Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Számítástudományi és Információelméleti Tanszék

Megosztott cache heterogén multiprocesszoros rendszerekben

Tudományos Diákköri Konferencia

Készítette
Jani Lázár

Konzulens
Dr. Mann Zoltán

2014. október 22.

Tartalomjegyzék

Kivonat

A processzorgyártók a nagyobb számítási teljesítmény elérése érdekében a CPU magok mellé GPU magokat kezdenek integrálni, mert a korábban alkalmazott technikák (órajel frekvencia növelése, több magos processzorok) fizikai akadályok miatt már csak korlátozott mértékben alkalmasak a sebesség fokozására. Azokat a processzorokat, amelyek különböző típusú processzormagokból épülnek fel, heterogén processzoroknak nevezik. A heterogén processzorok még nem terjedtek el széles körben, de a jövőben ez várhatóan változni fog. A memóriák sávszélessége és késleltetése nem fejlődik olyan mértékben, mint a processzorok teljesítménye, ezért a gyorsítótár (*cache*) egyre fontosabb a maximális teljesítmény eléréséhez, a korábban alkalmazott gyorsítótár menedzselési módszerek azonban heterogén processzorok esetén nem alkalmazhatóak az eltérő működésű magok miatt.

Munkám célja egy heterogén processzorban alkalmazott közös gyorsítótár működésének optimalizálása volt. A MacSim szimulátor és különböző tesztprogramok felhasználásával megvizsgáltam a CPU és GPU magok eltérő viselkedését a memória-alrendszer szempontjából. A szimulációs eredmények azt mutatták, hogy a CPU magok sokkal érzékenyebbek a gyorsítótár méretére, mint a GPU magok, azonban az utóbbiak mégis nagyobb részt foglalnak el a közös gyorsítótárból.

A szakirodalom is foglalkozik a heterogén processzorok megosztott gyorsítótárjának menedzselésével, azonban az eddigi publikációk nem vették figyelembe a tesztprogramok adatméreteinek hatását a gyorsítótár igényre. A dolgozatban bemutatott új algoritmus a közös gyorsítótárat particionálja a CPU és GPU magok között az egyes magokon végrehajtott alkalmazások igényeinek és az adatméretek függvényében.

Abstract

The processor manufacturers are integrating GPU cores besides the CPU cores to further improve the computational performance of the processors, because earlier techniques (e.g. increasing the clock frequency or core count) can not improve the performance efficiently because of physical limitations. These processors, which contain GPU cores and CPU cores as well, are called heterogeneous processors. The heterogeneous processors are not the mainstream type of processor today, but it will change in the future. To utilize the capabilities of the new processors, the cache becomes more important, because the latency and the bandwidth of the memories does not improve as much as the performance of the processors. The cache management methods used in homogeneous multiprocessor systems can not be adopted in heterogeneous system because of the different behavior of the CPU and GPU cores.

The purpose of my work was to optimize the operation of the shared cache in heterogeneous processors. With the use of the MacSim simulator and benchmark programs, I have examined the different behavior of the CPU and GPU cores from the memory-hierarchy perspective. The results showed, that the CPU cores are more sensitive to the size of the cache, but the GPU cores occupy a larger part of the shared cache.

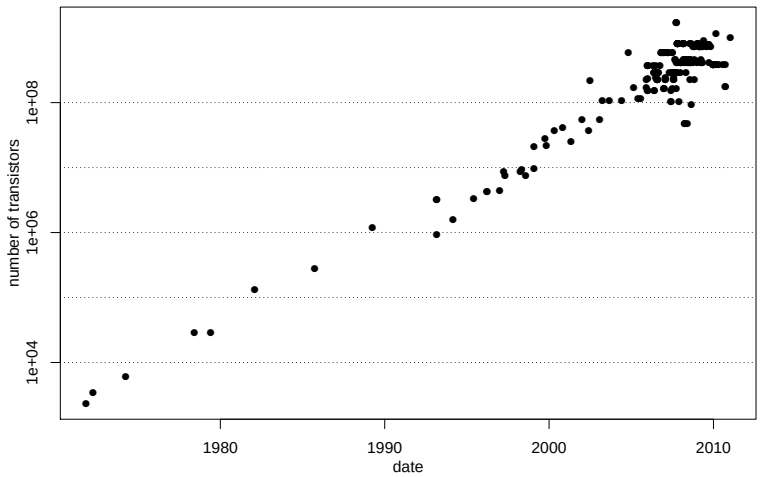
Publications about shared cache management of heterogeneous processors can be found in the literature, but none of these articles have taken into consideration the effect of different input size of the benchmarks to the cache requirements yet. The algorithm introduced in this paper can partition the shared cache between the CPU and GPU cores according to the behavior and the input size of the applications.

1. fejezet

Technológiai háttér

1.1. Moore törvénye, és a következmények

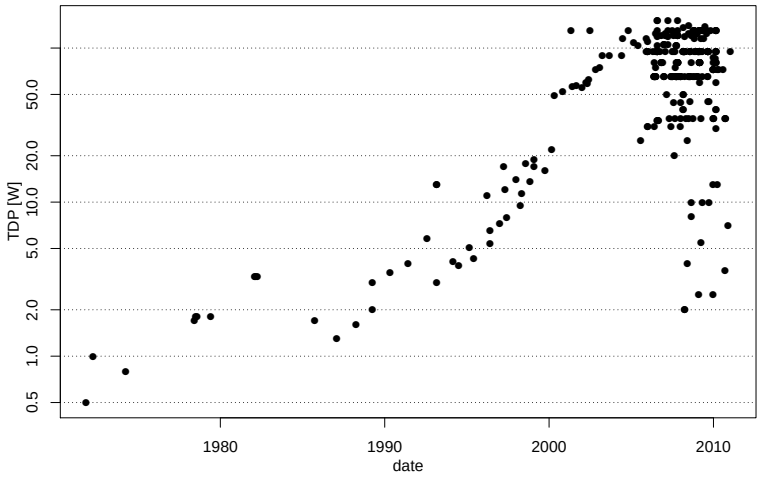
Gordon Moore 1965-ben megjelent tanulmányában megállapította, hogy a félvezető gyártástechnológia gazdaságosabbá válása miatt 1975-ig az egy lapkára integrálható tranzisztorok száma két évente duplázódni fog [1]. Ez a kijelentés azóta irányadóvá vált a félvezetőipar számára, így Moore „törvénye" önbeteljesítő jóslat lett, és napjainkig érvényes (1.1. ábra).



1.1. ábra. *Processzor tranzisztorszámának változása [2]*

A gyártósorok fejlődése kisebb méretű tranzisztorok előállítását tette lehetővé, ami kedvező hatással volt a fogyasztásra és a kapcsolási időre. A Dennard-féle skálázódás szerint a MOSFET (*metal oxide silicon field effect transistor*) fizikai méreteinek és feszültségeinek csökkentése egy arányossági tényezővel (κ), valamint az adalékolás növelése a fogyasztást $1/\kappa^2$-tel csökkenti, így a disszipáció sűrűség változatlan marad (a tranzisztor területe is $1/\kappa^2$-tel változik) [3].

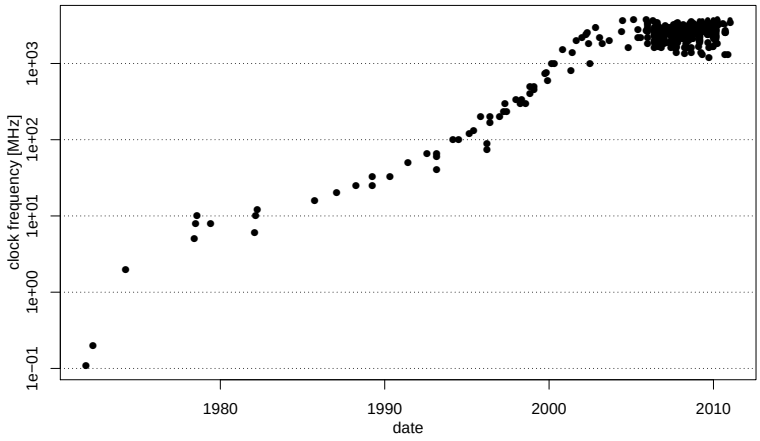
Ez a skálázódás az utóbbi évtizedben veszélybe került a méretcsökkenéssel párhuzamosan felerősödő parazita hatások miatt, aminek következtében a processzorok számítási teljesítményének további növelésére a korábban használt módszerek alkalmatlanná váltak. A nem kívánatos effektusok csökkentéséhez és a kisebb fizikai méret eléréséhez számos új technológiai megoldást alkalmaznak a félvezetőgyártók.



1.2. ábra. *Processzorok TDP-jének változása [2]*

Az 1.2. ábra a processzorok TDP-jének (*thermal design power*) alakulását mutatja be. Az elmúlt 10 évben a maximális fogyasztás nem nőtt, ami annak a következménye, hogy a disszipáció sűrűség elérte a gazdaságos (lég-)hűtés határát.

A processzorok esetén is alkalmazott CMOS (*complementary metal oxide semiconductor*) technológia esetén egy digitális áramkör fogyasztása a dinamikus és a statikus fogyasztásból tevődik össze. Az előbbi egyenesen arányos az átkapcsolások számával, ennek megfelelően az órajel frekvenciájával is. A fogyasztási korlát (*power wall*) következtében az órajel frekvenciája is stagnál (*frequency wall*) az utóbbi évtizedben [4](1.3. ábra).



1.3. ábra. *Processzorok órajel frekvenciájának változása [2]*

Miközben Moore törvénye még mindig érvényes, és az egy lapkára integrált tranzisztorok száma továbbra is növekszik, a Dennard-féle skálázódás végével a disszipáció sűrűség is egyre nagyobb. A disszipációs korlát miatt nem üzemelhet egy időben minden tranzisztor maximális frekvencián, az ilyen módon kihasználatlan tranzisztorok (*dim/dark silicon*) száma exponenciálisan növekszik az újabb gyártástechnológiák bevezetésével [5], relatív költségük viszont egyre kisebb a fogyasztáshoz képest[6].

1.2. Processzorok számítási teljesítménye

A processzorok sebessége egy program futtatásával mérhető, a számítási teljesítményét a program végrehajtásához szükséges idő (t_{CPU}), vagy egy utasítás végrehajtásához szükséges órajelek száma (*cycle per instruction*, CPI) jellemzi. A futásidő a végrehajtáshoz szükséges órajelek számának ($\#cycle$) és az órajel frekvenciájának (f_{clk}) hányadosa (1.1. egyenlet).

$$t_{CPU} = \frac{\#cycle}{f_{clk}} \tag{1.1}$$

A CPI a program végrehajtásához szükséges órajelek számának és a program utasításainak számának hányadosa (1.2. egyenlet).

$$CPI = \frac{\#cycle}{\#instruction} \tag{1.2}$$

A második egyenletet $\#cycle$-re rendezve, és azt behelyettesítve az első egyenletbe adódik az 1.3. összefüggés [7].

$$t_{CPU} = CPI \cdot \#instruction \cdot \frac{1}{f_{clk}} \tag{1.3}$$

A processzor sebességét a 2000-es évek elejéig az órajel frekvenciájának növelésével javították, amit az egyre gyorsabb kapcsolásra képes tranzisztorok, és architektúrális megoldások (például pipeline) tettek lehetővé. Az előző fejezetben leírt disszipációs korlát azonban meggátolja a frekvencia további növelését, amit az ITRS (*International Technology Roadmap for Semiconductors*) előrejelzésekben történt változások is alátámasztanak. 2001 előtti előrejelzés szerint az órajel frekvenciája évenként 41%-kal növekedett volna, azonban a 2011-es előrejelzésben már csak évi 4%-os növekedés található [8]. A gyártók emiatt a frekvencia korlát miatt az egy utasítás végrehajtásához szükséges órajelek számának (CPI) csökkentésével növelik a processzorok számítási teljesítményét különböző szintű párhuzamosítással.

Az utasítás szintű párhuzamosítást (ILP, *instruction level parallelism*) a pipeline bevezetésével kezdték használni. Mivel az egymást követő utasítások függhetnek egymás eredményeitől, ezért ez a megoldás nem mindig hatékony. A pipeline kihasználtságának növelése érdekében a gyártók elágazás becslést (*branch prediction*), sorrenden kívüli (OoO, *out of order*) utasítás végrehajtást illetve regiszter átnevezést (*register renaming*) is alkalmaznak. Az ILP-vel párhuzamosítható utasítások száma ezekkel a technikákkal korlátozott, a fenti funkciókat ellátó modulok további bővítése az elérhető sebességnövekedéshez képest most már aránytalanul nagy költségű (fogyasztás és terület) hardvert eredményezne [7]. Mivel egyszerre csak egy utasítás hajtódik végre, ami egy eredményt ad, ezért a Flynn-féle csoportosításban ezek a processzorok a SISD (*single instruction single data*) csoportba

sorolhatóak (1.1. táblázat).

A gyártók az ILP mellett adat szintű párhuzamosítást (DLP, *data level parallelism*) is alkalmaznak. DLP esetén ugyanaz a művelet több adaton is végrehajtható. Az x86 utasításkészlet esetén ez új utasítások (MMX, SSE, SSE2, SSE3, SSSE3, SSSE4.1, AVX) bevezetésével járt együtt, amelyek operandusai vektorok. A Flynn-féle csoportosításban ezek az új utasításkészletek az SIMD (*single instruction multiple data*) kategóriába esnek (1.1. táblázat).

Általános célú processzorokon futtatott algoritmusok nem mindegyike vektorizálható, így a gyártók a DLP mellett szál szintű párhuzamosítást (TLP, *thread level parallelism*) is elkezdtek alkalmazni. Ennek egyik módja a több szálúsítás (*multithreading*), amely esetén a processzor végrehajtóegysége megosztott erőforrás több szál között. Egy időben csak egy szál hajtódik végre, azonban gyorsan át lehet váltani egy másik szálra [7], ha a végrehajtott szál valamilyen erőforrásra várakozik (például memória olvasás). A másik módszer esetén egy rendszerben szorosan csatolnak egymáshoz több processzort, amelyek közös címtartományt használnak (SMP, *symmetric multiprocessors*). A multiprocesszorok általában egy lapkára integrálva 2-16 processzort (ebben a környezetben processzormagot) tartalmaznak. A multiprocesszorok egy időben több utasítást hajtanak végre több adaton, így a MIMD (*multiple instruction multiple data*) csoportba tartoznak.

