Semestrální projekt MI-PAP.16 LS2019/2020:

GPU Cryptohash Recovery

(Pavlína Kopecká), Libor Kuchař

magisterské studium, FIT CVUT, Thákurova 9, 160 00 Praha 6

Datum

Definice problému a popis sekvenčního algoritmu:

Popis problému:

Mnoho aplikací (zejména těch webových) si musí nějakým způsobem ukládat hesla svých uživatelů. V současné době stále existují weby, které ukládají hesla jako plaintext, toto řešení je ale nebezpečné, protože pokud se útočník nějakým způsobem dostane k databázi hesel, tak mu nic nebrání se k původnímu heslu dostat. Proto je doporučeno hesla zahashovat pomocí silné kryptografické hashovací funkce a následně tuto hodnotu uložit.

Bezpečná hashovací funkce by měla splňovat následující tři body:
(https://cs.wikipedia.org/wiki/Kryptografická_hašovací_funkce)

1. **Odolnost vůči získání předlohy** – hashovací funkce je jednosměrná.
2. **Odolnost vůči získání jiné předlohy** – hashovací funkce má omezený obor hodnot, existuje tedy více předloh, který mají stejný výsledný hash.
3. **Odolnost vůči nalezení kolize:** Je obtížné systematicky najít dvojici vstupů (x,y), pro které h(x)=h(y).

Jak je patrné z 1. bodu, nelze z hashe systematicky získat původní hodnotu. Jediný způsob, jak získat původní hodnotu je nějakým způsobem „odhadnout“ tuto hodnotu a zahashovat jí pomocí použité kryptografické funkce a tyto hodnoty následně porovnat. Jakým způsobem ale původní heslo odhadnout?

1. **Útok hrubou silou** – jediné 100% spolehlivé řešení, vyzkouší se všechny možné kombinace hesel v dané délce. Počet kombinací je ale a^L kde a je délka abecedy (malá + velká písmena a čísla je 62) a L je délka hesla. Od určité délky není časově možné heslo prolomit.
2. **Slovníkový útok** – používají se hesla ze slovníku. Tento typ útoku má ten problém, že pokrývá pouze malé množství všech možných hesel, na druhou stranu velká část uživatelů používá velmi slabá hesla, na která tento typ útoku můžeš postačovat. V dnešní době existují už komplexní slovníky obsahující nejčastější hesla.
3. **Rozšířený slovníkový útok** – tato metoda využívá jako základ slovníkový útok, ale slova z tohoto slovníku nějakým způsobem upravuje (např. zaměňuje velikost písmen, přehazuje podobná písmena, přidává před/za/do slova řetězce různé délky atd...)
4. **Duhové tabulky**

Všechny zmíněné útoky lze provádět pomocí CPU (např. slovníkový útok nemá cenu vůbec provádět na GPU z důvodu, že bottleneck je stejně čtení z disku), v případě delších komplexnějších hesel ale

CPU již selhává. Tato práce se tedy zabývá nástrojem na získávání hesel z MD5 hashů za použití CUDA, konkrétně se bude jednat o **útok hrubou silou**, a **rozšířený slovníkový útok.**

Popis sekvenčního řešení:

Hashovací funkce MD5 byla převzata a mírně upravena z https://gist.github.com/creationix/4710780.

Útok hrubou silou:

Největším problémem tohoto útoku bylo samotné generování všech řetězců dané délky. Původně tento typ útoku byl řešen rekurzí, kvůli přenositelnosti na GPU bylo nutné tento algoritmus upravit na iterativní.

Iterativní algoritmus funguje následovně:

1. K řetězci délky L, který obsahuje znaky z abecedy délky A se budeme chovat jako k číslu o počtu číslicích L a základu A. K tomuto číslu je možné navrhnout softwarovou sčítačkou a pomocí této sčítačky můžeme postupně proiterovat všechny řetězce o délce L. V případě že dojde k přenosu z nejvyššího řádu (poslední znak se vrátí opět na první) tak víme, že jsme proiterovali všechny možnosti.
2. Nastavíme hodnoty sčítačky na počáteční hodnotu (v sekvenčním algoritmu to jsou samé 0).
3. Tento řetězec zahashujeme pomocí funkce MD5.
4. Porovnáme výsledný hash se zadaným hashem. Pokud se shoduje vrátíme daný řetězec = jedná se o původní hodnotu a pokračujeme ke kroku 7), jinak pokračujeme na krok 5)
5. Inkrementujeme řetězec o 1 (přičteme k nejnižší číslici číslo 1).
6. Pokud nastane přenos z nejvyššího řádu tak byly vyzkoušeny všechny kombinace – pokračujeme krokem 7, jinak pokračujeme na krok 3)
7. V případě nalezení dané hodnoty jí vypíšeme, jinak vypíšeme „No matches“, uvolní se prostředky, program se ukončí.

Rozšířený slovníkový útok:

Finální verze programu obsahuje pouze omezený rozšířený slovníkový útok, podporuje pouze přidávání řetězců různé délky za dané slovo. Tento typ útoku slouží spíše pro ilustraci zrychlení na GPU.

Update 2020: Nová verze programu obsahuje také další variantu rozšířeného slovníkového útoku, který nahrazuje znak/y jiným definovaným znakem.

Sekvenční řešení vždy přečte jeden řetězec ze slovníku, poté následně tento řetězec zahashuje pomocí funkce MD5, které následně porovná se zadaným hashem. Pokud se hash shoduje tak tento řetězec je původní heslo. Pokud hash neshoduje, tak se za daný řetězec pomocí bruteforce funkce přidávají všechny řetězce dané délky a abecedy. Tyto rozšířené řetězce se taktéž hashují pomocí funkce MD5 a následně porovnávají se zadaným hashem.

Substituce znaků je řešena rekurzivně, v případě že se nalezne znak, který se má nahradit na pozici N tak se nahradí a na tento upravený řetězec se zavolá stejná funkce která pracuje se slovem s počáteční adresou N+1. Po dokončení dané rekurze se slovo vrátí do původního stavu a zavolá se rekurze na neupravené slovo s počáteční adresou slova N+1.

