DELTA — Střední škola informatiky a ekonomie, s.r.o.
Ke Kamenci 151, Pardubice

Využití reverzního inženýrství pro tvorbu open-source alternativy k CUDA Driver API

Příjmení, jméno: **Horkel Antonín**
Třída: **4.B**
Studijní obor: **Informační technologie 18-20-M/01**
Školní rok: **2025/2026**

Zadání maturitního projektu z informatických předmětů

Jméno a příjmení: Antonín Horkel
Pro školní rok: 2025/2026
Třída: 4. B
Obor: Informační technologie 18-20-M/01
Téma práce: Open-source alternativa k CUDA Driver API pro Linux a FreeBSD
Vedoucí práce: Ing. Zdeněk Drvota

Způsob zpracování, cíle práce, pokyny k obsahu a rozsahu práce

Cíl práce

Cílem tohoto projektu je vyvinout open-source alternativu k proprietárnímu CUDA Driver API od společnosti NVIDIA. Projekt bude mít podporu pro operační systémy Linux a FreeBSD. Mezi jeho základní funkce bude patřit čtení zkompilovaných CUDA binárních souborů (cubin) pomocí vlastního parseru ELF souborů, jejich nahrávání na GPU a spouštění na command queue GPU. Dále bude umět alokovat a dealokovat paměť GPU a mapovat paměť GPU tak, aby byla přístupná pro CPU.

Očekávané výstupy

1. Hlavní část

- Rust knihovna (crate) pro komunikaci s GPU s funkcemi uvedenými níže.
- Ukázkové programy demonstrující tyto funkce.
- Uživatelská dokumentace knihovny.
- Jednoduchý program s CLI rozhraním umožňující vypsání všech NVIDIA GPU na zařízení a CUDA binárních souborů (cubin) na určitém GPU.

2. Sestavovací systém a reprodukovatelné vývojové prostředí (development shell)

- Implementováno pomocí Nix flake s využitím:
  - rust-overlay (https://github.com/oxalica/rust-overlay).
  - Crane (https://github.com/ipetkov/crane).

3. Testy a benchmarky

- Srovnání výstupů a rychlosti s CUDA Driver API na:
  - Zkompilovaných CUDA kernelech.
  - Triton (https://github.com/triton-lang/triton) kernelech.

Funkční požadavky

- Načítání CUDA binárních souborů (cubin) pomocí vlastního parseru ELF souborů.
- Spouštění výpočetních kernelů na command queue GPU.
- Alokace, dealokace paměti GPU a mapování paměti z GPU na CPU.
- Kopírování paměti (memcpy):
  - Na GPU.
  - Z GPU na CPU.
  - Z CPU na GPU.

Stručný časový harmonogram

- **Září:**
  - Tvorba Nix flake infrastruktury.
  - Analýza zdrojových kódů:
    - LibreCUDA (https://github.com/mikex86/LibreCuda).
    - Tinygrad NVIDIA runtime (https://github.com/tinygrad/tinygrad/blob/master/tinygrad/runtime/ops_nv.py).
  - Zahájení vývoje knihovny.
- **Říjen:** Vývoj verze 1.0 knihovny.
- **Listopad:**
  - Implementace ELF parseru pro cubin soubory.
  - Dokončení verze 1.0 knihovny.
- **Prosinec:**
  - Tvorba uživatelské dokumentace.
  - Vývoj ukázkových programů.
  - Implementace testů a benchmarků.
- **Leden:** Použití cuda_ioctl_sniffer (https://github.com/geohot/cuda_ioctl_sniffer) pro zjištění více informací o interním fungování CUDA Driver API.
- **Únor - Březen:** Práce na dokumentaci.

Prohlášení

Prohlašuji, že jsem maturitní projekt vypracoval samostatně, výhradně s použitím uvedené literatury.

V Pardubicích dne ....

Antonín Horkel

Poděkování

Poděkování patří především vedoucímu práce Ing. Zdeňku Drvotovi za odborné vedení, cenné rady a čas, který mi v průběhu zpracování práce věnoval.

Děkuji také své rodině za podporu během celého studia.

Anotace

Tato práce se zabývá vývojem open-source alternativy k proprietárnímu CUDA Driver API s využitím technik reverzního inženýrství. Výsledkem je knihovna v jazyce Zig poskytující parsery binárních formátů cubin a fatbin a rozhraní pro komunikaci s NVIDIA ovladači zařízení. V rámci práce byla analyzována šifrovaná data v binárních souborech nástrojů platformy CUDA, což umožnilo získat kompletní definice CUDA-specifických typů relokací.

Klíčová slova: CUDA, CUDA Driver API, cubin, fatbin, reverzní inženýrství, GPGPU, NVIDIA Linux open GPU kernel modules

Abstract

This work focuses on the development of an open-source alternative to the proprietary CUDA Driver API using reverse engineering techniques. The result is a library written in the Zig language that provides parsers for the cubin and fatbin binary formats and an interface for communicating with NVIDIA device drivers. As part of this work, encrypted data in the binary files of tools from the CUDA platform was analyzed, which made it possible to obtain complete definitions of CUDA-specific relocation types.

Keywords: CUDA, CUDA Driver API, cubin, fatbin, reverse engineering, GPGPU, NVIDIA Linux open GPU kernel modules

Obsah

1. Úvod 8
2. Reverzní inženýrství 8
3. Využité technologie 9
   - 3.1. Zig 10
   - 3.2. Nix 10
   - 3.3. Binary Ninja Free 10
   - 3.4. GDB 11
   - 3.5. Python 11
   - 3.6. Angr 11
   - 3.7. QEMU 11
4. Ovladače zařízení 11
   - 4.1. Odposlech komunikace 13
     - 4.1.1. Strace a ltrace 13
     - 4.1.2. LD_PRELOAD 13
     - 4.1.3. Odposlech pomocí debuggeru 14
     - 4.1.4. Ladící výpisy v ovladačích zařízení 14
5. Souborový formát cubin 15
   - 5.1. Sekce .nv.info 17
   - 5.2. Relokace 17
     - 5.2.1. Interpretace získaných dat 18
       - 5.2.1.1. Sekce .nv.rel.action 20
   - 5.3. Modifikace šifrovaných dat 20
     - 5.3.1. Lineární kongruentní generátor 21
     - 5.3.2. S-box 23
     - 5.3.3. Bitová nonekvivalence 24
     - 5.3.4. Inverzní zobrazení dešifrovací funkce 24
6. Souborový formát fatbin 25
7. Architektura 26
8. Metody testování 27
9. Možná rozšíření projektu 28
10. Závěr 29

1. Úvod

CUDA (zkratka pro *Compute Unified Device Architecture*) je platforma pro obecné výpočetní úlohy přesahující tradiční grafické renderování na NVIDIA grafických procesorech (anglicky *General-purpose computing on graphics processing units*, zkratka *GPGPU*). Platforma byla poprvé představena společností NVIDIA v roce 2006 a vize pro ni byla vytyčena v Lindholm et al. (2008) [1]. Poskytuje programovací model a sadu knihoven a nástrojů, které umožňují vývojářům psát paralelní kód v jazycích C a C++ a spouštět jej na grafickém procesoru. CUDA se postupně stala de facto standardem pro GPU akcelerované úlohy vyžadující vysoký výpočetní výkon, jako je strojové učení, vědecké simulace, zpracování obrazu či kryptografie.

CUDA Driver API [2] je user space komponenta CUDA ovladačů sloužící ke spouštění výpočetních kernelů na GPU, alokaci paměti na GPU a dalším operacím potřebným pro běh CUDA aplikací. Ve srovnání s CUDA Runtime API, které slouží ke stejným účelům, představuje nižší úroveň abstrakce a umožňuje větší kontrolu [3, 4]. Stejně jako většina částí platformy CUDA je CUDA Driver API proprietární s uzavřeným zdrojovým kódem, avšak existují projekty zaměřené na reimplementaci některých jejích částí pomocí reverzního inženýrství — mezi ně patří LibreCUDA [5], tinygrad [6] a Gdev [7, 8]. Tato práce navazuje na zmíněné projekty s cílem dosáhnout vyšší míry kompatibility s CUDA Driver API a prohloubit porozumění jeho vnitřní architektuře a implementačním detailům. Práce si neklade za cíl být binárně kompatibilní s CUDA Driver API ani reimplementovat celou jeho funkcionalitu.

Otevřená reimplementace CUDA Driver API přináší několik výhod. Umožňuje detailně porozumět mechanismům komunikace mezi CUDA aplikacemi a GPU hardwarem a získat explicitnější kontrolu nad nimi. Tyto poznatky mohou napomoct při optimalizaci výkonu CUDA aplikací. Zároveň otevřený zdrojový kód umožňuje jiným open-source projektům snížit závislost na proprietárním softwaru a zjednodušit distribuci. V neposlední řadě představuje krok k možné budoucí podpoře CUDA alternativy na operačních systémech, které CUDA oficiálně nepodporuje, ale je na ně možné portovat open-source NVIDIA ovladače zařízení, jako například Haiku [9].

