Masarykova univerzita
Fakulta informatiky

Programovatelné grafické procesory a jejich aplikace v kryptografii

Diplomová práce

Bc. Marek Čermák

2011

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

V Brně 22. 5. 2011

Poděkování

Děkuji Mgr. et Mgr. Janu Krhovjákovi, PhD. za odborné vedení této práce, za jeho rady a podněty, které značnou měrou přispěly k jejímu vytvoření.

Chtěl bych poděkovat rodičům za podporu a vytvoření podmínek ke studiu.

Dále bych chtěl poděkovat také MetaCentru, jmenovitě RNDr. Jiřímu Filipovičovi, a Laboratoři Bezpečnosti a Aplikované Kryptografie, jmenovitě RNDr. Marku Kumpoštovi, Ph.D. za poskytnutí výpočetní kapacity k testování praktické části.

Poslední poděkování patří Marušce Švomové za komplexní korekturu.

Shrnutí

Cílem této diplomové práce je detailní analýza možností využití programovatelných grafických procesorů (GPU) s architekturou CUDA (Compute Unified Device Architecture) pro kryptografické účely. Teoretická část práce je věnována rozdílům při programování na CPU a GPU, které jsou demonstrované na vzorových příkladech. Ve druhé části práce navazuje popisem konkrétních využití programovatelných grafických procesorů pro kryptologické účely a shrnutím současných výsledků dosažených v této oblasti. Praktickou částí práce jsou implementace šifrovacího algoritmu Kasumi, které jsou optimalizovany jak pro kryptografické (rychlost dané instance algoritmu) tak pro kryptoanalytické (množství instancí) účely. Součástí práce jsou také testy rychlosti daných implementací a jejich srovnání s ekvivalentními algoritmy běžícími na běžných CPU.

Klíčová slova

Bloková šifra, CUDA, GPU, KASUMI, kryptoanalýza, kryptografie.

Obsah

Úvod

Od svého vzniku pronikly počítače do každodenního života a pro moderní společnost je téměř nemožné představit si existenci bez nich. V mnohém usnadňují běžné a rutinní záležitosti nebo naopak přináší nové potřebné možnosti v různých oblastech lidské činnosti. S jejich rozvojem však rostla i potřeba zajistit bezpečí dat jimi zpracovávanými. K tomuto účelu byly vytvořeny různé prostředky, při jejichž návrhu se však ve většině případů nepředpokládal raketový růst objemu dat které je v současnosti potřeba zabezpečit. S tímto je spojen i nárůst času potřebný pro jejich zabezpečení. To způsobuje v zásadě dva problémy. Uživatelská nepřívětivost a čas, ve kterém data nejsou zabezpečena. Jaký typ problému v současné době převažuje, je však diskutabilní. Práce se zaměřuje pouze na druhý zmíněný problém, tedy časový úsek, ve kterém data nejsou chráněna.

K tomuto problému se váží možnosti, jak takovým situacím předcházet. Jako řešení se rýsují dva způsoby – zmenšit objem dat nebo urychlit průběh použitých algoritmů. Protože však ve většině případů není možné zmenšit datový soubor, je reálné pouze urychlení algoritmů, které je nezávislé na vlastních datech. Přestože stále platí Mooreův zákon [1], na jehož principu se nám zvyšují výpočetní možnosti procesorů, v poslední době dochází k rapidnímu nárůstu využití paralelismu k akceleraci algoritmů.

Již několik let se úspěšně daří paralelizovat některé problémy pomocí většího množství jader zapojených paralelně, a to jak na úrovní jader ve vícejádrových procesorech, tak na úrovni procesorů. Na začátku roku 2007 však přišla firma NVidia s novým paradigmatem, když představila CUDA (Compute Unified Device Architecture) [2] a tím umožnila programátorům vyvíjet aplikace, které využívají grafické karty pro paralelní výpočty nejen grafického charakteru. Přestože snahy o tyto výpočty probíhaly již před vznikem této architektury, teprve s uvedením CUDA dostalo využití grafických akcelerátorů nový rozměr.

Hlavním cílem této diplomové práce je detailní analýza možností využití programovatelných grafických procesorů pro kryptografické účely, vytvoření rešerše dosažených výsledků v této oblasti a predikce vývoje v této oblasti do budoucna. Praktickou částí bude využití teoretických znalostí k akceleraci šifry Kasumi pomocí grafického procesoru pro účely kryptografické i kryptoanalytické a prezentace dosažených hodnot na běžně dostupném hardwaru.

Cílem teoretické části této práce je představení možností programovatelných grafických procesorů, zejména pak architektury CUDA společnosti NVidia, a srovnání rozdílů mezi programováním pro CPU a GPU. Dále jsou rozebrána ukázková řešení základních problémů při psaní programů vhodných pro tuto architekturu.

Na předchozí část navazuje představení základních šifrovacích principů a souhrn výsledků dosažených u různých kryptografických algoritmů při použití grafických akcelerátorů. Následuje detailní popis šifry Kasumi a úvaha nad možnostmi paralelizace tohoto algoritmu. Praktickou částí je souhrn měření dosažených při akceleraci Kasumi pomocí grafické karty a jejich diskuze.

V první kapitole této práce jsou definovány základní pojmy z oblasti programovatelných grafických procesorů. Dále jsou v této části uvedeny důležité základní pojmy z oblasti kryptologie.

Druhá kapitola pojednává o programovatelných grafických procesorech. Je zde rozebrán hlavní rozdíl v architektuře CPU a GPU. Následně je popsána architektura CUDA s důrazem na jednoduchost a srozumitelnost. Jsou také ukázány nejčastější problémy týkající se implementace programů určených pro programovatelné grafické procesory a způsob jejich předcházení. Závěr této kapitoly je věnován programovacímu jazyku CUDA C určenému právě pro architekturu CUDA. Tato část obsahuje jednoduché příklady a praktické ukázky, které využívají teoretických znalostí předchozí části

a uplatňují je v praxi.

Třetí kapitola rozebírá současný stav výzkumu v oblasti využití programovatelných grafických procesorů pro kryptologické účely. Zaměřuje se zejména na v současnosti používané standardy v oblasti kryptografie – AES, DES a A5/1. V této části jsou popsány jak dosažené výsledky, tak i postupy, které k nim vedly.

Čtvrtá kapitola je zaměřená na samotnou šifru Kasumi – její detailní popis, analýzu jednotlivých komponent a shrnutí dosažených úspěchů v oblasti kryptoanalýzy.

V páté kapitole jsou stručně popsány nástroje použité pro tvorbu experimentálních programů praktické části. Dále jsou představeny řešené problémy oblasti kryptologie (rychlost šifrování, útok hrubou silou, generování rainbow chains) a postupy vedoucí k jejich paralelizaci nad šifrou KASUMI pomocí GPU. Závěrem každé části je pak diskuze dosažených výsledků pro každý problém.

Diskuze na konci práce shrnuje všechna důležitá fakta z předchozích kapitol a nastiňuje pravděpodobný vývoj v dané oblasti do budoucna.

Kapitola 1

Základní terminologie

V této části jsou uvedeny a vysvětleny základní pojmy v oblasti programování GPU a kryptografie, které jsou nutné pro pochopení dalších kapitol.

1.1 GPU

GPU (*grafický procesor*, z angl. *Graphics Processing Unit*) [3] je velice výkonný specializovaný procesor umístěný na grafické kartě.

GPGPU (*General-purpose Computing on Graphics Processing Units*) [4] je technika využití GPU pro výpočty jiného než grafického charakteru. Jedná se pak o využití proudového zpracování (z angl. stream processing), které je typické pro paralelismus typu SIMD (jedna instrukce, více dat, z angl. Single Instruction Multiple Data).

CUDA (*Compute Unified Device Architecture*) je paralelní výpočetní architektura vyvinutá firmou NVidia umožňující využít grafickou kartu jako prostředek k paralelizaci práce nad teoreticky libovolnými daty. Podrobněji je tato architektura popsána v dalších kapitolách.

Výpočetní schopnosti (z angl. *Compute Capability*) zařízení udávají typ použité architektury ve vztahu k operacím, funkcím a podpůrným nástrojům, které zařízení podporuje. Přehled funkcí jednotlivých verzí výpočetních schopností lze nalézt např. na [2].

Kernel značí funkci běžící po spuštění paralelně na GPU [9].

Svazkem (z angl. *warp* [28]) rozumíme 32 vláken běžících současně na jednom multiprocesoru grafické karty.

Jako CUDA *blok* (z angl. *block*) označujeme sdružení vláken vykonávajících stejnou instrukci paralelně s možností jejich vzájemné synchronizace a sdílené paměti. *Mřížkou* (z angl. *grid*) rozumíme sdružení bloků určené právě pro jeden GPU čip.

1.2 Kryptologie

Kryptologie je věda, zabývající se utajováním a skrýváním informací (avšak i jejich následným získáváním) a dělí se na kryptografii a kryptoanalýzu. *Kryptografie* se pak zabývá tvorbou zašifrovaných zpráv a *kryptoanalýza* naopak jejich následným zpětným získáváním.

Čistý text (z angl. *plaintext*) značí nezašifrovaná data v původní formě a také data po dešifrování. Oproti tomu *zašifrovaným textem* rozumíme data zašifrována pomocí určité šifry.

Šifrování je proces, při kterém je čistý text transformován na text zašifrovaný pomocí šifrovacího algoritmu (šifry). Jedná se o reverzibilní proces, při kterém se data nesoucí nějakou informaci transformují na data připomínající náhodný soubor znaků při zachování původní informace. K jejich dešifrování je pak třeba znát určitou specifickou (tajnou) informaci zvanou klíč. *Dešifrováním* je rozumněn zpětný proces, tedy získání čistého textu ze zašifrovaného textu.

Jako *klíč* je označena speciální informace, pomocí níž jsou data šifrována a dešifrována. Bez znalosti správného klíče se po spuštění dešifrovacího algoritmu změní data opět na (pseudo)náhodnou sekvenci znaků. Jako *podklíč* označujeme části klíče pro dané kolo šifry a který je získán algoritmem plánování klíčů (z angl. key-scheduling).

Padding je způsob vyplnění prázdných bitů v bloku, aby šifra mohla pracovat vždy s celým blokem.
Podrobnější informace ohledně zmíněných pojmů lze nalézt například v [21].

1.3 Shrnutí

V této kapitole byly vysvětleny základní pojmy z oblasti programování grafických karet a kryptologie. Nabyté poznatky budou využity v dalších kapitolách, kde s nimi bude pracováno již v kontextu jejich využití.

Kapitola 2

Programovatelné grafické procesory

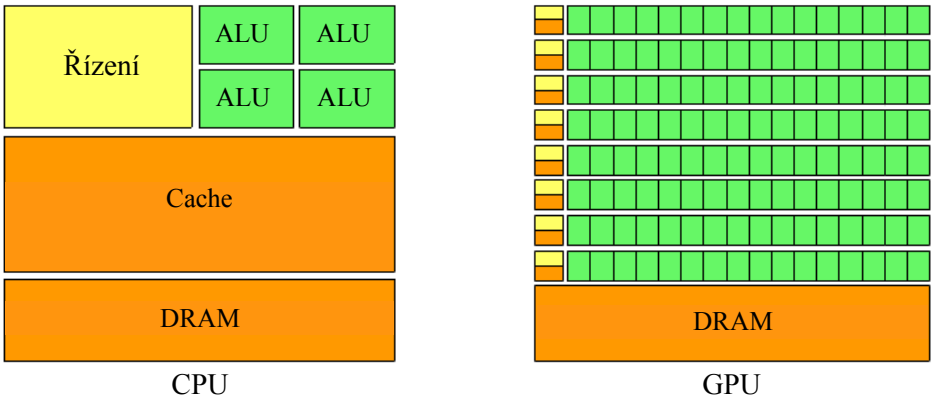
Tato kapitola se zaměřuje na stručný popis architektur CPU a GPU a jejich srovnání z pohledu možností, jež obě poskytují pro programování obecných výpočtů. Dále je uveden rozbor architektury CUDA a základních principů využívaných při programování GPU.

2.1 Architektura

2.1.1 Srovnání CPU a GPU

Jako první je nezbytné si uvědomit, že primární úkoly CPU a GPU jsou rozdílné. Již z tohoto faktu lze jednoznačně usuzovat, že i jejich architektura je rozdílná a každá z nich se hodí na jiné typy úloh. Zatímco CPU primárně tvoří jádro počítače, kde jsou zpracovávány všechny instrukce vedoucí k požadovanému výsledku, primární úlohou GPU je pomáhat CPU s intenzivními výpočty pro správné zobrazení obrazu na monitoru. Toto rozdělení je výsledkem historického vývoje. S dřívějšími 8bitovými grafickými výpočty CPU většinou nemělo problémy, avšak s nástupem 16bitových a 32bitových hodnot pro zobrazení barev začala zátěž na CPU neúměrně růst. Proto vznikly grafické akcelerátory, jejichž primárním cílem bylo ulehčit právě této zátěži na straně CPU. V současnosti tak grafické karty excelují ve výpočtech v plovoucí čárce a na tuto funkci se postupně nabalují další a další funkcionality, zejména grafického charakteru (například podpora přehrávání HD videa).

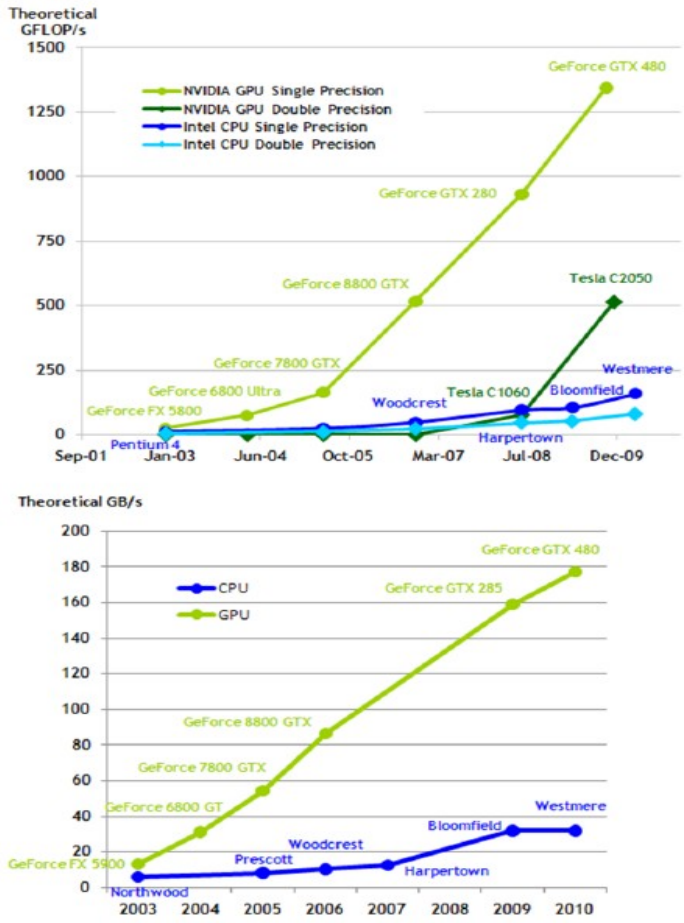
Z předchozích faktů poté vyplývá, že grafická karta (potažmo GPU) je natolik specializovaný hardware, že nedokáže zvládnout všechny úlohy, které umí řešit CPU. Naopak GPU zvládá úlohy pro které je určeno mnohem rychleji než CPU.



Obrázek 2.1: Srovnání CPU a GPU z hlediska architektury.

Popišme si tedy hlavní rozdíly, které od sebe CPU a GPU (obr. 2.1) odlišují a proč platí výše zmíněné vlastnosti. CPU využívá několika jader v řádech jednotek (až desítek), oproti tomu GPU využívá několika desítek multiprocesorů. Už toto naznačuje, že u CPU převládá kontrola

a optimalizace běhu nad propustností dat, přičemž u GPU je důraz kladen právě na propustnost dat. Toto souvisí také s dalším hlavním rozdílem, kterým je spouštění instrukcí. Pro CPU je typické spouštění instrukcí mimo pořadí (out-of-order) [5] a pro GPU je naopak typické spouštění instrukcí v pořadí (in-order) [5]. Z toho také vyplývá, že paralelismus na CPU může do značné míry divergovat v rámci jednotlivých vláken. Oproti tomu divergence vláken u GPU může způsobit značný pokles výkonu právě z důvodu výrazného rozdílu v použité architektuře. Dalším rozdílem je typ použitého paralelismu, což má opět návaznost na počet aritmeticko-logických jednotek a na jejich uspořádání. Podle Flynnovy taxonomie [6] můžeme tvrdit, že CPU využívá MIMD (více instrukcí, více dat, z anglického Multiple Instruction Multiple Data) [6] a SIMD (jedna instrukce, více dat, z angl. Single Instruction Multiple Data) [6] pro krátké vektory a GPU využívá SIMT (jedna instrukce, více vláken, z angl. Single Instruction Multiple Threads) pro dlouhé vektory [7]. Posledním podstatným rozdílem je velikost cache paměti – zatímco CPU využívá velké cache paměti, u GPU bývá zpravidla podstatně menší cache, která je většinou určena pouze pro čtení (v současnosti však již některé GPU obsahují hierarchii programovatelných cache pamětí [8]). Toto téma souvisí s hierarchií pamětí, která je rozebrána dále.



