

TDK-dolgozat

Tóth Bálint

NAGYFOKÚAN PÁRHUZAMOS MULTI-GPU ALGORITMUSOK KUTATÁSA PRE-EXASCALE ARCHITEKTÚRÁKRA

Szerző: Tóth Bálint

Témavezető: Dr. Juhász Zoltán, egyetemi docens

Pannon Egyetem

Műszaki Informatikai Kar, Villamosmérnöki és Információs
Rendszerek Tanszék

Lezárás: 2024. május

Nagyfokúan párhuzamos multi-GPU algoritmusok kutatása pre-exascale architektúrákra

Készítette: Tóth Bálint

Programtervező Informatikus MSc.

*Műszaki Informatikai Kar, Villamosmérnöki és Információs Rendszerek
Tanszék*

Témavezető: Dr. Juhász Zoltán

A modern kutatás elképzelhetetlen szuperszámítógép támogatás nélkül, ami segít szimulációk vagy adatelemzések hatékony végrehajtásában. E dolgozat az agyi elektromos jelek feldolgozásának szuperszámítógépes támogatását vizsgálja meg. Az EEG (Elektroencefalográfia) nagy időbeli felbontással képes követni az emberi agyban lejátszódó folyamatokat. A magas mintavételi frekvencia, a minél több mérőelektroda használata, valamint a komplex jel- és adatfeldolgozó algoritmusok eredménye a megnövekedett, gyakran többnapos nagyságrendben lévő számítási idő. A számítógép rendszerek Moore törvény által jellemzett fejlődése napjainkra megállt, további gyorsulás csak párhuzamos architektúrák és algoritmusok alkalmazásával lehetséges. A mai szuperszámítógépek többsége grafikus kártyákat (GPU) alkalmaz a nagy számítási teljesítmény elérése érdekében. A GPU technológiával a párhuzamosságnak egy olyan költséghatékony szintje érhető el, ami drasztikusan csökkenti az algoritmusok futási idejét, viszont a hagyományostól jelentősen eltérő programozási modellt és algoritmustervezési megközelítést igényel. Különösen fontos a párhuzamos algoritmusok tervezése a több ezer GPU kártyát használó ún. pre-exascale (több 100 petaFlop/s) rendszerek esetén, ahol az adatmozgatás, kommunikáció és szinkronizáció ideje jelentőssé válik, emiatt a nagyságrendekkel kisebb párhuzamos rendszerekre fejlesztett és optimalizált algoritmusok nem alkalmazhatók ugyanolyan hatékonysággal. A dolgozat a vonatkozó szakirodalom ismertetése után bemutatja az EEG mérési adatok előfeldolgozáshoz kiválasztott jelfeldolgozó algoritmusokat (konvolúció, Short-Time FFT, Wavelet-transzformáció), majd ezek párhuzamos CUDA implementáció terveit egy és több GPU kártyára. A kiválasztott algoritmusok CUDA-aware MPI segítségével végzik a node-ok és GPU-k közötti kommunikációt. A dolgozat a futási eredmények bemutatásával és a teljesítményelemzésből levont tanulságok ismertetésével zárul.

Tartalom

1.	Bevezetés	1
2.	Irodalmi áttekintés	3
2.1.	GPU algoritmusok és fő tulajdonságaik	6
2.2.	EEG technológia és jelfeldolgozási problémák	11
2.3.	A Komondor szuperszámítógép	13
3.	Algoritmustervezés	15
3.1.	Konvolúció	15
3.2.	Short-Time-Fourier Transzformáció (STFT)	19
3.3.	Wavelet transzformáció	22
3.4.	Implementációs sajátosságok	26
4.	Eredmények	30
4.1.	Konvolúció	30
4.2.	Short-Time Fourier transzformáció	32
4.3.	Wavelet transzformáció	34
5.	Összegzés	36
	Felhasznált irodalom	37

1. Bevezetés

Az agyi képalkotást forradalmasító EEG nagy felbontással gyűjt nagy mennyiségű adatot az emberi agyban lezajló elektromos folyamatokról, ami a több kHz mintavételezési frekvenciának és a nagy számú mérőelektroda használatának következménye. Jelek előfeldolgozásától gépi tanulásig számos adatintenzív folyamatból áll egy tudományos EEG kísérlet feldolgozása, amik a jelenleg elérhető szoftverekkel hardveres gyorsítás nélkül akár több napig is futhatnak. Ennek következtében elengedhetetlen a szuperszámítás igénybevétele.

A szuperszámítógépek által elérhető számítási teljesítmény kizárólag hardveres gyorsítással (pl. GPU, FPGA) biztosítható. Ilyen nagyfokúan párhuzamos rendszerekben a hagyományos programozási paradigmák használhatatlanok, helyettük párhuzamos programozási modellek alkalmazása szükséges. Az általánosan elterjedt párhuzamosítási módszerek alapja az probléma részekre osztása és az így keletkezett részfeladatok megoldása. A párhuzamosság növelésével azonban a SPMD (Single Process Multiple Data) alapon működő párhuzamos modellek nem alkalmazhatók. Extrém párhuzamosság figyelhető meg grafikus kártyákban (GPU) is, amelyekre a CUDA párhuzamos programozási modell alkalmazható olyan programok készítésére, amelyek nem csak nagyságrendekkel kevesebb idő alatt futnak le, mint szekvenciális változatuk egy processzoron, de a probléma méretének növelésével a GPU kihasználtsága is növekszik.

Algoritmusok párhuzamosítása során azok skálázhatósága kiemelten fontos és minden esetben közvetlenül függ az algoritmus tulajdonságaitól. Skálázhatóságnak két típusát különböztetjük meg: strong scaling és weak scaling. A strong scaling azt jellemzi, hogy egy algoritmus futási ideje milyen ütemben csökken, amikor fix problémamérethez több számítási egységet adunk. A weak scaling azt a teljesítménynövekedést írja le, amikor a számítási egységek hozzáadása mellett azok lokális feladatmérete változatlan marad.

A két skálázhatóság különbségeinek demonstrálása érdekében végezzünk el egy rövid gondolkísérletet. Tételezzük fel, hogy egy kutatás során csoportosan vizsgálunk 20 alanyt EEG méréssel, minden esetben 128 csatornával 2 kHz-en mintavételezve 60 percig. Ebben az esetben egy egységnyi mérési regisztrátum egy alanytól 128 darab lebegőpontos szám (a csatornák értékei egy mérési ponton). Egy másodperc alatt 2000 mérési adatpont keletkezik, így 32 bites számábrázolással számolva egy alany méréséből keletkezett adat 60 perc felvétel után hozzávetőlegesen 3,6 GB. Tételezzük fel továbbá,

hogy a teljes előfeldolgozó folyamatra léteik egy szimpla-GPU implementáció, ami úgy lett megtervezve, hogy egy alany adatain dolgozzon. Ebben az esetben, ha rendelkezésre áll 20 GPU, a probléma egyszerűen skálázható: egy GPU-n fut egy alany feldolgozása. Azonban több ezer GPU-t magába foglaló pre-exascale architektúrán ez a megoldás korántsem ideális, mivel a rendelkezésre álló kapacitás töredékét használja csak ki a 20 GPU. Az első eset weak scaling megoldást ír le, mivel minden egyes alany esetén hozzáadunk egy GPU-t a rendszerhez, így a lokális problémaméret változatlan. A második esetben az összes GPU kihasználása kizárólag strong scaling algoritmussal valósítható meg, mivel az egy elemi mérésre jutó adatot szükséges tovább particionálni, hogy használható legyen több számítási egység.

A jelen kutatás célja három kiválasztott EEG előfeldolgozó algoritmus (konvolúció, short-time Fourier transzformáció és wavelet transzformáció) olyan multi-GPU változatának megtervezése, amelyekkel strong scaling érhető el szuperszámítógép architektúrákon. Mindennek elérése érdekében elengedhetetlen a GPU architektúra által biztosított lehetőségek hatékony kihasználása, valamint az eszközök közti kommunikáció minimalizálása. Pre-exascale számítógép jelenleg Magyarországon nem áll rendelkezésre, így tesztelés céljából a Komondor szuperszámítógép GPU partícióját vettem igénybe.

2. Irodalmi áttekintés

A szuperszámítógépek napjaink elengedhetetlen tudományos eszközei, amik nélkül szimuláció alapú kutatások nem lennének lehetségesek. Számos tudományterület, például fizika, biológia és csillagászat új eredményei köszönhetők nagy teljesítményű számítógépeknek [1]. A High Performance Computing (HPC) több módon is támogathatja a korszerű kutatásokat. Egyrészt, szimulációkkal nagy pontossággal tervezhetők előre kísérletek, amik eredményei a későbbiekben valós fizikai apparátuson, vagy alanyokon igazolhatók. Másrészt mérés és kísérletek során nagy mennyiségű adat keletkezik, amit a nagy teljesítményű számítás nélkül beláthatatlanul hosszú idő feldolgozni PC eszközökkel. A jelen dolgozatban ismertetett jelfeldolgozó algoritmusok multi-GPU megvalósítását is ez motiválta.

A számítástechnika, így a nagy teljesítményű számítás fejlődésének egy jellemzője Moore törvénye, amely évtizedekig hajtotta a mikroprocesszorok fejlődését. Moore törvénye kimondja, hogy a processzorokat alkotó tranzisztorok fizikai mérete közel azonos mértékben csökken bizonyos idő elteltével. Ennek a szabálynak egy következménye a számítási teljesítmény, illetve órajel frekvencia kétféleképpen kétszerezése, ami azonban a Dennard-skálázásból következik [2]. A Dennard-skálázás szerint a mikroprocesszorokat felépítő MOSFET-ek méretének csökkenésével azok energiasűrűsége azonos marad, így az egy tranzisztorra vonatkozó működési feszültség és áramerősség csökken [1]. A két skálázási törvény úgy kapcsolódott össze, hogy a tranzisztorok stabil csökkenésével a mikroprocesszorokba azonos területen ugyanakkora energiasűrűsége több tranzisztor került és így azok feszültségének csökkentése mellett az órajelük frekvenciája emelhető volt. Ez a folyamat hozzávetőlegesen 2004-ig tartható volt, azóta fokozatosan lelassult. Ennek oka, hogy konstans feszültség mellett az áramkörök órajelfrekvenciájának növelésével az áramkör disszipációja olyan magasra nőne, hogy lehetetlen lenne lehűteni [2].

Miután processzorok egyéni teljesítménye az imént leírtak miatt jelentős mértékben tovább nem növelhető, a nagy számítási igényű feladatok elvégzésének egyetlen lehetséges módja a párhuzamos technológiák alkalmazása. Ily módon a számítási teljesítmény továbbra is fokozatosan növekszik, azonban a jelenség fontos velejárója, hogy a hardver és szoftvertervezési optimalizációs célok a következőképpen változnak [2] szerint. Órajel növelése helyett a hardver skálázásából és parallelizmusból származik a teljesítmény növekedése. Egyre fontosabb tényező a nagy teljesítményű rendszerek energiafelvétele is, mivel a párhuzamosság miatt egy rendszerben eredendően

ilyen eszközből felépített nagy hálózatok kapacitása, miután az ilyen magas szintre skálázott processzor belső komplexitásának nagy része használatlan marad.

Ahogy a világ szuperszámítási teljesítménye fokozatosan közelít az exascale teljesítménytartomány mindennapossá válásához, szükséges új megközelítésű algoritmusok kifejlesztése, amelyek képesek a nagyfokú párhuzamosság kihasználására. Kiemelten fontossá vált a feldolgozandó adatok lokalitása, mivel a modern rendszerekben a memória sávszélessége képezi a szűk keresztmetszetet [1]. A párhuzamos programozásnak egy teljesen más formája szükséges több millió mikroprocesszor hatékony használatára. Az általános, adatok partícionálásán és egységes program lefuttatásán alapuló SPMD programozási modell ilyen mértékben nem alkalmazható. Ennek oka, hogy ebben az esetben a szükséges memória hálózati kommunikációval kerül szétosztásra, ami sok kis csomag esetén teljesen ellehetetleníti a hatékony működést. Mindemellett a jelenleg HPC környezetben használt algoritmusok, különösen a GPU-ra írtak nagyságrendekkel alacsonyabb párhuzamossági szintre lettek optimalizálva, így azok nem feltétlenül skálázódnak hatékonyan.

Az Európai Unió szuperszámítógép stratégiájának fontos eleme az EuroHPC Joint Undertaking. Az alapítvány célja a kontinens szuperszámítási kapacitásának kiépítése és a tagállamok szuperszámítási kapacitásának fejlesztéseinek levezénylése. A projekt keretében települt számos petascale, három pre-exascale (LUMI, Leonardo, MARENOSTRUM 5), valamint telepítés alatt áll Európa első exascale szuperszámítógépe (JUPITER) Németországban [4]. A program emellett felzárkózásként szolgál az Egyesült Államok, valamint Kína és Japán szuperszámítási kapacitásához. A korábbi teljesítmény alapú besorolások, petascale, pre-exascale és exascale a szuperszámítógépek maximális teljesítményére utal Flop/s mértékegységben. Pre-exascale architektúra alatt a száz petaFlop/s nagyságrendet értjük. Jelenleg a világon egy exascale teljesítményű szuperszámítógép üzemel, az Egyesült Államokbeli Frontier [5]. Mindemellett 2023-ban a Debreceni Egyetem kampuszán üzembe helyezték a Komondor névre hallgató petascale szuperszámítógépet, ami jelenleg a legnagyobb teljesítményű kutatás-fejlesztési célú számítógép Magyarországon, működéséért a Központi Informatikai Fejlesztési Ügynökség felel.