2. Reverzní inženýrství

Reverzní inženýrství (anglicky *reverse engineering*) je v kontextu softwarového inženýrství proces systematické analýzy a dekompozice softwaru za účelem pochopení jeho vnitřní struktury a funkcionality v situacích, kdy nejsou k dispozici zdrojové kódy, dokumentace nebo jiné materiály od původních vývojářů. Tento proces může zahrnovat statickou analýzu binárního kódu, dynamickou analýzu chování programu za běhu, analýzu síťové komunikace nebo interakcí s operačním systémem a studium proprietárních datových formátů. Získané poznatky je možno využít například při reimplementaci proprietárního softwaru, implementaci ovladačů zařízení, které nejsou pro určité prostředí dostupné, bezpečnostní analýze a hledání zranitelností, forenzní analýze malwaru a obnově ztracených zdrojových kódů.

Mezi základní nástroje používané při reverzním inženýrství patří:

- Disassemblery — nástroje převádějící strojový kód do assembleru. Umožňují analyzovat tok programu, identifikovat funkce a datové struktury.
- Dekompilátory — nástroje pokoušející se rekonstruovat zdrojový kód ve vyšším programovacím jazyce (obvykle v C) nebo mezikódu (anglicky *intermediate representation*, zkratka *IR*) ze strojového kódu. Výstup dekompilátoru obvykle není identický s původním zdrojovým kódem, protože kompilace je ztrátový proces (z hlediska informací o zdrojovém kódu, ne sémantiky), ale může výrazně usnadnit pochopení logiky programu.
- Debuggery — nástroje umožňující řízeně spouštět program, pozastavovat jeho běh na definovaných místech (breakpointech), krokovat jednotlivé instrukce a zkoumat stav operační paměti a registrů. Moderní debuggery také často podporují skriptování.

Proti reverznímu inženýrství může být software typicky chráněn pomocí:

- Obfuskace kódu — transformace kódu do podoby, která zachovává jeho funkcionalitu, ale výrazně ztěžuje jeho pochopení.
- Šifrování a komprese
- Kontroly integrity — program může za běhu ověřovat, zda nebyl jeho kód nebo stav modifikován.
- Technik proti debugování — program dokáže detekovat, zda běží pod debuggerem a případně změnit své chování nebo ukončit běh.

3. Využité technologie

3.1. Zig

Zig je systémový programovací jazyk navržený pro vývoj nízkoúrovňového softwaru s důrazem na explicitní kontrolu nad pamětí a hardwarovými zdroji. Jazyk nabízí manuální správu paměti bez garbage collectoru a garantuje, že nedochází k žádným skrytým alokacím. Významnou výhodou je nativní interoperabilita s jazykem C — Zig umožňuje přímo importovat hlavičkové soubory jazyka C pomocí nástroje translate-c. Součástí jazyka je rovněž integrovaný sestavovací systém, který umožňuje definovat proces sestavení projektu přímo v jazyce Zig bez závislosti na externích nástrojích jako GNU Make nebo CMake [10].

Původní zadání projektu předpokládalo implementaci v jazyce Rust [11]. Ten se však ukázal jako méně vhodný, protože komunikace s ovladači zařízení pomocí systémových volání ioctl vyžaduje použití unsafe bloků a práci s raw pointery, čímž se záruky paměťové bezpečnosti kompilátoru obcházejí. Rust navíc na rozdíl od Zigu neposkytuje nativní interoperabilitu s jazykem C. Oficiální nástroj bindgen určený pro tento účel závisí na knihovně libclang a má oproti translate-c omezenější kompatibilitu se standardem C99, zejména při zpracování maker preprocesoru.

Nevýhodou jazyka Zig je jeho relativní nezralost. Jazyk dosud nedosáhl stabilní verze 1.0, což znamená možné změny narušující zpětnou kompatibilitu mezi verzemi.

3.2. Nix

Nix je čistě funkcionální správce balíčků a sestavovací systém, který zajišťuje reprodukovatelnou a deklarativní správu softwarových závislostí [12]. V rámci projektu je Nix využíván pro reprodukovatelné sestavení knihovny a definici vývojového prostředí (anglicky *development shell*), které obsahuje všechny závislosti potřebné pro vývoj projektu. Jelikož byl místo jazyka Rust zvolen jazyk Zig, nebyly použity původně zamýšlené nástroje rust-overlay [13] a Crane [14], ale jejich alternativy pro jazyk Zig — zig-overlay [15] pro správu verzí kompilátoru Zig a zon2nix [16] pro integraci závislostí ze souboru build.zig.zon.

3.3. Binary Ninja Free

Binary Ninja je platforma pro reverzní inženýrství s integrovaným disassemblerem a dekompilátorem [17]. Verze Free je bezplatná varianta s omezenou funkcionalitou, která však pro účely tohoto projektu postačuje. V rámci projektu byla použita pro analýzu binárních souborů nástrojů platformy CUDA.

3.4. GDB

GDB (GNU Debugger) je debugger pro programy napsané v jazycích C, C++ a dalších [18]. Umožňuje řízeně spouštět programy, nastavovat breakpointy, krokovat instrukce a zkoumat stav paměti a registrů. GDB podporuje skriptování v jazyce Python, což umožňuje automatizaci analýzy a interpretace zachycených dat. V rámci projektu byl GDB použit pro odposlech systémových volání ioctl prováděných CUDA aplikacemi.

3.5. Python

Python je vysokoúrovňový interpretovaný programovací jazyk s dynamickým typováním [19]. V rámci projektu byl použit pro implementaci pomocných skriptů, včetně skriptů pro GDB a nástroje pro dešifrování dat z binárního souboru nástroje nvdisasm.

3.6. Angr

Angr je open-source framework v jazyce Python pro binární analýzu [20, 21]. Umožňuje načítat spustitelné soubory, analyzovat jejich strukturu a simulovat jejich běh pomocí symbolického vykonávání. V rámci projektu byl použit pro extrakci a dešifrování dat z binárního souboru nástroje nvdisasm.

3.7. QEMU

QEMU (zkratka pro *Quick Emulator*) je open-source emulátor a virtualizační nástroj umožňující emulaci různých CPU architektur [22]. V rámci projektu byl použit pro automatické testování na odlišných CPU architekturách, viz kapitola 8.

4. Ovladače zařízení

Pro NVIDIA grafické karty existuje několik možností ovladačů zařízení. Na operačních systémech Linux a FreeBSD jsou dostupné oficiální proprietární ovladače, které poskytují plnou funkcionalitu a optimální výkon, avšak znemožňují analýzu vnitřní implementace. Stejně tak jsou pro tyto systémy dostupné open-source moduly jádra od NVIDIA (anglicky *NVIDIA open GPU kernel modules*) pro GPU s architekturou Turing a novější, licencované pod duální licencí GPL a MIT [23]. Tyto moduly podporují pouze GPU vybavené GSP (GPU System Processor) [24], což je integrovaný mikroprocesor s proprietárním firmwarem, který zajišťuje inicializaci a správu GPU. Pouze na Linuxu je dále dostupný komunitní projekt Nouveau [25], který je plně open-source, ale nepodporuje novější GPU architektury a postrádá podporu pro CUDA.

Společnost NVIDIA rovněž poskytuje repozitář open-gpu-doc obsahující oficiální dokumentaci ve formě hlavičkových souborů jazyka C [26]. Tato dokumentace je však velmi strohá — většina obsahu tvoří pouze definice maker bez jakéhokoliv popisu jejich účelu, což značně ztěžuje její pochopení. Zdrojový kód open-source modulů jádra může posloužit jako doplňující zdroj informací a příklad použití některých hardwarových funkcí.

Komunikace s NVIDIA ovladači zařízení probíhá prostřednictvím systémového volání ioctl a dalších systémových volání jako mmap.

Moduly open-source ovladačů:
- nvidia.ko — hlavní modul zajišťující komunikaci s GPU prostřednictvím RM API (Resource Manager API).
- nvidia-modeset.ko — modul pro správu zobrazovacích režimů.
- nvidia-drm.ko — modul poskytující rozhraní DRM (Direct Rendering Manager) pro integraci s linuxovým grafickým stackem.
- nvidia-uvm.ko — modul implementující UVM (Unified Virtual Memory) pro správu virtuální paměti sdílené mezi CPU a GPU [27, 28].

CUDA Driver API komunikuje primárně s moduly nvidia.ko a nvidia-uvm.ko prostřednictvím speciálních souborů zařízení:
- /dev/nvidiactl — řídící soubor pro RM API operace jako alokace zdrojů a konfigurace GPU.
- /dev/nvidia-uvm — soubor pro UVM operace včetně mapování paměti mezi CPU a GPU.
- /dev/nvidia*N* — soubor pro konkrétní GPU s indexem *N*, používaný pro operace specifické pro dané zařízení.