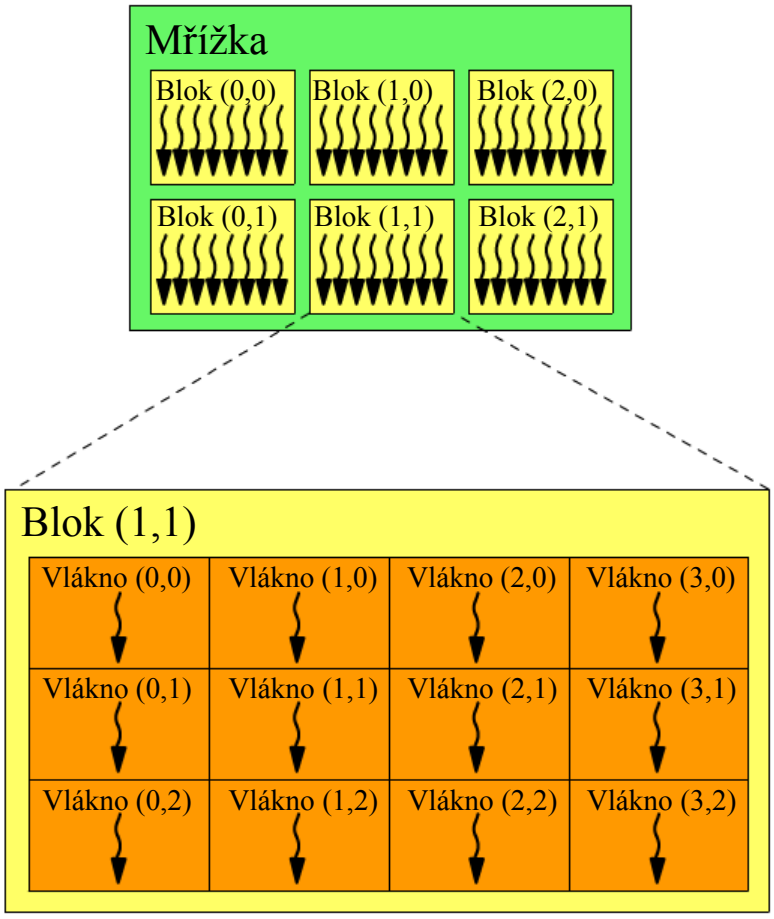
Obrázek 2.2: Srovnání CPU a GPU podle počtu operací v plovoucí čárce za sekundu [9].

Shrnutím předchozích rozdílů lze dostat, že GPU je sice velmi výkonné oproti CPU, avšak méně univerzální, protože obětovává cache paměť a řízení běhu za zvýšený výkon. Pro názornost tohoto rozdílu je na obrázku 2.2 vidět porovnání maximálního počtu instrukcí pro vybrané CPU a GPU.

2.1.2 CUDA [2, 7]

Jak již bylo zmíněno v úvodu jedná se o paralelní architekturu využívající grafické karty k obecným výpočtům. Pro tento účel CUDA podporuje více programovacích jazyků (např. C, Fortran, OpenCL) a pro mnohé další lze nalézt neoficiální podporu (.NET, Java, Ruby). Než si ukámeme psaní programů využívajících grafických karet, tak je nutné popsat základní části použité architektury z důvodu lepší představy možností, které nám programovatelné grafické akcelerátory nabízí.

V předchozí části bylo uvedeno, že GPU využívá SIMT model a nyní si uvedeme, co tento pojem v praxi znamená. Jedná se o stovky až tisíce vláken běžících zároveň s ještě větším množstvím dalších vláken čekajících na spuštění. Všechna vlákna ovšem provádí stejné instrukce ve stejnou dobu, pouze nad různými daty. S tím souvisí hierarchie vláken (obr. 2.3) a škálování jejich počtu [10].



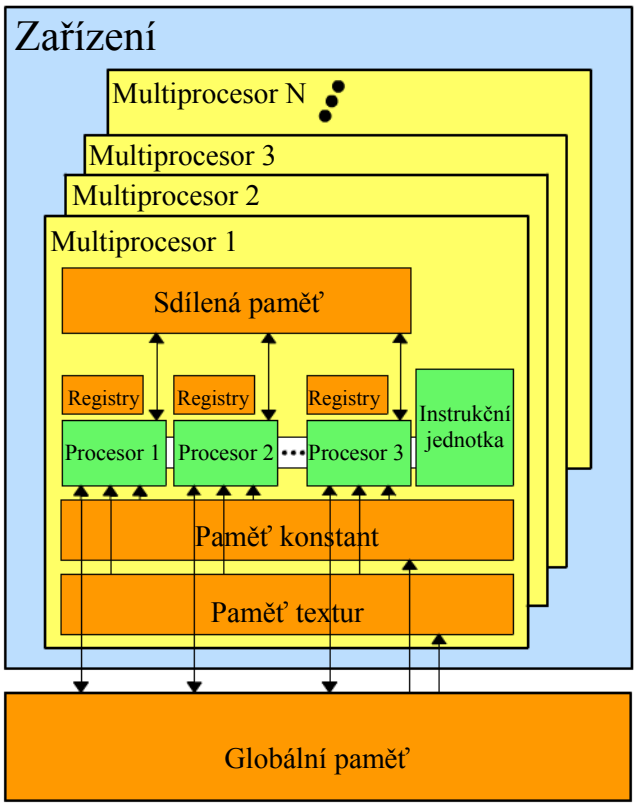
Obrázek 2.3: Hierarchie vláken v architektuře CUDA.

2.1.2.1 Hierarchie vláken

Vlákno tvoří základní prvek této hierarchie. V kontextu programovatelných grafických akcelerátorů by však vlákno mělo být mnohem „lehčí“ než jeho CPU ekvivalent – každé vlákno by mělo dělat pouze malou část celkové úlohy (nejvýše několik desítek instrukcí) a nemělo by divergovat ve svém běhu vzhledem k ostatním vláknům. Vlákna jsou rozdělena do CUDA bloků. Jedná se o bloky sdružující vlákna, která mohou běžet na sobě nezávisle a tedy paralelně. Každý blok umožňuje synchronizaci vláken v něm obsažených a poskytuje jim sdílenou paměť. Jednotlivé bloky jsou poté shluknuty do mřížky. Ta udává sdružení všech bloků, které vykonávají stejné instrukce, a velikostně tak odpovídá problému pro jeden čip grafické karty. Blok velikostně odpovídá problému pro jeden multiprocesor a nikdy tak nemůže být vykonáván na více multiprocesorech. Může se však stát, že jeden multiprocesor bude zpracovávat více bloků. Vlákna jsou v rámci multiprocesoru přiřazena na jednotlivé procesory pro proudové zpracování dat, které tvoří jádra multiprocesorů. Tato hierarchie nám, kromě hlubšího pochopení jak jsou vlákna zpracovávána na GPU, také umožňuje identifikovat jednotlivá vlákna v rámci celé mřížky.

2.1.2.2 Hierarchie pamětí [11, 12]

Dalším stavebním kamenem nutným pro efektivní programování na GPU je znalost hierarchie pamětí.

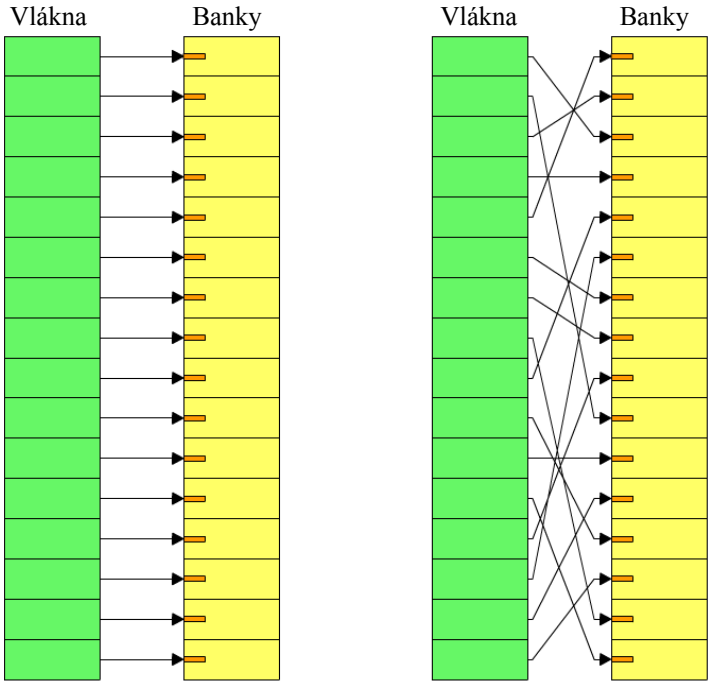


Obrázek 2.4: Hierarchie pamětí v architektuře CUDA.

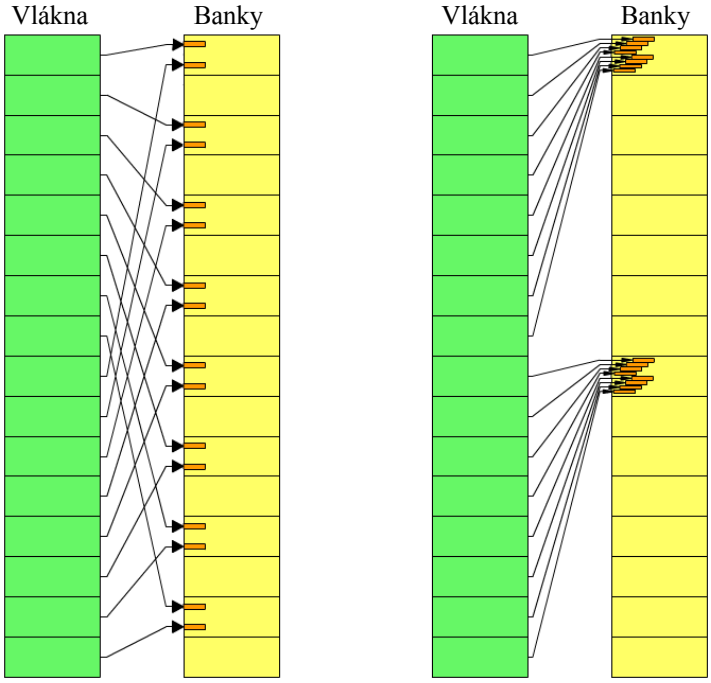
Na obrázku 2.4 lze nalézt pět základních typů paměti – registry, sdílenou paměť, paměť konstant, paměť textur a globální paměť (paměť zařízení – grafické karty). Ná následujících řádcích je zmíněno, jak se od sebe liší, jaké mají vlastnosti a jaké rozličné funkce zastávají.

Nejprve budou představeny *registry*, které tvoří lokální paměť každého procesoru. Obdobně jako u CPU se jedná o velice rychlou paměť, která je však velice malá. Používá se pro ukládání lokálních proměnných a mezivýsledků běhu jednotlivých vláken. Při nedostatku místa v registrech je použita tzv. lokální paměť. Označení lokální je poněkud zavádějící, protože fyzicky je umístěna v globální paměti a je tedy velice pomalá v porovnání s registry. Proto je důležité dbát na to, aby vlákna nepoužívala nadměrné množství lokálních proměnných, což opět souvisí s jejich „lehkostí“. Registry mají životnost threadu.

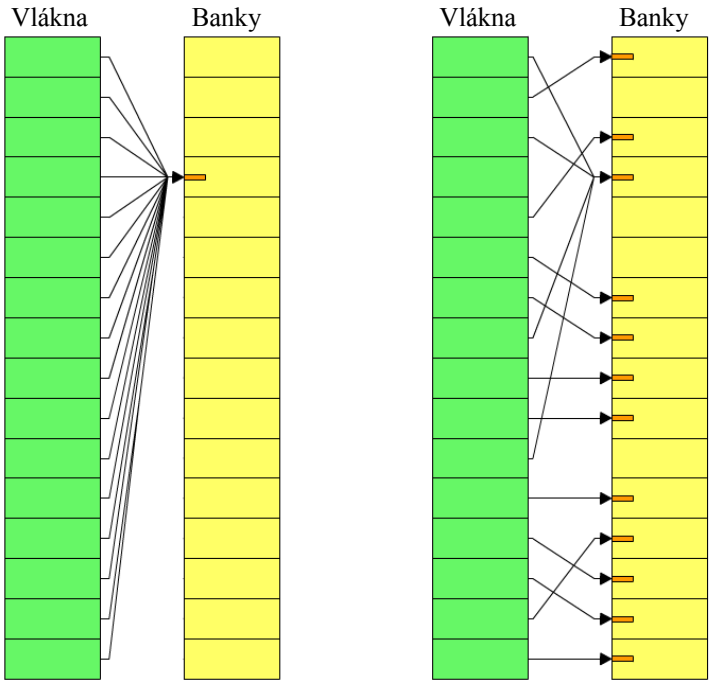
Dalším typem paměti je *sdílená paměť*, která se váže k multiprocesoru. Každý multiprocesor má svoji vlastní sdílenou paměť a každé vlákno v bloku běžícím na daném multiprocesoru může přistupovat k této sdílené paměti. Výhodami sdílené paměti jsou rychlost (za určitých podmínek jako u registrů) a možnost přístupu z libovolného vlákna v daném bloku. Aby však byl potenciál sdílené paměti využit naplno, je nutné předcházet konfliktům banků (obr 2.6) [13]. Sdílená paměť je totiž rozdělena na tzv. banky, což jsou paměťové moduly konstantní velikosti. Jedná se uspořádání, ve kterém po sobě jdoucí 32bitová slova jsou ukládána do po sobě jdoucích banků. Banků je 16 pro grafické karty s výpočetními schopnostmi verze 1.x a 32 pro zařízení s výpočetními schopnostmi verze 2.x a vyšší. Konfliktem banků pak rozumíme stav, kdy přistupuje více vláken ke stejným bankům (viz obrázky 2.5, 2.6 a 2.7). Zde je nutné také rozlišit zařízení s rozdílnými výpočetními schopnostmi (verze 1.x a 2.x), protože díky rozdílům v architektuře se za určitých okolností chovají jinak [9]. Zařízení s výpočetními schopnostmi verze 1.x zvládají broadcast, když všechny vlákna daného bloku přistupují k jedinému banku. V takovém případě je použita jediná instrukce a všechna vlákna dostanou požadovaná data. Pokud však jediné vlákno přistoupí na jinou adresu, tak dochází ke konfliktu a k serializace celé operace. Zařízení s výpočetními schopnostmi 2.x jsou na tom vývojově dále a zvládají i násobný broadcast v rámci jedné transakce. Konflikt tak může vzniknout pouze v případě, že dvě nebo více vláken přistupuje k různým 32bitovým slovům ve stejném banku. Pokud přistupují k různým bytům v rámci stejného slova, pak žádný konflikt nevzniká. Sdílená paměť má životnost bloku.



Obrázek 2.5: Přístup bez konfliktů banků.



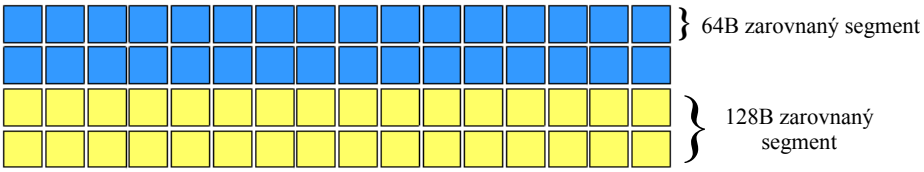
Obrázek 2.6: Vícecestné konflikty banků.



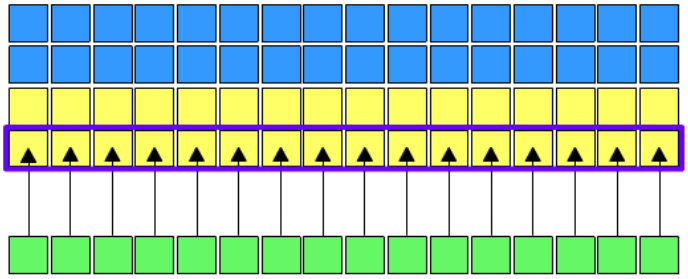
Obrázek 2.7: Typy broadcastu jednoho banku mezi více vláken.