1.1. táblázat. *Flynn-féle csoportosítás [7]*

	Adatfolyam	
Utasításfolyam	SISD	SIMD
	MISD	MIMD

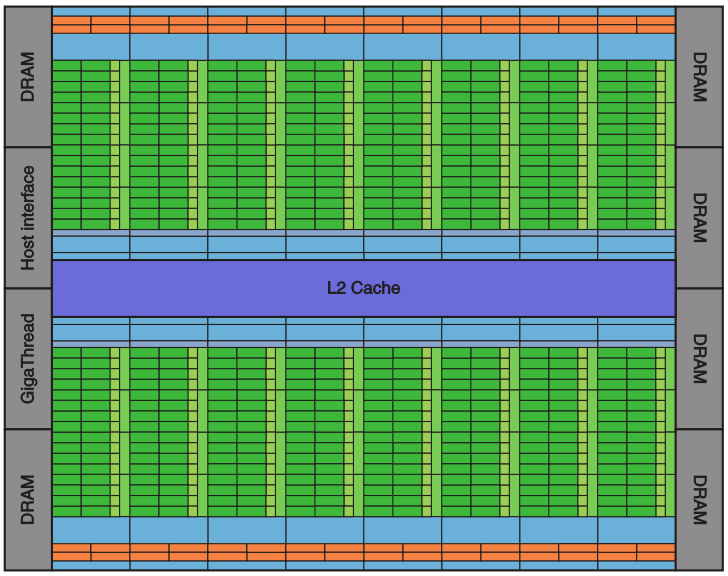
Az 1.1. fejezetben leírt fogyasztási korlát, és az emiatt kihasználatlan tranzisztorok miatt napjainkban az energiahatékonyság fontosabb tényezővé vált, mint a lapkaméret [4]. A több magos processzorok teljesítményét is a fogyasztás korlátozza, ami miatt egy időben nem működhet minden processzormag maximális frekvencián. A nagy x86 processzorgyártók (AMD, Intel) olyan megoldásokat alkalmaznak a többmagos processzorjaikban, amelyek lehetővé teszik bizonyos magok órajel frekvenciájának emelését (amennyiben nincs mindegyik teljesen kihasználva, és belefér a fogyasztási keretbe), ezek a technikák azonban nem növelik meg a processzor összesített számítási teljesítményét.

A heterogén processzorok a „megszokott” felépítésű processzormagok mellett olyan masszívan párhuzamos működésre optimalizált processzormagokat tartalmaznak, amelyeket eddig GPU-kban (*graphics processing unit*) alkalmaztak (például ilyen az Intel Sandy Bridge, Intel Ivy Bridge, Intel Haswell, AMD Llano, AMD Trinity és az AMD Kaveri architektúra). A modern GPU magok általános számítási feladatokra is alkalmasak OpenCL (*Open Computing Language*) vagy CUDA (*Compute Unified Device Architecture*) keretrendszerek felhasználásával, és egységnyi területre és fogyasztásra vetítve nagyobb számítási teljesítménnyel rendelkeznek, mint a CPU magok (részletesebben ld. 1.3. fejezet).

1.3. GPU általános programozása

Az utóbbi évtizedben a GPU-k rohamosan növekedő számítási kapacitásának kihasználására egyre nagyobb igény mutatkozott. Eleinte erre csak nagyon korlátozott lehetőség adódott, mivel grafikus műveletekre kellett konvertálni a feladatot, de a CUDA keretrendszer megjelenésével az NVIDIA általánosan programozhatóvá tette a GPU-kat. Az OpenCL nyelv a CUDA-hoz hasonló programozhatóságot kínál, és több gyártó is támogatja [7].

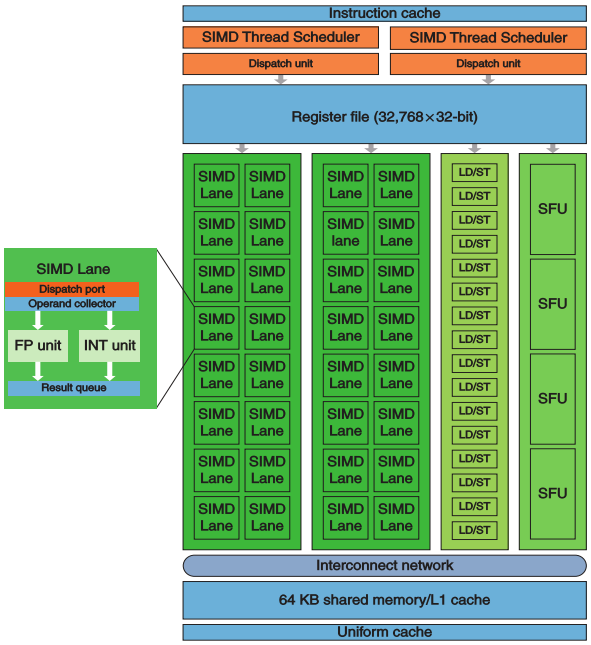
A CUDA nyelvű programok két részre bonthatóak, a CPU-n (*host*) futó kódra, ami inicializálja és meghívja a GPU-n (*device*) végrehajtott *kernelt*. Miután a GPU befejezte a működést, a CPU-nak vissza kell másolnia az eredményt a saját memóriájába. A kernelből a fordító 32 elemű SIMD utasításfolyamot (*warp*) állít elő, aminek egy eleme egy *thread*. A warp-ok csoportokba, úgynevezett *thread block*-okba tömörülnek, a thread block-ok összességét *grid*-nek nevezik. Kernelhíváskor a C-ben megszokott paraméterlistán kívül meg kell adni a grid és a thread block dimenzióit.



1.4. ábra. *NVIDIA GF110 GPU blokkvázlat [7]*

A GPU programozása során a megfelelő teljesítmény eléréséhez elkerülhetetlen a GPU belső felépítésének ismerete. Az 1.4. ábra az NVIDIA GF110 kódnevű GPU blokkvázlatát mutatja be. A GPU 16 darab többszálúsított SIMD processzort (NVIDIA terminológiája szerint *streaming multiprocessor*, SM), 768 KB L2 cache-t, 6 csatornás memóriavezérlőt és egy GigaThread nevű globális ütemezőt tartalmaz. A legutóbbi felelős azért, hogy a thread blockokat az egyes SM-ekhez rendelje. Az 1.5. ábrán egy SM blokkvázlata látható.

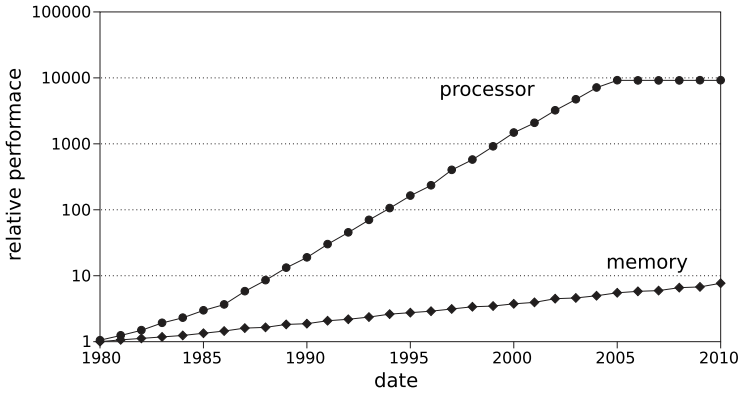
Egy streaming multiprocessor privát utasítás cache-sel és két ütemezővel rendelkezik. Ezek 4 órajelenként ütemeznek egy-egy warpot a végrehajtóegységekre, összesen legfeljebb 48 warpot (1536 szálat) tudnak kezelni. A regiszterfájl mérete 128 KB, vagyis 32768 darab 32 bites adatot képes eltárolni, azonban egy thread maximum 63-at használhat. A SM összesen 64 KB L1 gyorsítótárral illetve megosztott memóriával (*shared memory*) rendelkezik, amit 16/48 vagy 48/16 arányban lehet felosztani.



1.5. ábra. *NVIDIA Fermi streaming multiprocessor [7]*

1.4. Memória alrendszer

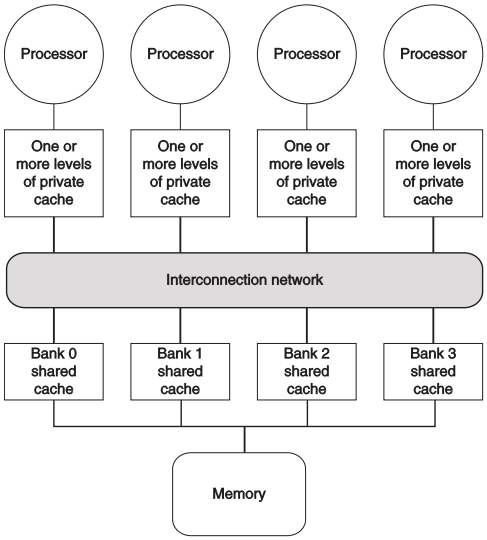
A programozók igénye szerint a memória nagy kapacitású és gyors lenne, azonban gazdaságosan ezek a tulajdonsok nem érhetőek el egyszerre. Az operatív memória alapjául szolgáló DRAM (*dynamic random access memory*) sebessége az évek során lényegesen lassabban fejlődött a processzorok sebességéhez képest [9] (*memory wall*) (1.6. ábra).



1.6. ábra. *Memória és processzor egy szálas sebességének relatív viszonya [7]*

A lokalitás elve szerint a legtöbb program nem egyenletes eloszlással fér hozzá az utasításokhoz, illetve az adatokhoz. Ezt kihasználva a kis méretű és gyors valamint a lassabb, de nagy méretű memóriákból olyan hierarchia tervezhető, amely ötvözi a gyors hozzáférést és a nagy kapacitást (1.7. ábra) [7].

Lokalitás lehet időbeli (*temporal locality*), vagyis egy nemrég beolvasott elem valószínűleg újra kelleni fog, illetve térbeli (*spatial locality*), ami azt jelenti, hogy a beolvasott elem



1.7. ábra. *Egy multiprocesszoros rendszer memória alrendszere [7]*

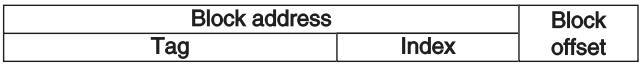
címéhez közel eső elemekre is szükség lehet [4].

A kis méretű gyors hozzáférésű memória (gyorsítótár, *cache*) az operatív memória tartalmának egy részét tárolja a programozó számára transzparens módon. A gyorsítótár blokkosítva (*cache line*) tárolja az elemeket, kihasználva az adatok és utasítások térbeli lokalitását. Az egy line-ba tartozó elemeket egyszerre olvassa a memóriából és írja vissza, ami megnöveli a memória effektív sávszélességét (*burst*-ös írás/olvasás).

A szervezés szempontjából három fajta cache-t lehet megkülönböztetni.

- Direkt leképezés (*direct mapped*) esetén egy elem memóriacíme egyértelműen meghatározza, hogy a gyorsítótárban melyik blokkban és azon belül milyen pozícióban kell lennie.
- Teljesen asszociatív (*fully associative*) gyorsítótárban a cache line-oknak nincs fix helyük, így egy új line bármelyik pozícióba kerülhet.
- Csoport asszociatív (*set associative*) megoldás az előző kettő ötvözete. Bizonyos számú blokk egy csoportba van rendezve. A memóriacím meghatározza, hogy egy elem melyik csoportban, illetve blokkon belül hol van, azonban a csoporton belül akármelyik blokkban lehet.

Egy memóriacím az 1.8. ábrának megfelelően három részre bontható. Az offset határozza meg a cache line-on belüli pozíciót. Az index határozza meg a csoportot (teljesen asszociatív esetben a csoportok száma 1, direkt leképezés esetén a blokkok számával egyezik meg), a cím maradék része a tag, amely a gyorsítótárban azonosítja a cache line-t.



1.8. ábra. *Egy memóriacím felbontása tag, index és offset mezőkre [7]*

A tag mező méretét az 1.4. összefüggés határozza meg a cache paraméterek és a memóriacím méretének függvényében.

$$tag\ bits = memory\ address\ bits - \log_2(number\ of\ sets) - \log_2(block\ size) \tag{1.4}$$

A gyorsítótár méretét az 1.5. egyenlet szerint lehet kiszámítani.

$$cache\ size = associativity \cdot number\ of\ sets \cdot line\ size \tag{1.5}$$

A processzor működés közben folyamatosan írja és olvassa a memóriát (utasítások és operandusok beolvasása, eredmények kiírása). Memória olvasás esetén a processzor először megnézi a gyorsítótárat, hogy az tartalmazza-e a keresett elemet (tag mező azonosítja a gyorsítótárban a blokkot). Találat (*cache hit*) esetén onnan olvassa be az adatot (vagy utasítást), ha nincs találat (*cache miss*), akkor az alacsonyabb szintű memóriához fordul (ez lehet egy másik gyorsítótár, vagy az operatív memória is). Az utóbbit négy csoportra lehet bontani [7].

- *Compulsory miss* esetén a processzor először fér hozzá az adott blokkhoz, amit így be kell tölteni a gyorsítótárba.
- Ha a cache nem tudja eltárolni a program végrehajtása közben az összes blokkot, *capacity miss* fordul elő.
- A nem teljesen asszociatív gyorsítótárban egy blokk csak bizonyos pozíciókban lehet. Ha egy új blokk olyan helyre kerül, ahol már van egy másik blokk, *conflict miss* történik.
- Multiprocesszoros rendszerben szükség lehet a koherencia megtartására. Ha egy közös gyorsítótárból két processzor is privát cache-be másolja ugyanazt, majd az egyik visszaírja az eredménnyel együtt a közös gyorsítótárba, a másik processzornak újra be kell töltenie azt a privát cache-be, hogy az adatok konzisztensek maradjanak. Ezt a jelenséget *coherence miss*-nek nevezik.

Memóriaírás során, különösen multiprocesszoros rendszerek esetén, fontos az adatkonzisztencia megőrzése. Két elterjedt memória-írási stratégia létezik, amelyek kezelik a memória tartalom, és a cache-ben található másolat koherenciájának problémáját.

- *write-through* esetén az adatot gyorsítótárban és a memóriában is beírja egy művelettel.
- *write-back* stratégia csak a cache-ben módosítja az adatot, a memóriába csak akkor íródik bele, ha a blokk kikerül a gyorsítótárból.

A teljesen asszociatív és csoport asszociatív gyorsítótárakban a memóriából behozott új blokknak nincs meghatározott helye, így döntést kell hozni, hogy melyik pozícióba kerüljön,

illetve melyik blokk legyen lecserélve. A legáltalánosabban használt módszer a legrégebben használt elemet cseréli le a gyorsítótárban (*least recently used*, LRU) [7].

A gyorsítótár hatékonysága növelhető, ha egy időben akár több cache miss-szel is képes foglalkozni. A blokkoló gyorsítótár az első cache miss esetén nem fogad több memória hozzáférést, amíg a szükséges adat nem töltődik be a memóriából. A nem blokkoló gyorsítótár több cache miss-t is képes egyszerre kezelni. Az ezekhez szükséges információkat az MSHR-ben (*miss status hold register*) tárolja.

1.5. Memória alrendszer optimalizálása

Az 1.4. fejezetben bemutatott korlát miatt a memória alrendszer nagymértékben befolyásolja a processzorok teljesítményét. Már 1994-ben felismerték, hogy gyorsítótárban csak előtöltő (*prefetcher*) használatával küszöbölhető ki a (*compulsory miss*) és érhető el 100%-os találati arány [9].

