NI-GPU Finální zpráva

Zadání (z mailu):
Dobrý den,

Chtěl bych zvolit vlastní téma semestrální práce, pokud ještě takovou možnost mám:

Chtěl bych zkusit využit znalosti, které dostanu z NI-GPU v sondě https://github.com/CESNET/ipfixprobe. Akorát zatím jsem neměl žádné zkušenosti s programováním grafiky, a tudíž nevím, jak to budu moct využit. Zatím předpokládám, že to bude buď nějaké paralelní parsování paketů nebo paralelní vkládání do paměti. Od svého kolegy na CESNETu jsem dozvěděl, že mají nějaký server s grafikou, takže ve finále by tam mohlo být nějaké porovnání původní a nové implementací na provozu generovaném ze síťového profilu(https://github.com/CESNET/FlowTest).

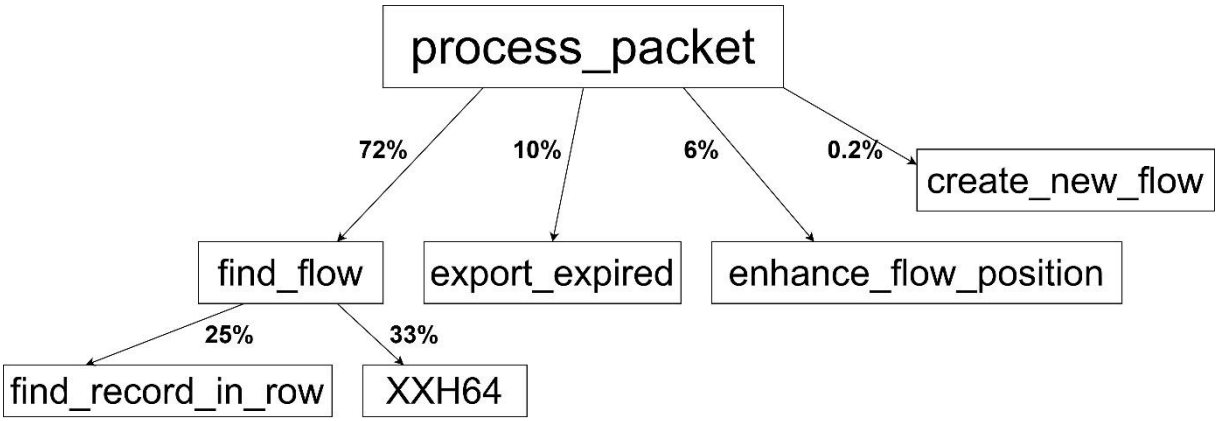
Neměl jsem možnost být na prvním cviku, takže omlouvám se, pokud to takhle vůbec nejde.

Ve své úloze semetrálce jsem se snažil použít možnosti grafické karty k urychlení práce síťového analyzátoru **ipfixprobe.** Samotný analyzátor je velký program a má širokou funkcionalitu, ale jeho hlavní činnost spočívá ve vytváření statistik síťových toků. Tok je pojem ze standardu IPFIX, který se definuje jako množina paketů se stejnými vlastnostmi zaznamenaných na určitém observačním bodě během určitého časového intervalu. Za tyto vlastnosti se pokládá zdrojová a cílová ip adresa, zdrojový a cílový port, protokol z ip hlavičky (a občas i vlan id z ethernet headeru). **Ipfixprobe** musí pro každý paket určit tok, do kterého tento paket náleží a následně statistiky toku obnovit. Původní implementace je čistě sekvenční, a pro každý paket se vyvolává parser, který se snaží vyparsovat zajímavé hodnoty z paketu(ip adresy, porty..), následně pro tento tok se vytváří 2 klíče – dopředný a zpětný, protože nevíme, jestli ten tok v paměti máme jako tok z uzlu A na uzel B, nebo B → A. Následně se tento klíč hashuje, a podle této hodnoty se koná vyhledávání v hash tabulce.

Setup:

Veškeré testování jsem prováděl na serveru **netmonstorage** z cesnetí sítě. Server má grafickou kartu **Tesla P100-PCIE-16GB.** Původně jsem chtěl posílat pakety z jiného stroje přes nfb karty (speciální zařízení pro posílaní paketů ve velkých rychlostech – do 100gbps). Ale **netmonstorage** takovou kartu nemá, takže měl jsem číst pakety pcap souboru. Tento přístup má velkou nevýhodu v tom, že ipfixprobe začíná ztrácet více času čtením dat z disku než samotným zpracováváním paketů. Profilování ukazuje, že cca 60% času je načítaní paketů ze souboru a parsování (při čtení ze síťového rozhraní je ta

hodnota mnohém menší – cca 30%), a pak 30% je hashování paketů a vkládání do tabulky. Ze schématu dolu je vidět, že funkce **process_packet**, což je hlavní funkce na vkládání rozparsovaného paketu do tabulky, stráví třetinu času ve funkci XXH64, což je hashovací funkce.



Pro testy sondy jsem použil stejný postup, který se používá na cesnetu při ohodnocování nových verzi sondy a veškeré testy jsem prováděl na profilu sítě – pcap soubor, vygenerovaný takovým způsobem, aby svým charakterem nejvíc odpovídal živému provozu. Konkrétně byl použít dopolední provoz během pracovního dne. Rozměr výsledného pcapu – 20gb.

Návrh vylepšení:

Jelikož pro síťové vstupní pluginy je typicky, že pakety se načítají po „burstech“ – větších celcích, a zároveň parsování jednotlivých paketů jsou na sebe vzájemně nezávislé operace, zdá se vhodným parsovat všechny pakety současně.

