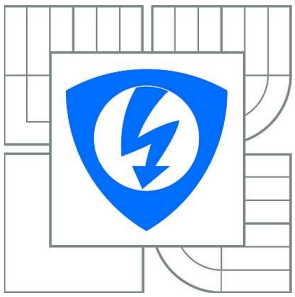
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF TELECOMMUNICATIONS

APLIKACE VYUŽÍVAJÍCÍ PARALELNÍ ZPRACOVÁNÍ PRO KRYPTOGRAFICKÉ VÝPOČTY

APPLICATIONS FOR PARALLEL PROCESSING IN CRYPTOGRAPHY

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

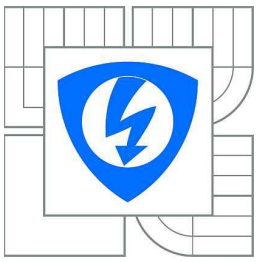
AUTOR PRÁCE — Bc. JAROMÍR ŠÁNEK

AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN HAJNÝ, Ph.D.

SUPERVISOR

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský navazující studijní obor
Telekomunikační a informační technika

Student: Bc. Jaromír Šánek ***ID:*** 129364
Ročník: 2 ***Akademický rok:*** 2013/2014

NÁZEV TÉMATU:

Aplikace využívající paralelní zpracování pro kryptografické výpočty

POKYNY PRO VYPRACOVÁNÍ:

Implementujte jednoduchý program pro výpočty revokační funkce na PC. Změřte výkon implementace pro různé délky modula. Implementujte stejné operace paralelně na platformě CUDA. Změřte výkon implementace pro různé délky modula. Porovnejte výkon na platformě PC bez a s využitím akcelerace CUDA i pro různé grafické karty. Vyberte vhodnou platformu pro implementaci, výběr zdůvodněte. Vyberte a popište minimálně 3 algoritmy pro modulární mocnění, implementujte je na vybrané platformě. Změřte čas nutný pro mocnění s využitím 1024 b a 2048 b modula. Implementujte client/server aplikaci pro centralizovaný výpočet revokační funkce protokolu HM12. Zaměřte se na optimalizaci výpočtu revokačního testu pomocí běhu ve více vláknech a na více procesorech.

DOPORUČENÁ LITERATURA:

[1] STALLINGS, William. Cryptography and network security: principles and practice. Seventh edition. xix, 731 pages. ISBN 01-333-5469-5.
[2] HAJNÝ, J.; MALINA, L. Unlinkable Attribute-Based Credentials with Practical Revocation on Smart-Cards. In Smart Card Research and Advanced Applications. Lecture Notes in Computer Science. LNCS. Berlin: Springer- Verlag, 2013. s. 62-76. ISBN: 978-3-642-37287- 2. ISSN: 0302- 9743.
[3] SANDERS, Jason. CUDA by example: an introduction to general-purpose GPU programming. 1st print. Upper Saddle River: Addison-Wesley, 2010, xix, 290 s. ISBN 978-0-13-138768-3.

Termín zadání: 10.2.2014 ***Termín odevzdání:*** 28.5.2014

Vedoucí práce: Ing. Jan Hajný, Ph.D.
Konzultanti diplomové práce:

doc. Ing. Jiří Mišurec, CSc.
Předseda oborové rady

Abstrakt

Tato práce se zabývá paralelním programováním a modulárním mocněním. V první části je srovnána rychlost funkcí modulárního mocnění z různých knihoven C/C++ na CPU. V druhé části se práce zabývá technologií CUDA, je zde změřena rychlost funkce modulárního mocnění z upravené knihovny LibTomMath pro technologii CUDA na GPU a porovnána s rychlostí stejné funkce běžící na CPU. Poslední část je věnována implementaci aplikací „Klient – Server“ pro výpočet revokační funkce protokolu HM12.

Klíčová slova: CUDA, GMP, CUMP, LIBTOM, OpenMP, HM12

Abstract

This thesis is about parallel programming. In the first part of the thesis is compared speed of functions modular exponentiation from various C/C++ libraries for CPU. In the second part is transformed the LibTomMath library from CPU to GPU CUDA technology. For devices CPU and GPU is compared speed of processing the operation of modular exponentiation from modified library. In conclusion are created two applications “Client – Server” for computing the revocation function of the protocol HM12.

Keywords: CUDA, GMP, CUMP, LIBTOM, OpenMP, HM12

ŠÁNEK, J. *Aplikace využívající paralelní zpracování pro kryptografické výpočty*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2014. 73 s. Vedoucí diplomové práce Ing. Jan Hajný, Ph.D..

Prohlášení

Prohlašuji, že svou diplomovou práci na téma „**Aplikace využívající paralelní zpracování pro kryptografické výpočty**“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

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

V Brně dne ....

podpis autora

Poděkování

Děkuji vedoucímu práce Ing. Janu Hajnému, Ph.D., za velmi užitečnou a metodickou pomoc a cenné rady při zpracování diplomové práce.

V Brně dne …….. ……

podpis autora

Obsah

1 Úvod

Tato práce se zabývá paralelním programováním na platformách CPU a GPU a možnostmi efektivního výpočtu modulárního mocnění na těchto platformách.

Grafické karty se zpočátku používaly k zpracování 2D obrazu, v polovině 90. letech 20. století se do grafické karty přidal akcelerátor pro 3D grafiku a na konci 90. let se poprvé objevuje označení GPU (Graphics Processor Unit). GPU uměla zpracovávat T&L operace a tak mohla odlehčit procesoru. T&L operace se zabývají výpočty spojených s transformacemi a osvětlením. V roce 2003 výkon GPU překonal výkon (GFlops) klasického CPU a tento nárůst výkonu neustále pokračuje hlavně díky počítačovým hrám.

V dnešní době se GPU používá pro celou škálu výkonnostně náročných aplikací. Pro výpočty na GPU vznikla celá řada technologií, například Compute Unified Device Architecture od společnosti Nvidia, ATI Stream od společnosti ATI nebo standard Open Computing Language pod záštitou Khronos [18].

CUDA je z těchto technologií nejrozšířenější, má kvalitní API a můžeme ji využívat na všech grafických akcelerátorech od série G80. CUDA používají k výpočtům například tyto aplikace: Autodesk 3ds Max, Adobe Photoshop CS6, Adobe Premiere CS6.

2 Cíle práce

Prvním cílem této práce je zanalyzování paralelních výpočtů modulárního mocnění na platformách CPU a GPU. Na platformě GPU se zaměřit na technologii CUDA, podrobně popsat tuto technologii, prozkoumat existující knihovny a pokusit se naprogramovat výpočet modulárního mocnění pro velká celá čísla na této platformě. Na platformě CPU popsat existující knihovny pro práci s velkými celými čísly typu BigInteger a změřit výkon těchto knihoven na funkci modulárního mocnění.

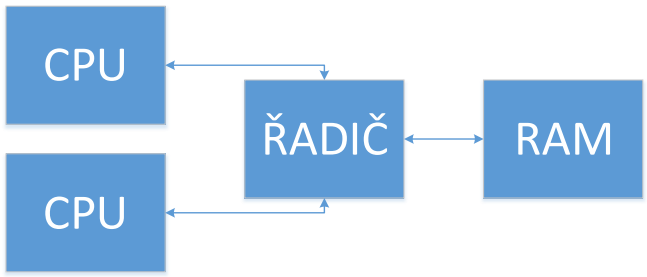
V druhé části této práce má být vytvořená aplikace „Klient – Server“, která bude zpracovávat revokační funkci protokolu HM12 [27]. Vytvořená aplikace má pracovat na vhodné platformě vybrané podle načerpaných znalostí z první části.

3 Paralelní aritmetické výpočty na CPU

Pro zpracování velkého množství dat na CPU je výhodné použít paralelismu a tím několikanásobně zkrátit celkovou dobu zpracování dat. Pro využití paralelního běhu vláken existuje několik knihoven, z nichž nejvýznamnější jsou standardy OpenMP a MPI.

3.1 OpenMP

OpenMP je nadstavbou jazyků Fortran/C/C++ a je to standard pro paralelní programování na CPU se sdílenou pamětí. U tohoto typu programování mají všechny procesory neomezený přístup do společné paměti. Ke společné paměti procesory obvykle přistupují přes společnou sběrnici. Pokud k této sběrnici připojíme hodně procesorů nebo jader, může dojít k zahlcení sběrnice [9].



Obrázek 1: Počítač se sdílenou pamětí

OpenMP je složen ze tří částí:

- OpenMP direktivy – určují jak má překladač paralelizovat části programu.
- OpenMP funkce – tyto funkce pomáhají zjistit počet současně běžících vláken nebo číslo aktuálního vlákna.
- Systémové proměnné – pomocí systémových proměnných lze nastavit parametry pro běh programu.

Příklad direktivy OpenMP:

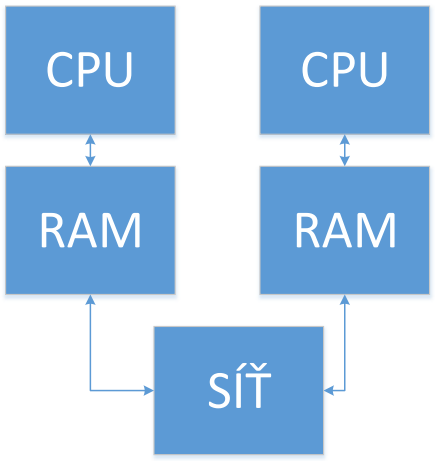
```
#pragma omp parallel {
… // každé vlákno vykoná příkazy tohoto bloku
}
```

Největší výhodou OpenMP je jednoduchost. Mezi další výhody tohoto standardu patří přenositelnost nebo možnost paralelizovat pouze kritické části kódu.

Naopak mezi nevýhody lze zařadit nutnost mít překladač podporující OpenMP, omezenost pouze na počítače se sdílenou pamětí a nižší škálovatelnost oproti MPI.

3.2 Message Passing Interface

Pro počítače s distribuovanou pamětí existuje standard MPI (Message Passing Interface). Sdílení dat u této knihovny je realizováno pomocí „posílání zpráv s daty“ mezi procesory. Oproti OpenMP je MPI univerzálnější (není závislé na architektuře), většina největších superpočítačů na světě je naprogramována pomocí této knihovny.



Obrázek 2: Počítač s distribuovanou pamětí

Při využití MPI procesy mezi sebou komunikují pomocí speciálních funkcí, každý proces má své vlastní proměnné. Procesory jsou od sebe odděleny pořadovým číslem a na každém procesoru běží stejný program samostatně [10].

Komunikace mezi procesy probíhá pomocí systému zpráv. Jednotlivé zprávy jsou „pakety dat“ vyměňovány mezi jednotlivými procesy. Systém zpráv musí splňovat několik základních pravidel. Proces musí zprávu registrovat do MPI. Zpráva musí mít identifikaci, kam má být doručena a požadovaný příjemce musí být schopen zprávy přijímat. Každý takový paket dat musí obsahovat několik základních informací: identifikaci posílajícího procesu, umístění dat v odesílajícím procesu, identifikaci datového typu, počet prvků (velikost pole), identifikaci přijímajícího procesu, umístění dat v přijímajícím procesu [11].

U MPI se vyskytují dva typy komunikace: point-to-point komunikace a globální komunikace.

3.3 Task Parallel Library

Tato knihovna se nachází v namespace System.Threading od .NET Framework 4.0. Knihovna obsahuje třídu Parallel. Tato třída je statická umožňuje jednoduše paralelizovat určitou část kódu. Třída obsahuje metody paralelní „For“ a paralelní „Foreach“. Obě metody fungují tak, že po jejich inicializaci rozdělí iterační proměnou do několika skupin, pak si z thread poolu určí počet vláken a mezi tyto vlákna přiřadí jednotlivé skupiny a vlákna začnou paralelně zpracovávat úlohu. Přerušit výpočet lze pomocí třídy ParallelLoopState, která umožňuje provést ve smyčce break [26].

Thread pool je fond vláken a jeho funkce je podrobně popsána v [25]. Defaultně je počet vláken nastaven na *počet jader procesoru* * 2, ale tento počet se dá změnit pomocí instance ParallelOptions.

4 Aritmetické výpočty na grafické kartě

Grafické karty v současné době mají větší výkon, než má CPU a tudíž v sobě ukrývají větší potenciál pro paralelní zefektivnění zpracování dat. Nejvíce rozšířené techniky pro využití GPU k výpočtům jsou CUDA a OpenCL.

4.1 CUDA

CUDA (Compute Unified Device Architecture) je technologie od společnosti NVIDIA. Jedná se o hardwarovou a softwarovou architekturu umožňující na GPU spouštět programy napsané v jazycích C, C++, C#, Java, Fortran, Python. Technologie byla představena v roce 2006 a v roce 2007 bylo vydáno SDK ve verzi 1.0.

CUDA zjednodušuje programování v GPGPU. GPGPU je zkratka pro provádění obecných výpočtů prostřednictvím grafických procesorů. CUDA odstraňuje nutnost pracovat s OpenGL a formulování úloh pomocí textur. Nevýhodou je, že tato technologie je určena pouze pro grafické karty od společnosti NVIDIA.

Výhodou pro vývojáře je velké spektrum nástrojů pro vývoj na CUDA a také různé vědecké knihovny CUFFT a CUBLAST.

4.1.1 Architektura CUDA

Program pro CUDA je obvykle rozdělen na dvě části, na tzv. „host kód“ a „device kód“. Host kód je spouštěn na hostitelském zařízení, kterým je obvykle běžné CPU. Naproti tomu „device kód“ se vykonává paralelně na zařízení CUDA. Kompletní CUDA program je tvořen z obou částí. NVCC (Nvidia CUDA Compiler) tyto části od sebe oddělí. Host kód bude zkompilován pomocí standartního C kompilátoru

a bude spouštěn na CPU. Device kód bude kompilován pomocí kompilátoru NVCC a bude namapován na GPU zařízení.

4.1.2 Kernel

Kernel je funkce, která se spouští z hosta a provádí ji paralelně mnoho vláken v CUDA zařízení. Ve skutečnosti CUDA vykonává funkce v modelu Single Multiple Data (SPMD). To znamená, že uživatel nakonfiguruje pro svůj kernel počet paralelně spouštěných vláken – stejný program použitý pro různá data bude vykonáván se stejným počtem paralelně běžících vláken [2].

CUDA zajišťuje paralelní výpočty pomocí abstrakce vláken, bloků a mřížek. Vlákna v kernelu jsou rozlišována pomocí indexů. Pro přístup k datům využívá každé vlákno svůj index. Bloky jsou skupiny vláken. Vlákna v rámci jednoho bloku mohou být prováděna sériově nebo paralelně a mohou být koordinována pomocí funkce pro synchronizaci. Tato funkce umožní zastavit vlákno v určitém bloku v kernelu do doby, než stejného bodu dosáhnou i ostatní vlákna. Každé vlákno je uvnitř bloku identifikováno indexem. Tento index je ve spuštěném kernelu dostupný pomocí zabudované proměnné *threadIdx*. Mřížka je skupina bloků. Synchronizace mezi jednotlivými bloky nejde nastavit. Bloky v mřížce jsou identifikovatelné indexem. Ve spuštěném kernelu je index dostupný přes proměnnou *blockIdx*. Vlákna v ke1rnelu tak můžeme identifikovat pomocí vzorce:

```
id = pořadí bloku * velikost bloku + pořadí vlákna,          (3.1)
```

což v jazyce C CUDA vypadá následovně:

```
int threadID = blockIdx.x * blockDim.x + threadIdx.x;
```

Uvedený vzorec je platný pro jednorozměrné bloky. Pokud máme vícerozměrné bloky, například u dvourozměrných bloků výpočet identifikátoru vlákna probíhá pomocí stejného vzorce s použitím souřadnice *y*.