2.1. GPU algoritmusok és fő tulajdonságaik

A grafikus kártyák eredeti funkciója háromdimenziós grafikus objektumok képernyőre vetítése. Ez a feladat az objektumok különböző pontjaira egymástól függetlenül végezhető, így a GPU-k architektúrája alkalmas nagyfokú párhuzamosságra. Az architektúra kihasználása és a párhuzamos programozás ezen stílusa ismert és gyakran alkalmazott módja nagy méretű számítási feladatok gyorsításának [6]. A programozható grafikus kártyák ily módon hatékonyan használhatók olyan feladatokra, amelyekre jellemző a magas szintű parallelizmus, amelyet az NVIDIA által gyártott eszközökben a CUDA programozási modell és a hozzá tartozó nyelv tesz lehetővé.

A GPU programozás nagy mértékben eltérő a hagyományos processzorokra, vagy mikroszámítógépekre írt szekvenciális kódok tervezésétől. Az imperatív paradigmát követő programnyelvek szemantikája valamilyen jellegű iterációval kezel sorrendi adatokat, például tömböket. Ezek a rendszerek párhuzamos végrehajtási egységek, processzormagok hozzáadásával viszonylag egyszerűen a probléma szintjén párhuzamosíthatók. Ebben az esetben az adatsor felosztásra kerül, és az így keletkezett rövidebb adatsoron hajtják végre ugyanazt a szekvenciális kódot a processzormagok.

Egy GPU esetén ezres nagyságrendben állnak rendelkezésre programozható végrehajtási egységek, így a párhuzamosság egy sokkal magasabb szintjét teszik lehetővé. Felmerülhet a kérdés, hogy mekkora mértékben célszerű a korábban említett módon partícionálni az adatsort oly módon, hogy a rendelkezésre álló hardver teljes számítási teljesítménye ki legyen használva. Előbb, vagy utóbb a rendelkezésre álló szálak száma egy ilyen környezetben eléri, sőt meg is haladhatja a feladat adatpontjainak számát, következésképpen minden adatelemhez egy szál fog tartozni. Ez a később részletesen is bemutatásra kerülő CUDA programozási modell egyik alapja.

Az előbbi gondolat egyik fontos következménye, hogy a legtöbb algoritmus esetén a szekvenciális megvalósításoktól radikálisan eltérő. Emiatt előfordulhat, hogy teljesen újszerű implementációk szükségesek extrém párhuzamos környezetekben, így GPU-k esetében is [1]. Természetesen nem minden algoritmus egyformán párhuzamosítható, így nem is mind alkalmas grafikus kártyán történő megvalósításra [7]. Bizonyos esetekben a GPU-n olyan algoritmusok működnek, amelyek hagyományos szekvenciális esetben rosszabbul teljesítenek. Mindez köszönhető az algoritmuson belüli lokalitásnak, valamint Amdahl törvényének [8].

A CUDA hardver modell alapja a párhuzamos egységek hierarchiája. Számunkra három logikai szint releváns: a CUDA mag, a Streaming Multiprocessor (SM) és maga a

GPU [9]. Legalacsonyabb szinten szerepel a CUDA mag, ami egy elemi műveletvégző egység és hozzávetőlegesen analóg egy CPU terminológiából ismert Aritmetikai és Logikai egységgel. A CUDA magok feletti hierarchia egység az SM, amely a kártya kivitelétől függően több magot foglal magába. Emellett az SM-en belül helyezkednek el a magok regiszter fájljai, illetve az L1 szintű gyorsítótár. A GPU kártya több SM összekapcsolása a szükséges ütemezőhardverrel, valamint az L2 szintű gyorsítótárral. Fontos megjegyezni, hogy a grafikus kártya fő memóriája a DRAM a chipen kívül helyezkedik el, így a gyorsítótárak hatékony kezelése a későbbiekben kiemelten fontos szerepet fog játszani.



2. ábra Streaming Multiprocessor logikai felépítése

A CUDA programozási modell a C/C++ programnyelvek egy minimális kiegészítése, amelyet úgy terveztek meg, hogy absztrakciókon keresztül tükrözze a mögöttes GPU architektúrát, mindezt jól ismert nyelvek szekvenciális szemantikájával [6]. A programozási modellben a korábban említett hardver hierarchia mind a szálak kezelésében, mind a programozható memória elérésében és használatában megjelenik. A modell alapja a kernel, egy kulcsszóval megjelölt függvény, amely egy szál működését írja le. Ez úgy is értelmezhető, hogy a programozási feladatban valamely tömb egy elemére pontosan milyen utasítások fussanak le egy magon. A hierarchia további részei szintaxis szempontjából a kernel függvény meghívásakor jelennek meg.

A kernel, mint szál további két logikai szintbe sorolható: blokk és grid. A blokk analóg egy SM egységgel a hardver implementációban. A blokkon belül a szálak 32-esével futnak le, ezt warp-nak nevezzük. Warp-on belül Single Instruction Multiple Thread (SIMT) alapú végrehajtás történik, mind a 32 szálon ugyanaz a gépi kódú utasítás

fut le egy időben[10]. A blokkokon belül továbbá lehetséges az explicit szinkronizálás. Végül a CUDA program legfelső egysége a grid, amely magába foglalja az összes elindított blokkot. A grid és a blokkok mérete egy kernel meghívásakor adható meg speciális szintaxissal: a kernel neve és argumentumai között tripla „<>” operátor között.

A szálak hierarchiájához hasonlóan a kernelben használható memória is illeszkedik a hardveres megvalósításra [11]. Minden CUDA mag rendelkezik saját regiszterekkel, amelyek az SM-en belül helyezkednek el. Minden kernelen belül deklarált primitív típusú változó automatikusan regiszterbe kerül. A regiszter a leggyorsabban írható-olvasható memória, elérése egy órajelciklus. Szintén az SM-ben található a blokk szintjével analóg L1 gyorsítótár, amelynek egy része, a Shared Memory, programozható. A Shared Memory-t a blokkon belül bármelyik szál elérheti alacsony késleltetéssel. A grid logikai szintjén található az L2 gyorsítótár, melynek az off-chip DRAM memória betöltésének gyorsítása, miután egy DRAM elérés több száz órajelciklusba is kerülhet.

Az imént ismertetett memória- és szálhierarchia adja a GPU hatékonyságát, azonban egyben ugyanezek a tényezők korlátozzák az elérhető maximális teljesítményt. Amennyiben egy program viszonylag kevés műveletet végez egy szálon, a teljesítményt korlátozó tényező a memória sávszélessége. Ebben az esetben a programot memória korlátoosnak mondjuk, mivel a műveleti egységeknek várniuk kell a memória beérkezésére. A másik véglet, amikor nagy mennyiségű számítás hárul a kernelekre és a rendelkezésre álló memóriasávszélesség nincsen teljes mértékben kihasználva. Ezt az esetet számítás korlátoosnak mondjuk. Hatékony GPU algoritmus tervezésekor három fő tényező figyelembevétele szükséges [12]: a legalacsonyabb szinten futó warp-ok hatékony megszervezése, az SM-ek telítése, valamint a számításhoz használt adatok lokalitásának fontossága.

Grafikus kártyákra írt algoritmusok teljesítménymérésének egyik jól ismert módja a Roofline Analysis. A módszer alapja a hardverre jellemző maximális számítási teljesítmény (FLOP/s), valamint memóriasávszélesség (GB/s), valamint az algoritmus aritmetikai intenzitása. Aritmetikai intenzitás alatt a lebegőpontos műveletek arányát értjük a végrehajtott memóriamozgatáshoz képest [13]. Egy algoritmus teljesítményét az aritmetikai intenzitásának függvényében grafikonon ábrázolva megkapható a Roofline Model grafikus reprezentációja is, amely egy növekvő lineáris és egy konstans függvényből áll össze. A kezdeti lineáris rész oka, hogy a számítási erőforrások növelésével a teljesítmény ideális esetben arányosan növekszik [12], ebben az esetben az

algoritmus számítási korlátos. A konstans szakasz a memóriasávszélesség következménye: hiába növekszik a párhuzamosság, a memóriabusz nem képes hatékonyan kiszolgálni a megnövekedett adatigényt. A görbe az ideális mennyiségeket adja meg, a jellemezni kívánt algoritmus a grafikonon egy pont formájában jelenik meg.

Nagy teljesítményt igénylő számítási feladatok túlnyomó többségében a GPU igénybevétele viszonylag kevés, jól ismert műveletre korlátozódik. Ezekre a feladatkora, például lineáris algebrai műveletek és egyenletmegoldók, elérhetők a GPU programozás technikai részleteit függvények mögé rejtő könyvtárak. A későbbiekben a cuFFT könyvtár kerül alkalmazásra frekvenciatartománybeli jelfeldolgozás céljából. Ezen könyvtárak elsődleges célja már létező kódbázisok teljesítményének javítása. Fontos megjegyezni azonban, hogy a könyvtárhasználat nem old meg minden problémát. Egyrészt a fejlesztő így elveszíti a kontrollt az algoritmus belső működése felett, másrészt figyelembe kell vennie a könyvtár memória-használati modelljét [14].

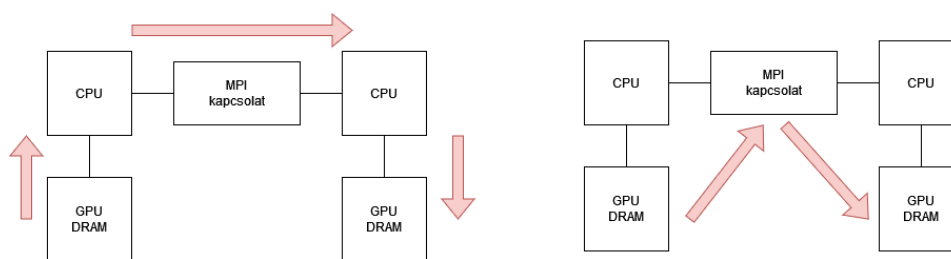
A jelen dolgozat elkészültekor a szuperszámítástechnikában széles körben elterjedt gyorsítókártya az NVIDIA Ampere architektúrájú A100 eszköze. A korábban ismertetett Komondor szuperszámítógép GPU partíciója is ezzel a hardverrel van felszerelve, így a későbbiekben bemutatásra kerülő tervezés és tesztelés szempontjából érdemes említést tenni az architektúra főbb tulajdonságairól. Az Ampere architektúrában jelent meg a harmadik generációs tenzor mag [15], az új TF32 típusú lebegőpontos számokkal, amelyekkel tovább gyorsítható hardveresen mátrixok szorzása. A második jellemző technológia, ami az Ampere architektúrával vált elérhetővé, klaszterben történő alkalmazást segítő Multi-Instance GPU, amely lehetővé teszi, hogy több konkurens feladat fusson egyszerre egy kártyán [16]. Ez nagyban hasonlít a GPU-n konkurens grid-ek futtatására, azonban ebben az esetben egymástól teljesen független folyamatok futhatnak egyszerre kölcsönös hátráltatás nélkül.

A tenzor mag az NVIDIA által kifejlesztett célhardver, amely kis méretű mátrixok szorzására és összeadására használható. Eredetileg a mesterséges intelligenciában gyakran felmerülő mátrix műveletek és a HPC alkalmazások motiválták a tenzor mag technológia megjelenését, mindemellett egyéb területeken is alkalmazható. Először a Volta architektúrában jelent meg 16 és 32 bites lebegőpontos számok támogatásával [15]. Későbbi architektúrákban további típusok kerültek bevezetésre a számítási sebesség gyors növekedésével egyetemben a korábban. Programozástechnikai szempontból a CUDA API által biztosított warp-on belüli $A=AB+C$ mátrix-FMA műveletek formájában

jelenik meg a tenzor magok használata [17]. Blokkmátrix szorzás algoritmussal a hardveres gyorsítás nagy méretű mátrixok számítására is használható.

A jelenlegi csúcstechnológiát képező NVIDIA GPU architektúra, a Hopper, újabb Mesterséges Intelligencia és szuperszámítási feladatokra optimalizált funkcionalitással lett ellátva. A Hopper architektúra által hajtott H100 gyorsítókártyában megjelenik a korábban említett Multi Instance GPU második generációs változata[18], valamint a tenzor magok negyedik generációja[15]. Mindemellett megjelent egy új logikai szint az eddigi szálhierarchiában. A blokk klaszter a logikai sorban a blokkok és a grid között szerepel, a blokkhoz hasonlóan lehet egy-, két- és háromdimenziós. A klaszter fő előnye, hogy ezen egységen belül szereplő blokkok Shared Memory L1 gyorsítótára egységes címtartomány alatt elérhető a klaszter minden blokkjából [19]. Ezzel a technológiával lehetővé válik a hierarchikus párhuzamosság egy újabb szintje, amely hatékonyabb algoritmusokat eredményez.