A gyorsítótárak optimalizálása a cache paramétereinek meghatározását jelenti (blokkok száma és mérete, asszociativitás, szintek száma) úgy, hogy a kapott konfiguráció egy vagy több mutató (fogyasztás, találatok száma, elérési idő) szerint legkedvezőbb tulajdonsággal rendelkezzen [10]. Ez történhet „tervezési" időben (*offline*) [11], aminek előnye az, hogy az optimum meghatározására sok idő és több információ áll rendelkezésre. Másik lehetőség a futásidőben (*online*) történő konfigurálás [12], ami ugyan többlet terheléssel jár, viszont így a gyorsítótár alkalmazkodni tud az alkalmazáshoz.

A memória korlát hatását (*scratch pad memory*, SPM) alkalmazásával is lehet csökkenteni [9]. Az SPM memória az operatív memória címtartományában található kis méretű, de gyors elérésű memóriát jelent. Mivel az SPM-ben nem kell eltárolni a cím tag mezőjét, ezért a cache-hez képest kisebb költséggel lehet ugyanakkora tároló kapacitást beépíteni. Megfelelő adat particionálással nagymértékű gyorsulást lehet elérni [13].

Az egy chipen megvalósított multiprocesszoros rendszerek (CMP, *chip multiprocessor*) esetén a memória alrendszer megtervezése összetettebb feladattá válik. A processzormagok között biztosítani kell a gyors adatmegosztást, a memória hozzáféréseknek nem szabad egymást blokkolniuk. A megfelelő sebességet általában egy megosztott gyorsítótár biztosítja. Ebben az esetben figyelni kell, hogy a párhuzamosan futó gyenge lokalitású szálak ne szorítsák ki az erős lokalitással rendelkező programokat. Ezt a problémát akár szoftveresen is ki lehet küszöbölni [14].

A heterogén processzorok esetén a megosztott gyorsítótárnak a különböző processzormagok eltérő jellegű memória hozzáférését is tolerálnia kell. A késleltetésre optimalizált, nagy egy szálas teljesítménnyel rendelkező processzormagok (CPU magok) érzékenyebbek a gyorsítótárak méretére, mint a sok párhuzamos szálat futtató (GPU) magok, mivel a sok párhuzamos szál megfelelő ütemezés mellett képes elfedni a memória késleltetését. A GPU magok a sok párhuzamos szál miatt lényegesen több memória olvasást és írást képesek kezdeményezni, így kiszorítják a CPU magok adatait. Heterogén processzorokban alkalmazott megosztott gyorsítótár optimalizálásával tudomásom szerint két publikáció foglalkozik. A [15]-ben bemutatott módszer processzormagok monitorozása alapján dönt az alkalmazott

cache menezdsment stratégiáról. A [16]-ben leírt algoritmus a processzor monitorozásával állapítja meg, hogy egy GPU-s alkalmazás mennyire tolerálja a nagy memóriakésleltetést. Az eredmény függvényében a GPU-s alkalmazás megkerülheti a megosztott LLC-t és közvetlen a memóriából olvashat be adatokat. A 2. fejezetben részletesebben visszatérek ezekre a megoldásokra.

2. fejezet

Gyorsítótár heterogén processzorokban

Az 1.2. fejezet bemutatta a CPU és GPU magok közös lapkára való integrálásának motivációját. Heterogén processzorok már megjelentek kereskedelmi forgalomban (ld. 1.2. fejezet), azonban az utolsó szintű gyorsítótár (*last level cache*, LLC) jellemzően csak a CPU magok között van megosztva. A CPU magokkal ellentétben a GPU magok némileg képesek tolerálni a memóriakésleltetést, azonban bizonyos alkalmazások igenyelhetik a nagy méretű gyorsítótárat, ezért ezekben az esetekben gyorsabb működést eredményezne, ha a CPU és a GPU magok egy közös LLC-n osztoznának.

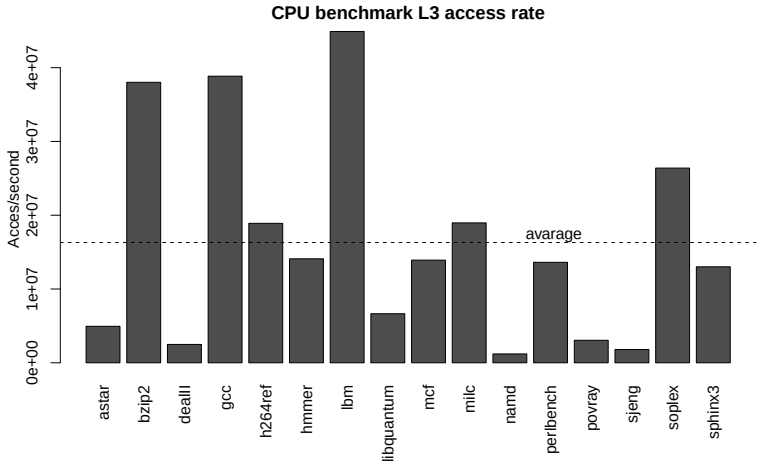
2.1. A közös gyorsítótár megvalósításának akadálya

A közös gyorsítótár megvalósítása több megoldandó problémát is felvet a különböző típusú magok eltérő viselkedése miatt. A CPU magok kis számú szálat kezelnek egyszerre, így a memória késleltetését nem tudják elfedni a végrehajtott szál cserélésével, ennek következtében érzékenyebbek a cache méretére, sebességére. Ezzel szemben a GPU magok akár ezernél is több szálat képesek ütemezni úgy, hogy a végrehajtóegységek kihasználása maximális legyen. Ha egy szál memória hozzáférés miatt várakozni kényszerül, akkor egy másik szál kerül végrehajtásra, ezért ezek jobban tolerálják a memória késleltetését, amit a hagyományos cache menedzsment módszerek nem vesznek figyelembe.

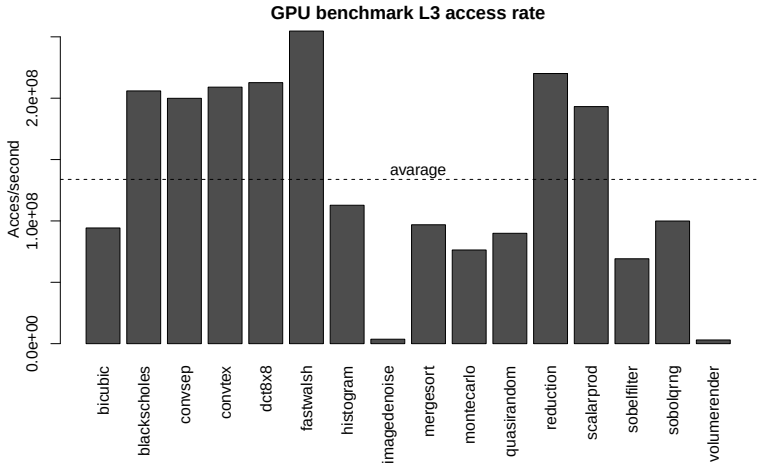
A heterogén processzorokban a közös gyorsítótár esetén az eddig általánosan alkalmazott LRU blokk cserélési stratégia önmagában nem megfelelő. A GPU magok akár egy nagyságrenddel több cache hozzáférést tudnak kezdeményezni egységnyi idő alatt (2.1 és 2.2. ábra), ezért kiszoríthatják a CPU-s alkalmazások adatait a megosztott cache-ből.

2.2. Korábbi megoldások

Két publikációról van tudomásom, amelyek új stratégiát mutatnak be a heterogén rendszerek megosztott gyorsítótárjának menedzseléséhez. Ezek ismert blokk cserélési stratégiák és cache menedzsment módszerek módosításával oldották meg az eltérő típusú processzormagok illesztését a közös gyorsítótárhoz. Az egyik ilyen ismert módszer a cache parti-



2.1. ábra. *CPU benchmarkok L3 cache hozzáférése (hit+miss) egységnyi idő alatt*

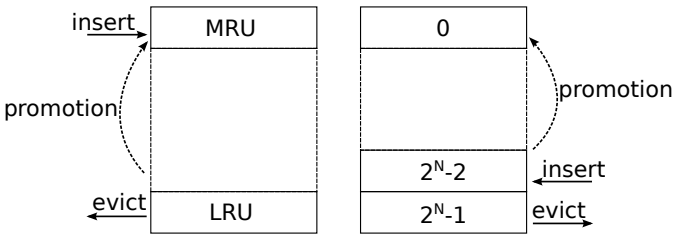


2.2. ábra. *GPU benchmarkok L3 cache hozzáférése (hit+miss) egységnyi idő alatt*

cionálás [17] (*utility based cache partitioning*, UCP). Az UCP futásidőben dinamikusan partícionálja a cache-t az alkalmazások között azoknak az igényei szerint. Ez a módszer heterogén processzorokban a GPU magokat favorizálja, a nagy számú cache hozzáférés (és cache hit) miatt.

Az LRU blokk cserélési stratégia helyett a (*re-reference interval prediction*, RRIP) [18] stratégiát használják, ami kedvezőbb tulajdonságokkal rendelkezik nem optimális memóriaolvasási mintázatok esetén. Az RRIP stratégia lényege, hogy egy beérkező új blokk alacsony élettartammal rendelkezzen a cache-ben, és csak találat esetén növekedjen az élettartama (LRU esetén egy beérkező új blokk maximális élettartammal rendelkezik). LRU terminológiában az RRIP stratégia esetén egy beérkező új blokk nem MRU helyre, hanem közel LRU helyre kerül, és találat esetén kerül MRU helyre (2.3. ábra). RRIP-ből létezik statikus (SRRIP) és dinamikus (DRRIP) változat.

Az SRRIP újonnan behozott blokkot mindig azonos pozícióba helyezi. Ha minden blokkhoz egy n bites tároló tartozik, akkor egy új blokk $2^n - 2$-es értékkel (*re-reference prediction*



2.3. ábra. *Az LRU és az SRRIP blokk cserélési stratégiák működése*

value, RRPV) kerül be a gyorsítótárba, és csak találat esetén csökken a számláló értéke (*promotion policy*).

A DRRIP egy kevert módszer, amely az SRRIP, és a *bimodal RRIP* stratégiákat alkalmazza. A BRRIP egy újonnnan behozott blokkot alacsony valószínűséggel 2^n-2 RRPV-vel, nagyobb valószínűséggel 2^n-1 RRPV-vel helyezi el a cache-ben. A DRRIP *set dueling* módszerrel két különböző cache seten SRRIP és BRRIP stratégiákat alkalmazva összehasonlítja a találati arányokat, és a cache többi részén a kedvezőbbet alkalmazza [18].

A [15]-ben bemutatott TAP (*thread-level parallelism-aware cache management policy*) stratégia két GPU mag monitorozásából (*core sampling*), és cache blokk élettartam normalizálásból tevődik össze. A TAP a megfigyelt GPU magokon eltérő cache menedzsment stratégiát alkalmazva eldönti, hogy a kettőből melyik a kedvezőbb, és a többi GPU magon is jobb stratégiát használja. A TAP-UCP módszer az UCP stratégián alapul, de figyelembe veszi a GPU alkalmazások érzékenységét a cache-re. Ha egy GPU alkalmazás nem érzékeny a cache-re, akkor csak kis szeletet kap a megosztott cache-ből. Az élettartam normalizálás lehetővé teszi, hogy a GPU alkalmazás ne foglaljon el túl sok cache területet.

A TAP-RRIP a DRRIP stratégia módosítása. A processzormag monitorozásából kiderül a GPU alkalmazás jellege, és ha az érzéketlen a gyorsítótárra, akkor BRRIP stratégiát alkalmaz a GPU alkalmazások esetén. További változás, hogy a GPU alkalmazások cache blokkja nem léptetődik elő, így kisebb lesz az élettartama a gyorsítótárban.

TAP-pel átlagosan 0% és 20% közötti gyorsulást értek el az LRU stratégiához viszonyítva különböző összetételő workloadok esetén [15], azonban ez a módszer több esetben nem nyújt megoldást. A processzormagok monitorozása feltételezi, hogy a GPU magok működése szimmetrikus, tehát mindegyik mag ugyanannyi LLC hozzáférést kezdeményez és ugyannyi utasítást hajt végre. Tapasztalataim szerint akár 30% eltérés is lehet a GPU magok által végrehajtott utasítások számában. A cache blokk élettartam normalizálást végző algoritmus hardverigénye sem elhanyagolható a tárolókhoz viszonyítva, ugyanis tartalmaz egy osztást is, ami áramkörben megvalósítva sok erőforrást igényelhet.

A [16]-ban bemutatott módszerben (*heterogeneous last level cache management*, HeLM) a GPU magok elegendően nagy számú szál vagy cache érzéketlen alkalmazás esetén nem fordulnak a megosztott gyorsítótárhoz, hanem közvetlen a memóriából olvassák ki az adatot (*cache bypass*), ezáltal a CPU magokra nagyobb méretű gyorsítótár jut. A HeLM független a cache blokk cserélési stratégiától, így alkalmazható vele LRU, vagy RRIP is. A HeLM-RRIP stratégiával átlagosan 10% és 20% közötti gyorsulást értek el az LRU-hoz viszonyítva [16].

2.3. CPU-GPU szimulátorok

Jelenleg nem kaphatóak olyan heterogén processzorok, amely a CPU és GPU magok között megosztott gyorsítótárat tartalmaznak, ezért szimulátort kellett használnom az LLC viselkedésének modellezéséhez. Négy olyan szimulátorról van tudomásom, amely képes CPU és GPU magokat megosztott memóriával és gyorsítótárral szimulálni.

A Multi2Sim [19] szimulátor x86-os tesztprogramokat módosítás nélkül képes futtatni, azonban a GPU magokon futó OpenCL alkalmazásokat a Multi2Sim runtime library-jéhez kell linkelni.

A MacSim [20] egy trace alapú szimulátor. A szimulálni kívánt alkalmazásokból trace fájlt kell előállítani, amely az utasításfolyamot írja le, majd a szimulátor ezt beolvasva modellezi a processzor működését.

A heterogén processzorok megosztott gyorsítótárjával kapcsolatos két publikáció az előző két szimulátort használta, ezért én is ezeket próbáltam ki.

Először a Multi2Sim-et teszteltem, ami elég jól dokumentált, a weboldalon található fórumban pedig további információkat lehet szerezni a szimulátorral kapcsolatban más felhasználóktól. Sajnos a tesztelés idején stabil verzió nem támogatta a CPU és GPU magok megosztott memória alrendszerét, ezért fejlesztői verziót voltam kénytelen használni, ami azonban tesztelésre alkalmatlan volt a gyakori hibák miatt.

A MacSim egy publikus levelező listával rendelkezik, ahol nagyon sok hasznos információ megtalálható a telepítéssel, a működéssel valamint a szoftverhibákkal kapcsolatban. Sajnos ennél is előfordul rendellenes működés, de kevesebb, mint a Multi2Sim esetén. A munkám során ezt a szimulátort használtam, az ezzel kapcsolatos tapasztalataim a függelék F.1. fejezetében találhatóak.

Általánosságban elmondható ezekről a szimulátorokról, hogy körülményesen telepíthetőek és használhatóak, a kis méretű célközönség miatt nagyon kiforratlanok, gyakran fordult elő nem várt vagy hibás működés, és bizonyos esetekben nem a dokumentációnak megfelelő viselkedést mutattak.