RM API je interní rozhraní NVIDIA ovladačů pro správu hardwarových zdrojů. Operace RM API zahrnují alokaci objektů (kanály, paměťové buffery, kontexty), konfiguraci hardwaru a odesílání příkazů na GPU. Příkazy pro GPU jsou organizovány do příkazových front (anglicky *command queues*), přičemž příkazy pro spuštění kernelu jsou popsány metadaty QMD (Queue Metadata).

4.1. Odposlech komunikace

Pro analýzu komunikace mezi CUDA aplikacemi a ovladači zařízení existuje několik přístupů, z nichž každý nabízí odlišnou úroveň detailu a vyžaduje různou míru úsilí při implementaci.

4.1.1. Strace a ltrace

Nejjednodušším přístupem je použití nástrojů strace [29] a ltrace [30]. Strace slouží k zachytávání systémových volání, ltrace naproti tomu zachytává volání funkcí dynamických knihoven. Oba nástroje interně využívají systémové volání ptrace, které umožňuje jednomu procesu sledovat a řídit běh jiného procesu.

Hlavní výhodou těchto nástrojů je, že na rozdíl od ostatních metod nevyžadují žádnou konfiguraci ani programování a poskytují okamžitý přehled o systémových voláních prováděných sledovanou aplikací.

Zásadním omezením je však způsob zobrazení výstupu. U složitějších datových struktur předávaných v systémových voláních, například ukazatelů na struktury, zobrazují tyto nástroje pouze adresy v paměti bez možnosti náhledu do jejich obsahu. Z tohoto důvodu je výstup prakticky nepoužitelný pro účely reverzního inženýrství komunikace s NVIDIA ovladači.

4.1.2. LD_PRELOAD

Sofistikovanějším přístupem je využití proměnné prostředí (anglicky *environment variable*) LD_PRELOAD. Tato proměnná umožňuje specifikovat sdílenou knihovnu, která je dynamickým linkerem načtena před všemi ostatními knihovnami. Díky tomu může tato knihovna nahradit nebo obalit funkce z jiných knihoven, včetně funkcí standardní knihovny libc provádějících systémová volání jako ioctl. Nástroje envyhooks [31] a cuda_ioctl_sniffer [32] využívají tento přístup k zachytávání a analýze komunikace CUDA aplikací s ovladači zařízení.

Výhodou těchto nástrojů je, že na rozdíl od strace a ltrace poskytují strukturovaný a čitelný výstup, protože knihovna pro zachytávání má přístup k úplnému kontextu volání včetně obsahu datových struktur.

Nevýhodou je, že LD_PRELOAD funguje pouze pro programy dynamicky linkující standardní knihovnu libc. V případě statického linkování nebo přímých systémových volání bez použití libc tato metoda selhává.

4.1.3. Odposlech pomocí debuggeru

Větší flexibilitu poskytuje odposlech systémových volání pomocí debuggeru. Stejně jako strace a ltrace využívá debugger systémové volání ptrace pro sledování jiného procesu. Debugger však umožňuje nejen zachytávat systémová volání prováděná sledovaným procesem, ale i automatizovat jejich analýzu a interpretaci prostřednictvím skriptování. Na rozdíl od předchozích metod lze pomocí debuggeru nejen pasivně sledovat komunikaci, ale také pozastavit běh sledované CUDA aplikace na definovaných místech, například při volání konkrétních funkcí CUDA Driver API, a prozkoumat aktuální stav procesu.

Výhody tohoto přístupu jsou následující:
- Možnost skriptování pro automatizovanou analýzu a dekódování zachycených dat přímo během běhu programu.
- Funguje nezávisle na tom, zda sledovaný program využívá standardní knihovnu libc nebo ji linkuje staticky.
- Možnost využít další funkce debuggeru, jako je nastavení breakpointů, inspekce paměti či modifikace stavu procesu.

Nevýhodou naopak je, že pro smysluplnou interpretaci zachycené komunikace je nutné vytvořit skripty, které dekódují datové struktury použité v systémových voláních.

4.1.4. Ladící výpisy v ovladačích zařízení

Poslední možností je využití ladících výpisů v open-source NVIDIA ovladačích zařízení. Nastavením parametru `DEBUG=1` při sestavování těchto ovladačů pomocí nástroje GNU Make se povolí vypisování ladících zpráv do vyrovnávací paměti systémových zpráv (anglicky *system message buffer*, často označováno také jako *kernel ring buffer*) [23], jejíž obsah lze zobrazit příkazem dmesg. Ladící výpisy jsou v kódu ovladačů realizovány pomocí maker jako `NV_PRINTF` a funkce `nv_printf`. Tento přístup rovněž umožňuje vkládat vlastní volání těchto maker na libovolná místa ve zdrojovém kódu ovladačů, čímž lze získat podrobnější přehled o průběhu komunikace.

Hlavní výhodou této metody je možnost hlubokého náhledu do vnitřního stavu ovladačů a způsobu zpracování komunikace.

Nevýhody naopak jsou:

- Vyžaduje znalost konkrétního místa ve zdrojovém kódu ovladačů, kde je daná část komunikace zpracovávána.
- Jakákoliv modifikace zdrojového kódu vyžaduje opětovné sestavení ovladačů (což s parametrem `DEBUG=1` trvá řádově jednotky minut) a následně jejich načtení namísto aktuálně používaných. Používané moduly jádra nelze uvolnit, dokud je využívá některý z běžících procesů (například X11 server, Wayland kompozitor či CUDA aplikace).

Pro potřeby této práce byl primárně využíván vlastní GDB skript a nástroj envyhooks [31].

5. Souborový formát cubin

Cubin (CUDA binary) soubory jsou binární soubory ve formátu ELF [33] obsahující kompilovaný GPU strojový kód. Formát cubin není oficiálně dokumentovaný, avšak platforma CUDA poskytuje nástroje cuobjdump a nvdisasm [33], které slouží ke stejným účelům jako readelf a objdump z balíčku GNU binutils [34–36], pouze specificky pro cubin soubory. Přestože výstupní formát těchto nástrojů rovněž není zdokumentovaný ani zcela intuitivní, mohou výrazně napomoci při analýze struktury cubin souborů.

Cubin soubory obvykle obsahují:

- Kompilovaný GPU strojový kód pro konkrétní výpočetní možnost (anglicky *compute capability*, často psáno jako *SMXY* nebo *sm_XY* kde *X* označuje číslo generace a *Y* verzi v rámci dané architektury [37]) v sekci .text.*func*, kde *func* je dekorovaný název (anglicky *name mangling*) konkrétního kernelu.
- Konstantní data v sekci .nv.constant*N* a .nv.constant*N*.*func*, kde *N* je číslo konstanty a *func* dekorovaný název kernelu. Pokud v názvu sekce není *func*, konstantní data jsou dostupná všem kernelům.
- Metadata o registrech, sdílené paměti, velikosti bloků vláken a dalších parametrech, viz kapitola 5.1.
- Relokace, viz kapitola 5.2.

Formát ELF (zkratka pro *Executable and Linkable Format*) [38] je velmi rozšířený souborový formát používaný nejen pro spustitelné soubory, ale i dynamické knihovny (anglicky *shared libraries*), výpisy paměti při pádu programu (anglicky *core dumps*) a objektový kód (anglicky *object code*) mimo jiné. Původně byl vyvinut v Unix System Laboratories pro operační systém UNIX System V Release 4.0 (zkratka *SVR4*) [39, 40]. Formát je velmi flexibilní a podporuje například různou endianitu. Používá se v operačních systémech Linux, FreeBSD, NetBSD, OpenBSD, Solaris a dalších.

Struktura ELF souboru se skládá z hlavičky ELF, tabulky hlaviček segmentů (anglicky *program header table*) obsahující popis dostupných segmentů (například typ) a odkazy na ně, tabulky hlaviček sekcí (anglicky *section header table*) obsahující popis dostupných sekcí (například jméno a typ) a odkazy na ně, a samotných dat odkazovaných z těchto tabulek. Každý segment přitom obsahuje jednu nebo více částí.

5.1. Sekce .nv.info

Cubin soubory obsahují sekci .nv.info a dále sekci .nv.info.*func* pro každý kernel, kde *func* je jeho dekorovaný název [41–43]. Tyto sekce obsahují metadata potřebná pro správné spuštění kernelu, například velikost lokální paměti přidělené na jedno vlákno, která je určena atributy MIN_STACK_SIZE, FRAME_SIZE a MAX_STACK_SIZE.

Sekce .nv.info se skládají z jednoho nebo více atributů. Každý atribut začíná bajtem označujícím formát atributu, po kterém následuje bajt s identifikátorem atributu. Formát atributu určuje strukturu jeho hodnoty:

- NVAL (1) — atribut nemá přiřazenou žádnou hodnotu.
- BVAL (2) — hodnota atributu je uložena v následujícím jednom bajtu.
- HVAL (3) — hodnota atributu je uložena v následujících dvou bajtech.
- SVAL (4) — následující dva bajty obsahují délku n a následujících n bajtů obsahuje hodnotu atributu.

