Párhuzamos eszközök programozása beadandó feladat dokumentáció

Kórik Edina

GDVK5H

Tartalom

[A feladat 3](#_Toc197979425)

[Futtatás 3](#_Toc197979426)

[A wav fájlok beolvasása 3](#_Toc197979427)

[main.c 4](#_Toc197979428)

[Hibakeresés 5](#_Toc197979429)

[runner.c 6](#_Toc197979430)

[Mérési adatok 7](#_Toc197979431)

[Vektorizálás hatása 8](#_Toc197979432)

[Buffer mapping hatása 9](#_Toc197979433)

# A feladat

A beadandó feladat célja WAV formátumú hangfájlok elemzése. A fájl megfelelő beolvasását követően a bal és jobb csatorna hangmintái közötti különbség kerül vizsgálatra. Ezeket a különbségeket az OpenCL kernel adott elemszámú blokkokban átlagolja.

# Futtatás

A make parancs hatására a *main.c* és a *runner.c* forrásfájlok lefordításra kerülnek, és különálló futtatható állományok (main és runner) jönnek létre. Ezek a programok a main, illetve runner parancsokkal indíthatók el.

# A wav fájlok beolvasása

A WAV fájlok kezeléséhez használt adatszerkezet az *include* mappában a *wav.h* állományban található meg. A struktúra két fő részből épül fel: a fájl fejlécéből és az audioadatokat tartalmazó mintákból.

A fejléc tartalmazza a WAV formátum azonosításához és értelmezéséhez szükséges információkat, mégpedig:

* a fájl típusát jelző „RIFF” karakterláncot,
* a teljes fájlméretet,
* a „WAVE” azonosítót,
* a „fmt” szekció nevét,
* a formátumleíró szekció méretét,
* a hangformátum típusát (pl. PCM),
* a csatornák számát (mono vagy sztereó),
* a mintavételezési frekvenciát,
* a bájtsebességet,
* a blokkhangolást,
* valamint a mintánkénti bitszámot.

Az audioadatok egy tömbben tárolt mintákból állnak, ahol minden minta egy audio típusú struktúra. Ez a struktúra két 16 bites egész számot tartalmaz, amelyek rendre a bal és a jobb csatorna hangmintáit reprezentálják. A minták számát egy külön num\_samples változó jelöli.

A WAV fájlok beolvasását végző függvény a src mappában található wav.c fájlban található. A read\_wav\_file függvény feladata egy megadott fájl beolvasása és ellenőrzése. Először megnyitja a fájlt bináris olvasásra, majd beolvassa a WAV fejlécet. Ezután ellenőrzi, hogy a fejlécben szereplő azonosítók ("RIFF" és "WAVE") megfelelnek-e a szabványos WAV formátumnak.

Amennyiben a fájl érvényes, a függvény kiolvassa az audioadatokat is, és eltárolja azokat egy WAV típusú struktúrában. Az audioadatokat a data szekcióban keresi meg, és a fájlban szereplő blokkméret alapján meghatározza a hangminták számát. A program feltételezi, hogy a minták sztereó formátumban vannak, vagyis minden minta bal és jobb csatornát is tartalmaz.

A fejlécelemek opcionálisan kiíratásra is kerülhetnek, például a mintavételezési frekvencia, csatornaszám, valamint a bitsűrűség.

# main.c

Ez a C program egy WAV hangfájlt dolgoz fel OpenCL segítségével, párhuzamos számítási erőforrásokat kihasználva. A célja az, hogy a hangfájl bal és jobb csatornájának mintáiból megadott csoportonként átlagokat számoljon ki egy OpenCL kernel segítségével, majd az eredményeket visszatöltse a CPU memóriájába, és kiértékelje a futási időket.

A program az elején a parancssori argumentumokat vizsgálja meg. Egyetlen paramétert vár: a feldolgozandó minták számát csoportonként. Ha a felhasználó hibás számot ad meg (például negatív értéket vagy nem számot), a program hibát jelez és leáll.