A szuperszámítógépekben elérhető, esetenként több száz GPU működésének szervezése a GPU algoritmusok tervezéséhez hasonló körütekintést igényel. Ebben az esetben az adatlokalitás problémaköre minden egyéb szempontnál fontosabb, mivel a szimpla GPU-n tapasztalható memória olvasási késleltetéshez képest egy klaszter node-jai közti kommunikáció nagyságrendekkel lassabb. GPU klaszter kiépítése során a kártyák jellemzően PCIe, vagy NVLink segítségével vannak összekötve, így az adatátviteli sebesség felső korlátját ezen technológiák átviteli sebessége és sávszélessége adja. Hasonlóan az egy GPU esetben alkalmazott Memory Latency Hiding módszerekhez, lehetséges megfelelő lokalitással és számítási intenzitással a kommunikációból adódó időbeli késés elrejtése.



3. ábra CUDA-aware MPI szemléltetése

Multi-GPU algoritmusok implementációjára gyakran használt módszer a Message Passing Interface (MPI). Ebben az esetben az adatsor SPMD modell alapján kerül felosztásra, majd a szegmensek MPI segítségével kerülnek a klaszter megfelelő gépére és csak ezután kerül sor az adat GPU DRAM-ba másolására. Könnyen belátható, hogy ezzel a módszerrel az adat tesz egy felesleges kitérőt a node operatív tárába

ahelyett, hogy egyből a cél-GPU memóriájába kerülne. Mindemellett érdemes megemlíteni, hiszen ily módon szuperszámítógépre készült MPI programok, valamint egy GPU-hoz készült CUDA programok viszonylag egyszerűen integrálhatóak.

A hagyományos MPI ezen hiányossága javítható úgynevezett CUDA-aware MPI alkalmazásával. A CUDA-aware MPI tulajdonképpen a már GPU DRAM-ba másolt memória egyéb node-okra küldését teszi lehetővé a kód megváltoztatása nélkül a 3. ábrán látható módon. A CUDA-aware MPI fő problémája, hogy a kommunikációt a gazdagép CPU oldalán végzi, CUDA kernelből ezek a funkciók nem elérhetők. Az OpenSHMEM nyílt forráskódú szabványának NVIDIA implementációja, az NVSHMEM lehetővé teszi kártyák közti adatmozgatást közvetlenül CUDA kernelből. Egy másik hasonló programozási modell, az NCCL pedig az MPI interfészéhez hasonló kollektív műveleteket tesz lehetővé több GPU kártyákból álló hálózatokban. Ily módon pont-pont, egy-sok és sok-sok kapcsolat, valamint multi-GPU redukciós sémák alakíthatók ki az MPI interfész mellőzésével, valamint a keretrendszerek lehetővé teszik a klaszteren belüli globális memóriacímzési tér létrehozását [20] [21].

2.2. EEG technológia és jelfeldolgozási problémák

Az agyi képalkotás egy korszerű eszköze az Elektroencefalogram (EEG), melynek segítségével nagy időbeli felbontású kép kapható az agykéreg elektronikus aktivitásáról nem invazív módon. Klinikai gyakorlatban Epilepszia és ADHD vizsgálatára használatos, emellett kognitív folyamatok is feltérképezhetők az emberi agyon belül. A kutatási EEG mérések jellemzően magas mintavételezési frekvenciát (tipikusan 1 – 2 kHz, esetenként akár 32 kHz) és minimum 64 vagy 128 elektródát használnak [22]. Ennek következtében egy néhány perces EEG mérés több száz MB feldolgozandó adatot is generálhat, hosszabb vizsgálatoknál GB méretű adatfájlokról beszélünk. Ilyen feladatméretek mellett fontos a nagy számítási teljesítményű párhuzamos rendszerek használata, miután egyszerű PC eszközön egy GPU használatával is meglehetősen időigényes egy kísérleti alany méréseinek feldolgozása.

Az EEG-vel kapott mérési eredmények feldolgozása több számításigényes lépésből áll, amit több EEG feldolgozó keretrendszer (EEGLAB, Fieldtrip stb.) támogat. A későbbiekben az EEGLAB pipeline [23] előfeldolgozási lépéseire fókuszálunk a dolgozatban, mivel ezek a leginkább számításigényesek. Az adatelőkészítés a folyamat első lépése, melynek során a zajjal terhelt mérésekből olyan adatsor keletkezik, amely alkalmas statisztikai és gépi tanuló algoritmusok számára. Tipikus feldolgozási lépések

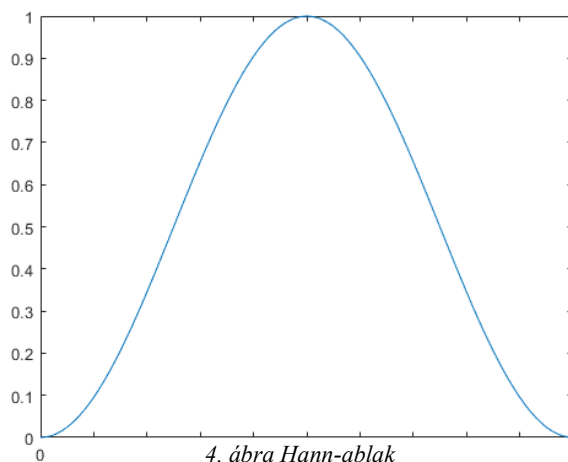
az újra mintavételezés, a különböző digitális szűrőkkel történő jelszűrés, műtermék eltávolítás, ami a nem agyi eredetű elektromos impulzusokat távolítja el a mérésből, valamint a jel energiaspektrumsűrűségének kiszámítása. Műtermékek keletkezhetnek a mérési alany mozgásából, illetve impulzusszerű környezeti behatásoktól. Az energiaspektrum sűrűsége frekvenciaelemzéssel végezhető el különböző Fourier transzformáció alapú periodogram módszerekkel. A spektrum időbeli változása idő-frekvencia elemzéssel kapható meg, a későbbiekben részletesen ismertetett algoritmusokkal.

Fizikai jelek egyik fontos jellemzője azok frekvenciaspektruma, így az EEG előfeldolgozás egyik alapművelete a spektrumszámítás, melynek matematikai alapja a Fourier transzformáció. Számítógépen megvalósított Fourier transzformáció esetén annak diszkrét változatát alkalmazzuk a Cooley-Tukey-féle Fast Fourier Transform algoritmussal számolunk. Egy jel diszkrét Fourier transzformáltja az alábbi összefüggéssel adható meg:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi nk/N}, \quad k = 0, 1, 2, \dots, N-1$$

Az agyi működés vizsgálata érdekében további fontos szempont a frekvenciaspektrum időbeli megváltozása. Ezt idő-frekvencia elemzésnek nevezzük, amely átmenet jelek idő- és frekvenciatartománybeli leírása között. A meghatározhatatlansági reláció következtében mindkét tartomány ismerete egyszerre nagy pontossággal nem lehetséges [24]. Ez a gyakorlatban annyit jelent, hogy hosszabb időszelvény spektruma sűrűbb – tehát nagy pontossággal meghatározható – míg egy rövid időszelvényből a kevesebb minta miatt pontatlan spektrumot lehet előállítani. Számos ismert módszer létezik idő-frekvenciaelemzésre, ezek közül a Short Time Fourier Transform és Wavelet Transform algoritmusok multi-GPU változatai kerülnek bemutatásra.

Idő-frekvencia elemzés során a jelet szinte minden esetben szegmentáljuk. A szegmensek spektrumának számításakor fellépő frekvenciatartománybeli zaj elkerülése érdekében a szegmenseket ablakozzuk. Az ablak egy olyan jel, amely a kezdő- és végpontjában nulla, köztük pedig tetszőleges értékeket vehet fel. Ily módon a megfelelő ablakfüggvény alkalmazásával a szegmensek közötti frekvenciatartománybeli áthallás



4. ábra Hann-ablak

csökkenthető [25]. A legegyszerűbb ablakfüggvény a négyszögjel, amely a megengedett tartományon belül egy, körülötte zérus értéket vesz fel. A későbbiekben a Hann-féle ablakot tekintjük, ami egy paraméteres haranggörbe.

2.3. A Komondor szuperszámítógép

A későbbiekben bemutatott multi-GPU algoritmusok a Komondor szuperszámítógép architektúráját figyelembe véve készültek el. A folyamatok közti kommunikáció és a felhasználható grafikus kártyák architektúrája is fontos tényező, így röviden ismertetem a szuperszámítógép által biztosított lehetőségeket. A jelen dolgozat megírásakor a világ szuperszámítógépeit rangsoroló Top500 listán a 266-ik helyet foglalja el 4,51 petaFlop/s teljesítményével [5]. Korszerű folyadékhűtésének köszönhetően a Green500 listán a 24-ik helyen szerepel. A rendszer architektúrája négy fő partícióból áll. A *CPU only* partícióban kizárólag AMD EPYC processzorok érhetők el, összesen 368 darab. A *Mesterséges Intelligencia (AI)* partícióban vegyesen találhatók meg AMD EPYC processzorok és NVIDIA A100 grafikus kártyák. Nagy mennyiségű gyors elérésű tárhellyel és memóriával látták el a *Big Data (Data Analytics)* partíciót, amelyen Intel Xeon processzorok érhetők el. Az egész rendszer HPE Slingshot interconnect technológiával van összekötve egy három szintű, több mint 10 petaByte kapacitású perzisztens tárhellyel [26].

A számítógép teljesítményének jelentős részét a *Gyorsított (GPU)* partíció adja. A GPU sor 58 node-ból áll, mindegyikben négy darab Ampere architektúrájú NVIDIA A100 grafikus kártyával, így összesen 232 GPU használható egyszerre számításra a rendszerben [26]. A folyamatok közti kommunikáció lebonyolítása érdekében a Komondor több párhuzamos programozási modellhez biztosít implementációt. A későbbiekben bemutatott algoritmusok a Cray MPICH modell CUDA-aware változatára készültek. Ily módon az MPI 3.1 szabványnak megfelelő MPI funkcionalitás érhető el az egy node-on belüli GPU kártyák közti közvetlen kommunikációval. A különböző modelleket, futtatási környezeteket és előre telepített programokat a Komondoron modulokba szervezték. A modulok fordítás és futtatás során dinamikusan betölthetők, ennek köszönhetően a rendszer minden esetben a megfelelő módon linkeli a program által használt könyvtárakat.

3. Algoritmustervezés

A következő jelfeldolgozó algoritmusok minden esetben több csatornás jelek figyelembevételével készülnek. A feldolgozandó adat minden esetben sorfolytonosan helyezkednek el a memóriában, tehát a különböző csatornák egymást után egyenlő hosszúságban. Ez alapvetően meghatározza a későbbi GPU programok indexelési sémáját, valamint a grid és blokkok felosztását. Miután a grid vízszintes dimenziójának maximális megengedett mérete minden esetben nagyságrendekkel nagyobb a függőlegesnél, az adat felosztása szempontjából előnyös ez a fajta adatrepresentáció.

Az adattárolási struktúra mellett két további tervezési elv vonatkozik minden algoritmusra. Elsősorban, a GPU programok minden esetben alkalmazhatók legyenek a teljes probléma megoldására, azokba beépített multi-GPU architektúrát feltételező kommunikáció ne kerüljön. Ezt a kritériumot a tesztelhetőség érdekében állítom, mivel így megmutatható ugyanannak az algoritmusnak a párhuzamos skálázhatósága a GPU-k számának – és így a párhuzamosság fokának – megnövelésével. Továbbá ily módon az implementáció egy GPU kódtól szigorúan elválasztott MPI részegységgel alakítható át multi-GPU algoritmussá, de a kernelek validációja a fejlesztés során egy kártyán elvégezhető.

A második kritérium, hogy az előbbi GPU kódtól függetlenül működő kommunikáció esetén olyan adatfelosztási séma kerüljön kialakításra, amivel strong scaling-et lehetővé tevő multi-GPU algoritmus jöjjön létre. Ennek megfelelően a probléma méretének felskálázása nélkül is legyen lehetőség több – esetenként több száz – GPU hozzáadására. Ennek a vezérelvnek a figyelembevételével olyan algoritmusok építhetők fel, amelyek hatékonyan alkalmazhatók pre-exascale szuperszámítógép architektúrákon. Az EEG mérések során kapott jelek időben mintavételezettek, így a későbbiekben az adatsor indexelése során a $s[nT]=s(t)$ jelölést alkalmazom, amelyben T a mintavételezési periódusidőt, n pedig a minta sorszámát jelöli az adatsorban.