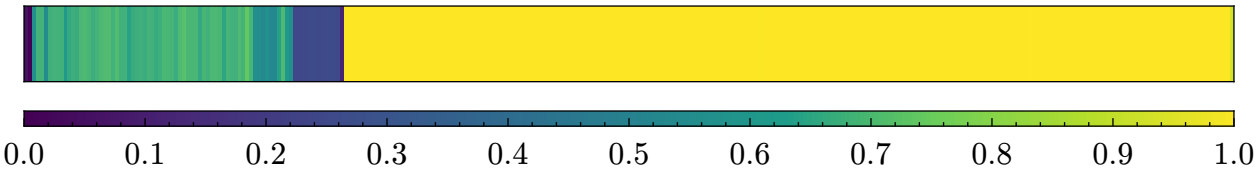
Metadata v sekci .nv.info.*func* platí pouze pro příslušný kernel v sekci .text.*func*, zatímco metadata v sekci .nv.info se vztahují ke všem kernelům v cubin souboru. Atributy typu SVAL v sekci .nv.info však často obsahují index příslušného kernelu v tabulce symbolů, čímž je určeno, ke kterému kernelu se daný atribut vztahuje.

Například atribut REGCOUNT v sekci .nv.info má formát SVAL. Po identifikátoru atributu následují dva bajty určující délku (hodnota 8), poté čtyři bajty obsahující index příslušného kernelu v tabulce symbolů a zbývající čtyři bajty obsahují skutečný vyžadovaný počet registrů.

5.2. Relokace

U ELF spustitelných souborů je relokace proces zajišťující propojení symbolických odkazů nebo link-time adres v kompilovaném programu se skutečnými adresami ve virtuální paměti [44]. To je nezbytné, když skutečné adresy ve virtuální paměti, kde mají být funkce a data umístěny za běhu programu, nejsou známy v době linkování. Existuje několik typů relokací pro různé účely, například pro umístění absolutní adresy nebo umístění adresy relativní k čítači instrukcí (anglicky *program counter*, zkratka *PC*, na některých CPU architekturách zvaný také instrukční ukazatel, anglicky *instruction pointer*, zkratka *IP*). Jednotlivé typy relokací používají různé výpočty výsledné hodnoty. Typy relokací a jejich výpočty jsou specifické pro každou architekturu procesoru.

Pro správné načtení cubin souborů je nezbytné znát tyto detaily pro CUDA-specifické typy relokací. Současné implementace načítání cubin souborů v projektech LibreCUDA [5] a tinygrad [6] obsahují seznam výpočtů relokací, který je značně nekompletní. Oficiální CUDA nástroje cuobjdump a nvdisasm sice umožňují vypsat relokace použité v cubin souboru a zobrazit o nich některé doplňující informace, avšak nezobrazují výpočet relokací [33]. Při pokusu vyhledat názvy relokací v binárních souborech těchto nástrojů pomocí nástroje strings z balíčku GNU binutils [34, 45] se ukázalo, že tyto názvy se v nich vůbec nevyskytují jako běžné textové řetězce. Analýza entropie binárních souborů nepřinesla u nástroje cuobjdump žádné zajímavé výsledky, avšak nvdisasm obsahuje rozsáhlý úsek dat s téměř maximální entropií, viz graf 1. To není typické pro běžné spustitelné soubory ve formátu ELF a naznačuje přítomnost komprimovaných nebo šifrovaných dat (případně obojího). Tuto skutečnost již dříve odhalil projekt denvdis, který zahrnuje dešifrovací nástroj pro tato data [46–48]. Tento nástroj však funguje pouze pro jednu konkrétní verzi nvdisasm, protože pracuje s pevně zakódovanými offsety. Navíc použitá dešifrovací strategie není nijak zdokumentována ani vysvětlena.



Graf 1: Entropie 16 KiB kusů binárního souboru programu nvdisasm verze 13.1.80 pro Linux x86-64 normalizovaná na $\langle 0; 1 \rangle$.

Analýzou binárního souboru nvdisasm pomocí dekompilátoru Binary Ninja Free [17] se podařilo identifikovat funkci zodpovědnou za dešifrování těchto dat. Analýza dále odhalila, že některá data jsou před zašifrováním komprimována algoritmem LZ4 (Lempel-Ziv 4) [49]. Na základě těchto zjištění byl vytvořen skript v jazyce Python [19] využívající framework pro binární analýzu angr [20, 21], který dokáže extrahovaná data dešifrovat a v případě potřeby dekomprimovat nezávisle na verzi nvdisasm. Pomocí dekompilátoru bylo zjištěno, že další nástroje platformy CUDA, konkrétně ptxas a nvlink, také obsahují šifrovaná data a využívají stejnou dešifrovací funkci.

5.2.1. Interpretace získaných dat

Získaná data jsou rozdělena na části odpovídající jednotlivým výpočetním možnostem (anglicky *compute capability*). Každá část vždy obsahuje tabulku s definicemi relokací a specifikaci instrukční sady SASS (zkratka pro *streaming assembler*) [33] pro danou architekturu. Analýza instrukční sady SASS přesahuje zaměření této práce, avšak projekty DocumentSASS [50]

(který získal stejná data prostřednictvím odposlechu volání funkce memcpy prováděných kompilátorem nvcc) a denvdis [46] se této problematice věnují podrobně.

Většina relokací je v dešifrovaných datech reprezentována v následujícím formátu:

```
{ "R_CUDA_64", 0xffffffffffffffff, False, False, 0,0, { { 0, 64} } }
{ "R_CUDA_ABS32_HI_32", 0xffffffff00000000, False, False, 0,0, { { 32, 32} } }
{ "R_CUDA_PCREL_IMM24_26", 0xffffffffffffffff, True, False, 0,0, { { 26, 24} } }
{ "R_CUDA_FUNC_DESC_8_32", "fdesc", 0x000000ff00000000, False, False, 0,0, { { 0,
8} } }
{ "R_CUDA_ABS56_16_34", 0xffffffffffffffff, False, False, 0,2, { { 16, 8}, { 34,
48} } }
```

Interpretace formátu:

1. Typ relokace.
2. Volitelný řetězec označující speciální typ symbolu — *fdesc* pro deskriptor funkce (anglicky *function descriptor*) nebo *unified* pro symbol v *unified memory*.
3. Bitová maska určující, které bity v cílové paměti relokace modifikuje.
4. Volitelný řetězec označující adresu v konstantní paměti — *ConstBankAddress0* pro konstantní paměť 0 nebo *ConstBankAddress2* pro konstantní paměť 2.
5. Boolean indikující relativní adresování vůči čítači instrukcí (PC-relative).
6. Neidentifikované pole, vždy *False*.
7. Neidentifikované pole, vždy 0.
8. Hodnota, o kterou je addend relokace logicky bitově posunut doprava před aplikací.
9. Pole rozsahů bitů, které relokace modifikuje, kde každý rozsah je definován dvojicí určující pozici (první číslo) a počet (druhé číslo) bitů v cílové paměti.

Díky těmto informacím je již možné odvodit výpočetní metodu a relokace aplikovat.

Některé relokace jsou v dešifrovaných datech reprezentovány v odlišném formátu od ostatních:

```
{ "R_CUDA_INSTRUCTION128", 17, 0, 128 }
{ "R_CUDA_YIELD_OPCODE9_0", 18, 0, 9 }
{ "R_CUDA_YIELD_CLEAR_PRED4_87", 19, 87, 4 }
```

Interpretace formátu:

1. Typ relokace.
2. Pravděpodobně interní identifikátor výpočetní metody.
3. Pozice bitů v cílové paměti, které relokace modifikuje.

4. Počet bitů v cílové paměti, které relokace modifikuje.

Výpočetní metodu se podařilo nalézt pouze pro relokace `R_CUDA_YIELD_OPCODE9_0` a `R_CUDA_YIELD_CLEAR_PRED4_87`. Tyto relokace jsou aplikovány, pokud CUDA kernel není spuštěn v kooperativním módu [51, 52]. První z nich modifikuje opkód (anglicky *opcode*, zkratka pro *operation code*) SASS instrukce YIELD na NOP, druhá vynuluje bity v instrukci specifikující yield predicate register (instrukce NOP tento registr nepodporuje). Informace o binárním formátu těchto instrukcí byly rovněž získány z dešifrovaných dat.

Pro zbývající relokace s odlišným formátem se nepodařilo určit výpočetní metody, jelikož se nepodařilo zkonstruovat CUDA kernel, jehož kompilace by tyto relokace generovala. Tato skutečnost však současně naznačuje, že tyto relokace jsou využívány pouze ve specifických okrajových případech, případně nejsou v současných verzích platformy CUDA používány vůbec. To, že je implementace v projektu nepodporuje, nepředstavuje z praktického hlediska významné omezení.