OpenMP:

Útok hrubou silou:

Důležité na paralelním řešení je práci spravedlivě rozdělit mezi výpočetní vlákna. To se dělá tím způsobem, že je znám celkový počet slov, které je nutné otestovat – [Velikost abecedy]^[Délka

slova]. Když známe počet kombinací, tak můžeme víceméně (většinou se nám nepoštěstí, aby celkový počet kombinací byl dělitelný počtem vláken) rozdělit danou práci mezi jednotlivá vlákna. To je uskutečněno pomocí

```
#pragma omp parallel private(hashPlaceHolderNew)
	{
#pragma omp for schedule(dynamic)

		for (int c = 0; c < cores; c++)
		{

		}
```

V tomto foru se v první řadě zjistí počáteční slovo, které má dané vlákno vykonávat. Počáteční slovo se určuje podle ID vlákna, které se vynásobí počtem slov, které má každé vlákno vykonat – tímto vynásobením nezískáme přímo slovo, ale číslo, určující pořadí daného slova. Pomocí dělení a modulení jsme schopni zjistit počáteční permutaci jednotlivých písmen. Poté postupujeme ke 3. kroku sekvenčního řešení a pokračujeme stejným způsobem s tím rozdílem, že v případě nalezení výsledku se nastaví flag, který ostatním vláknům indikuje, že již byl nalezený výsledek a nemá cenu pokračovat dál ve výpočtu. Samotná kontrola flagu probíhá každých x slov (v měřeních je hodnota x nastavena na 1000 slov). V případě, že je tento flag nastavený na 1 končí výpočet.

Pro tuto úlohu (a ještě více pro slovníkový útok) by se hodily OpenMP tasky, bohužel jejich použití nebylo možné, protože velká část měření probíhá na sestavě s Windows 10, kde se jako kompilátor využívá MSVC++. Tento kompilátor z neznámého důvodu podporuje maximálně verzi OpenMP 2.0, tato verze tasky nepodporuje.

Rozšířený slovníkový útok:

Na rozdíl od sekvenčního řešení, kde se ze souboru vždy načetlo pouze jedno slovo, se kterým se následně pracovalo se zde dělá to, že se před samotnou paralelní částí načte větší množství slov – v tomto případě dané konstantou OMP_WORDS_PER_CORE, které se násobí počtem jader.

Poté se na toto pole slov pustí paralelní for s dynamickým plánovačem a každé vlákno odebírá z tohoto pole vždy slovo po slově a následně s každým slovem pracuje jako sekvenční kód. Dynamický plánovač je z důvodu, že v případě substituce znaků se s každým slovem pracuje jinou dobu (záleží na délce slova, počet znaků, které se musí substituovat atd..) a tento plánovač zajistí to, že bude zátěž rovnoměrně rozvrstvena mezi všechna vlákna. Po vykonání paralelní části se zjišťuje jestli už byl přečten celý slovník, pokud ne, tak se opět načte další sada slov a opět se na to zavolá paralelní for.

Zde stojí za zmínku to, že na rozdíl od openMP brute force se slovy pracuje ve stejném pořadí jako by se pracovalo v sekvenčním kódu, takže by mělo být zřetelnější zrychlení – v případě OpenMP záleží na štěstí jak správně „rozsekáme“ stavový prostor pro jednotlivá vlákna.

V případě, že se volá kód s 1 vláknem tak se volá sekvenční verze rozšířeného slovníkového útoku.

Vektorizace:

Vektorizace MD5:

Samotný algoritmus nelze příliš vektorizovat, protože u hlavní smyčky data z aktuálně prováděné iterace závisí na předešlé iteraci. Do jisté míry by bylo možné vektorizovat tuto část(vnější cyklus):

```
// Process the message in successive 512-bit chunks: for each 512-bit chunk of padded message do
```

Většinou ale funkce MD5 pracuje s kratšími slovy, než je 512 bitů, takže by vektorizace příliš výkonu navíc nepřinesla (spíše naopak).

Další možnost vektorizace MD5 je v jednu chvíli počítat více slov najednou. V tomto případě by ale vzrostla režie, musela by se řešit délka slova, protože vnější smyčka se vykonává podle délky slova. Tohle by ale bylo možné řešit jednoduchým omezením, že heslo nemůže být delší než určitý počet znaků.

Vektorizace porovnávání hashu:

Výsledný hash je 128 bitový. Současný program porovnává hash po čtyřech 32 bitových částech daného hashe a kód vypadá následovně:

```
void isHashEqualNew(uint32_t * hash1, uint32_t * hash2, bool * ret)
{
	*ret = true;
	for (int i = 0; i < 4; i++)
	{
		if (hash1[i] != hash2[i])
		{
			*ret = false;
			return;
		}
	}
}
```

Jak je zřetelné z kódu v hodně případů cyklus proběhne pouze jednou (takže se provedou 2 načtení z paměti a jedna porovnávací funkce).

V případě vektorizace, by bylo možné porovnat hash najednou. Ovšem, bylo by pak nutné výsledek z vektorového registru nahrát zpátky do paměti a poté porovnat skalárně. Ve výsledku by to vypadalo tak, že v každé kontrole shodnosti hashe se načte dvakrát z paměti do registrů (referenční hash, kontrolavaný hash), poté se provede jedna porovnávací instrukce, poté se výsledek uloží do paměti. A poté se z paměti část po části čte a kontroluje. Vektorizace by program dokázala reálně zrychlit pouze kdyby nastavovala flagy procesoru (odpadlo by ukládání výsledku do paměti a následné čtení z paměti), jinak by se jednalo o zpomalení.

CUDA:

Popis řešení pro GPU:

Útok hrubou silou:

Nejprve host nakopíruje používanou abecedu a zadaný hash do paměti konstant GPU a připraví paměť pro uložení nalezeného řetězce. Poté následně spustí kernel s N bloky a M vlákny (obě konstanty jsou nastavitelné během spuštění programu). Kernel se spouští zvlášť pro řetězec každé délky (takže např, pokud zkoušíme hesla v rozsahu 1-3 znaků, tak se nejprve pustí kernel pro řetězce délky 1, pak 2 atd..). Po spuštění kernelu se čeká až je dokončen a host následně zkontroluje, jestli se podařilo nalézt původní řetězec. Pokud ano, tak se vypíše a program končí, pokud ne pokračuje se s dalším rozsahem, či pokud vyčerpal všechny možnosti tak program končí s hláškou „No matches“.

Poté se na device pro každé vlákno zjistí, od jaké počáteční konfigurace sčítačky má začít počítat. Pro ošetření, aby se projely všechny kombinace se používá zaokrouhlování přídělu práce směrem nahoru, je tedy možné, že některá vlákna vykonávají stejnou práci. Začátek práce je určeno 64-bitovým číslem (bylo by možné rozšířit i na 128, či 256-bitové číslo), počáteční permutace se zjistí tak, že se toto číslo postupně modulí a dělí (stejný princip jako bychom převáděli třeba desítkové číslo na dvojkové).

