Implementácia modifikovaného algoritmu SHA-1 v jazyku C (CPU) a CUDA (GPU)

Peter Kaňuch
Fakulta informatiky a informačných technológií
Slovenská Technická Univerzita
Slovenská republika, 841 04 Bratislava IV
Email: xkanuch@stuba.sk

***Abstrakt*—Keďže grafické procesory sú prispôsobené pre paralelné spracovanie úloh na viacerých jadrách, implementácia takto prispôsobeného problému zníži čas potrebný na jeho vykonanie oproti klasickým počítačovým procesorom. V článku sa zaoberáme základnou architektúrou CPU a GPU procesorov, implementáciou hashovacieho algoritmu SHA-1 upraveného pre paralelné spracovanie uvedeného v článku [10], s cieľom poukázať na hlavné rozdiely v architektúrach týchto procesorov. Našim hlavným cieľom je však ukázať, prečo v súčasnosti neexistuje automatický paralelizmus výpočtovo náročných úloh a ich vykonávanie na grafických procesoroch.**

Keywords*—SHA-1, Modified SHA-1, CPU architecture, GPU architecture, Automatic parallelism for GPU

I. Úvod

V posledných rokoch sa grafické procesory začali dostávať do popredia, nie len na spracovanie obrazu, ale aj na vykonávanie, či spracovanie rôznych náročných výpočtových úloch. Architektúra grafických procesorov bola navrhnutá a optimalizovaná pre paralelné spracovanie inštrukcií a dát, zatiaľ čo CPU sú optimalizované pre rýchle sequenčné spracovanie toku programu.
Zvyšovanie rýchlosti a výpočtovej výkonnosti počítačov je čoraz viac žiadanejšie. Na rôzne výpočtovo náročné úlohy sa používajú multipočítače, no aj pre tieto výkonné stroje by niektoré úlohy trvali dlho. Preto sa vymýšľajú rôzne spôsoby, ako dosiahnúť lepší výkon. Jedným zo spôsobou je preniesť spracovanie náročných úloh na grafické procesory. To si však vyžaduje určité technické znalosti. V tomto článku preto porovnáme základnú architektúru CPU a GPU. Vysvetlíme princíp fungovania algoritmu SHA-1 a jeho modifikovanej paralelizovanej verzii, pretože hashovacie funkcie sú používané v oblasti počítačovej bezpečnosti. Daný algoritmus implementujeme v programovacom jazyku C pre klasické a v jazyku CUDA pre grafické procesory. Výsledky ukazujú porovnanie nameraných časov vykonávania daného algoritmu na CPU a GPU. Taktiež daný algoritmus porovnávame s implementáciou štandartne používaného nástroja pre výpočet hashovacích funkcií SHA. V závere diskutujeme, prečo ešte v súčasnosti neexistuje automatický paralelizmus a automatické vykonávanie výpočtovo náročných úloch na grafických procesoroch.

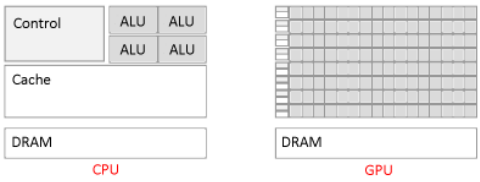
II. CPU verzus GPU architektúra

Aj keď sa zdá, že grafické procesory by mohli svojou výkonnosťou nahradiť klasické počítačové procesory, nie je tomu tak. CPU boli navrhnuté pre dosiahnutie čo najlepšieho výkonu pre sequenčné spracovanie dát, grafické procesory boli navrhnuté za účelom paralelného vykonávania inštrukcií. Pri porovnaní týchto procesorov CPU majú/sú [7]:

1) menej výpočtových jednotiek
2) optimalizované pre sériové operácie
3) nízku toleranciu latencie
4) podporu pre paralelné spracovanie (novšie verzie)

a GPU:

1) viac výpočtových jednotiek
2) vstavané pre paralelné operácie
3) vysokú toleranciu latencie
4) vysokú priepustnosť
5) lepšiu logiku riadenia