5.2.1.1. Sekce .nv.rel.action

Cubin soubory často obsahují sekci s názvem .nv.rel.action. Výstup nástroje cuobjdump pro soubory obsahující tuto sekci odhaluje, že sekce uchovává strukturovaná metadata o relokacích. Konkrétně typ relokace, typ symbolu, logický bitový posun doprava addendu a rozsahy bitů, které relokace modifikuje (pozici a počet bitů).

Není jasné, k čemu sekce slouží a vyhledávání názvu sekce prostřednictvím internetových vyhledávačů Google [53], DuckDuckGo [54], Baidu [55] a Yandex [56] nepřineslo žádné relevantní výsledky. Domnívám se, že sekce slouží k zajištění zpětné kompatibility s aplikacemi využívajícími starší CUDA verze. Ty nemusí znát definice relokací zavedených v novějších verzích platformy CUDA a sekce .nv.rel.action jim poskytuje potřebné informace. Tuto hypotézu podporuje skutečnost, že typy relokací obsažené v této sekci odpovídají relokacím zavedeným v novějších verzích platformy.

V rámci implementace parseru cubin souborů jsou data z této sekce využívána pro verifikaci správnosti definic relokací extrahovaných z dešifrovaných dat.

5.3. Modifikace šifrovaných dat

Možnost modifikovat šifrovaná data v binárních souborech nástrojů platformy CUDA má praktické využití pro vytváření modifikovaných verzí nástrojů s upraveným chováním.

Předpokladem pro modifikaci je odvození šifrovací funkce jako inverzního zobrazení k dešifrovací funkci. Dešifrovací funkce je definována následovně. Vstupem je inicializační hodnota x, která závisí na šifrovaných datech a uspořádaná n-tice šifrovaných bajtů $E = (e_1, e_2, ..., e_n)$. Výstupem je uspořádaná n-tice dešifrovaných bajtů $D = (d_1, d_2, ..., d_n)$.

Algoritmus 1: Dešifrovací algoritmus

```
1  S[256] = {0x64, 0x80, ..., 0xCA}        ▷ S-box jako LUT, viz kapitola 5.3.2.
2  a = 0x41C64E6D                          ▷ LCG násobitel, viz kapitola 5.3.1.
3  c = 0x3039                              ▷ LCG inkrement, viz kapitola 5.3.1.
4  E = (e_1, e_2, ..., e_n)               ▷ Vstupní šifrované bajty jako uspořádaná n-tice.
5  D = (d_1, d_2, ..., d_n)               ▷ Výstupní dešifrované bajty jako uspořádaná n-tice.
6  function DEŠIFROVAT(x, E)
7      y ← 0
8      l ← ¬x mod 2^8
9      for i ∈ ⟨1; n⟩ ∩ ℕ do
10         if (i − 1) mod 4 = 0 then
11             y ← (ax + c) mod 2^32
12             x ← y
13         else
14             y ← y ≫ 8
15         end
16         d_i ← (S[(e_i ⊕ l) mod 2^8] ⊕ x) mod 2^8
17         l ← e_i
18     end
19     return (x, y, l, D)
20 end
```

$\neg$	Bitová negace.
$\gg$	Logický bitový posuv doprava, $x \gg n = x/2^n$.
$\oplus$	Bitová nonekvivalence, viz kapitola 5.3.3.

Algoritmus využívá dvě kryptografické primitiva: výpočet $(ax + c) \bmod 2^{32}$ na řádku 11 odpovídá lineárnímu kongruentnímu generátoru, viz kapitola 5.3.1 a vyhledání v tabulce S na řádku 16 odpovídá aplikaci S-boxu, viz kapitola 5.3.2.

5.3.1. Lineární kongruentní generátor

Lineární kongruentní generátor (anglicky *Linear Congruential Generator*, zkratka *LCG*) je algoritmus, který generuje posloupnost pseudonáhodných čísel. Je definován rekurentním vztahem [57, 58]:

$$X_{i+1} = (aX_i + c) \bmod m$$

Kde:

X_0	semínko	$X_0 \in \mathbb{N}^0, 0 \leq X_0 < m$
a	násobitel	$a \in \mathbb{N}^0, 0 \leq a < m$
c	inkrement	$c \in \mathbb{N}^0, 0 \leq c < m$
m	modulus	$m \in \mathbb{N}^+$

Nvdisasm používá stejné konstanty pro násobitele a inkrement jako funkce rand v GNU C Library (glibc) verze 2.42 [59, 60]. Glibc používá 2^{31} jako modulus, ale nvdisasm nepoužívá žádný explicitně, místo toho používá 32-bitový registr pro ukládání výsledku $aX_i + c$, čímž implicitně používá 2^{32}.

Pro nalezení inverzního zobrazení LCG je potřeba přeskládat jeho rovnici a využít převrácené hodnoty a v modulární aritmetice (anglicky *modular multiplicative inverse*) [61]:

$$\begin{aligned} X_{i+1} &\equiv aX_i + c && \pmod m \\ X_i &\equiv \frac{X_{i+1} - c}{a} && \pmod m \\ X_i &\equiv a^{-1}(X_{i+1} - c) && \pmod m \\ X_i &= a^{-1}(X_{i+1} - c) \end{aligned}$$

Kde a^{-1} je převrácená hodnota a v modulární aritmetice a je definovaná jako:

$$a \times a^{-1} \equiv 1 \pmod m$$

Můžeme ji nalézt pomocí rozšířeného Eukleidova algoritmu (anglicky *extended Euclidean algorithm*) [62], viz algoritmus 2. Jedná se o algoritmus, kterým lze nalézt největší společný dělitel (nsd) dvou celých čísel o a p a koeficienty Bézoutovy rovnosti, což jsou celá čísla x a y, kde:

$$ox + py = \text{nsd}(o, p)$$

Kde:

$$o \in \mathbb{N}^+, p \in \mathbb{N}^0, 0 \leq p \leq o$$

a^{-1} existuje, pokud $\text{nsd}(a, m) = 1$ a je rovno $x \bmod m$.

Algoritmus 2: Převrácená hodnota v modulární aritmetice pomocí rekurzivní implementace rozšířeného Eukleidova algoritmu

```
1  function Rozšířený-Eukleidův-Algoritmus(o, p)
2      if o = 0 then
3          return (p, 0, 1)
4      else
5          (g, y, x) ← Rozšířený-Eukleidův-Algoritmus(p mod o, o)
6          return (g, x − y⌊p/o⌋, y)
7      end
8  end
9  function Převrácená-Hodnota-V-Modulární-Aritmetice(a, m)
10     (g, x, y) ← Rozšířený-Eukleidův-Algoritmus(a, m)
11     if g = 1 then
12         return x mod m
13     end
14     ▷ Funkce je definovaná pouze v případě, že g = nsd(a, m) = 1.
15 end
```

5.3.2. S-box

S-box (zkratka pro *substitution box*) je z matematického hlediska funkce, která přeměňuje vstup v m-bitovém prostoru na výstup v n-bitovém [63]:

$$S : \{0,1\}^m \to \{0,1\}^n$$

Inverzní zobrazení existuje pouze tehdy, když platí $m = n$ a zároveň je S-box bijektivní. Bijektivnost znamená, že S-box je současně injektivní (prosté zobrazení) — pokud $S(x_1) = S(x_2)$, pak nutně $x_1 = x_2$ a surjektivní (zobrazení na) — S-box pokrývá všech 2^n možných výstupních hodnot. Inverzní S-box S^{-1} pak splňuje $S^{-1}(S(x)) = x$ pro všechna $x \in \{0, ..., 2^{n-1}\}$.

S-boxy se používají zejména v kryptografii a jsou jedním ze základních stavebních prvků symetrických šifer. V praxi jsou obvykle implementovány jako vyhledávací tabulky (anglicky *lookup table*, zkratka *LUT*).

Nvdisasm používá 8×8-bitový S-box. Ten se nepodařilo identifikovat s žádným z následujících veřejně známých S-boxů stejné velikosti:

- Advanced Encryption Standard (AES)/Rijndael S-box (ISO/IEC 18033-3:2010, FIPS 197) [64, 65]
- S-box blokové šifry Camellia (ISO/IEC 18033-3:2010, RFC 3713) [66, 67]
- S-boxy blokové šifry ARIA (KS X 1213:2004, RFC 5794) [68]
- S-boxy blokové šifry SEED (ISO/IEC 18033-3:2010, RFC 4269) [69]

- S-box blokové šifry Kuznyechik (GOST R 34.12-2015, RFC 7801) [70] a hašovací funkce Streebog (GOST R 34.11-2012, RFC 6986) [71]

Vyhledávání pomocí internetových vyhledávačů Google [53], DuckDuckGo [54], Baidu [55] a Yandex [56] rovněž nepřineslo žádné výsledky. Je proto pravděpodobné, že nvdisasm používá vlastní S-box, který může být i náhodně generovaný.

