HADMÉRNÖK

IX. Évfolyam 1. szám - 2014. március

Sergyán Szabolcs – Szénási Sándor - Vámossy Zoltán
sergyan.szabolcs@nik.uni-obuda.hu - szenasi.sandor@nik.uni-obuda.hu - vamossy.zoltan@nik.uni-obuda.hu

A GRAFIKUS HARDVEREN (GPGPU) IMPLEMENTÁLT ALKALMAZÁSOK SEBEZHETŐSÉGEI

Absztrakt

A grafikus kártyák eredetileg a képernyőn látható kép megjelenítéséért feleltek, az utóbbi évtizedben azonban ez a szerep jelentősen megváltozott, egyre több funkció végrehajtására lettek alkalmasak. Az első 3D gyorsítókártyák megjelenése (majd integrálása) jelentősen átalakította a GPU-k (grafikus vezérlő egységek) piacát. Napjainkban pedig a GPGPU (általános célú grafikus vezérlő egység) programozás már egyre inkább általánosnak tekinthető, különösen a nagy számításigényű alkalmazások területén. Míg azonban a játékprogramok és a kezdeti kutatások során a biztonság kérdése nem jelentett különösebb problémát, napjainkban már számos GPGPU alkalmazás dolgozik érzékeny (személyes, üzleti, állami) adatokkal. Emiatt érdemes foglalkoznunk az ezen a területen megjelenő biztonsági résekkel és lehetséges támadási módszerekkel.

Traditionally, graphics cards were responsible for the visualization of the content in the screen; however these devices have more and more tasks in the last few decades. The appearance of the first 3D accelerator cards changes the video adapter industry, in the next few years these new functions had been integrated into the GPUs (Graphical Processing Units).Nowadays GPGPU (General Purpose Graphical Processing Unit) programming becomes more and more general, especially in the field of High Performance Computing. In the first time, in case of games and initial research projects, data security was not an important factor. However nowadays, there are several GPGPU applications working with sensitive (personal, business, governmental) data. This paper deals with several questions of possible security holes and attack methods.

Kulcsszavak: *GPGPU, grafikus kártya, CUDA, biztonság ~ GPGPU, graphics card, CUDA, security*

BEVEZETÉS

A grafikus kártyák eredetileg a képernyő memóriában (karakteres és grafikus) található adatoknak a monitoron való megjelenítéséért feleltek, a 90-es évek közepén azonban ez fokozatosan kibővült számos egyéb funkcióval. Ezek közül a leglátványosabb és legismertebb a 3 dimenziós megjelenítés gyorsítása. Számos indok segítette elő ezen funkció kialakulását, egyrészt akkoriban egyre inkább elterjedtek a 3D grafikát tartalmazó játék és tervezőprogramok [1], másrészt jól látható, hogy a különböző alkalmazások 3D megjelenítési moduljai egymáshoz nagyon hasonlóak (még akkor is, ha maguk a programok teljesen különbözőek is voltak), illetve ez a funkció nagyon jól párhuzamosítható. Így először külön, majd integrált megoldásként megjelent ennek hardveres gyorsítása. A CPU-khoz hasonlóan itt is megjelentek az egyre több végrehajtóegységgel bíró változatok, hiszen a jól párhuzamosítható 3D megjelenítést jól lehetett gyorsítani a többmagos eszközökkel [2].

Az első generációkban ezek a kártyák különálló, egymástól jelentősen különböző alkatrészeket tartalmaztak a különféle feladatok végrehajtásához: vertex shaderek végezték a 3D-2D leképezést, pixel shaderek végezték az árnyékolást, textúrázást, illetve a geometry shaderek végezték a modellek módosításával kapcsolatos teendőket. A gyakorlatban azonban kiderült, hogy nagyon nehéz ezen eszközök megfelelő arányainak megtalálása, hiszen feladatonként nagyon változó, hogy hány darab vertex, illetve pixel shaderre van szükség. A több éves fejlődés folyamán a megoldást végül az jelentette, hogy ezek az eszközök az egyes generációk során egyre közelebb kerültek egymáshoz, majd végül megjelent az úgynevezett unified shader model, ahol már elmondhatjuk, hogy tulajdonképpen azonos típusú, általános célú eszközök helyezkednek el a kártyákon, amelyek mindegyik feladat végrehajtására alkalmasak.