Ezután beolvassa a remalomfold.wav nevű hangfájlt, amelynek mintáit a *read\_wav\_file* függvény tölti be egy WAV típusú struktúrába. A fájl teljes mintaszámából és a megadott minták/csoport arányból kiszámítja a feldolgozandó csoportok számát. Ez azért fontos, mert a kernel minden csoporton külön fog dolgozni, és így optimalizáltabb párhuzamos feldolgozás végezhető.

A program ezután előkészíti az OpenCL környezetet: lekéri az elérhető platformot és eszközt (GPU-t), majd létrehoz egy OpenCL kontextust. Ha bármelyik lépés során hiba történik, az errorArray tömb segítségével részletes hibainformációt nyújt.

Miután betölti és lefordítja az OpenCL kernel kódját az averages.cl fájlból, beállítja a szükséges fordítási opciókat (például a NUM\_SAMPLES\_PER\_GROUP makrót, amelyet a kernel fordításakor figyelembe vesz). A program figyeli, hogy a kernel fordítása sikeres volt-e, és ha nem, kiírja a fordítási naplót a hibakereséshez.

A sikeres fordítás után létrehozza az avg nevű OpenCL kernelt. Ez a kernel hajtja majd végre az átlagolási műveletet minden egyes mintacsoporton.

A fő részben a program lefoglalja a szükséges puffereket, melyek a bal és jobb hangsáv mintáit, valamint az eredményeket tartalmazzák. Ezeket a puffer objektumokat a CL\_MEM\_ALLOC\_HOST\_PTR attribútummal hozza létre, így később közvetlenül is elérhetők a CPU számára. A bufferek ezután leképezésre kerülnek (clEnqueueMapBuffer), így a program bele tudja másolni a WAV fájlból kiolvasott bal és jobb csatorna értékeket.

Miután feltöltötte az adatokat a bufferekbe, azok vissza lesznek unmapelve, majd a kernel argumentumai beállításra kerülnek. A clEnqueueNDRangeKernel függvénnyel elindítja a párhuzamos feldolgozást: minden csoportot egy munkacsoportként kezel, és a csoporton belüli minták száma határozza meg a lokális munkacsoport méretét. A globális munkaméretet úgy módosítja, hogy illeszkedjen a lokális mérethez.

A kernel futása előtt és után időmérést is végez, kétféleképp: egyrészt a clock() segítségével (ez a host oldali mérés), másrészt az OpenCL által biztosított clGetEventProfilingInfo segítségével, amely sokkal pontosabb GPU-időt ad vissza nanomásodperc pontossággal.

Miután a kernel befejezte a számítást, a program ismét leképezi az eredmény-puffert, és beolvassa belőle az egyes csoportokra számított átlagokat. Ezeket ki is írja a konzolra (jelenleg kommentelve van a kiíratás egy része, de könnyen aktiválható).

A program végül kiírja az összesített teljesítményadatokat: hány mintacsoportot dolgozott fel, mennyi időt vett igénybe a kernel futása, és mennyi volt a teljes programfutás ideje (beleértve az adatmásolásokat és inicializációt is).

Ha bárhol hiba történik, a goto cleanup szerkezet segítségével a program felszabadítja az összes erőforrást (memóriát, OpenCL objektumokat, stb.), hogy ne hagyjon maga után memória- vagy erőforásszivárgást.

# Hibakeresés

Az OpenCL hibakódok könnyebb értelmezését segíti az *errors.c* fájl. Az assets mappában az *errorcodes.csv* fájlban vannak eltárolva a hibakódok értékei, nevei, és leírásai.

A *readErrorsFromFile* függvény egy struct errorcode típusú tömböt vár paraméterként, amelyet a függvény feltölt a CSV fájlban található adatok alapján. Az errorcode struktúra definíciója az include mappában az *errorcodes.h* fájlban található. A fájl megnyitása a fopen függvénnyel történik, olvasási módban ("r"). Ha a fájl nem nyitható meg, a perror függvénnyel kiírja a hiba okát a standard hibakimenetre, majd visszatér a függvényből, további feldolgozás nélkül.