Poté se zavolá funkce podobná sekvenční funkci, ovšem s jiným počátečním nastavením sčítačky než samé 0. Tato funkce také vždy po určitém počtu iterací (podle nastavení THRESHOLD) kontroluje, jestli nějaké z vláken již původní hodnotu nenašlo (pokud ano skončí práci).

V případě nalezení správného hesla (hashe sedí) se toto heslo zapíše do paměti, kterou připravil host. Vzhledem k nízké pravděpodobnosti, že dvě vlákna naleznou řetězec se stejným otiskem se vůbec neřeší atomické operace.

Rozšířený slovníkový útok:

Vzhledem k libovolné velikosti slovníku (teoreticky může být velký i několik TB) a předem neznámé velikosti RAM a VRAM program tento slovník „porcuje“ po blocích určité délky. Načítání slov na hostovi probíhá ve dvou fázích. V první fázi se zjišťuje, jak dlouhé je nejdelší slovo v dané skupině slov a kolik slov bude vlastně načteno. Všechna slova budou muset být zarovnána na délku nejdelšího slova (aby bylo možné se stringy smysluplně pracovat). Počet slov se určí v závislosti na nastavení GRANULARITY (určuje na kolik slov bude zarovnávat, např. pokud je GRANULARITY 100 a při 565. slovu nebude stačit paměť načte se pouze 500 slov) a MEMORY_RATIO, který určuje, jak velkou část paměti VRAM využít na ukládání slov. Maximální délka slova je omezena konstantou MAX_WORD_LENGTH.

Po získání počtu slov v dané iteraci se následně tato slova načtou do paměti hostitele, vzhledem k nemožnosti použití strlen na device se tato informace ukládá na poslední bajt daného slova (vždy je alokováno max(strlen) + 2, jeden bajt pro null byte a druhý právě pro uložení délky aktuálního slova). Vzhledem k datovému rozsahu byte (resp. unsigned char) pro uložení maximální délky slova tato implementace nepočítá se slovy delších, než je 255 znaků.

Před samotným spuštěním kernelu je ještě do paměti symbolů nakopírován hash, slovníky a pravidla pro rozšířený slovníkový útok. Taktéž se připraví paměť na uložení řetězce, který má stejný hash jako hledaný hash. Poté se nakopírují slova ze slovníku do paměti VRAM, slova jsou uložena v 1D poli a každé slovo je zarovnané na délku nejdelšího slova + 2. Poté se konečně zavolá kernel, během vykonávání kernelu se připraví do paměti hosta další slova ze slovníku.

Na straně device se podle bloku a vlákna získá oblast paměti nad kterou má dané vlákno pracovat, poté je podobný postup jako v sekvenční variantě s tím rozdílem, že vždy po konstantním počtu vyzkoušených slov (určený v DICTIONARY_THRESHOLD) se kontroluje, jestli již náhodou nebyl řetězec s odpovídajícím hashem nalezen.

Měření:

Sestava 1 – Windows	
Procesor	AMD Ryzen 5 1600 **@3.4GHz**
Grafická karta	MSI GeForce GTX 1080 SEA HAWK X (GDDR5X 10108MHz) GTX1080 (1708MHz)
Paměť	RAM: G.SKILL 16GB KIT DDR4 3200MHz CL14 Flare X for AMD
Pevný disk (umístění slovníku):	Seagate BarraCuda 2TB 7200RPM
Operačný systém	Windows 10 Professional
Poznámka:	Úplná optimalizace (/Ox)

Sestava 2 – STAR	
Procesor	2ks 6core Xeon 2620 v2 @ 2.1Ghz
Grafická karta	GeForce RTX 2080 Ti

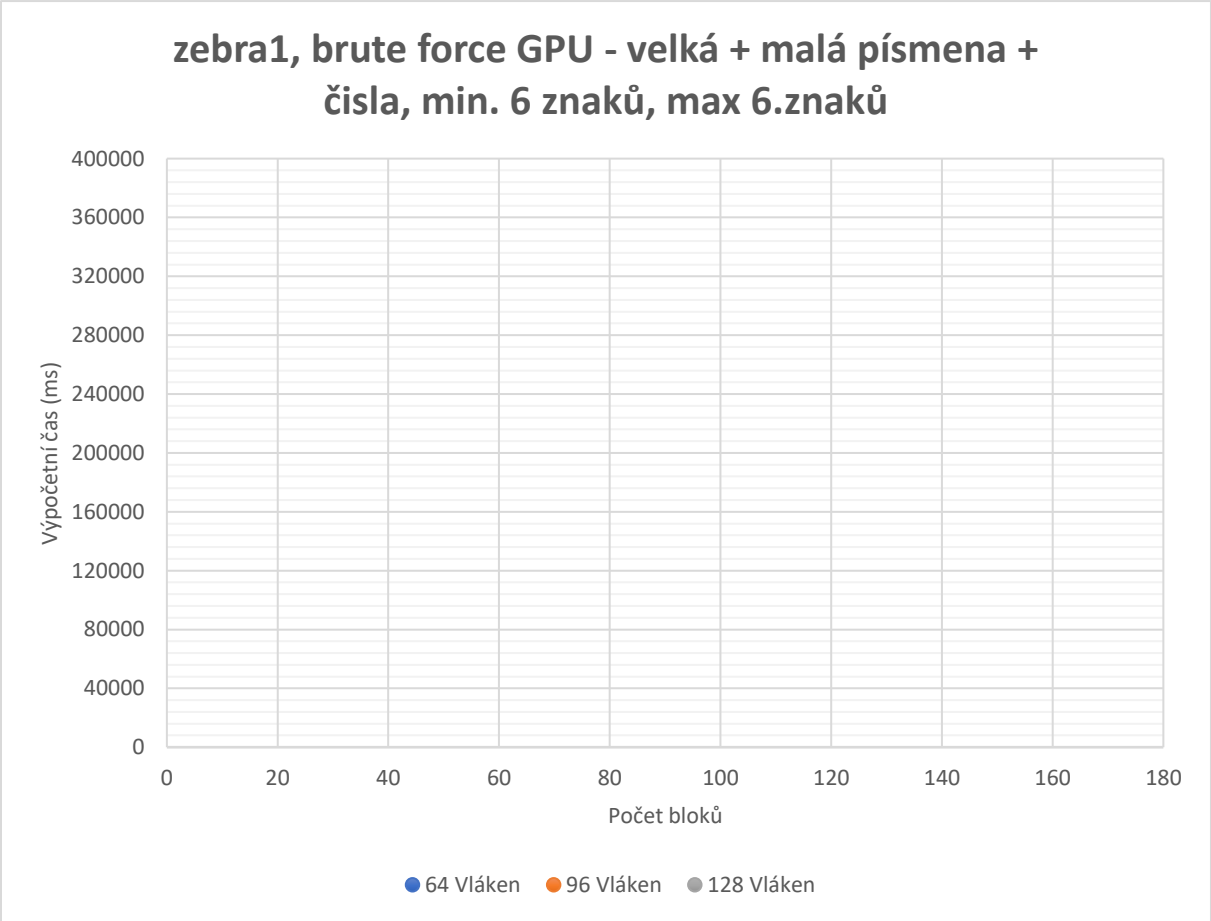
Paměť	32 GB
Operačný systém	CentOS Linux 7 (Core)
Poznámka:	Bez Optimalizace **OpenMP varianta:** -O3 -mavx -fopt-info-vec