Az fentieken kívül két szimulátorról van tudomásom, amely támogatja a heterogén processzorok közös memória alrendszerét, de ezeket nem volt alkalmam kipróbálni. A gem5-gpu [21] két különálló szimulátor (a gem5 és a GPGPU-Sim) összekapcsolásából született. Ez a szimulátor támogatja a másolás nélküli adatmegosztást is a CPU és GPU magok között. A fusionSim [22] szimulátorban is elérhető ez a funkció, azonban csak 1 CPU magot támogat.

2.4. A CPU-s és GPU-s programok viselkedése

A munkám során CPU-s és GPU-s tesztprogramokat is használtam a megfelelő típusú magok viselkdésének vizsgálatához. Mivel a MacSim szimulátor használata mellett döntöttem, ezért a [15]-ben található programokat használtam. A CPU-s benchmarkok a SPEC CPU 2006 csomagból valóak, a tracefájlok megtalálhatóak a MacSim weboldalán [20], a GPU-s programok a CUDA 4.1 SDK-ból származnak. A felhasznált tesztprogramok listája a 2.1. táblázatban található.

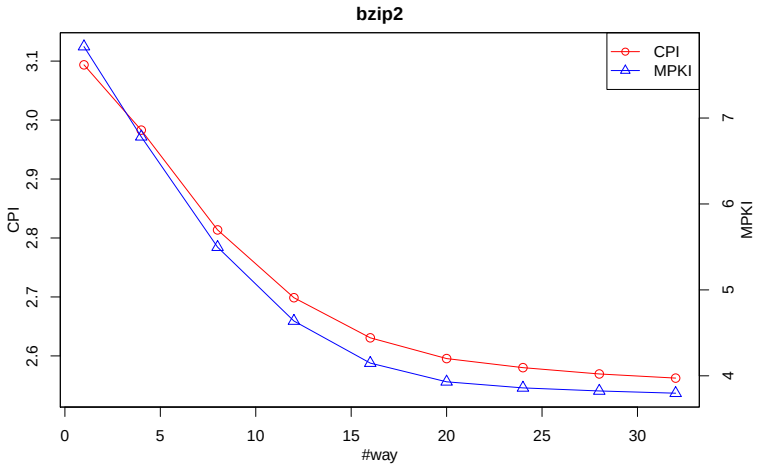
2.1. táblázat. *A felhasznált tesztprogramok*

GPU	bicubic, blackscholes, convsep, convtex, dct8x8, fastwalsh, histogram, imagedenoise mergesort, montecarlo, quasirandom, reduction, scalarprod, sobelfilter, sobolqrng, volumerender
CPU	astar, bzip2, dealII, gcc h264ref, hmmer, lbm, libquantum, mcf, milc, namd, perlbench, povray, sjeng, soplex, sphinx3d

Az egyes processzormagok és a benchmarkok viselkedésének vizsgálata során a cache érzékenységre (a futásidő és cache méret kapcsolatára) voltam kíváncsi. A dolgozatban ezt kapcsolatot a program karakterisztikájának is nevezem. A szimulációk során az utolsó szintű gyorsítótár asszociativitásának változtatásával csökkentettem a cache méretét, a többi paramétert változatlanul hagytam (1.5. egyenlet). Az eredmények kiértékelésekor arra jutottam, hogy CPU-s benchmarkok az alábbi két kategóriába sorolhatóak cache érzékenység szempontjából.

1. Gyorsítótár méretének csökkenésére az ezer utasításra vetített cache missek száma (*misses per kilo instruction*, MPKI) és a CPI is nő. Egy ilyen benchmarkhoz tartozó CPI és MPKI értékek láthatóak a 2.4. ábrán.

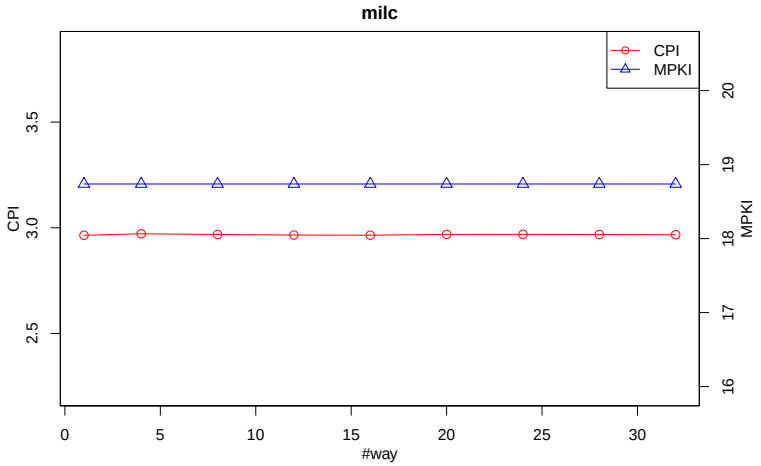
2. Ebbe a kategóriába azok az alkalmazások tartoznak, amelyek a gyorsítótár méretére érzéketlenek (2.5. ábra). Ezt okozhatja a nagy méretű adathalmaz (*working set*), vagy az adatok többszöri használatának hiánya (stream jellegű alkalmazások).



2.4. ábra. *A bzip2 CPU benchmark különböző cache méret esetén*

A GPU-s alkalmazások között gyorsítótár érzékenység szempontjából három csoport különböztethető meg.

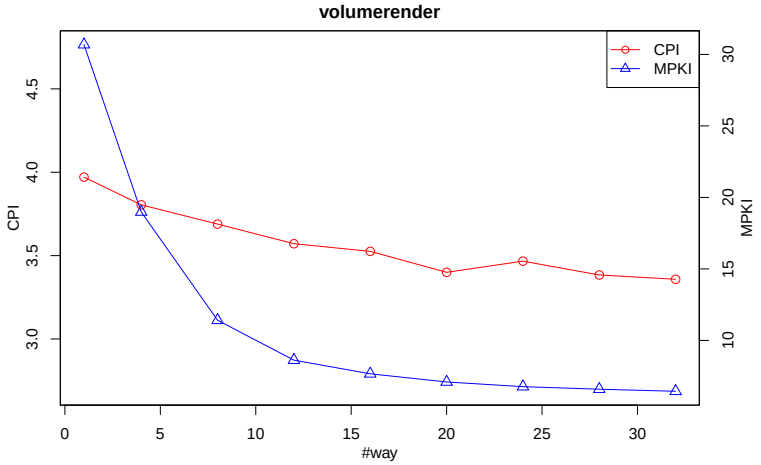
1. Gyorsítótár méretének csökkenésére az MPKI és a CPI is nő. Egy ilyen alkalmazáshoz tartozó CPI és MPKI értékek láthatóak a cache méretének függvényében a 2.6. ábrán.



2.5. ábra. *A milc CPU benchmark különböző cache méret esetén*

2. Cache méretének csökkenésére nő az MPKI, de a CPI nem változik (2.7. ábra). Ezek az alkalmazások tolerálják a kisebb cache-t, mert a sok szál képes elfedni a memóriakésletetést. Ilyen kategória nincs a CPU-s alkalmazások között.

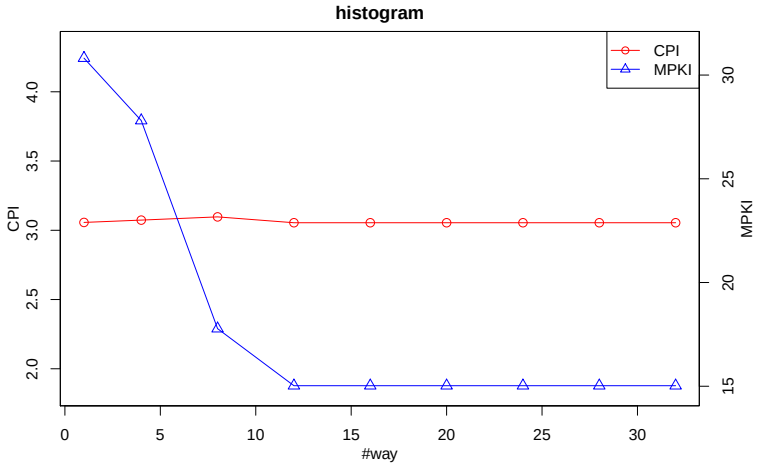
3. Ez a csoport invariáns a gyorsítótár méretére (2.8. ábra). Ezt okozhatja a nagy méretű adathalmaz, vagy az adatok többszöri használatának hiánya (stream jellegű alkalmazások).



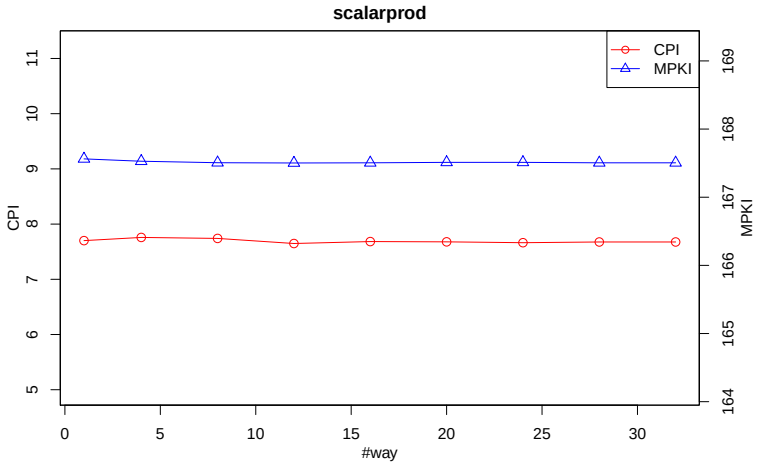
2.6. ábra. *Volumerender GPU benchmark különböző cache méret esetén*

A fenti csoportok nincsenek rögzítve, lehetséges közöttük átjárás. A különböző bemenő adatméret esetén egészen más viselkedést mutathat egy alkalmazás. A 2.9. ábrán látható, hogy a cache méretére érzékeny benchmark kisebb adatméret esetén érzéketlenné válik a cache-re. Ez amiatt lehetséges, mert a bemenő adat olyan kicsi, hogy a legkisebb konfiguráció esetén is belefér a gyorsítótárba. Ezzel a jelenséggel a [15] és a [16] publikáció sem foglalkozott, így ezek új megfigyelésnek tekinthetőek.

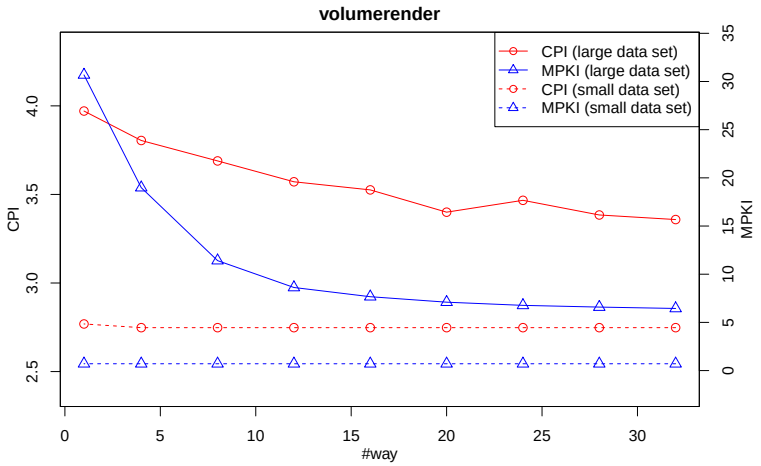
Nem minden alkalmazás mutat ilyen drasztikus változást a kisebb méretű adat esetén. A 2.10. ábrán látható benchmark esetén kisebb adatméret hatására kevesebb lett a cache miss és kisebb lett a CPI, de az alkalmazás cache-érzéketlen jellege nem változott.



2.7. ábra. *Histogram GPU benchmark különböző cache méret esetén*

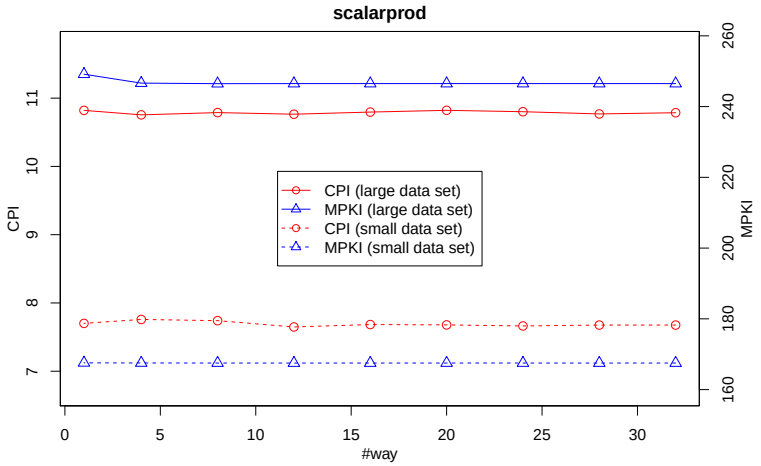


2.8. ábra. *Scalarprod GPU benchmark különböző cache méret esetén*



2.9. ábra. *Volumerender GPU benchmark különböző cache méret esetén*

A CPU és GPU benchmarkok között vannak a cache szempontjából hasonlóan viselkedő típusok (érzékenyek a cache-re, vagy érzéketlenek), azonban bizonyos GPU benchmarkoknak nincs a CPU-sok között megfelelő kategória. A CPU és GPU benchmarkok L3



2.10. ábra. *Scalarprod GPU benchmark különböző input és cache méret esetén*

hozzáféréseinek száma is jelentősen eltérhet egymástól. Ez a különbség főleg a processzormagok eltérő felépítéséből fakad. A GPU magok több száz párhuzamos szálat futtatnak, amik külön-külön memóriaolvasást kezdeményezhetnek, így akár nagyságrendi különbség is lehet az L3 hozzáférések számában a GPU javára.

A cache érzékenység a CPU-s benchmarkoknál sokkal drasztikusabb teljesítményromlást okoz, mint a GPU-s alkalmazások esetén. Például a bzip2 program MPKI-je 106%-al növekedett a LLC méretének csökkentésével, eközben a CPI is 21%-al nőtt. GPU-s programoknál a volumerender az egyik legérzékenyebb a cache méretére, de a 376%-os MPKI növekedés is csak 18%-os CPI növekedést okozott.

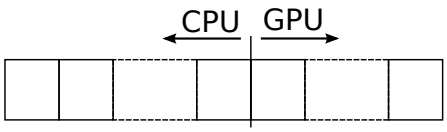
3. fejezet

LLC particionálás

A 2.2. fejezetben bemutatott módszerek a programok futása közben monitorozzák a processzort és a gyorsítótárat, és ez alapján választanak cache menedzsment stratégiát, azonban ezeknek az „online” megoldásoknak a hardverigénye jelentős lehet. A [15] és [16] cikkek csak az extra tárolókat említik költségként, az algoritmusok végrehajtásához szükséges hardver mérete és fogyasztása nem ismert. Ez a fejezet bemutat egy saját fejlesztésű algoritmust (*LLC partitioning for heterogeneous processors*, LP4HP), amely az LLC-t „offline” particionálja, így minimális hardverigénnyel rendelkezik.