A fájl minden sorát külön-külön olvassa be egy 512 karakteres pufferbe. Minden sorban egy hiba kódját és hozzá tartozó három további mezőt vár: a hiba nevét (error\_flag), a kapcsolódó függvény nevét (function), valamint egy szöveges leírást (description). Ezeket pontosvesszővel (;) elválasztva kell szerepeltetni a fájlban. A sor feldolgozása a sscanf függvénnyel történik, amely pontosan négy mezőt próbál beolvasni a megadott formátum szerint.

Amennyiben a sscanf sikeresen olvassa be mind a négy mezőt, a program átalakítja a hiba kódját egész számmá (int) az atoi függvénnyel. Ezután feltölti a megfelelő mezőket az errorArray adott indexénél. Az strncpy függvényt használja a karaktertömbök másolására, hogy biztosítsa a buffer túlcsordulás elkerülését. Fontos, hogy a másolásnál mindig egy karakterrel kevesebbet ad meg a célméretnél, így helyet hagy a lezáró nullának (\0).

Minden sikeres sor beolvasása után növeli az errorCount számlálót, hogy a következő hiba a tömb következő elemébe kerüljön. Ez így folytatódik a fájl végéig.

Miután az összes sort feldolgozta, a függvény bezárja a fájlt a fclose függvénnyel, és ezzel befejezi működését.

A *printErrorDetails* függvény három paramétert vár: egy egész számot (errorCode), amely a keresett hiba kódja; egy errorArray nevű tömböt, amely errorcode struktúrákat tartalmaz; valamint egy errorCount nevű egész számot, amely azt jelzi, hány érvényes elem található a tömbben.

A függvény egy for ciklust használ arra, hogy végigmenjen az errorArray tömb elemein. Minden iterációban megvizsgálja, hogy az aktuális hibaelem code mezője megegyezik-e a megadott errorCode értékkel. Ha talál egyezést, akkor kiírja a hibához tartozó összes részletet, azaz a hiba kódját, a hiba megnevezését (error\_flag), a kapcsolódó függvény nevét, valamint a hiba szöveges leírását. Mindezt strukturált és olvasható formában teszi, soronként. Az információk megjelenítése után kiír egy elválasztó vonalat, majd return utasítással kilép a függvényből, hiszen a keresés sikeres volt.

Ha a ciklus lefutása után sem talál egyező hibaazonosítót, akkor a függvény kiírja, hogy a megadott kód nem található a hibák listájában. Ez egy hasznos visszajelzés lehet például hibakeresés során vagy felhasználói visszajelzésként.

# runner.c

A runner program egy CSV fájlba írja az adatokat, miközben egy külső programot (itt main.exe) futtat, különböző paraméterekkel, mint például a csoportonkénti minták száma. A kimeneti adatokat elemzi.

A program először megpróbálja megnyitni a results.csv fájlt olvasásra. Ha a fájl létezik, akkor ellenőrzi, hogy tartalmazza-e már a szükséges fejlécet. Ehhez az első sort (fejlécet) olvassa be, és megnézi, hogy szerepel-e benne a "NumSamplesPerGroup" kifejezés. Ha igen, akkor a fájl már tartalmazza a fejlécet, és a program nem írja újra.

Ha a fájl sikeresen megnyílt, akkor a program átállítja a fájlt hozzáfűzés (append) módba, hogy új adatokat fűzhessen hozzá anélkül, hogy felülírná a meglévő tartalmat. Ha a fájlt nem sikerül megnyitni, akkor a program hibaüzenetet ír ki, és leáll.

Ha a fejléc nem létezett a fájlban, akkor a program hozzáadja a szükséges fejlécsort, amely tartalmazza az oszlopneveket: NumSamplesPerGroup, NumGroups, AvgKernelExecutionTime, AvgKernelProcessingTime, AvgTotalProcessingTime.

