OpenCL alapú eszközök verifikációja és validációja a gyakorlatban

Fekete Tamás

2015. December 3.

Szoftver verifikáció és validáció tantárgy

BME AUT
Department of Automation and Applied Informatics

Áttekintés

- Miért és mennyire fontos a megfelelő validáció és verifikáció OpenCL alapú eszközök esetében a tapasztalataim alapján?
  - Miért nagy kihívás?
  - Milyen gyakorlati megoldásokat találtam
- **A folyamatos Verifikáció és validáció ma már a fejlesztés és kutatás során is alapvető eszköz,** hiszen az igények folyamatosan változnak menet közben *(pl. kutatás eredmények szerint vagy Agilis szoftver fejlesztés során ügyfél kérésére)*
- Túlságosan formális módszerek algoritmusok fejlesztésére/kutatásra
  - Lassú és összetettebb problémák esetén túl bonyolúlt
  - Másnak nehéz megérteni, javítani
- ***Gyakorlatorientált verifikációt és validációt vizsgálom ebben a munkában***

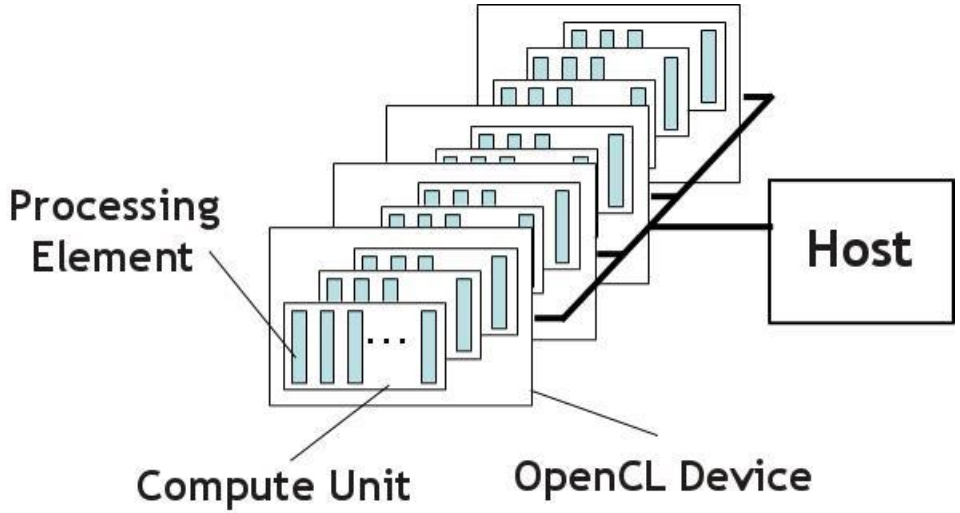
OpenCL keretrendszer

- **Csak egy interfészt definiál (*C nyelven*),** Khronos Group adja ki
- **Célja több hardver egy közös, egységes interfészen történő elérése**
- A hardvergyártók maguk végzik a megvalósítást

- Eltérés lehet a működésben
  - Elsősorban a **teljesítmény** optimalizációjában mutatkozik meg
  - A gyártók **különböző verzióját** támogatják a keretrendszernek
    (pl. NVIDIA GPU: 1.2, AMD GPU: 2.1)
  - *Többnyire grafikus vezérlők esetén használják, de támogják FPGA és DSP gyártók is*

OpenCL programozási modell (2/1)

- A verifikáció és validáció szempontjából is lényeges *(ábra)*
- Elsősorban **párhuzamosan futó algoritmusok** fejlesztésére készült
- Fontos, hogy az *OpenCL kód* és *adat* nem a „host” memóriában van/fut, hanem az OpenCL keretrendszer által kezelt célhardveren
  - **Végrehajtás elött át kell másolni az adatokat (opcionális) és a binárist**