3.1. Alapötlet

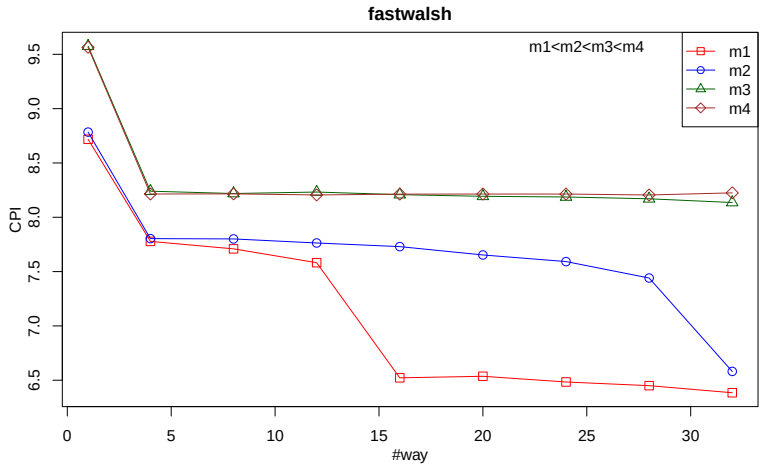
A CPU-s és a GPU-s alkalmazások eltérő hozzáférési sebessége alapján (2.1. és 2.2. ábra) feltételezhető, hogy egy közös, homogén hozzáférésű LLC esetén a GPU-s alkalmazások a gyorsítótár jelentősen nagyobb szeletét foglalják el a CPU-s alkalmazásokhoz képest. Ez abban az esetben, ha a CPU-s alkalmazás cache érzékeny (CPI függ a cache mérettől), számottevően ronthatja a CPU-s program sebességét. Ez fordítva is igaz, tehát ha a CPU futtat egy olyan alkalmazást, amely nagy hozzáférési sebességet és sok cache misst eredményez, úgy egy GPU-s alkalmazás adatait kiszoríthatja az LLC-ből. Ez utóbbi lehetőség feltehetően nem annyira kritikus mint az előbbi, mert a programok viselkedésének vizsgálata során a GPU-s benchmarkok kisebb mértékben reagáltak a cache méret csökkentésére, az MPKI növekedésével nem járt akkora mértékű CPI növekedés, mint a CPU-s alkalmazások esetén (ld. 2.4. fejezet). Összességében tehát a CPU-s programok érzékenyebbek a gyorsítótár méretére, de a GPU-s alkalmazások hajlamosak lehetnek nagyobb szeletet elfoglalni.



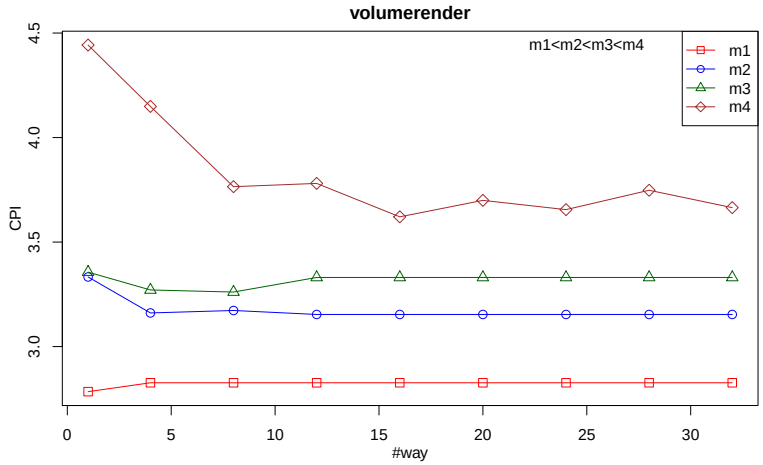
3.1. ábra. *Az LLC particionálása a CPU és GPU magok között*

A fentiek alapján ha az LLC két részre, egy CPU-s és egy GPU-s szeletre lenne osztva (3.1. ábra), akkor gyorsulhatna a cache érzékeny alkalmazás. Ennek a particionálásnak

adaptálódnia kell az eltérő összetételű *workloadokhoz*, ezért nem lehet statikusan meghatározni a CPU-s és a GPU-s cache szeletek méretét. A futásidőben történő cache particionálás optimális eredményt adna, de jelentős hardverigénnyel járhat. Egy algoritmus, amely közvetlen a programok indítása előtt konfigurálná az LLC-t, nem okozna többletköltséget, és a programok viselkedésének ismeretében megfelelően particionálhatná a gyorsítótárat. A 2.4. fejezetben látható, hogy a futtatott programok sebessége a cache méretétől (s) és az inputmérettől (w) függenek ($CPI\,(s, w)$) (ld. 3.2. és 3.3. ábra), így az adatméret és a karakterisztika függvényében megállapítható az optimális cache konfiguráció. Egy „offline" particionáló algoritmus működéséhez szükséges adatbázis fokozatosan bővíthető, ha ugyanazon programokat gyakran kell futtatni. Ez jellemző például a beágyazott processzorok esetén, ahol a rendszer konfigurálása után nem változik a végrehajtandó program, de egy asztali számítógépeken használt alkalmazások halmaza sem túlságosan nagy.



3.2. ábra. *A fastwalsh GPU tesztprogram CPI-je a cache méret függvényében, 4 különböző input mérettel*



3.3. ábra. *A volumerender GPU tesztprogram CPI-je a cache méret függvényében, 4 különböző input mérettel*

Az összes lehetséges bemenő adatmérettel nem lehet lemérni az egyes programok sebességét, ezért egy becslést kell alkalmazni, amely a lemért adatok és a program input-

mérete alapján határozza meg a várható sebességet. Legyen $CPI_A(s) = CPI(s, m_A)$ és $CPI_B(s) = CPI(s, m_B)$ ahol $CPI_A(s)$ az m_A adatmérethez tartozó és $CPI_B(s)$ az m_B adatmérethez tartozó CPI adatsor. Ezek alapján egy köztes, m_X adatmérethez a 3.2. egyenlet alapján becsülhető a $CPI_X(s)$ függvény.

$$\frac{CPI_X(s) - CPI_A(s)}{CPI_B(s) - CPI_A(s)} = \frac{m_X - m_A}{m_B - m_A} \qquad m_B > m_X > m_A \quad (3.1)$$

$$CPI_X(s) = \frac{m_X - m_A}{m_B - m_A} \cdot (CPI_B(s) - CPI_A(s)) + CPI_A(s) \qquad m_B > m_X > m_A \quad (3.2)$$

Ez a becslés az alábbi feltételezéseken alapul:

- rendelkezésre állnak különböző ismert inputmérethez (m) felvett $CPI(s)$ adatsorok, ahol s a cache mérete
- az inputméret és a cache méret meghatározza a futásidőt
- a CPI monoton az inputméret függvényében
- az inputméret vagy a cache méret kis változása a futásidő kis módosulását okozza

A fenti feltételeknek megfelelően az LLC particionálását végző algoritmus (LP4HP) bemenete az adatbázis, amely meghatározott adatméretek mellett tartalmazza a $CPI(s)$ karakterisztikát, az LLC asszociativitása (cache utak száma), valamint a futtatott program adatmérete. Ez alapján az LP4HP meghatározza a megosztott gyorsítótár CPU-s és GPU-s szeleteinek méretét.

3.2. Az algoritmus célfüggvénye

Heterogén processzorok számítási teljesítményének jellemzéséhez nem megfelelő pusztán az 1.2. fejezetben leírt CPI mérőszám (1.2. egyenlet). A CPU és a GPU magok órajelének frekvenciája általában eltérő, ezért a számítási teljesítményük összegzéséhez be kell vezetni az egységnyi idő alatt végrehajtott utasítások számát (*instruction per second*, IPS) az 1.1. egyenlet alapján.

$$IPS = \frac{\#instruction}{t_{CPU}} = \frac{\#instruction}{\#cycle} \cdot f_{clk} \quad (3.3)$$

A CPU és GPU magok összesített teljesítményét a 3.5. egyenlet írja le.

$$IPS = \frac{\#instruction_{CPU}}{t_{CPU}} + \frac{\#instruction_{GPU}}{t_{GPU}} \quad (3.4)$$

$$IPS_{total} = IPS_{CPU} + IPS_{GPU} \quad (3.5)$$

A maximális teljesítmény eléréséhez meg kell keresni az összesített teljesítményt leíró 3.5. egyenlet szélsőértékét (maximumát). Mivel a CPU és a GPU teljesítménye is függhet

a cache mérettől, ezért az algoritmus az $IPS(x, y)$ függvény maximumát keresi meg a 3.7. egyenlet által meghatározott egyenes mentén.

$$IPS_{total}(x, y) = IPS_{CPU}(x) + IPS_{GPU}(y) \tag{3.6}$$

$$x + y = cache\ associativity \qquad x > 0,\ y > 0 \tag{3.7}$$

Az összesített teljesítmény maximalizálása nem mindig célravezető. Amennyiben a CPU-n futó cache érzékeny program sebessége sokkal kisebb, mint a GPU-s programé, a 3.6. és 3.7 egyenletek által meghatározott cache konfiguráció nem lesz kedvező a CPU-s alkalmazásnak. Ezért egy olyan döntési mechanizmust is kipróbáltam, ami a nagyobb cache-sel elérhető gyorsulás (3.8. egyenlet) alapján particionálja a gyorsítótárat. A particionáló algoritmus a maximálisan elérhető CPU-s és GPU-s gyorsulást a 3.9. egyenlet szerint a 3.7. egyenes mentén keresi meg.

$$speedup(s) = \frac{IPS(s)}{min(IPS(s))} \tag{3.8}$$

$$speedup_{total}(x, y) = \sqrt{speedup_{CPU}(x) \cdot speedup_{GPU}(y)} \tag{3.9}$$

A 3.9. egyenlet a CPU-s és GPU-s alkalmazások gyorsulásának mértani közepét határozza meg. Mivel [15] és [16] cikkekben is ezt a metrikát alkalmazták, ezért én is ezt alkalmaztam.

3.3. LLC particionáló algoritmus (LP4HP) működése

A cache particionálását végző algoritmus (LP4HP) nem futásidőben működik, ezért szükség van előre mért adatokra, amelyből meghatározhatja az egyes programok gyorsítótár igényét. Mivel lehetetlen lemérni a programok sebességét az összes lehetséges input mérettel, ezért az algoritmus becslést alkalmaz a várható IPS meghatározásához.

```
w_avg(ipsA,ipsB,mA,mB,mX,assoc) begin
 for(i=1 to assoc-1) do
  ipsX[i] = (mX-mA)/(mB-mA)*(ipsB[i]-ipsA[i])+ipsA[i]
 end
 return ipsX
end
```

3.4. ábra. *Az IPS-t becslő függvény pszeudókódja*

A 3.4. ábrán az ips_x-et becslő w_avg függvény pszeudókódja látható. Az LP4HP a keresett ips_X-et az ismert ips_A és az ips_B IPS adatsorok m_A, m_B és m_X inputméretekkel vett súlyozott átlagából határozza meg (hasonlóan a 3.2. egyenlethez). A 3.1. fejezetben a teljesítmény CPI-vel van kifejezve, de ez jelen esetben csak egy konstanssal való szorzást jelent. A függvény *assoc* változója a LLC útjainak számát (asszociativitását) tárolja, az ips_A és ips_B, valamint a hozzájuk tartozó m_A és m_B inputméreteket egy adatbázisból

választja ki az *estimate_ips* függvény (3.5. ábra) úgy, hogy az m_A és m_B a legközelebb essen a futtatni kívánt program inputméretéhez (m_X).

```
estimate_ips(data_ips,data_m,m_x,assoc) begin
 if(m_x<data_ips[2]) then
  ips_x = w_avg(data_ips[1],data_ips[2],data_m[1],data_m[2],m_x)
 end
 else if(data_m[length(data_m)-1]≤m_x) then
  ips_x = w_avg(data_ips[length(m)-1],data_ips[length(data_m)],
               data_m[length(data_m)-1],data_m[length(data_m)],m_x,assoc)
 end
 else
  for(i=2 to length(data_m)-2) do
   if((data_m[i]≤m_x) & (m_x<data_m[i+1])) then
    ips_x = w_avg(data_ips[i],data_ips[i+1],
                 data_m[i],data_m[i+1],m_x,assoc)
    break
   end
  end
 end
 return ips_x
end
```

3.5. ábra. *A becsléshez használt IPS adatsorok kiválasztása*

Az *estimate_ips* függvény a kiválasztott alkalmazáshoz tartozó IPS adatsorait ($data_{ips}$) és az ezekhez tartozó inputméreteket ($data_m$) kapja meg az adatbázisból, valamint az alkalmazás pillanatnyilag beállított inputméretét (m_X). Ezek alapján a függvény kiválasztja a $data_m$-ből az m_X-hez legközelebb eső két inputméretet és a hozzájuk tartozó két IPS adatsort, majd ezekkel az adatokkal meghívja a w_avg függvényt, ami becslést ad a várható IPS-re. Az adatbázisban található IPS adatsorok, valamint inputméretek számát a $length(data_m)$ adja meg.