5.3.3. Bitová nonekvivalence

Exkluzivní disjunkce (anglicky *exclusive or*, zkratka *XOR*, značka $\oplus$) je komutativní binární logická operace, která nabývá pravdivé hodnoty právě tehdy, když je pravdivý pouze jeden z jejích operandů. Určíme-li, že 0 reprezentuje nepravdu a 1 pravdu, pak tato operace funguje stejně jako sčítání v dvouprvkovém konečném tělesu (dvouprvkovém Galoisově tělesu, značené GF(2)). Exkluzivní disjunkce je také involucí, což znamená, že je sama sobě inverzním zobrazením:

$$A = (A \oplus B) \oplus B$$

Z této vlastnosti vyplývá:

$$A = B \oplus C \iff B = A \oplus C$$

Bitová nonekvivalence (anglicky *bitwise XOR*, značka $\oplus$) je komutativní binární operace, která aplikuje exkluzivní disjunkci na každou dvojici odpovídajících bitů v binární reprezentaci obou operandů. Jelikož je exkluzivní disjunkce involucí, je involucí i bitová nonekvivalence.

5.3.4. Inverzní zobrazení dešifrovací funkce

Šifrovací funkce je inverzním zobrazením k dešifrovací funkci. Vstupem jsou hodnoty x, y, l odpovídající výstupním hodnotám dešifrovací funkce a uspořádaná n-tice dešifrovaných bajtů $D = (d_1, d_2, ..., d_n)$. Výstupem je inicializační hodnota x a uspořádaná n-tice šifrovaných bajtů $E = (e_1, e_2, ..., e_n)$, které lze použít jako vstup dešifrovací funkce pro získání původních hodnot.

Inverzní zobrazení dešifrovací funkce lze odvodit systematickým zpětným průchodem původního algoritmu. Tento postup je možný díky invertibilitě použitých kryptografických primitiv, konkrétně inverznímu zobrazení LCG popsanému v kapitole 5.3.1, inverznímu S-boxu S^{-1} popsanému v kapitole 5.3.2 a vlastnosti involuce bitové nonekvivalence popsané v kapitole 5.3.3. Výsledný šifrovací algoritmus využívá funkci pro výpočet převrácené hodnoty v modulární aritmetice definovanou v algoritmu 2.

Alternativně by bylo možné inverzní funkci získat automaticky pomocí SMT řešiče (anglicky *satisfiability modulo theories solver* nebo pouze *SMT solver*) jako Z3 [72].

Algoritmus 3: Šifrovací algoritmus

```
1  S^{-1}[256] = {0x44, 0xD2, ..., 0x09}   ▷ Inverzní S-box jako LUT, viz kapitola 5.3.2.
2  a = 0x41C64E6D                          ▷ LCG násobitel, viz kapitola 5.3.1.
3  c = 0x3039                              ▷ LCG inkrement, viz kapitola 5.3.1.
4  D = (d_1, d_2, ..., d_n))               ▷ Vstupní bajty jako uspořádaná n-tice.
5  E = (e_1, e_2, ..., e_n))               ▷ Výstupní šifrované bajty jako uspořádaná n-tice.
6  function Šifrovat(x, y, l, D)
7      a^{-1} ← Převrácená-Hodnota-V-Modulární-Aritmetice(a, 2^32)
8      e_n ← l
9      for i ∈ ⟨n; 2⟩ ∩ ℕ do
10         l ← (S^{-1}[(d_i ⊕ x) mod 2^8] ⊕ l) mod 2^8
11         e_{i-1} ← l
12         if (i − 1) mod 4 = 0 then
13             y ← x
14             x ← a^{-1}(y − c) mod 2^32
15         else
16             y ← y ≪ 8
17         end
18     end
19     return (x, E)
20 end
```

$\ll$	Logický bitový posuv doleva, $x \ll n = x \times 2^n$.
$\oplus$	Bitová nonekvivalence, viz kapitola 5.3.3.

6. Souborový formát fatbin

Fatbin (zkratka pro *fat binary*) je kontejnerový formát používaný platformou CUDA pro sdružování více variant kompilovaného kódu pro GPU do jednoho souboru [73]. Tento přístup umožňuje aplikacím podporovat více výpočetních možností (anglicky *compute capability*) bez nutnosti dodávat samostatné binární soubory pro každou cílovou architekturu GPU nebo rekompilace. Za běhu aplikace CUDA runtime vybere nejvhodnější variantu kódu pro aktuálně přítomný hardware.

Struktura fatbin souboru se skládá z hlavičky fatbin, po které následuje sekvence sekcí. Každá sekce obsahuje vlastní hlavičku a data. Data sekce mohou nabývat následujících typů:

- Cubin soubor obsahující zkompilovaný GPU strojový kód pro konkrétní výpočetní možnost.
- PTX (Parallel Thread Execution) — nízkoúrovňový mezikód v textové podobě nezávislý na konkrétní architektuře GPU, který je kompilován do strojového kódu za běhu [74].
- NVVM IR — mezikód založený na LLVM IR [75].

Sekce mohou být volitelně komprimovány algoritmem LZ4 (Lempel-Ziv 4) [49] nebo zstd (Zstandard) [76].

V kontextu hostitelských spustitelných souborů ve formátu ELF jsou fatbin data typicky vložena do sekce .nv_fatbin a registrována prostřednictvím segmentu .nvFatBinSegment. Tento mechanismus umožňuje CUDA runtime za běhu lokalizovat a načíst příslušné GPU binární soubory.

7. Architektura

Projekt se skládá ze tří hlavních modulů. Architektura je navržena tak, aby bylo možné jednotlivé moduly v mnoha případech použít nezávisle.

Parsery binárních formátů cubin (struct `Cubin`) a fatbin (struct `Fatbin`) zpracovávají CUDA binární soubory. Parser cubin souborů extrahuje z ELF struktury CUDA-specifické sekce, symboly, metadata a relokace. Parser fatbin souborů načítá fatbin ze souboru nebo paměťového bufferu, extrahuje cubin soubory pro cílovou architekturu GPU a podporuje dekomprimaci sekcí komprimovaných algoritmy LZ4 [49] a zstd [76].

Modul ovladače (struct `Driver`) zajišťuje komunikaci s NVIDIA ovladači zařízení prostřednictvím RM API a UVM API popsaných v kapitole 4. Poskytuje funkce pro inicializaci GPU, alokaci paměti, správu virtuálního adresního prostoru a synchronizaci.

Příkazové fronty (struct `CommandQueue`) slouží k sestavování sekvencí příkazů pro GPU. Příkazy jsou kódovány do pushbufferů, jejichž virtuální adresy se zapisují do kruhového bufferu GPFifo (GPU FIFO). Zápis do MMIO registru GPU (doorbell) signalizuje dostupnost nových příkazů. Příkazová fronta podporuje dva typy: výpočetní (compute) pro spouštění kernelů a DMA pro přenosy dat. Pro odeslání příkazů na GPU je vyžadována instance `Driver`. Struktura QMD pro spouštění kernelů obsahuje virtuální adresu obrazu programu, počet registrů, velikost sdílené paměti, velikost bloku a mřížky (anglicky *grid*) a virtuální adresy konstantních bufferů s parametry. Projekt podporuje formáty QMD od architektury Turing po Blackwell.

Projekt poskytuje nativní Zig API pro software psaný v tomto jazyce. Pro interoperabilitu s jinými jazyky je k dispozici C API s hlavičkovým souborem. Sestavením do statické knihovny je možné projekt využít z libovolného programovacího jazyka podporujícího volání funkcí s C ABI.

8. Metody testování

Parsery formátů cubin a fatbin jsou testovány metodou fuzz testování (fuzzing), která spočívá v automatickém generování vstupních dat s cílem odhalit chyby jako paměťové úniky nebo neplatné chování. Jazyk Zig poskytuje integrovanou podporu pro fuzz testování.

Projekt dále obsahuje funkční testy ověřující správnost parsování reálných souborů a komunikace s ovladači zařízení. Tyto testy současně slouží jako ukázkové příklady použití knihovny.

Testování interakce s GPU hardwarem je omezeno dostupností fyzických zařízení, protože pro NVIDIA GPU neexistuje softwarová emulace umožňující spouštění CUDA kernelů. Projekt byl proto testován výhradně na grafické kartě NVIDIA GeForce RTX 4060 Laptop GPU založené na mikroarchitektuře Ada Lovelace [77].

Build systém projektu podporuje automatické testování na odlišných CPU architekturách prostřednictvím emulátoru QEMU, pokud je nainstalovaný a dostupný.

Výkonnostní benchmarky projekt nezahrnuje, protože primárně zajišťuje komunikaci s ovladači zařízení, což většinou nepředstavuje významnou část celkové doby běhu CUDA aplikací a není kritické z hlediska výkonu. Očekávaný výkonnostní profil je srovnatelný s CUDA Driver API.

9. Možná rozšíření projektu

Prvním možným rozšířením je podpora akcelerovaného kódování a dekódování videa a obrázků. NVIDIA GPU obsahují dedikované jednotky NVENC (zkratka pro *NVIDIA Encoder*) pro kódování videa, NVDEC (zkratka pro *NVIDIA Decoder*) pro dekódování videa [78] a jednotku pro akcelerované zpracování obrázků ve formátu JPEG [79].