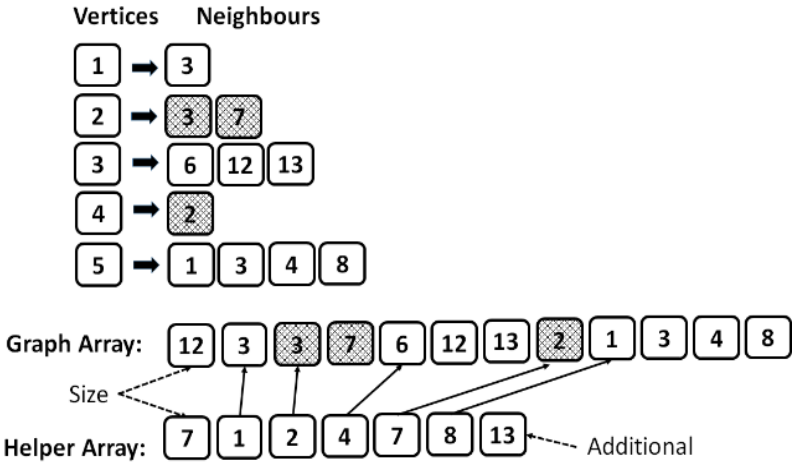


OpenCL programozási modell (2/2)

- A célhardeveren futó forráskód nem része a „host” forráskódjának, általában **külön fájlban is van**
- Futásidőben a *„host”* az OpenCL keretrendszer interfészein keresztűl az OpenCL forrásból készít egy **binárist** és tölti le a céleszközre
- **Nincs lehetőség a „szokásos” debuggolási lehetőségekre**
  - pl. tetszőleges IDE-k általi „lépekedés” a kódban
  - A legtöbb gyártó biztosít valamilyen debuggolási lehetőséget, de gyakran komplikáltak és egymástól eltérőek
- Az OpenCL-ben írt kódot *(függvényét kernelnek hívjuk)*, C nyelven kell írni kevés megszorítás mellett

Be- és kimeneti adatok (OpenCL 1.2)

- **Csak 1 dimenziós** tömböt lehet másolni *(mutató tömb sem megengedett)*
- **A bemeneti adatokat le kell képezni**
  - **Verifikiálni és validálni kell az alkalmazott módszert**
  - Funckionalitást tesztelem, visszafele is átalakítom az adatokat
- Kimeneti adatokat szintén vissza kell másolni a hostra, ahhoz, hogy fel lehessen dolgozni azokat
  - Tesztadatokat kell visszamásolni és feldolgozni
- **ADATÁRAMLÁS ellenőrzése! Ehhez szoftver adatmodelt készítek.**



Tervezés (2/1)

- **Gyakorlati példa:** mintakeresés gráfokban
- **Automata és egyéb matematikai módszerek**
  - **Automata implementáció készűlt is**
- **Interface alapú tervezés**, nemcsak a legfelső, hanem közbenső szinteken is interfacekkel dolgozom a host kódban
  - Jól definiálható elvárások az egyes modulokkal szemben
  - Belső implementáció változtatása esetén a ráépülő modulokat nem kell javítani
- **Pszeudókód készítése, mivel itt a kernel algoritmus a legfontosabb és a legkritikusabb elem**

Tervezés (2/2)

- Memória management fontos a kernel kódban (*sok hibalehetőség*):
  - dinamikusan nem lehet memóriát allokálni
- Többnyire a programozási modellekben **4 memóriaterületet** különböztetünk meg:
  - host, globális, privát és konstans értékeknek *(az utolsó az eszköz globális memóriaterületén belül)*
- **Skálázhatóság** lényeges szempont, mivel nem lehet tudni, hogy mikor fogy el a memória
  - Verifikálom és tesztelem **(kézzel korlátozom a foglalható memória méretét) -> ugyanolyan eredménynek kell lennie korlátozás nélkűl is**
  - A grafikus vezérlők többnyire kisebb memóriával rendelkeznek mint a host