A 3.6. ábra az *LP4HP* működését mutatja be. Az algoritmus bemenő paraméterei az IPS adatsorok ($ipsdb_{cpu}$, $ipsdb_{gpu}$), a hozzájuk tartozó input méretek ($inputsizedb_{cpu}$, $inputsizedb_{gpu}$), a futtatni kívánt CPU-s és GPU-s alkalmazás neve ($name_{cpu}$, $name_{gpu}$) inputmérete (m_{cpu}, m_{gpu}), az LLC asszociativitása ($assoc$) valamint egy bool változó ($gmean$), amely a metrikát határozza meg. Az algoritmus az adatbázisból kiválasztja a CPU-s és GPU-s program adatait, és meghívja az *estimate_ips* függvényt, ami a becsült IPS-sel tér vissza. A $gmean$ változó értékének függvényében az LP4HP az összesített teljesítmény ips_{total} értékeket számítja ki a 3.6. egyenlet szerint, vagy kiszámítja a gyorsulást a 3.9. egyenletnek megfelelően, majd meghatározza a kiszámított értékek maximumának helyét, ami a CPU-ra eső cache asszociativitásának felel meg.

```
LP4HP(ipsdb_cpu,ipsdb_gpu,inputsizedb_cpu,inputsizedb_gpu,
      name_cpu,name_gpu,m_cpu,m_gpu,assoc,gmean)
begin
 ips_cpu = estimate_ips(ipsdb_cpu[name_cpu],inputsizedb_cpu[name_cpu],m_cpu,assoc)
 ips_gpu = estimate_ips(ipsdb_gpu[name_gpu],inputsizedb_gpu[name_gpu],m_gpu,assoc)
 if(gmean == false) then
  for(i=1 to assoc-1) do
   ips_total[i] = ips_cpu[i]+ips_gpu[assoc-i]
  end
  for(i=1 to assoc-1) do
   if (ips_total[i] > max) then
    max = ips_total[i]
    partition = i
   end
  end
 end
 else
  speedup_cpu = norm(ips_cpu)
  speedup_gpu = norm(ips_gpu)
  for(i=1 to assoc-1) do
   speedup_total[i] = sqrt(speedup_cpu[i]*speedup_gpu[assoc-i])
  end
  for(i=1 to assoc-1) do
   if (speedup_total[i] > max) then
    max = speedup_total[i]
    partition = i
   end
  end
 end
 return partition
end
```

3.6. ábra. *Az LP4HP pszeudókódja*

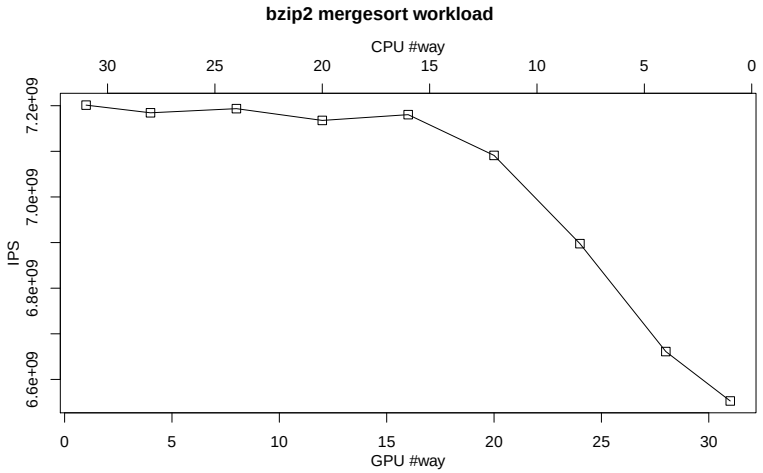
Amennyiben az alkalmazások lehető legnagyobb gyorsítása a cél, akkor a CPU és a GPU becsült IPS értékeit a 3.7. ábrán látható függvény normálja, majd az LP4HP a 3.9. egyenlet szerint kiszámítja a várható gyorsulást, és megkeresi az eredményben a maximális értéket.

```
norm(ips, assoc) begin
 for(i=1 to assoc-1) do
  if (ips[i] < min) then
   min = ips[i]
  end
 end
 for(i=1 to assoc-1) do
  result[i] = ips[i]/min
 end
 return result
end
```

3.7. ábra. *IPS értékek normálása*

Az algoritmus működését illusztrálja a 3.8. és a 3.9. ábra. A 3.8. ábra a bzip2 és a mergesort programokból összeállított workload esetén mutatja be az összesített IPS függését a cache particionálástól.

A 3.9. ábra ugyanazon workload esetén mutatja meg a gyorsulás mértékét a cache parti-



3.8. ábra. *IPS függése a cache particionálástól*

cionálás függvényében. Látható, hogy a bzip2 és a mergesort programok esetén különböző cache konfiguráció adódik az összesített IPS és a gyorsulás maximalizálása alapján.

3.9. ábra. *Gyorsulás függése a cache particionálástól*

Az LP4HP $O(n)$ futásidejű, ahol n a megosztott cache asszociativitása. Az algoritmus tárigényét főleg az adatbázis mérete befolyásolja, ez a cache asszociativitásától és az adatsorok számától is lineárisan függ. Összességében az LP4HP gyors, kicsi a memóriaigénye, így jól használható és nem okoz jelentős többletterhelést.

Az LP4HP működéséből fakadóan mindig lesz GPU-s és CPU-s cache partició, tehát nem fordulhat elő olyan eset, hogy valamelyik alkalmazás kiszorul a gyorsítótárból.

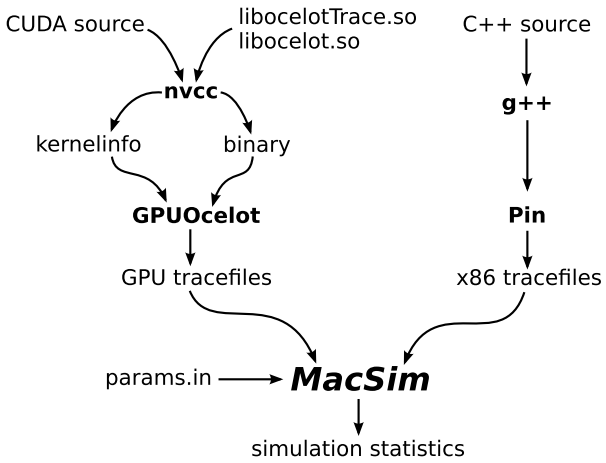
4. fejezet

Implementáció és eredmények

A megosztott gyorsítótár viselkedésének vizsgálatához a MacSim szimulátort használtam (ld. 2.3. fejezet). Ez a szimulátor tracefájl alapú, a „futtatni" kívánt programból tracefájlt kell generálni, amit a processzormodell beolvas. A 3.3. fejezetben pszeudókóddal leírt algoritmus prototípusát R nyelven implementáltam.

Ez a fejezet a tracefájlok generálásával és a szimulációval kapcsolatos munkát, valamint az elért eredményeket mutatja be.

4.1. A szimulációs környezet kialakítása



4.1. ábra. *A szimulációs környezet felépítése*

A MacSim szimulátor tracefájlok alapján modellezi a processzor viselkedését. A tracefájlokat a processzormodell által végrehajtani kívánt programból kell generálni, külső program felhasználásával. A GPU programokból a GPUOcelottal [23], az x86 programokból Pinnel [24] lehet ilyen fájlokat előállítani. A munkám során GPU programok tracefájljait állítottam elő, mert a [15]-ben és a [16]-ban használt SPEC CPU 2006 benchmarkok nem szabadon hozzáférhetőek, viszont a MacSim szimulátor weboldalán elérhető a programok egy része tracefájlok formájában. A MacSimet és a GPUOcelotot kevesen használják, ezért nincsen kiforrott módszer a telepítésre és használatra, a dokumentáció hiányos és kompatibilitási gondok is előfordultak, ezért a telepítéshez szükséges lépéseket a függelék F.1. fejezetében

foglaltam össze. Ezek egy része megtalálható különböző weboldalakon és levelező listákon, de bizonyos problémákat nekem kellett megoldanom.

Telepítést követően az F.1.6. fejezet szerint lehet előállítani tracefájlokat a CUDA SDK programjaiból. A tesztprogramok egy részét módosítanom kellett, hogy állítható legyen az inputméretük. Ez abból állt, hogy új parancssori argumentumokat határoztam meg, amivel az inputméretet lehet beállítani. A programokat újra le kellett fordítanom, de ez után már lehetett állítani az összes használt benchmark input méretét. A GPU-s programból generált trace fájlok kis méretűek, de nagyon sok van belőlük, így egy benchmarkhoz tartozó trace fájlok összesen több gigabájtnyi helyet is elfoglalhatnak. Mivel a MacSim csak egy magot használt, ezért több szimulációt futtattam egy időben. Hagyományos merevlemezek sok kis méretű fájlt nagyon lassan olvasnak, ezért a tracefájlokat SSD-n tároltam.

A MacSimmel végzett szimulációk során egy heterogén processzort modellező konfigurációt használtam (4.1. táblázat). A CPU magoknak két szintű privát, a GPU magoknak egy szintű privát gyorsítótárja van. A CPU magok L2 és a GPU magok L1 gyorsítótárja egy hálózaton (*network on chip*, NoC) keresztül érik el a megosztott közös gyorsítótárat.

A CPU magok 4 utas szuperskalár egységek, out-of-order utasítás ütemezéssel és elágazásbecsléssel. A GPU magok felépítése hasonló az 1.3. fejezetben bemutatott Fermi SM-hez.

4.1. táblázat. *A MacSim beállításai*

CPU	L1I cache 32 KB, 2-way set-associative
	L1D cache 16 KB, 4-way set-associative, line size 64 byte, 3 cycles latency
	L2 cache 256 KB, 8-way set-associative line size 64 byte, 8 cycles latency
	4 cores, 3 GHz
	4-wide superscalar, out-of-order instruction scheduler
	gshare branch predictor
GPU	6 cores, 1,5 GHz
	no branch predictor
	L1I cache 4 KB, 2-way set-associative, line size 64 byte, 2 cycles latency
	L1D cache 16 KB, 4-way set-associative, line size 64 byte
LLC	1 GHz, 30 cycles latency, 8 bank, 1 cycle latency
	8 MB, 32 associative, line size 64 byte
Memory	dual channel, 1,6 GHz

4.2. Tesztprogramok karakterizálása

A telepítést követően elkezdtem ismerkedni a benchmarkok (ld. 2.1. táblázat) működésével, és lefuttattam néhány szimulációt. Egyes GPU-s tesztprogramok több kernelt is tartalmaznak, amelyek egy problémára nyújtanak különböző megoldásokat. A bicubic benchmark-ban nem csak a bicubic, hanem például bilineáris textúraszűrés is megtalálható,

de az imagedenoise-ban is több implementáció található zajszűrésre. Ezeknél a tesztprogramoknál nagy tárigény és futásidő miatt csak egy-egy megoldást használtam (bicubic esetén a bicubic, imagedenoise esetén a *k-nearest neighbour* zajszűrést választottam).

Az előzetes várakozásnak megfelelően (ld. 1.5. fejezet) a CPU-s programok esetén az LLC méretének csökkentése nagyobb mértékben növelte a futási időt, mint GPU-s programok esetén. A legkisebb és legnagyobb beállított LLC méret mellett a CPI és MPKI adatok változásának korrelációja CPU-s benchmarkoknál $0,7$, míg GPU-s programok esetén csak $0,19$ volt.

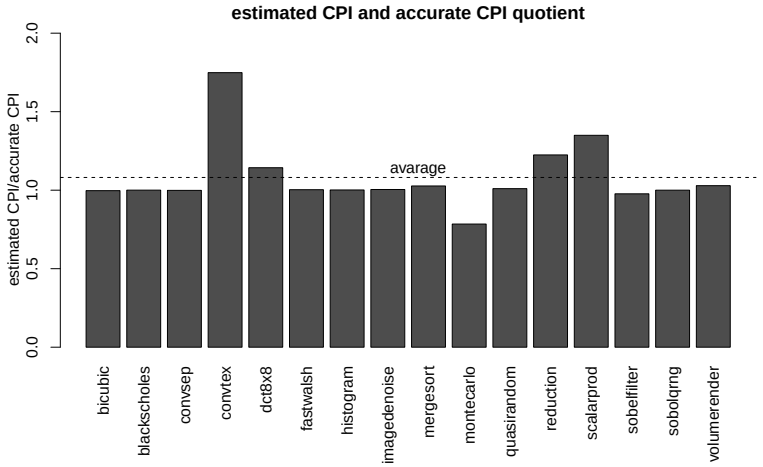
A GPU-s programok viselkedésének vizsgálatakor derült ki, hogy a futási sebesség nem csak az LLC méretétől, hanem az input méretétől is függhet (2.9. és 2.10. ábra). Mivel az input méretének hatása a sebességre legalább akkora, mint a cache méretének hatása, ezért ezt a jelenséget is figyelembe kellett venni cache particionáláskor.

Az input méretének hatását minden GPU-s benchmark esetén megvizsgáltam. Az összes tesztprogramnak beállítottam 4 különböző input méretet (lehetőség szerint egyenletesen elosztva), és így generáltam a tracefájlokat. Ugyan bizonyos benchmarkok esetén csak egy kernelből készültek tracefájlok, összesen így is 71 GB tárhelyet foglaltak el az SSD-n.

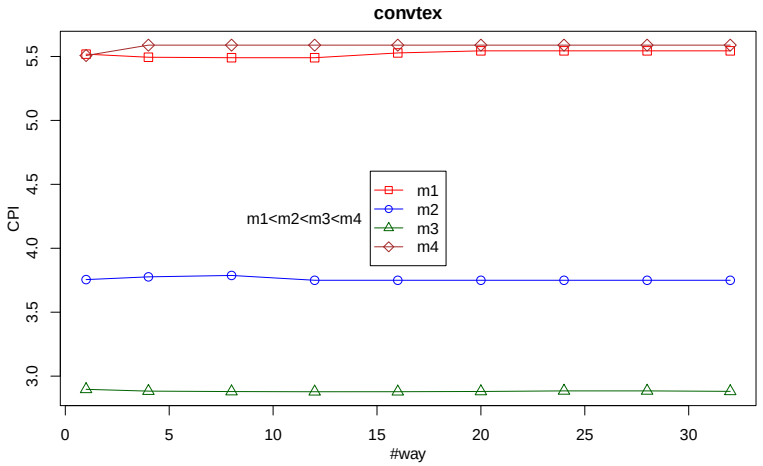
A munkát kicsit hátravetette, hogy a tracefájlok előállítása során beleütköztem az ext fájlrendszer korlátjába. Alapértelmezett partícióbeállítások esetén kevés volt a tárhelyhez az inode-ok száma (túl sok kisméretű fájlt tároltam a partíción). Emiatt kénytelen voltam újra particionálni az SSD-t, és kézzel beállítani az inode-ok számát (ld. F.1.1. fejezet).

A tracefájlok előállítása után a MacSim-mel szimuláltam az összes tesztprogramot különböző LLC méret esetén. A cache méretet az utak számának változtatásával állítottam be 1 és 32 között, 4-es lépésekben (kivéve a 4 és 1 utas konfiguráció esetén). A 16 benchmark 4 eltérő inputmérettel és 9 cache konfigurációval összesen 576 workloadot eredményezett, egy szimuláció pedig átlagosan két óráig futott.

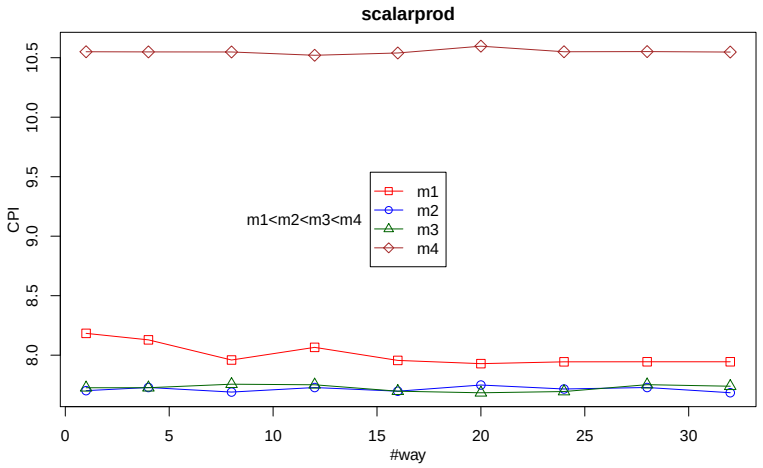
A benchmarkok karakterizálása 4 input mérettel körülbelül egy hétig tartott. Az eredményekből (végrehajtott utasítások száma, futásidő, inputméret) egy adatbázist hoztam létre. Minden tesztprogramhoz előállítottam még egy tracefájlt az eddigiektől eltérő input mérettel, amit majd a heterogén (CPU-s és GPU-s programokat egyaránt tartalmazó) workloadokhoz használtam fel. A particionáló algoritmus az eltárolt eredmények és az új tracefájl input mérete alapján becsüli meg a várható futási sebességet (ld. 3.3. fejezet). A becslés átlagosan 8%-kal tért el a szimuláció alapján számított értéktől, 11 tesztprogram esetén az eltérés kevesebb, mint 3%, tehát a becslés jónak mondható (4.2. ábra).