3.1. Konvolúció

A konvolúció számos tudományterületben alkalmazott sorművelet, a digitális jelfeldolgozásban szűrők megvalósítására használható, amelyek az eredeti jelnek csak bizonyos frekvenciakomponenseit eresztik át. Megkülönböztetünk alul- és felüláteresztő, valamint sávszűrőket. Alkalmazásuk célja EEG vizsgálatok során a mérés során jelbe kerülő nem kívánatos frekvenciakomponensek – például az 50Hz zaj – eltávolítása az

adatokból. Időben mintavételezett jelek esetén az alábbi összefüggés írja le a konvolúciót:

$$c[m] = \sum_{n=0}^{N-1} w[n] \cdot s[m-n]$$

Az összefüggésben az s változó a bemenő jelre, a w változó pedig a konvolúciós ablakra vonatkozik. Mint látható, a konvolúció eredménye egy adatpontra a szomszédjai súlyozott átlaga, a súlyokat a szűrő típusa és paraméterei határozzák meg. Nagy felbontású adatokra és nagy méretű ablakokra a konvolúció számításigényes művelet, azonban hatékonyan párhuzamosítható, miután az eredmény minden eleme egymástól függetlenül kiszámítható. Problémát jelent ilyenkor, hogy minden elem számára szükséges a szomszédos cellák ismerete [27]. Az alábbi pszeudokóddal írható le a szekvenciális algoritmus működése, ami a bemeneti adat hosszától (M), az ablak hosszától (N) és a csatornák számától (C) függően $C \cdot (N + M - 1)$ lépést végez.

```

for c := 0 to C do
  for m := 0 to M+N-1 do
    result[c,m] := 0.0
    for n := 0 to N-1 do
      result[c,m] += w[n] * s[c,m-n]
    end
  end
end

```

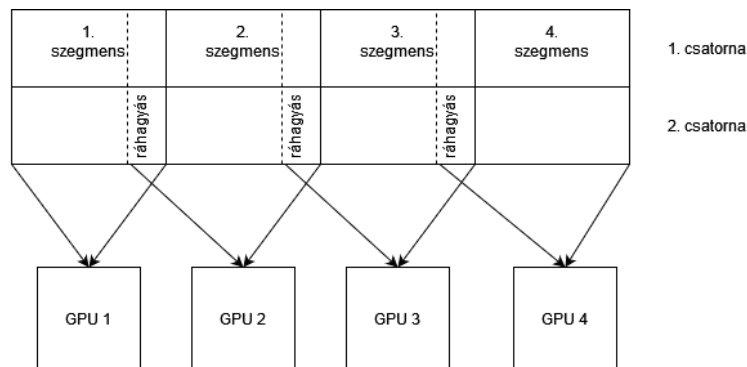
Közvetlenül a konvolúciós szummából következik a GPU megvalósítás. Az algoritmus bemenete sorfolytonosan tárolt többcsatornás jel, így a CUDA szálhierarchiában egy kétdimenziós grid szükséges. Ebben a gridben a csatornák eredményeinek adatpontjai a vízszintes dimenzióban helyezkednek el, soronként egy csatorna, így az adat előzetes módosítás nélkül közvetlenül használható a GPU-ra másolás után. Miután a konvolúció során minden eredménypont egymástól függetlenül meghatározható, belső szinkronizáció nem szükséges, a kernel minden példánya teljes mértékben párhuzamosan futhat. A két külső iteráció ezzel a módszerrel párhuzamosság útján realizálható, azokra a kernelen belül már nincsen szükség.

A következő megoldandó probléma a memóriáhozáférés optimalizálása. Miután a konvolúciós ablak minden csatorna esetén megegyezik, valamint általános esetben sokkal rövidebb, mint a jel hossza, alkalmazható a CUDA memóriamodellből ismert shared memory. Ennek köszönhetően a kernel indulásakor minden blokk a globális memóriából a saját shared memory-jába másolja az ablakot, így a kernelben futó

ciklusnak kizárólag a jel értékeit szükséges minden esetben betölteni. Ily módon hatékonyan kihasználható a CUDA által biztosított hierarchikus memória.

Az így létrejött GPU algoritmus képes tetszőleges sorfolytonosan tárolt jel csatornánként konvolúciójának eredményét megadni, így korábban előírányzott első feltétel teljesül. A paraméterezés kialakítása lehetővé teszi, hogy több lehetséges multi-GPU elosztási séma alkalmazható legyen a kernel megváltoztatása nélkül. A legkézenfekvőbb szétosztási rendszer az adatsor csatornák mentén történő elválasztása. Ebben az esetben a sorfolytonos tárolás következtében a kommunikáció egy MPI_Scatter kollektív művelet formájában jelenik meg. A módszer hátrányát az adja, hogy a csatornák száma közvetlenül korlátozza az alkalmazható párhuzamosság mértékét, így nem feltétlenül alkalmazható hatékonyan extrém párhuzamos rendszereken.

A párhuzamosság skálázhatóságának javítása érdekében szükséges egy olyan elosztási rendszer, amely a bemeneti adathossztól és a csatornák számától függetlenül lehetővé teszi a multi-GPU alkalmazást. Ebből a célból az adat a csatornahossz mentén kerül felosztásra, amelyben az egyes folyamatok a szegmens minden csatornáját megkapják. Ezzel a módszerrel azonban felmerül két komplikáció. Elsősorban, minden folyamat esetén az összetartozó csatornaszegmensek nem folytonosak a memóriában. Ez azt jelenti, hogy egy adott szegmens bármely csatornájától C távolságra van a szegmens következő csatornája. A felosztásra ily módon a folytonos MPI kollektív műveletek (MPI_Scatter, MPI_Gather) nem alkalmazhatók, azok aszimmetrikus változatára az MPI_Scatterv és MPI_Gatherv függvényre van szükség.



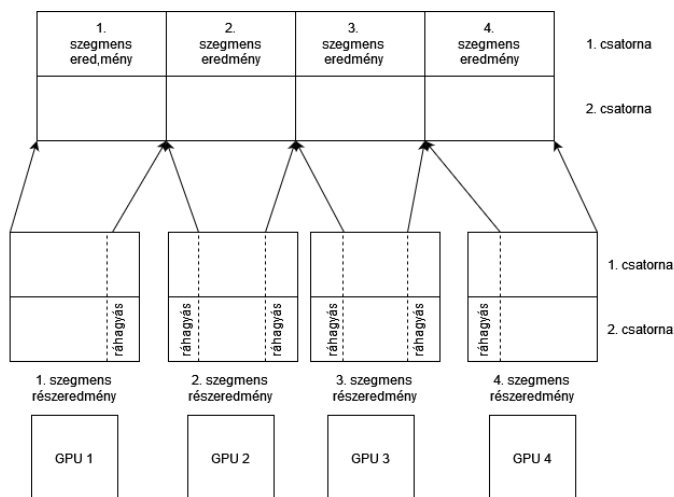
5. ábra Konvolúciós algoritmus szétosztási séma

A második komplikáció, hogy a szegmensek között átlapolást kell alkalmazni, mivel a részeredmények csak akkor korrektek numerikusan, ha azok határpontjai kiszámításához a szomszédos szegmensek megfelelő elemei ismertek a folyamat számára. Belátható, hogy az első szegmens elejét és az utolsó szegmens végét leszámítva minden köztes szegmens esetén a két szomszédosból $N-1$ hosszú ráhagyás szükséges,

ahol N a konvolúciós ablak hossza. Ez a ráhagyás a szegmens minden csatornája esetén megjelenik, így extrém esetben jelentősen megnövelheti a kommunikált adat méretét, azonban a gyakorlatban az ablak hossza nem számottevő, így az algoritmus különösebb gond nélkül skálázható.

Az imént ismertetett szétosztási rendszerben a csatornák mentén is lehetséges a problémátér skálázása. Tételezzük fel, hogy egy alany esetén 128 csatornán történik mérés. Ebben az esetben egy multi-GPU architektúrán a feladat szétosztható tetszőleges mennyiségű GPU-ra, amennyiben az átlapolódások rövidebbek, mint az egy kártyára jutó csatornahossz. Ebben az esetben minden kártyán kétdimenziós grid-en zajlik a számítás, azonban a 128 csatorna nem feltétlenül fedi le oly mértékben a GPU kapacitását, hogy az maximális teljesítményen működjön. Csoportos vizsgálatok feldolgozása esetén azonban a memóriapuffer méretétől függően hozzáadhatók több alany csatornái is a rendszerhez anélkül, hogy jelentős teljesítménybeli csökkenés lenne tapasztalható.

A csatornák felosztásának következtében a részeredmények összegyűjtésére is aszimmetrikus séma szükséges. A ráhagyások miatt a szegmenshatárokon lévő eredmény értékek nem korrektek, így minden esetben a ráhagyás mértékének megfelelő cellától szükséges kezdeni a részeredmények összeillesztését. A szétosztáshoz hasonlóan az első és utolsó szegmens esetén nem szükséges mindkét oldalon a ráhagyás beszámítása. Az eredmények összegyűjtését az alábbi ábra szemlélteti. A módszer kifejezett hátránya, hogy a kommunikációs műveletek kizárólag szakaszonként, iterációban folytathatók le, így a hálózati összeköttetést több adatküldési művelettel terheli.



6. ábra Konvolúciós algoritmus, részeredmények összegyűjtése

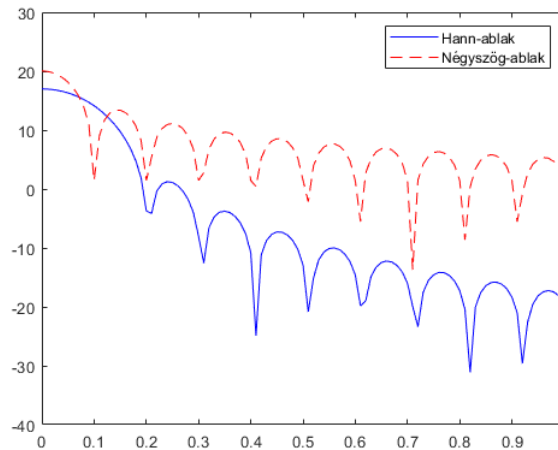
3.2. Short-Time-Fourier Transzformáció (STFT)

Jelek frekvenciaspektrumának pontos időbeli változása a korábban említett meghatározhatatlansági reláció következtében nem megvalósítható, azonban számos közelítő módszer létezik az idő-frekvenciaspektrum meghatározására. A short-time-Fourier transzformáció széles körben alkalmazott algoritmus, meglehetősen intuitív módon megközelíthető. Az algoritmus alapja a jel felosztása rövid szegmensekre, majd a szegmenseken Fourier-transzformáció elvégzése. Nyilvánvaló, hogy a szegmensek hossza a mintavételezési törvény miatt nem lehet tetszőlegesen rövid, alacsonyabb frekvenciák hosszú időablakokat igényelnek [28]. A diszkrét STFT algoritmus formálisan a következőképpen definiálható [25], ahol s a jelet, w az ablakfüggvényt reprezentálja:

$$X_m(\omega) = \sum_{n=-\infty}^{\infty} s(n)w(n - mN)e^{-j\omega n}$$

Az algoritmus a következő lépésekből áll. Paraméterként ismert a szegmensek hossza, valamint az átlapolásuk mértéke. A gyakorlatban 50% átlapolást alkalmazunk ezért egy csatornán belül az m -ik szegmens kezdőpontja az $m \cdot m_{overlap}$ összefüggéssel határozható meg. Minden szegmens esetén valamilyen ablakfüggvény alkalmazása szükséges, mivel előfordulhat, hogy a szegmenshatáron nem nulla amplitúdójú értékek találhatók, ami spektrumtorzulást okoz. Ezután minden szegmensre kiszámításra kerül a spektrum a Cooley-Tukey FFT algoritmussal.

Fontos megjegyezni, hogy az ablakozással a jel megváltozik (az ablak modulálja a jelet), ami a frekvenciatartománybeli ábrázolásban is megjelenik. Mindig olyan ablakot célszerű alkalmazni, amely frekvenciaspektrumában minél kevesebb magas frekvenciájú felharmónikus jelenik meg. Példának okáért az alábbi ábrán látható módon a meglehetősen egyszerű négyszög-ablak spektrumának eloszlása sokkal szélesebb. Az általunk későbbiekben használt Hann-féle ablakfüggvény esetén azonban sokkal keskenyebb, így ideális erre a célra [29]. A további tervezés során egy újabb kritérium olyan implementáció készítése, amelyben az ablakfüggvény kevés módosítással megváltoztatható.



7. ábra Hann-ablak és Négyyszög-ablak összehasonlítása frekvenciatartományban, dB skálán

Miután a Fourier transzformáció széles körben alkalmazott művelet, így elérhető előre elkészített megvalósítás a cuFFT könyvtár formájában. A könyvtárat az ismert FFTW interfész alapján alakították ki, valós és komplex adatsorokra biztosít hatékony $n \cdot \log(n)$ aszimptotikus futási idejű implementációt. A cuFFT interfész lehetővé teszi több rövidebb jel egyszerre történő transzformációját – batch-FFT [30] – így az STFT megvalósításakor használható szegmensek transzformálására.