Třetím typem paměti v architektuře CUDA je *globální paměť* zařízení. Jedná se o paměť typu DRAM [22] a jako taková je výrazně pomalejší než výše uvedené paměti (stovky cyklů GPU). Její výhodou je velikost. Před spuštěním kernelu je nutné vždy nahrát potřebná data do globální paměti, aby s nimi kernel mohl při výpočtu pracovat. K této paměti je poté důležité přistupovat z kernelu zarovnaně (pro zařízení s výpočetními schopnostmi verze nižší než 2.0). Pořadí jednotlivých vláken při zarovnaném přístupu však může být permutované (viz obrázek 2.9). Pokud je ovšem přistupováno do globální paměti nezarovnaně (obrázek 2.10, 2.11), může docházet k drastickému poklesu výkonu. Globální paměť je totiž rozdělena do 64bytových segmentů, které jsou sdruženy po dvou do 128bytových segmentů (obrázek 2.8). Aby nedocházelo k serializaci přístupu do paměti, je nutné přenášet velká slova najednou, přičemž jedna transakce může přenášet pouze 32bytová, 64bytová nebo 128bytová slova. Obecně lze říci, že čím větší výpočetní schopnosti zařízení, tím lépe zařízení zvládá nezarovnaný přístup. Pro nová zařízení architektury Fermi [8] (výpočetní schopnosti 2.0 a vyšší) jsou přidány dvě úrovně paměti cache, které značně urychlují a usnadňují práci s globální pamětí.

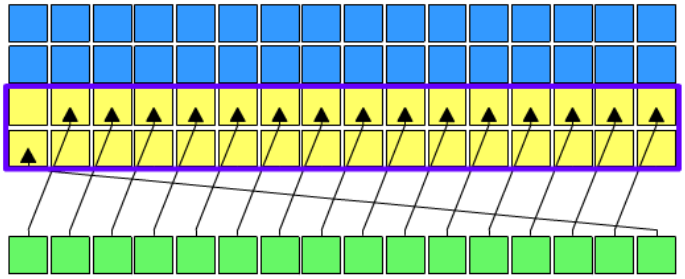
Zajímavým jevem u globální paměti je tzv. partition camping [14]. U starších grafických karet (výpočetní schopnosti 1.x) občas nastává ztráta propustnosti dat v případě, že je některá část globální paměti využívána více než zbývající. Globální paměť je rozdělena do několika regionů, které vzdáleně mohou připomínat banky u sdílené paměti. V případě, že program z nějakého důvodu využívá pouze určité části regionů či celé regiony na úkor zbytku, může nastat zpomalení běhu programu. Obecně se hůře odlaďuje než konflikty banků, protože může nastávat pro běžící program jen za určitých podmínek (například sudé rozměry matice). Tento jev však není natolik zpomalující jako konflikty banků. Poslední poznámkou k této části je, že globální paměť má životnost celé aplikace.



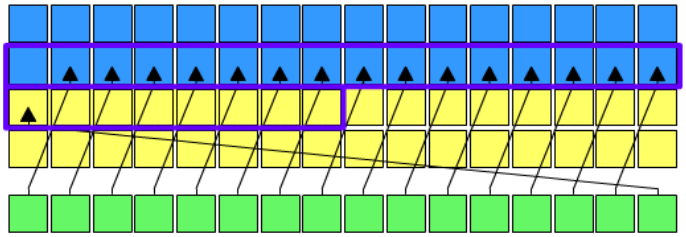
Obrázek 2.8: Organizace globální paměti



Obrázek 2.9: Zarovnaný přístup do globální paměti bez permutace přístupu jednotlivých vláken.



Obrázek 2.10: Nezarovnaný přístup do globální paměti (s využitím jedné transakce pro zařízení s výpočetními schopnostmi 1.2 a vyšší).



Obrázek 2.11: Nezarovnaný přístup do globální paměti s využitím dvou transakcí.

Jako další typ paměti je popsána *paměť konstant*. Jedná se o paměť určenou pouze pro čtení, a přestože je fyzicky umístěna v globální paměti zařízení, je optimalizována na rychlost přístupu (s využitím paměti cache). Tato paměť je velice rychlá, a pokud všechna vlákna čtou ze stejného místa, dosahuje rychlosti registrů. V opačném případě dochází k serializaci a zpomalení celého výpočtu až na rychlost globální paměti. Zapisovat do paměti konstant může pouze hostitelské zařízení (typicky běžné PC) voláním speciální funkce. Velikost paměti konstant je omezena na 64 KB. Tato paměť má životnost celé aplikace, lze ji tedy používat pro více kernelů bez nutnosti její opětovné inicializace. Tato vlastnost společně s rychlostí z ní dělá velice užitečný nástroj. Stejně jako globální paměť má životnost celé aplikace.

Dalším speciálním typem paměti je *paměť textur*. Jedná se v podstatě o namapování části globální paměti do paměti jednotky textur (z angl. Texture Unit), která tak v podstatě tvoří cache pro tuto část dat. Neplatí pro ni omezení spojitého přístupu a lze ji tedy využít právě jako částečnou náhradu cache paměti. Paměť textur má více adresovacích módů (1D, 2D a 3D fetch) a nejlepších výsledků dosahuje, pokud jsou požadovaná data umístěna blízko sebe v globální paměti. I tato paměť má životnost celé aplikace.

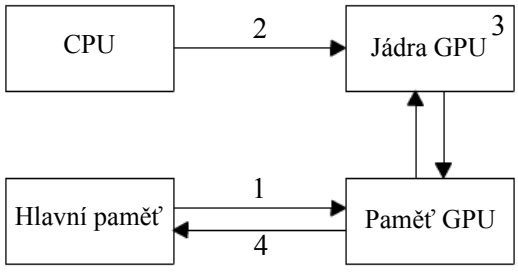
Tímto výčet různých typů pamětí končí a lze se tak přesunout k poslední části architektury CUDA, ve které si popíšeme proces výpočtu.

2.1.2.3 Proces výpočtu [2]

Přestože je využití programovatelných grafických procesorů pro obecné výpočty poměrně jednoduchým procesem (obr. 2.12), je dobré si zmínit jednotlivé kroky nutné k úspěchu.

Základním poznatkem je uvědomit si, že kód určený pro hostitelské zařízení a pro grafický akcelerátor je nutné od sebe oddělit. Pro kód běžící na GPU se zažil pojem kernel. Protože však kernel neumí pracovat s pamětí hostitelského zařízení, ale pouze s pamětí grafické karty, je nutné před výpočtem alokovat paměť na grafické kartě a potřebná data přehrát z paměti hostitele. Po úspěšné alokaci paměti a zkopírování potřebných dat lze spustit výpočet na GPU. Výsledky běhu výpočtu budou zpravidla opět uloženy v paměti zařízení. Po skončení běhu kernelu lze přehrát získaná data z paměti grafického akcelerátoru zpět do paměti hostitelského zařízení a alokované místo v paměti grafické karty opět uvolnit.

Protože přehrávání dat z paměti pracovní stanice do paměti zařízení či naopak je velmi pomalé, je dobré snažit se překrýt tyto operace jinými výpočty na GPU, aby nedocházelo ke ztrátě výkonu. Tato metoda ovšem nemusí vždy fungovat, a protože se jedná o poněkud složitější koncept, nebudeme se jím v této práci zabývat.



Obrázek 2.12: Proces výpočtu na GPU.

2.2 Základy programování CUDA C

V této kapitole jsou popsány základní konstrukce používané při programování aplikací využívající grafické karty. Zaměřuje se na jazyk CUDA C, který je rozšířením jazyka C právě o příkazy umožňující využití grafických karet k akceleraci výpočtu.

2.2.1 Kernel

Základem každého programu využívajícího programovatelné grafické akcelerátory je kernel, což je funkce běžící na GPU. Jedná se o instrukce prováděné vlákny paralelně. Protože se jedná o jiný kód, než ten určený pro CPU, je potřeba jej v programu odlišit. K tomu je využito kvalifikátoru (z angl. qualifier) `__global__`, který je deklarován před metodu určenou jako kernel (viz příklad 2.1). Tím je překladači dáno na vědomí, že tuto metodu je třeba přeložit právě pro běh na GPU.

```
__global__ void SoucetVektoru(float* A, float* B, float* C){

     int i = blockIdx.x * blockDim.x + threadIdx.x;
     C[i] = A[i] + B[i];
}
```

Příklad 2.1: Deklarace kernelu – metody běžící na GPU.

Existují i další kvalifikátory určující, pro které zařízení je metoda určena. Jak již bylo naznačeno výše, metody označené kvalifikátorem `__global__` lze volat pouze z kódu hostitelského zařízení a jejich překlad je určen pouze pro běh na GPU. Kvalifikátor `__device__` pak značí metodu překládanou opět pouze pro GPU, ale kterou lze volat pouze z kódu určeného pro GPU. Posledním kvalifikátorem tohoto typu je `__host__`, který značí, že metoda je kompilována pouze pro pracovní stanici a lze ji volat pouze z kódu hostitelského zařízení. Pokud při definici metody uvedeme jako kvalifikátor jak `__device__`, tak i kvalifikátor `__host__`, bude tato metoda kompilována pro oba typy zařízení.

Nyní již máme program (či jeho část) určenou pro GPU definován, je možné tedy přistoupit k jeho spuštění. Protože se jedná o kód běžící na GPU, je i pro tuto část funkcionality definován speciální operátor. V předchozích částech byla popsána hierarchie vláken, což objasnilo, jakým způsobem bude následující část probíhat. Spuštění kernelu probíhá pomocí volání `jménoKernelu<<<početBloků, počet vláken v bloku>>>(vstupní parametry)`. Pro kernel definovaný v příkladě 2.1 by vypadalo jeho spuštění jako v příkladě 2.2.

```
int main(){
     ...
     SoucetVektoru<<<N/VelikostBloku + 1, VelikostBloku>>>(A, B, C);
     ...
}
```

Příklad 2.2: Spuštění kernelu.

Posledním důležitým chybějícím článkem týkajícím se spouštění kernelů je výpočet jedinečného identifikátoru pro každé spuštěné vlákno. Protože pro značný počet programů je práce s jednotlivými vstupy založena na postupném zpracovávání prvků polí, je důležité umět jednoznačně identifikovat jednotlivá vlákna. K této identifikaci je využita právě hierarchie vláken, která byla uvedena v předchozí části (kapitola 2.1.2.1). Výpočet zmíněného identifikátoru je velmi jednoduchý a byl již naznačen v příkladu 2.1, kde však nebyl nijak komentován. Jedná se o triviální výpočet umístění vlákna v bloku a umístění bloku v mřížce. Pro jednorozměrné bloky je situace stejná jako v příkladě 2.1, tedy

identifikátor = pořadí bloku * velikost bloku + pořadí vlákna, (1)

což po převedení do CUDA C bude vypadat takto

```
int id = blockIdx.x * blockDim.x + threadIdx.x;
```
(2)

V případě jiného rozdělení bloků (např. dvourozměrného) probíhá výpočet identifikátoru podle potřeby výpočtu (např. stejným výpočtem pro souřadnici `y`).

Tyto znalosti jsou dostatečné k úspěšnému spuštění kernelu na GPU a jsou dobrým základem pro vstup do psaní kódu pro GPU. V kapitole o procesu výpočtu bylo uvedeno, že před každým výpočtem je nutné nejprve nahrát data určená ke zpracování do paměti grafické karty, aby s nimi bylo možné pracovat.

2.2.2 Práce s pamětí

2.2.2.1 Deklarace a alokace

V předchozích částech byla uvedena hierarchie pamětí. Následují ukázky, jak s jednotlivými typy pamětí pracovat v CUDA C a jakým způsobem lze jednotlivé paměti využít. Vše je demonstrováno na názorných příkladech.

Obdobně jako v kapitole hierarchie pamětí se nejprve podíváme na globální paměť. Již víme, že je oddělena od paměti CPU a pokud tuto paměť chceme využít, je nutné ji nejdříve alokovat. Zde existuje paralela s pamětí CPU, jelikož je možné alokovat globální paměť staticky či dynamicky. Pokud se rozhodneme pro statickou alokaci paměti, stačí před proměnnou uvést kvalifikátor `__device__` (viz příklad 2.3) a překladač pak automaticky zajistí umístění této proměnné v paměti GPU. Pro dynamickou alokaci paměti je v jazyce CUDA C rezervován příkaz `cudaMalloc(void ** devPtr, size_t size)`, jehož parametry jsou ukazatel na tuto část paměti a její velikost (viz příklad 2.4).

```
__device__ float d_A[128];
```

Příklad 2.3: Statická deklarace proměnné v globální paměti GPU.

```
int size = 16;
float * d_A;
cudaMalloc(&d_A, size * sizeof(float));
```

Příklad 2.4: Dynamická deklarace proměnné v globální paměti GPU.

Deklarace proměnných v ostatních typech pamětí probíhá obdobně jako u paměti globální. Pro sdílenou paměť je vyhrazen kvalifikátor `__shared__` pro statickou i dynamickou alokaci. Rozdíl při deklaraci je pak vidět na příkladech 2.5 a 2.6, přičemž je dobré poznamenat, že v příkladě 2.6 je nutné dodržovat offsety, aby se adresy deklarovaných polí nekryly.

```
__shared__ float array[128];
```

Příklad 2.5: Statická alokace sdílené paměti GPU.

```
extern __shared__ sharedArray[];
float *array0 = (float*)sharedArray;
int *array1 = (int*)&array0[128];
short *array2 = (short*)&array1[64];
```

Příklad 2.6: Dynamická alokace sdílené paměti GPU.

Při použití dynamické alokace sdílené paměti je však nutné znát její celkovou velikost před spuštěním kernelu, protože je potřeba uvést tuto hodnotu při jeho spuštění.

Další pamětí v pořadí je paměť konstant, kterou nelze alokovat dynamicky. Proto je vždy deklarována pomocí kvalifikátoru `__constant__` (příklad 2.7). Je důležité mít na paměti, že se jedná o paměť pouze pro čtení a všechna data musí být nahrána do paměti konstant před spuštěním kernelu, který ji má využívat.

```
__constant__ float constantArray[128];
```

Příklad 2.7: Deklarace paměti konstant.

Paměť textur přesahuje rámec této práce, a protože není v praktické části využita, není zde popsána.

2.2.2.2 Kopírování dat

Alokace paměťového místa sama o sobě nestačí, protože proměnná ihned po alokaci neobsahuje žádná data. Proto je třeba do alokovaného prostoru potřebná data nahrát. K tomu jsou v CUDA C vyhrazeny speciální metody, které zajišťují přenos dat mezi různými druhy pamětí.

Základním příkazem zajišťujícím přenos dat je `cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)`, jehož parametry jsou ukazatel

na cílovou adresu, ukazatel na zdrojovou adresu, velikost přenášených dat a typ přenosu (příklad 2.8). První tři parametry není třeba vysvětlovat, ovšem čtvrtý parametr může být na první poněkud zavádějící. Jedná se o pouhé určení, zda se data budou přenášet z paměti hostitele do paměti hostitele (`cudaMemcpyHostToHost`), či do paměti zařízení (`cudaMemcpyHostToDevice`), nebo z paměti zařízení do paměti zařízení (`cudaMemcpyDeviceToDevice`), či do paměti hostitelské stanice (`cudaMemcpyDeviceToHost`).

Pro paměť konstant je používán příkaz `cudaMemcpyToSymbol(const char *symbol, const void *src, size_t count, size_t offset, enum cudaMemcpyKind kind)`, který obdobně jako v předchozím případě kopíruje data dané velikosti ze zdrojové adresy do cílové adresy. Ta je v tomto případě definována jako ukazatel na konstantu a offset v bytech. Protože se paměť konstant vždy nachází v zařízení, je nutné použít jako typ přenosu pouze `cudaMemcpyHostToDevice` nebo `cudaMemcpyDeviceToDevice`.

V CUDA C existuje ještě mnoho jiných příkazů pro kopírování paměti, avšak pro obsah této práce opět není jejich popis a význam důležitý. Pro zájemce je však možno snadno dohledat jejich podrobný výpis například v [16].

2.2.2.3 Práce s daty a dealokace

V tento moment jsou již v paměti potřebná data pro běh kernelu a nic tedy nebrání v jeho spuštění. Použití alokovaných pamětí a potažmo jejich obsahu se žádným způsobem nevymyká standardní práci s proměnnými, jen je třeba při kopírování dat dát pozor na správnou alokaci, kopírování i volání jednotlivých proměnných.