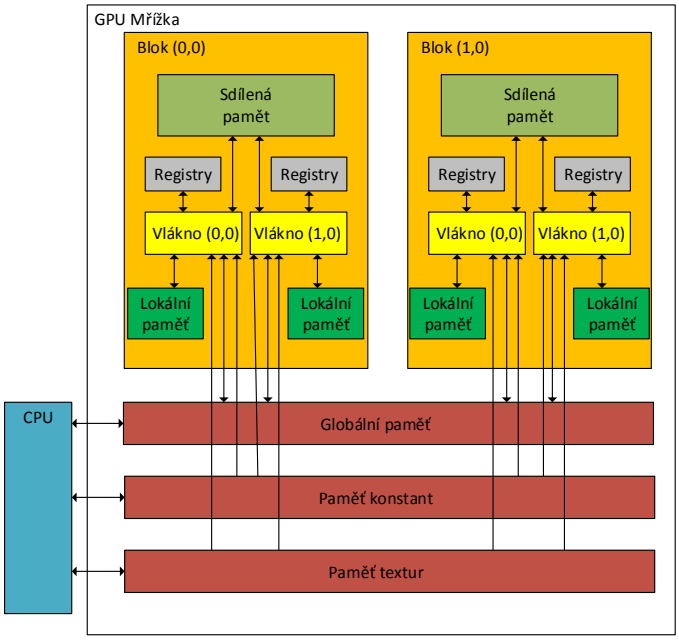
Synchronizace kernelu se může provádět přímo zavoláním synchronizační funkce nebo nepřímo. Nepřímo se provádí, pokud se host pokusí o přístup k paměti na zařízení. V obou případech dojde k zablokování hosta do doby, než budou dokončeny všechny dříve spouštěné kernely.

Kernel se spouští okamžitě podle nastavené konfigurace a podle argumentů funkce pokud je CUDA zařízení v klidovém stavu. Po spuštění kernelu host pokračuje na další řádek kódu. V tomto bodě může CUDA zařízení i host současně provádět své samostatné programy.

Balíky vláken, které jsou zpracovávány v jednom okamžiku se názyvají warpy. Každý warp je vykonáván v tzv. SIMD módě (Single Instruction Multiple Data), tj. všechny vlákna uvnitř warpu musí vykonávat v daném čase stejné instrukce. Při SIMD může vzniknout problém, který se nazývá branch divergence. Při tomto problému může dojít k velké ztrátě výkonu (až 50 %) [12].

4.1.3 Paměťový model

CUDA využívá různých typů pamětí: registry, lokální, sdílenou, globální, paměť textur a paměť konstant [1].



Obrázek 3: Paměťový model

Registry:

- jsou uloženy na multiprocesorech,
- rozdělení mezi jednotlivé procesory plánuje překladač,
- každé vlákno může přistupovat jen ke svým registrům,
- jsou nejrychlejší, protože jsou uloženy na čipu.

Lokální paměť:

- využívá se v případě vyčerpání registrů,
- je přístupná pouze jednomu vláknu,
- je pomalá, protože je uložena v globální paměti.

Sdílená paměť:

- je přístupná všem vláknům v rámci bloku,
- je rychlá, protože je uložena přímo na čipu,
- vlákna k přístupu využívají tzv. banky,
- definuje se pomocí příkazu `__share__`,

- nevýhodou je, že tato paměť je omezená a pokud kernel žádá více sdílené paměti než na multiprocesoru je, tak nelze ani spustit.

Globální paměť:

- sdílená pro všechny multiprocesory,
- řádově pomalejší než registry a sdílená paměť.

Paměť konstant:

- určena pouze pro čtení,
- může být rozesílána broadcastem,
- definuje se pomocí `__constant__`.

Paměť textur:

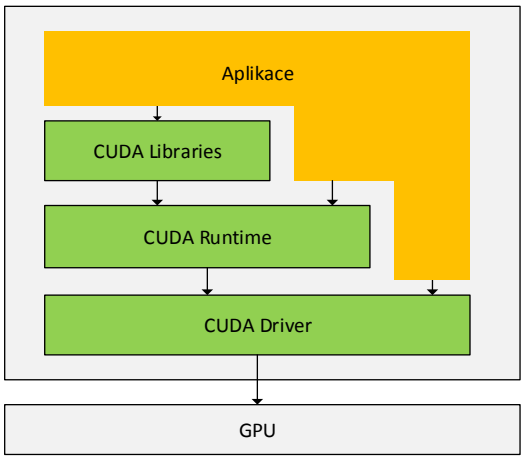
- určena pouze pro čtení,
- vlastní omezenou cache,
- slouží pro práci s texturami.

Tabulka 1: Používané paměti na CUDA – shrnutí:

Typ	**Umístění**	**Cachování**	**Přístup**	**Viditelnost**	**Životnost**
Registry	Na čipu	Ne	R/W	1 vlákno	Vlákno
Lokální	RAM	Ano od v. 2.0	R/W	1 vlákno	Vlákno
Sdílená	Na čipu	Ne	R/W	Vlákna v rámci bloku	Blok
Globální	RAM	Ano od v. 2.0	R/W	Všechna vlákna + host	Do uvolnění
Paměť textur	RAM	Ano	R	Všechna vlákna + host	Do uvolnění
Paměť konstant	RAM	Ano	R	Všechna vlákna + host	Do uvolnění

4.1.4 CUDA API

Kompletní CUDA kód se skládá z dvou částí z kódu pro hosta a kódu pro GPU zařízení. Aby bylo možné tyto části nějak jednoduše identifikovat, vznikly CUDA Driver API a CUDA Runtime API [5]. Na obrázku 4. můžeme vidět hierarchii jednotlivých API.



Obrázek 4: CUDA Runtime API

Jednotlivé funkce tak můžeme deklarovat pomocí těchto příkazů:

`__device__` – funkce, která má toto označení může být vykonávána pouze na GPU a volána taktéž pouze ze zařízení GPU,

`__host__` – naopak takto definovaná funkce může být volána a vykonávána pouze na CPU zařízení,

`__global__` – takto je definovaný kernel, funkce která je vykonávána pouze na GPU, ale volána je z části kódu určeného pro CPU. Kernel je volán pomocí syntaxe `kernel_name<<<…>>>(args)`,

`__device__ __host__` – takto je definována funkce, která může být volána i vykonávána na GPU i CPU zařízení.

Příkazy pro definování proměnných na GPU:

`__shared__` – definování proměnné, která bude uložena ve sdílené paměti,

`__constant__` – proměnná, která bude definována jako konstanta.

Další potřebné příkazy a funkce pro programování CUDA aplikace:

Příklad alokace paměti na GPU (jedná se o globální paměť):

```
cudaMalloc((void**) parametr1, parametr2)
```

parametr1 je ukazatel na alokovanou paměť, *parametr2* určuje požadovanou velikost pro alokaci paměti.

Příklad kopírování dat z paměti hosta do paměti zařízení – konkrétně do globální paměti:

```
cudaMemcpy(parametr1, parametr2, parametr3, parametr4)
```

parametr1 jde o ukazatel cílové paměti, *parametr2* je ukazatel zdrojové paměti, *parametr3* udává velikost kopírované paměti v bytech, *parametr4* definuje z jakého druhu paměti do jakého druhu paměti, bude kopírováno. *Parametr4* může nabývat například těchto hodnot:

- cudaMemcpyHostToDevice – bude kopírováno z paměti hosta do paměti zařízení,
- cudaMemcpyDeviceToHost – bude kopírováno z paměti zařízení do paměti hosta.

Příklad uvolnění prostředků v paměti GPU zařízení:

```
cudaFree(parametr1)
```

parametr1 je ukazatel na požadovanou paměť.

V této kapitole jsem uvedl nejčastěji používané definice funkcí při vývoji CUDA aplikací. Více o definici funkcí se dočtete na webových stránkách společnosti NVIDIA a v [8].

4.1.5 Optimalizování aplikace v CUDA

Není těžké vytvořit aplikaci využívající CUDA, ale pokud chceme využít potenciál CUDA a co nejlepšího paralelního zpracování je nutné kernel optimalizovat, aby se zbytečně neztrácel výkon.

Homogenita warpu má velký vliv na propustnost. Pokud všechna vlákna vykonávají stejné instrukce, pak všechny stream procesory v multiprocesoru mohou provádět stejné instrukce paralelně. Pokud ale jeden nebo více vláken ve warpu provádějí jinou instrukci než ostatní vlákna, pak musí být warp rozdělen a výpočet začíná být prováděn sériově. Tento problém snižuje propustnost, vlákna se čím dál více rozpadají do menších skupin. Proto mezi vlákny musíme udržovat co největší homogenitu [2].

CUDA aplikaci můžeme také optimalizovat pomocí použití vhodnějších typů pamětí. Rychlejší typy pamětí například paměť konstant nebo sdílená paměť mohou výrazně zvýšit výkon aplikace.

4.2 OpenCL

OpenCL (Open Computing Language) je standard pro paralelní programování. Od roku 2008 vývoj probíhá pod dohledem konsorcia Khronos, které sdružuje zastoupení nejvýznamnějších společností v tomto oboru. Mezi firmami, které jsou zastoupeny v konsorciu, jsou například AMD, IBM nebo NVIDIA.

Standard OpenCL definuje abstraktní hardwarové zařízení a k němu softwarové rozhraní. Pomocí softwarového rozhraní mohou jednotlivé aplikace využívat výpočetní možnosti jednotlivých hardwarových komponent. OpenCL tvoří abstraktní modely, které určují požadované vlastnosti a chování OpenCL, OpenCL Framework (obsahuje definici OpenCL API) a specifikace programovacího jazyka (OpenCL C).

Abstraktní modely se rozdělují na model platformy, exekuční model, paměťový model a programovací model [13].

Výhodou OpenCL oproti technologii CUDA je to, že OpenCL je standard. OpenCL lze využívat grafických kartách (NVIDIA GeForce řada 8xxx a vyšší, ATI Radeon HD od řady 4xxx), na klasických procesorech x86 s podporou instrukcí SSE3, na novějších mobilních čipech, atd.

5 Výpočty s velkými čísly

Tato kapitola se zabývá výpočty s velkými čísly, které jsou potřebné pro výpočty modulárního mocnění. V kapitole jsou podrobně popsány algoritmy násobení, dělení a druhé mocniny a dále jsou zde popsány metody výpočtů modulárního mocnění.

5.1 Násobení

Pokud x a y jsou celá čísla vyjádřené pomocí číselné soustavy b: $x = (x_n x_{n-1} \dots x_1 x_0)_b$ a $y = (y_t y_{t-1} \dots y_1 y_0)_b$, pak součin $x \times y$ pak bude mít nejvíce $(n + t + 2)$ číslic. Algoritmus 1 je založen na klasické metodě „tužka a papír“ a využívá dílčích součinů. Jestliže čísla x_j a y_i jsou dvě čísla z báze b, pak můžeme zapsat $x_j \times y_i = (uv)_b$, kde u a v jsou čísla z číselné soustavy s bází b a u může být 0 [28].

Pseudokód 1: Násobení velkých čísel [28]

Vstup: kladná celá čísla x a y, mající $n + 1$ a $t + 1$ číslic číselné soustavy b
Výstup: součin $x \times y = (w_{n+t+1} \dots w_1 w_0)_b$ v číselné soustavě b

1. Pro i od 0 do $(n + t + 1)$ vykonej: $w_i \leftarrow 0$
2. Pro i od 0 do t vykonej:
   2.1 $c \leftarrow 0$
   2.2 Pro j od 0 do n vykonej: $(uv)_b = w_{i+j} + x_j \cdot y_i + c$, $w_{i+j} \leftarrow v$, $c \leftarrow u$
   2.3 $w_{i+n+1} \leftarrow u$
3. Vrať $((w_{n+1}\, w_n \dots w_1\, w_0)_b)$

Příklad výpočtu:

Jsou dána dvě čísla x a y, kde $x = x_3x_2x_1x_0 = 8236$ a $y = y_2y_1y_0 = 452$ (čísla jsou v desítkové soustavě), $n = 3$ a $t = 2$

Tabulka 2: Ukázka výpočtu násobení

i	j	c	$w_{i+j} + x_j y_i + c$	u	v	w_6	w_5	w_4	w_3	w_2	w_1	w_0
0	0	0	0 + 12 + 0	1	2	0	0	0	0	0	0	2
	1	1	0 + 6 + 1	0	7	0	0	0	0	0	7	2
	2	0	0 + 4 + 0	0	4	0	0	0	0	4	7	2
	3	0	0 + 16 + 0	1	6	0	0	1	6	4	7	2
1	0	0	7 + 30 + 0	3	7	0	0	1	6	4	7	2
	1	3	4 + 15 + 3	2	2	0	0	1	6	2	7	2
	2	2	6 + 10 + 2	1	8	0	0	1	8	2	7	2
	3	1	1 + 40 + 1	4	2	0	4	2	8	2	7	2
2	0	0	2 + 24 + 0	2	6	0	4	2	8	6	7	2
	1	2	8 + 12 + 2	2	2	0	4	2	2	6	7	2
	2	2	2 + 8 + 2	1	2	0	4	2	2	6	7	2
	3	1	4 + 32 + 1	3	7	3	7	2	2	6	7	2

5.2 Druhá mocnina

U algoritmu 1 je výraz $(uv)_b$, ve kterém u a v byly celá čísla vyjádřená pomocí jedné cifry. U metody druhé mocniny se může celé číslo u vyjádřit s přesností na dvě cifry $0 \leq u \leq 2(b\text{-}1)$ viz algoritmus 2 [28].

Princip druhé mocniny je založen na skutečnosti, že se rovnají členy nad diagonálou a pod diagonálou, které vznikají při vytváření dílčích součinů. Proto tyto součiny stačí vypočítat pouze jednou a výsledky dílčích součinů vynásobit dvěma [22].

Pseudokód 2: Druhá mocnina [28]

Vstup: přirozené číslo x, mající t číslic a je v číselné soustavě s bází b
Výstup: součin $x \cdot x = (w_{2t-1} \ldots w_1 w_0)_b$ v číselné soustavě s bází b

1. Pro i od 0 do $(2t - 1)$ vykonej: $w_i \leftarrow 0$
2. Pro i od 0 do $(t - 1)$ vykonej:
   2.1 $(uv)_b \leftarrow w_{2i} + x_i \cdot x_i$, $w_{2i} \leftarrow v$, $c \leftarrow u$
   2.2 Pro j od $(i + 1)$ do (t – 1) vykonej:
      $(uv)_b = w_{i+j} + 2x_j \cdot x_i + c$, $w_{i+j} \leftarrow v$, $c \leftarrow u$
   2.3 $w_{i+t} \leftarrow u$
3. Vrať $((w_{2t-1} w_{2t-2} \ldots w_1 w_0)_b)$

Příklad výpočtu:

Je dáno číslo $x = 638$, $t = 3$ a $b = 10$

Tabulka 3: Ukázka výpočtu druhé mocniny

i	j	$w_{2i} + x_i^2$	$w_{i+j} + 2x_jy_i + c$	u	v	w_5	w_4	w_3	w_2	w_1	w_0
0	-	0+64	-	6	4	0	0	0	0	0	4
	1	-	0+2*8*3+6	5	4	0	0	0	0	4	4
	2	-	0+2*8*6+5	10	1	0	0	0	1	4	4
				10	1	0	0	10	1	4	4
1	-	1+9		1	0	0	0	10	0	4	4
	2		10+2*6*3+1	4	7	0	0	7	0	4	4
				4	7	0	4	7	0	4	4
2	-	4+36		4	0	0	0	7	0	4	4
				4	0	4	0	7	0	4	4

5.3 Modulární redukce pomocí klasického dělení

Dělení je nejsložitější operace ze základních matematických operací (sčítání, odčítání, násobení a dělení). Algoritmus s vícenásobnou přesností vypočítá podíl q a zbytek r po dělení čísla x číslem y číselné soustavě s bází b.

