Komunikacija grafičkog i središnjeg procesora u heterogenom računalnom sustavu

Sočković, Augustin

Sveučilište u Rijeci – Odjel za informatiku

Diplomski studij informatike, smjer Poslovna informatika

Augustin Sočković

Komunikacija grafičkog i središnjeg procesora u heterogenom računalnom sustavu

Diplomski rad

Mentor: v. pred. dr. sc. Vedran Miletić

Rijeka, 10. prosinca 2019.

Sažetak

U radu je opisana komunikacija osnovnog i grafičkog procesora u heterogenom računalnom sustavu, definirana heterogenua sustavska arhitektura (HSA), te skiciran "povijesni okvir" razvoja CPU-a i GPU-a do tehnologije koja je omgućila HSA.
Opisan je način rada platforme za razvoj aplikacija za heterogenu sustavsku arhitekturu, specifično način na koji grafički i središnji procesor unutar iste komuniciraju. Odabrana je platforma ROCm zbog otvorenosti i dostupnosti AMD-ovog grafičkog procesora.

Ključne riječi

CPU, GPU, ROCm, rocBLAS, rocSPARSE, rocRAND, AMD, HPC, heterogeno, računarstvo, arhitektura

Sadržaj

1. Uvod....1
2. Definicija heterogene sustavske arhitekture....2
2.1. Kratka povijest GPU računarstva – problemi koji su riješeni uz pomoć HSA....2
2.2. Osnova heterogene sustavske arhitekture....9
2.2.1. Memorijski model heterogene sustavske arhitekture....9
2.2.2. HSA model za kreiranje redova čekanja (engl. queuing model)....10
2.2.3. HSAIL virtualna ISA....11
2.2.4. Priključivanje HSA konteksta....11
2.3. Specifikacije HSA....12
2.3.1. Specifikacija sustavske arhitekture HSA platforme....12
2.3.2. Specifikacija HSA za vrijeme izvršavanja....12
2.3.3. Priručnik s referencama za HSA programera („HSAIL spec”)....13
2.4. HSA softver....13
3. Heterogeno računarstvo....16
3.1. Prednosti heterogenog računarstva....18
3.2. Nedostaci heterogenog računarstva....18
3.3. Razlozi razvoja heterogenog računarstva....18
4. HSA API za vrijeme izvođenja....20
4.1. Uvod u HSA vrijeme izvođenja....20
4.2. HSA API za bazično vrijeme izvođenja....23
4.2.1. Inicijalizacija vremena izvođenja i isključivanje....23
4.2.2. Obavijesti o vremenu izvođenja....25
4.2.3. Informacije o sustavu i HSA agentu....26
4.2.4. Signali....27
4.2.5. Redovi čekanja....28
4.2.6. Architected queuing language....29
4.2.7. Memorija....31
4.2.8. Kodni ciljevi i izvršenja....33
4.3. Produženo HSA vemena izvođenja....35
4.3.1. HSAIL finalizacija....35
4.3.2. Slike i uzorci....36
4.4. HSA API za vrijeme izvođenja – zaključno....39
5. Radeon Open Compute (ROC)....40
5.1. Biblioteka rocRAND....41
5.1.1. Općenito o biblioteci rocRAND....41
5.1.2. Podržani ROCm generatori slučajnih brojeva....41
5.1.3. Primjer: benchmark_rocrand_generate.cpp....42
5.2. Biblioteka rocBLAS....45
5.2.1. Općenito o biblioteci rocBLAS....45
5.2.2. Primjer: rocBLAS....45
5.3. Biblioteka rocSPARSE....47
5.3.1. Općenito o biblioteci rocSPARSE....47
5.3.2. Primjer: rocSPARSE....47
6. Zaključak....50
Literatura....51