Obr. 1. CPU vz. GPU

A. Architektúra počítačových procesorov

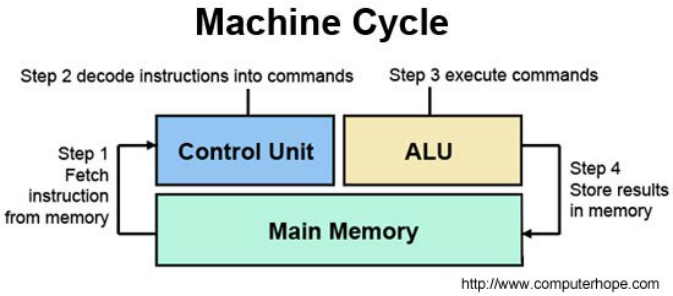
CPU (Central Processing Unit) alebo aj procesor, je hlavný komponent počítača, ktorý načítava, spracováva a vykonáva inštrukcie nad rôznymi dátami. Procesor pozostáva z dvoch hlavných častí:

- kontrolná jednotka (CU)
- aritmeticko-logická jednotka (ALU)

Základný procesor pozostáva z piatich fáz [1]:

- IF (Instruction Fetch) - fáza, v ktorej sa načítava inštrukcia z pamäte do procesora na základe adresy v registry IP/PC (instruction pointer/program counter) a jeho následnej inkrementácii na ďalšiu adresu
- ID (Instruction Decode) - zabezpečuje dekódovanie inštrukcie
- EX (Execute) - fáza, v ktorej procesor vykonáva rôzne výpočty pomocou ALU

- MEM (Memory Access) - v tejto fáze, procesor pristupuje do pamäte pre načítanie alebo uloženie dát
- WB (Write Back) - procesor zapíše výsledky(hodnoty z predošlých fáz vypočítané v ALU) alebo hodnoty načítané z pamäte v predchádzajúcej fáze do registrov



Obr. 2. Základný cyklus procesora

Takáto základná architektúra bola postupne vylepšovaná rôznymi mechanizmami (prúdovým spracovaním, detekciou a riešením hazardov, či závislostí, predikciou vetvenia a iné) pre dosiahnutie čo najlepšieho výkonu, t.j. dosiahnutie čo najmenšieho CPI (CPI - počet cyklov na inštrukciu).

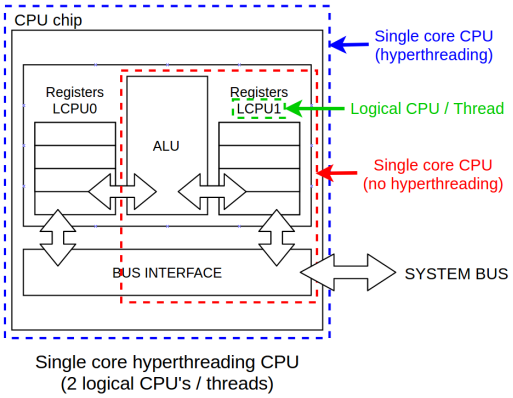
Jedným z vylepšení procesora je *prúdové spracovanie.* To spočíva v tom, že v každom hodinovom cykle procesora dokážeme začať spracovávať novú inštrukciu [1]. Tým dokážeme znížiť vykonávanie jednej inštrukcie z piatich cyklov na hodnotu blízkej jedna. Nie je to však jednoduché. Vznikajú takzvané *hazardy* a *závislosti v programe*.

Ďalším vylepšením procesora, kedy inžinieri chceli zvýšiť výkonnosť procesora, bolo vymyslením *Hyper-threading*-u. Hyper-threading vytvára z jedného fyzického, viacero logických procesorov [3]. Procesor podporujúci danú technológiu si dokáže zapamätať viacero stavov procesora pomocou pridanej kompletnej sady jednotlivých registrov (základných, kontrolných, ...). Princíp fungovania je jednoduchý: V každom čase sa spracováva len jedna úloha. Procesor však dokáže veľmi rýchlo prepínať medzi jednotlivými úlohami, čo vytvára pocit, že bežia súčasne.
Základnou nevýhodou hyper-threading-u je, že ostatné súčasti procesora (ALU, MMU, FPU, SIMD) zdieľa medzi úlohami (viď obr. 3 [6]) [4]. Preto tento spôsob zlepšenia nám nezvýši výkon pri výpočtovo náročných úlohách (napr. násobenie matíc).