Po úspěšném běhu programu máme výsledná data opět uložena v použité paměti a k jejich získání použijeme opět výše zmíněné metody kopírování dat, tentokrát však v opačném směru (`cudaMemcpyDeviceToHost`). Pokud již nebudeme dále s dynamickou pamětí pracovat, je dobrou praktikou ji dealokovat. Pro tuto funkcionalitu existuje v CUDA C jednoduchý příkaz, který je velice podobný příkazu ve standardním jazyce C. Příkaz `cudaFree(void *devPtr)` uvolní dynamicky alokovanou paměť pro jiné využití a všechna data v této paměti jsou tímto příkazem zahozena. Tento příkaz funguje pouze pro paměti alokované pomocí `cudaMalloc()` a `cudaMallocPitch()`. Pro všechny ostatní ukazatele vrací chybové hlášení.

Protože správná práce s pamětí je klíčovou součástí programování v CUDA C, je dobré vždy testovat, zda kopírovaná data (v obou směrech) opravdu odpovídají požadovaným výsledkům. Většina chyb vychází právě ze špatné práce s paměťmi.

```
int size = 16;

// alokace paměti stanice
float *host_A = (float*)malloc(size);

// alokace paměti zařízení
float *device_A;
cudaMalloc(&device_A, size * sizeof(float));

// kopírování dat z paměti hostitele do paměti zařízení
cudaMemcpy(device_A, host_A, size, cudaMemcpyHostToDevice);
```

```
...

//spuštění kernelu

...

// kopírování dat (zpracovaných) dat z paměti zařízení zpět do
// paměti stanice
cudaMemcpy(host_A, device_A, size, cudaMemcpyDeviceToHost);

// uvolnění použité paměti zařízení
cudaFree(device_A);

...
```

Příklad 2.8: Demonstrativní použití `cudaMemcpy()` a `cudaFree()`.

2.3 Shrnutí

V kapitole byla popsána architektura GPGPU s užším zaměřením na architekturu CUDA a její srovnání s CPU. Byla ukázána hierarchie vláken a pamětí, kterých je využito při práci s architekturou CUDA a popsán proces výpočtu na GPU.

V druhé části se nacházelo seznámení se základními programátorskými konstrukcemi v CUDA C, které jsou nezbytné pro úspěšné psaní programů pro GPU architektury CUDA.

Kapitola 3

Využití GPU v kryptologii

V této kapitole se na úvod nachází seznámení s možnostmi využití GPU pro kryptologické (tj. kryptografické i kryptoanalytické) účely, jehož součástí je pohled na současný stav výzkumu v této oblasti z hlediska použitých postupů a dosažených výsledků. V závěru kapitoly je uvedeno, jaké jsou v oblasti využití grafických karet pro kryptologické účely predikce do budoucna. Tato část předpokládá znalost základních principů fungování blokových šifer a alespoň obecné povědomí o jednotlivých algoritmech (viz např. [22]). Záměrem této kapitoly není přinést komplexní a detailní výpis všech publikací na toto téma, ale uvést do problematiky z hlediska dosažených výsledků a jejich vývoje v čase.

3.1 AES

Jako první je uvedena jedna z nejdůležitějších šifer současnosti – AES (Advanced Encryption Standard). Tato šifra získala v oblasti akcelerace pomocí GPU velkou pozornost a zaměřuje se na ni netriviální množství prací. Z toho důvodu jsou vybrány pouze ty ilustrativnější přístupy pro potřeby této práce.

3.1.1 Naivní přístup

Základní princip paralelizace šifry s využitím jednoho vlákna na zpracování jenoho bloku šifry.

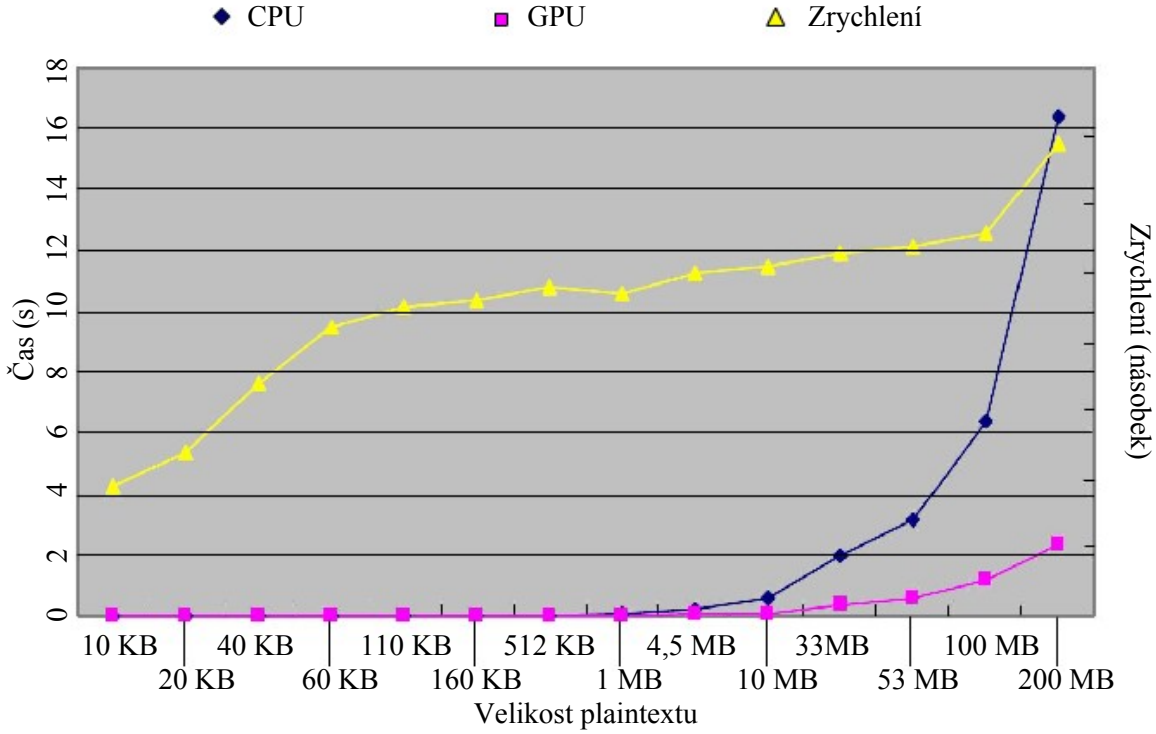
3.1.1.1 Princip akcelerace

Jako naivní se jeví přístup zmíněný v [17]. Před šifrováním je nahrán čistý text i podklíče pro jednotlivá kola do globální paměti zařízení. Poté je čistý text rozdělen do bloků délky 16 bytů a spuštěn výpočet. Při výpočtu jsou každému bloku nahrány potřebné klíče do sdílené paměti a každému vláknu příslušná část čistého textu. Není uvedeno, proč byla pro klíče upřednostněna sdílená paměť před pamětí konstant. Teoreticky by paměť konstant při takovém využití měla dosahovat v nejhorším případě zhruba stejného výkonu jako paměť sdílená.

Vlastní šifrování je realizováno na maticích 4x4 pomocí vyhledávacích tabulek [18]. Tento přístup je z hlediska použité architektury lepším řešením než sekvenční maticový výpočet. Pomocí čtyř tabulek o celkové velikosti 4 KB (každá tabulka obsahuje 256 4bytových čísel) a čtyř operací exkluzivního součtu tak lze realizovat jedno kolo šifrování pro jeden sloupec matice. Jedno vlákno zpracovává jeden blok, a přestože autoři tvrdí, že v daném kole potřebují pouze 4 vyhledání v tabulkách a 4 operace exkluzivního součtu na celý blok, tak dokument [18] str. 18 toto popírá s tím, že je potřeba 4 vyhledání a 4 operace exkluzivního součtu na každý sloupec matice v daném kole. I přes tento nedostatek v teoretickém základu je však dosaženo znatelného zrychlení oproti ekvivalentní CPU implementaci (viz dále).

3.1.1.2 Dosažené výsledky

Hardware na kterém byly provedeny testy je Intel Core 2 Duo E8200 s 1GB RAM a NVIDIA GeForce GTS250 běžící na Windows XP. Dosažené výsledky jsou přehledně shrnuty na obrázku 3.1 [17]. Důležitým poznatkem je, že skutečná výpočetní síla grafické karty se projeví až při větším objemu šifrovaných dat. I pro menší velikosti datových souborů je grafický akcelerátor poměrně rychlejší, avšak v absolutním čase je tento rozdíl zanedbatelný. I relativní poměr vyjadřující zrychlení roste s velikostí šifrovaných dat. Tento růst je dán faktem, že při větším zatížení je časová náročnost kopírování dat mezi paměťmi použitého výpočetního systému a grafické karty vykompenzována rychlostí výpočtu GPU.



Obrázek 3.1: Výsledky testů pro naivní implementaci.

3.1.2 Vylepšený naivní přístup [19]

3.1.2.1 Princip akcelerace

Úvodní studie je obdobná jako v předchozím případě, avšak zde je teoretická práce dovedena dále (přestože časově předchází naivní implementaci) a tedy i výsledky jsou znatelně lepší.

Tato práce naplno těží z možnosti paralelismu šifry AES [18], a proto jedno vlákno zajišťuje šifrování pouze jediné 4bytové části, tedy právě jednoho sloupce matice bloku šifry. Autor zmiňuje, že testováním bylo dosaženo nejlepších výsledků pro CUDA bloky velikosti 256 vláken, tedy šifrování

1024 bytů paralelně, a proto jsou prezentované výsledky právě pro tuto velikost CUDA bloku. Dalším vylepšením je využití paměti textur oproti sdílené paměti pro uložení rozšířeného klíče. Tímto krokem odpadne režie spojená s přenosem dat do sdílené paměti pro každý blok – paměť textur má životnost aplikace (viz kapitola 2.1.2.2).

3.1.2.2 Dosažené výsledky

Hardware, na kterém byly provedeny testy, je Intel Pentium IV (3.0 GHz) a NVIDIA GeForce 8800 GTX. CPU implementace, s níž je GPU implementace srovnávána, vychází z knihovny OpenSSL a srovnání bylo provedeno pro AES-128 a AES-256. Dosažené výsledky jsou shrnuty v tabulkách 3.2 a 3.3.

Velikost vstupního souboru		Čas GPU [ms]	Čas CPU [ms]	Zrychlení
2 KB	Čas výpočtu	0,15	<1	–
	Celkový čas	0,26		–
512 KB	Čas výpočtu	0,73	9	12,33
	Celkový čas	2,1		4,28
1 MB	Čas výpočtu	1,31	19	14,5
	Celkový čas	3,74		5,08
4 MB	Čas výpočtu	4,78	74	15,48
	Celkový čas	19,9		5,32
8 MB	Čas výpočtu	9,39	148	15,76
	Celkový čas	27,34		5,41

Tabulka 3.2: Výsledky pro šifrování šifrou AES-256.

Velikost vstupního souboru		Čas GPU [ms]	Čas CPU [ms]	Zrychlení
2 KB	Čas výpočtu	0,15	<1	–
	Celkový čas	0,27		–
512 KB	Čas výpočtu	0,6	9	15
	Celkový čas	1,9		4,74
1 MB	Čas výpočtu	1,04	19	18,23
	Celkový čas	3,5		5,43
4 MB	Čas výpočtu	3,8	74	19,47
	Celkový čas	13		5,69
8 MB	Čas výpočtu	7,55	148	19,6
	Celkový čas	25		5,92

Tabulka 3.3: Výsledky pro šifrování šifrou AES-128.

Z výsledků je patrné, že bylo docíleno znatelného zrychlení v obou případech a pro všechny velikosti vstupního souboru. Lze také vypozorovat růst zrychlení v závislosti na velikosti vstupního souboru, ze kterého lze usuzovat, že pro ještě větší vstupní soubory by mohlo být zrychlení pravděpodobně ještě větší (obdobně jako u naivního přístupu (viz kapitola 3.1.1)).

3.1.3 Komplexní analýza [20]

3.1.3.1 Princip akcelerace

V této práci jsou detailně rozebrány všechny různé přístupy k počtu vláken na jeden blok čistého textu i k využití různých typů pamětí pro jednotlivé složky šifry.

První metodou je šifrování 16 bytů pomocí jediného vlákna. Tento způsob nevyžaduje žádnou synchronizaci vláken ani žádná společná data ve sdílené paměti. Druhým přístupem je jedno vlákno pro 4 nebo 8 bytů – tato metoda však přidává režii spojenou se sdílenou pamětí a synchronizací vláken. Posledním zmíněným přístupem je jedno vlákno na 1 byte. Tato metoda je však využita pouze pro porovnání ostatních metod, protože nedokáže naplno využít výpočetních jednotek GPU kvůli malé délce dat.

Testování implementací zahrnovalo i metody využití různých typů pamětí. Pro T-boxy [18] a klíč bylo využito paměti konstant nebo sdílené paměti. Čistý text využívá globální paměti před zahájením vlastního šifrování, a poté je zkopírována potřebná část do sdílené paměti CUDA bloku, aby bylo dosaženo lepšího výkonu. Toto neplatí pro implementaci, kde je celý blok šifry zpracováván jedním vláknem, protože tento přístup dokáže pracovat pouze s registry a není tedy potřeba využívat jakoukoliv jinou paměť.

Zajímavou myšlenkou je také volba struktury pro ukládání čistého textu ve sdílené paměti. Pro snížení konfliktů banků pro jednotlivé implementace je využito buď uložení celého bloku do jednoho pole, nebo naopak uložení každé zpracovávané části čistého textu (1/4/8 B) do jednoho pole, přičemž v jednom poli jsou data ze stejných pozic několika bloků čistého textu. Pro každou implementaci je pak využito možnosti, která dává lepší výsledky.

Dalším důležitým prvkem je využití asynchronního čtení a zápisu globální paměti, čímž je snížena prodleva mezi paměťovými operacemi a vlastním šifrováním na minimum. Přesnější popis této části však chybí.

3.1.3.2 Dosažené výsledky

Pro tuto analýzu bylo využito následujícího hardwaru: Core i7 Quad i7-920 (2,66 GHz), 6 GB RAM, NVIDIA GeForce GTX 285 běžícího na CentOS 5.3. Implementovanou šifrou je AES-128, klíč i čistý text je generován CPU, a poté zkopírován do globální paměti zařízení. Velikost čistého textu je 256 MB. Běh je prováděn na 60 CUDA blocích o velikosti 512 vláken pro všechny implementace a vždy je využito 4 T-boxů. Dosažené výsledky jsou přehledně vidět v tabulce 3.4.

Z tabulky je patrné, že nejvyšší rychlosti dosahuje implementace 16 bytů na vlákno (35,2 Gbps). 16bytová implementace dosahuje nejvyššího průtoku dat díky absenci synchronizace jednotlivých vláken mezi sebou a absenci konfliktů banků, které by vznikaly při použití sdílené paměti. Čistý text je zpracováván výhradně v registrech, a to přináší zrychlení oproti ostatním implementacím.

Zajímavým faktem je snížení výkonu implementace, pokud je pro T-box využito paměti konstant. Protože se jedná o vyhledávání v tabulce, je přístup k jednotlivým prvkům náhodný a nahodilý a nedokáže využít potenciálu paměti cache, kterou paměť konstant disponuje. Oproti tomu

sdílená paměť vykazuje lepší vlastnosti pro potřeby náhodného čtení z ní. Rozdíl mezi využitím sdílené paměti a paměti konstant pro jednotlivé podklíče autoři udávají jako 2 %, což připisují způsobu čtení klíče, který je souvislý.

Důležitým poznatkem je skutečnost, že pro různé velikosti čistého textu, počty vláken v bloku a počty bloků mohou jednotlivé výsledky vycházet různě a různé implementace být pro konkrétní situaci lepší. V tomto výzkumu však byly tyto proměnné fixní s cílem zjistit nejvyšší možný výkon při fixních parametrech. I proto jsou výsledky uvedeny bez příslušného kopírování dat do a z paměti grafické karty. V případě započtení těchto operací klesá výkon implementace 16bytů na vlákno na 13,4 Gbps, což je poměrně markantní rozdíl. Při použití stejného principu (16B/vlákno) a překrytí čtení a zápisu paměti grafické karty s výpočtem dojde k navýšení výkonu na 22,0 Gbps.

Závěrem je pro GPU implementaci uvedeno zrychlení oproti CPU o faktor 28,39. Možnost vlivu zvoleného OS na výsledky autoři nijak nekomentují.