A cuFFT könyvtár használata a következő lépésekben jelenik meg. A transzformáció használatához egy konfigurációs-inicializáló lépésre van szükség (cuFFT Plan struktúra létrehozása), amelyben megadható a batch szám, valamint a transzformáció típusa. Az így létrejött konfigurációs objektummal a GPU memóriában található adatok megadásával hajtható végre a megfelelő transzformációs lépés [30]. Implementációs szempontból fontos megjegyezni, hogy a cuFFT funkcionálisai nem elérhetők közvetlenül CUDA kernelekből. Ennek következtében több részegységből álló megoldás elkészítése szükséges: egy saját készítésű kernel, amely a szegmentálást és ablakozást végzi, valamint egy host-oldali kód, ami a megfelelően előkészített adatokra elvégzi a transzformációkat egy cuFFT hívás segítségével.

Az előre megvalósított FFT algoritmust figyelembe véve az algoritmus az alábbi pszeudokód által írható le:

```

for c := 0 to C do
  for m := 0 to M do
    FFT(s .* w[n - m*N])
  end
end

```

Ismert, hogy a cuFFT könyvtárban implementált Fourier transzformáció $n \cdot \log(n)$ futási idejű, így a szegmensek számától függően $C \cdot M \cdot N \cdot \log(n)$ az algoritmus teljes futási ideje, ahol C a csatornák száma, N a szegmensek hossza, M pedig a szegmensek száma, ami 50% átlapolást feltételezve L hosszú csatornák esetén

$$M = 2 \frac{L-N}{N}.$$

Párhuzamosítás szempontjából vizsgálva a problémát látszik, hogy minden szegmens egymástól függetlenül számítható, azonban a batch-FFT megvalósítása miatt szükséges a szegmensek átlapoló részeinek duplikálása. Ezt szem előtt kell tartani a szétosztási rendszer megtervezésénél. Multi-GPU rendszer kialakításához szükséges egy magas szintű szegmentálás, amely a csatornákat folyamatok és GPU kártyák között szétosztja, valamint egy alacsony szintű szétosztás, ami egy csatornát az STFT algoritmus szerint átlapolással szegmentálja, ami alkalmas bemenet a cuFFT transzformációs függvényének.

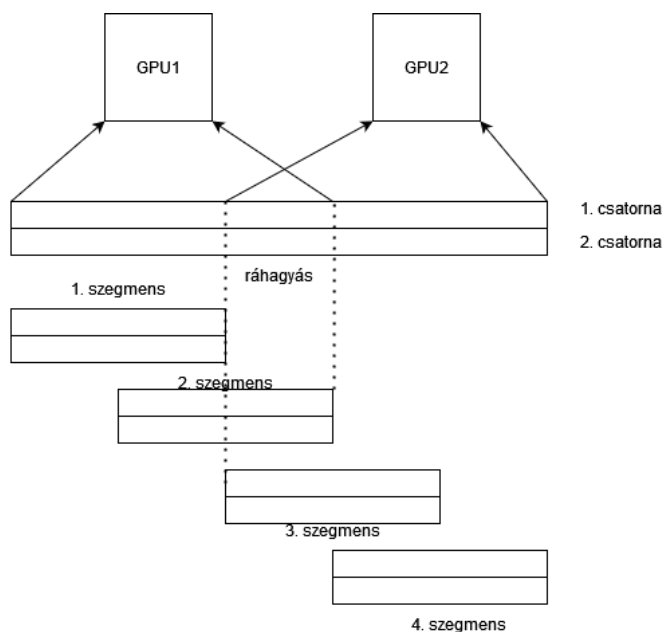
A létrehozandó CUDA kernel egyik feladata az ablakozás elvégzése, ehhez minden elem esetén szükséges az ablak megfelelő értékének ismerete. A korábban említett ablak konfigurálhatósági követelmény egy lehetséges megoldása az ablakok előre generálása. Ekkor azonban minden szegmens esetén memóriaeléréseket eredményez, azonban a kernel számítási intenzitásának növelése és a felesleges memóriamozgatás elkerülése érdekében célszerű egy GPU függvény által helyben generálni az értékeket.

Az elosztási kernel további feladata, hogy a csatornákat a szegmenshosszok függvényében a megfelelő alakra hozza a memóriában. A batch-FFT működéséhez egy csatorna minden szegmenségek sorfolytonosan egymás után kell elhelyezkednie a memóriában, valamint egy csatornának a másik után. A batch-FFT meghívásakor a batch-ek számát az egy csatornára jutó szegmensek és a csatornák számának szorzata adja, hosszukat pedig értelemszerűen a korábbi szegmenshossz. A felhasznált GPU memória minimalizálása érdekében a szétosztás során a szegmensek értékei egyből a kimeneti memóriarészbe kerülnek a megfelelő pozíciókra és a transzformáció helyben működő változata került felhasználásra.

Folyamatok közötti szétosztás során ismételten lényeges szempont a párhuzamosság skálázása, így a csatornák mentén történő elosztás nem megfelelő. A szegmensek rendszerint rövidek, így nagy számú minta esetén a szegmensek száma jelentősen nagy. Emiatt célszerű azok mentén felosztani az adatsor minden csatornáját és

ezeket egységben kezelni a konvolúció szétosztási sémájához hasonlóan. A felosztás a szegmensek számán alapul, így azok száma korlátozza a használható grafikus kártyák számát, miután egy kártyára legalább egy szegmensnek jutnia kell.

A konvolúcióhoz hasonlóan ez az elosztás lehetővé teszi több kísérleti alanytól származó csatornák hozzáillesztését a rendszerhez, a párhuzamosság foka és a hatékonyság ebben az esetben nem változik. A szétosztás ebben az esetben is aszimmetrikus, így az MPI_Scatterv függvény szükséges a művelet elvégzéséhez.



8. ábra STFT szétosztási séma

A szegmensek Fourier-transzformált alakja önmagában képezi az algoritmus kimenetét. Ennek megfelelően az eredmény háromdimenziós. Az adatok memóriabéli elhelyezésének konzisztenciája érdekében a sorfolytonosság megmarad, így a szegmensek transzformáltjai egymás után találhatók a kimenetben. Egy csatorna szegmenseit a következő csatorna szegmensei követik, így módon indexelhető az eredményvektor. A begyűjtés során így a folyamatok csatornái esetén aszimmetrikusan kell, hogy történjen, mivel a folyamatokból visszagyűjtött szegmensek különböző csatornákhöz tartoznak.

3.3. Wavelet transzformáció

A wavelet transzformáció egy újabb idő-frekvencia elemzésre használható módszer, amelyet kifejezetten a Fourier transzformáció és az azon alapuló idő-frekvencia elemző eszközök hátrányainak kiküszöbölése érdekében fejlesztettek ki. Rövid időtartamú oszcillációk használata időben lokalizált frekvenciaspektrum

meghatározására Alfred Haar nevéhez köthető. A fő eltérés a Fourier transzformációtól az alkalmazott bázis függvény időbelisége [31]. Míg a Fourier transzformáció esetén a szinusz és koszinusz függvények végtelen oszcillációk, a wavelet véges energiájú, rövid időtartamú oszcilláció. Ennek megfelelően wavelet olyan időtartománybéli függvény, ami teljesíti az alábbi két feltételt [32]:

- Véges energia

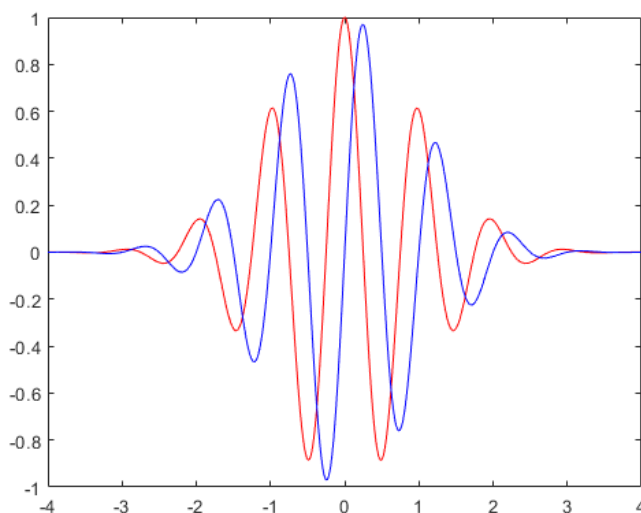
$$\int_{-\infty}^{\infty} |f(t)|^2 dt \neq \infty$$

- Nulla átlag

$$\int_{-\infty}^{\infty} f(t) dt = 0$$

Számos ismert és gyakorlatban alkalmazott bázis-wavelet létezik, a későbbiekben a Morlet-féle wavelet-et alkalmazzuk EEG előfeldolgozásra, azonban az algoritmus kialakítása lehetővé teszi egyéb bázis wavelet-ek megvalósítását is. A Morlet-féle wavelet az alábbi képlettel előállított haranggörbével tompított komplex szinuszoid:

$\Psi(t) = e^{j\omega_0 t} e^{-\frac{t^2}{2}}$. A jel valós és képzetes komponensének viszonya a 9. ábrán látható.



9. ábra Morlet-féle bázis-wavelet valós és képzetes komponense

A wavelet transzformáció művelete egy konvolúció a vizsgált jel és a bázis wavelet különböző skálázott változatai között. A skálázás kiválasztható olyan módon, hogy a vizsgálat számára fontos frekvenciaértékekkel egyezzen meg az így előállított wavelet frekvenciája. A műveletet folytonos esetben az alábbi konvolúció írja le [32], amelyben $\Psi(t)$ egy tetszőleges bázis-wavelet.

$$W\{f(t)\}(a, b) = \int_{-\infty}^{\infty} f(t) \Psi\left(\frac{t-b}{a}\right) dt$$

Diszkrét idejű mintavételezett időtartományban az integrált felváltja a korábban említett konvolúciós szumma. Ennek megfelelően az összes csatornán elvégzett számítást az alábbi pszeudokód írja le:

```

for c := 0 to C do
  for f := fmin to F do
    for m := 0 to M+N-1 do
      re := 0.0
      im := 0.0
      for n := 0 to N-1 do
        re += w_re[c, f, n] * s[c, m-n]
        im += w_im[c, f, n] * s[c, m-n]
        result[c, f, m] := sqrt(re*re + im*im)
      end
    end
  end
end
    
```

Mint látható, az algoritmus által végrehajtott műveletek száma függ a bemeneti jel méretétől (M) és a vizsgált frekvenciakomponensek számától (F). Értelmszerűen a csatornák (C) száma egy újabb iterációt ad a programba, a bázis wavelet hossza (N) pedig a mintavételezési frekvenciától függ. Ennek következtében azonos frekvencián mintavételezett jelek összehasonlításakor a wavelet hossz konstans értéknek vehető. Ennek megfelelően az algoritmus $C \cdot F \cdot (M + N - 1)$ lépést hajt végre.

A frekvenciakomponensek száma az algoritmus paraméterezéséből következik. Az időtartomány mintavételezéséhez hasonlóan három paraméter alapján (minimum és maximum frekvencia, valamint a frekvenciakomponensek száma) az alábbi összefüggéssel állíthatók elő a frekvenciakomponensek:

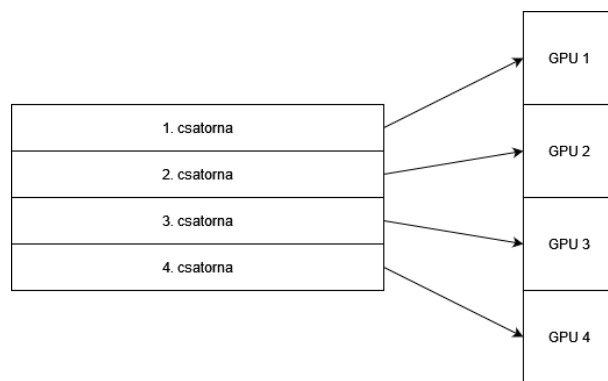
$$f_{min} + \frac{f_{max} - f_{min}}{F}$$

Miután a wavelet transzformáció a korábban részletesen ismertetett konvolúció műveletre vezethető vissza, az elkészítendő műveletvégző kernel CUDA szálhierarchia használati módja ahhoz nagyban hasonlít. A különböző frekvenciakomponensek feldolgozása miatt a kernelen belül egy ciklusra van szükség, hogy a grid konfigurációt ne kelljen megváltoztatni. Ennek megfelelően az adatvektor továbbra is kétdimenziós, azonban az eredményvektor háromdimenziós. Miután a definícióból adódóan a Morlet-wavelet komplex értékű függvény, a wavelet generáló és a transzformációt elvégző kernelekben is komplex aritmetika került alkalmazásra.