Pseudokód 3: Dělení velkých čísel [28]

Vstup: přirozené čísla $x = (x_n \dots x_1\ x_0)_b$ a $y = (y_t \dots y_1\ y_0)_b$, kde $n \geq t \geq 1$, $y_t \neq 0$
Výstup: podíl $q = (q_{n-t} \dots q_1\ q_0)_b$, zbytek $r = (r_t \dots r_1\ r_0)_b$, $x = qy + r$, $0 \leq r < y$

1. Pro i od 0 do $(n - t)$ vykonej: $q_i \leftarrow 0$
2. Pokud $(x \geq yb^{n-t})$ vykonej: $q_{n-t} \leftarrow q_{n-t} + 1$, $x \leftarrow x - yb^{n-t}$
3. Pro i od n do $(t + 1)$ vykonej:
   3.1 Jestli $(x_i = y_t)$ pak $q_{n-t-1} \leftarrow b - 1$; jinak $q_{n-t} \leftarrow \lfloor (x_i b + x_{i+1})/y_t \rfloor$
   3.2 Pokud $(q_{i-t-1}(y_t b + y_{t-1}) > x_i b^2 + x_{i-1} + x_{i-2})$ vykonej: $q_{i-t-1} \leftarrow q_{i-t-1} - 1$
   3.3 $x \leftarrow x - q_{i-t-1}\, yb^{i-t-1}$
   3.4 Jestli $(x < 0)$ pak $x \leftarrow x + yb^{i-t-1}$, $q_{i-t-1} \leftarrow q_{i-t-1} - 1$
4. $r \leftarrow x$
5. Vrať (q, r)

Druhý krok algoritmu 3 je vykonáván nejvíce jedenkrát jestliže $y_t \geq \frac{b}{2}$ a b je sudé. Podmínka $n \geq t \geq 1$ může být nahrazena podmínkou $n \geq t \geq 0$ za předpokladu $x_j = y_j = 0$ a index $j < 0$. Odhad podílu číslic q_{n-t-1} v kroku 3.1 není nikdy menší než skutečná hodnota podílu číslic. Mimoto jestli je $y \geq \left\lfloor \frac{b}{2} \right\rfloor$, pak se krok 3.2 neopakuje více než dvakrát. Jestliže je krok 3.1 upraven do podoby $q_{n-t-1} \leftarrow \lfloor (x_i b^2 + x_{i-1} b + x_{i-2})/(y_t b + y_{t-1}) \rfloor$ tak pak je odhad téměř vždy správně a krok 3.2 je opakován nejvýše jednou. Pro dosažení podmínky $y \geq \left\lfloor \frac{b}{2} \right\rfloor$ musíme vykonat normalizaci [28]. Při normalizaci jsou nahrazeny čísla x a y čísly λ_x a λ_y. Podíl po dělení $\frac{\lambda x}{\lambda y}$ je stejný jako $\frac{x}{y}$ a zbytek je λ-krát větší než zbytek po dělení $\frac{x}{y}$. Pokud zvolíme λ jako mocninu 2, pak můžeme vykonávat jednoduché bitové posuny doleva nebo doprava pro získání výsledku.

Pokud při normalizaci se zvětší počet číslic čísla x o 1, pak každá iterace kroku 3 vyžaduje $t + 3$ jednociferných násobení. Algoritmus pseudokódu 3 vyžaduje $(n - t)(t + 3)$ jednociferných násobení a $n - t$ jednociferných dělení, protože je algoritmus vykonán $n - t$ krát.

Příklad výpočtu:

Jsou dána dvě celá čísla x a y, $x = 628212154$, $y = 27531$, $n = 8$, $t = 4$

Tabulka 4: Ukázka výpočtu dělení se zbytkem

i	q_4	q_3	q_2	q_1	q_0	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
-	0	0	0	0	0	6	2	8	2	1	2	1	5	4
8	2	0	0	0	0		7	7	5	9	2	1	5	4
7	2	2	0	0	0		2	2	5	3	0	1	5	4
6	2	2	8	0	0			5	0	5	3	1	5	4
5	2	2	8	1	0				2	3	0	0	0	4
	2	2	8	1	8						9	7	5	6

5.4 Montgomeryho redukce

Montgomeryho redukce je technika, která zefektivní výpočty modulárního násobení a mocnění.

Je-li celé kladné číslo m a celá čísla R a T, kde $R > m$ a současně GCD $(m, R) = 1$ a $0 \leq T \leq mR$. Výpočet provede $TR^{-1} \bmod m$ bez použití klasického dělení [28]. $TR^{-1} \bmod m$ se nazývá Montgomeryho redukce T modula m vzhledem k R.

Pokud x a y jsou celá čísla, která splňují podmínku $0 \leq x,y < m$, potom nechť jsou $x' = xR \bmod m$ a $y' = yR \bmod m$. Montgomeryho redukce výrazu $x'y'$ pak je $x'y'R^{-1} \bmod m = xyR \bmod m$.

Například $x^5 \bmod m$, kde x je celé číslo $1 \leq x < m$. Nejprve se vypočítá $x' = xR \bmod m$. V dalším kroku se vypočítá Montgomeryho redukce $x'x'$, ve které je $A = x^2R^{-1} \bmod m$. Montgomeryho redukce A^2 je $A^2R^{-1} \bmod m = x^4R^{-3} \bmod m$. Montgomeryho redukce $(A^2R^{-1} \bmod m)x$ je $(A^3R^{-1})XR^{-1} \bmod m = x^5R^{-4} \bmod m = x^5R \bmod m$. Vynásobením hodnoty R^{-1} a redukce modula m se dostane $x^5 \bmod m$ [28].

Jestliže je m vyjádřeno celým číslem délky n v číselné soustavě s bází b, pak $R = b^n$.

Pokud celá čísla m a R splňují podmínky GCD(m, R) = 1, $m' = -m - 1 \bmod R$ a číslo T bude celé číslo z podmínky $0 \leq T < mR$ a jestliže je splněna podmínka $U = Tm' \bmod R$, pak $(T + Um) / R$ je celé číslo a $(T + Um) / R \equiv TR^{-1} \bmod m$.

Pseudokód 4: Montgomeryho redukce [28]

Vstup: přirozené čísla m mající n číslic a bázi číselné soustavy b,
$(m, b) = 1$, $R = b^n$, $m' = -m^{-1} \bmod b$, $T < mR$, mající $2n$ číslic
Výstup: $TR^{-1} \bmod m$
1. $A \leftarrow T$ (kde $A = (a_{2n-1} \ldots a_1 a_0)_b$)
2. Pro i od 0 do $n - 1$ vykonej:
   2.1 $u_i \leftarrow a_i m' \bmod b$
   2.2 $A \leftarrow A + u_i m b^i$
3. $A \leftarrow A/b^n$
4. Jestli $A \geq m$ pak $A \leftarrow A - m$
5. Vrať (A)

5.5 Montgomeryho násobení

Montgomeryho násobení kombinuje Montgomeryho redukci a klasické násobení dvou čísel. Algoritmus 5 prokládá násobení s redukcí, proto nelze využít metodu druhé mocniny.

Pseudokód 5: Montgomeryho násobení [28]

Vstup: přirozené čísla m, x a y mající n číslic a bázi číselné soustavy b,
$0 \leq x, y < m$, $(m, b) = 1$, $R = b^n$, $m' = -m^{-1} \bmod b$
Výstup: $xyR^{-1} \bmod m$
1. $A \leftarrow 0$ (kde $A = (a_n \ldots a_1 a_0)_b$)
2. Pro i od 0 do $n - 1$ vykonej:
   2.1 $u_i \leftarrow (a_0 + x_i y_0) m' \bmod b$
   2.2 $A \leftarrow (A + x_i y + u_i m)/b$
3. $A \leftarrow A/b^n$
4. Jestli $A \geq m$ pak $A \leftarrow A - m$
5. Vrať (A)

Montgomeryho násobení vyžaduje $2n(n + 1)$ násobení jednociferných čísel, protože krok 2.1 vyžaduje 2 násobení jednociferných čísel, krok 2.2 vyžaduje počet těchto násobení $2n$ a celkově jsou tyto kroky vykonány n-krát.

Dělení v tomto algoritmu jsou realizovány jako posuny. Proto není potřeba v tomto algoritmu vykonávat operace, které jsou nutné pro klasické dělení, jako například odhad číslic podílu a následné vykonávání korekčních kroků [22].

Nevýhoda Montgomeryho násobení je konverze operátorů do speciálního tvaru před výpočtem a zpětný převod výsledku ze speciálního tvaru do klasické podoby po dokončení výpočtu.

5.6 Modulárního mocnění

Existuje několik metod, jak efektivně aplikovat funkci modulárního mocnění. Mezi nejznámější patří:

- přímočarý postup,
- binární modulární mocnění z leva doprava s využitím metody squre-and-multiply,
- zleva doprava k-nární modulární mocnění s k-bitovým pevným oknem.

Tyto metody jsou podrobně popsány v [15] a [16].

Knihovna GMP používá k modulárnímu mocnění algoritmus 2^k-nárný s posuvným oknem, kde číslo k se volí podle velikosti exponentu. Čím bude exponent větší, tím bude větší i hodnota čísla k. Modulární násobení používá buď jednoduché dělení nebo Montgomeryho metodu REDC [15]. Podobnou metodu modulárního mocnění využívají i knihovny OpenSSL, MPIR a LibTomMath, s tím rozdílem, že LibTomMath v algoritmu modulárního mocnění `s_mp_exptmod` využívá Barrettovu redukci a v algoritmu `mp_exptmod_fast` používá Montgomeryho redukci [17].

Základní binární mocnění

Tento algoritmus je založen na technikách druhého mocnění a násobení (squre-and-multiply).

Pokud $t + 1$ je bitová délka binární reprezentace exponentu e a $wt(e)$ vyjadřuje počet jedniček v této binární reprezentaci, tak celkový počet násobení, které algoritmus provádí je $wt(e) - 1$ a druhé mocnění provádí t-krát. Jestliže exponent e leží v rozmezí $0 \leq e < |G| = n$, pak můžeme očekávat $\lfloor \lg n \rfloor$ druhých mocnin a ½ ($\lfloor \lg n \rfloor + 1$) násobení. Jestliže se zamyslíme nad skutečností, že náročnost výpočtu druhé mocniny je přibližně stejná jako násobení pak můžeme očekávat počet násobení $3/2 \lfloor \lg n \rfloor$ [28].

Pseudokód 6: Binární mocnění zprava doleva

Vstup: element $g \in$ G a celé číslo $e \geq 1$
Výstup: g^e
1. A ← 1, $S \leftarrow g$
2. Pokud $e \neq 0$ vykonej:
   2.1 Jestli e je liché pak $A \leftarrow A \cdot S$
   2.2 $e \leftarrow \lfloor e/2 \rfloor$
   2.3 Jestli $e \neq 0$ pak $S \leftarrow S \cdot S$
3. Vrať (A)

Pokud je exponent e lichý, tak algoritmus násobí $A \times S$. V některých případech je efektivnější spočítat $A \times g$ než $A \times S$ viz pseudokód 7.

Pseudokód 7: Binární mocnění zleva doprava

Vstup: element $g \in G$ a celé kladné číslo $e = (e_t\ e_{t-1}\ \ldots\ e_1\ e_0)_2$
Výstup: g^e
1. $A \leftarrow 1$
2. Pro i od t do 0 vykonej:
   2.1 $A \leftarrow A \cdot A$
   2.2 Jestli $e_i = 1$ pak $A \leftarrow A \cdot g$
3. Vrať (A)

Algoritmus je podobný tomu předchozímu, $t + 1$ je bitová délka binární reprezentace exponentu e a $wt(e)$ vyjadřuje počet jedniček v této binární reprezentaci, počet násobení a druhých mocnění je stejný jako v předchozím případě. Tento algoritmus se liší od předchozího v druhém kroku pěvně daným atributem g [28]. Pokud má g speciální strukturu, tak násobení může být podstatně rychlejší než násobení dvou různých čísel.

Algoritmus posuvného okna

Algoritmus posuvného okna ve srovnání s předchozími algoritmy snižuje množství násobení, počet provedení druhých mocnin zůstává stejný. Element k udává velikost okna.

Algoritmus k-nárného mocnění s posuvným oknem pracuje na principu dvou bloků, které jsou od sebe vzájemně odděleny nulami. Nejprve exponent rozložíme do bloků, kde každý blok bude mít maximální délku k. Tyto bloky budou od sebe odděleny nulovými bloky. Nulové bloky slouží k umocňování na druhou. Exponent se prochází bit po bitu od MSB po LSB [22]. Blok začíná v místě, na kterém je bit nastavený na 1 a končí po přečtení k bitů nebo po bitu po kterém následuje k bitů nulových. Celý postup je ukázán viz Pseudokód 8:

Pseudokód 8: Mocnění pomocí posuvného okna [28]

Vstup: element $g \in G$, celé kladné číslo $e = (e_t\, e_{t-1} \ldots e_1\, e_0)_2$, $e_t = 1$, $k \geq 1$
Výstup: g^e
1. Před počítání
   1.1 $g_1 \leftarrow g$, $g_2 \leftarrow g^2$
   1.2 Pro i od 2 do $(2^{k-1} - 1)$ vykonej: $g_{2i+1} = g_{2i-1} \cdot g_2$
2. $A \leftarrow 1$, $i \leftarrow t$
3. Pokud $i \geq 0$ vykonej:
   3.1 Jestli $e_i = 0$ potom vykonej: $A \leftarrow A^2$, $i \leftarrow i - 1$
   3.2 Jestli $e_i \neq 0$ najdi nejdelší bitový řetězec $e_i\, e_{i-1} \ldots e_l$ také že $i - l + 1 \leq k$ a $e_l = 1$ a vykonej:
   $A \leftarrow A^{2\wedge(i-l+1)} \cdot g(e_i\, e_{i-1} \ldots e_l)_2$, $i \leftarrow l - 1$
4. Vrať (A)

Montgomeryho mocnění

Algoritmus Montgomeryho mocnění pro výpočet modulárního mocnění v sobě kombinuje modulární násobení a binární mocnění zleva doprava. Při použití tohoto algoritmu musíme nejprve transformovat operand x do Montgomeryho formy [28]. Akumulátor A se nastaví na hodnotu 1. V dalším kroku pro modulární násobení a mocnění použijeme algoritmus binárního mocnění zleva a doprava. V posledním kroku převede výsledek do klasické formy z Montgomeryho formy. Tento algoritmus je znázorněný pomocí pseudokódu viz Pseudokód 9.

Pseudokód 9: Montgomeryho mocnění [28]

Vstup: $m = (m_{l-1} \ldots m_0)_b$, $(m, b) = 1$, $R = b^l$, $m' = -m^{-1} \bmod b$,
$e = (e_t \ldots e_0)$ a $e_t = 1$, celé číslo x, $1 \leq x < m$
Výstup: $x^e \bmod m$
1. $\bar{a} \leftarrow \text{Mont}(x, R^2 \bmod m)$, $A \leftarrow R \bmod m$
2. Pro i od t do 0 vykonej:
   2.1 $A \leftarrow \text{Mont}(A, A)$
   2.2 Jestli $e_i = 1$ pak $A \leftarrow (A, \bar{a})$
3. $A \leftarrow \text{Mont}(A, 1)$
4. Vrať (A)

5.7 Knihovny pro práci s modulární aritmetikou

Součástí praktické části projektu bylo vytvoření programu, který bude používat proměnné typu „Big Integer“ (velikost proměnných až 1024 bitů). Pro matematické operace s takto velkými čísly existují různé knihovny, například GMP, LibTomMath, MPIR, CUMP [19].

GMP

GNU Multiple Precision Arithmetic Library neboli GPM je open-source knihovna, která je vyvíjena od roku 1991 a je psána v jazyce C. Přesnost knihovny je dána použitým hardwarem a využívají ji počítačové systémy Mathematica nebo Maple. Knihovna se musí sestavit pomocí GNU compiler Collection (GCC).