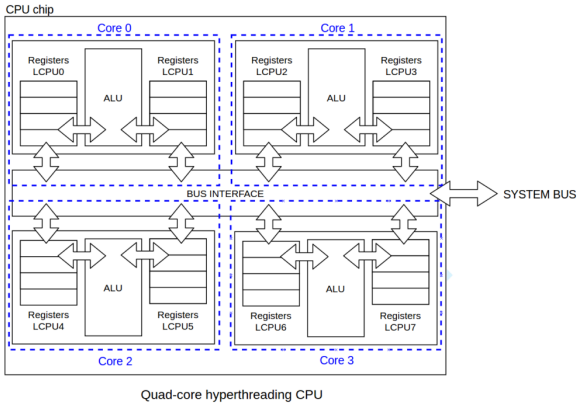
Iným vylepšením procesora je pridanie viacerých jadier procesora na jeden spoločný čip (viď obr. 4 [6]). Týmto spôsobom sa nám znásobi aj počet vykonávacich jednotiek čím dosiahneme aj znásobenie výpočtového výkonu. Súčasné bežné procesory v prenosných počítačoch obsahujú dve až štyri jadrá. Najvýkonnejšie procesory pozostávajú zo šiestich a viac jadier (12 až 18).

B. Architektúra grafických procesorov

Grafické procesory sú navrhnuté tak, aby čo najrýchlesjšie dokázali vykonať paralelizovateľný kód. Napríklad vykonať rovnaké inštrukcie nad rôznymi dátami. Do návrhu boli zahrnuté 3 myšlienky [5]:



Obr. 3. 1-jadrový procesor s hyper-threading



Obr. 4. 4-jadrový procesor s hyper-threading

1) Pre GPU odstrániť tie časti CPU, ktoré umožňujú rýchle sequenčné výkonávanie inštrukcií.

2) Znížiť náročnosť riadenia inštrukcií vo viacerých ALU.

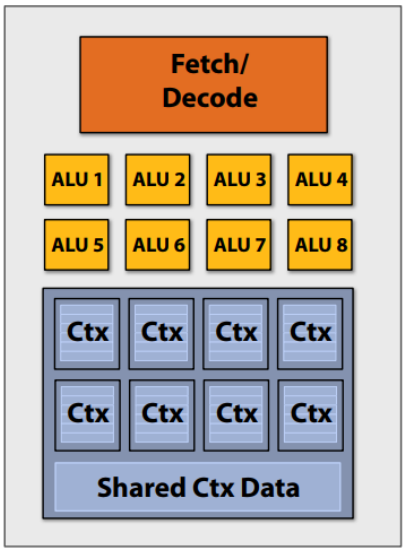
3) Predchádzať hazardom pomocou začatia vykonávania iného bloku.

Grafický procesor pozostáva z nasledujúcich častí [2]:

1) Globálna pamäť
2) Multi-procesor - ten sa skladá z:
   - Kontrolné jednotky
   - Registre
   - Výkonné jednotky
   - Cache pamäte

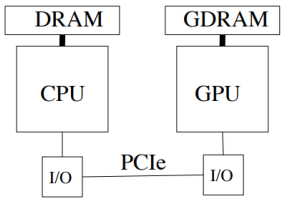
Programy, napísané pre grafické procesory, nevedia pristupovať k dátam uloženým v hlavnej pamäti [7]. Taktiež, GPU boli navrhnuté pre rýchle paralelné spracovanie dát a nepodporujú niektoré funkcie operačného systému. Z tohto dôvodu musia CPU a GPU procesory spolupracovať. Za vykonanie programu na grafike je zodpovedný CPU [2]. Ten má za úlohu:

- Prekopírovať dáta potrebné pre vykonanie programu z hlavnej pamäte do video pamäte.
- Vložiť program (GPU kernel) naprogramovaný programátorom na grafický procesor.



Obr. 5. Jedno jadro grafického procesora

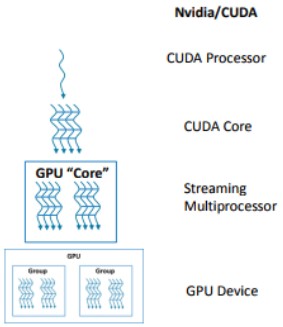
- Po dokončení a spracovaní dát nakopírovať späť výsledné dáta z video pamäte do hlavnej pamäte



Obr. 6. Spojenie CPU a GPU

C. Spracovanie dát na grafických procesoroch

Grafické procesory v jednom jadre spracovávajú dáta podľa modelu *SIMT*, čo v preklade znamená rozdelenie rovnakej, nezávislej práce na rovnaké synchronizované vlákna [12].