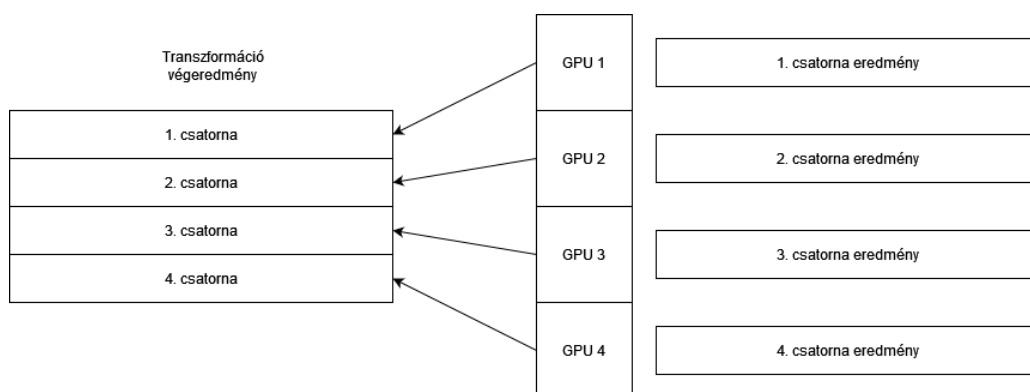
A konvolúció során alkalmazott ablak (template) a jelen esetben megfelelnek wavelet transzformáció során alkalmazott skálázott bázis-wavelet-eknek. Ennek megfelelően az így létrejövő konvolúciók egy ún. filterbank struktúrát alkotnak, ami több sávszűrő együttes alkalmazása. A bázis-wavelet-ek hossza kizárólag a mintavételezési frekvenciától, valamint a megcélzott frekvenciasáv méretétől függ. Ily módon a filterbank a konfigurációs paraméterek alapján a GPU kártyán egyből annak operatív tárhelyébe generálható, így csökkenthető a host – GPU memóriamásolás. Magas mintavételezési frekvencián a waveletek mérete több MB méretűre nőhet, így a shared memory nem alkalmazható a nagyszámú frekvenciakomponens esetén gyorsítótárazásra. Ennek következtében a hardver limitációi miatt a konvolúció során a globális memória használata indokolt.

A folyamatok közti adatszétosztás során csak bizonyos esetekben alkalmazható a konvolúciónál látott megoldás. Ennek oka, hogy nagy mintavételezési frekvencia esetén a wavelet hossza – és így a szegmensek szélén a ráhagyás – jelentős méretűvé válik, ami nagy mennyiségű felesleges adatmásolást eredményez, valamint kisebb problémákra ellehetetleníti a felosztást. Ennek következtében a wavelet transzformáció esetén indokolt a csatornák szétosztása. Ily módon a párhuzamosság skálázhatósága korlátozott a csatornák által, azonban egyrészt az elvégzett EEG kísérletek szinte minden esetben 128 és 256 csatornával zajlanak, így a gyakorlatban megfelelő fokú párhuzamosság biztosítható 128 GPU kártyával. Másrészről, amennyiben egy GPU végzi egy csatorna transzformációját, a grid oly módon alakítható át, hogy a csatornák helyett frekvenciakomponensek képezzék az „y” dimenziót, így tovább gyorsítva a kernel futását.

A szétosztási rendszerben továbbá ily módon nem szükséges ráhagyásokat küldeni a csatornák között, így a felesleges adatok kommunikálása és az aszimmetrikus szétosztó műveletek nem szükségesek. Az egész művelet egy MPI_Scatter művelettel elvégezhető az alábbi ábra alapján.



10. ábra Wavelet transzformáció szétoztási séma



11. ábra Wavelet transzformáció összegyűjtési séma

Miután a GPU-k a jelen szétoztási rendszerben egész csatornákkal dolgoznak, nem szükséges az aszimmetrikus kommunikációs művelet az eredmények összegyűjtésekor. Ennek megfelelően az MPI_Gather művelet elégséges a végeredmény előállításához. Amennyiben a konvolúciónál ismertetett szétoztási sémát alkalmazzuk wavelet transzformációhoz, az eredmény összegyűjtése során nem csak csatornánként lenne szükség aszimmetrikus adatmásolásra, hanem a csatornák esetén minden frekvenciakomponensre. Ez azt eredményezi, hogy a frekvenciatartománybeli felbontás és a párhuzamosság fokának növelésével drasztikusan megnőne a kommunikációs műveletek száma, tehát kevés nagy üzenet helyett sokkal több kis üzenetet kéne a rendszernek kezelnie.

3.4. Implementációs sajátosságok

Általános esetben egy GPU program a következő lépésekből áll: memória lefoglalása a kártyán, memória átmásolása a host memóriából az imént lefoglalt memóriába, a kernel elindítása az adatokon, majd az eredmények visszamásolása a GPU-ból a host memóriába. Miután a Komondor szuperszámítógépen az MPICH programozási környezet használatához speciális fordító szükséges, ami azonban nem képes CUDA kernelek programkódjait lefordítani, így szükségessé vált a GPU és host

kódok szétválasztása külön fájlokba. Az imént felvázolt GPU program struktúra alapján ez könnyen megoldható, mivel egyetlen függvényben elhelyezhető a teljes vezérlő kód, ami a GPU memóriát és a kernelek meghívását kezeli. A lefordított kódok linkelési fázisához szintúgy használható a Cray környezet C fordítója.

CUDA-aware MPI használatához azonban szükségesek a kommunikációt lebonyolító kódban a GPU-n található mutatók, így az egy függvényben megvalósított memóriamásolással és kernelhívással a megoldás nem kivitelezhető, mivel ily módon nem érhető el közvetlenül a GPU-n lefoglalt memória. A Cray környezet C fordítója képes a host oldali CUDA API függvényeit lefordítani és linkelni, azonban a későbbiekben ismertetett lokális tesztelés és a GPU kód szigorú elkülönítése végett egy olyan interfész került kialakításra, amely becsomagolja az összes felhasznált CUDA függvényt az eredetiek szerinti paraméterezéssel. Ezzel a módszerrel elérhető, hogy az adatok szétosztását és az eredmények összegyűjtését végző kód teljes mértékben független legyen a CUDA könyvtáraitól, így bármilyen MPI kompatibilis fordítóval és környezettel tesztelhető legyen. Az alábbi táblázatban látható az elkészített függvények deklarációi.

1. táblázat CUDA programozói felülethez készített wrapper-függvények deklarációi és funkcionálisai

<code>int malloc_device(int gpu_idx, void** dev_ptr, size_t length)</code>	Memória lefoglalása GPU-n-
<code>int free_device(int gpu_idx, void* arg1)</code>	GPU-n lefoglalt memória felszabadítása.
<code>int memcpy_host_device(int gpu_idx, void* dev_ptr, void* host_ptr, size_t length)</code>	Host-ról GPU-ra történő memóriamásolás.
<code>int free_device(int gpu_idx, void* arg1)</code>	GPU-ról host-ra történő memóriamásolás.

A szuperszámítógépre készült programok elosztott memóriás, több GPU-t tartalmazó rendszert feltételeznek, azonban az algoritmustervezési és tesztelési fázisban ez az infrastruktúra többnyire nem elérhető. A közvetlenül a szuperszámítógépen történő fejlesztés Integrált Fejlesztői Környezet és Grafikus felület hiányában a gyakorlatban korántsem praktikus megoldás. Miután a kommunikációért felelős kód irányába a GPU-hoz köthető komponensek a korábban ismertetett wrapper-interfészen keresztül jelennek

meg, a közös interfészt használva a GPU működés hagyományos C kóddal utánozható. Ebben az esetben a GPU memóriaműveleteket a gazdagép operatív tárolójában elvégzett műveletek helyettesítik.

Fejlesztés és tesztelés során egy hagyományos PC-n a mock függvények objektumkódjának linkelésével egy virtuális MPI klaszterrel szimulálható a később szuperszámítógépen GPU memóriával zajló kommunikáció, mindeközben a GPU kód ettől teljesen függetlenül egy grafikus kártyán tesztelhető és validálható. Az így létrejött kódszervezési struktúra alkalmasnak bizonyult mind a három bemutatott algoritmus átlátható módon történő fejlesztésére oly módon, hogy a tesztelési fázisból alkalmazási fázisba elegendő a linkelés módosítása az interfész C helyettesítő megvalósításáról a CUDA objektumkódjára.

A megvalósított algoritmusok mindegyike esetén szükséges a CUDA környezetben fellépő belső hibák detektálása, valamint a futási idő mérése. CUDA hibák detekciója az ilyen környezetben gyakran alkalmazott procedurális C makrókkal történik, amikkel a CUDA API függvények körbe vehetők. Minden CUDA függvény egy lefutási eredményt kódoló értékkel tér vissza, így az a makró törzsében kiolvasható és ellenőrizhető. Alkalmazása az alábbi kódpéldában látható. A futási idő mérésére számos lehetőség áll rendelkezésre C programnyelven, amelyek implementációja operációs rendszertől függően változhatnak. A feltételes fordítás végett az időmérés is procedurális makrókkal lett megvalósítva, továbbá fordítási időben megadható, hogy melyik operációs rendszerre jellemző kóddal értékelődjenek ki.

```
CHECK_ERR(cudaMalloc((void**)&dev_ptr, N * sizeof(float)));
```

Az MPI szinte minden elosztott memóriás rendszeren megtalálható, széles körben alkalmazott programozási modell, azonban számos esetben közvetlen gátja programok skálázásának. Ezek a nehezen visszakövethető problémák felléphetnek mind a párhuzamosság, mind a feladat bemenetének növelésével [33]. A korábban bemutatott algoritmusok esetén a probléma méretének növelésével két probléma lép fel, mindkét eset az MPI interfész kialakításának következménye. Minden MPI függvény esetén a kommunikált adat hossza 32 bites előjeles egész típusú, emiatt legfeljebb $2^{31}-1$ elem küldhető el egy hívásban. Bármely hosszabb adatsor esetén a változó túlsordul, ami negatív elemszám küldését és futási idejű hibát eredményez.

Kollektív műveletek esetén a probléma forrásának megtalálásához azok közelebbi vizsgálata szükséges. A kollektív aszimmetrikus szétosztás művelet

(MPI_Scatterv) művelet deklarációja alább látható. A függvényben a *counts* változóval dinamikusan konfigurálható a különböző folyamatoknak küldendő üzenetek hossza, a *displacements* változóval pedig a folyamatoknak küldendő adatszegmensek kezdőpontja. A korábban említett $2^{31}-1$ elemnél hosszabb kezdő adat esetén a változó túlcsordulása miatt ismételt futási hibát okoz (displacement buffer corruption [33]).

```
int MPI_Scatter(const void* buffer_send,
               int count_send,
               MPI_Datatype datatype_send,
               void* buffer_recv,
               int count_recv,
               MPI_Datatype datatype_recv,
               int root,
               MPI_Comm communicator);
```

A korábban ismertetett problémák következtében a kollektív műveletekre szükségessé vált a szétosztás és összevonás (MPI_Scatter, MPI_Gather) egyéni megvalósítása ismert, kis méretű szegmensekre bontással. Így a megvalósított program ellenőrzi, hogy a küldendő adathossz meghaladja-e a kritikus mértéket, amely felett a manuálisan implementált pont-pont kommunikációra bontott műveletekkel végzi az adatok küldését, ellenkező esetben a beépített kollektív műveleteket használja.

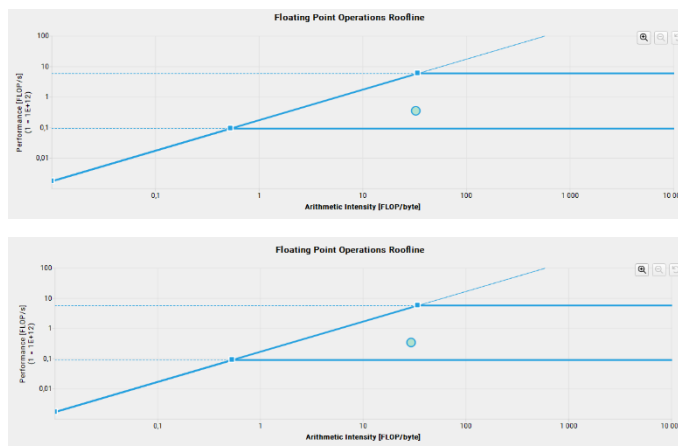
4. Eredmények

Az elkészült algoritmusok teljesítménye két módon került mérésre. A kernelek tesztelése egy NVIDIA RTX 3050 kártyán történt, melynek célja azok teljesítményének jellemzése volt a roofline modell segítségével. A roofline elemzés az NVIDIA Nsight Compute Profiler program használatával történt. A multi-GPU algoritmusok tesztfuttatása a korábban részletesen bemutatott Komondor szuperszámítógépen történt. A tesztek célja igazolni, hogy a létrehozott multi-GPU felosztási sémák valóban lehetővé teszik a párhuzamosság skálázását bármilyen problémaméretre. Miután nagyfokú párhuzamos rendszerekben a kommunikációs idő már számottevő, az egy kártyára jutó kernelek futási ideje mellett megmérésre került az algoritmus teljes futási ideje. Ez minden esetben a szétosztáskor megtörténő első kommunikációs lépés elejétől az összegyűjtés végéig tartó időintervallumot jelenti, így a teljes időben szerepel az összes lefuttatott kernel és kommunikáció. Minden grafikonon feltüntetésre került továbbá az ideális gyorsulás referenciaegyenese, amely esetén feltételezzük, hogy a párhuzamos végrehajtási egységek növelésével egyenesen arányosan csökken az egy GPU megoldás futási ideje. Minden mért idő ezredmásodperc egységben értelmezendő.