Parametry blocks a threads

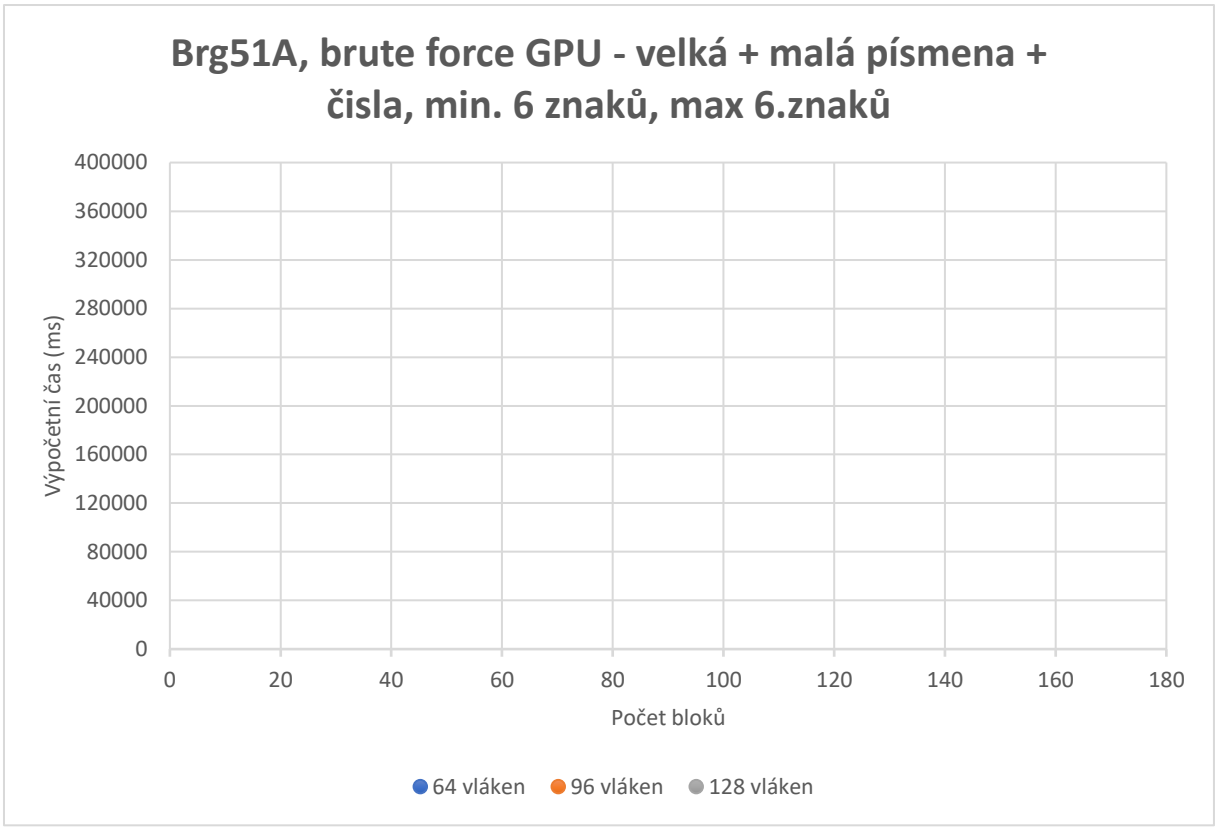
Tyto dva parametry mají na výpočetní čas obrovský vliv, správné nastavení těchto parametrů dokáže chod programu značně urychlit, naopak špatně zvolené parametry délku dobu zvyšují. Bohužel měření ukazují, že neexistuje univerzální nastavení těchto parametrů pro všechny hashe, ke všemu se tyto parametry liší i grafická karta od grafické karty. I při špatném nastavení parametrů je výpočet na GPU rychlejší, než výpočet na CPU.

Brute force GPU:

Následující data byla měřena na sestavě 1.



Nejkratší čas pro zebra1 vychází na **24 sekund** pro konfiguraci 96 vláken a 16 bloků, nejvyšší čas je **362 sekund** pro konfiguraci 128 vláken a 112 bloků. Průměrný čas napříč všemi konfiguracemi je **247 sekund.** Pro 64 vláken je průměrný čas **217 sekund,** pro 96 vláken **221 sekund** a pro 128 vláken **301 sekund**.