Obr. 7. Príklad spracovania na rôznych úrovniach GPU

III. Opis riešenia

Pre porovnanie časov vykonávania programu na CPU a GPU, sme sa rozhodli naprogramovať algoritmus SHA-1 v jazyku C pre CPU a v jazyku CUDA pre grafické procesory. Klasický algoritmus SHA však nie je paralelizovateľný, a preto sme implementovali jednu z jeho modifikácií pre paralelné spracovanie. Výber problému nie je veľmi vhodný vzhľadom na porovnanie architektúry procesorov, keďže daný problém nepozostáva zo zložitých aritmeticko-logických operácií.

A. Algoritmus SHA-1 a jeho modifikácia

Všeobecný algoritmus SHA pozostáva z nasledujúcich krokov:

1) Predspracovanie
   - Zarovnanie správy
   - Rozdelenie na bloky
   - Nastavenie počiatočného výsledku hash-u
2) Výpočet hash-u

Princíp fungovania SHA:

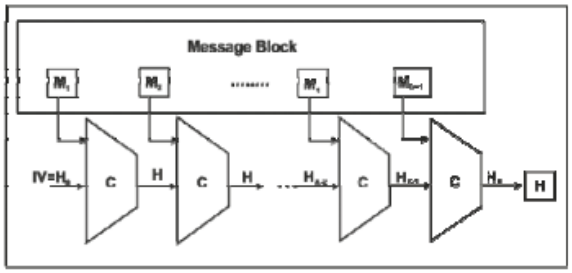
1) Zarovnanie správy pomocou nulových bajtov a doplnením veľkosti správy na koniec
2) Rozdelenie správy na bloky o veľkosti 64 bajtov
3) Nastavenie počiatočného výsledku hash-u
4) Vykonanie operácií a pripojenie medzivýsledku ku finálnemu hashu pre každý blok

Algorithm 1 SHA-1 [8]

function SHA($Message\ M$)
  padding(M) ▷ Put zero bytes into message and its size at the end
  Set H_0 ▷ Initial value of hash
  for each $block \in \mathcal{M}$ **do** ▷ 512 bit block size
    for $i = 0$ to 15 **do**
      $W_i = M_i$
    end for
    for $i = 16$ to 79 **do**
      $W_i = ROTL^1(W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16})$
    end for
    Set $a, b \dots e$ with $H_{0..4}^{i-1}$
    for $i = 0$ to 79 **do** ▷ f = Ch for i == 0 ... 19; Parity for i == 20 ... 39 & 60 ... 79; Maj for i == 40...59
      $T = ROTL^5(a) + f(b, c, d) + e + K_i + W_i)$
      $e = d$
      $d = c$
      $c = ROTL^{30}(b)$
      $b = a$
      $a = T$
    end for
  end for
  $H_{0..4}^{i-1} = a, b \dots e + H_{0..4}^{i-1}$
end function

Princíp fungovania modifikovanej SHA:

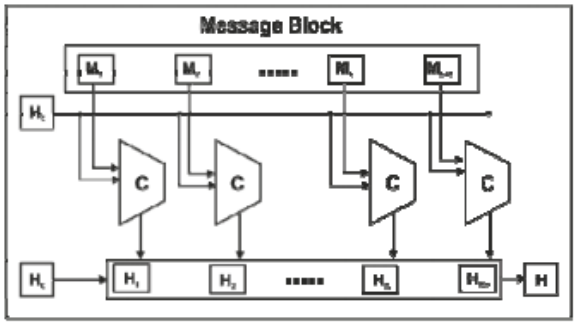
1) Rozdelenie správy na bloky o veľkosti 64 bajtov
2) Pre každý blok
   a) Nastavenie počiatočnej hodnoty
   b) Vykonanie operácií
   c) Pripojenie výsledku k počiatočnej hodnote
3) Spojenie výsledkov jednotlivých blokov
4) Ak veľkosť bloku nerovná sa 20 bajtov $\Longrightarrow$ pokračuj v bode 1.