4.2. ábra. *A becsült CPI és a szimuláció alapján számított CPI hányadosa*



4.3. ábra. *A convtex GPU tesztprogram CPI-je a cache méret függvényében, 4 különböző input mérettel*



4.4. ábra. *A scalarprod GPU tesztprogram CPI-je a cache méret függvényében, 4 különböző input mérettel*

A becsült és a számított CPI értékek nagy eltérését néhány esetben az okozza, hogy

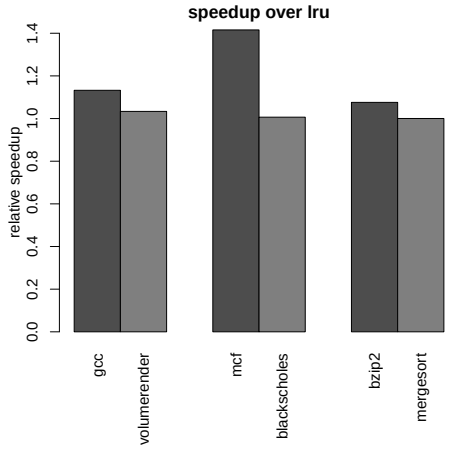
bizonyos benchmarkok esetén a CPI nem monoton változik az inputméret függvényében (4.3. és 4.4. ábra). Ennek oka az lehet, hogy különböző adatméretek esetén eltérhet a párhuzamosan elindított szálak száma, ami hatással van a teljesítményre is.

4.3. Heterogén workloadok szimulációja

A MacSim szimulátorban az LLC-t három paraméter meghatározásával lehet particionálni, de az első szimulációk alkalmával nem volt hatásuk a futásidőre. A forráskód ellenőrzése után kiderült, hogy a szimulátor ugyan elő volt készítve az LLC particionálására, azonban a funkció nem volt elérhető. A kód kisebb módosítása elhárította ezt az akadályt.

A CPU-s és GPU-s programokat is tartalmazó szimulációk során előfordulhat, hogy az egyik alkalmazás előbb fejezi be a működést, mint a másik. Ez nem kívánatos jelenség, mert akkor lehet pontos információkat kinyerni az ilyen típusú workloadok viselkedéséről, ha minden program egyszerre fejezi be a működést. A szimulátor ezért tartalmaz egy kapcsolót (a paraméterfájlban lehet beállítani), ami a leállt alkalmazásokat indítja újra, ha van még olyan program, ami nem futott le egyszer sem. A CPU-s és GPU-s benchmarkokat is tartalmazó szimulációk során én is használtam ezt a funkciót, ez azonban jelentősen meghosszabbítja a MacSim futási idejét, egy szimuláció heterogén workload esetén 10 órán keresztül is futott.

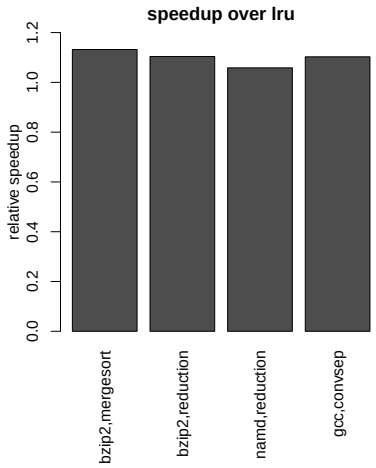
Az LP4HP algoritmus működését 14 heterogén (egy CPU-s és egy GPU-s programot tartalmazó) workload szimulációjával vizsgáltam meg. Ezek tartalmaztak cache-re érzékeny, illetve érzéketlen CPU-s és GPU-s alkalmazásokat is. A 3.8. és a 3.9. ábrának megfelelően az algoritmusban alkalmazott két metrika eltérő eredményeket adott bizonyos esetekben.



4.5. ábra. *A cache érzékeny CPU-s alkalmazások relatív gyorsulása 3.9. metrika esetén*

A 3.9. egyenlet (mértani közép) alkalmazása esetén a cache érzékeny CPU alkalmazások jelentősen gyorsultak a particionálatlan LLC-s eredményekhez képest, miközben a GPU-s alkalmazások sebessége változatlan maradt. Ilyen esetekben a 3.9. egyenlet szerint számított gyorsulás 1% és 19% között alakult (a 4.5. ábrán külön látható a CPU-s és a GPU-s alkalmazás sebességének változása).

Az összesített teljesítmény maximalizálásakor (3.6. egyenlet) azok a workloadok gyor-



4.6. ábra. *A cache érzékeny CPU-s alkalmazások relatív gyorsulása 3.6. metrika esetén*

sultak, amelyeknél a CPU és a GPU magok teljesítménye kiegyenlített volt, és valamelyik alkalmazás cache érzékeny volt. Az összesített teljesítmény ezek a workloadoknál 0% és 13% között nőtt (4.6. ábra).

Az algoritmus az elvárásoknak megfelelően főleg a CPU-s alkalmazásokat gyorsította, a cache érzékeny programokat akár 40%-al is (mcf és blackscholes programok esetén). Az LP4HP a TAP és a HeLM eredményeit nem mindig éri el, de az „offline" működésnek köszönhetően jelentősen kisebb hardveres költséggel ér el hasonló mértékű gyorsulást.

5. fejezet

Összefoglalás

A fizikai korlátok miatt a processzorok teljesítményét a gyártók nem tudják a korábban alkalmazott módszerekkel tovább növelni. A fogyasztási korlát a frekvencia és a magok számának növelését is akadályozza, ezért a proceszorgyártók egy szilíciumlapkára integrálnak CPU és GPU magokat is (heterogén processzor). Utóbbiak nagy teljesítményű, párhuzamos működésre optimalizált processzorok, amelyeket általános célú feladatok végrehajtására is fel lehet használni.

A heterogén processzorok egyre szélesebb körű elterjedése miatt mindinkább fontosabb lesz az ilyen típusú processzorok maximális kihasználása, így nem csak CPU, hanem GPU magokat is használni kell a számításigényes feladatok ellátására.

Két publikációról van tudomásom, amely a heterogén processzorok megosztott gyorsítótárjának működésének optimalizálásával foglalkozik. Ezek „online" algoritmusok, amelyek folyamatosan monitorozzák a processzor paramétereit, és ezek alapján az információk alapján választják ki a legkedvezőbb stratégiát. A futási időben működő módszerek hátránya, hogy jelentős lehet a hardverigényük.

A munkám célja egy olyan, újfajta algoritmus kidolgozása volt, amely képes kiküszöbölni a heterogén processzorokban alkalmazott CPU és GPU magok eltérő viselkedése által okozott negatív hatásokat. A dolgozatban bemutatott LP4HP az alábbi tulajdonságokkal rendelkezik:

- „offline" működésű, ezáltal nincs extra hardverigénye
- core samplinggel ellentétben nem lassítja a működést egy nem megfelelő cache menedzsment stratégia
- core sampling alkalmazásakor a GPU magok asszimetrikus működése ronthatja az elérhető eredményt, de az LP4HP nem érzékeny erre
- ismert IPS(s,w) adatok alapján képes megbecsülni egy alkalmazás várható futási sebességét
- képes adaptálódni az aktuális workload igényeihez
- az algoritmus futásideje és tárigénye is kedvező

Az algoritmust a MacSim szimulátor felhasználásával próbáltam ki. Előállítottam IPS(s,w) adatsorokat, és karakterizáltam a benchmarkokat. Az LP4HP algoritmus prototípusát R nyelven implementáltam, és 14 heterogén workloadon kipróbáltam a működését. Az LLC particionálásával a maximális teljesítmény akár 13%-al, az egyes alkalmazások sebessége pedig 40%-al növekedett.

Az algoritmust tovább lehetne bővíteni, hogy egy időben több alkalmazás számára is tudjon cache partíciót létrehozni (jelenleg egy CPU-s és egy GPU-s alkalmazásnak ad cache szeletet). Ez a fejlesztési irány azonban egy idő után zsákutca lehet, ugyanis a cache-t nem lehet tetszőlegesen sok partícióra osztani, az LLC elaprózódása a teljesítmény csökkenéséhez vezethet.

Egy másik kutatási lehetőség az IPS(s,w) adatok alapján egy olyan karakterizáló algoritmus fejlesztése, amelynek kimenete egy cache élettartam érték lenne. Ezzel egy olyan blokk cserélési stratégiát lehetne megvalósítani, amely futásidőben statikus prioritásokkal látja el az egyes alkalmazásokat. A nagyobb prioritású program adatának nagyobb élettartama lenen a cache-ben, a kisebb prioritású programnak kisebb.

A CPU és GPU magok közötti adatmegosztás az integrációnak köszönhetően gyorsul. A jövőben nem csak olyan heterogén processzorok várhatóak, amelynek van közös gyorsítótárjuk, hanem olyan szoftveres keretrendszerek, amelyek az ilyen típusú processzorok programozását egyszerűbbé és hatékonyabbá teszik. Az adatmegosztás közös címtérkép használatával nem jár majd adatmásolással. Ilyen helyzetekben kritikus lesz, hogy nagyobb teljesítmény érdekében a közös adat a cache-ben maradjon. „Offline” algoritmus cache particionálással és cache blokk élettartam optimalizálással ilyen esetben is kedvező eredményeket adhat.

Köszönetnyilvánítás

Köszönöm Dr. Mann Zoltán folyamatos útmutatását, amellyel az elmúlt évben a munkámat segítette.

Irodalomjegyzék

[1] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[2] Danowitz, Andrew and Kelley, Kyle and Mao, James and Stevenson, John P. and Horowitz, Mark. CPU DB: Recording Microprocessor History. *Commun. ACM*, 55(4):55–63, April 2012.

[3] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, October 1974.

[4] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[5] N. Goulding-Hotta, J. Sampson, Qiaoshi Zheng, V. Bhatt, J. Auricchio, S. Swanson, and M.B. Taylor. GreenDroid: An architecture for the Dark Silicon Age. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 100–105, Jan 2012.

[6] M.B. Taylor. Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1131–1136, June 2012.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[8] AB. Kahng. The ITRS design technology and system drivers roadmap: Process and status. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–6, May 2013.

[9] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[10] Wei Zang and Ann Gordon-Ross. A Survey on Cache Tuning from a Power/Energy Perspective. *ACM Comput. Surv.*, 45(3):32:1–32:49, July 2013.

[11] Pablo Viana, Ann Gordon-Ross, Edna Barros, and Frank Vahid. A Table-based Method for Single-pass Cache Optimization. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, GLSVLSI '08, pages 71–76, New York, NY, USA, 2008. ACM.

[12] Weixun Wang, P. Mishra, and S. Ranka. Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 948–953, June 2011.

[13] P. Chakraborty and P.R. Panda. SPM-Sieve: A framework for assisting data partitioning in scratch pad memory based systems. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*, pages 1–10, Sept 2013.

[14] Xiaoning Ding, Kaibo Wang, and Xiaodong Zhang. ULCC: A User-level Facility for Optimizing Shared Cache Performance on Multicores. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 103–112, New York, NY, USA, 2011. ACM.

[15] Jaekyu Lee and Hyesoon Kim. TAP: A TLP-aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[16] Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai. Managing Shared Last-level Cache in a Heterogeneous Multicore Processor. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 225–234, Piscataway, NJ, USA, 2013. IEEE Press.

[17] M.K. Qureshi and Y.N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 423–432, Dec 2006.

[18] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). *SIGARCH Comput. Archit. News*, 38(3):60–71, June 2010.

[19] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques* , Sep. 2012.

[20] MacSim. Simulator for Heterogeneous Architecture, July 2014. `https://code.google.com/p/macsim/`.

[21] Jason Power, Joel Hestness, Marc Orr, Mark Hill, and David Wood. gem5-gpu: A Heterogeneous CPU-GPU Simulator. *Computer Architecture Letters*, 13(1), Jan 2014.

[22] FusionSim. FusionSim simulator, July 2014. `http://www.fusionsim.ca/home`.

[23] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 353–364, New York, NY, USA, 2010. ACM.

[24] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

Rövidítések jegyzéke

BRRIP	bimodal re-reference interval prediction
CMOS	complementary metal oxide silicon
CMP	chip multiprocessor
CPI	clock per instruction
CPU	central processing unit
CUDA	compute unified device architecture
DLP	data level parallelism
DRAM	dynamic random acces memory
DRRIP	dynamic re-reference interval prediction
GPU	graphics processing unit
HeLM	heterogeneous LLC management
ILP	instruction level parallelism
IPS	instruction per second
ITRS	international technology roadmap for semiconductors
LLC	last level cache
LP4HP	LLC partitioning for heterogeneous processor
LRU	least recently used
MIMD	multiple instruction multiple data
MISD	multiple instruction single data
MOSFET	metal oxide semiconductor field effect transistor
MPKI	misses per kilo instruction
MRU	most recently used
MSHR	miss status hold register
NoC	network on chip
OoO	out of order
OpenCL	open computing language
RRIP	re-reference interval prediction
RRPV	re-reference prediction value
SDK	software development kit
SIMD	single instruction multiple data
SISD	single instruction single data
SMP	symmetric multiprocessor
SM	streaming multiprocessor

SPM	scratch pad memory
SRRIP	static re-reference interval prediction
TAP	TLP aware cache management policy
TDP	thermal design power
TLP	thread level parallelism
UCP	utility-based cache partitioning

Függelék

F.1. A szimulátor telepítése

A dolgozatban bemutatott eredményeket a MacSim szimulátor felhasználásával állítottam elő. A MacSim tracefájl alapján ciklus helyesen modellezi a processzor architektúráját. Ilyen tracefájl letölthető a MacSim GitHub oldaláról (SPEC06 benchmarkok), illetve CPU-hoz Pin (hivatkozás) felhasználásával, GPU-hoz GPUOcelottal állítható elő. A szimulátor, és a GPU-s frontend telepítése körülményes és sok hibalehetőséget rejt magában. Mivel a MacSim és a GPUOcelot sincs megfelelően dokumentálva, ezért több hetes munkába telt a rendszer feltelepítése és beállítása. Mások későbbi munkájának megkönnyítése érdekében ebben a fejezetben összegyűjtöttem a telepítéssel kapcsolatos tapasztalataimat.

F.1.1. Telepítés előtti teendők

A GPUOcelot és a MacSim szimulátor telepítése előtt érdemes megfontolni, hogy a szimulációhoz szükséges tracefájlok hova kerülnek. Ezek összességében meglehetősen nagy helyet foglalnak, azonban rengeteg kis méretű fájlból állnak, ezért az adott partíció, ahova kerül, előbb kifogyhat inodesból (ext fájlrendszer esetén) mint tárhelyből. Emiatt szükséges lehet manuálisan meghatározni a partíció inodesainak számát (ezt csak formázáskor lehet megtenni ext3 és ext4 esetén). Ehhez szükség szerint `gparted`dal hozzunk létre egy új partíciót, majd terminálban mkfs-sel hozzunk létre a partíción egy új fájlrendszert. Például ext4 fájlrendszer esetén:

```
$ mkfs.ext4 -N <inodeok száma> /dev/<partíció neve>.
```

Tapasztalatom szerint 50GB-os partícióhoz 10 millió inode elég.