Bytů na vlákno	16	16	16	8	8	4	4	1	1
Paměť pro T-box	konst.	sdílená	sdílená	sdílená	sdílená	sdílená	sdílená	sdílená	sdílená
Paměť pro klíč	konst.	konst.	sdílená	konst.	sdílená	konst.	sdílená	sdílená	konst.
Výkon [Gbps]	5	34,4	35,2	26,9	23,9	25,3	25	2,9	2,6

Tabulka 3.4: Výsledky komplexní analýzy šifry AES-128.

3.2 DES

Další šifrou, na kterou se zaměříme, je DES (Data Encryption Standard). Protože se jedná dnes již o „zastaralou" šifru, je dobré ji zmínit alespoň pro úplnost. I přes některé dnes již překonané designové chybičky je stále dobrým ukazatelem možností akcelerace, zejména pak její bit-slicing implementace [23]. Pro téma šifry DES akcelerované grafickou kartou však neexistuje větší množství publikací, proto se zaměříme pouze na běžně dostupné práce.

3.2.1 Standardní DES [24]

3.2.1.1 Princip akcelerace

V práci [24] uvádí autoři dvě rozdílné implementace šifry DES pro grafickou kartu. Rozdíl mezi implementacemi je v tom, kde je prováděno odvozování podklíčů – jestli na CPU nebo na GPU. Důvodem tohoto kroku je nemožnost paralelizovat generování podklíčů a následné zkoumání vlivu tohoto faktu na efektivitu zbytku šifrovacího algoritmu. V obou případech byla jako referenční CPU verze použita volně dostupná implementace [26].

Pro GPU verzi algoritmu využívající CPU k plánování klíčů jednotlivých kol bylo pro uložení S-boxů i vygenerovaných podklíčů využito paměti konstant. Pro kompletní GPU verzi byl použit stejný algoritmus a stejné paměti. Rozdíl nastává ve využití vláken, kde vždy první dvě vlákna každého bloku obstarávají výpočet podklíčů pro danou instanci algoritmu. Hlavní klíč je nahrán v globální paměti grafické karty a vygenerované podklíče jsou uloženy ve sdílené paměti.

Datové soubory byly vygenerovány předem pomocí pseudonáhodné funkce v jazyce C#. Vygenerovaná data měla velikost 1 KB, 10 KB, 100 KB, 1 MB, 10 MB a 100 MB a byla použita

stejná pro všechny provedené testy. Klíč byl zvolen pravděpodobně náhodně a také zůstal neměnný pro všechny pokusy. Není uvedeno, jakých velikostí CUDA bloků bylo použito.

3.2.1.2 Dosažené výsledky

Pro testování byl použit následující hardware: Intel Core 2 Duo E8400 (3 GHz), 8 GB RAM, Nvidia Tesla C870 1.0 (1,35 GHz, 1,5 GB paměti) běžící na Windows XP SP3. Všechny implementace byly zkompilovány ve vývojovém prostředí Microsoft Visual Studio 2008 s CUDA verze 2.1. Do prezentovaných výsledků je započítáno pouze generování podklíčů, šifrování a nezbytné paměťové operace mezi hostitelským zařízením a grafickou kartou. Dosažené výsledky jsou shrnuty v tabulce 3.5.

Přestože tato práce není teoreticky nejlépe podložena (alespoň v její publikované části) přináší jeden důležitý důkaz ohledně akcelerace šifrovacích algoritmů, který lze (dle mého názoru) aplikovat na všechny šifry využívající algoritmus plánování klíčů – je výhodnější jej provést na CPU, a poté výsledek nahrát do paměti grafické karty než nechat grafickou kartu, aby podklíče pro jednotlivá kola generovala ona.

Vstupní soubor	Čas CPU [ms]	Čas GPU [ms]	Čas CPU/GPU [ms]	Zrychlení
1 KB	0,022	0,247	0,178	0,124
10 KB	0,179	1,39	1,23	0,146
100 KB	1,79	1,81	1,65	1,085
1 MB	18,2	6,24	5,59	3,256
10 MB	182,0	44,3	38,7	4,703
100 MB	1820,0	425,0	369,0	4,93

Tabulka 3.5: Výsledky akcelerace standardní implementace šifry DES.

3.2.2 Maximalizace počtu šifrovaných bloků za sekundu

3.2.2.1 Princip akcelerace

Základní úvahou první části práce [25] je, že maximální využití paralelizace nastává pouze v případě, že je dostatek datových bloků k šifrování. Taková skutečnost nastává například při šifrování pevných disků, nebo při útocích hrubou silou. Jako výchozí možnost pro testování autoři zvolili právě druhý případ, tedy útok hrubou silou.

Každé vlákno dostane stejný čistý text a rozdílný klíč a po zašifrování je provedena kontrola výsledku oproti správné hodnotě zašifrovaného textu. Vnitřní stav šifry je uložený v registrech, všechny konstanty potřebné pro běh algoritmu jsou uloženy v paměti konstant. Jedinou výjimku tvoří S-boxy, které nelze staticky nahrát do sdílené paměti, a proto jsou každému bloku nahrány do jeho sdílené paměti z paměti konstant. Globální paměti je využito pouze pro zápis správného řešení a běh výpočtu tedy nezpomaluje. Pro maximalizaci výkonu je výpočet prováděn na čistém a šifrovaném textu bez iniciální a závěrečné permutace, které jsou vypočítány na straně hostitele. Kernel tak pracuje pouze na 16 kolech šifry a výsledek porovnává s předzpracovaným šifrovaným textem. Kontrola výsledku probíhá asynchronně po každém spuštění kernelu, a proto nezvyšuje celkovou dobu běhu.

3.2.2.2 Dosažené výsledky

Experiment byl proveden na stroji Intel Core 2 Quad Q6600 s grafickou kartou NVidia GeForce GTX 260 (192 jader běžících na frekvenci 576 MHz, 896 MB paměti GDDR3). Jako sběrnice je použito PCI-Express 1.0, protože základní deska plně nepodporovala verzi 2.0. Operační systém je Gentoo Linux x86_64 verze 10.0 a použitý byl CUDA Toolkit verze 2.2. Výsledky jsou průměrem 300 měření a je v nich zahrnut čas potřebný na paměťové operace mezi grafickou kartou a použitou pracovní stanicí.

Výsledků bylo dosaženo na maximálním počtu CUDA bloků na mřížku, tedy 65 535, z důvodu snížení režijních nákladů na minimum. Z testů pro různé množství vláken na CUDA blok vyplývá, že již při 64 vláknech na CUDA blok dochází k maskování přístupů do sdílené paměti dostatečným nárůstem výkonu. Při více vláknech již nedochází k dalšímu výraznému nárůstu výkonu (viz tabulka 3.6). Nejrychlejší implementace využívá 128 vláken na CUDA blok, a díky tomu je rychlejší o faktor 5,6 oproti OpenSSL implementaci (viz tabulka 3.7).

Vláken na CUDA blok	Výkon [10^6 klíč/s]
32	61,34
64	75,12
128	75,33
192	72,64
256	75,2

Tabulka 3.6: Výsledky akcelerace útoku hrubou silou na šifru DES.

Výkon CPU [10^6 klíč/s]	Výkon GPU [10^6 klíč/s]	Zrychlení
13,32	75,33	5,66

Tabulka 3.7: Porovnání výkonu útoku hrubou silou na šifru DES pomocí CPU a GPU.

3.2.3 Útok hrubou silou na bit-slicing implementaci

3.2.3.1 Princip akcelerace

Protože se jedná o bit-slicing [23] implementaci je princip akcelerace poměrně složitý. Popsána tedy bude pouze hlavní myšlenka, nikoliv podrobný postup, který lze nalézt v [25].

Základem je využití registrů k maximalizaci výkonu algoritmu. Velikost registrů v architektuře CUDA je 32 bitů, a proto bylo zvoleno 32 bloků šifry pro simultánní běh. Do každého registru jsou nahrány n-té bity každého šifrovaného bloku čistého textu a při běhu je zpracován vždy celý registr.

Prvním krokem je rozložení čistého textu a odpovídajícího šifrovaného textu do 64 hodnot. Výsledek je uložen v paměti konstant, odkud je nahráván do sdílené paměti paralelně (každé vlákno nahrává jednu hodnotu).

Druhým prvkem je generování testovaných klíčů, které probíhá ve třech fázích. První část klíče

(nejvíce důležité bity) je počitadlo počtu spuštěných kernelů, které je předáváno jako parametr při spuštění kernelu. Druhou část tvoří výpočet z umístění bloku v mřížce, přičemž výpočet probíhá na grafické kartě. Poslední část klíče, kterou tvoří nejméně důležité bity, je jednoznačně dána indexem vlákna. Autoři tvrdí, že tento přístup byl experimentálně vyhodnocen jako výpočetně rychlejší než generování klíčů jiným způsobem (vliv generování klíčů je menší než 1 % času výpočtu).

Po vygenerování jsou klíče uloženy v registrech společně s vnitřním stavem šifry. Veškeré permutace, expanze a výběry byly upraveny tak, aby využívaly přejmenování registrů místo přiřazování proměnných z minimalizace režijních nákladů. Přesný popis způsobu využití registrů tímto způsobem však chybí.

Jako jediné operace vyžadující výpočet zbývají S-boxy. Autoři přistupují k S-boxům jako k nelineárním funkcím a jejich výpočet je tvořen logickými operacemi na bitové úrovni.

Na konci šifrování pak zbývá pouze porovnat výsledky se správným zašifrovaným textem, což se děje na grafické kartě. Výsledek porovnání uložený v globální paměti je přečten asynchronně hostitelským systémem, zatímco na grafické kartě běží další kernel. Pokud je klíč nalezen, je výpočetním systémem rekonstruován.

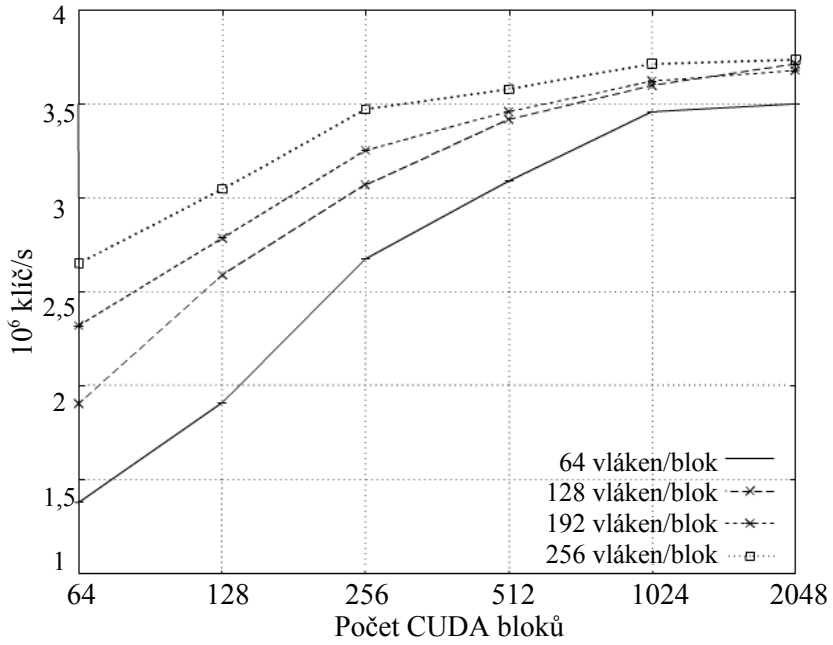
3.2.3.2 Dosažené výsledky

Experiment byl proveden na stejném hardware jako předchozí příklad (kapitola 3.2.2), tedy na stroji Intel Core 2 Quad Q6600 s grafickou kartou NVidia GeForce GTX 26, OS Gentoo Linux x86_64 verze 10.0 a CUDA Toolkit verze 2.2. Dosažené výsledky jsou i u této implementace průměrem 300 měření. Čas potřebný na paměťové operace mezi grafickou kartou a hostitelským zařízením je zahrnut.

Dosažené výsledky jsou shrnuty v obrázku 3.7. Je patrné, že bit-slicing implementace je více závislá na množství dat k dispozici. Maximální výkon 373.58 miliónů klíčů za sekundu dosahuje 10násobné rychlosti oproti referenční implementaci pro CPU (tabulka 3.8). Tu tvoří speciálně upravený kód využívající SSE2 instrukcí [27] a 128bitové registry v procesorech Intel Pentium 4 a vyšších. Oproti standardní implementaci šifry DES pro grafickou kartu pak dosahuje bit-slicing implementace zrychlení o faktor 5.

Výkon CPU [10^6 klíč/s]	Výkon GPU [10^6 klíč/s]	Zrychlení
36,85	373,58	10,138

Tabulka 3.8: Porovnání výkonu útoku hrubou silou na bit-slicing implementaci šifry DES pomocí CPU a GPU.



Obrázek 3.7: Výsledky akcelerace útoku hrubou silou na bit-slicing implementaci šifry DES.

3.3 A5/1

3.3.1 A5/1 Security Project [25]

3.3.1.1 Princip akcelerace

V rámci tohoto projektu byly vytvořeny Rainbow tabulky pro proudovou šifru A5/1. K jejich vygenerování bylo využito distribuované síly programovatelných grafických akcelerátorů. Na adrese [30] je popsán postupný vývoj v rychlosti generování zmíněných tabulek. Detailní popisy jednotlivých přístupů však chybí.

Ve všech následujících případech jsou výstupem A5/1 [31] 64bitové části proudu klíčů. Prvním přístupem je pouhé převedení kódu na architekturu CUDA, kde jedno vlákno zpracovává jeden A5/1 stav. Pro tento přístup je tedy zapotřebí 64 iterací ve smyčce pro každé vlákno. Vzhledem k absenci měření CPU ekvivalentu tohoto kódu je tento přístup považován za referenční.

První optimalizací je výpočet čtyř bitů jednoho proudu klíčů pomocí jednoho vlákna pomocí vyhledávacích tabulek emulujících použité logické operace. Přestože došlo k nárůstu počtu instrukcí ve vlákně, tak počet iterací v rámci vlákna klesl na 16, což vedlo k nárůstu výkonu.

Druhou optimalizací byl přechod na bit-sliced implementaci, tedy 64 různých částí proudu. Přestože se opět zvedl počet instrukcí na jednu část proudu klíčů, tak i tato implementace zajistila další nárůst výkonu.

Kvůli použitým typům instrukcí autoři vyzkoušeli použít také rozdílné architektury programovatelných grafických procesorů.

3.3.1.2 Dosažené výsledky

Díky distribuovanému přístupu nejsou dostupné informace o použitém hardware, kromě testů jednotlivých implementací prováděných na grafické kartě NVidia GeForce GTX 280 a ATI Radeon HD 5870.

Základní implementace dosahuje rychlosti 32 * 10^6 vygenerovaných 64bitových částí za sekundu na GTX 280. Po první optimalizaci bylo naměřeno přibližně 4.53násobné zrychlení, tedy 140-150 * 10^6 částí za sekundu opět na GTX 280. Implementace po druhé optimalizaci dosáhla při měření výkonu kolem 500 * 10^6 částí proudu klíčů za sekundu. Po převedení na zrychlení dostaneme přibližně 3,45násobné zrychlení oproti první optimalizaci a celkové zrychlení o faktor 15,63 oproti referenční hodnotě.

Po převedení bit-slicing implementace na grafickou kartu HD 5870 bylo dosaženo výsledku 1600 * 10^6 částí za sekundu. Jelikož se jedná o kartu technologicky dále než GTX 280, autoři uvádějí, že na kartě ekvivalentní GTX 280 (HD 4870) by bylo dosaženo zhruba polovičních výsledků (tedy 800 * 10^6 částí/s). Veškeré dosažené výsledky jsou shrnuty v tabulce 3.9.

Přínosem této práce (pomineme-li ten kryptografický přínos [32]) je zcela určitě skutečnost, že CUDA není jediná použitelná architektura a pro některé případy nemusí být nejlepším řešením.

Implementace	Výkon GTX 280 [10^6 částí/s]	Výkon HD 5870 [10^6 částí/s]	Zrychlení oproti základní implementaci	
Základní	32	–	–	–
První optimalizace	140 - 150	–	4,53	–
Bit-slicing	500	1600	15,63	50

Tabulka 3.9: Porovnání výkonu jednotlivých implementací generování rainbow tabulek pro šifru A5/1.

3.4 Shrnutí

Ve třetí kapitole byly představeny důležité principy akcelerace šifer AES, DES a A5/1, které jsou základními kameny pro akceleraci i jiných zde nezmíněných šifer (a obecně i některých nekryptografických algoritmů). Byly ukázány rozdíly výkonu při různých přístupech podložené praktickými experimenty i porovnání různých architektur.