A program ezt követően egy ciklusban végigmegy a csoportonkénti minták számán (4, 8, 16, ..., 1024). Minden egyes minta számára (mely a csoportonkénti minták számát jelenti) a program 10 iterációt futtat, és az eredményeket összegzi.

Minden iterációban a program egy parancsot alkot, amely a main.exe programot futtatja a jelenlegi csoportonkénti minták számával, és a kimenetet egy pipe segítségével olvassa be. A kimeneti adatokat, amelyek JSON-szerű formátumban vannak, a program parszolja, és az értékeket tárolja a megfelelő változókban.

Minden iteráció után a program összegzi a különböző időtartamokat (kernel végrehajtási idő, feldolgozási idő, teljes feldolgozási idő), majd ezek alapján kiszámítja az átlagos értékeket. A kiszámolt átlagokat pedig hozzáfűzi a CSV fájlhoz a megfelelő sorban.

A ciklus végén a fájlt bezárja, és ha a program bármelyik iteráció során hibát észlel (például a main.exe nem fut le sikeresen), akkor hibaüzenetet ad ki, és leáll. A sikeres futtatás esetén a program az összes eredményt a fájlba menti.

Ez a program tehát a külső alkalmazás futtatása után összegyűjti és átlagolja a különböző futtatási eredményeket, majd ezeket tárolja egy CSV fájlban, hogy később elemezni lehessen.

# Mérési adatok

A mérési adatok a *meresek.xlsx* fájlban találhatóak.

Az *Averages for 100 iterations* fül alatt a program százszori egymás utáni futtatásának átlagolt eredményei láthatók. Az első táblázat annak a verziónak az eredményei alapján készült, amelyben nem használ a program vektorokat. A **NumSamplesPerGroup** értéke azt mutatja, hogy egy feldolgozási csoport hány mintát tartalmaz, míg a **NumGroup**` azt jelzi, hány ilyen csoportból áll össze a teljes feldolgozás.

Az **AvgKernelExecutionTime**, vagyis az átlagos kernel futási idő a legkisebb minta/csoport értéknél (4) a legmagasabb, ekkor 2,825 ms. Ez az érték jelentősen csökken 8 minta/csoportnál (1,085 ms), majd 16–128 mintás beállításnál stabilizálódik 0,75–0,87 ms között. A 256 mintás csoportméret esetén ismét emelkedés figyelhető meg (1,163 ms), ami arra utal, hogy túl nagy csoportméret esetén a párhuzamos feldolgozás hatékonysága csökkenhet.

A **AvgKernelProcessingTime**, vagyis a host oldal várakozási ideje nagyon alacsony, minden esetben 0,01–0,02 ms. Ez azt mutatja, hogy a CPU (host oldal) gyakorlatilag nem tölt időt azzal, hogy a GPU feldolgozásának befejezésére várjon. Ez kedvező, mivel azt jelzi, hogy a feldolgozás hatékony, és a host–device kommunikáció nem jelent szűk keresztmetszetet. Emellett az alacsony várakozási idő arra utal, hogy a program jól kihasználja a GPU párhuzamos feldolgozási képességeit, és a CPU–GPU együttműködés hatékonyan működik.

Az **AvgTotalProcessingTime**, vagyis az átlagos teljes feldolgozási idő 25,48 és 28,66 ms között mozog. A legkisebb érték 32 minta/csoport esetén figyelhető meg, ami arra utal, hogy ez a konfiguráció adja a legrövidebb teljes futási időt. Ugyanakkor 8, 16 és 64 mintás beállítások is hasonlóan alacsony értékeket produkálnak (26–27 ms körül), vagyis ezek is hatékonynak tekinthetők. A legmagasabb teljes feldolgozási a legkisebb csoportméretnél jelentkezik (28,66 ms 4 mintánál), ami valószínűleg a sok csoport miatti többletterhelésből adódik.