F.1.2. CUDA toolkit telepítése

A GPUOcelot segítségével CUDA programokból tracefájlt lehet generálni a MacSim számára, azonban a fordításához szükség van az 5.0-ás CUDA toolkitre. Ez letölthető az NVIDIA weboldaláról: `https://developer.nvidia.com/cuda-toolkit-50-archive`. Itt válasszuk a 11.10-es Ubuntuhoz tartozó toolkitet (64 bites rendszerre a 64 bites verziót). Letöltés után futtathatóvá kell tenni a fájlt:

```
$ sudo chmod +x cuda_5.0.35_linux_64_ubuntu11.10-1.run
```

Az 5.0-ás toolkitben található `nvcc` (NVIDIA compiler) 4.6-os `gcc`-vel kompatibilis, azonban új linux disztribúciókon (például 14.04-es Ubuntu esetén) ennél újabb verzió van fennt. Az `update-alternatives` package felhasználásával egyszerre több `gcc` lehet egy

rendszerben, az aktuálisan használt verzió kijelöléséhez egy szimbolikus linket lehet beállítani. Telepítés előtt töröljük ki a `gcc`-hez és `g++`-hoz tartozó összes alternatívát (ha volt):

```
$ sudo update-alternatives --remove-all gcc
$ sudo update-alternatives --remove-all g++
```

Telepítsük a 4.6-os és 4.8-as `gcc`-t és `g++`-t:

```
$ sudo apt-get install gcc-4.6 gcc-4.8 g++-4.6 g++-4.8
```

Állítsuk be az alternatívákat:

```
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.6 10
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 20
$ sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.6 10
$ sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.8 20
```

A két verzió közül ezután az alábbi paranccsal lehet választani:

```
$ sudo update-alternatives --config gcc
$ sudo update-alternatives --config g++
```

Válasszuk ki a 4.6-os `gcc`-t. Ellenőrzésképpen adjuk meg az alábbi parancsot

```
$ gcc --version
```

Ha a megfelelő verziószámot írja ki, folytathatjuk a CUDA toolkit telepítését a letöltött állomány elindításával:

```
$ sudo ./cuda_5.0.35_linux_64_ubuntu11.10-1.run
```

A telepítő megkérdezi, hogy miket telepítsen fel, ekkor csak a toolkitet válasszuk ki! Ebben az esetben csak a GPUOcelot segítségével tudunk futtatni CUDA programokat, de ez volt a cél.

F.1.3. GPUOcelot telepítése

A CUDA toolkit telepítése után bizonyosodjunk meg, hogy a `Flex Bison Scons` és `Subversion` csomagok fel vannak telepítve:

```
$ sudo apt-get install flex bison scons build-essential subversion
```

Ezt követően töltsük le `svn`-ről a GPUOcelot legújabb verzióját (az alábbi parancs létrehoz egy gpuocelot nevű könyvtárat a jelenlegi könyvtárba):

```
$ svn checkout http://gpuocelot.googlecode.com/svn/trunk/ gpuocelot
```

A letöltés eltarthat egy darabig, mert a csomagban található tesztek elég nagyok.

Fel kell telepíteni a `Boost`, `LibGlew` és `LibGlut` libeket:

```
$ sudo apt-get install libboost-dev libboost-system-dev libboost-filesystem-dev
    libboost-thread-dev
$ sudo apt-get install libglew1.6-dev
$ sudo apt-get install freeglut3 freeglut3-dev
```

Ha eddig minden lépést megcsináltunk, feltelepíthetjük a GPUOcelotot (ehhez a gpuocelot könyvtárba kell belépni):

```
$ cd <letöltés helye>/gpuocelot
<letöltés helye>/gpuocelot$ sudo ./build.py --install --no_werr --no_llvm
```

A `no_werr` nélkül hibaüzeneteket dobhat a fordító nem használt változók miatt, a `no_llvm` pedig azért kell, mert nem szeretnénk X86 vagy AMD backendet használni.

Lehetséges hibaüzenetek a GPUOcelot telepítése közben

Fordítás elején a következő hibaüzenetet kaphatjuk:

```
In file included from ocelot/ocelot/parser/interface/PTXLexer.h:11:0,
                 from ocelot/ocelot/parser/interface/PTXParser.h:16,
                 from ocelot/ocelot/ir/implementation/Module.cpp:10:
.release_build/ocelot/ptxgrammar.hpp:352:14: error: 'PTXLexer' is not a member of '
    parser'
 int yyparse (parser::PTXLexer& lexer, parser::PTXParser::State& state);
              ^
.release_build/ocelot/ptxgrammar.hpp:352:32: error: 'lexer' was not declared in
    this scope
 int yyparse (parser::PTXLexer& lexer, parser::PTXParser::State& state);
                                ^
.release_build/ocelot/ptxgrammar.hpp:352:47: error: 'parser::PTXParser' has not
    been declared
 int yyparse (parser::PTXLexer& lexer, parser::PTXParser::State& state);
                                           ^
.release_build/ocelot/ptxgrammar.hpp:352:65: error: 'state' was not declared in
    this scope
 int yyparse (parser::PTXLexer& lexer, parser::PTXParser::State& state);
                                                              ^
.release_build/ocelot/ptxgrammar.hpp:352:70: error: expression list treated as
    compound expression in initializer [-fpermissive]
 int yyparse (parser::PTXLexer& lexer, parser::PTXParser::State& state);
                                                                      ^
scons: *** [.release_build/ocelot/ocelot/ir/implementation/Module.os] Error 1
```

Ezt a hibaüzenetet a ptxgrammar.hpp fájlban a 352.sor kitörlésével el lehet tüntetni:

```
<letöltés helye>/gpuocelot$ sudo nano .release_build/ocelot/ptxgrammar.hpp
```

Itt az alábbi sort kell kitörölni/kikommentelni:

```
int yyparse (parser::PTXLexer& lexer, parser::PTXParser::State& state);
```

Előfordulhat, hogy nem találja a `Boost` libeket a fordító. Ez azért lehet, mert a telepítő scriptben még korábbi nevükön szerepelnek ezek, így a nevük tartalmaz egy '-mt' tagot, ami felesleges. Töröljük ki az '-mt'-t az összes .py kiterjesztésű scriptből, ahol megtaláljuk.

Telepítés után lépjünk be a `/usr/local/lib` könyvtárba, és ellenőrizzük, hogy ott van-e a `libocelot.so` lib. Ha nincs itt, akkor nem volt sikeres a telepítés.

Ezután lépjünk be a `/usr/local/bin` könyvtárba, és futtassuk az alábbi parancsot:

```
/usr/local/bin$ ldd OcelotConfig
```

Ha minden jól működik, akkor egy listát kell látnunk a binárishoz linkelt libekről.

Előfordulhat, hogy az alábbi hibaüzenetet kapjuk:

```
Inconsistency detected by ld.so: dl-version.c: 224: _dl_check_map_versions:
    Assertion `needed != ((void *)0)' failed!
```

Ekkor adjuk meg az alábbi környezeti változót:

```
export LD_PRELOAD=/usr/lib/x86_64-linux-gnu/mesa/libGL.so.1
```

Ezt követően már ki kell listáznia a binárishoz linkelt libeket.

Ha a `libocelot.so` mellett azt kapjuk, hogy `not found`, akkor a linker nem kapta meg az `/usr/local/lib` útvonalat a keresési útvonalai közé. Adjuk meg az alábbi parancsot:

```
/usr/local/lib$ sudo ldconfig
```

Majd ellenőrizzük az eredményt az `ldd OcelotConfig` parancs kiadásával. Ha nem kaptunk hibaüzenetet, akkor feltehetjük a tracefájl generálásához szükséges libet is.

F.1.4. libOcelocelootTrace.so telepítése

Ha a `libOcelot.so` feltelepült, feltehetjük a tracefájl generálásához szükséges libet is. Ehhez szükség van a zlib package-re:

```
$ sudo apt-get install zlib1g-dev
```

Ezt követően nyissuk meg a `X86TraceGenerator.cpp` fájlt a `<letöltés helye>/gpuocelot/trace-generators/traces/implementation` könyvtárban. Keressük meg a következő sort, és írjuk át a 82-est 84-esre:

```
assert(ir::PTXInstruction::Nop == 82);
```

Ezen kívül az alábbi két sort duplikálni kell (eredetileg 591. és 617. sor):

```
{ (int)GPU_MAD , (int)GPU_MAD64 , (int)GPU_MAD    , (int)GPU_MAD64   }, // Mad
{ (int)GPU_SHL , (int)GPU_SHL64 , (int)GPU_INVALID, (int)GPU_INVALID }, // Shl
```

A fenti két sor mindegyikét másoljuk az eredeti alá még egyszer.

Ha ez kész, feltehetjük a `libOcelotTrace.so`-t:

```
<letöltés helye>/gpuocelot/trace-generators$ sudo scons install
```

Végül generáljuk újra a linker cache-t:

```
/usr/local/lib$ sudo ldconfig
```

F.1.5. Tesztek lefordítása

Ahhoz, hogy a MacSimhez tracefájlokat lehessen generálni, a teszteket a `libocelotTrace.so`-hoz kell fordítani. A 4.6-os `gcc` ha úgy ítéli meg, hogy egy lib nem szükséges, akkor nem linkeli hozzá a futtatható állományhoz. Ahhoz, hogy ezt elkerüljük, egy flag-et meg kell adni a linkernek. A `<letöltés helye>/gpuocelot/tests/` könyvtáron belül a `cuda2.2`, `cuda2.3`, `cuda3.2`, `cuda4.1sdk` és `parboil` könyvtárakban módosítani kell az `SConscript` scripteket. Adjuk hozzá az '-locelotTrace' libet az '-locelot' mellé (vesszővel elválasztva). Ezen kívül keressük meg a scriptben a compiler flageket (`env.Append(CPPFLAGS)`) vagy a LIBPATH-t (`env.Append(LIBPATH)`). Adjunk ide a scripthez egy új sort: `env.Append(LINKFLAGS = ['Wl,--no-as-needed'])`.

Ezt követően futtassuk az alábbi parancsot (fontos, hogy a `gcc`-t ekkor 4.6-os verzióra állítsuk):

```
<letöltés helye>/gpuocelot$ sudo ./build.py --no_werr --no_llvm --test_lists=parboil
 --test_level=full
```

Ez a parancs lefordítja a `cuda2.2`, `cuda2.3`, `cuda3.2`, `cuda4.1sdk` és `parboil` könyvtárakban található teszteket, és lefuttatja a `parboil` tesztjeit (ez tart a legkevesebb ideig). Ha minden lefordult, és a teszt is hiba nélkül lefutott, már majdnem kész vagyunk tracefájlok generálására.

F.1.6. Tracefájlok generálása

A lefordított benchmarkok futtatásához szükség van két fájlra, az egyik leírja a program egyes kernelei által használt regiszterek számát és a lefoglalt shared memória igényét, a másik a GPUOcelotot konfigurálja. Ez utóbbinak a teszt binárisával azonos könyvtárban kell lennie. Egy lehetséges konfiguráció:

```
{
    ocelot: "ocelot",
    trace: {
        database: "traces/database.trace",
        x86Trace: true
    },
    cuda: {
        implementation: CudaRuntime,
        runtimeApiTrace: "trace/CudaAPI.trace"
    },
    executive: {
        devices:                  ["emulated"],
        preferredISA:             "nvidia",
        optimizationLevel:        "none",
        reconvergenceMechanism:   "ipdom",
        defaultDeviceID:          0,
        required:                 False,
        asynchronousKernelLaunch: True,
        port:                     2011,
        host:                     "127.0.0.1",
        workerThreadLimit:        2,
        warpSize:                 32
    },
    optimizations: {
        subkernelSize: 10000
    }
}
```

A regiszterek és a shared memória használata meghatározza, hogy hány szálat lehet egy processzormagra (NVIDIA esetén Streaming Multiprocessor) ütemezni. A szükséges információkat az `nvcc`-vel lehet előállítani a kerneleket tartalmazó CUDA fájlokból. Ezek általában a `cu` kiterjesztésű fájlokban találhatóak:

```
/usr/local/cuda/bin/nvcc --cubin --ptxas-options=-v -arch sm_20 <letöltés helye>/
    gpuocelot/tests/cuda4.1sdk/tests/dxtc/*.cu -I. -I<letöltés helye>gpuocelot/
    tests/cuda4.1/tests/dxtc -I<letöltés helye>/gpuocelot/tests/cuda4.1sdk/sdk/
```

A fenti parancsra az alábbi eredmény adódik:

```
ptxas info : 0 bytes gmem, 76 bytes cmem[2]
ptxas info : Compiling entry function '_Z8compressPKjS0_P5uint2i' for 'sm_20'
ptxas info : Function properties for _Z8compressPKjS0_P5uint2i
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info : Used 37 registers, 2048 bytes smem, 60 bytes cmem[0], 8 bytes cmem[16]
```

A kernel információt tartalmazó fájl formátuma a következő:

```
<kernel neve> <regiszerek száma> <lefoglalt shared memória mérete>
```

A fenti példa esetén:

```
_Z8compressPKjS0_P5uint2i 37 2048
```

Egy benchmark több kernelt is tartalmazhat. A kernel információs fájlban az összes kernel adatát fel kell tüntetni.

A kernel információs fájl szerkesztése után be kell állítani 4 környezeti változót, amik szükségesek ahhoz, hogy a benchmark tudja, hogy hova kerüljenek a tracefájlok és hol található a kernel információs fájl. A `TRACE_NAME` változó tárolja a benchmark nevét, a `TRACE_PATH` a tracefájlok elérési útonalát, a `KERNEL_INFO_PATH` a kernel információs fájl elérési útját, a `COMPUTE_VERSION` a processzormagra vonatkozik (NVIDIA megnevezése Compute Capability). Az alábbi példa egy lehetséges beállítást ír le:

```
export TRACE_NAME="dxtc"
export TRACE_PATH="<trace fájlok helye>/$TRACE_NAME"
export KERNEL_INFO_PATH="<kernel info fájl helye>/$TRACE_NAME/kernelinfo"
export COMPUTE_VERSION="2.0"
```

Ezt követően már csak futtatni kell a lefordított benchmarkot. Érdemes megnézni, milyen paramétereket lehet átadni a terminálban az egyes teszteknek, ezek jelentősen befolyásolhatják a futásidőt (például gyakran feleslegesen sok iterációt tartalmaz egy benchmark).

F.1.7. MacSim telepítése

A MacSimet a fejlesztőcsapat GitHub oldaláról és SVN-ről is le lehet tölteni, azonban SVN-es verzióban korábban nem volt benne a McPAT modell. A letöltést követően az alábbi paranccsal lehet lefordítani a szimulátort.

```
<letöltés helye>/macsim$ ./build.py --power
```

A „power” argumentum az energyintrospector modult fordítja le a MacSim-hez, enélkül nem lehet használni a McPAT modellt.