Z rozebraných experimentů lze vypozorovat, že trend paralelizace pronikla již i do oblasti kryptologie. Vzhledem k úspěchům dosaženým na tomto poli tak lze předpokládat, že v budoucnu bude kladen ještě větší důraz na možnosti paralelizace jednotlivých aspektů této vědy.

Kapitola 4

KASUMI

V této kapitole je podrobně rozebrána šifra Kasumi, která je cílovou šifrou implementace v praktické části.

4.1 Design [33]

Kasumi je bloková šifra založená na principu tzv. Feistelovy sítě [34] a je tvořena osmi koly. Vstupními parametry jsou 64bitový vstupní blok, 128bitový klíč a výstupem je zašifrovaný 64bitový výstupní blok. Nejprve se blíže podíváme na jednotlivé komponenty potřebné pro šifrování a až poté bude detailně popsán celkový algoritmus.

4.1.1 Algoritmus plánování klíčů

Protože je v každém kole využito 128 bitů klíče je důležité ukázat, jak jsou tyto bity vypočítány z hlavního klíče.

Na vstupu je hlavní klíč rozdělen na 8 stejně velikých 16bitových částí K = K1 || K2 || K3 || K4 || || K5 || K6 || K7 || K8, které tvoří první pole podklíčů (K_j). Druhé pole podklíčů (K_j') je vypočítáno z prvního aplikací funkce XOR na hodnoty K_j a hodnotu danou tabulkou 4.1,

$$\mathbf{K_j' = K_j \oplus C_j} \tag{3}$$

pro hodnoty j = 1 až 8.

C1	0x0123
C2	0x4567
C3	0x89AB
C4	0xCDEF
C5	0xFEDC
C6	0xBA98
C7	0x7654
C8	0x3210

Tabulka 4.1: Konstantní hodnoty pro generování základních podklíčů šifry Kasumi.

Pro jednotlivá kola jsou pak podklíče vypočítány na základě tabulky 4.2, kde operace <<< n značí rotaci bitů vlevo o n míst. Takto vzniklé hodnoty jsou již hodnoty použité jako vstup do jednotlivých funkcí.

	Číslo kola šifry							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$\mathbf{KL_{i,1}}$	K1 <<<1	K2 <<< 1	K3 <<< 1	K4 <<< 1	K5 <<< 1	K6 <<< 1	K7 <<< 1	K8 <<< 1
$\mathbf{KL_{i,2}}$	K3'	K4'	K5'	K6'	K7'	K8'	K1'	K2'
$\mathbf{KO_{i,1}}$	K2 <<<5	K3 <<<5	K4 <<<5	K5 <<<5	K6 <<<5	K7 <<<5	K8 <<<5	K1 <<<5
$\mathbf{KO_{i,2}}$	K6 <<<8	K7 <<<8	K8 <<<8	K1 <<<8	K2 <<<8	K3 <<<8	K4 <<<8	K5 <<<8
$\mathbf{KO_{i,3}}$	K7 <<<13	K8 <<<13	K1 <<<13	K2 <<<13	K3 <<<13	K4 <<<13	K5 <<<13	K6 <<<13
$\mathbf{KI_{i,1}}$	K5'	K5'	K7'	K8'	K1'	K2'	K3'	K4'
$\mathbf{KI_{i,2}}$	K4'	K5'	K6'	K7'	K8'	K1'	K2'	K3'
$\mathbf{KI_{i,3}}$	K8'	K1'	K2'	K3'	K4'	K5'	K6'	K7'

Tabulka 4.2: Výpočet podklíčů pro všechny funkce a kola šifry Kasumi.

4.1.2 Funkce FL (obrázek 4.1)

Vstupem funkce FL je 32 bitů dat (I) a 32 bitů podklíče (KL_i). Oba vstupní parametry jsou následně rozděleny na dvě 16bitové části

$$\mathbf{I = L \,||\, R} \tag{4}$$

$$\mathbf{KL_i = KL_{i,1} \,||\, KL_{i,2}}. \tag{5}$$

Následuje výpočet nových hodnot L' a R', které probíhá následovně

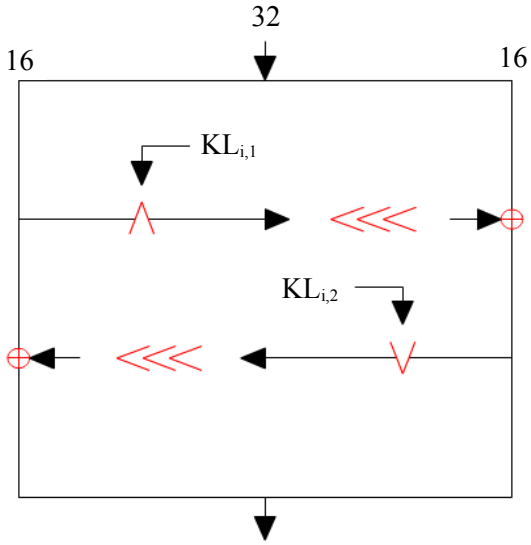
$$\mathbf{R' = R \oplus ((L \wedge KLi,1) <<< 1)} \tag{6}$$

$$\mathbf{L' = L \oplus ((R' \vee KLi,2) <<< 1)}, \tag{7}$$

kde $\wedge$ značí operaci AND a $\vee$ značí operaci OR mezi odpovídajícími si bity.

Po rozdělení vstupních hodnot a výpočtu výsledků nových částí L' a R' definujeme výstupní 32bitovou hodnotu jako konkatenaci dvou hodnot, tedy

$$\mathbf{O = L' \,||\, R'}. \tag{8}$$



Obrázek 4.1: Funkce FL.

4.1.3 Funkce FO (obrázek 4.2)

Vstupem funkce FO je 32bitů dat I a dva 48bitové podklíče KO_i a KI_i. Obdobně jako ve funkci FL jsou vstupní data i klíče rozděleny na 16bitové části

$$I = L_0 \,||\, R_0 \tag{9}$$

$$\mathbf{KO_i = KO_{i,1} \,||\, KO_{i,2} \,||\, KO_{i,3}} \tag{10}$$
$$\mathbf{KI_i = KI_{i,1} \,||\, KI_{i,2} \,||\, KI_{i,3}}. \tag{11}$$

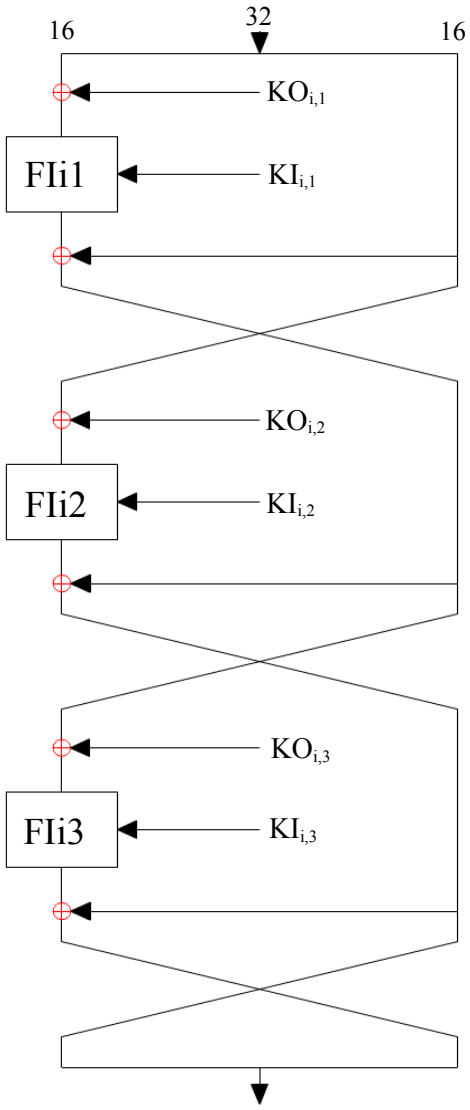
Vlastní výpočet funkce probíhá ve třech identických kolech (j = 1 až 3) s různými parametry

$$\mathbf{R_j = FI(L_{j-1} \oplus KO_{i,j}, KI_{i,j}) \oplus R_{j-1}} \tag{12}$$

$$\mathbf{L_j = R_{j-1}}. \tag{13}$$

Výstupem je pak 32bitová hodnota

$$\mathbf{O = L_3 \,||\, R_3}. \tag{14}$$



Obrázek 4.2: Funkce FO.

4.1.4 Funkce FI (obrázek 4.3)

Funkce FI má jako vstupní parametry 16 bitů dat (I) a 16 bitů podklíče ($KI_{i,j}$). Po načtení vstupu následuje opět rozdělení obou vstupních hodnot, tentokrát ovšem na části různých velikostí

$$\mathbf{I = L_0 \,||\, R_0}, \tag{15}$$

kde L_0 má velikost 9 bitů a R_0 velikost 7bitů.

Obdobně pak probíhá rozdělení podklíče

$$\mathbf{KI_{i,j} = KI_{i,j,1} \parallel KI_{i,j,2},} \tag{16}$$

kde $KI_{i,j,1}$ má velikost 7 bitů a $KI_{i,j,2}$ má velikost 9 bitů.

Tato funkce dále využívá dvou S-boxů – S7 a S9. Podrobněji jsou tyto S-boxy popsány v další části.

Před popisem vlastního výpočtu je třeba zmínit dvě pomocné funkce, a sice ZE(x) a TR(x). Funkce ZE(x) dostane na vstup 7bitovou hodnotu a rozšíří ji na 9bitovou přidáním dvou nul na místo nejvýznamnějších bitů. Funkce TR(x) přijímá jako parametr 9bitovou hodnotu. Poté odebere dva nejvýznamnější bity a vrátí zbývajících 7 bitů na výstup.

Následuje již proces vlastního výpočtu, kde postupně vypočteme následující hodnoty

$$\mathbf{L_1 = R_0} \tag{17}$$
$$\mathbf{R_1 = S9[L_0] \oplus ZE(R_0)} \tag{18}$$

$$\mathbf{L_2 = R_1 \oplus KI_{i,j,2}} \tag{19}$$
$$\mathbf{R_2 = S7[L_1] \oplus TR(R_1) \oplus KI_{i,j,1}} \tag{20}$$

$$\mathbf{L_3 = R_2} \tag{21}$$
$$\mathbf{R_3 = S9[L_2] \oplus ZE(R_2)} \tag{22}$$

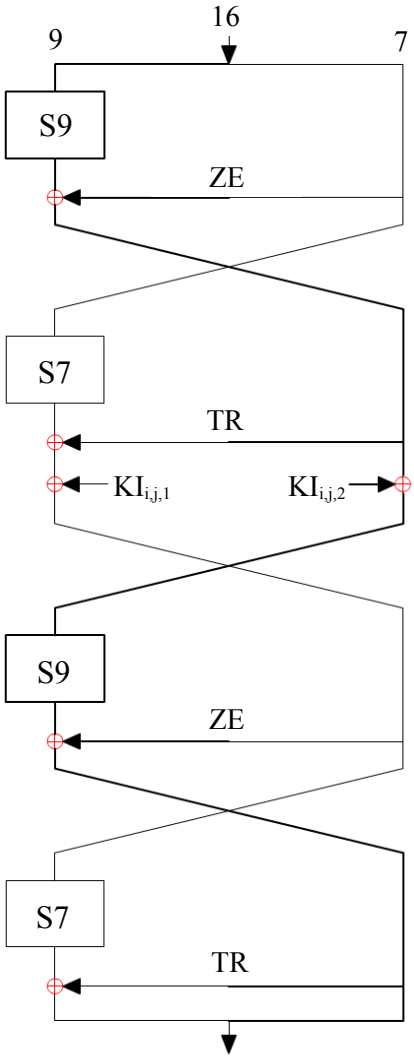
$$\mathbf{L_4 = S7[L_3] \oplus TR(R_3)} \tag{23}$$
$$\mathbf{R_4 = R_3.} \tag{24}$$

Výstupem funkce je hodnota

$$\mathbf{O = L_4 \parallel R_4.} \tag{25}$$

4.1.5 S-boxy

Jak již bylo zmíněno v předchozí části Kasumi využívá dvou S-boxů. Oba jsou navrženy pro snadnou implementaci v hardware – pomocí kombinační logiky nebo vyhledávací tabulky. První S-box označený S7 mapuje 7bitový vstup na 7bitový výstup a lze jej nalézt v příloze A.1. Druhý S-box nazývaný S9 mapuje 9bitový vstup na 9bitový výstup a je popsán v příloze A.2.



Obrázek 4.3: Funkce FI.

4.1.6 Popis šifrování

Před začátkem vlastního šifrování je 64bitový vstup rozdělen na dvě 32bitové části označené L_0 a R_0, kde L značí levou část vstupního bloku, R značí pravou část vstupního bloku a index (zde 0) značí kolo šifry. Následuje osm kol vlastního šifrování (obrázek 4.4), která jsou definována následovně

$$\mathbf{R_i = L_{i-1}, \quad L_i = R_{i-1} \oplus f_i(L_{i-1}, RK_i)} \tag{26}$$

Funkce f_i je rozdílná pro lichá a sudá kola. Pro lichá kola šifrování (1., 3., 5. a 7.) je funkce f_i definována jako

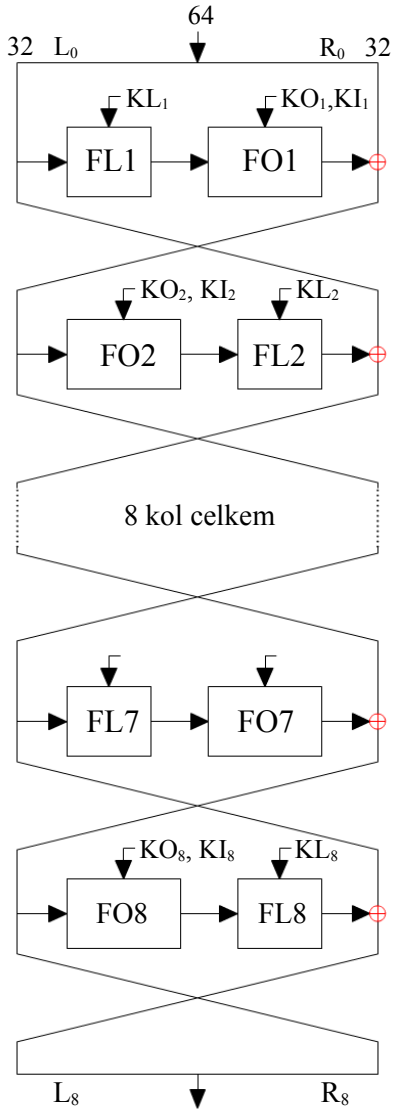
$$\mathbf{f_i(I, RK_i) = FO(FL(I, KL_i), KO_i, KI_i)} \tag{27}$$

a pro sudá kola šifrování (2., 4., 6. a 8.) je definice funkce f_i

$$\mathbf{f_i(I, RK_i) = FL(FO(I, KO_i, KI_i), KL_i)}. \tag{28}$$

Po provedení osmi kol šifrování jsou výsledné hodnoty L_8 a R_8 spojeny a poslány na výstup

$$\mathbf{O = L_8 \,||\, R_8}. \tag{29}$$



Obrázek 4.4: Struktura šifry KASUMI.

4.2 Kryptoanalýza

KASUMI vznikla úpravou šifry MISTY1 [36] za účelem snadnější implementaci v hardware a splnění podmínek pro bezpečnostní využití v 3G mobilní komunikaci [35]. Provedené úpravy však vedly ke snížení bezpečnosti původního algoritmu, čehož je důkazem úspěšný related-key útok [37]. Zmíněný útok funguje proti šifře KASUMI, ale selhává při použití proti šifře MISTY1. Autoři uvádějí, že tento útok nemusí fungovat v praxi proti implementaci A5/3 použité přímo v 3G systémech, ale že jejich cílem bylo ukázat snížení bezpečnosti šifry úpravami, které byly provedeny.

Tento útok je prozatím posledním významným krokem v oblasti bezpečnosti šifry KASUMI. Přecházela mu však řada dalších experimentů, z nichž prvním byla tzv. „nemožná" diferenciální analýza [38], která teoreticky dokáže získat všech 128 bitů hlavního klíče 6kolové KASUMI [39].

Dalším krokem v prolomení bezpečnosti šifry KASUMI je related-key boomerang útok [40]. V roce 2005 došlo k úspěšnému prolomení standardní 8kolové KASUMI rychleji než útokem hrubou silou [41]. Tento útok vyžaduje $2^{54,6}$ vybraných čistých textů, z nichž každý musí být zašifrován jedním ze čtyř příbuzných (tj. vzájemně souvisejících) klíčů. Jeho časová složitost je ekvivalentní šifrování $2^{76,1}$ bloků.