Ezzel pedig elérkezett az idő, hogy már nem csak grafikai, hanem tetszőleges alkalmazásokat lehetett futtatni a grafikus kártyákon. A GPGPU program legfontosabb célja a minél nagyobb teljesítmény elérése, illetve az ennek leginkább megfelelő modell kidolgozása [3]. Ennek érdekében pedig a fejlesztők gyakran minden más tényezőt figyelmen kívül hagytak, köztük a biztonságot is. Az esetek jelentős részében (pl. játékprogramok esetén) ez nem jelent problémát, de mivel egyre több ipari alkalmazásban jelennek meg ezek az alkalmazások (pl. bankok, elemző és biztonságtechnikai cégek), mindenképpen érdemes részletesebben foglalkozni a biztonságpolitikai kérdésekkel is [4].

NEM SZÁNDÉKOS HIBALEHETŐSÉGEK

Szoftver biztonság területén általában elsőre mindenki a külső támadásokra gondol, azonban legalább ennyire fontos az is, hogy az egyes programok mennyire képesek a stabil működésre. Ez jelentős részben a programfejlesztői környezeten múlik, hogy mennyire tudják a fejlesztőket segíteni abban, hogy hibátlanul működő kódokat készítsenek [5].

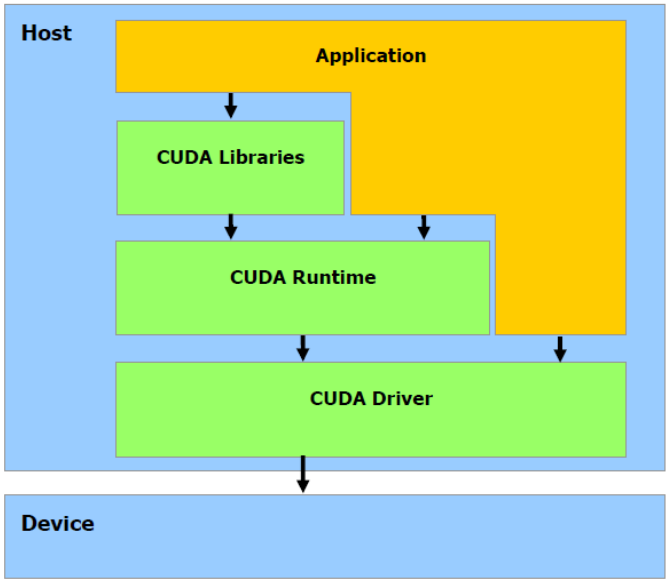
Hardver és szoftver hiányosságok

A GPGPU-ra való fejlesztés napjainkban már egy hétköznapi tevékenységnek számít, a fejlesztőeszközök azonban még nem tudták elérni azt a szintet, amit a hagyományos nyelveknél megszokhattunk. Rendelkezésre állnak ugyan a szükséges eszközök (fordítóprogram, nyomkövető program, teljesítménymérő program), ezek azonban nem mindig teljesen megbízhatók, és általában csak korlátozott funkciókkal bírnak (ami részben persze természetes, hiszen pl. a kód optimalizálás területén a C fordítóknak több évtized előnyük van).

A programkód felépítésénél ugyan rendelkezésre állnak már a modern programozás eszközei, akár objektum-orientált programok készítésére is lehetőség van a grafikus kártyán, ezek azonban még mindig nem terjedtek el teljesen. Maga a kódolás sok szempontból egy több

évtizeddel ezelőtti stílust idéz (globális változók, mutatók intenzív használata, kézi memória foglalás és felszabadítás, optimalizálás az áttekinthetetlenség árán is).

Mindezek a jellemzők azzal a káros következménnyel járnak, hogy a GPGPU kódok gyakran jóval megbízhatatlanabbak, mint hagyományos társaik. Érdemes megfigyelni a modern programozási nyelvek fejlődését (Java/C#), amelyeknél alapvető tervezési szempont volt a biztonságos kód készítésének lehetősége (szemétgyűjtő mechanizmus, mutatók elhagyása, OOP alapok, fejlett kivételkezelési technikák stb.), ezek azonban a grafikus kártya kódoknál egyelőre még hiányoznak.



1. ábra. A grafikus kártya elérésének szintjei.

A magasabb szintek az alattuk lévőkön keresztül férnek hozzá a tényleges eszközhöz.[6]

Grafikus kártya, mint fekete doboz

A grafikus kártyák programozása során számos olyan lépés jelenik meg, amelyeket a programozó nem tud pontosan befolyásolni, így az általa készített, majd továbbított programkód se lehet mindig teljesen megbízható. A felsorolt problémák között talán ezt tekinthetjük a legenyhébbnek, de mindenképpen érdemes említést tenni róla.