Obr. 8. Grafické znázornenie klasického algoritmu SHA

Algorithm 2 Modified SHA-1 (pseudo code) [10]

function MSHA($Message\ M$)
  $padding(M)$ ▷ Put zero bytes into message and its size at the end
  $Blocks(M)$ ▷ Breaking message into 64 bytes blocks
  for each $block \in \mathcal{M}$ **do** ▷ Each block is independent
    $H_i = SHA(block)$
  end for
  $H+ = H_i$ ▷ + mean concatenate
  if (size(H) != 20B) **then** $MSHA(H)$
  else return
  end if
end function



Obr. 9. Grafické znázornenie modifikovaného algoritmu SHA

B. Vlastnosti SHA-1 a jeho modifikovanej verzie

Pre všetky hashovacie funkcie je dôležité spĺňať nasledujúce vlastnosti [9]:

- Ireverzibilnosť hashu (Preimage resistance) - pre daný hash je ťažké nájsť správu, po ktorej zahashovaní dostaneme pôvodný hash
- Odolnosť voči kolíziám - je ťažké nájsť také dve rôzne správy, ktorých výsledky hashu sa rovnajú

Daný modifikovaný algoritmus SHA-1 by mal spĺňať všetky vlastnosti hashovacích funkcií, viď článok [10].

C. Testovacie prostredie

Dané navrhnuté riešenie implementácie modifikovaného algoritmu SHA-1 bolo realizované na nasledujúcom technickom vybavení počítača (Hardware):

- Procesor: Intel Core i3-4100M 2.50GHz, 2 Core(s), 4 Logical Processor(s)
- Hlavná pamäť (RAM): 12.0 GB DDR3, 2.5GHz
- Grafický procesor: NVIDIA GeForce GT 740M, 2.0 GB DDR3

a programového vybavenia:

- Operačný systém: Windows 8.1 - 64-bit
- Vývojové prostredie:
  - Microsoft Visual Studio Enterprise 2015 v14.0.25431.01
  - Nsight Visual Studio Edition v5.4

D. Implementácia

Vyššie opísaný modifikovaný algoritmus SHA-1 sme implementovali v jazyku C a v jazyku CUDA. Pre porovnanie výpočtového výkonu daných procesorov sme sa rozhodli merať čas len funkcie, ktorá priamo vykonáva daný výpočet hashu. Čas potrebný na prenos dát z hlavnej pamäte do video pamäte a späť a čas potrebný pre načítanie súboru do pamäte sme nezáratavali do celkového času z dôvodu, že pri daných časoch by sme už neporovnávali samotný výkon procesorov, čo bolo našou úlohou.
Implementácia na CPU vyzerá rovnako ako opísaný algoritmus pomocou pseudo kódu vyššie v článku. Implementácia v jazyku CUDA pre grafické procesory je trochu odlišná. Rozdelenie správy do jednotlivých blokov je zabezpečené pomocou indexovania na grafickej karte.

Algorithm 3 Modified SHA-1 (pseudo code CUDA)

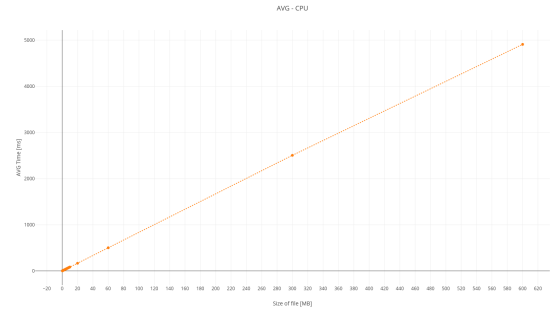
function CUDA($Message\ M$)
  $padding(M)$ ▷ Put zero bytes into message and its size at the end
  $copyToDevice(M)$ ▷ Copy message to video RAM
  $SHA <<< blocks, 512 >>> (m);$ ▷ blocks - number of message blocks, 1 block = 1 core GPU, 512 - number of threads for 1 block
  $H = copyToHost()$ ▷ Copy result to host from video RAM
  if (size(H) != 20B) **then** $cuda(H)$
  else return
  end if
end function

Meranie času na CPU bolo realizované pomocou knižnice $< sys\backslash timeb.h >$ a v jazyku CUDA sme použili $cudaEvent$ funckie cudaEventRecord() pre časové pečiatky vykonávania na GPU a cudaEventElapsedTime() pre vypočítanie rozdielu časov.