Knihovna obsahuje několik kategorií [15]:

- mpz – tato kategorie pracuje s celými čísly a obsahuje přes 150 aritmetických a logických funkcí.
- mpq – tato kategorie je určena pro racionální čísla a tvoří ji něco kolem 35 různých aritmetických funkcí.
- mpf – kategorie pracující s čísly s plovoucí desetinnou čárkou. V této kategorii nalezneme přes 70 funkcí.

LibTomMath

LibTomMath je knihovna, která obsahuje funkce pro práci s velkými čísly a současně vyhovuje standardu ISO C. Knihovnu tak lze použít na různých platformách Linux, Windows založených na architektuře x86, architektuře ARM, atd. LibTomMath poskytuje širokou škálu optimalizovaných aritmetických a logických operací, například funkce pro Barrettovu redukci, Montgomeryho redukci, atd. (seznam všech možných operací nalezneme v [17]).

MPIR

MPIR (Multiple Precision Integers and Rationals) je open-source knihovna, která vychází z knihovny GMP a je psána v jazyku C. Knihovna má stejné části jako GMP. Hlavní výhodou MPIR oproti GMP je větší uživatelská přívětivost a jednodušší instalace do Visual Studia. Komunita, která vyvíjí MPIR se skládá z profesionálních a amatérských matematiků, počítačových vědců a nadšenců. Přehled všech algoritmů nalezneme v manuálu MPIR [20].

OPENSSL

OpenSSL je open-source toolkit pro realizaci SSL a TLS protokolů. Projekt je vytvářen pomocí celosvětové komunity dobrovolníků komunikujících pomocí internetu. Jedna z částí této velké knihovny se nazývá „BN“, která pracuje s datovým typem „BIGNUM“ a obsahuje funkce pro celočíselnou aritmetiku. Seznam všech funkcí z této části nalezneme v [21].

CUMP

CUMP je další knihovna, která vychází z GMP. Knihovna je upravená pro práci s velkými čísly s využitím technologie CUDA. Bohužel knihovna v současné verzi 1.0.1 pokrývá pouze základní matematické operace sčítání, odčítání a násobení pro čísla s plovoucí desetinnou čárkou. Knihovna je určena pro 64-bit Linux s nainstalovaným toolkitem CUDA 4.0 a vyšší a GPU musí mít verzi „Compute Capability“ alespoň 2.0.

GMP - BigInt: A GNU Multi-Precision Library for .NET

Jedná se o klasickou knihovnu GMP, která byla upravena Emilem Stefanoven pro technologie C# a .NET.

The Legion of the Bouncy Castle

The Legion of the Bouncy Castle je API pro vývoj, kryptografických aplikací v technologii C#. Součástí API je knihovna pro práci s datovým typem Big Integer.

System.Numerics

Tato knihovna je pro technologii .NET a je součástí balíčku .NET Framework 4.5. Knihovna obsahuje funkce pro práci s libovolně velkými čísly.

Knihovna pro práci s modulární aritmetikou s velkými čísly na mikrokontrolérech řady MSP430.

Tato knihovna byla vyvinuta na Fakultě elektrotechniky a komunikačních technologií na VUT v Brně. Knihovna vychází z knihovny OpenSSL a je určena pro matematické výpočty modulární aritmetiky s velkými čísly.

6 Praktická část

Úkolem v praktické části semestrálního projektu bylo vytvořit program, který srovná rychlost modulárního mocnění s využitím knihoven popsaných ve čtvrté kapitole a pak jednu z těchto knihoven upravit tak, aby bylo možné funkci pro modulární mocnění spouštět na technologii CUDA.

V diplomové práci jsem si měl vybrat vhodnou platformu pro výpočet modulárního mocnění a měl jsem vytvořit aplikaci typu „Klient – Server“ pro centralizovaný výpočet revokační funkce protokolu HM12. Tento výpočet měl běžet na vybrané platformě paralelně.

6.1 Software použitý při vývoji aplikace

Visual Studio 2012 a NVIDIA CUDA Toolkits 5.5

Jedná se vývojové prostředí IDE od firmy Microsoft. Výhodou tohoto vývojového prostředí je editor kódu, který obsahuje IntelliSense. Technologie IntelliSense umožňuje například automatické dokončování slov při psaní kódu nebo zobrazení informací o parametrech.

CUDA Toolkits 5.5 přidává do Visual Studia možnost vytváření CUDA aplikací v programovacím jazyku C/C++. Toolkits obsahuje kompilátor pro NVIDIA GPU, matematické knihovny, nástroje pro ladění a optimalizaci výkonu aplikací. Společně s CUDA Toolkit 5.5 se také nainstalují uživatelské manuály, programovací příručky, které jsou užitečné při tvorbě vlastní aplikace.

Visual Studio 2013 a Visual Studio Online

Během vývoje aplikace společnost Microsoft přišla na trh s novou řadou Visual Studia. Visual Studio 2013 přináší oproti dřívější verzi vylepšenou inspekci kódu.

Visual Studio Online vychází z nástrojů Team Foundation a běží na serverech Windows Azure. VS Online umožňuje spravování projektu v cloudu, obsahuje webový editor kódu s označením Monaco, systém pro správu požadavků (requirements) a testovacích scénářů (test suites). Testovací scénáře jsem při vývoji respektive při testování aplikace nevytvářel, ale průběžně jsem si vytvářel jednotlivé požadavky pro vývoj a zaznamenával chyby při testování, abych zajistil dostatečnou kvalitu aplikace.

NVIDIA NSIGHT Visual Studio Edition 3.2

Další doplněk pro Visual Studio, který je dostupný zdarma pro registrované vývojáře CUDA NVIDIA. Tento nástroj pomáhá při analyzování výkonu aplikace.

NVIDIA NSIGHT Eclipse Edition 3.1

Plugin pro vývojové prostředí Eclipse. Tento plugin umožní vytvářet aplikace pro CUDA v operačním systému Ubuntu 13.10.

6.2 Zařízení použité při vývoji

Při vývoji aplikace jsem používal grafickou kartu NVIDIA GeForce GT 555M. Tato grafická karta je založena na architektuře Fermi. Podrobnou specifikaci této karty najdeme v tabulce 2.

Grafická karta NVIDIA GeForce GT 555M:

Tabulka 5: Specifikace grafické karty NVIDIA GeForce GT 555M:

CUDA Driver Version / Runtime Version	6.0/5.5
CUDA Capability	2.1
Celková velikost globální paměti	2048 MBytů
CUDA Cores	96
GPU Clock Rate	1505 MHz
Memory Clock Rate	1570 MHz
Celková velikost paměti konstant	65536 bytů
Velikost sdílené paměti v bloku	49152 bytů
Celkový počet registrů v blok	32768
Velikost warpu	32
Maximální počet vláken na multiprocesor	1536
Maximální počet vláken v bloku	1024
Max dimension size of a thread block (x,y,z)	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z)	(65535, 65535, 65535)

Pro výpočty na CPU jsem měl k dispozici 4-jádrový procesor Intel Core i7 2630QM. Základní specifikace tohoto procesoru jsou uvedeny v tabulce 3.

Tabulka 6: Specifikace procesoru 2630QM:

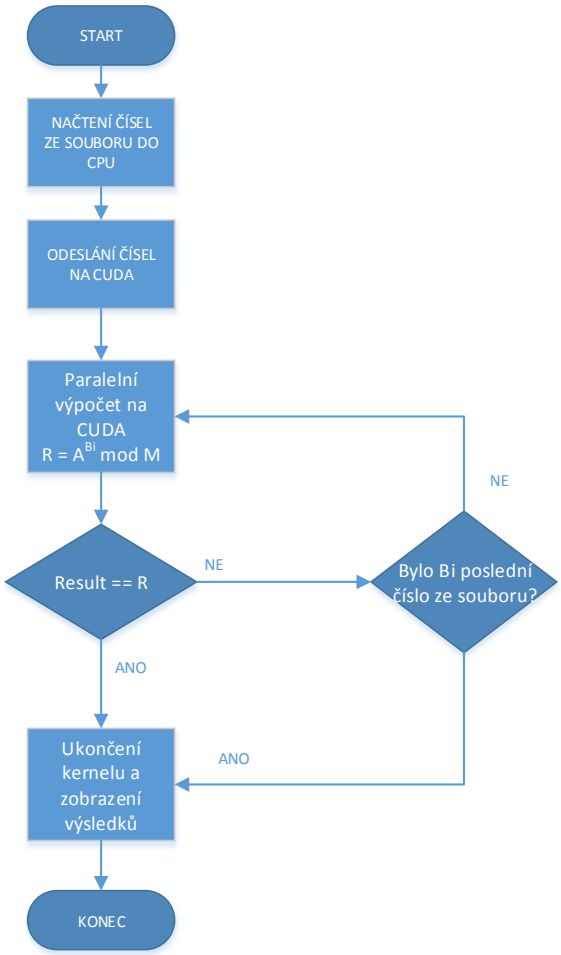
Počet jader	4
Počet vláken	8
Clock speed	2,0 GHz
Max Turbo Frekvence	2,9 GHz

Intel Smart Cache	6 MB
Instruction Set	64-bit

Pro testování aplikací jsem použil grafickou kartu MSI GeForce GTX 650TI Power Edition a procesor Intel Xeon X3440 2.53 GHz. Podrobná specifikace těchto komponent je uvedena v [23] respektive v [24].

6.3 Analýza programů porovnávající CUDA a CPU

Před samotnou realizací projektu jsem si problém pečlivě zanalyzoval. Pro lepší pochopení, jak má program pracovat, jsem si vytvořil v prostředí MS Visio vývojový diagram, který je zobrazen na obrázku 5 – zobrazen je vývojový diagram aplikace využívající technologii CUDA (pro aplikaci využívající pouze CPU je vývojový diagram podobný – pouze nebudou zobrazeny bloky posílající data do GPU).



Obrázek 5: Vývojový diagram aplikace pro CUDA

6.4 Realizace programů porovnávající CUDA a CPU

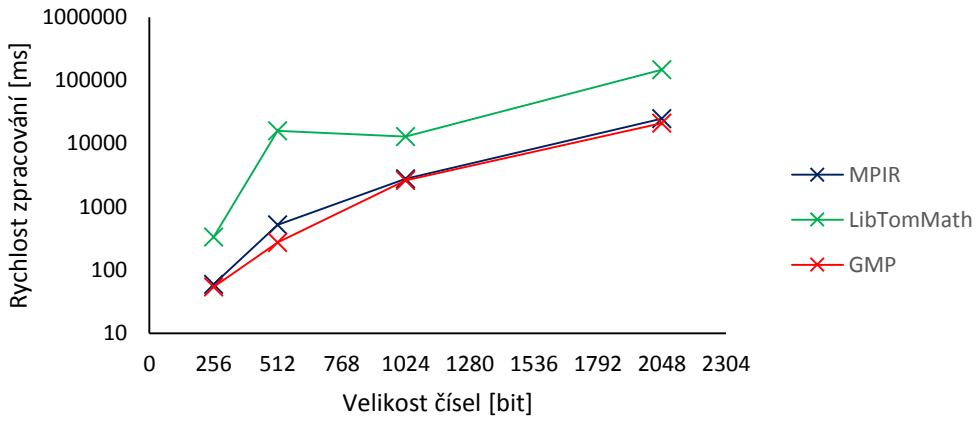
6.4.1 Realizace aplikace využívající k paralelnímu zpracování CPU

Po analýze problému jsem se rozhodl vytvořit aplikaci pro modulární mocnění pomocí knihoven GMP, LibTomMath, MPIR, OpenSSL.

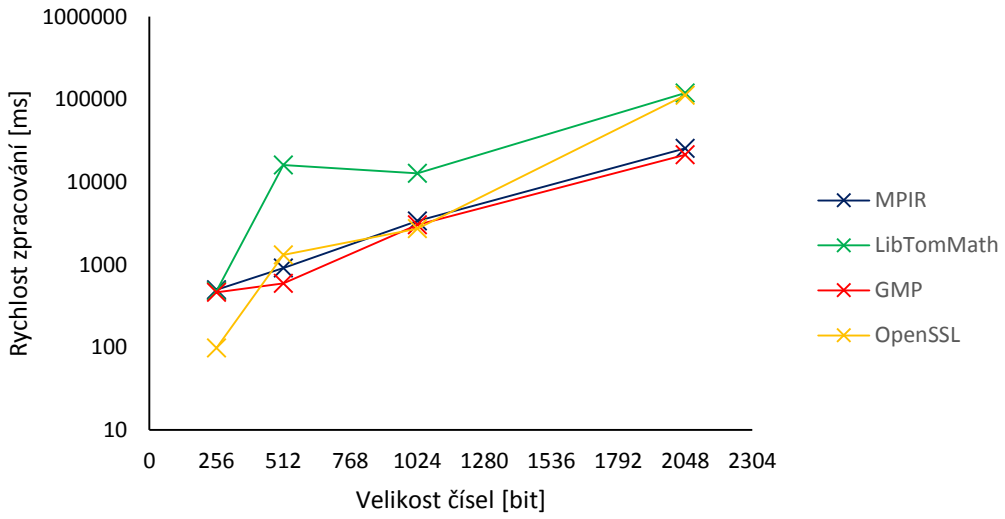
Pro paralelní běh aplikace jsem si vybral standard OpenMP, protože k vývoji aplikace jsem měl k dispozici pouze počítač s jedním procesorem i7.

Program pro modulární mocnění využívající knihovny LibTomMath a MPIR jsem vytvářel ve Visual Studiu 2012 ve Windows 8.1. S knihovnami GMP a OpenSSL jsem pracoval v Ubuntu.

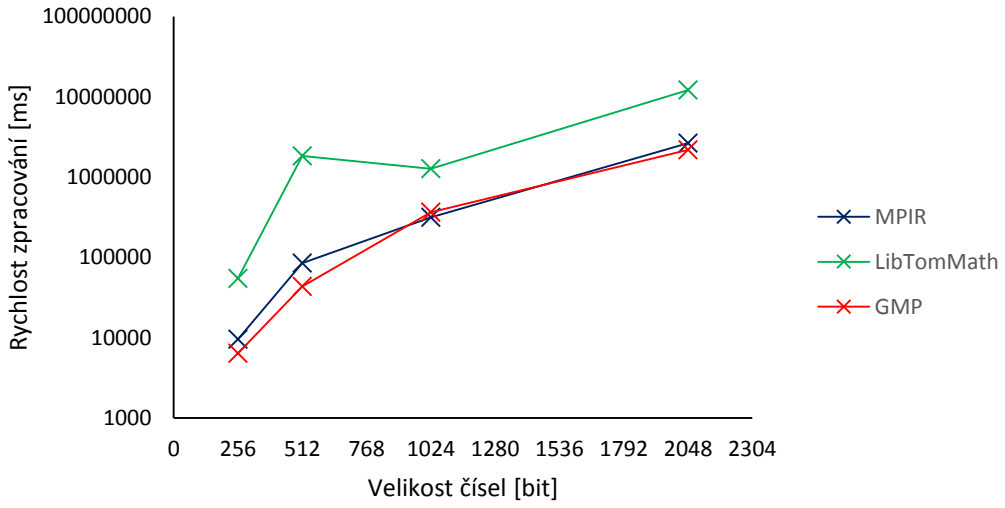
Práce s jednotlivými knihovnami byla bezproblémová, nejvíce mi vyhovovala knihovna MPIR, protože se mi s ní nejlépe pracovalo ve Visual Studiu 2012. Jednotlivé programy byly testovány na číslech o velikosti 256 bitů, 512 bitů, 1024 bitů a 2048 bitů a exponent byl načítaný ze souborů, které obsahovaly 10 000 čísel respektive 1 000 000 čísel.