Ezen a területen még kevésbé terjedtek el a teljesen ingyenesen elérhető fordítóprogramok, osztálykönyvtárak, amelyekre bármikor lehet alapozni, hiszen a forráskód elérhető, bármikor áttanulmányozható. Egy kritikus rendszer fejlesztése során ez alapvető szempont lehet, és ezt a napjainkban legelterjedtebb CUDA programozási környezet nem teljesíti. Az Nvidia által készített eszközök, fordítóprogramok, osztálykönyvtárak nagyon jól használhatók, azonban azok pontos működése nem teljesen áttekinthető. A program írásához számos szintet igénybe vehet a fejlesztő, de mindig lesznek olyan területek, ahol nincs teljes átlátása a folyamat felett (Ábra 1.).

Hasonló problémát jelent a program futtatása is. A GPGPU kódok nem közvetlenül az operációs rendszer által kerülnek végrehajtásra, hanem az egész folyamatban lényeges szerepet játszik a grafikus kártya meghajtó program. Az indítandó kernelek ugyanis mindig ezen keresztül jutnak el a GPU-ra, és mindez megint csak felvet biztonsági problémákat. A szándékos károkozás mellett érdemes kitérni a hibalehetőségre is, ugyanis maguk a szerzők is találkoztak már olyan esetekkel, amikor egy egyébként tökéletes(nek tűnő) program egy meghajtó program frissítése után már hibás adatokat szolgáltatott.

Indeterminisztikus végrehajtás

Számos definíció létezik az algoritmus fogalmára, alapvetően az alábbi tulajdonságokat tekintjük mérvadónak: egy függvény kiszámítására szolgáló, véges számú, jól definiált utasítás sorozat; amely egy kezdőállapotból indul, az utasítások meghatározzák a végrehajtás menetét, majd véges számú lépést követően megáll a végső állapotban. Általában azt feltételezzük, hogy az egyik állapotból a másikba való átlépés determinisztikus, alapvető, hogy az algoritmus nem tartalmazhat nem pontosan definiált utasításokat. Ha egy utasításnak mégis több lehetséges kimenete van, akkor is önkényesen választanunk kell ezek közül egyet.

A fentiekből kiindulva, általában azt feltételezhetjük, hogy egy algoritmus mindig ugyanazt a kimenetet fogja visszaadni azonos bemeneti adatok esetében (még abban az esetben is, ha létezik több érvényes megoldás) [7]. Ez általában igaz is a tradicionális szekvenciális algoritmusokra, a többszálú megvalósítások esetében azonban már jóval összetettebb lehet a helyzet. A szálak pontos ütemezése ugyanis már nem a programozón múlik, hanem mindez futásidőben, a processzor aktuális terhelésének megfelelően történik meg (amit számos, az aktuálisan vizsgált programon kívüli körülmény is befolyásolhat). Természetesen írhatunk olyan programokat, amelyek kiküszöbölik ezeket az ütemezési különbségeket, de ez általában csak a teljesítmény nagyon erőteljes leromlásával együtt valósítható meg.

Érdemes persze azt is megjegyezni, hogy ezek a programok nem feltétlenül rosszabbak, mint a hagyományos programkódok. Elképzelhető, hogy az alkalmazás minden egyes futtatás során más-más eredményt ad vissza, ez azonban nem jelenti azt, hogy az eredmény hibás. Gondoljunk csak egy egyszerű példára: adjuk vissza egy szám valamelyik osztóját. Abban az esetben, ha több osztó is van, akkor több megoldás is elképzelhető. Szekvenciális algoritmusok esetében célszerűen egy egyszerű ciklussal megpróbáljuk megkeresni az első valódi osztót, míg párhuzamos algoritmusok esetén esetleg megpróbáljuk felbontani a lehetséges osztókat tartalmazó intervallumot több kisebb részre, és ezeken belül egy időben több processzorral is keresünk. Fontos különbség, hogy az első esetben a program minden egyes alkalommal ugyanazt a legkisebb osztót fogja visszaadni, míg a második esetben elképzelhető, hogy minden futás során mást, attól függően, hogy éppen milyen ütemezéssel futottak le a konkurens szálak. Azonban lényeges megjegyezni, hogy a feladat eredménye szempontjából a második eredmény semmivel se tekinthető rosszabbnak, mint az első.