A 100 futtatás átlaga alapján a legjobb teljesítmény 32 minta/csoport körül érhető el, de 8, 16 és 64 mintás csoportméret esetén is hasonlóan jó eredmények születnek. A túl kicsi csoportok (pl. 4) megnövelik a kernel futási és teljes feldolgozási időt a sok csoport okozta többletterhelés miatt. Túl nagy csoportméretnél (pl. 256) szintén nő a kernel futási idő, ami a párhuzamos feldolgozás hatékonyságának csökkenésére vagy a hardveres korlátokra utalhat. A host oldal várakozási ideje minden esetben elhanyagolhatóan kicsi, így nem jelent teljesítménybeli problémát.

## Vektorizálás hatása

A második (kék) táblázat annak a vektorokat használó verzió eredményeit mutatja. A két táblázat összehasonlítása világosan mutatja, hogy a vektorosított megközelítés jelentősen csökkentette a kernel futási idejét, különösen a kisebb csoportméreteknél.

A nem vektoros verzióban például 4 minta/csoport esetén a kernel átlagos futási ideje 2,825 ms volt, míg a vektorosított változatnál ugyanez az érték csupán 1,071 ms. Hasonló különbség figyelhető meg 8 minta/csoport beállításnál is: a nem vektoros verzióban 1,085 ms, míg a vektorosított változatban mindössze 0,242 ms a kernel futási ideje. A 16 mintás esetben még látványosabb a gyorsulás, ahol az eredeti 0,873 ms helyett csak 0,075 ms-ra csökkent a végrehajtási idő.

Ez a gyorsulás elsősorban annak köszönhető, hogy a vektoros adattípusok – például short4, short8, stb. – használatával a kernel kód képes egy utasítással több adatot feldolgozni, így csökken az utasítások száma, a memóriahozzáférések mennyisége, és általában véve hatékonyabbá válik az erőforrások kihasználása. A GPU architektúrák – különösen azok, amelyek támogatják SIMD (Single Instruction, Multiple Data) feldolgozást – jól skálázhatók ilyen típusú optimalizálás esetén.

Fontos megfigyelés az is, hogy míg a kernel futási ideje jelentősen csökkent, addig a teljes feldolgozási idő (AvgTotalProcessingTime) csak mérsékelten. Ez arra utal, hogy a teljes végrehajtási időt más tényezők – például adatmozgatás, szinkronizáció vagy a CPU-GPU közötti kommunikáció – is korlátozzák. Ennek ellenére a vektorosított verzió egyértelműen hatékonyabb, különösen akkor, ha a feldolgozás domináns része a GPU oldalon történik.

## Buffer mapping hatása

A harmadik (narancssárga) táblázat buffer mapping[[1]](#footnote-1) nélküli eredmények alapján készült. Az figyelhető meg, hogy a buffer mapping használata jelentős teljesítményjavulást eredményez mind a kernel futási idő, mind a teljes feldolgozási idő szempontjából.

A legnagyobb különbség az alacsony mintaszámnál figyelhető meg: 4 mintás csoport esetén a kernel futási idő buffer mappinggel 2,825 ms, míg nélküle 6,879 ms – vagyis több mint kétszeresére nő, ha nincs mapping. Ez azzal magyarázható, hogy kis méretű, de sokszor ismétlődő adatmásolás esetén a hagyományos clEnqueueWriteBuffer / clEnqueueReadBuffer műveletek nagy overheadet jelentenek.

A teljes feldolgozási idő is látványosan javul. A buffer mappinges változatban 4 mintánál 28,66 ms, míg nélküle 36,01 ms – vagyis közel 7,5 ms különbség. Ez az arány a nagyobb mintaszámoknál csökken, de még 128-as és 256-os csoportméretnél is több mint 2 ms megtakarítás figyelhető meg a teljes futásidőben.

1. A buffer mapping egy olyan művelet, amely lehetővé teszi, hogy a host közvetlenül hozzáférjen a GPU memóriájához másolás nélkül, így gyorsabb adatkezelést biztosít. [↑](#footnote-ref-1)