Druhým možným směrem je reverzní inženýrství instrukční sady SASS na základě dešifrovaných dat z nvdisasm (viz kapitola 5.2) a implementace assembleru pro ni.

10. Závěr

Cílem této práce bylo vytvořit open-source alternativu k proprietárnímu CUDA Driver API s využitím technik reverzního inženýrství. Tohoto cíle bylo dosaženo implementací knihovny v jazyce Zig, která poskytuje parsery binárních formátů cubin a fatbin a rozhraní pro komunikaci s NVIDIA ovladači zařízení.

V rámci práce byla analyzována šifrovaná data v binárních souborech nástrojů platformy CUDA. Podařilo se identifikovat použitou dešifrovací funkci, což umožnilo získat kompletní definice CUDA-specifických typů relokací. Tyto definice jsou nezbytné pro správné načítání cubin souborů a dosud nebyly v žádném open-source projektu plně přítomné. Rovněž bylo odvozeno inverzní zobrazení dešifrovací funkce, které umožňuje modifikaci šifrovaných dat.

Výsledná knihovna umožňuje alokovat paměť na GPU, načítat cubin soubory a spouštět výpočetní kernely bez závislosti na proprietárním CUDA Driver API. Projekt tak přispívá k hlubšímu porozumění interním mechanismům platformy CUDA a otevírá možnosti pro další výzkum a vývoj v oblasti open-source GPGPU.

Reference

[1] LINDHOLM, Erik, John NICKOLLS, Stuart OBERMAN a John MONTRYM. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* [online]. 2008, **28**(2), 39–55. Dostupné z: doi:10.1109/MM.2008.31

[2] NVIDIA CORPORATION. *CUDA Driver API Documentation* [online]. [vid. 2026-03-15]. Dostupné z: https://docs.nvidia.com/cuda/cuda-driver-api/index.html

[3] NVIDIA CORPORATION. *CUDA Runtime API Documentation* [online]. [vid. 2026-03-15]. Dostupné z: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[4] NVIDIA CORPORATION. *Difference between CUDA Driver and Runtime APIs* [online]. [vid. 2026-03-15]. Dostupné z: https://docs.nvidia.com/cuda/cuda-driver-api/driver-vs-runtime-api.html

[5] MIKEX86. *LibreCUDA* [online]. [vid. 2026-03-15]. Dostupné z: https://github.com/mikex86/LibreCuda

[6] TINY CORP. *tinygrad NVIDIA runtimes* [online]. [vid. 2026-03-15]. Dostupné z: https://github.com/tinygrad/tinygrad/blob/master/tinygrad/runtime/ops_nv.py

[7] KATO, Shinpei, Michael MCTHROW, Carlos MALTZAHN a Scott BRANDT. Gdev: First-Class GPU Resource Management in the Operating System. In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)* [online]. Boston, MA: USENIX Association, 2012, s. 401–412. ISBN 978-931971-93-5. Dostupné z: https://www.usenix.org/conference/atc12/technical-sessions/presentation/kato

[8] KATO, Shinpei, Michael MCTHROW, Carlos MALTZAHN a Scott BRANDT. *Gdev: Open-Source GPGPU Runtime and Driver Software* [online]. [vid. 2026-03-15]. Dostupné z: https://github.com/shinpei0208/gdev

[9] X512. *Porting Nvidia driver for Turing+ GPUs to Haiku* [online]. březen 2025 [vid. 2026-03-15]. Dostupné z: https://discuss.haiku-os.org/t/haiku-nvidia-porting-nvidia-driver-for-turing-gpus/16520

[10] ZIG SOFTWARE FOUNDATION. *Zig Programming Language* [online]. [vid. 2026-03-15]. Dostupné z: https://ziglang.org/

[11] THE RUST FOUNDATION. *Rust Programming Language* [online]. [vid. 2026-03-15]. Dostupné z: https://rust-lang.org/

[12] NIXOS FOUNDATION. *NixOS: The Purely Functional Linux Distribution* [online]. [vid. 2026-03-15]. Dostupné z: https://nixos.org/

[13] OXALICA. *rust-overlay: Pure and reproducible nix overlay of binary distributed rust toolchains* [online]. [vid. 2026-03-15]. Dostupné z: https://github.com/oxalica/rust-overlay

[14] PETKOV, Ivan. *crane: A Nix library for building cargo projects* [online]. [vid. 2026-03-15]. Dostupné z: https://github.com/ipetkov/crane

[15] HASHIMOTO, Mitchell. *zig-overlay: Nix Flake for Zig* [online]. [vid. 2026-03-15]. Dostupné z: https://github.com/mitchellh/zig-overlay

[16] OLLIE, Jeffrey C. *zon2nix* [online]. [vid. 2026-03-15]. Dostupné z: https://github.com/jcollie/zon2nix

[17] VECTOR 35. *Binary Ninja, verze 5.2 free* [online]. [vid. 2026-03-15]. Dostupné z: https://binary.ninja/

[18] FREE SOFTWARE FOUNDATION. *GDB: The GNU Project Debugger* [online]. [vid. 2026-03-15]. Dostupné z: https://sourceware.org/gdb/

[19] PYTHON SOFTWARE FOUNDATION. *Python Language Reference, verze 3.14* [online]. [vid. 2026-03-15]. Dostupné z: http://python.org/

[20] SHOSHITAISHVILI, Yan, Ruoyu WANG, Christopher SALLS, Nick STEPHENS, Mario POLINO, Audrey DUTCHER, John GROSEN, Siji FENG, Christophe HAUSER, Christopher KRUEGEL a Giovanni VIGNA. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: *IEEE Symposium on Security and Privacy*. 2016.

[21] COMPUTER SECURITY LAB AT UC SANTA BARBARA, their associated CTF team, Shellphish, the open source community, SEFCOM at Arizona State University a RHELMOT. *angr: A powerful and user-friendly binary analysis platform* [online]. [vid. 2026-03-15]. Dostupné z: https://github.com/angr/angr

[22] BELLARD, Fabrice a QEMU TEAM. *QEMU* [online]. [vid. 2026-03-15]. Dostupné z: https://www.qemu.org/

[23] NVIDIA CORPORATION. *NVIDIA Linux Open GPU Kernel Module Source* [online]. [vid. 2026-03-15]. Dostupné z: https://github.com/NVIDIA/open-gpu-kernel-modules

[24] NVIDIA CORPORATION. *GSP Firmware* [online]. [vid. 2026-03-15]. Dostupné z: https://download.nvidia.com/XFree86/Linux-x86_64/595.58.03/README/gsp.html

[25] X.ORG FOUNDATION. *Nouveau: Accelerated Open Source driver for NVIDIA GPUs* [online]. [vid. 2026-03-15]. Dostupné z: https://nouveau.freedesktop.org/

[26] NVIDIA CORPORATION. *Documentation of NVIDIA chip/hardware interfaces* [online]. [vid. 2026-03-15]. Dostupné z: https://github.com/NVIDIA/open-gpu-doc

[27] NVIDIA CORPORATION. *CUDA Programming Guide: Unified Memory* [online]. [vid. 2026-03-15]. Dostupné z: https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/unified-memory.html

[28] HARRIS, Mark. *Unified Memory for CUDA Beginners* [online]. 19. červen 2017 [vid. 2026-03-15]. Dostupné z: https://developer.nvidia.com/blog/unified-memory-cuda-beginners

[29] KRANENBURG, Paul, Branko LANKESTER a Rick SLADKEY. *strace: the linux syscall tracer* [online]. [vid. 2026-03-15]. Dostupné z: https://strace.io/

[30] CESPEDES, Juan. *ltrace* [online]. [vid. 2026-03-15]. Dostupné z: https://ltrace.org/

[31] GUILLEMARD, Mary. *envyhooks: hooking library to dump command buffers from the NVIDIA drivers* [online]. [vid. 2026-03-15]. Dostupné z: https://gitlab.freedesktop.org/nouveau/envyhooks

[32] HOTZ, George. *cuda_ioctl_sniffer* [online]. [vid. 2026-03-15]. Dostupné z: https://github.com/geohot/cuda_ioctl_sniffer

[33] NVIDIA CORPORATION. *CUDA Binary Utilities Documentation* [online]. [vid. 2026-03-15]. Dostupné z: https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html

[34] FREE SOFTWARE FOUNDATION. *GNU Binutils* [online]. [vid. 2026-03-15]. Dostupné z: https://gnu.org/software/binutils/

[35] FREE SOFTWARE FOUNDATION. *GNU Binutils Documentation: readelf Utility* [online]. [vid. 2026-03-15]. Dostupné z: https://sourceware.org/binutils/docs/binutils/readelf.html

[36] FREE SOFTWARE FOUNDATION. *GNU Binutils Documentation: objdump Utility* [online]. [vid. 2026-03-15]. Dostupné z: https://sourceware.org/binutils/docs/binutils/objdump.html