A gyakorlatban azonban mégis jóval több problémát okoz egy ilyen nem determinisztikus algoritmus, aminek a legfőbb oka, hogy meglehetősen nehéz tesztelni. A tesztelés során általában elvárjuk, hogy ha egy megadott bemenetre a program hibásan válaszolt (vagy nem válaszolt), akkor újabb futtatásokkal tudjuk reprodukálni ezt a hibát. Alapvetően ezen a tényen alapulnak a különféle hibakereső eszközök is (töréspont, nyomkövetés). Mivel a GPU alkalmazásoknál előfordulhat, hogy ugyanazokra az adatokra egyszer jó, egyszer pedig hibás adatot ad válaszként a rendszer, ez jelentősen megnehezíti a biztonságos, garantáltan jól működő alkalmazások fejlesztését.

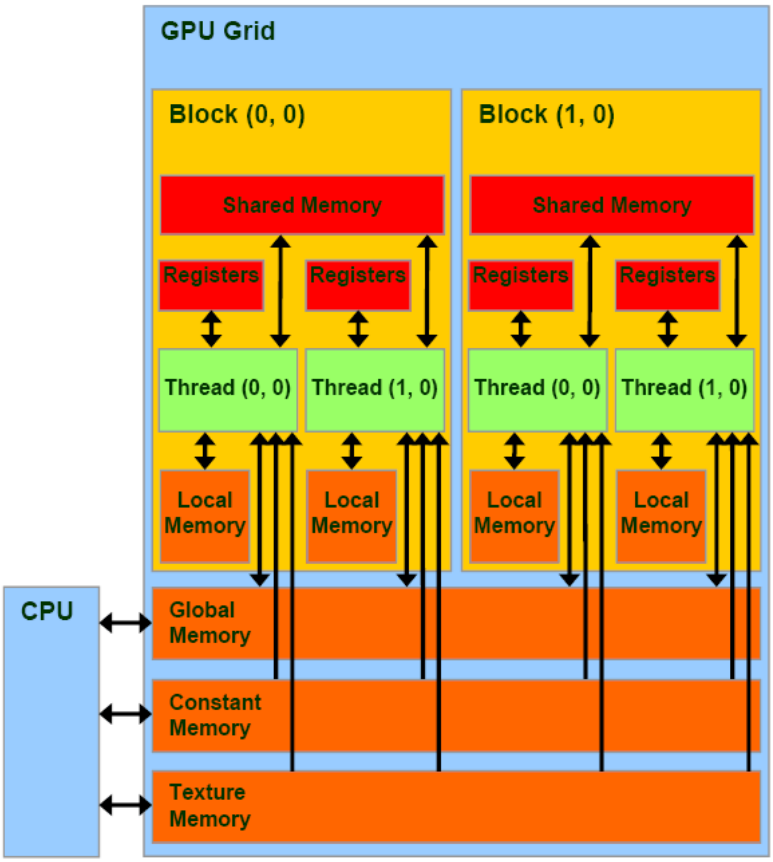
A tesztelés mellett ugyanis a hibakeresés ugyanilyen problémákat jelenthet. Még ha találunk is egy olyan bemenetet, amely hibás kimenetet okoz, akkor is meglehetősen nehézkes lehet a hiba helyének megkeresése. A klasszikus nyomkövető eszközök ugyan már rendelkezésre állnak többszálú környezetekben is, azonban semmi se garantálja, hogy a nyomkövetés során pont ugyanazt az eredményt adja majd az algoritmus, és így magát a hiba keletkezésének helyét is megtaláljuk majd. Arról nem is beszélve, hogy magának a nyomkövetésnek a használata már önmagában is hatással van a rendszer működésére, tehát lehet, hogy egészen más eredményekhez juthatunk a segítségével.

SZÁNDÉKOS TÁMADÁSOK

A fentieket ugyan meglehetősen kellemetlen, de mégis kezelhető problémáknak tekinthetjük, amelyeket kellő odafigyeléssel és időráfordítással meglehetős biztonsággal kezelhetünk. Ezeknél jóval veszélyesebbek a szándékos támadások, hiszen azok lehetnek egészen váratlanok, illetve jellegükből adódóan nehezen felderíthetőek. Maga a GPU a támadások eszköze is lehet [8], de egyben a támadás célpontja is, mi csak ez utóbbi lehetőséggel foglalkozunk most.