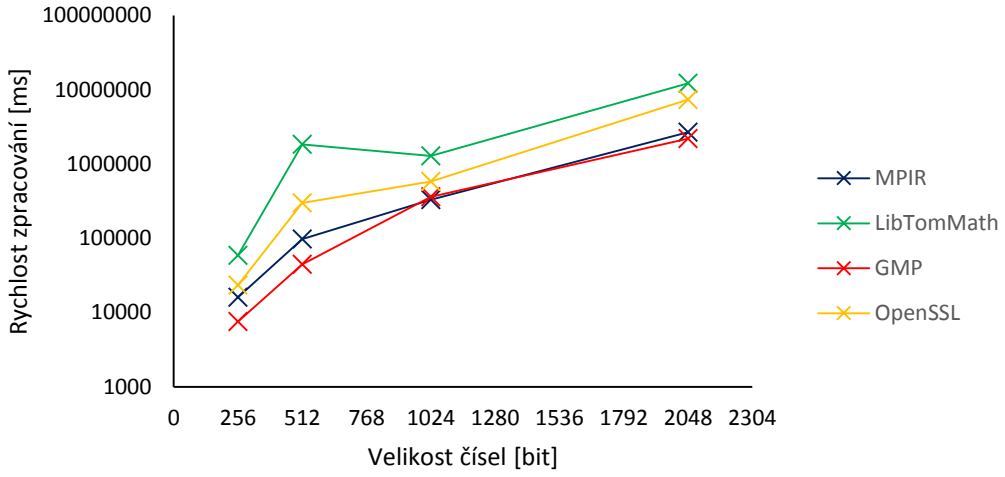
Graf 1: Srovnání rychlosti funkce modulárního mocnění v daných knihovnách pro soubory obsahující 10000 čísel (měřena pouze rychlost zpracování funkce modulárního mocnění)



Graf 2: Srovnání celkové rychlosti knihoven při použití funkce modulárního mocnění pro soubory obsahující 10000 čísel (měřena celková rychlost zpracování všech použitých funkcí při výpočtu)



Graf 3: Srovnání rychlosti funkce modulárního mocnění v daných knihovnách pro soubory obsahující 1000000 čísel (měřena pouze rychlost zpracování funkce modulárního mocnění)



Graf 4: Srovnání celkové rychlosti knihoven při použití funkce modulárního mocnění pro soubory obsahující 1000000 čísel (měřena celková rychlost zpracování všech použitých funkcí při výpočtu)

Testování probíhalo na čtyřjádrovém procesoru INTEL i7 2630QM. V této části projektu jsem se setkal s jediným problémem. Problém souvisel s použitím knihovny OpenMP. Problém byl způsobený špatně definovanou proměnnou, všechny vlákna zapisovaly do stejné proměnné, což způsobovalo pád aplikace. Tuto chybu jsem vyřešil tím způsobem, že jsem z jednoduché proměnné vytvořil pole.

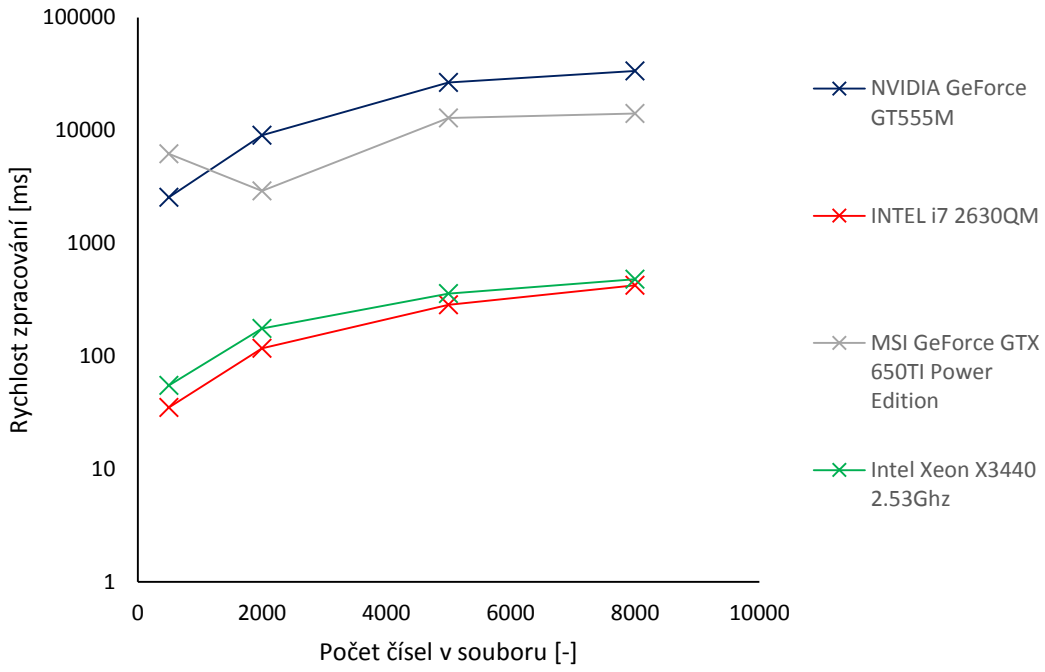
6.4.2 Realizace aplikace využívající CUDA pomocí LibTomMath

Pro vytváření programu využívající CUDA jsem se rozhodl program vytvářet ve vývojovém prostředí Visual Studio 2012. Do tohoto vývojového prostředí bylo nutné doinstalovat NVIDIA CUDA Toolkit. V době vývoje byl dostupný NC Toolkit ve verzi 5.5, který ještě neřešil problém s alokováním paměti a posílám dat

na GPU a zpět do CPU. Od verze 6.0 by měl být tento problém odstraněn a programování na CUDA bude značně ulehčeno.

Po instalaci vývojového prostředí jsem mohl začít se samotnou realizací projektu. Nejprve bylo nutné rozbalit a podrobně zanalyzovat knihovnu LibTomMath. Musel jsem zjistit, které funkce jsou potřeba k funkci modulárního mocnění. V dalším kroku bylo potřeba funkce upravit do stavu, aby je bylo možné volat na grafické kartě. Před některé funkce stačil připsat prefix `__device__ __host__`, jiné bylo potřeba přeprogramovat kvůli tomu, že CUDA nezná některé funkce ze základních knihoven pro C/C++. Ve čtvrtém kroku bylo nutné vytvořit vlastní funkci pro modulární mocnění spustitelnou na CUDA a realizovanou pomocí funkcí z předchozího kroku. V posledním kroku bylo nutné doprogramovat tutéž funkci pro modulární mocnění, ale tentokrát určenou pro CPU, abych mohl porovnat rychlost.

Výpočet CUDA byl několikanásobně pomalejší oproti CPU viz Graf 5. Pomalost přisuzuji problému s divergencí warpu popsaného v kapitole 2.5.



Graf 5: Porovnání rychlosti upravené knihovny LibTomMath na GPU a CPU – testování probíhalo na souborech obsahující 256 bit čísla

6.4.3 Realizace aplikace využívající CUDA pomocí knihovny vycházející z OPENSSL

Knihovna vznikla na Fakultě elektrotechniky a komunikačních technologií na VUT. V knihovně není obsažena funkce pro modulární mocnění, ale pouze pro modulární násobení. Při realizaci programu, který by k modulárnímu násobení využíval technologii CUDA, jsem postupoval obdobně jako u knihovny LibTomMath.

Při vývoji jsem zjistil, že funkce pro modulární mocnění nepracuje správně ani na CPU. Po konzultaci s vývojářem této knihovny jsem se dozvěděl, že knihovna je primárně určena pro 16 bitové mikrokontroléry a obsahuje nevhodné datové typy pro vyšší architektury, tudíž se nedá na klasickém CPU ani na CUDA použít.

6.4.4 Testování knihovny CUMP

Knihovna CUMP je jediná dostupná knihovna pro práci s velkými datovými typy na CUDA. Tato knihovna je vyvíjena pomocí funkcí z knihovny GMP. Avšak vývoj knihovny je teprve na začátku. V současné verzi je knihovna ve verzi 1.0.1. CUMP v této verzi obsahuje pouze funkce sčítání, odčítání a násobení pro mpf (high-level floating-point arithmetic functions).

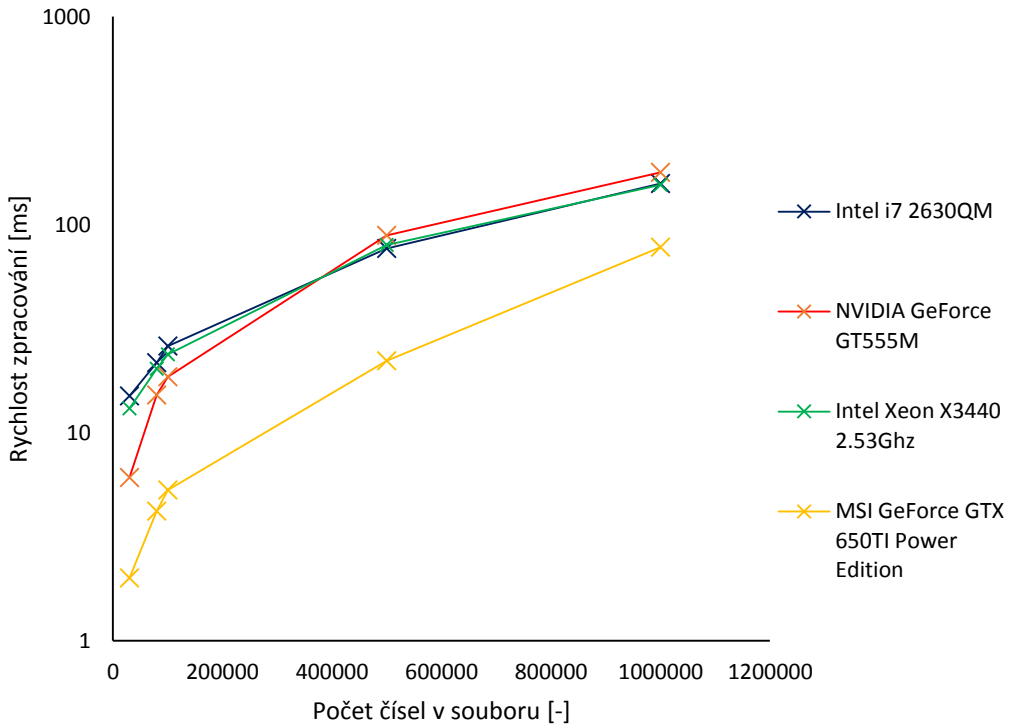
Testování probíhalo na operačním systému Ubuntu 13.10. Při zprovozňování CUDA na Ubuntu se objevil problém. K testování byl určen osobní počítač Lenovo Y570, který obsahuje technologii NVIDIA Optimus. Pro tuto technologii nemá NVIDIA ovladače pro Linux a po instalaci nových driverů ke grafické kartě a CUDA Toolkit zůstala po přihlášení uživatele do OS Ubuntu pouze černá plocha. Tento problém byl nakonec vyřešen pomocí softwaru třetí strany Bumblebee.

Testování CUMP na CUDA probíhalo proti stejné funkci v knihovně GMP na CPU. Testování probíhalo pro vzorec:

$$X[n] = A[n] * B\ [n], \tag{6.1}$$

,kde A a B byly čísla o velikosti 1024 bitů a $n=\{30\ 000, 80\ 000, 100\ 000, 500\ 000, 1\ 000\ 000\}$.

Pro menší počty čísel byly výsledky výrazně lepší pro CUDA, ale se zvyšováním počtu čísel v souboru se výsledky postupně vyrovnávaly, až pro ty největší pole čísel bylo CPU rychlejší než NVIDIA GeForce GT555M, což je zvláštní, protože podle teoretických znalostí bych spíše očekával opačný průběh. Myslím, že postupné vyrovnávání výkonu CUDA a CPU mohlo být způsobeno neúplně optimalizovanou knihovnou CUMP a podobným problémem jako u testování upravené knihovny LibTomMath.



Graf 6: Porovnání rychlosti knihoven GMP a CUMP pro funkci násobení (testování probíhalo se soubory obsahující 1024 bit čísla)

6.5 Analýza programu Klient - Server

Pro vývoj aplikace typu „Klient – Server“ jsem se rozhodl využít technologii CPU, protože během výzkumu v semestrálním projektu se mi nepodařilo dosáhnout efektivního výpočtu modulárního mocnění na technologii CUDA. Zjistil jsem, že pro technologii CUDA nejsou dostupné žádné knihovny pro modulární aritmetiku velkých celých čísel a při pokusu o transformaci existující knihovny pro CPU na platformu GPU se rovněž nepodařilo dosáhnout efektivních výpočtů.

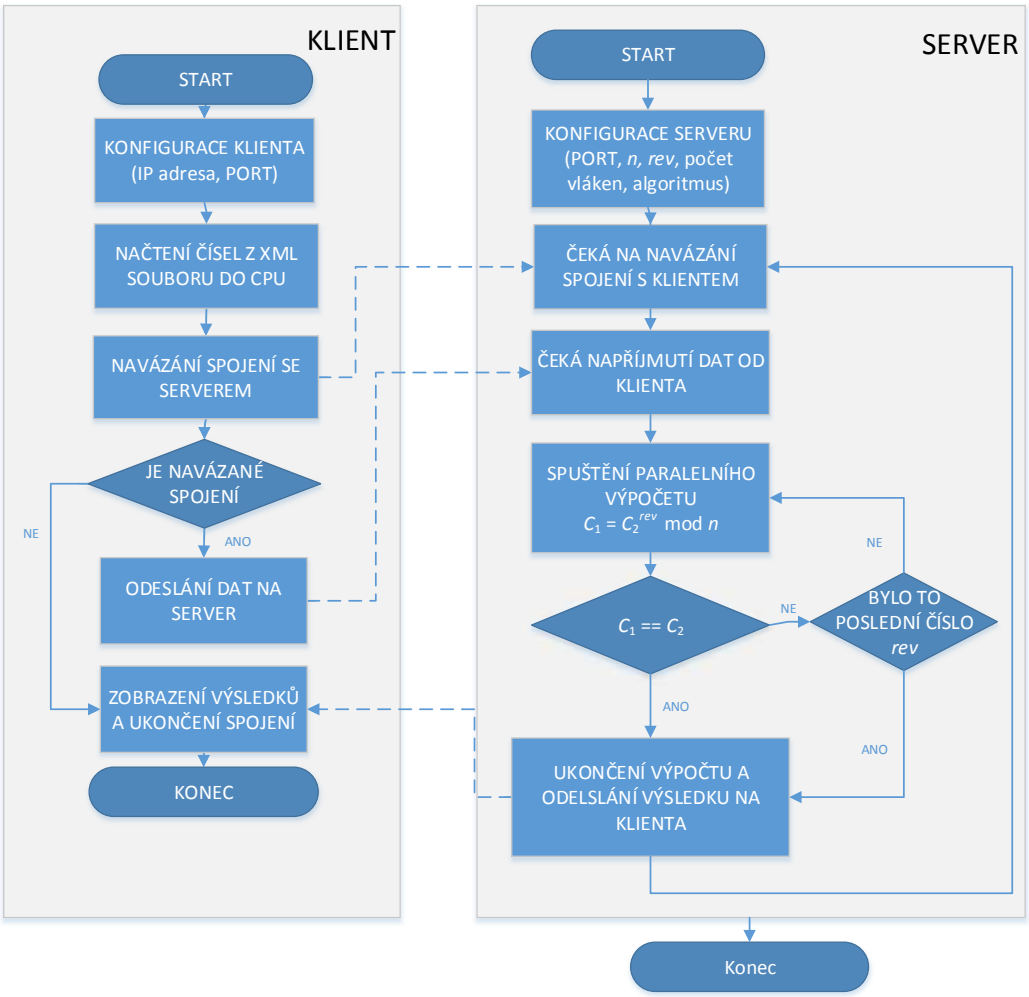
Pro implementaci aplikací „Klient – Server“ jsem si zvolil programovací jazyk C#, protože jsem chtěl vytvořit uživatelsky přívětivé aplikace s GUI a navíc .NET Framework obsahuje knihovny pro práci s velkými čísly a pro paralelní

programování. Revokační funkce protokolu HM12 ověřuje jestli c_1 se nikdy nerovná c_2 podle vzorce [27]:

$$c_1 \equiv c_2^{rev} \bmod n \tag{6.2}$$

,kde čísla *rev* a *n* jsou načítány s předem nadefinovaného místa v aplikaci Server a dvojici čísel c_1 a c_2 odešle uživatel do aplikace Server z aplikace Klient.