Když jsou pakety rozparsováné, musíme z nej vytvořit hash, což je taky nezávislá operace.

Implementace:

Jelikož ipfixprobe je velký projekt s komplikovaným build systémem s velkým množstvím závislostí, nejde celý projekt zbuildit nvidia kompilátorem. Veškerý cuda kód musí být zkompilován jako dynamická knihovna a slinkován s původním programem. Vytvořené funkce jsou velké, takže to, že se nepoužil inlining by nemělo vadit.

Úprava parsingu:

Během úpravy parseru jsem narazil na následující problémy:

1) Měl jsem vyřešit transfer dat na GPU, jelikož data paketu jsou původně v CPU paměti. K tomuto jsem opravil původní alokace, aby veškeré buffery pro pakety se

alokovali přes `cudaHostAlloc(&pkts, sizeof(Packet) * pkts_size, cudaHostAllocMapped);` Samotný buffer pro data paketů je alokovaný podobně. Problém je ten, že pakety jsou sami sobě velké – desítky kilobajtů. A proto nemůžeme alokovat buffery pro celé pakety. Vyhnut se tomu dalo tím, že pro parsing v podstatě potřebujeme jenom začátek paketu. Ve svých experimentech jsem vyzkoušel 256 bajtů, a je toto dostatečné číslo pro parsování všech paketů z testového pcapů. Kvůli tomu, jak funguje knihovna libpcap, musím sekvenčně kopírovat data paketů sekvenčně z libpcap bufferu do společného CPU-GPU bufferu

2) Cuda nepodporuje výjimky, což je součást původního parseru, proto mělo se to přepsat bez výjimek
3) Původní program aktivně používá bitfieldy, což se beze žádného warningu kompiluje v cudě, ale padá v runtimu pří přístupu
4) Hodnoty v paketu, jako třeba ip4 adresy, nejsou z pohledu paměťového prostoru počítače zarovnány, tudíž přístup k nezarovnané hodnotě na CPU vede na zpomalení, ale na GPU je to pád vlákna.

Úprava hashování:

1) Původní hashovací funkce je rychlá, a patří ke třídě nejrychlejších nekryptografických hashovacích funkcí. Zároveň má dobré statistické vlastnosti – zejména rozptyl. Má moc optimální implementace, ale s využitím intrisiků, které na GPU nejsou, takže nelze tuto funkce použít. Nepodařilo se mi najít nějakou optimální nekryprografickou hash funkce, proto jsem otestoval více možnosti
2) Úprava je podobná jak u parsování, každé GPU vlákno musí nakopírovat data z rozparsovaného paketu do souvislého bufferu, na kterém je následně volaná hashovací funkce.

Testování:

Referenční implementace – původní sekvenční implementace. Úspěšnost řešení se měří celkovým časem, který program spotřeboval k načtení celého pcapu.

Reference	25s
První GPU implementace	1m 47s

Je vidět, že první verze je hodně pomalá. První věc, kterou budu testovat, je kolik paketů současně se vyplatí parsovat, tedy optimální rozměr síťového busrtu.

64(původní)	1m 47s
128	1m 28s

256	1m 15s
512	1m 9s
1024	59s
2048	1m7s
4096	1m 5s

Zdá se zvyšovat rozměr busrtu dál nemá smysl, a optimální hodnota je vedle 1024.
Dále jsem zkoušel různé rozměry bloku pří spuštění kernelu, bylo vyzkoušeno 128, 256, 512, 1024. Z výsledků neplyne, že to má na výsledek velký efekt.

Následně jsem rozhodnul rozdělit proces vytváření dvou hashe na 2 časti, tedy aby jednu hash vytvářelo jenom jedno vlákno.

2 hashe v jednom vlákně	59s
1 hash v každém vlákně	55s

Pak jsem rozhodnul zmenšit rozměr nakopírovaného paketu, ukázalo se, že 128 bajtů je dostatečně pro pasování všech paketů.

256 bajtů	59s
128 bajtů	41s

Následně jsem vyzkoušel různé hashovací funkce, které by se dalo rozdělit na 3 skupiny: triviální sekvenční, pokročilé sekvenční a SIMD-style. Triviálně sekvenční funkce zpracovávají buffer postupně ve smyčce, pokročile sekvenční – podobně jako triviální, ale zkonstruované lépe a využívající se v praxi – v této skupině je jenom superfasthash. SIMD-style skupina se snaží zpracovávat celý buffer najedenou, bez smyček. Výsledky jsou podobné, ale superfasthash ukazuje nejlepší.

Triviální sekvenční	41s
superfasthash	35s
SIMD-style	42s

Taky jsem vyzkoušel načítat data paketu do lokálního bufferu před parsingem, ale tento přístup neměl úspěch, čas zpracovávání se jenom zvýšil.

Závěr:

Nepodařilo se mi zrychlit ipfixprobe, pravděpodobně tato úloha není úplně vhodná pro GPU multithreading. Za hlavní problém považuji intenzivní komunikace s paměti a nevhodné sekvenční kopírování paketů do sdílené

paměti, jelikož jedná z nejdůležitějších vlastností původní implementaci je *zero-copy*, což prakticky znamená, že příchozí pakety se nekopíruji. Taky byl vidět nedostatek kvalitních hash funkcí pro GPU. Pravděpodobně bych měl se nějakým způsobem vyhnut pravidelnému spuštění kernelu s každým burstem a využít něco jako gpu thread-pool.