1. Uvod

U sklopu ovog rada opisati ćemo načine na koje osnovni (CPU) i grafički procesor (GPU) komuniciraju u heterogenom računalnom sustavu. Najprije ćemo definirati heterogenu sustavsku arhitekturu te napraviti "povijesni okvir" odnosno opisati razvoj CPU-a i GPU-a kao zasebnih procesnih jedinica. Nakon toga, nabrojat ćemo osnovne principe na kojima počiva heterogena sustavska arhitektura, poput HSA memorijskog modela, HSA modela za kreiranje redova čekanja, HSAIL virtualnog ISA, priključivanje HSA konteksta te HSA softver. Zatim ćemo navesti HSA spcifikacije za: sustavsku arhitekturu HSA platforme, HSA vrijeme izvođenja i priručnik s referencom za HSA programera. Dodatno, istaknuti ćemo prednosti i nedostatke heterogenog računarstva te razloge zbog kojeg je HSA razvijena i standardizirana. Zatim ćemo definirati HSA API za vrijeme izvođenja, njezinu osnovnu postavku te produženo vrijeme izvođenja. Praktični dio rada odnosi se na rad sa Radeon Open Compute (ROCm) platformom. Koristiti ćemo tri različite ROCm biblioteke (rocRAND, rocBLAS i rocSPARSE) kako bi smo na primjeru njihovih izvornih kodova pojasnili načine upotrebe HSA arhitekture pri računanju stvarnih problema iz domene HPC računarstva, konkretnog područja primjene u matematici. U zaključku ćemo čitatelju ponuditi vlastito viđenje budućih zbivanja u svijetu HSA sustavske arhitekture te spomenuti neke od proizvođača, njihove projekte poboljšanja trenutnih GPU i CPU sustava.

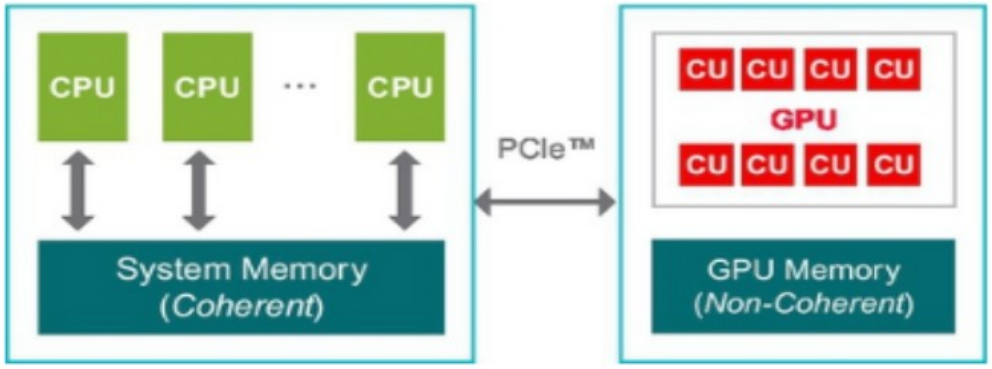
2. Definicija heterogene sustavske arhitekture

Heterogena sustavska arhitektura (engl. *heterogeneous system architecture*, kraće HSA) označava niz proizvođačkih specifikacija koje omogućuju integraciju središnjeg (engl. *central processing unit*, kraće CPU) i grafičkog procesora (engl. *graphics processing unit*, kraće GPU) pomoću iste sabirnice, koji na taj način međusobno dijele memoriju i zadatke. Cilj platforme HSA jest smanjiti komunikacijske zastoje između CPU-a, GPU-a i drugih računarskih uređaja te učiniti ove uređaje kompatibilnijima iz perspektive programera (smanjivanje potrebe za planiranjem migracije podataka). [1] U današnjem vremenu kontinuiranog tehničkog napretka, potreba za pristupom rješavanja poslovnih problema koji HSA omogućuje je sve veća. Na primjer, nove aplikacije namijenjene dubinskom učenju (engl. *deep learning*), obrada slike ili videozapisa visoke razlučivosti (engl. *ultra-high definition*) ili virtualna stvarnost (engl. *virtual reality*) su računalni procesi koji zahtijevaju ogromnu količinu procesorske snage uz otegotnu okolnost da se navedeni procesi pokreću tj. izvode na sve manjim uređajima. HSA se pokazala kao rješenje navedenih problema, na području industrije i akademske zajednice jer pokazuje znatno bolje performanse po vatu (engl. *performance per watt*) s obzirom na konkurentska rješenja. [2]

2.1. Kratka povijest GPU računarstva – problemi koji su riješeni uz pomoć HSA

GPU računarstvo započelo je ranih 2000-ih u razdoblju kada se GPU još uvijek nalazila na perifernoj kartici (što je zapravo tipično za računala) priključena na CPU putem **PCI** (engl. *peripheral component interconnect*[1]) sabirnice što je prikazano na sljedećoj Slici 1.