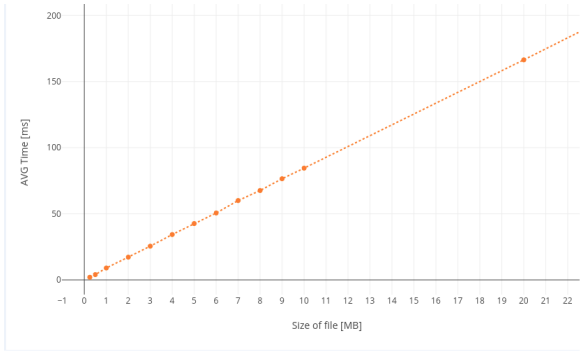
IV. Výsledky a grafické porovnanie

A. Provonanie implementácií

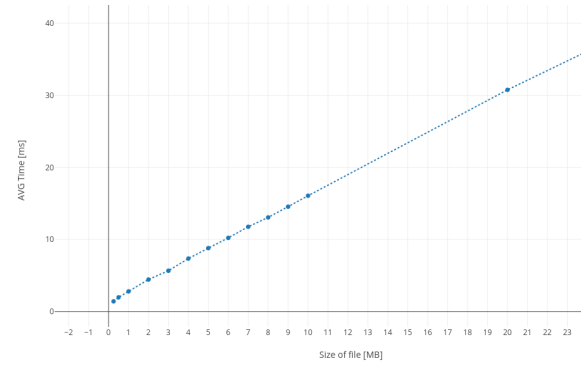
Dané implementácie sme navzájom porovnali na vzorke rôznych veľkosti súborov od 0,250MB až po 600MB. Dané merania pre každú veľkosť súboru sme uskutočnili 10 krát pre obe implementácie. Následne sme vypočítali priemer z meraní a zrýchlenie medzi CPU a GPU.



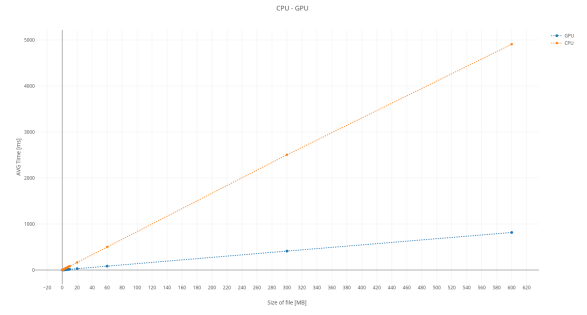
Obr. 10. Priemerný čas výpočtu MSHA-1 na CPU



Obr. 11. Priemerný čas výpočtu MSHA-1 na CPU (detailnejšie zobrazenie)



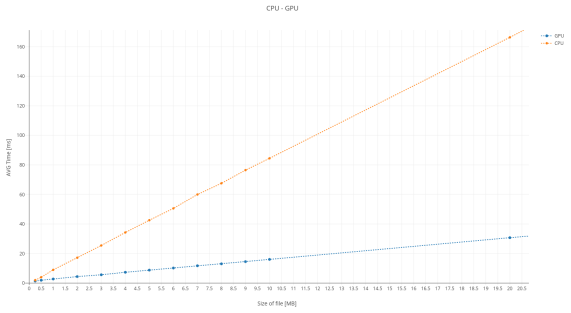
Obr. 12. Priemerný čas výpočtu MSHA-1 na GPU (detailnejšie zobrazenie)



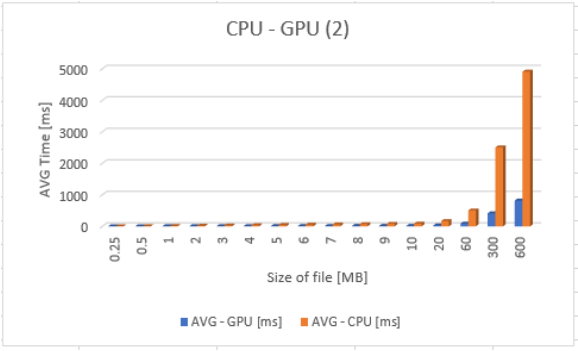
Obr. 13. Porovnanie CPU a GPU časov pre jednotlivé veľkosti súborov

B. *Porovnanie implementácie s reálnym nástrojom*

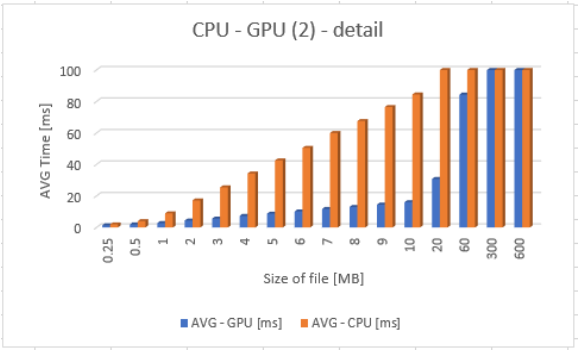
Implementáciu sme taktiež porovnali s reálnym nástrojom *sha1sum*, používaným na hashovanie súborov, dostupným pre