Pro lepší představu jak má aplikace pracovat jsem si nejdříve vytvořil vývojový diagram.



Obrázek 6: Vývojový diagram aplikace Klient – Server

6.6 Realizace aplikace Klient – Server

Realizaci aplikace lze rozdělit do několika kroků. V první části implementace požadované aplikace jsem se rozhodl vytvořit kostru obou programů. Vytvořil jsem dvě jednoduché aplikace, které se sebou komunikovali pomocí IP adresy a portu. K vytvoření těchto programů jsem použil knihovnu System.Net.

Vytvořené aplikace fungují na principu:

1. Uživatel v aplikaci „Server“ zadá číslo portu a klikne na tlačítko Start.
2. Aplikace „Server“ začne naslouchat na daném portu.
3. Uživatel v aplikaci „Klient“ zadá IP adresu a stejný Port jako u aplikace „Server“ a zvolí xml soubor, který chce na straně „Serveru“ zpracovat.
4. Klientská aplikace převede zvolený soubor včetně jeho názvu na pole bajtů, naváže spojení s aplikací „Server“. Pokud se spojení naváže, aplikace odešle pole bajtů a začne čekat na odpověď.
5. Aplikace „Server“ příjme pole bajtů, toto pole transformuje na původní soubor, který zpracuje, a odešle výsledek aplikaci „Klient“. Po odeslání výsledku „Server“ uzavře spojení s „Klientem“.
6. „Klient“ příjme výsledek a zobrazí jej. Po zobrazení výsledku „Klient“ uzavře navázané spojení.

V druhé části implementace programu jsem se zaměřil na realizaci výpočtů modulárního mocnění. Nejprve jsem zkusil naprogramovat několik svých algoritmů podle načerpaných znalostí z kapitoly 6 teoretického úvodu a nakonec jsem do projektu rozhodl přidat algoritmy vytvořené pomocí dostupných open source knihoven pro C#.

Algoritmus Simple Mod Exp - Byte Array version

Simple Mod Exp je první z vytvořených algoritmů. Algoritmus nejprve transformuje velká čísla do polí bajtů a tyto pole pak postupně zpracovává. Tato metoda využívá modulární násobení realizované pomocí jednoduchého sčítání polí bajtů a k redukci je použit algoritmus odčítání.

Algoritmus Square&Multiply Mod Exp - Byte Array version

Stejně jako předchozí algoritmus i tento nejprve převede číslo do pole bajtů. Mocnění v tomto výpočtu probíhá binárně zleva doprava s pomocí algoritmů násobení, druhé mocniny a jednoduché modulární redukce. Pro zrychlení násobení jsem použil Karatsubovu metodu. Jednoduchá modulární redukce vyhází z klasického dělení, ale na rozdíl od dělení šetří paměťové nároky, protože neukládá výsledek podílu. Výsledek podílu je zbytečný pro modulární mocnění.

Algoritmus Montgomery Mod Exp - Byte Array version

Poslední z vytvořených algoritmů, který transformuje číslo do pole bajtů, je Montgomery Exp Mod. Na začátku tohoto algoritmu se potřebné čísla transformují do Montgomeryho formy. Samotné mocnění probíhá zleva doprava a je realizované pomocí algoritmu Montgomeryho násobení. Na konci výpočtu se čísla převedou zpět z Montgomeryho formy. Pokud chceme využít tento algoritmus, musíme použít liché číslo pro modulo.

Algoritmus EmilGMP

Tento algoritmus je založený na metodě modulárního mocnění z knihovny od vývojáře Emila Stefanova. Emil Stefanov transformoval knihovnu GMP z jazyka C do programovacího jazyka C#. Jedná se o nejefektivnější algoritmus v mé aplikaci.

Algoritmus Mongomery Mod Exp - Bouncy Castle C#

Dalším algoritmem je Montgomeryho mocnění vytvořené pomocí funkcí z knihovny Bouncy Castle C#. Jako u předchozího Montgomeryho i tady se nejdříve čísla transformují do Montgomeryho formy. Pomocné čísla, která se transformují do Montgomeryho formy z modula, jsou transformovány ihned po načtení modula do paměti aplikace a ne až při zahájení výpočtu. Samostatné mocnění je realizováno formou zleva doprava. Na konci opět proběhne konverze výsledku z Montgomeryho formy.

Algoritmus Square&Multiply Mod Exp - Bouncy Castle C#

Squre&Multiply Mod Exp je upravený algoritmus z knihovny Bouncy Castle C#. Modulární mocnění probíhá zleva doprava a obsahuje funkce z knihovny Bouncy Castle C# pro druhou mocninu, násobení, klasické dělení.

Algoritmus Square&Multiply Mod Exp - Chew Keong Library

Algoritmus z knihovny vytvořené vývojářem Chew Keongem. Modulární mocnění v tomto algoritmu probíhá zleva doprava a používá Barrettovu redukci.

Algoritmus Fast Mod Exp - System.Numerics

Algoritmus je vytvořen pomocí knihovny System.Numerics, která je dostupná přímo v .NET Framework 4.5. Knihovna obsahuje datový typ BigInteger. Ukázka tohoto algoritmu viz pseudokód 10.

Pseudokód 10: Fast Mod Exp - System.Numerics

Vstup: operátor *a,* exponent *e,* modulo *m,* pomocná proměnná *temp*
Výstup: *R*
1. *temp* = 1
2. Když e > 0:
   2.1 Jestli *e* je liché, vykonej *temp* = (*temp* * *a*) % *m*
   2.2 *e* = *e* >> 1
   2.3 *a* = (*a* * *a*) % *m*
3. *R* = (*temp* % *m*)
4. Vrať (*R*)

Algoritmus Fast Mod Exp Rec - System.Numerics

Tento algoritmus je stejně jako předchozí realizován pomocí knihovny System.Numerics. Jedná se o rekurzivní metodu výpočtu modulárního mocnění (viz pseudokód 11). Více o této metodě rekurzivního výpočtu modulárního mocnění v [29].

Pseudokód 11: Fast Mod Exp Rec - System.Numerics

Vstup: operátor *a,* exponent *e,* modulo *m,* pomocné proměnné: *zero, one, two, temp*
Výstup: *R*

1. *zero* = 0, *one* = 1, *two* = *one* + *one*
2. Jestli *e* = 0 pak vrať (*R* =1)
3. Jestli *e* = 1 pak vrať (*R* = *a* % *m*)
4. Jestli (*e* % *two*) = 0 vykonej:
   2.1 *temp* = fastModExpRec(*a*, (*e* / *two*), *m*)
   2.2 *R* = (*temp* * (*temp*)) % (*m*)
   2.3 Vrať (*R*)
3. *R* = *a* * (fastModExpRec(*a*, (*e* - *one*), *m*)) % (*m*)
4. Vrať (*R*)

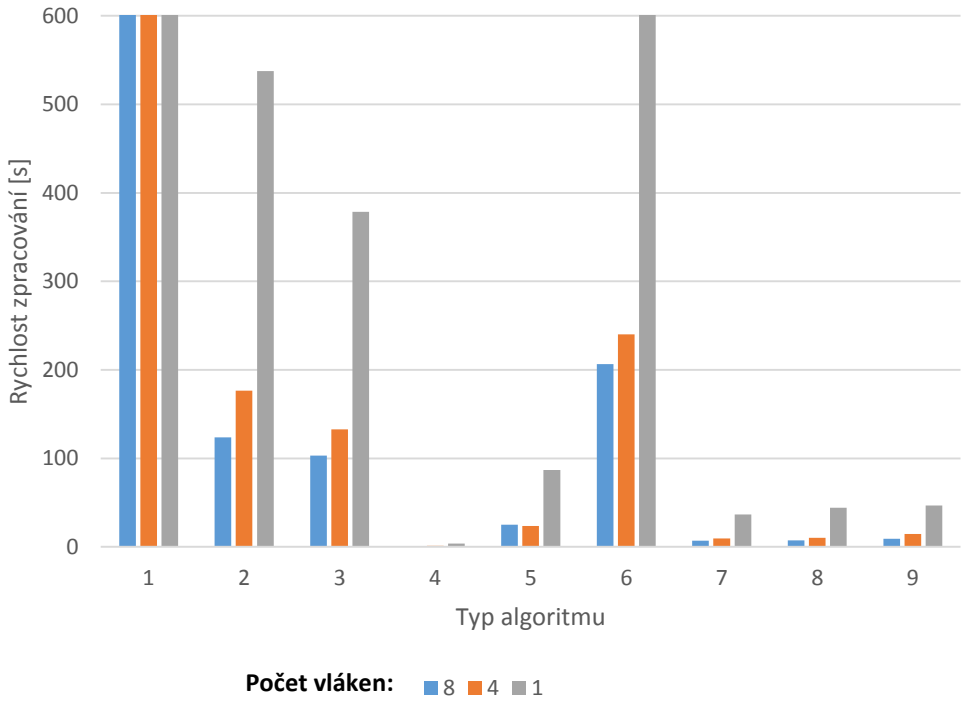
V třetí části implementace aplikací „Klient – Server“ jsem musel vyřešit problém paralelního zpracování dat. Pro paralelní zpracování dat jsem se rozhodl použít knihovnu System.Threading.Task, která je obsažena v .NET Framework 4.5. Uživatel při nastavování serverové aplikace si tak může zvolit počet vláken procesoru, které bude aplikace využívat. Paralelní výpočet je nastaven tím způsobem, že každé vlákno vypočítá rekurzivní funkci protokolu HM12 pro jeden exponent – to znamená, že v ideálním případě může 4-jádrový procesor, který má 8 vláken, ověřovat výpočet pro 8 exponentů zároveň.

6.6.1 Testování aplikace

Testování vytvořené aplikace probíhalo na počítačích s procesory Intel i7 2630 QM a Intel Xeon X3440 2.53 GHz a na operačních systémech Windows 7 a Windows 8.1. Nejprve jsem testoval rychlost jednotlivých algoritmů pro 1, 4 a 8 vláken tohoto procesoru. Při tomto měření byly použity čísla o velikosti 1024 bit a počet exponentů *rev* byl 1000. Výsledky měření můžeme vidět v tabulce 7 a grafu 7.

Tabulka 7: Porovnání rychlosti zpracování dat vytvořenou aplikací při postupném nastavování maximálního využití 1, 4, 8 vláken procesoru i7 2730 QM

Počet vláken	Simple Mod Exp - Byte Array version (1)	Square&Multiply Mod Exp - Byte Array version (2)	Montgomery - Byte Array version (3)	Mod Exp - EmilGMP (4)	Mongomery Mod Exp - Bouncy Castle C# (5)	Square&Multiply Mod Exp - Bouncy Castle C# (6)	Square&Multiply Mod Exp - Chew Keong Library (7)	Fast Mod Exp - System.Numerics (8)	Fast Mod Exp Rec - System.Numerics (9)
8	876,572	123,79	103,29	0,949	25,144	206,356	7,082	7,324	9,221
4	1321,881	176,69	132,90	1,05	23,471	240,198	9,616	10,208	14,705
1	4127,252	537,61	378,60	3,662	86,795	750,728	36,667	44,221	46,831



Graf 7: Porovnání rychlosti zpracování dat vytvořenou aplikací při postupném nastavování maximálního využití 1, 4, 8 vláken procesoru i7 2730 QM

Tabulka 8: Porovnání rychlosti zpracování dat vytvořenou aplikací při postupném nastavování max využití 1, 4, 8 vláken na procesoru Intel Xeon X3440 2.53 GHz

Počet vláken	Simple Mod Exp - Byte Array version (1)	Square&Multiply Mod Exp - Byte Array version (2)	Montgomery - Byte Array version (3)	Mod Exp - EmilGMP (4)	Mongomery Mod Exp - Bouncy Castle C# (5)	Square&Multiply Mod Exp - Bouncy Castle C# (6)	Square&Multiply Mod Exp - Chew Keong Library (7)	Fast Mod Exp - System.Numerics (8)	Fast Mod Exp Rec - System.Numerics (9)
8	614,61	91,953	88,922	1,084	19,919	143,342	9,69	8,883	10,947
4	928,92	124,433	97,368	1,301	22,48	169,459	10,554	11,95	13,451
1	3488,6	469,927	362,87	4,745	76,893	633,143	39,964	45,097	45,97

Z měření lze vidět, že čím více vláken do výpočtu zapojujeme, tím je výpočet rychlejší. Bohužel pokud zapojíme do výpočtu 8 vláken, není rychlost zpracování 8x rychlejší než zpracování výpočtu pouze jedním vláknem, je to dáno vlastnostmi knihovny System.Threading. Z tabulek a grafu můžeme vidět, že nejlepších výsledků dosahuje výpočet pomocí knihovny EmilGMP, naopak nejpomalejší jsou algoritmy založené na rozkladu velkých čísel do pole bajtů.

Testování nad daty používanými pro protokol HM12

Druhá část testování probíhala nad vygenerovanými daty pro schéma protokolu HM12. V tabulce 9 jsou uvedeny velikosti čísel pro jednotlivé varianty měření. Varianty 1 a 2 jsou určeny přímo pro schéma protokolu HM12 popsaného v [27]. Varianta číslo 3 je určena pro inovovanou verzi protokolu HM12.