Ze zmíněných útoků lze vypozorovat, že šifra KASUMI není teoreticky tak bezpečná, jak se v době návrhu předpokládalo, a že modifikace způsobené během vývoje výrazně snížily její bezpečnost.

4.3 Shrnutí

Kapitola čtyři detailně popsala šifru KASUMI – všechny aspekty jejího designu (algoritmus plánování klíčů, funkce FL, FO, FI, S-boxy S7 a S9) a průběh šifrování jednoho bloku včetně odpovídajícího grafického znázornění. V druhé části byly zmíněny také důležité milníky v kryptoanalytické historii probrané šifry.

Kapitola 5

CUDAKasumi

Cílem praktické části jsou implementace šifry KASUMI na architektuře CUDA a srovnání jejich výkonu s implementacemi určenými pro CPU v oblasti kryptografie i kryptoanalýzy.

5.1 Použité technologie

Všechny programy jsou napsané v jazyce C/C++ s využitím CUDA C. Pro zkompilování bylo použito kompilátoru g++ pro soubory s příponou .c a kompilátoru nvcc pro soubory s příponou .cu (viz makefile soubory na DVD). Základem všech implementací KASUMI je demonstrativní kód uvedený v dokumentu [33]. GPU implementace co nejvěrněji kopíruje svoji ekvivalentní CPU implementaci kvůli přesnosti měření.

Pro měření byly použity dvě různé počítačové sestavy. Sestava 1 byla použita při vývoji i ladění programu a je tedy cílovou platformou pro všechny pokusy. Druhá stanice byla dostupná pouze pro první část měření a výsledky jsou určený pro srovnání výkonu jednotlivých grafických karet.

Hardware specifikace první platformy je Intel Core i7-950 (3 GHz), 5GB paměti RAM, GeForce GTX 285 (2GB paměti) běžící na distribuci Ubuntu 9.04.

Parametry druhé pracovní stanice jsou následující: Quad Core AMD Phenom II X4 940 (3GHz), 8 GB DDR2 RAM (1066 MHz), dvě NVIDIA GeForce GTX 480 (1.5GB paměti) běžící na distribuci Debian.

5.2 Šifrování dat fixní délky

Na základě zkušeností obsažených v kapitole 3 bylo jako výchozí experiment zvoleno šifrování datového souboru fixní délky pomocí jediného vlákna na CPU a následné porovnání s vícevláknovou implementací pro architekturu CUDA. Cílem toho experimentu je demonstrovat průběh zrychlení v závislosti na velikosti vstupních dat. Podpůrným výstupem je demonstrativní kód (viz přiložené DVD) zaměřený na názornou implementaci teoretických znalostí z kapitol 2 a 4.

Pro první sérii experimentů byly použity obě výše uvedené sestavy. Z důvodu demonstrativnosti kódu a snazší interpretace a srovnání výsledků byla použita pouze jedna grafická karta a jednovláknová CPU implementace. Pro uložení jednotlivých podklíčů do paměti grafického akcelerátoru byla zvolena paměť konstant. Vstupem algoritmu byly soubory velikosti 1KB, 10KB, 100KB, 1MB, 10MB a 100MB, které byly vygenerovány pseudonáhodně pomocí příkazu `dd if=/dev/urandom of=<jmenoSouboru> bs=<velikost KB/MB> count=<počet KB/MB>` a jsou přiloženy na DVD.

Algoritmus byl spuštěn pro obě implementace (CPU i GPU) a každou velikost 100x a aritmetické průměry dosažených výsledků pro obě zkoumané platformy jsou shrnuty v tabulkách 5.1 a 5.2.

Vstupní data	Čas CPU [ms]	Čas GPU [ms]	Zrychlení
1 KB	<1	1 035	–
10 KB	2	1 073	–
100 KB	18	1 080	0,01
1 MB	155	1 022	0,15
10 MB	983	1 040	0,95
100 MB	8 743	1 326	6,59

Tabulka 5.1: Shrnutí výsledků experimentu šifrování dat fixní délky na platformě 1.

Velikost dat	Čas CPU [ms]	Čas GPU [ms]	Zrychlení
1 KB	<1	90	0,01
10 KB	1	91	0,01
100 KB	13	97	0,13
1MB	133	107	1,24
10 MB	1 313	136	9,65
100 MB	13 204	477	27,68

Tabulka 5.2: Shrnutí výsledků experimentu šifrování dat fixní délky na platformě 2.

Z výsledků jsou patrné obdobné závěry jako u experimentů v kapitole 3, tedy že efektivní akcelerace pomocí GPU vyžaduje větší datové vstupy z důvodu vyšších nároků na paměťové operace. Pro dostatečně velký datový vstup však časová náročnost paměťových operací je výrazně překonána přínosem akcelerace. Pro menší datové vstupy však tvoří hranici, pod kterou se při výpočtu nelze dostat.

Dále je nutno zmínit, že výsledky dosažené na obou pracovních stanicích jsou velice ovlivněny použitou architekturou grafické karty. Zatímco GTX 285 je zařízení starší a s výpočetními schopnostmi verze 1.3, tak GTX 480 již využívá architekturu Fermi se všemi výhodami a její výpočetní schopnosti jsou verze 2.0. Je zřejmé, že vývoj GPU udělal touto novou generací opravdu velký krok kupředu.

Zajímavým vedlejším výsledkem byla skutečnost, že při spuštění GPU implementace v cyklu (kernel je obalen CPU cyklem), od druhé iterace dále docházelo k výraznému nárůstu (zhruba 30%) výkonu oproti první iteraci. Důvod této skutečnosti však není znám, proto veškeré výsledky shrnuté v tabulce 5.1 byly spuštěny samostatně a mimo cyklus.

5.3 Útok hrubou silou

Druhým provedeným experimentem je využití GPU k akceleraci útoku hrubou silou na šifru KASUMI. Tento experiment byl zvolen, protože převedení algoritmu útoku hrubou silou na GPU vyžaduje odlišný přístup než první provedený experiment. Hlavním cílem tohoto experimentu je otestovat zrychlení exekuce složitějších vláken.

Pro útok hrubou silou je potřeba opakovaně načítat části hlavního klíče a zbylé bity generovat pomocí unikátních identifikátorů jednotlivých vláken. Z tohoto důvodu je potřeba opakovaně používat kopírování dat z paměti stanice do paměti grafické karty. Dalším důsledkem předchozích faktů je využití více kernelů, protože maximální počet vláken na jeden kernel zdaleka nepokrývá potřeby pro

vygenerování celého 128bitového klíče. Poslední důležitou skutečností je, že kvůli generování podklíčů až po úspěšném vygenerování klíče na GPU je třeba podklíče vypočítávat v rámci vlákna a lokální paměti. Jiný přístup by přinesl značné zpomalení kvůli ceně paměťových operací a navíc by z hlediska množství generovaných dat a jejich rychlého přepisu neměl praktické opodstatnění.

Složitost každého vlákna tedy roste, protože musí obstarat vygenerování části vlastního hlavního klíče, následované vygenerováním všech podklíčů, zašifrováním daného čistého textu, porovnáním výsledku se známým správným zašifrovaným textem a vyhodnocením volby všech vláken, zda byl klíč nalezen. Tomu všemu předchází vyhodnocení podmínky, zda již byl klíč nalezen. V kladném případě urychluje běh spuštěného kernelu vypuštěním všech zbylých instrukcí jednotlivých vláken. Důvodem pro zmíněné řešení je skutečnost, že CUDA C neobsahuje žádný příkaz umožňující zevnitř vlákna ukončit právě běžící kernel.

CPU implementace útoku využívá vnořené cykly pro postupné generování celého klíče. V nejvnitřnějším cyklu pak běží vlastní výpočet podklíčů, šifrování zadaného bloku i porovnání výsledku. V případě nalezení klíče jsou všechny další cykly zrušeny pomocí příkazu break.

Implementace určená pro GPU využívá také generování větší části klíče pomocí vnořených cyklů mimo GPU a vygenerovaná část je poté nahrána do paměti grafického adaptéru a předána jako parametr kernelu. Před vlastním cyklem je také do paměti grafické karty nahrán čistý a jemu odpovídající šifrovaný text a proměnná udávající stav nalezení klíče.

Pro měření byla použita první platforma (viz kapitola 5.1). Z důvodu příliš mnoha lokálních proměnných pro jedno vlákno je nutné kód pro GPU kompilovat s možností `--maxrregcount=16`, která nastaví maximální počet registrů pro jedno vlákno na 16. Na jednom multiprocesoru totiž není dost registrů pro všechny lokální proměnné všech vláken v bloku zároveň. Každé vlákno potřebuje právě své hodnoty, na které nesmí ostatní vlákna přistupovat ani je sdílet – čistý text je překopírován a vlákno pracuje s lokální kopií, podklíče jsou vygenerovány jako lokální proměnné. Proto nelze využít sdílené paměti ani paměti konstant. Z časových důvodů byl jako testovací klíč zvolen klíč s hexadecimálním zápisem `00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 48`. Jako čistý text byla zvolena první testovací hodnota v dokumentu [42] a ji odpovídající zašifrovaná hodnota. Test byl proveden celkem desetkrát pro každou z implementací a výsledky jsou shrnuty v tabulce 5.3. Zdrojové kódy jsou opět přiloženy na DVD.

Čas CPU [s]	Čas GPU [s]	Zrychlení
3 737	105,92	35,25
3 728	105,98	35,18
3 728	105,96	35,18
3 726	105,97	35,16
3 730	105,92	35,22
3 729	105,96	35,19
3 724	105,97	35,14
3 725	105,95	35,16
3 724	105,97	35,14
3 728	105,98	35,18

Tabulka 5.3: Výsledky dosažené pro útok hrubou silou šifru KASUMI.

Z tabulky 5.3 lze vysledovat, že i pro tento problém dosahuje GPU implementace výrazně vyšší rychlosti (průměrně o faktor 35,18) a to i přes využití globální paměti pro některé proměnné. Při udržení tohoto faktoru i pro složitější klíče je tak možné pomocí GPU akcelerovat útok hrubou silou

proti šifře KASUMI o více než 2^5. Pro tento typ útoku je tak akcelerace grafickou kartou výbornou metodou zrychlení algoritmu. Vyšší zrychlení než u standardního šifrování je dáno menším využitím paměťových operací a zvýšením průtoku dat algoritmem. Tento experiment dokazuje, že SIMT paralelismus GPU dokáže zvládat i složitější operace v rámci jednoho vlákna při udržení velké míry akcelerace.

5.4 Generování Rainbow-chains

Posledním experimentem praktické části je akcelerace opakovaného šifrování dat za účelem zjištění pravděpodobnostního rozložení zašifrovaných hodnot v prostoru. Motivací pro tento experiment je evaluace návrhu šifry. Principem evaluace je pro pozorovanou podmnožinu předem daných bitů šifrovaného textu vypozorovat, zda všechny nabývají předem dané hodnoty (pro naše účely 0) a po kolika iteracích šifrování se tak stalo. Protože pravděpodobnost, že jeden pevně daný bit nabývá hodnoty 0 je 1/2, lze říci, že pravděpodobnost hodnoty 0 pro všech x předem daných bitů je 2^{-x}. Z toho lze usuzovat, že dobře navržená šifra bude takové hodnoty pro zmíněných x bitů nabývat průměrně každých 2^x šifrovacích iterací.

Cílem tohoto experimentu je zmíněná evaluace šifry KASUMI jak pomocí CPU, tak i GPU implementace. Nejprve jsou vygenerována náhodná data s 20 nulovými bity na nejméně významných místech, která jsou následně řetězově zpracovávána. Pro předdefinovaný počet bloků je zkoumáno, v kolika šifrovacích iteracích bude tento počet bloků dané charakteristiky nalezen. Tento počet je poté zprůměrován a čas potřebný pro výpočet zaznamenán. Výstupem obou implementací je průměrný počet iterací potřebných pro dosažení bloku obsahujícího nuly na místech dvaceti nejméně významných bitů a také časy potřebné k výpočtu.

CPU implementace šifruje opakovaně jeden blok, dokud v takto vzniklém řetězu nenalezne požadovaný počet bloků s výše zmíněnou charakteristikou. GPU implementace na základě požadovaného počtu bloků nejprve zjistí, zda je zadán počet dělitelný alespoň 2^5, aby byl v každém CUDA bloku alespoň celý warp vláken. Pokud je zadaný počet bloků dělitelný 2^6, 2^7, 2^8 nebo 2^9, tak je zvýšen počet vláken v CUDA bloku. Pro počet bloků vyšší než jeden CUDA blok (dané velikosti), tak dochází k přepočítání celkového počtu CUDA bloků. Dále pokud je CUDA bloků méně než 32, ale jsou velké (2^6 a větší), tak jsou algoritmicky zmenšeny na polovinu a je zvýšen jejich počet. Jedná se o jednoduché rozložení zátěže mezi více multiprocesorů, které by mělo alespoň částečně zvednou výkon. Poté je pro každé vlákno pseudonáhodně vygenerován jeden blok dané charakteristiky a všechny vzniklé bloky jsou nahrány do paměti GPU. Následuje spuštění kernelu, který zajišťuje spuštění vlákna nad každým generovaným blokem, dokud se šifrovacími iteracemi nedostaneme k dalšímu charakteristickému bloku. Počet iterací je zaznamenán do globální paměti. Všechny výsledky jsou poté přehrány do paměti stanice, kde je CPU cyklus sečte. Aritmetický průměr výsledků je poté vypsán na výstup. Zdrojový kód je přiložen na DVD.

Byla testována také implementace využívající sdílenou paměť a opakované šifrování nad daty (obdobně jako CPU implementace). Využívala všech dostupných multiprocesorů (30 pro GTX 285) zároveň, což omezovalo počet testovaných bloků na násobky 960 (celkový počet proudových procesorů GTX 285), a cyklus byl uvnitř vláken. Díky rozdílnosti v jednotlivých výpočtech se však tato implementace ukázala jako slepá cesta pro přílišnou divergenci vláken. S maximální akcelerací o faktor 5,34 pro 3840 bloků tak je zmíněna pouze pro úplnost.

Počet bloků	Čas CPU [s]	Čas GPU [s]	Zrychlení	Průměrný počet CPU iterací na blok	Průměrný počet GPU iterací na blok
256	169,41	146,48	1,15	1 013 567	1 053 995
512	343,99	132,38	2,60	1 029 151	1 073 647
1 024	696,69	129,95	5,36	1 042 103	1 017 121
0	1 354,39	138,17	9,80	1 037 647	1 043 039
2 048	1 405,16	155,90	9,01	1 051 294	1 026 048
0	2 007,41	144,82	13,86	1 028 271	1 042 507
4 000	2 761,46	164,64	16,77	1 057 500	1 039 749
4 096	2 826,36	158,09	17,88	1 062 331	1 019 466
0	3 412,93	195,54	17,45	1 045 698	1 066 931
8 192	5 561,73	284,87	19,52	1 039 769	1 052 382
12 800	8 660,75	327,97	26,41	1 036 852	1 061 899
100 000	68 525,70	2 481,35	27,62	1 049 536	1 046 002

Tabulka 5.4: Výsledky dosažené pro generování rainbow chains šifry KASUMI.

Tento experiment přinesl očekávané výsledky. Dokazují, že šifra KASUMI byla z hlediska pravděpodobnostního rozložení jednotlivých hodnot šifrovaných textů navržena dobře (tabulka 5.4) a odchylka všech měření od očekávané hodnoty 2^{20} nebyla vyšší než 3,33 % pro počet CPU iterací potřebných pro 256 bloků. Naměřenou odchylku tak lze přisuzovat nedostatečnému počtu zkoumaných bloků. Pro více bloků dochází k postupné konvergenci průměrného počtu iterací právě k hodnotě 2^{20}. Z toho lze usuzovat, že v tomto aspektu šifra KASUMI splňuje požadované návrhové kritérium.

Druhým pozorovaným výsledkem je opět podstatně vyšší rychlost GPU implementace (tabulka 5.4). Protože pro dostatečně velký počet testovaných bloků je faktor zrychlení 27,62, lze předpokládat, že se jedná o přibližné maximální zrychlení použité implementace.

5.5 Shrnutí

Pátá kapitola ukazuje, že i v kryptologii má paralelismus své opodstatnění. Bylo ukázáno celkem šest rozdílných implementací pro tři různé kryptologické problémy, přičemž u každého problému byl testován jiný přístup k využití paralelizace. U všech problémů vedlo využití GPU k urychlení použitého algoritmu. Je tedy zřejmé, že lze programovatelných grafických akcelerátorů v kryptologii s úspěchem využít.