Obr. 14. Porovnanie CPU a GPU časov pre jednotlivé veľkosti súborov (detailnejšie zobrazenie)



Obr. 15. Porovnanie CPU a GPU časov pre jednotlivé veľkosti súborov

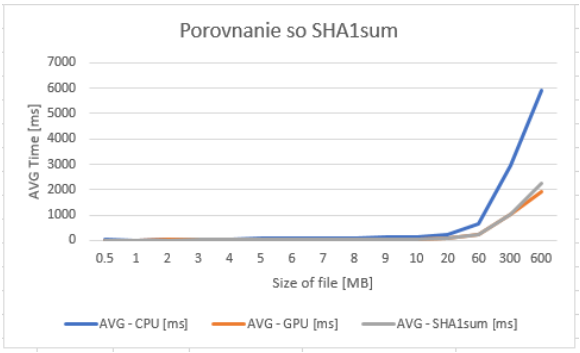


Obr. 16. Porovnanie CPU a GPU časov pre jednotlivé veľkosti súborov (detailnejšie zobrazenie)

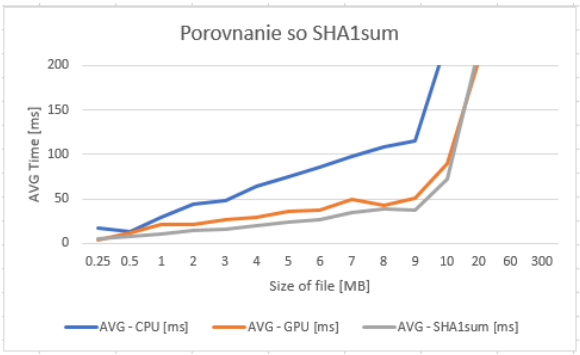
platformy Windows aj Linux. Do takéhoto porovnania sme započítali aj časy pre načítavanie súboru do pamäte. Nezarátavali sme však čas potrebný na presun dát z hlavnej pamäte do video pamäte. Pre odmeranie vykonávania hashu nástrojom sha1sum sme použili štandartne dostupný nástroj *time*. Dané merania sme taktiež vykonali na pre rôzne veľkosti ako v predchádzajúcom prípade.

V. Automatické využitie GPU

Viacjadrové procesory spolu s kompilátormi v dnešnej dobe podporujú funkcionalitu pre automatický paralelizmus nie len na úrovni inštrukcií, ale aj jednoduchšie cykly v kóde [11]. Pre zložitejší zdrojový kód programu bolo vymyslené *OpenMP*. To dokáže s minimálnym úsilím programátora vykonávať výpočtovo náročnú úlohu na viacerých jadrách procesora CPU.



Obr. 17. Porovnanie s nástrojom sha1sum



Obr. 18. Porovnanie s nástrojom sha1sum (detailnejšie zobrazenie)

Multijadrové procesory sa však ešte stále nevyrovnajú grafickým procesorom. Pre automatické paralelizované spracovanie na grafických procesoroch by kompilátor musel:

- nájsť nezávislé časti programu
- prekopírovať dáta z pamäte RAM do video pamäte a späť
- rozdeliť dáta pre jednotlivé bloky a thready GPU
- preložiť kód pre procesor GPU

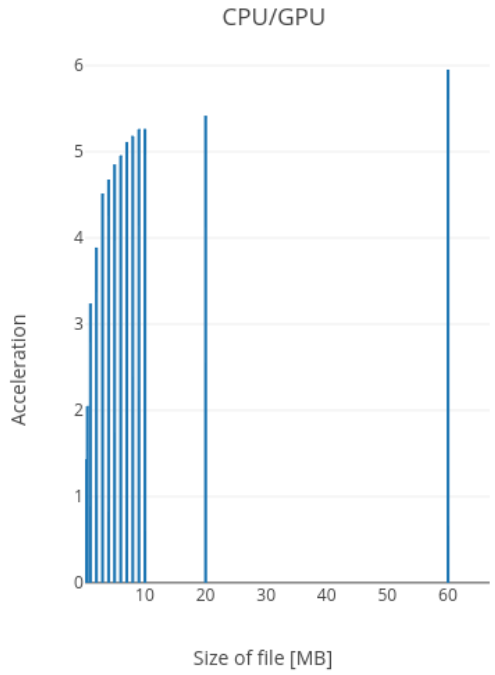
a zároveň by to muselo byť ovládané z CPU. Najnovšie grafické procesory však umožňujú spustiť väčší program (kernel), ktorý dokáže spúšťať menšie programy. To umožňuje spustenie bez potreby koordinácie CPU [2] a dosiahnutia tak vyššieho výkonu.