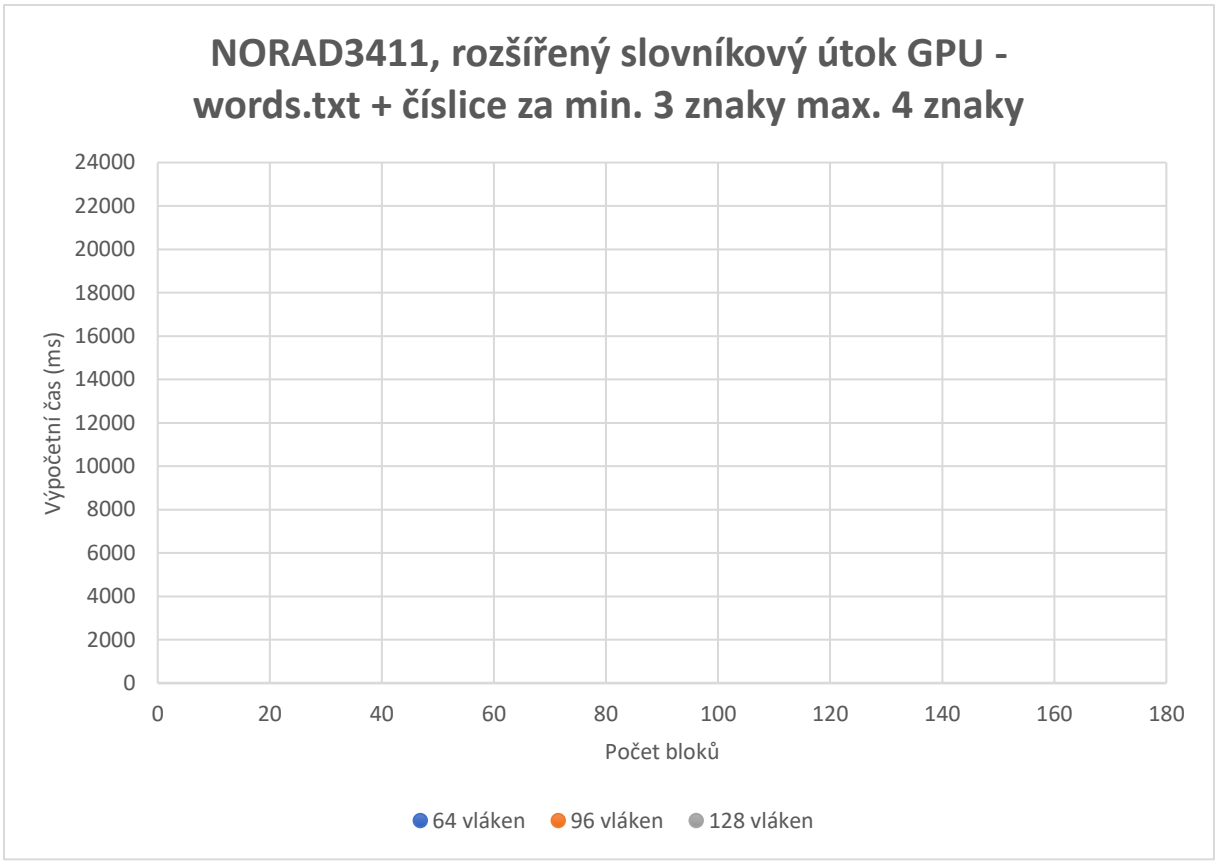


Nejkratší čas pro slovo Brg51A vychází **7,8 sekund** pro konfiguraci 96 vláken a 48 bloků, nejdelší čas je **350 sekund** pro konfiguraci 64 vláken a 80 bloků. Průměrný čas je **132 sekund.** Průměrný čas pro 64 vláken je **134 sekund**, průměrný čas pro 96 vláken je **113 sekund,** průměrný čas pro 128 vláken je **150 sekund.**

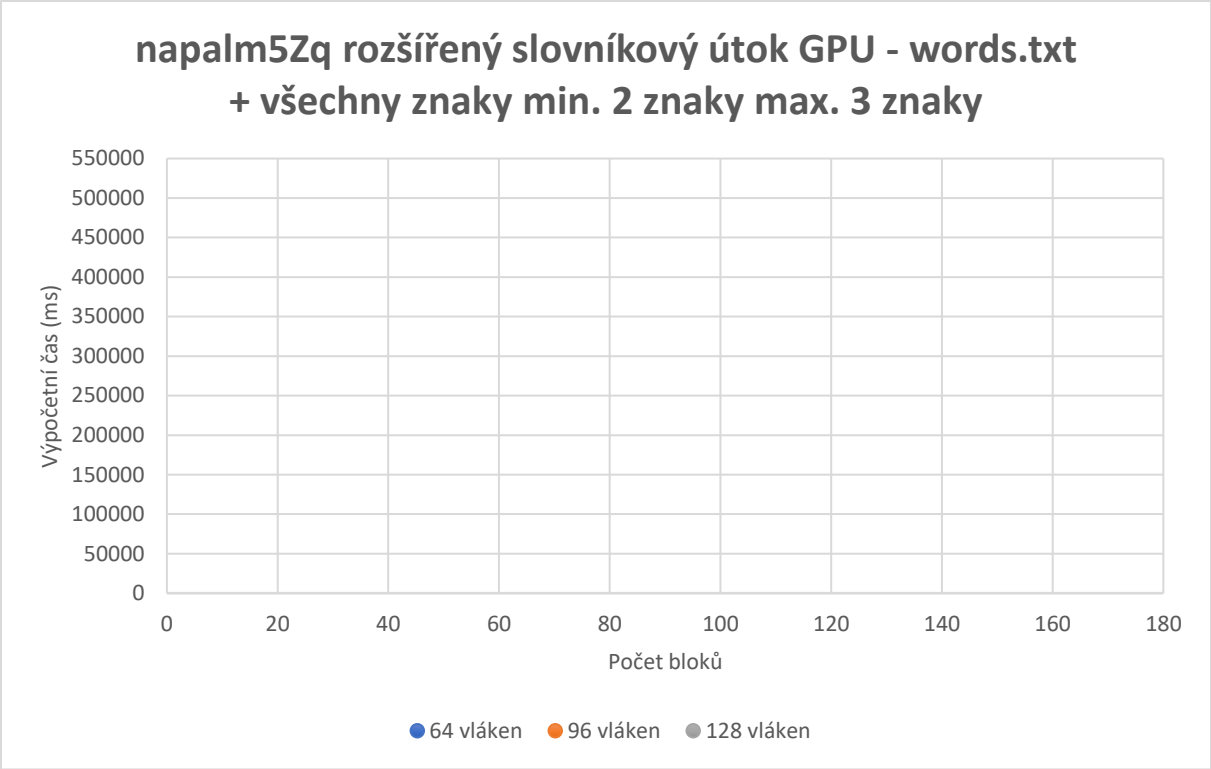
Tato dvě měření ukázala, že nastavení parametrů blocks/threads má v případě útoku hrubou silou velký vliv. Nejspíše ale nebude existovat kombinace parametrů, která by byla optimální pro všechny hashe.

Rozšířený slovníkový útok GPU:

Následující data byla měřena na sestavě 1.