Memory leaking – globális memória

Alapvető biztonsági követelmény, hogy ha egy számítógépen (multitaszk környezetben) egyidőben több alkalmazást futtatunk, akkor azok egymás adataihoz ne férjenek hozzá (hacsak nem ez a kívánt állapot). Természetesen ugyanez igaz a grafikus kártyákra is, hiszen nyilvánvaló biztonsági kockázatot [9] jelent, ha pl. egy GPGPU alapú titkosító program által lefoglalt memória régiókhoz hozzáférhetnek más programok is, ezzel még a titkosítás előtt el tudják lopni az értékes információkat.



2. ábra. CUDA memória hierarchia [10]

A probléma megoldását a *process isolation* jelenti, amely garantálni tudja, hogy egy processz ne tudjon hozzáférni más processzek munkájához. A gyakorlatban ez számos szoftver és hardver alapú módszer összehangolását foglalja magában. Magát az izolációt különféle szinteken tudjuk megvalósítani, lehet akár hardveres, operációs rendszer szintű, vagy akár alkalmazás szintű is (pl. egy böngésző esetén lényeges, hogy egy esetleges kártékony oldalt megjelenítő ablak ne férhessen hozzá egy másik oldalon megnyitott banki információkhoz).

GPU-k esetében a különféle technikák még meglehetősen kezdetlegesnek tekinthetők, bár megjelentek előrelépések aziránt, hogy az egyes GPU alkalmazások ilyen téren is biztonságosak legyenek, azonban ez még mindig számos támadási pontot rejt magában.

Az egyik támadási lehetőség a GPU globális memóriájának elérhetőségében rejlik (Ábra 2.). Különféle tesztek segítségével nagyon könnyen ellenőrizhető, hogy amennyiben egy CUDA alkalmazással lefoglalunk, használunk, majd felszabadítunk memóriát, akkor a következő, az előzővel paramétereiben megegyező memóriafoglalás nagy valószínűséggel ugyanazt a GPU memóriaterületet fogja majd visszaadni. Mivel a memória felszabadítás során nem történik meg az érintett területek törlése (nullákkal való feltöltése), így a következő lefoglaláskor az előző alkalmazás által hátrahagyott memóriatartalom érintetlenül olvasható.

Ebben az esetben szerencsére egy egészen egyszerű megoldással sokat lehet tenni, a kernel futását követően, még a kilépés előtt (de mindenképpen a memória felszabadítása előtt) érdemes nullákkal feltölteni a használt memóriaterületet. A CUDA környezet természetesen csak akkor tudja odaadni az előzőleg használt memóriaterületet egy következő alkalmazásnak, amikor az előző azt már felszabadította, ennek köszönhetően a kernel futása közben ettől a támadástól még nem kell tartani, a kernel leállításakor kitörölt memóriatartalomnak köszönhetően pedig utána sem jelent veszélyt.