Az algoritmusok ismeretében sejthető, hogy a kevesebb műveletet végző kernelek esetén (STFT, Konvolúció) jelentős mértékben a kommunikációs időtől függ az összteljesítmény. Ennek oka az, hogy a kernelek futási ideje ilyenkor ezredrésze is lehet az adatmozgatáshoz szükséges kommunikációs időnek. Skálázhatóság szempontjából így külön értelmezendő a teljes algoritmus skálázhatósága nagy rendszerekre, illetve a kernel futási idejének csökkenése a párhuzamosság növelésével.

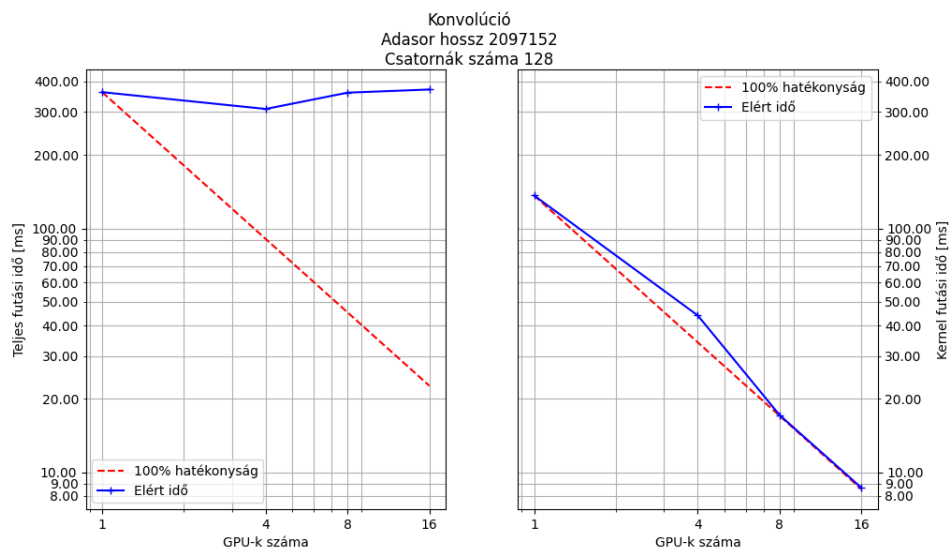
4.1. Konvolúció

A tervezés során létrehozott konvolúciós szétosztási séma hatékony működésének egyik alapvető feltétele, hogy a műveletvégző kernel teljesítménye több csatorna hozzáadásával ne csökkenjen. Ebből a célból a roofline analízis során megvizsgáltam a kernel teljesítményét egy alany – általános esetben 128 csatorna – valamint nyolc alany – 1024 csatorna – esetén. A roofline ábra 32-bites lebegőpontos számokkal végzett műveletekre vonatkozik, a felső egyenes a mérvadó korlát. Az ábrán látható, hogy mindkét esetben közel azonos 300 gigaFlop/s körüli teljesítmény érhető el, valamint a kernel a számítási és memóriakorlátozottság határán van. Ily módon a kernel a csatornák mentén skálázható.



12. ábra Konvolúciós algoritmus roofline diagram 128 csatorna (f) és 1024 csatorna (l) esetén

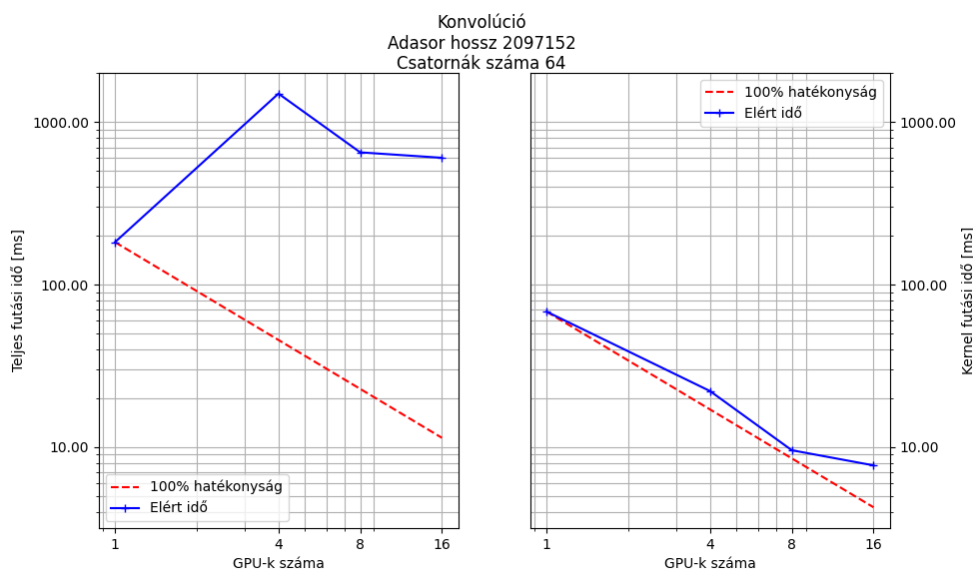
A csatornák tömbösített felosztásával az alábbi ábrán látható módon hat a párhuzamosság fokának növelése az algoritmusra. A mérési eredmény közel 2 millió elemet tartalmazó bemeneti vektorra vonatkozik 128 csatornával, ami a gyakorlatban nagyjából 15 perc hosszú EEG mérési adatnak feleltethető meg. A diagramon látható módon a grafikus kártyák számának emelésével az egy kártyára jutó kernel futási ideje közel arányosan csökken a kártyák számával. A kernel számítási intenzitása azonban alacsony, így a teljes futási idő jelentős részét a kommunikációs idő határozza meg, ennek megfelelően grafikus kártyák növelésével a futási idő azonos.



13. ábra Konvolúciós algoritmus futási idejének változása több GPU hozzáadásával

A végső szétosztás mellett megvizsgáltam a csatornák mentén történő szétosztás lehetőségeit is, ami az előbbinél határozottan rosszabb algoritmust eredményezett. Az ábrán látható, hogy még a csatornák számának csökkentése mellett az egy kártyára jutó kernel teljesítménye fokozatosan romlik. Ennek oka az, hogy ezzel a módszerrel olyan

grid jön létre a GPU-n belül, aminek szélessége a teljes jel hossza, magassága viszont a csatornák kis száma miatt alacsony. Ily módon a felosztás nem használja ki a kártya által biztosított lehetőségeket. Emellett a kommunikáció során kevesebb, nagyméretű üzenet kell az MPI implementációnak továbbítani, ami tovább növeli a kommunikációs időt.



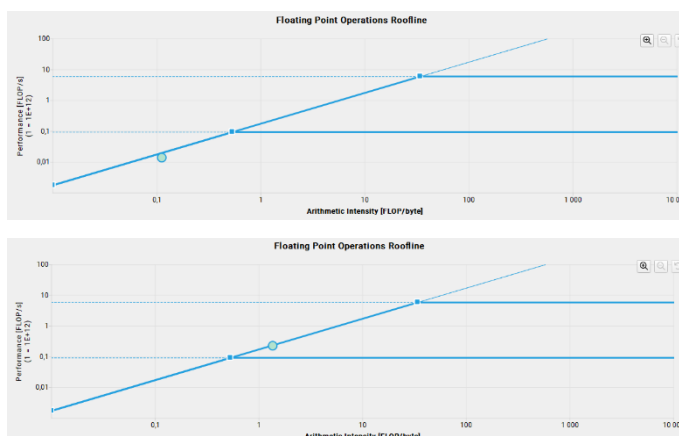
14. ábra Konvolúciós algoritmus futási idejének változása csatornák mentén történő szétosztással

Fontos megjegyezni, hogy az imént megfigyelt kommunikációs késleltetés minden algoritmus esetén azonos. A jelen esetben hozzávetőlegesen egy másodperc, ami egy ezredmásodperc alatt lefutó kernel mellett jelentős időbeli késleltetést okoz. Miután a jelfeldolgozó algoritmusok a gyakorlatban egy pipeline részét képezik, akkor egy közös szétosztási rendszer alkalmazásával több kernel egymás után futtatható, így a kommunikációból adódó késleltetés elhanyagolható. Példának okáért egy komplex jelfeldolgozó pipeline több szűrési és spektrumszámítási lépéssel, valamint Független Komponens Elemzés algoritmussal az imént vizsgált problémaméretre több száz másodpercig fut, így amellet egy adatszétosztás és begyűjtési műveletből adódó késleltetés nem számottevő.

4.2. Short-Time Fourier transzformáció

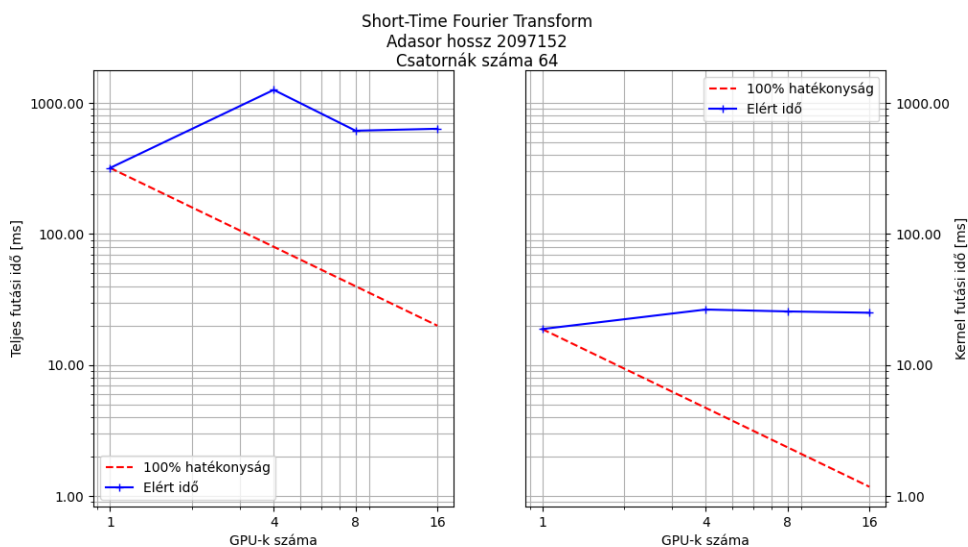
Miután a cuFFT könyvtár nem teszi lehetővé FFT függvény indítását kernelen belül, a probléma különvált egy GPU-n belüli szétosztásra és egy Fourier-transzformációra. Mindkét művelet végző kernel jelentős mértékű memóriát mozgat

minimális számítással. Az alábbi roofline diagramon a szétoztás és a transzformáció eredményei láthatók, mindkét kernelt a memória elérési sebessége korlátozza.



15. ábra STFT algoritmus kerneleinek roofline-diagramja: szegmensek szétoztása (f), FFT algoritmus végrehajtása (l)

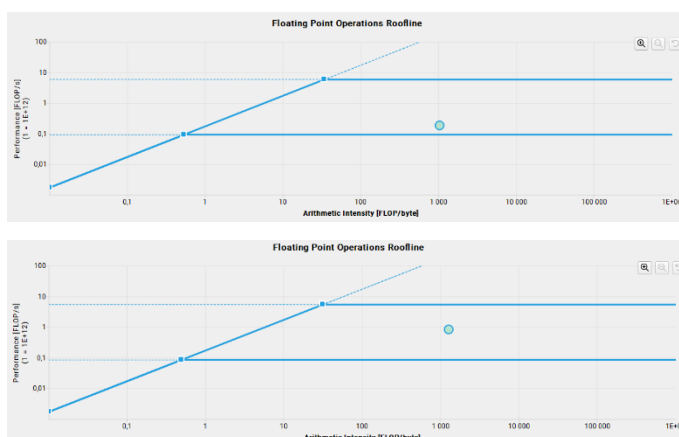
Az alacsony aritmetikai intenzitás és az FFT hatékony megvalósítása következtében a kommunikációs idő teljes mértékben dominálja az algoritmus összteljesítményét a vizsgált bemeneti adathossz tartományon. Az elvégzett mérések szerint a kommunikációs időhöz képest mind az elosztási és a transzformációs kernel futási ideje elenyésző, így a vizsgált bemeneti adatmennyiség nem képes több GPU számítási kapacitásának lefedésére. Az alábbi ábrán látható módon habár nagy mennyiségű adat mozgatása történik, az egy GPU-ra jutó kernel futási ideje nem változik, a GPU nem telített, így a futási idő sem változik. Ennek megfelelően az algoritmus alkalmas jelentősen nagyobb méretű adatsorok feldolgozására is.



16. ábra STFT algoritmus futási idejének változása több GPU hozzáadásával

4.3. Wavelet transzformáció

A wavelet transzformáció jelen megvalósítása nagyban hasonlít a konvolúciós algoritmushoz, azonban a kernelen belül szerepel egy újabb iterációs lépés. Ennek következtében az algoritmus aritmetikai intenzitása és teljesítménye azonos tesztadatok esetén jelentősen magasabb a konvolúció elemzésekor kapott értékeknél. Az alábbi roofline diagramokon megfigyelhető egy és 1024 csatorna esetén, hogy csatornák hozzáadásával a teljesítmény növekszik. A kernel számítási kapacitás által korlátozott, így vélhetően annak futási ideje mellett a kommunikációs idő kevésbé lesz jelentős, így az algoritmus jól skálázható multi-GPU rendszeren.