Závěr

Programovatelné grafické akcelerátory zažívají v poslední době obrovský rozmach a již dávno nejsou určený pouze pro pomocné grafické výpočty. Moderní grafické karty dokazují, že jejich využití pro paralelizaci úloh je levným zdrojem obrovské výpočetní síly. Urychlují tak nástup paralelismu jako náhrady zvyšování výkonu jednotlivých jader výpočetních zařízení, které se pomalu dostávají na své fyzikální hranice.

S tímto rozvojem je třeba změnit i přístupy a styly myšlení, které mnohdy zůstávají ještě v dobách jednovláknových aplikací. Paralelismus se stává dominantním přístupem v programování a je tedy nasnadě nezůstat pozadu. Programovatelné grafické karty nabízejí nové možnosti v širokém spektru oblastí a jako každá technologie mají kromě nespočtu pozitiv i svá omezení a zápory. Pokud však na tyto negativní aspekty při návrhu algoritmů budeme dbát, dají se částečně či úplně eliminovat nebo obejít. Pro některé typy úloh tak již nyní výpočty na grafických kartách překonávají ekvivalentní na klasických procesorech. Mezi tyto oblasti se zařazuje i kryptologie.

V této práci bylo demonstrováno, jak využít programovatelné grafické procesory k akceleraci tří odlišných problémů oblasti kryptologie – šifrování dat, útok hrubou silou na hlavní klíč šifry a generování Rainbow chains. Jako experimentální šifra byla zvolena KASUMI, která je standardem v oblasti šifrování GSM komunikace. Jedním z důvodů této volby byla absence obdobného výzkumu pro tuto šifru, přestože pro jiné šifrovací standary bylo již vydáno větší množství publikací.

Práce přináší ucelený vstup do problematiky programování aplikací určených pro grafické procesory, tři aplikace s podrobně dokumentovaným kódem a sérii výsledků měření výkonu použitých implementací. Prvním cílem práce bylo demonstrovat snadnost využití GPU pro obecné výpočty, čehož bylo dosaženo pomocí textu a názorných ukázek v kapitolách 2 a 3 a zdrojových kódů zmíněných aplikací. Druhým cílem práce bylo měření výkonu GPU implementací na šifře KASUMI pro účely srovnání s GPU implementacemi jiných šifer. Všechny experimenty dokázaly, že využití GPU pro obecné výpočty přináší velkou výpočetní sílu a ani šifra KASUMI není výjimkou. Výkon paralelní verze KASUMI určené pro grafické karty je srovnatelný s výkony jiných blokových šifrer (AES, DES). Třetím cílem byla snaha využít možnosti grafické karty pro akceleraci útoku hrubou silou na hlavní klíč šifry KASUMI. V experimentálním prostředí bylo dosaženi průměrného zrychlení o faktor 35,12, což značně předčilo očekávání. Posledním cílem byla evaluace návrhu šifry KASUMi z hlediska pravděpodobnostního rozložení hodnot šifrovaných bloků. S využitím CPU i GPU bylo ukázáno, že šifra KASUMI je v tomto ohledu navržena optimálně a že i pro tento problém je GPU implementace výrazně rychlejší než její CPU ekvivalent.

Další možností výzkumu je akcelerace bit-slice implementací jednotlivých šifer. Granularita tohoto problému by mohla totiž lépe odpovídat použité architektuře a jejich vzájemná synergie by tak mohla být ještě vyšší než u paralelizace standardních algoritmů.

Reference

[1] Mooreův zákon [online] 29. 9. 2010, [cit. 19. 2. 2011]. Dostupné na: http://cs.wikipedia.org/wiki/Moore%C5%AFv_z%C3%A1kon.

[2] CUDA [online] 19. 2. 2011, [cit. 19. 2. 2011]. Dostupné na: http://en.wikipedia.org/wiki/CUDA.

[3] GPU [online] 18. 2. 2011, [cit. 20. 2. 2011]. Dostupné na: http://en.wikipedia.org/wiki/Graphics_processing_unit.

[4] GPGPU [online] 18. 2. 2011, [cit. 20. 2. 2011]. Dostupné na: http://en.wikipedia.org/wiki/GPGPU.

[5] Out-of-order exectution [online] 10. 1. 2011, [cit. 20. 2. 2011]. Dostupné na: http://en.wikipedia.org/wiki/Out-of-order_execution.

[6] Flynn's taxonomy [online] 25. 1. 2011, [cit. 20. 2. 2011]. Dostupné na: http://en.wikipedia.org/wiki/Flynn%27s_taxonomy.

[7] Úvod, základy CUDA [online] podzim 2010, [cit. 20. 2. 2011]. Dostupné na: https://is.muni.cz/auth/el/1433/podzim2010/PV197/um/17858100/1.pdf?fakulta=1433;obdobi=5104;kod=PV197.

[8] NVIDIA's Next Generation CUDA Compute Architecture: Fermi [online] 2009, [cit. 24. 2. 2008]. Dostupné na: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[9] NVIDIA CUDA C Programming Guide 3.2 [online] 11. 9. 2010, [cit. 23. 2. 2011]. Dostupné z World Wide Web: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[10] Threads and blocks and grids, oh my! [online] 11. 6. 2008, [cit. 24. 2. 2011] Dostupné z World Wide Web: http://llpanorama.wordpress.com/2008/06/11/threads-and-blocks-and-grids-oh-my/.

[11] CUDA, Supercomputing for the Masses: Part 11 [online] 18. 3. 2009, [cit. 24. 2. 2011]. Dostupné na: http://www.drdobbs.com/high-performance-computing/215900921.

[12] CUDA, Supercomputing for the Masses: Part 5 [online] 30. 6. 2008, [cit. 24. 2. 2011]. Dostupné na: http://www.drdobbs.com/high-performance-computing/208801731.

[13] CUDA Programming Week 4. Shared memory and register [online] 2010, [cit. 25. 2. 2011]. Dostupné na: http://www.cs.nthu.edu.tw/~cherung/teaching/2010gpucell/CUDA04.pdf.

[14] Výkon GPU hardware [online] podzim 2010, [cit. 24. 2. 2011]. Dostupné na: https://is.muni.cz/auth/el/1433/podzim2010/PV197/um/17858100/3.pdf?fakulta=1433;obdobi=5104;kod=PV197.

[15] GPU hardware a paralelismus [online] podzim 2010, [cit. 15. 3. 2011]. Dostupné na: https://is.muni.cz/auth/el/1433/podzim2010/PV197/um/17858100/2.pdf?fakulta=1433;obdobi=5104;studium=484373;kod=PV197.

[16] Memory Management [online], [cit. 15. 3. 2011]. Dostupné na: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/group__CUDART__MEMORY.html.

[17] Parallel AES algorithm for fast Data Encryption on GPU [online] 16. – 18. duben 2010 [cit. 22. 3. 2011]. Dostupné na: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5486259.

[18] AES Proposal: Rijndael [online] březen 1999, [cit. 22. 3. 2011]. Dostupné na: http://www.daimi.au.dk/~ivan/rijndael.pdf.

[19] CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography [online] podzim 2010, [cit. 22. 3. 2011]. Dostupné na: http://www.manavski.com/downloads/PID505889.pdf.

[20] AES Encryption Implementation on CUDA GPU and Its Analysis [online] listopad 2010, [cit. 22. 3. 2011]. Dostupné na: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5695236.

[21] Handbook of Applied Cryptography [online] srpen 2001, [cit. 12. 4. 2011]. Dostupné na: http://www.cacr.math.uwaterloo.ca/hac/.

[22] Dynamic random-access memory [online] březen 1999, [cit. 9. 3. 2011]. Dostupné na: http://en.wikipedia.org/wiki/Dynamic_random-access_memory.

[23] A Fast New DES Implementation in Software [online] 1997, [cit. 13. 4. 2011]. Dostupné na: http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/../../CS0891.ps.gz.

[24] AES and DES Encryption with GPU [online] 2009, [cit. 14. 4. 2011]. Dostupné na: http://bioinformatics.louisville.edu/ouyang/Luken-Ouyang-Desoky-PDCCS2009.pdf.

[25] Record Setting Software Implementation of DES Using CUDA [online] duben 2010, [cit. 14. 4. 2011]. Dostupné na: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5695236.

[26] DES and Triple-DES encryption/decryption Algorithm [online] 1998, [cit. 14. 4. 2011]. Dostupné na: http://amadousarr.free.fr/crypto/des1.pdf.

[27] SSE2 [online] 7. 11. 2011, [cit. 16. 4. 2011]. Dostupné na: http://en.wikipedia.org/wiki/SSE2.

[28] GPU Computing [online] duben 2010, [cit. 19. 4. 2011]. Dostupné na: http://geo.mff.cuni.cz/jednooci_slepym/lh-GPU1.pdf.

[29] A5/1 Security Project [online] 2009, [cit. 20. 4. 2011]. Dostupné na: http://reflextor.com/trac/a51/wiki.

[30] Optimization history [online] 2010, [cit. 20. 4. 2011]. Dostupné na: http://reflextor.com/trac/a51/wiki/Performance.

[31] Backclocking A5/1 [online] podzim 2010, [cit. 20. 4. 2011]. Dostupné na: http://reflextor.com/trac/a51/wiki/BackclockA51.

[32] GSM – SRSLY? [online] prosinec 2009, [cit. 20. 4. 2011]. Dostupné na: http://events.ccc.de/congress/2009/Fahrplan/attachments/1519_26C3.Karsten.Nohl.GSM.pdf.

[33] 3GPP TS 35.202 Document 2: KASUMI specification (Release 8) [online] prosinec 2008, [cit. 21. 4. 2011]. Dostupné na: http://www.3gpp.org/ftp/specs/archive/35_series/35.202/35202-800.zip.

[34] Feistel cipher [online] 9. 3. 2011, [cit. 21. 4. 2011]. Dostupné na: http://en.wikipedia.org/wiki/Feistel_cipher.

[35] KASUMI (block cipher) [online] 7. 2. 2011, [cit. 23. 4. 2011]. Dostupné na: http://en.wikipedia.org/wiki/KASUMI_%28block_cipher%29.

[36] New Block Encryption Algorithm MISTY [online] červenec 1996, [cit. 23. 4. 2011]. Dostupné na: http://classic-web.archive.org/web/20000823133547/http://www.mitsubishi.com/ghp_japan/misty/misty_e_b.pdf.

[37] A Practical-Time Attack on the A5/3 Cryptosystem Used in Third Generation GSM Telephony [online] 10. 2. 2010, [cit. 23. 4. 2011]. Dostupné na: http://eprint.iacr.org/2010/013.pdf.

[38] Impossible differential cryptanalysis [online] 17. 4. 2011, [cit. 23. 4. 2011]. Dostupné na: http://en.wikipedia.org/wiki/Impossible_differential_cryptanalysis.

[39] Cryptanalysis of Reduced-Round MISTY [online] 2001, [cit. 23. 4. 2011]. Dostupné na: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=55B36FD46D3B0CC1A7E3D8BBC75017DA?doi=10.1.1.59.7609&rep=rep1&type=pdf.

[40] Boomerang attack [online] 24. 7. 2010, [cit. 23. 4. 2011]. Dostupné na: http://en.wikipedia.org/wiki/Boomerang_attack.

[41] A Related-Key Rectangle Attack on the Full KASUMI [online] podzim 2010, [cit. 22. 3. 2011]. Dostupné na: http://www.ma.huji.ac.il/~nkeller/kasumi.ps.

[42] 3GPP TS 35.203 V9.0.0 Document 3: Implementors' Test Data [online] prosinec 2009, [cit. 20. 5. 2011]. Dostupné na: http://cryptome.org/3gpp/35203-900.pdf.

Příloha A – Doplňující materiál šifry KASUMI

A.1 S7

S7[128] = {
54, 50, 62, 56, 22, 34, 94, 96, 38, 6, 63, 93, 2, 18, 123, 33,
55, 113, 39, 114, 21, 67, 65, 12, 47, 73, 46, 27, 25, 111, 124, 81,
53, 9, 121, 79, 52, 60, 58, 48, 101, 127, 40, 120, 104, 70, 71, 43,
20, 122, 72, 61, 23, 109, 13, 100, 77, 1, 16, 7, 82, 10, 105, 98,
117, 116, 76, 11, 89, 106, 0, 125, 118, 99, 86, 69, 30, 57, 126, 87,
112, 51, 17, 5, 95, 14, 90, 84, 91, 8, 35, 103, 32, 97, 28, 66,
102, 31, 26, 45, 75, 4, 85, 92, 37, 74, 80, 49, 68, 29, 115, 44,
64, 107, 108, 24, 110, 83, 36, 78, 42, 19, 15, 41, 88, 119, 59, 3 }

A.2 S9

S9[512] = {
167, 239, 161, 379, 391, 334, 9, 338, 38, 226, 48, 358, 452, 385, 90, 397,
183, 253, 147, 331, 415, 340, 51, 362, 306, 500, 262, 82, 216, 159, 356, 177,
175, 241, 489, 37, 206, 17, 0, 333, 44, 254, 378, 58, 143, 220, 81, 400,
95, 3, 315, 245, 54, 235, 218, 405, 472, 264, 172, 494, 371, 290, 399, 76,
165, 197, 395, 121, 257, 480, 423, 212, 240, 28, 462, 176, 406, 507, 288, 223,
501, 407, 249, 265, 89, 186, 221, 428, 164, 74, 440, 196, 458, 421, 350, 163,
232, 158, 134, 354, 13, 250, 491, 142, 191, 69, 193, 425, 152, 227, 366, 135,
344, 300, 276, 242, 437, 320, 113, 278, 11, 243, 87, 317, 36, 93, 496, 27,
487, 446, 482, 41, 68, 156, 457, 131, 326, 403, 339, 20, 39, 115, 442, 124,
475, 384, 508, 53, 112, 170, 479, 151, 126, 169, 73, 268, 279, 321, 168, 364,
363, 292, 46, 499, 393, 327, 324, 24, 456, 267, 157, 460, 488, 426, 309, 229,
439, 506, 208, 271, 349, 401, 434, 236, 16, 209, 359, 52, 56, 120, 199, 277,
465, 416, 252, 287, 246, 6, 83, 305, 420, 345, 153, 502, 65, 61, 244, 282,
173, 222, 418, 67, 386, 368, 261, 101, 476, 291, 195, 430, 49, 79, 166, 330,
280, 383, 373, 128, 382, 408, 155, 495, 367, 388, 274, 107, 459, 417, 62, 454,
132, 225, 203, 316, 234, 14, 301, 91, 503, 286, 424, 211, 347, 307, 140, 374,
35, 103, 125, 427, 19, 214, 453, 146, 498, 314, 444, 230, 256, 329, 198, 285,
50, 116, 78, 410, 10, 205, 510, 171, 231, 45, 139, 467, 29, 86, 505, 32,
72, 26, 342, 150, 313, 490, 431, 238, 411, 325, 149, 473, 40, 119, 174, 355,
185, 233, 389, 71, 448, 273, 372, 55, 110, 178, 322, 12, 469, 392, 369, 190,
1, 109, 375, 137, 181, 88, 75, 308, 260, 484, 98, 272, 370, 275, 412, 111,
336, 318, 4, 504, 492, 259, 304, 77, 337, 435, 21, 357, 303, 332, 483, 18,
47, 85, 25, 497, 474, 289, 100, 269, 296, 478, 270, 106, 31, 104, 433, 84,
414, 486, 394, 96, 99, 154, 511, 148, 413, 361, 409, 255, 162, 215, 302, 201,
266, 351, 343, 144, 441, 365, 108, 298, 251, 34, 182, 509, 138, 210, 335, 133,
311, 352, 328, 141, 396, 346, 123, 319, 450, 281, 429, 228, 443, 481, 92, 404,
485, 422, 248, 297, 23, 213, 130, 466, 22, 217, 283, 70, 294, 360, 419, 127,
312, 377, 7, 468, 194, 2, 117, 295, 463, 258, 224, 447, 247, 187, 80, 398,
284, 353, 105, 390, 299, 471, 470, 184, 57, 200, 348, 63, 204, 188, 33, 451,
97, 30, 310, 219, 94, 160, 129, 493, 64, 179, 263, 102, 189, 207, 114, 402,
438, 477, 387, 122, 192, 42, 381, 5, 145, 118, 180, 449, 293, 323, 136, 380,
43, 66, 60, 455, 341, 445, 202, 432, 8, 237, 15, 376, 436, 464, 59, 461 }