Byl testován vliv bloků/vláken na rozšířený slovníkový útok. Jako slovník se používal slovník některých anglických slov (cca. 500 000 slov) a jako pravidla se použilo přidávání řetězců dané abecedy (pouze číslice) v rozsahu 3-4 znaků. Jako heslo bylo zvoleno slovo **NORAD3411.** Nejkratší čas vychází na **6,2 sekund** pro kombinaci 64 vláken a 32 bloků, nejdelší čas **22,8 sekund** pro kombinaci 128 vláken a 48 bloků. Průměrný čas vychází na **15 sekund.** Průměrný čas pro 64 vláken vychází na **14,6 sekund**. Průměrný čas pro 96 vláken vychází na **14,55 sekund**, průměrný čas pro 128 vláken vychází na **16 sekund.**



Nejnižší čas je **82 sekund** pro kombinaci 128 vláken a 16 bloků, nejdelší výpočetní čas je **517 sekund,** průměrný čas je **274 sekund**. Průměrný čas pro 64 vláken je **284 sekund**, pro 96 vláken **255 sekund** a pro 128 vláken je **285 sekund.**

Testování na sestavě 1:

Sekvenční řešení:

Útok hrubou silou - CPU		
Heslo	**Abeceda**	**Výpočetní čas (ms)**
99999	Malé a velká písmena, číslice rozsah min. 5 max 5	236912
9999	Malé a velká písmena, číslice rozsah min. 4 max 4	3839
zebra1	Malé a velká písmena, číslice rozsah min. 6 max 6	13680000
~A9C	Všechny znaky min. 4 max 4	6115
J@K1!	Všechny znaky min 4 max 5	1262598
Test výkonu[1]	Všechny znaky min 4 max 4	20031

Výpočetní výkon pro sekvenční útok hrubou silou:

$$\frac{Hash}{s} = \frac{[Celkový\ počet\ slov]}{[doba\ vykonávání]s}$$

$$\frac{Hash}{s} = \frac{94^4}{20} = \frac{78074896}{20}$$

$$\frac{Hash}{s} = 3903744$$

Sekvenční řešení útoku hrubou silou má výpočetní výkon **3,9 MHash /s.**

Rozšířený slovníkový útok - CPU			
Heslo	**Slovník**	**Pravidla**	**Výpočetní čas (ms)**

[1] Zkouška výpočetní síly, vložení špatného hashe => nutnost prozkoumat celý stavový prostor

ZZZ989	Cca 500k slov	Přidávání za slovo čísla délky 1 a čísla délky 3	121666
napalm5e	Cca 500k slov	Přidávání za slovo čísla a písmena délky 2	246923
NORAD3411	Cca 500k slov	Přidávání za slovo čísla v rozsahu 3-4 znaků	750494
azotemic9S	Cca 500k slov	Přidávání za slovo čísla a písmena délky 2	30351
Test výkonu	Cca 500k slov	Přidávání za slova čísla délky 2	12303

Výpočetní výkon pro rozšířený slovníkový útok:

Výpočet je stejný jako v případě výpočtu výkonu pro útok hrubou silou, počet všech možných slov se získá tím způsobem, že se vezme následující suma:

$$\sum_{k=0}^{\max počet\ znaků} [počet\ slov\ slovníku] * [velikost\ abecedy]^k$$

Testovací slovník obsahuje přesně **466551 slov**, za každé z těchto slov můžeme přidat celkem 100 různých čísel (00-99) celkově teda program musí otestovat **466551 + 466551*100** slov.

Rozšířený slovníkový útok pomocí CPU má výpočetní výkon přibližně **3,830MHash/s**, takže přibližně srovnatelný se výpočetním výkonem útoku hrubou silou.

Paralelní řešení - OpenMP:
Pravidla a abeceda jsou shodná se sekvenčním řešením, proto je zde nebudu uvádět.

	Útok hrubou silou – výpočetní čas(s)						
Heslo/vlákna:	**1**	**2**	**4**	**6**	**8**	**10**	**CUDA(nejlepsi cas)**
99999	236,9	160,8	80,7	58,0	51,3	50,8	2,9
9999	3,83	2,76	1,99	0,874	0,862	0,829	0,378
~A9C	6,1	8,26	2,57	4,00	2,61	0,230	0,362
J@K1!	1262,6	442,28	662,84	571,88	165,78	275,42	4,643
	Útok hrubou silou – výpočetní výkon (MHash/s)						
Test výkonu	3,9	5,7	11,3	10,8	13,46	15,3	301

V případě slov, které jsou v rámci vlákna testována vždy jako poslední (nezáleží na počtu vláken

U výpočetního výkonu lze vidět (s výjimkou 6 vláken, kde si nejsem jistý, proč dochází k poklesu výkonu), že s přidávajícím počtem vláken roste výpočetní výkon – s přimhouřenými očima lze říct lineárně. Tento lineární růst končí na cca 6 vláknech a poté se růst zpomaluje. Je to dané tím, že testovací procesor má 6 fyzických jader, ke všemu sestava 1 je můj osobní počítač, kde běží dalších x procesů, které procesor zaměstnávají.

	Rozšířený slovníkový útok – výpočetní čas(s)						
Heslo/vlákna:	**1**	**2**	**4**	**6**	**8**	**10**	**CUDA(nejlepsi cas)**
ZZZ989	121,7	83,8	42,5	30,5	26,9	24,89	2,03
napalm5e	246,9	170,4	86,4	61,3	54,8	50	2,75

lupuSErythemAToSus[2]	32,27	16,25	9,3	6,9	7,13	5,9	N/A[3]
	Rozšířený slovníkový útok – výpočetní výkon (MHash/s)						
Test výkonu	3,83	5,54	10,6	13,9	15,87	18,12	331

V případě rozšířeného slovníkového útoku je jasně zřetelné, že výkon roste lineárně s přibývajícím počtem vláknem. Podobně jako u útoku hrubou silou nad 6 vláken výkon už neroste tak rychle, ale mezi 2-4-6 vlákny je ve všech případech očekávaný nárust. Toto je dané tím, že se slova skutečně testují v tom pořadí, v jakém by se testovaly v sekvenčním kódu a není zde takový prvek „náhody" jako v případě útoku hrubou silou.

Řešení pomocí CUDA:

Vzhledem k velkému vlivu parametrů threads a blocks na celkový čas bude v tabulce uveden nejlepší možný naměřený čas, nejhorší možný naměřený čas a průměrný naměřený čas.

Útok hrubou silou - GPU				
Heslo	**Abeceda**	**Výpočetní čas (ms)**		
		Nejlepší	**Nejhorší**	**Průměrný**
zebra1	Malé a velká písmena, číslice rozsah min. 6 max 6	24033	362224	246851
99999	Malé a velká písmena, číslice rozsah min. 5 max 5	2892	6218	5136
9999	Malé a velká písmena, číslice rozsah min. 4 max 4	378	432	413
~A9C	Všechny znaky min. 4 max 4	362	677	518
J@K1!	Všechny znaky min 4 max 5	4643	47982	26766
Test výkonu	Všechny znaky min 5 max 5	22163	48871	40996

Výpočetní výkon pro útok hrubou silou realizovaný na GPU:

Výpočet je totožný s výpočtem pro sekvenční útok na hrubou silou, jen se vezme jiný počet všech slov (94^5).