Tabulka 9: Varianty velikosti čísel pro protokol HM12:

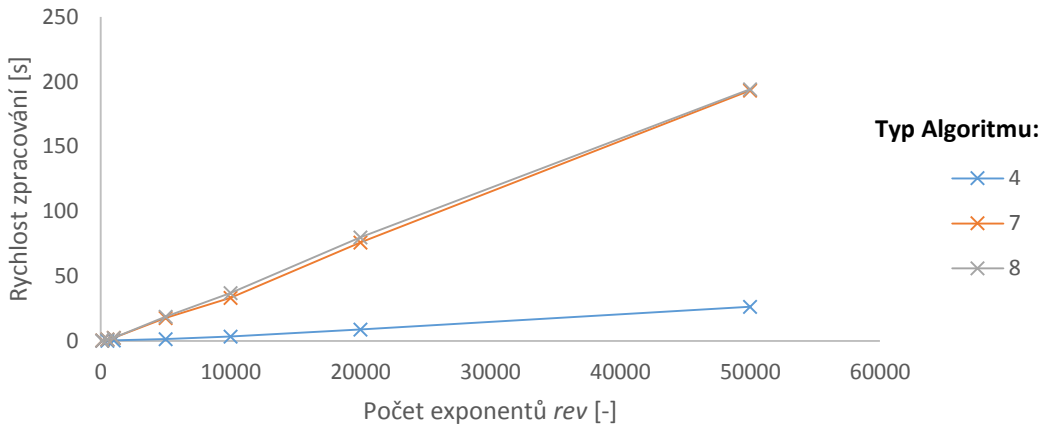
Varianta měření	Velikost c_1	Velikost c_2	*Velikost n	Velikost rev
1	1023	1024	1024	322
2	1391	1390	1392	483
3	1392 (menší než n)	1389	1392	1391

Čísla n byly generovány jako $n = r \cdot r \cdot s$, kde r a s jsou prvočísla

Varianta 1:

Tabulka 10: Výsledky měření varianty 1

Počet exponentů *rev* v souboru	Simple Mod Exp - Byte Array version (1)	Square&Multiply Mod Exp - Byte Array version (2)	Montgomery - Byte Array version (3)	Mod Exp - EmilGMP (4)	Mongomery Mod Exp - Bouncy Castle C# (5)	Square&Multiply Mod Exp - Bouncy Castle C# (6)	Square&Multiply Mod Exp - Chew Keong Library (7)	Fast Mod Exp - System.Numerics (8)	Fast Mod Exp Rec - System.Numerics (9)
100	20,67	2,931	2,287	0,058	1,396	4,937	0,303	0,289	0,318
500	124,8	14,417	10,495	0,188	6,673	33,332	1,153	1,163	1,293
1000	289,3	45,752	30,953	0,332	18,609	68,342	2,311	2,311	2,576
5000	-	226,57	180,23	1,494	108,55	374,738	17,741	18,664	18,91
10000	-	-	-	3,353	209,32	-	33,172	36,982	42,504
20000	-	-	-	8,711	-	-	75,93	79,902	89,114
50000	-	-	-	26,313	-	-	193,31	194,30	224,08

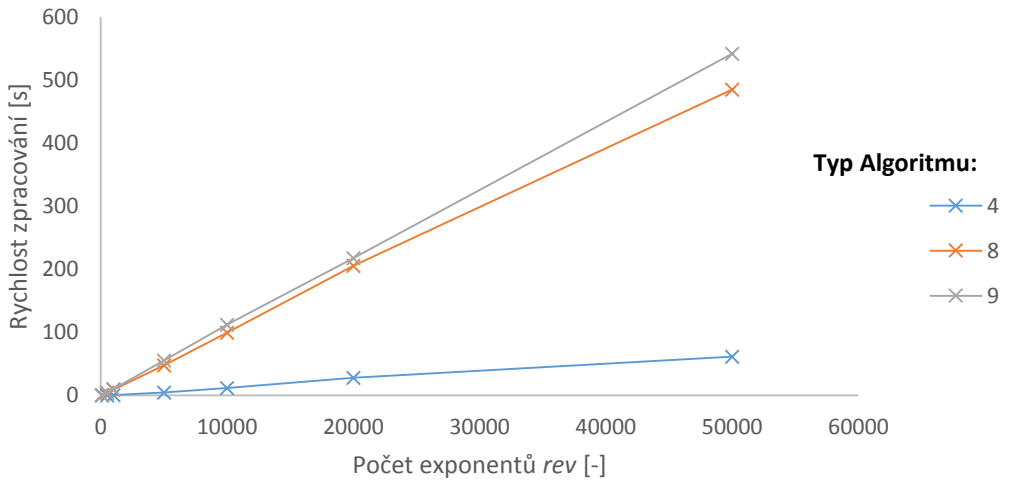


Graf 8:Varianta 1 – Rychlost zpracování revokační funkce protokolu HM12

Varianta 2:

Tabulka 11: Výsledky měření varianty 2

Počet exponentů *rev* v souboru	Mod Exp - EmilGMP (4)	Mongomery Mod Exp - Bouncy Castle C# (5)	Square&Multiply Mod Exp - Bouncy Castle C# (6)	Fast Mod Exp - System.Numerics (8)	Fast Mod Exp Rec - System.Numerics (9)
100	0,134	3,182	16,759	0,719	0,805
500	0,466	21,383	89,052	4,235	4,619
1000	0,834	43,935	186,783	9,269	10,361
5000	4,983	243,044	941,053	47,714	55,618
10000	11,92	475,975	-	99,677	112,31
20000	27,916	-	-	205,572	217,783
50000	61,689	-	-	485,229	541,973

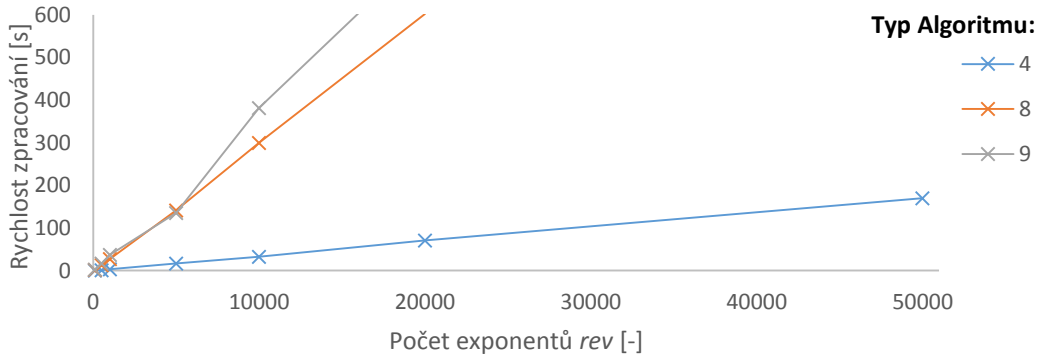


Graf 9: Varianta 2 – Rychlost zpracování revokační funkce protokolu HM12

Varianta 3:

Tabulka 12: Výsledky měření varianty 3

Počet exponentů *rev* v souboru	Mod Exp - EmilGMP (4)	Mongomery Mod Exp - Bouncy Castle C# (5)	Square&Multiply Mod Exp - Bouncy Castle C# (6)	Fast Mod Exp - System.Numerics (8)	Fast Mod Exp Rec - System.Numerics (9)
100	0,301	4,223	36,401	2,026	2,549
500	1,124	21,946	232,04	13,345	16,936
1000	3,235	62,873	453,991	27,149	37,25
5000	16,993	299,504	2510,49	141,65	135,236
10000	32,314	-	-	299,88	381,327
20000	70,497	-	-	602,441	752,254
50000	170,12	-	-		



Graf 10: Varianta 3 – Rychlost zpracování revokační funkce protokolu HM12

Testování probíhalo na počítači s procesorem Intel i7 2630 QM při využití všech 8 vláken. Nejlepších výsledků opět dosáhl algoritmus vytvořený pomocí knihovny EmilGMP, tento algoritmus si bez problémů poradil i s různou bitovou velikostí čísel vzorce 6.2. Při měření druhé a třetí varianty se objevil problém u algoritmů

rozkládající čísla na pole bajtů a u algoritmu vytvořeného pomocí knihovny „Chew Keong“. První tři algoritmy využívající rozložení čísla do pole bajtů byly vytvořeny s podmínkou ze vzorce 6.2 – číslo c_1 musí mít stejnou nebo větší bitovou velikost než číslo n. Pokud není tato podmínka splněna tak se při zpracování dat objeví výjimka.

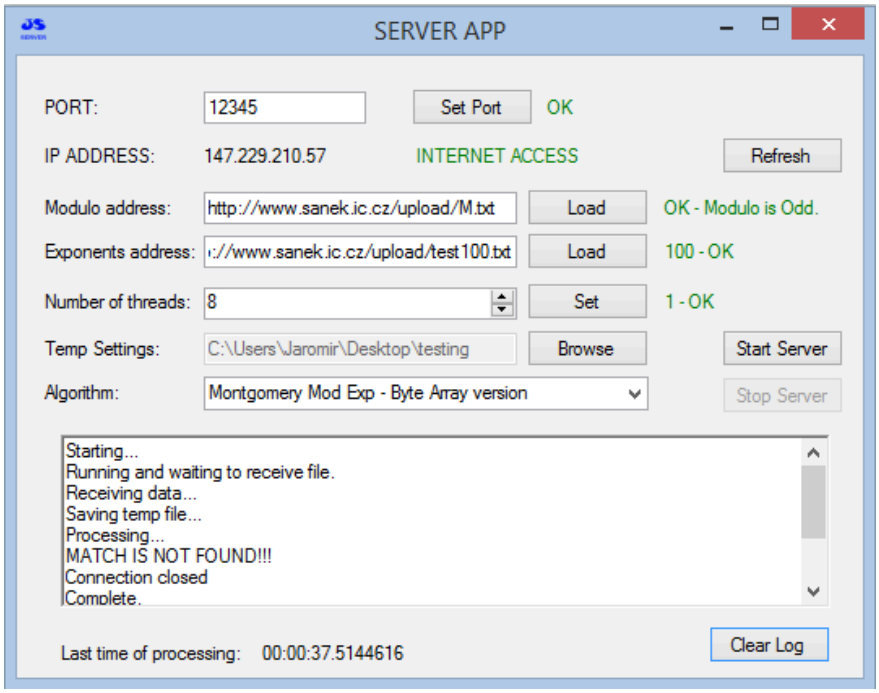
6.7 Práce s programem

Tato podkapitola popisuje ovládání a konfigurování vytvořených aplikací. Programy vyžadují nainstalovaný .NET Framework 4.5. Programy byly vyzkoušeny na systémech Windows 7 64bit a Windows 8 64bit. Příloha diplomové práce obsahuje balíčky *.msi pro nainstalování aplikací Klient a Server. Po nainstalování aplikací se vytvoří zástupci pro spuštění na ploše a v nabídce start.

Pro správné fungování programů je potřebné vytvořit výjimky na firewallu.

6.7.1 Návod pro práci s aplikací Server

Po spuštění aplikace Server se uživateli zobrazí grafické rozhraní aplikace (viz obrázek 7), které musí správně nakonfigurovat.



Obrázek 7: Aplikace Server

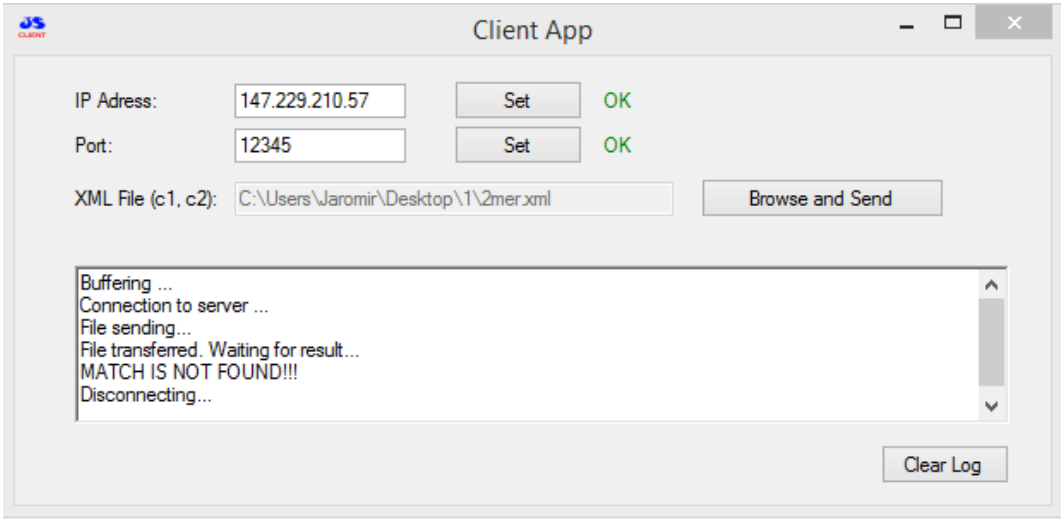
Uživatel musí vyplnit tyto parametry:

- Port – celé kladné číslo.
- Modulo address – adresa, kde je uložen soubor s modulem. Soubor musí být ve formátu *.txt a číslo v něm musí být v desítkové soustavě bez mezer, čárek, …
- Exponents address – adresa, kde je uložen soubor s exponenty *rev*. Soubor musí být opět ve formátu *.txt a čísla v něm musí být v desítkové soustavě bez mezer, čárek, … Jako oddělovač čísel se v souboru používá zalomení řádku (klávesa enter).
- Number of threads – zde uživatel nastaví počet vláken procesoru, které bude aplikace při výpočtu využívat.
- Temp Settings – uživatel definuje složku, do které se dočasně uloží přijatý soubor od Klienta. Po dokončení výpočtu se přijatý soubor automaticky odstraní.

- Algorithm – uživatel si zvolí typ algoritmu, jaký bude aplikace používat pro výpočet.
- Refresh – toto tlačítko aktualizuje načtenou IP adresu a ověří připojení k internetu.
- Start Server – po kliknutí na toto tlačítko aplikace začne čekat na připojení Klienta.
- Stop Server – toto tlačítko přeruší naslouchání aplikace. Klient se nebude moci připojit.
- Log – zobrazení výsledků a stavů aplikace. Aplikace může zobrazit tyto výsledky:
    - MATCH IS FOUND!!! – při výpočtu byla nalezena shoda.
    - MATCH IS NOT FOUND!!! – při výpočtu nebyla nalezena shoda.
    - MODULO ERROR!!! – chyba související s modulem.
    - DATA ERROR!!! – zvolený algoritmus nebyl schopen zpracovat zadané parametry.
- Clear Log – po kliknutí na tlačítko se vymaže log.
- Last time of processing – údaj udává čas posledního bezchybného výpočtu.

6.7.2 Návod pro práci s aplikací Klient

Po spuštění aplikace se opět zobrazí grafické rozhraní (viz obrázek 8).



Obrázek 8: Aplikace Klient

Popis aplikace Klient:

- IP Address – uživatel musí zadat IP adresu počítače na kterém je spuštěná aplikace Server.
- Port – uživatel musí zadat stejné číslo portu jaké je nastavené v aplikaci Server.
- XML File – uživatel musí vybrat soubor *.xml, který obsahuje čísla parametry c_1 a c_2 protokolu HM12, kde $c_1 = c_2^{rev} \bmod n$. Xml soubor musí být v tomto tvaru:

```
<?xml version="1.0" encoding="utf-8"?>
<modular>
<c1>8888546564123</c1>
<c2>4567898445646</c2>
</modular>
```

- Log – zobrazení výsledků a stavů aplikace. Aplikace může zobrazit tyto výsledky:
  - MATCH IS FOUND!!! – při výpočtu byla nalezena shoda.
  - MATCH IS NOT FOUND!!! – při výpočtu nebyla nalezena shoda.

    - MODULO ERROR!!! – chyba související s modulem.
    - DATA ERROR!!! – zvolený algoritmus nebyl schopen zpracovat zadané parametry.
- Clear Log – po kliknutí na tlačítko se vymaže log.

6.8 Zhodnocení vytvořené aplikace

Výstupem této práce jsou v jazyce C# vytvořené dvě aplikace „Klient – Server“ na výpočet revokační funkce protokolu HM12. Obě aplikace jsou funkční a připraveny na další testování popřípadě na rozšíření. Testování aplikací probíhalo na dvou procesorech značky Intel a na operačních systémech Windows 7 a Windows 8.1. Aplikace „Server“ pro výpočet modulárního mocnění revokační funkce protokolu HM12 obsahuje celkem 9 algoritmů, z nichž některé jsou založeny na vlastních funkcích a ostatní na existujících open source knihovnách. Počet těchto algoritmů je snadno rozšiřitelný. Pokud chceme používat algoritmy založené na Montgomeryho metodě, musí být hodnota modula lichá. Po načtení modula do aplikace Server se ihned vytvoří pomocné proměnné, které jsou potřebné pro Montgomeryho metody.

Z naměřených výsledků je patrné, že algoritmy založené na vlastních funkcích jsou pomalejší než použité algoritmy z existujících open source knihoven pro velká čísla. Z algoritmů založených na vlastních funkcích v testech nejlépe dopadl algoritmus založený na technice Montgomeryho mocnění.

7 Závěr

První část této práce byla zaměřena na zanalyzování výkonu dostupných knihoven pro CPU obsahujících funkci pro modulární mocnění. Po realizaci a otestování vytvořených aplikací jsem zjistil, že nejlepších výsledků, které můžeme najít v grafech 1-4, dosahují knihovny GMP a MPIR.

U knihovny LibTomMath jsem byl překvapen, že modulární mocnění pro 512 bitová čísla je pomalejší než pro 1024 bitová čísla. Tato situace byla způsobena tím, že u 512 bitových čísel LibTomMath využíval při modulárním mocnění Barrettovu redukci a u 1024 čísel použil Montgomeryho redukci.

V druhé části jsem se zaměřil na zanalyzování technologie CUDA a jejího možného použití při modulárním mocnění velkých čísel, které má uplatnění v asymetrických kryptosystémech při šifrování a dešifrování zpráv.