VI. Záver

V článku sme porovnali implementáciu modifikovanej SHA-1 pre paralelné spracovanie na CPU a GPU, kde sme dosiahli určité zrýchlenie (viď tabuľka I). Tieto implementácie sme taktiež porovnali aj s bežne používaným nástrojom sha1sum, kde naša implementácia algoritmu na CPU bola o čosi menej efektívna, čo odôvodňujeme optimalizáciou zdrojového kódu a trochu inou časovou náročnosťou klasickej a modifikovanej SHA-1. Implementáciou pre GPU sme sa však priblížili k hodnotám časov tohto nástroja. Môžme teda predpokladať, že pri použití zdrojového kódu z nástroja v našej implementácii na GPU, by sme zrýchlili výpočet hashu.

Tabuľka I. CPU vs. GPU

Size [MB]	AVG - GPU [ms]	AVG - CPU [ms]	CPU/GPU
0.25	1.4047648	2	1.423725879
0.5	1.958352	4	2.042533722
1	2.7868832	9	3.229414135
2	4.4332639	17.2	3.879760012
3	5.6588672	25.5	4.506202231
4	7.3489853	34.3	4.66731101
5	8.7965731	42.6	4.842794974
6	10.2195169	50.6	4.95131037
7	11.7593665	60	5.102315673
8	13.0595455	67.6	5.1762904
9	14.5545724	76.5	5.256080213
10	16.0727678	84.5	5.257339685
20	30.7733154	166.4	5.407282181
60	84.3981097	501.7	5.944445933
300	411.9776641	2504.7	6.079698533



Obr. 19. Zobrazenie zrýchlenia implementácie medzi CPU a GPU

Literatúra

[1] HENNESSY, John L.; PATTERSON, David A. Computer architecture: a quantitative approach. Elsevier, 2007.

[2] TATOURIAN, Alan. Nvidia gpu architecture and cuda programming environment. URL http://code. msdn. microsoft. com/windowsapps/NVIDIA-GPU-Architecture-45c11e6d, 2013.

[3] PRECOMPUTATION, Speculative. Hyper-Threading Technology.

[4] Intel Pentium 4 3.06 GHz with Hyper-Threading support, http://ixbtlabs. com/articles2/pentium43ghzht/, Accessed: 12.11.2017.

[5] How a GPU Works - Kayvon Fatahalian 15-462 (Fall 2011), https://www. cs.cmu.edu/afs/cs/academic/class/15462-f11/www/lec_slides/lec19.pdf, Accessed:12.11.2017.

[6] Differences between physical CPU vs logical CPU vs Core vs Thread vs Socket, http://www.daniloaz.com/en/differences-between-physical-cpu-vs-logical-cpu-vs-core-vs-thread-vs-socket/, Accessed: 12.11.2017.

[7] The Continuing Importance of GPUs For More Than Just Pretty Pictures - Jeff Rowe, https://www10.mcadcafe.com/blogs/jeffrowe/2017/03/16/the-continuing-importance-of-gpus-for-more-than-just-pretty-pictures/, Accessed:12.11.2017

[8] GALLAGHER, Patrick; DIRECTOR, Acting. Secure hash standard (shs). FIPS PUB, 1995, 180-3.

[9] ROGAWAY, Phillip; SHRIMPTON, Thomas. Cryptographic hash-

function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In: International Workshop on Fast Software Encryption. Springer, Berlin, Heidelberg, 2004. p. 371-388.

[10] KISHORE, Neha; KAPOOR, Bhanu. An efficient parallel algorithm for hash computation in security and forensics applications. In: Advance Computing Conference (IACC), 2014 IEEE International. IEEE, 2014. p. 873-877.

[11] Automatic Parallelization with Intel® Compilers, Intel, November 2011, https://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers, Accessed: 17.11.2017

[12] GPU Architectures, https://courses.cs.washington.edu/courses/cse471/13sp/lectures/GPUsStudents.pdf, Accessed: 21.11.2017