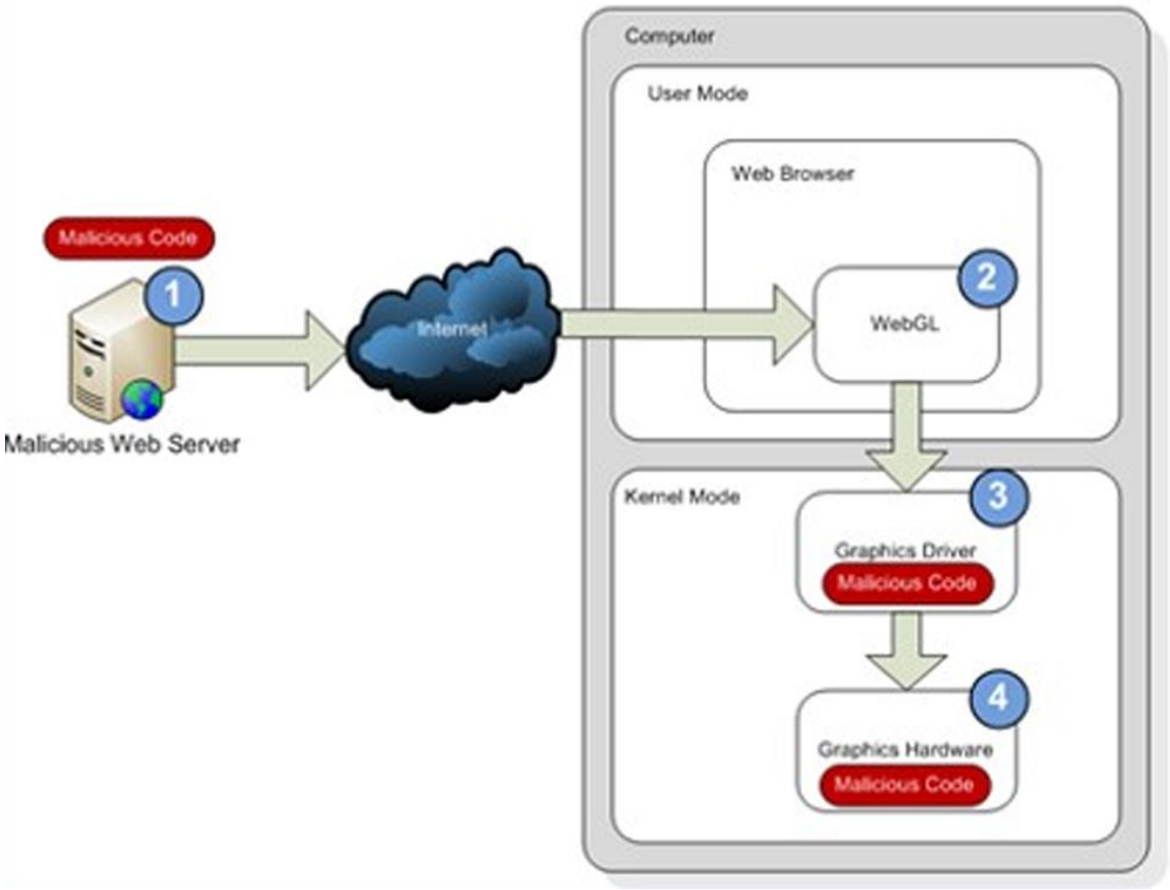
Memory leaking – megosztott memória

A GPU-k meglehetősen összetett memória hierarchiával rendelkeznek (Ábra 2.). A legnagyobb területet az úgynevezett globális memória teszi ki, amely elérhető mind a CPU, mind pedig a GPU számára. A GPU-k emellett rendelkeznek egy úgynevezett megosztott memória (shared memory) területtel is, ami valamilyen szinten megfelel a CPU-k esetében jól ismert gyorsítótárnak, habár működésük jelentősen különböző.

A megosztott memóriában tárolt adatok jelenleg szintén nem tekinthetők teljesen biztonságosnak. A GPU működéséből az ütemező ugyanazokat a végrehajtó-egységeket egymást követően más-más kernelek futtatására jelölheti ki. Ezek a kernelek érkezhetnek különböző kontextusból is, és amennyiben nem történt meg a megosztott memória törlése, a kontextusváltátást követően az aktuálisan futó szálnak lehetősége nyílik arra, hogy hozzáférjen az előző szálak által használt adatokhoz.

Ahogy Roberto Di Pietro et al. [11] mintaprogramok segítségével is meg tudta mutatni, ennek az a negatív hatása, hogy a káros programok az ilyen kontextusváltásokat követően is hozzáférnek az előző alkalmazások adataihoz.

A globális memóriában látható hasonló eseményekhez viszonyítva ez számos pozitív és negatív tulajdonsággal bír. A pozitív, hogy a megosztott memória a CPU számára teljesen láthatatlan, az csak a GPU kerneleken keresztül érhető el. Így az itt eltárolt adatok kiolvasásához is GPU kernelek futtatására van szükség. A jelenség meglehetősen nagy hátránya azonban az, hogy a GPU szálak számára a kiolvasás megoldható, és ami még kockázatosabb, hogy mindez futásidőben történik, tehát a kernel leállításakor történő törlés már nem elégséges, a kiolvasás addigra esetleg már rég megtörtént (érdemes egyébként megjegyezni, hogy a kernel leállításakor a megosztott memóriában a GPU meghajtó eleve végrehajt egy törlést, tehát ezt nem is szükséges explicit módon kiadni).



3. ábra. WebGL támadás vázlata. 1) A felhasználó megtekint egy weboldalt, ahol WebGL script található 2) a WebGL-en keresztül ez feltölti a szükséges kódokat a grafikus kártyára 3) ez a kód tartalmazhat ártó szándékú kódokat 4) ezen keresztül magát a grafikus hardvert lehet támadni (pl. lefagyasztani)[12]

DOS támadások

A Denial-of-Service támadás már mindenki számára ismerős, habár elsősorban más területeken találkozhatunk vele [13]. GPU-k esetében az alapelv tulajdonképpen azonos, magát az eszközt kell túlterhelni valamilyen formában annyira, hogy az ne tudjon válaszolni az őt érő kérésekre, így ne tudja elvégezni a neki szánt feladatokat. Ez főleg akkor lehet kritikus, ha egy számítógépben csak egy grafikus kártya található, és ezt sikerül annyira leterhelni, hogy az már nem képes ellátni az operációs rendszer által neki szánt megjelenítési feladatokat.