V projektu jsem přepracoval knihovny pro modulární mocnění primárně určené pro CPU k použití na GPU. Bohužel výsledky jsou značně negativní, protože aby bylo možné knihovny efektivně implementovat tak by se musely kompletně celé přeprogramovat. Vytvoření nových knihoven pro CUDA nebo kompletní přeprogramování stávajících knihoven z CPU na CUDA nebylo cílem této práce.

Při vývoji aplikací na platformě CUDA jsem se setkal s nespočtem pádů systémů „modrých smrtí“, které byly způsobeny kompatibilitou nových ovladačů od firmy NVIDIA a operačním systémem MS Windows 8.1.

V průběhu realizace projektu jsem se také snažil kontaktovat několik vývojářů, kteří se v minulosti problémem modulárního mocnění na CUDA už zabývali, bohužel ani jeden z nich mi neodpověděl.

Pro vytvoření aplikace „Klient – Server“ na výpočet revokační funkce protokolu HM12 jsem si zvolil platformu CPU, protože na platformě GPU se mi nepodařilo dosáhnout efektivních výsledků s dostupnými knihovnami a vývoj vlastní knihovny pro výpočty modulární aritmetiky pro GPU nebyl obsahem této diplomové práce.

Vytvořené aplikace můžeme vidět na obrázcích 7 a 8 a výsledky testování této aplikace nalezneme v kapitole 6.6.1.

Použitá literatura

[1] WU, Hao. *Implementation of public key algorithms in CUDA*. Gjøvik, 2010. Dostupné z:http://brage.bibsys.no/hig/bitstream/URN:NBN:no-bibsys_brage_15995/1/Master Thesis_ Hao Wu_080338.pdf. Master's Thesis. Gjøvik University College.

[2] SZERWINSKI, Robert. *Efficient Cryptography on Graphics Hardware*. Bochum, 2008. Dostupné z: http://www.emsec.rub.de/media/crypto/attachments/files/2010/04/da_szerwinski.pdf. Diploma Thesis. Ruhr-University Bochum. Vedoucí práce Prof. Dr.-Ing. Christof Paar.

[3] STUHL, Miroslav. *Computing Strongly Connected Components with CUDA*. Brno, 2013. Dostupné z: http://is.muni.cz/th/207551/fi_m/dp.pdf. DIPLOMA THESIS. Masarykova Univerzita.

[4] ČERMÁK, Marek. *Programovatelné grafické procesory a jejich aplikace v kryptografii*. Brno, 2011. Dostupné z: http://is.muni.cz/th/207496/fi_m/DPv1.0.pdf. Diplomová práce. Masarykova Univerzita.

[5] LOU, Xin. *Acceleration of Distance-to-Default with GPU*. Stockholm, 2012. Dostupné z: http://www.diva-portal.org/smash/get/diva2:557179/FULLTEXT01.pdf. Master's Thesis. Royal Institute of Technology.

[6] BOBROV, Maksim. *Cryptographic Algorithm Acceleration Using CUDA Enabled: GPUs in Typical System Configurations*. Rochester, 2010. Dostupné z: https://ritdml.rit.edu/bitstream/handle/1850/12952/MBobrovThesis8-2010.pdf?sequence=1. Master's Thesis. Rochester Institute of Technology.

[7] SANDERS a Edward KANDROT. *CUDA by Example*. Boston: Addison Wesley, 2010. ISBN 978-0131387683.

[8] CUDA C PROGRAMMING GUIDE. [online]. [cit. 2013-11-03]. Dostupné z: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[9] FURST, Jiří. *Úvod do OpenMP* [online]. [cit. 2013-12-15]. Dostupné z: www.civ.cvut.cz/ESF/info/run1.php?did=44

[10] FELT ČVUT. *Paralelní programování* [online]. 2009 [cit. 2013-12-15]. Dostupné z: lynx1.felk.cvut.cz/pte/files/2009/PTE-07.pdf

[11] ŠÍSTEK, Jakub. *Úvod do paralelního programování 2* [online]. 2007. vyd. [cit. 2013-12-15]. Dostupné z: www.civ.cvut.cz/ESF/info/run1.php?did=69

[12] ZAHRAN, Mohamed. *Graphics Processing Units (GPUs): Architecture and Programming: Lecture 5: CUDA Threads*[online]. 2011 [cit. 2013-12-15]. Dostupné z: http://www.mzahran.com

[13] *OPEN CL* [online]. 2013 [cit. 2013-12-15]. Dostupné z: http://www.khronos.org/opencl/

[14] GOPAL, Vinodh, James GUILFORD. INTEL CORPORATION. *Fast and Constant-Time Implementation of Modular Exponentiation* [online]. USA [cit. 2013-12-15]. Dostupné z: https://www.cse.buffalo.edu/srds2009/escs2009_submission_Gopal.pdf

[15] GRANLUND, Torbjorn. *The GNU Multiple Precision Arithmetic Library* [online]. 2013 [cit. 2013-12-15]. 5.1.3. Dostupné z: https://gmplib.org/gmp-man-5.1.3.pdf

[16] RIZZO, Ottavio. *On the complexity of the 2k-ary and of the sliding window algorithms for fast exponentiation* [online]. 2004 [cit. 2013-12-12]. Dostupné z: http://air.unimi.it/bitstream/2434/10853/3/19.pdf

[17] RASMUSSEN, Mads a Greg ROSE. *Multi–Precision Math* [online]. 2010 [cit. 2013-12-15]. Dostupné z: http://libtom.org/files/ltm-0.42.0.zip

[18] SELVEK, Jiří. *Využití GPU při výpočtech*. Opava, 2007. Dostupné z: http://students.math.slu.cz/JiriSelvek/Langer/Dokumentace_GPU_vypocty.doc. SLEZSKÁ UNIVERZITA.

[19] Performance Measurements of other Multi-Precision Libraries. *The Glasgow Haskell Compiler* [online]. 2008 [cit. 2013-12-23]. Dostupné z: https://ghc.haskell.org/trac/ghc/wiki/ReplacingGMPNotes/PerformanceMeasurements

[20] *MPIR: The Multiple Precision Integers and Rationals Library* [online]. 2012 [cit. 2013-12-26]. Dostupné z: http://www.mpir.org/mpir-2.6.0.pdf

[21] BN. *OPENSSL* [online]. [cit. 2013-12-26]. Dostupné z: https://www.openssl.org/docs/crypto/bn.html#DESCRIPTION

[22] KUBANDA, Karol. *Modulárna aritmetika pre kryptografické výpočty v čipových kartách*. Brno, 2010. Diplomová práce. Masarykova univerzita.

[23] *MSI* [online]. 2012 [cit. 2013-12-27]. Dostupné z: http://www.msi.com/product/vga/N650Ti-PE-1GD5.html#/?div=Specification

[24] *INTEL* [online]. 2012 [cit. 2013-12-27]. Dostupné z: http://ark.intel.com/products/42928/Intel-Xeon-Processor-X3440-8M-Cache-2_53-GHz

[25] PUŠ, Petr. Poznáváme C# a Microsoft. NET 52. díl – ThreadPool. Dostupné z: http://www.zive.cz/clanky/poznavame-c-a-microsoft-net-52-dil--threadpool/sc-3-a-128106/default.aspx

[26] ČERMÁK, Jakub. TASK PARALLEL LIBRARY. Dostupné z: http://www.dotnetportal.cz/clanek/130/Parallel-Extensions-for-NET

[27] HAJNÝ, Jan a Lukáš MALINA. Unlinkable Attribute-Based Credentials with Practical Revocation on Smart-Cards. Dostupné z:http://cardis.iaik.tugraz.at/proceedings/cardis_2012/CARDIS2012_5.pdf

[28] MENEZES, Alfred a Scott VANSTONE. *HANDBOOK of APPLIED CRYPTOGRAPHY*. 5. vyd. 1996. ISBN 0849385237. Dostupné z: http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.99.2838&rep=rep1&type=pdf

[29] *Recursion: Fast Exponentiation*. Dostupné z: www.cs.ucf.edu/~dmarino/ucf/cop3502/lec/RecursionFastExp.doc

Příloha A – Seznam zkratek

BN – Big Number

CPU – Central Processing Unit

CUDA – Compute Unified Device Architecture

CUMP – CUDA Multiple Precision Arithmetic Library

GCC – GNU Compiler Collection

GCD – Největší společný dělitel (Greatest common divisor)

GMP – GNU Multiple Precision Arithmetic Library

GPGPU – General-Purpose Computing on Graphics Processing Units

GPU – Graphic Processing Unit

IDE – Integrated Development Environment

LSB – Least Significant Bit

MPI – Message Passing Interface

MPIR – Multiple Precision Integers and Rationals

MSB – Most Significant Bit

NC – NVIDIA CUDA

NVCC – NVIDIA's CUDA Compiler

OpenCL – Open Computing Language

OpenMP – Open Multi-Processing

SPMD – Single Multiple Data

SIMD – Single Instruction Multiple Data

SSL – Transport Layer Security

TLS – Secure Sockets Layer

T&L – Transform & Lighting

Příloha B – Tabulky měření výkonů knihoven

Tabulka 13: Srovnání rychlosti funkce modulárního mocnění v daných knihovnách pro soubory obsahující 10000 čísel

Velikost čísel [bit]	256			512			1024			2048		
Knihovna	**MPIR**	**LTM**	**GMP**	**MPIR**	**LTM**	**GMP**	**MPIR**	**LTM**	**GMP**	**MPIR**	**LTM**	**GMP**
MIN [ms]	57	325	49	503	11350	259	1894	12253	1831	18683	118758	18953
MAX [ms]	67	348	66	550	17950	306	3468	13763	2876	26817	188601	23363
AVG [ms]	59,3	334,7	54,3	519,1	15956,2	274	2789	12876,9	2627,3	24741,8	146925,2	21069,8

Tabulka 14: Srovnání celkové rychlosti daných knihoven při použití funkce modulárního mocnění pro soubory obsahující 10000 čísel

Velikost čísel [bit]	256				512			
Knihovna	**MPIR**	**LTM**	**GMP**	**OpenSSL**	**MPIR**	**LTM**	**GMP**	**OpenSSL**
MIN [ms]	452	466	248	93	879	11443	459	949
MAX [ms]	579	561	566	107	978	18041	768	1811
AVG [ms]	496,2	480,9	461,6	98,5	914,8	16058,1	595	1316,3
Velikost čísel [bit]	**1024**				**2048**			
Knihovna	**MPIR**	**LTM**	**GMP**	**OpenSSL**	**MPIR**	**LTM**	**GMP**	**OpenSSL**
MIN [ms]	2357	9285	2281	2175	19329	118934	19355	70033
MAX [ms]	4650	13947	3384	2915	28299	122265	23813	132547
AVG [ms]	3393,8	12705,4	3031,7	2734,8	25504,4	119637,2	21399,5	112499

Tabulka 15: Srovnání rychlosti funkce modulárního mocnění v daných knihovnách pro soubory obsahující 1000000 čísel

Velikost čísel [bit]	**256**			**512**		
Knihovna	**MPIR**	**LTM**	**GMP**	**MPIR**	**LTM**	**GMP**
MIN [ms]	9147	51945	6395	82066	1819491	42105
MAX [ms]	10600	59779	7437	86097	1853793	43982
AVG [ms]	9556,9	54742	6645,4	84412,9	1834571	43326,9
Velikost čísel [bit]	**1024**			**2048**		
Knihovna	**MPIR**	**LTM**	**GMP**	**MPIR**	**LTM**	**GMP**
MIN [ms]	313554	1217817	353809	2602800	12070353	2172058
MAX [ms]	314675	1308312	373648	2698483	12266719	2202590
AVG [ms]	314042,7	1271135,8	363993,9	2645753	12167422	2188102

Tabulka 16: Srovnání celkové rychlosti daných knihoven při použití funkce modulárního mocnění pro soubory obsahující 1000000 čísel

Velikost čísel [bit]	**256**				**512**			
Knihovna	**MPIR**	**LTM**	**GMP**	**OpenSSL**	**MPIR**	**LTM**	**GMP**	**OpenSSL**
MIN [ms]	14752	55193	7220	17683	95746	1826754	44019	283458
MAX [ms]	17617	61936	8309	30605	99885	1860591	45571	310624
AVG [ms]	15998,7	59204,1	7541,6	23483,4	98004,2	1847123	44811,1	301100,9
Velikost čísel [bit]	**1024**				**2048**			
Knihovna	**MPIR**	**LTM**	**GMP**	**OpenSSL**	**MPIR**	**LTM**	**GMP**	**OpenSSL**
MIN [ms]	331941	1234909	351589	558452	2632483	12111030	2176892	7403521
MAX [ms]	333817	1326365	376896	615065	2728513	12314218	2207418	7425835
AVG [ms]	332736,1	1290899,2	363600,4	583419,1	2676630	12230374	2191974	7416664

Tabulka 17: Porovnání rychlosti upravené knihovny LibTomMath na GPU (NVIDIA GeForce GT555M) a CPU (1 vlákno – i7 2630QM) - testování probíhalo na souborech obsahující 256 bit čísla

Počet čísel v souboru	**500**			**2000**		
	MIN	**MAX**	**AVG**	**MIN**	**MAX**	**AVG**
CPU [ms]	74	81	76,4	321	414	375,6
GPU <<<256,515>>> [ms]	2841	3870	3104,9	9423	13049	10717,7
GPU <<<256,256>>> [ms]	2286	2795	2551,5	7821	9921	9045,67
GPU <<<256,128>>> [ms]	1779	3127	2719,2	8351	14932	12125,2
Počet čísel v souboru	**5000**			**8000**		
	MIN	**MAX**	**AVG**	**MIN**	**MAX**	**AVG**
CPU [ms]	795	870	839,8	1186	2263	1672,7
GPU <<<256,515>>> [ms]	21722	36721	29124,6	26224	44709	34276
GPU <<<256,256>>> [ms]	23489	31338	26480,83	27613	44282	33472,33
GPU <<<256,128>>> [ms]	22388	42353	31562,6	29638	56443	45496,8

*Pozn. Čísla v závorkách značí nastavení kernelu CUDA

Tabulka 18: Porovnání rychlosti upravené knihovny LibTomMath na GPU (NVIDIA GeForce GT555M a MSI GeForce GTX 650TI PE) a CPU (Intel i7 2630QM a Intel Xeon X3440 2.53 GHz) - testování probíhalo na souborech obsahující 256 bit čísla

Počet čísel v souboru	**500**			**2000**		
	MIN	**MAX**	**AVG**	**MIN**	**MAX**	**AVG**
CPU – i7 [ms]	27	54	35	106	149	117
CPU – Xeon [ms]	33	109	54,76	124	220	175
GPU – 555M [ms]	2286	2795	2551,5	7821	9921	9045,67
GPU – 650TI [ms]	3230	12078	6197,36	800	9510	2905,82
Počet čísel v souboru	**5000**			**8000**		
	MIN	**MAX**	**AVG**	**MIN**	**MAX**	**AVG**
CPU – i7 [ms]	261	328	283,5	402	435	422,83
CPU – Xeon [ms]	327	375	356,78	454	498	479,64
GPU – 555M [ms]	23489	31338	26480,83	27613	44282	33472,33
GPU – 650TI [ms]	85845	20831	12871,1	4235	34860	14089,91

*Pozn. Na GPU bylo nastavení kernelu CUDA <<<256,256>>>

Příloha C – Obsah CD

Struktura kořenového adresáře přiloženého CD:

- **Zdrojové kódy** – složka obsahuje zdrojové kódy k vytvořeným aplikacím „Klient – Server“
- **Install** – složka obsahuje instalační soubory pro nainstalování aplikací „Klient – Server“
- **TestData** – testovací data na kterých byly aplikace testovány
- **xsanek00.pdf** – elektronická verze diplomové práce