic Framework. In *USENIX Annual Technical Conference, General Track*, 2003, s. 181–196.
URL http://www.usenix.org/events/usenix03/tech/keromytis.html

[7] Leffler, S. J.: Cryptographic device support for FreeBSD. In *Proceedings of the BSD Conference 2003 on BSD Conference*, Berkeley, CA, USA: USENIX Association, 2003, s. 8–8.
URL http://www.usenix.org/event/bsdcon03/tech/leffler_crypto/leffler_crypto.pdf

[8] Lerchundi Osa, G.: *Fast Implementation of Two Hash Algorithms on nVidia CUDA GPU*. Diplomová práce, Universitat Politecnica de Catalunya, Feb. 2009.
URL http://upcommons.upc.edu/pfc/bitstream/2099.1/7933/1/Masteoppgave.pdf

[9] Ludvig, M.: CryptoDev for Linux. 2010-12-30.
URL http://www.logix.cz/michal/devel/cryptodev/

[10] McCullough, D.: OCF-Linux - Asynchronous Crypto Acceleration for Linux. 2010-12-30.
URL http://ocf-linux.sourceforge.net/

[11] Morris, J.: The Linux Kernel Cryptographic API. 2003-04-01.
URL http://www.linuxjournal.com/article/6451

[12] NVIDIA Corporation: *CUDA C Best Practices Guide*. 2011.
URL http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf

[13] Wikipedia, The Free Encyclopedia: Block cipher modes of operation. 2010, [cit. 2011-01-05].
URL http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

[14] Wikipedia, The Free Encyclopedia: Advanced Encryption Standard. 2011, [cit. 2011-05-14].
URL http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[15] Volkov, V.: Better performance at lower occupancy. GPU Technology Conference 2010, 2010.
URL http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf

[16] WWW stránky: Kerckhoffův princip. [cit. 2011-05-12].
URL http://www.sms007.cz/index.php?lang=cs&type=special&page=kerckhoff

[17] WWW stránky: OpenSSL. 2010-12-30.
URL http://www.openssl.org/

[18] WWW stránky: OpenCL. 2011, [cit. 2011-01-02].
URL http://www.khronos.org/opencl/

[19] WWW stránky: CUDA SM Structure. 2011-01-01.
URL http://gputoaster.files.wordpress.com/2010/12/smcuda.png

[20] WWW stránky: CUDA Memory model. 2011-01-10.
URL http://www.ixbt.com/video3/images/cuda/cuda4.png

Dodatek A

Obsah CD

Adresářová struktura datového nosiče:

- **bin** Adresář pro binární soubory, zde budou uloženy všechny varianty spustitelných souborů po jejich překladu. Pro platformu Windows zde budou umístěny také dynamické knihovny potřebné pro spuštění.
- **doc** Adresář se zdrojovými kódy této diplomové práce v LaTeX-u.
- **include** Adresář s externími knihovnami, na nichž je překlad závislý. Primárně potřebný pro překlad na platformě Windows.
- **install** Adresář s kopiemi instalačních souborů CUDA Toolkit-u, pod kterým probíhal vývoj.
- **lib** Adresář se zdrojovými kódy *Intel AESNI Sample* knihovny. Pro platformu Windows další dodatečné knihovny (OpenSSL) nutné pro sestavení aplikace.
- **src** Zdrojové kódy implementace algoritmu AES v prostředí CUDA, OpenCL, AESNI.
- **readme.txt** Textový soubor s dodatečnými informace o překladu a spuštění.