Mindez nem csak elméletben, hanem a gyakorlatban is nagyon könnyen tetten érhető. Különösebb támadási szándék nélkül is, egyszerűen a GPU fejlesztés során is néha előfordul, hogy az elkészített program tesztelése során valamilyen hiba történik, és a program túl sokáig nem válaszol. Ennek eredménye lehet egy egyszerű hibaüzenet, de szerencsétlenebb esetben az egész operációs rendszer összeomlása is.

Mindez támadási céllal is használható, miként azt Patterson [14] is bemutatta. Az általa készített segédprogram folyamatosan hívásokat intézett a grafikus kártya irányába, ami miatt az nem tudott válaszolni az operációs rendszer meghajtónak. Az operációs rendszertől függ az ilyen támadások kimenete. Az általunk is vizsgált Windows7 esetében be van építve egy védelem, aminek köszönhetően abban az esetben, ha a grafikus kártya meghajtó nem válaszol egy megadott időn belül (néhány másodpercre kell gondolni, de ez módosítható), akkor az operációs rendszer egy hibaüzenet megjelenítése mellett automatikusan leállítja és újraindítja a hibásan működő modult.

A gyakorlatban azonban ez számos problémába ütközhet. Egyrészt, maga a védelem nem tudta minden esetben meghiúsítani a támadást, bizonyos esetekben maga az operációs rendszer is összeomlott. Másrészt, bizonyos esetekben éppen az intenzív GPU használat miatt ki kell kapcsolni ezt a védelmet, mivel az leállítana minden, néhány másodpercnél tovább futó GPU kernelt (függetlenül attól, hogy azok éppen értékes munkát végeznek). Harmadrészt, valószínűleg ki lehet játszani ezt a védelmet egy olyan támadással, ami a néhány másodperces terhelések közé néha egy-egy kisebb szünetet iktat, ami miatt az operációs rendszer továbbra is üzemel, azonban a számítógép gyakorlatilag használhatatlanná válik.

A probléma részben orvosolható azzal az egyszerű megoldással, ha nem csak egy, hanem több grafikus kártyát használunk a gépben. Így a kártékony szoftverek csak a számukra kijelölt kártyát tudják túlterhelni, ez azonban nem befolyásolja a második, a képernyő megjelenítésért felelős kártya működését. Ez utóbbi célra gyakran még az alaplapra integrált, meglehetősen kis teljesítményű kártyák is alkalmasak lehetnek.

Malware támadások

GPU alapú kártevőkről napjainkban még nem igazán beszélhetünk, de ez nem zárja ki, hogy ezek nem jelenthetnek egészen komoly veszélyt a közeljövőben [15]. A legnagyobb problémát az jelenti, hogy napjaikban a vírusirtó eszközök nincsenek felkészítve az ezekhez hasonló károkozók keresésére, illetve ez technikailag is meglehetősen nehezen kivitelezhető, hiszen a GPU mind az adatok, mind pedig az aktuálisan futtatott programok szempontjából egy sokkal nehezebb kontrollálható eszköz, mint amit hagyományosan megszoktunk.

Részben persze megnyugtató, hogy maga a GPU egy alapvetően független eszköznek tekinthető, ami meglehetősen korlátos képességekkel bír. Így ugyanis attól nem kell tartanunk, hogy a GPU-n futó kódok közvetlenül kárt okozzanak a merevlemez tartalmában, vagy akár hálózati kommunikációt folytassanak, hiszen napjainkban erre egyszerűen nincs technikai lehetőségük. De érdemes megjegyezni, hogy a fejlesztések egyre inkább afelé haladnak, hogy a GPU-kat minél inkább függetleníteni tudjuk a CPU-któl, ez pedig óhatatlanul is az általuk elérhető egyéb erőforrások irányába vezet.

ÖSSZEGZÉS

Az önálló GPU alkalmazások napjainkban még ritkák, azonban az újszerű eszközök használata egyre gyakrabban jelenik meg különféle kiegészítő modulokban (megjelenítés, grafikus gyorsítás, adatpárhuzamos számítások stb.). Ennek megfelelően a támadásokról is ritkán esik szó, de érdemes felkészülni arra, hogy ezek a közeli jövőben megjelennek és folyamatosan egyre inkább elterjedhetnek.