S optimálně nastavenými parametry THREADS/BLOCKS vychází výpočetní síla hrubé síly na GPU na **331 MHash /s** (téměř 85x více než CPU varianta), při špatném nastavení parametrů vychází výpočetní výkon na **150 MHash/s** (cca 39x rychlejší než CPU varianta) a průměrný výpočetní výkon napříč nastaveními vychází je **179 MHash/s** (cca 46x rychlejší než CPU varianta).

Rozšířený slovníkový útok - GPU					
Heslo	**Slovník**	**Pravidla**	**Výpočetní čas(ms)**		
			Nejlepší	**Nejhorší**	**Průměrný**

[2] Použití substitučního pravidla, kde se 8 nejčastějších anglických písmen (jak velkých, tak i malých) nahrazují (malými/velkými písmeny) eariotnsEARIOTNS -> EARIOTNSeariotns

[3] Substituce znaku za jiný znak není v GPU verzi podporována

ZZZ989	*	Přidávání za slovo čísla délky 1 a čísla délky 3	2020	3257	2724
napalm5e	*	Přidávání za slovo čísla a písmena délky 2	2753	9151	5866
NORAD3411	*	Přidávání za slovo čísla v rozsahu 3-4 znaků	6240	22782	15066
azotemic9S	*	Přidávání za slovo čísla a písmena délky 4	2439	8902	4688
Test výkonu		Přidávání za slova čísla délky 2	15506	26764	22184

* Cca 500k slov

U slovníkového útoku hodně záleží na umístění daného slova ve slovníku, např. u slova azotemic, které je ve slovníku na začátku není zrychlení na GPU tak výrazné.

Výpočetní výkon pro rozšířený slovníkový útok:

Výpočet probíhá stejným způsobem jako v případě sekvenčního řešení. S optimálně nastavenými parametry je výpočetní síla rozšířeného slovníkového útoku na GPU **301 MHash/s** (cca 79x rychlejší než sekvenční řešení), při špatném nastavení parametrů je výpočetní síla **174 MHash/s** (cca. 46x výkonější) a průměrný výpočetní výkon je **210 MHash/s** (cca 55x výkonnější než sekvenční řešení).

Testování na sestavě 2:

Sekvenční řešení:

Útok hrubou silou - CPU		
Heslo	**Abeceda**	**Výpočetní čas (s)**
99999	Malé a velká písmena, číslice rozsah min. 5 max 5	551
9999	Malé a velká písmena, číslice rozsah min. 4 max 4	8,68
~A9C	Všechny znaky min. 4 max 4	14,12
J@K1!	Všechny znaky min 4 max 5	2935,05

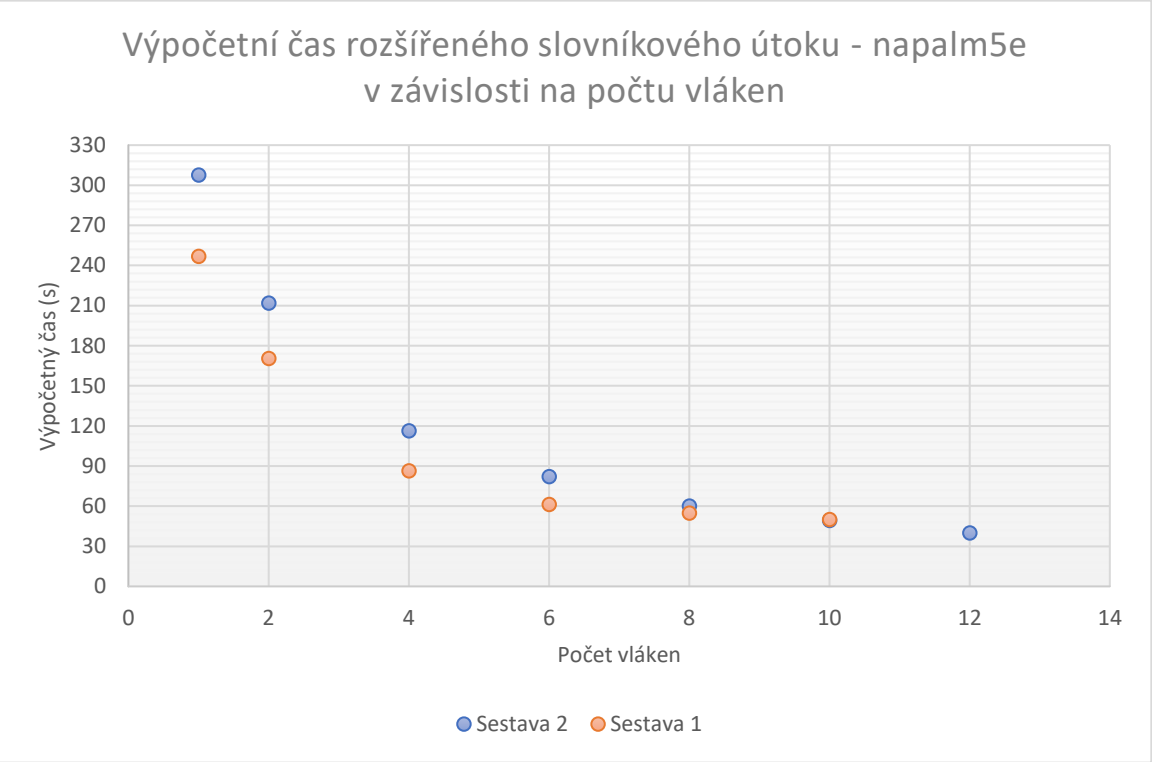
Rozšířený slovníkový útok - CPU			
Heslo	**Slovník**	**Pravidla**	**Výpočetní čas (s)**
ZZZ989	Cca 500k slov	Přidávání za slovo čísla délky 1 a čísla délky 3	292,20
napalm5e	Cca 500k slov	Přidávání za slovo čísla a písmena délky 2	582,233
azotemic9S	Cca 500k slov	Přidávání za slovo čísla a písmena délky 2	72,54

Paralelní řešení - OpenMP:

	Útok hrubou silou – výpočetní čas(s)							
Heslo / vlákna	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**CUDA**
99999	284,3	193,3	104,58	71,315	55,46	43,69	36,16	2,84
9999	4,61	3,15	1,7	1,16	0,95	0,8	0,7	1,23
~A9C	7,4	10,39	1,9	5,32	2,13	0,2	2,13	1,23

J@K1!	1539	533,6	587,4	576,8	144,6	242,3	304,5	3,11

	Útok hrubou silou – výpočetní čas(s)							
Heslo / vlákna	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**CUDA**
napalm5e	307,5	211,8	116,3	82,1	60	49,2	39,9	9,18
azotemic9S	38,15	25,93	14,23	9,99	7,34	5,95	4,98	2,51
lupuSErythemAToSus	30	19,95	12,16	8,12	6,99	5,34	5,27	N/A



	Počet vláken						
	1	**2**	**4**	**6**	**8**	**10**	**12**
Očekávaná délka výpočtu:	1	0.5	0.25	0,166	0,125	0,1	0,083
Sestava 1:	1	0,69	0,35	0,25	0,22	0,2	N/A
Sestava 2:	1	0,69	0,38	0,27	0,2	0,16	0,13

Jak ukazuje graf a tabulka výše tak v obou případech pokles doby výpočtu neodpovídá očekávanému trendu. Zde ale stojí za zmínku, že sekvenční kód je jednodušší a nepracuje tolik s pamětí jako právě paralelní kód. Ale i přesto, pokles výpočetního času je zřetelný.

Jak ukazuje tabulka, ale i graf, tak trend poklesu délky výpočtu je podobný (v některých případech totožný) na obou sestavách, takže kód není závislý ani na architektuře CPU, ani na použitém překladači.

Paralelní řešení – Xeon Phi

Program kompilovaný pro Xeon Phi byl spouštěn pod OS, jehož implementace date neumožnovala výpis nanosekund (ale pouze sekund), měření tudíž nebude s přesností na milisekundy, ale pouze na sekundy. U většiny instancí to příliš velký vliv nemá s výjimkou 9999.

	Útok hrubou silou – výpočetní čas(s)								
Heslo/V.	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**244**
99999	2709	1468	769	356	180	106	81	51	33
9999	43	24	12	6	3	2	2	1	1
~A9C	71	78	14	13	14	7	3	<1	<1
J@K1!	14673	4082	4096	951	1057	233	248	71	112

	Rozšířený slovníkový útok – výpočetní čas(s)								
Heslo/Vlákna	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**244**
napalm5e	2839	1485	783	395	197	103	73	37	36
azotemic9S	350	187	96	49	26	15	7	7	7
lupuSErythemAToSus	280	150	88	55	35	22	16	17	19

Řešení pomocí CUDA

Zde je opět uváděn vždy pouze nejlepší, nejhorší a průměrný čas (kde se liší počty threadů a bloků)

Útok hrubou silou - GPU				
Heslo	**Abeceda**	**Výpočetní čas (s)**		
		Nejlepší	**Nejhorší**	**Průměrný**
zebra1	Malé a velká písmena, číslice rozsah min. 6 max 6	28,72	374,78	115,05
99999	Malé a velká písmena, číslice rozsah min. 5 max 5	2,84	7,49	4,15
9999	Malé a velká písmena, číslice rozsah min. 4 max 4	1,23	1,38	1,29
~A9C	Všechny znaky min. 4 max 4	1,23	1,87	1,39
J@K1!	Všechny znaky min 4 max 5	3,11	48,61	14,96

Lze si všimnou, že pro zebra1 je rozdíl mezi nejlepším a nejhorším časem zhruba 170 %. Zatímco pro 99999 se jedná o 90 %. U zebra1 bylo nejlepšího výsledku dosaženo za použití 128 bloků a 96 threadů. U 99999 64 bloků a 64 threadů.

Pro slovo 99999 je nejlepší čas zhruba 180x lepší než při sekvenčním řešení. I nejhorší naměřený čas je stale mnohem lepší než sekvenční řešení. To platí i pro ostatní případy, vždy došlo ke zlepšení.

Rozšířený slovníkový útok - GPU					
Heslo	**Slovník**	**Pravidla**	**Výpočetní čas(s)**		
			Nejlepší	**Nejhorší**	**Průměrný**
ZZZ989	*	Přidávání za slovo čísla délky 1 a čísla délky 3	2,03	3,93	2,55
napalm5e	*	Přidávání za slovo čísla a písmena délky 2	9,18	58,19	20,57

NORAD3411	*	Přidávání za slovo čísla v rozsahu 3-4 znaků	18,46	158,61	50,3022
azotemic9S	*	Přidávání za slovo čísla a písmena délky 4	2,51	14,03	5,85

Opět jsou naměřené časy lepší než sekvenční. Například pro azotemic9S je průměrný naměřený čas lepší zhruba 12x než u sekvenčního řešení.

Vzhledem k tomu, že měření na Sestavě 2 bylo provedeno s využitím jiného kompililátoru a bez optimalizace, tak porovnání měření na těchto dvou strojích může být zavádějící.

Závěr:

Měření ukázalo, že grafická karta je vhodný nástroj na prolamování kryptografických hashů. I ne příliš dobře optimalizované řešení nabízelo výrazné zrychlení. Jedná se o zcela základní řešení, které toho příliš mnoho neumí a ani z daleka nedosahuje kvalit ostatních nástrojů (hashcat), co by bylo možné do budoucna zlepšit:

- **Rozšíření možností slovníkového útoku** – aktuálně pouze podporuje přidávání řetězců za slova, v budoucnu by bylo možné triviálně implementovat přidávání znaků před/do slova. Dále by bylo možné implementovat nahrazování znaků za jiné znaky (např. 0->O, I – 1 atd..), či třeba spojování více řetězců za sebe. U těchto rozšířený by bylo ale třeba zvážit potřeba řazení daných slov podle délek, aby nedocházelo k nerovnoměrné práci napříč vlákny.
- **Optimalizace MD5 algoritmu**
- **Změření vlivu THRESHOLD, DICTIONARY_THRESHOLD, GRANULARITY a dalších na výpočetní čas**
- **Vytvořit velký statistický vzorek na zkoumání vlivu hodnot BLOCKS a THREADS** - pomocí tohoto vzorku by bylo pak možné určit „nejideálnější“ paremetry napříč problémy..
- **Atd**..

Literatura:

https://cs.wikipedia.org/wiki/Kryptografická_hašovací_funkce

https://courses.fit.cvut.cz/MI-PRC/media/labs/report_crypto.pdf