[37] LLVM DEVELOPER GROUP. *LLVM User Guide for NVPTX Back-end: NVPTX Architecture Hierarchy and Ordering* [online]. [vid. 2026-03-15]. Dostupné z: https://llvm.org/docs/NVPTXUsage.html#nvptx-architecture-hierarchy-and-ordering

[38] XINUOS INC. *Generic ABI: Executable and Linkable Format (ELF) Specification* [online]. [vid. 2026-03-15]. Dostupné z: https://gabi.xinuos.com/

[39] XINUOS INC. *System V Application Binary Interface Edition 4.1* [online]. [vid. 2026-03-15]. Dostupné z: https://sco.com/developers/gabi/

[40] XINUOS INC. *System V Application Binary Interface Edition 4.1* [online]. [vid. 2026-03-15]. Dostupné z: https://sco.com/developers/devspecs/gabi41.pdf

[41] HAYES, Ari B., Fei HUA, Jin HUANG, Yanhao CHEN a Eddy Z. ZHANG. *Decoding CUDA Binary - File Format* [online]. únor 2019. Dostupné z: doi:10.5281/zenodo.2339027

[42] HAYES, Ari B., Fei HUA, Jin HUANG, Yanhao CHEN a Eddy Z. ZHANG. *Decoding CUDA Binary artifact* [online]. prosinec 2018. Dostupné z: doi:10.5281/zenodo.2337060

[43] 哈哈波. *CUDA Launch Kernel*的流程解析 [online]. 15. říjen 2025 [vid. 2026-03-15]. Dostupné z: https://zhuanlan.zhihu.com/p/1961519233591674250

[44] XINUOS INC. *Generic ABI: Executable and Linkable Format (ELF) Specification: Relocation* [online]. [vid. 2026-03-15]. Dostupné z: https://gabi.xinuos.com/elf/06-reloc.html

[45] FREE SOFTWARE FOUNDATION. *GNU Binutils Documentation: strings Utility* [online]. [vid. 2026-03-15]. Dostupné z: https://sourceware.org/binutils/docs/binutils/strings.html

[46] REDPLAIT. *denvdis: NVIDIA SASS Disassembler and Related Tools* [online]. [vid. 2026-03-15]. Dostupné z: https://github.com/redplait/denvdis

[47] REDPLAIT. *ptx internals* [online]. 24. únor 2025 [vid. 2026-03-15]. Dostupné z: https://redplait.blogspot.com/2025/02/ptx-internals.html

[48] REDPLAIT. *nvidia sass disassembler* [online]. 6. březen 2025 [vid. 2026-03-15]. Dostupné z: https://redplait.blogspot.com/2025/03/nvidia-sass-disassembler.html

[49] COLLET, Yann. *LZ4: Extremely Fast Compression Algorithm* [online]. [vid. 2026-03-15]. Dostupné z: https://lz4.org/

[50] 0XD0GF00D. *DocumentSASS: Unofficial description of the CUDA assembly (SASS) instruction sets.* [online]. [vid. 2026-03-15]. Dostupné z: https://github.com/0xD0GF00D/DocumentSASS

[51] NVIDIA CORPORATION. *CUDA Programming Guide: Cooperative Groups* [online]. [vid. 2026-03-15]. Dostupné z: https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cooperative-groups.html

[52] HARRIS, Mark a Kyrylo PERELYGIN. *Cooperative Groups: Flexible CUDA Thread Programming* [online]. 4. srpen 2017 [vid. 2026-03-15]. Dostupné z: https://developer.nvidia.com/blog/cooperative-groups

[53] *Google* [online]. [vid. 2026-03-15]. Dostupné z: https://google.com/

[54] *DuckDuckGo* [online]. [vid. 2026-03-15]. Dostupné z: https://duckduckgo.com/

[55] *Baidu Search* [online]. [vid. 2026-03-15]. Dostupné z: https://baidu.com/

[56] *Yandex Search* [online]. [vid. 2026-03-15]. Dostupné z: https://yandex.com/

[57] THOMSON, W. E. A Modified Congruence Method of Generating Pseudo-random Numbers. *The Computer Journal* [online]. 1958, **1**(2), 83. ISSN 0010-4620. Dostupné z: doi:10.1093/comjnl/1.2.83

[58] KNUTH, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3. vyd. B.m.: Addison-Wesley, 1981. ISBN 0-201-03822-6.

[59] GNU PROJECT. *GNU C Library (glibc)* [online]. [vid. 2026-03-15]. Dostupné z: https://sourceware.org/glibc/

[60] GNU PROJECT. *GNU C Library (glibc) zdrojový kód, funkce rand* [online]. [vid. 2026-03-15]. Dostupné z: https://sourceware.org/git/?p=glibc.git;a=blob;f=stdlib/random_r.c;hb=glibc-2.42#l381

[61] NAYUKI. *Fast skipping in a linear congruential generator* [online]. 18. červen 2016 [vid. 2026-03-15]. Dostupné z: https://nayuki.io/page/fast-skipping-in-a-linear-congruential-generator

[62] MCCONNELL, R.M., K. MEHLHORN, S. NÄHER a P. SCHWEITZER. Certifying algorithms. *Computer Science Review* [online]. 2011, **5**(2), 119–161. ISSN 1574-0137. Dostupné z: doi:https://doi.org/10.1016/j.cosrev.2010.09.009

[63] MEGHANATHAN, N., B.K. KAUSHIK a D. NAGAMALAI. *Advances in Networks and Communications: First International Conference on Computer Science and Information Technology, CCSIT 2011, Bangalore, India, January 2-4, 2011. Proceedings* [online]. B.m.: Springer Berlin Heidelberg, 2010. Communications in Computer and Information Science. ISBN 978-3-642-17877-1. Dostupné z: https://books.google.cz/books?id=pXOS4ZTUJLYC

[64] DAEMEN, Joan a Vincent RIJMEN. *The design of Rijndael: AES — the Advanced Encryption Standard*. B.m.: Springer-Verlag, 2002. ISBN 3-540-42580-2.

[65] *Advanced Encryption Standard (AES)* [online]. 2023. Dostupné z: doi:10.6028/NIST.FIPS.197-upd1

[66] AOKI, Kazumaro, Tetsuya ICHIKAWA, Masayuki KANDA, Mitsuru MATSUI, Shiho MORIAI, Junko NAKAJIMA a Toshio TOKITA. The 128-Bit Block Cipher Camellia. *IEICE TRANSACTIONS on Fundamentals*. 2002, (1), 11–24.

[67] MATSUI, Mitsuru, Shiho MORIAI a Junko NAKAJIMA. *A Description of the Camellia Encryption Algorithm* [online]. duben 2004. Dostupné z: doi:10.17487/RFC3713

[68] KIM, Jaeheon, Jooyoung LEE, Choonsoo KIM, Jungkeun LEE a Daesung KWON. *A Description of the ARIA Encryption Algorithm* [online]. březen 2010. Dostupné z: doi:10.17487/RFC5794

[69] YOON, Jaeho, Sungjae LEE, d.h. CHEON, Jaeil LEE a Hyangjin LEE. *The SEED Encryption Algorithm* [online]. prosinec 2005. Dostupné z: doi:10.17487/RFC4269

[70] DOLMATOV, Vasily. *GOST R 34.12-2015: Block Cipher "Kuznyechik"* [online]. březen 2016. Dostupné z: doi:10.17487/RFC7801

[71] DOLMATOV, Vasily a Alexey DEGTYAREV. *GOST R 34.11-2012: Hash Function* [online]. srpen 2013. Dostupné z: doi:10.17487/RFC6986

[72] MICROSOFT CORPORATION. *Z3 Theorem Prover* [online]. [vid. 2026-03-15]. Dostupné z: https://github.com/Z3Prover/z3

[73] EILING, Niklas. *CUDA Fatbin Decompression* [online]. [vid. 2026-03-15]. Dostupné z: https://github.com/n-eiling/cuda-fatbin-decompression

[74] NVIDIA CORPORATION. *Parallel Thread Execution ISA* [online]. [vid. 2026-03-15]. Dostupné z: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[75] NVIDIA CORPORATION. *NVVM IR Specification* [online]. [vid. 2026-03-15]. Dostupné z: https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html

[76] COLLET, Yann a Murray KUCHERAWY. *Zstandard Compression and the application/zstd Media Type* [online]. říjen 2018. Dostupné z: doi:10.17487/RFC8478

[77] NVIDIA CORPORATION. *NVIDIA Ada Lovelace Architecture* [online]. [vid. 2026-03-15]. Dostupné z: https://www.nvidia.com/en-us/geforce/ada-lovelace-architecture/

[78] NVIDIA CORPORATION. *NVIDIA Video Codec SDK* [online]. [vid. 2026-03-15]. Dostupné z: https://developer.nvidia.com/video-codec-sdk

[79] NVIDIA CORPORATION. *nvJPEG: GPU-accelerated JPEG decoder, encoder and transcoder* [online]. [vid. 2026-03-15]. Dostupné z: https://developer.nvidia.com/nvjpeg