Hibakezelés

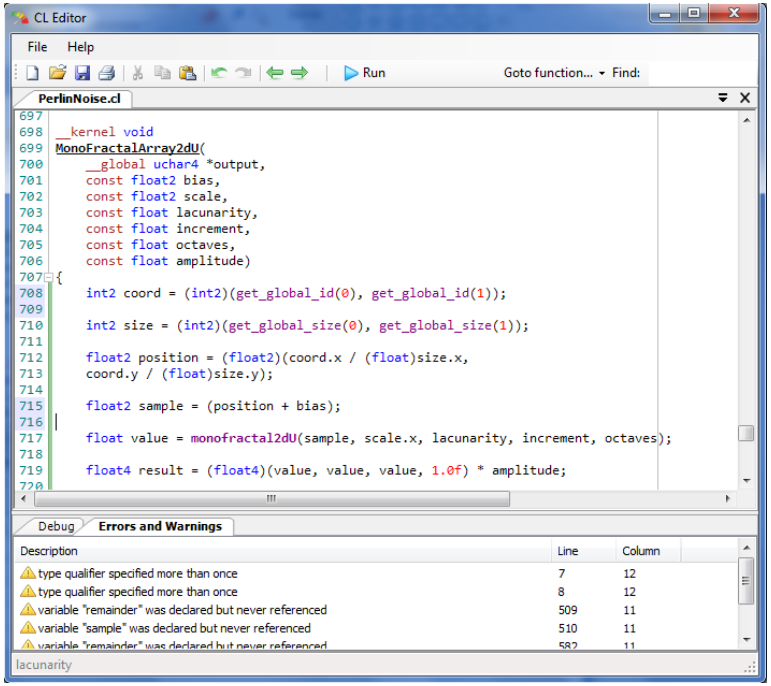
- **Elsődlegesen** az operációs rendszer egy idő után, ha nem válaszól a grafikus vezérlő, akkor **„reset”-eli azt és tájékoztat a „hibáról”**
- Az OpenCL **keretrendszer** is tud hibás működést észrevenni és hibakóddal visszatérni
  - A specifikáció tartalmazza a hibakódok jelentéseit
  - Gyakran nem elég specifikusak
- **Működés közben saját hibakódokat definiálok és kimenő paraméterként visszaadom**
  - **A kódnak le kell futnia**
  - **Funkcionális hibák megtalálására alkalmas**
  - ***pl. túl kicsi a kimeneti tömb, nincs több hely benne***

Statikus és dinamikus ellenőrzés

- **Statikus kódelemzésre ingyenes online ellenőrzés is elérhető, de számtalan ingyenes alkalmazás is van**
  - **A jobbak már fizetősek**
- **Két dinamikus szintű tesztelést végzek:**
- **Unit/modul tesztek:** funkcionális követelmények ellenőrzésére használom, valamint fejlesztés során
- **Esettanulmányok:** a nem funkcionális követelmények ellenőrzésére
  - Nagy bemeneti adathalmaz
  - Futási idő és skálázhatóság vizsgálata