Mindezt támogatják azok a törekvések is, amelyek afelé irányulnak, hogy a grafikus kártya minél inkább egy általános célú végrehajtó eszköz legyen. Ehhez már elkészült a megfelelő hardver és a szükséges szoftver környezet, és a jövőbeni tervek alapján hamarosan a GPU elszigeteltsége is változni fog. Hozzáférhet a PCI-E buszon keresztül a többi hardver elemhez, a központi memóriához, illetve várhatóan előbb-utóbb integrálva lesz a CPU mellé, ami újabb lehetőséget nyújthat majd a támadók számára.

Szoftver téren is egyre több helyen jelenik meg az új eszköz, a játékok már régóta használják a GPU-t a megjelenítés mellett a fizikai szimulációkra is, de pl. a MATLAB esetében is van már lehetőség GPGPU használatra a számítások végrehajtásakor. Napjainkban kezd terjedni a WebGL (Ábra 3.) komponens, amely lehetővé teszi, hogy weboldalakban letöltött JavaScript is hozzáférjen a grafikus kártyához, ami megint egy újabb potenciális csatornát nyithat a rosszindulatú kódok számára. Mivel ez a lehetőség az összes webportál típusnál adott [16], így ez különösen megkönnyíti majd az ilyen károkozók tevékenységét.

Felhasznált irodalom

[1] T. Hashimoto, T. Suzuki, H. Aoshima, A. Rövid, „Real-Time and High Precision 3D Shape Measurement Method”, Acta Polytechnica Hungarica, vol. 10, no. 8, 2013, pp. 139-152

[2] L. Vokorokos, E. Chovancová, J. Radušovský, M. Chovanec, “A Multicore Architecture Focused on Accelerating Computer Vision Computations”, Acta Polytechnica Hungarica, vol. 10, no. 5, 2013, pp. 29-43

[3] J Tick, C. Imreh, Z. Kovács, “Business Process Modeling and the Robust PNS Problem”, Acta Polytechnica Hungarica, vol. 10, no. 6, 2013, pp. 193-204

[4] V. Póser, T. Schubert, M. Kozlovszky, D. Prém, “Security on-demand megoldások az informatikai infrastruktúrákban”, Hadmérnök, vol. 8, no. 3, pp. 211-222

[5] H. Jeon, M. Wilkening, V. Sridharan, S. Gurumurthi, G. Loh, “Architectural Vulnerability Modeling and Analysis of Integrated Graphics Processors”, The 9th IEEE Workshop on Silicon Errors in Logic - System Effect (SELSE), Stanford, 2013

[6] CUDA Programming Guide, http://developer.download.nvidia.com/compute/cuda /3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf

[7] R. L. Bocchino Jr., V. S. Adve, S. V. Adve, M. Snir, „Parallel Programming Must Be Deterministic by Default”, HotPar ’09, Berkeley, 2009

[8] A. Keszthelyi, „About Passwords”, Acta Polytechnica Hungarica, vol. 10., no. 6, 2013

[9] E. Tóth-Laufer, M. Takács, I. J Rudas, “Interactions Handling Between the Input Factors in Risk Level Calculation”, 11th International Symposium on Applied Machine Intelligence and Informatics (SAMI 2013), Herlany, 2013

[10] J. van Oosten, „CUDA Memory Model”, 3D Game Engine programming, 2011

[11] R. D. Pietro, F. Lombardi, A. Villani, “CUDA Leaks: Information Leakage in GPU Architectures”, arXiv:1305.7383, 2013

[12] T. O’Brien, „WebGL flaw leaves GPU exposed to hackers”, www.engadget.com, 2013.11.

[13] Zs. Haig, „Classification of Information Based Attacks”, Hadtudományi Szemle, vol. 2, no. 3, 2009, pp. 9-14

[14] M. J. Patterson, „Vulnerability analysis of GPU computing”, MSc thesis, 2013

[15] G. Vasiliadis, M. Polychronakis, S. Ioannidis, “GPU-Assisted Malware”, 5th International Conference on Malicious and Unwanted Software (MALWARE), 2010

[16] S. Munk, M. Molnár, “Web portálok típusai, jellemzőik”, Hadmérnök, vol. 4, no. 1, 2009, pp. 235-253