17. ábra Wavelet transzformáció roofline diagramja: egy csatorna (f), 128 csatorna (l)

A wavelet transzformáció esetén a szétosztás csatornák mentén zajlik. Az alábbi összehasonlító ábrán látható módon az egy kártyára jutó kernel futási ideje mellett az algoritmus teljes futási ideje is csökken. A párhuzamosság ily módon történő skálázhatóságát a feldolgozott csatornák száma is befolyásolja, mivel ettől is jelentős mértékben függ a kernel futási ideje. Az alábbi táblázatban 65536 hosszú adatsorra különböző számú csatornával végrehajtott wavelet transzformáció futási eredményei lettek összefoglalva.

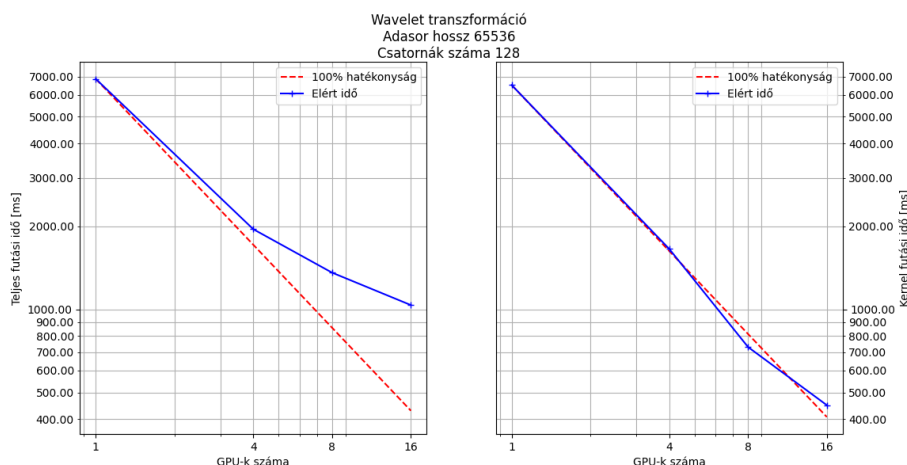
2. táblázat Wavelet kernel futási idők ezredmásodpercben

GPU-k száma	Csatornák száma		
	32	64	128
1	1645.76	3238.08	6514.74
4	464.24	731.037	1657.02
8	295.79	449.58	730.30
16	280.88	296.77	449.33

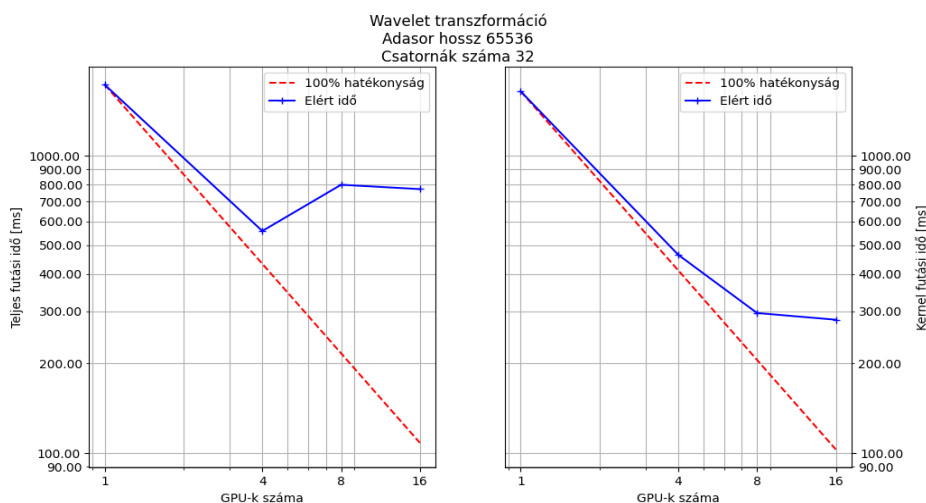
3. táblázat Teljes wavelet transzformáció futási idők ezredmásodpercben

GPU-k száma	Csatornák száma		
	32	64	128
1	1732.17	3410.19	6864.00
4	558.25	904.15	1954.36
8	798.57	966.46	1359.50
16	772.18	824.50	1036.99

Szemléltető célból alább láthatók az eredmények grafikusán is ábrázolva. Megfigyelhető, hogy a kevesebb csatorna esetén már nem a transzformációs kernel futási ideje a domináns tényező, így a kommunikációs idő miatt jelentősen csökken a párhuzamosság mentén történő skálázás lehetősége. A kernel futási idejének növekedését az alternatív konvolúciós algoritmushoz hasonlóan az okozza, hogy a grid dimenziói miatt az algoritmus nem tudja kihasználni a GPU teljesítményét.



18. ábra Wavelet transzformáció futási idejének változása GPU-k hozzáadásával 128 csatorna esetén



19. ábra Wavelet transzformáció futási idejének változása GPU-k hozzáadásával 32 csatorna esetén

5. Összegzés

Dolgozatomban EEG adatok előfeldolgozási lépéseiben alkalmazott legfontosabb algoritmusok (konvolúció, short-time Fourier transzformáció, wavelet transzformáció) párhuzamos megvalósításával foglalkoztam. A cél olyan nagyfokú párhuzamos GPU implementációk kifejlesztése volt, amik szuperszámítógépeken, több GPU kártya egyidejű felhasználásával is futtathatók és ezáltal további sebességnövekedést eredményeznek.

A dolgozatban ismertetett mérési eredmények alapján megállapítható, hogy a bemutatott algoritmusok közül a konvolúció és az azon alapuló wavelet transzformáció hatékonyan alkalmazható multi-GPU architektúrákon. Az STFT algoritmus skálázhatósága nem megfelelő, ami a cuFFT konstans végrehajtási idejének következménye. Ennek megfelelően a tesztelés során alkalmazott probléma méretekre nem indokolt több GPU kártya alkalmazása. A wavelet transzformáció eredményei demonstrálják, hogy az egy GPU-ra készült algoritmusok esetén alkalmazott latency-hiding – a memóriamásolási idő „elrejtése” a teljes futási időben nagy mennyiségű számítás elvégzésével – hatékonyan alkalmazható elosztott memóriájú rendszereken a hálózati kommunikáció esetén is.

Az alkalmazott algoritmusok demonstrálják a multi-GPU tervezési elvek alkalmazását, azonban számos továbbfejlesztési lehetőség felmerül velük kapcsolatban. A folyamatok közti kommunikáció jelentős része kiváltható a szuperszámítógép architektúra fájlrendszerének segítségével. Ennek megfelelően a nagysebességű átmeneti tárban elhelyezett mérési adatokat tartalmazó fájlt egy fő folyamat helyett minden node egyik folyamata olvassa be egyből a GPU memóriájába, majd onnan osztja szét a node többi kártyájának.

Az MPI korábban részletesen leírt limitációi jelentősen megnehezítik annak használatát nagy méretű számítási feladatok esetén, így komoly módosítások és egyénileg készített kollektív kommunikációt helyettesítő megoldások nélkül nem alkalmas az EEG mérések során keletkezett adat hatékony feldolgozására. Az NVIDIA több, közvetlenül GPU kernelből hívható kommunikációs lehetőséget biztosít ennek a problémának a kezelésére, például a korábban említett NVSHMEM és NCCL kommunikációs függvénykönyvtárakat, amikkel például a központi adatelosztásból fakadó szegmens átlapolás egy globális memóriacímter olvasással helyettesíthető. Ennek alkalmazhatósága további vizsgálatokat és algoritmustervezést igényel.

Felhasznált irodalom

- [1] D. A. Reed and J. Dongarra, 'Exascale computing and big data', *Commun ACM*, vol. 58, no. 7, pp. 56–68, 2015.
- [2] P. Kogge and J. Shalf, 'Exascale computing trends: Adjusting to the "new normal" for computer architecture', *Comput Sci Eng*, vol. 15, no. 6, pp. 16–26, 2013.
- [3] K. O'Brien, L. Di Tucci, G. Durelli, and M. Blott, 'Towards exascale computing with heterogeneous architectures', in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017, pp. 398–403.
- [4] 'EuroHPC Joint Undertaking'. Accessed: Apr. 07, 2024. [Online]. Available: https://eurohpc-ju.europa.eu/index_en/
- [5] 'TOP500'. [Online]. Available: <https://www.top500.org/>
- [6] J. Nickolls and W. J. Dally, 'The GPU computing era', *IEEE micro*, vol. 30, no. 2, pp. 56–69, 2010.
- [7] S. Cook, 'CUDA programming: a developer's guide to parallel computing with GPUs', Newnes, 2012, pp. 37–52.
- [8] R. Farber, 'CUDA application design and development', Elsevier, 2011, p. 191.
- [9] S. Cook, 'CUDA programming: a developer's guide to parallel computing with GPUs', Newnes, 2012, pp. 69–97.
- [10] S. Cook, 'CUDA programming: a developer's guide to parallel computing with GPUs', Newnes, 2012, pp. 21–36.
- [11] S. Cook, 'CUDA programming: a developer's guide to parallel computing with GPUs', Newnes, 2012, pp. 107–150.
- [12] K. Z. Ibrahim, S. Williams, and L. Oliker, 'Performance analysis of GPU programming models using the roofline scaling trajectories', in *International Symposium on Benchmarking, Measuring and Optimization*, 2019, pp. 3–19.
- [13] C. Yang, T. Kurth, and S. Williams, 'Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system', *Concurr Comput*, vol. 32, no. 20, p. e5547, 2020.
- [14] R. Farber, 'CUDA application design and development', Elsevier, 2011, pp. 36–37.
- [15] 'NVIDIA Tensor Cores'. [Online]. Available: <https://www.nvidia.com/en-us/data-center/tensor-cores/>
- [16] 'NVIDIA Ampere Architecture'. [Online]. Available: <https://www.nvidia.com/en-us/data-center/ampere-architecture/>
- [17] 'CUDA Programming guide - Warp matrix functions'. Accessed: Apr. 07, 2024. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-matrix-functions>
- [18] 'NVIDIA H100 Tensor Core GPU Architecture'. [Online]. Available: <https://resources.nvidia.com/en-us-tensor-core>

- [19] ‘CUDA Programming guide, Thread Block Clusters’. Accessed: Apr. 07, 2024. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-block-clusters>
- [20] ‘NVIDIA OpenSHMEM Library (NVSHMEM) Documentation’. [Online]. Available: <https://docs.nvidia.com/nvshmem/api/index.html>
- [21] ‘NVIDIA Collective Communications Library (NCCL)’. Accessed: Apr. 07, 2024. [Online]. Available: <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/>
- [22] Z. Juhasz and G. Kozmann, ‘A GPU-based soft real-time system for simultaneous EEG processing and visualization’, *Scalable Computing: Practice and Experience*, vol. 17, no. 2, pp. 61–78, 2016.
- [23] A. Delorme and S. Makeig, ‘EEGLAB: an open source toolbox for analysis of single-trial EEG dynamics including independent component analysis’, *J Neurosci Methods*, vol. 134, no. 1, pp. 9–21, 2004.
- [24] L. Cohen, ‘Time-frequency analysis’, vol. 778, Prentice Hall PTR Englewood Cliffs, 1995, pp. 44–52.
- [25] J. O. Smith, *Spectral Audio Signal Processing*. Julius Smith, 2008.
- [26] ‘Komondor’. Accessed: Apr. 10, 2024. [Online]. Available: <https://hpc.kifu.hu/hu/komondor>
- [27] D. B. Kirk and W. W. Hwu, ‘Chapter 7 - Parallel patterns: convolution: An introduction to stencil computation’, in *Programming Massively Parallel Processors (Third Edition)*, Third Edition., D. B. Kirk and W. W. Hwu, Eds., Morgan Kaufmann, 2017, pp. 149–174. doi: <https://doi.org/10.1016/B978-0-12-811986-0.00007-8>.
- [28] L. Cohen, ‘Time-frequency analysis’, vol. 778, Prentice Hall PTR Englewood Cliffs, 1995, pp. 93–112.
- [29] R. G. Lyons, ‘Understanding Digital Signal Processing’, Prentice Hall PTR, 2004, pp. 74–82.
- [30] ‘cuFFT Documentation’. Accessed: Apr. 13, 2024. [Online]. Available: <https://docs.nvidia.com/cuda/cufft/index.html>
- [31] A. Graps, ‘An introduction to wavelets’, *IEEE computational science and engineering*, vol. 2, no. 2, pp. 50–61, 1995.
- [32] L. Chun-Lin, ‘A tutorial of the wavelet transform’, *NTUEE, Taiwan*, vol. 21, no. 22, p. 2, 2010.
- [33] H. Li, Z. Chen, R. Gupta, and M. Xie, ‘Non-intrusively Avoiding Scaling Problems in and out of MPI Collectives’, in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 415–424. doi: [10.1109/IPDPSW.2018.00076](https://doi.org/10.1109/IPDPSW.2018.00076).