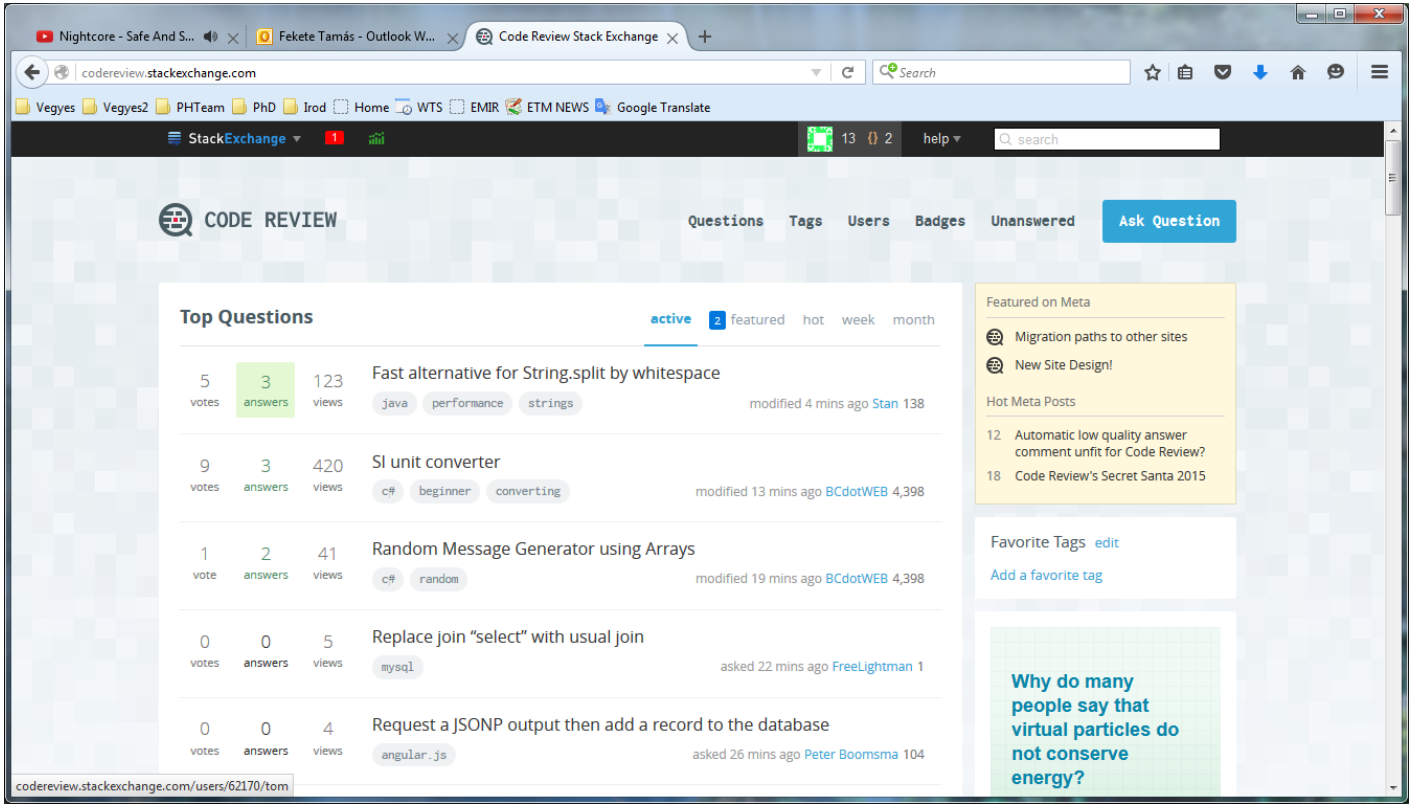
Statikus kód ellenőrzés

- **Számtalan program**, *pl. CLEEditor*
- **Online ellenőrzés**, *pl. Stream Computing*



Kód ellenőrzés - review

- **Kedvencem:**
  - https://codereview.stackexchange.com



Unit és modul tesztek

- **Funkcionális követelmények ellenőrzésére használom**, példák **(határérték ellenőrzése)**:
  - „Smoke” teszt új ötletek/implementáció elején
  - Egy elemet csak egyszer találjon meg keresés esetén
  - Ha nincs eredmény, akkor is jól működjön
  - Az összes extrém esetre egy-egy egyszerű példa
  - Negatív tesztesetek a tesztkörnyezet tesztelésére
- Rövid kód, gyorsan lefut, sok könyvtárban van támogatás hozzá *(pl. Boost)*

Esettanulmányok

- A nem-funkcionális követelmények tesztelésére használom
- **Nincsenek tesztesetek**
  - **Időbeli és memóriabeli korlátozás**
  - **Host verzióval hasonlítom össze**
- Nagy bemeneti adathalmaz:
  - Idő és memóriahasználat mérése
  - Mérési pontok elhelyezése
- **A legfontosabb szoftver jellemzők vizsgálata itt történik**:
  - skálázhatóság, megbízhatóság, hibakezelés, platform függetlenség

Teszt alapú fejlesztés (TDD)

- A unit és a modul tesztek már előre, az implementálás elött elkészűlnek
- **Egyszerre kezelem a teszteket és a célkódot,** így a folyamatos futtatással folyamatosan kapok visszajelzést, hogy a kód jól működik-e
- *Egyszerű példákat gyorsabban lehet debuggolni*
- **Kihívás:** előre pontosan tudni kell, hogy mit várok el az algoritmustól
  - > Kisebb algoritmusokkal szemben könnyen lehet előre tudni az elvárásokat

Automatizálás

- A Microsoft is támogat regressziós teszteket:
  - Ingyenesen lehet futtatni a TFS alatt tárolt forráskódokra
- „Egyszemélyes” fejlesztésnél nem hatékony, az eredmények kiértékelése alatt a teszt kézzel is könnyen futtatható
- Kisebb alkalmazások, algoritmusok esetén szintén nem szükséges

Dokumentáció

- **Előre készülő tervek**
  - **Optimalizáció:** a túlságosan rövid és a túlságosan hosszú tervezés is jelentősen megnöveli a megvalósítást
- **Utólagos használatra**
  - Segítik a régen írt kódokat megérteni
  - **Hibajavítás során már a dokumentáció erősen tudja sejtetni, hogy hol található a hiba**

Köszönöm a figyelmet!