1 **PCI –** je hardverska sabirnica pomoću koje se u PC računalo dodaju unutarnje komponente. Npr. PCI kartice se može dodati na za to predviđeni PCI utor na površini matične ploče te na taj način pruža dodatne ulazno/izlazne utore/portove na stražnjoj strani računala. [3]



Slika 1: Izračunavanje naslijeđenog GPU uređaja na diskretnim GPU karticama [30]

Kod naslijeđenog sustava s diskretnim GPU-om, jezgre CPU-a djeluju u sistemskoj memoriji, gdje se nalazi operativni sustav i aplikativni programi. Da bi postigli maksimalne performanse GPU čvorovi moraju koristiti vlastiti bazen (engl. *pool*) čija je implementacija na GPU kartici dosta čest slučaj. Već spomenuti, odvojeni bazeni često stvaraju velike „glavobolje" za programere, pošto je memoriju potrebno premještati između različitih memorijskih bazena, ovisno o tome na kojem procesoru će se izvršavati naredne operacije. Memorija sustava (engl. *system memory*) je obično velika, ali ne nudi vrlo veliku propusnost (engl. *bandwidth*). Memorija grafičke kartice je obično mnogo, no ima značajno veću propusnost. CPU može raditi usklađeno (engl. *coherently*) i s velikom propusnošću prema sistemskoj memoriji; međutim, središnji procesor može raditi samo neusklađeno i s mnogo nižim propusnim opsegom prema GPU memoriji. Suprotno tome, GPU obično djeluje neusklađeno na oba memorijska bazena, s velikom propusnom opsegom prema GPU memoriji i mnogo nižim propusnim opsegom ka središnjoj memoriji. U višeprocesorskom okruženju, sustav koji sadrži više memorijski bazena se često naziva NUMA (skraćeno od engl. *non-uniform memory access*) – ne-jedinstveni pristup memoriji. U CPU sustavima s više utičnica (engl. *socket*, još poznatih kao *symmetric multi-processing* ili kraće SMP) obično je identičan memorijski bazen priključen na svaku utičnicu. Stupanj nejednakosti/ne-identičnosti u takvim NUMA SMP sustavima relativno je nizak. Suprotno tome, naslijeđeni GPU računalni sustav ima ono što nazivamo ozbiljnim NUMA karakteristikama. [4]

Uz problem s više memorijskih bazena, naslijeđeni GPU-i koristili su potpuno različit virtualni adresni prostor iz CPU-a za istu aplikaciju ili unutar istog postupka. To ukazuje na situaciju da kada se komad podataka u sistemskoj memoriji nalazi na adresi A za CPU, toj istoj memoriji pristupa GPU s potpuno drugačijom memorijskom adresom B. To znači da se adresa ne može proslijediti između CPU-a i GPU-a i ne može se odmah preusmjeriti ni na jedan od ta dva procesorska uređaja.

Umjesto toga, softver mora intervenirati i izvršiti pretvorbu adresa (engl. *address conversion*). Što je još gore, rani GPU-ovi radili su u potpunosti iz odstupanja unutar memorijskih međuspremnika (engl. *memory buffers*), umjesto da su se koristile pune adrese ili pokazivači. CPU često radi sa strukturama podataka koje sadrže ugrađene pokazivače (engl. *embedded pointers*). Na samim početcima GPU računanja, svi takvi ugrađeni pokazivači morali su se promijeniti u offsete, a strukture podataka morale su biti potpuno "spljoštene" prije slanja obrade s CPU-a na GPU.

Sljedeći problem koji je bilo potrebno riješiti bilo je straničenje memorije. Neki od novijih sustava omogućavaju grafičkoj kartici pristup sistemskoj memoriji, ali uz određena ograničenja. Centralni procesori mogu samoinicijativno (raditi po svojoj volji) u svim sistemskim memorijama, a ukoliko je OS uklonio stranicu iz sistemske memorije, CPU će obavijestiti OS da povrati stranicu prije nastavka rada. Nasuprot tome, grafičke kartice mogu raditi samo sa onim stranicama sistemske memorije koje je OS prethodno „zamrznuo ili prikvačio". To znači da su ranije aplikacije morale upućivati pozive OS-u kako bi prikvačile takve memorijske stranice te na taj način spriječile da ih OS zapakira (engl. *page out*). Obično, na ovaj način operativni sustavi se zapravo pridržavaju pravila prema kojemu ne dopuštaju da više od polovice stranica u memoriji sustava budu prikvačene te na taj način omogućavaju responzivnost (mogućnost odgovaranja na vrijeme) sustava. To ograničava količinu sistemske memorije koju GPU može koristiti i opterećuje programera da unaprijed definira sve stranice kojima GPU može kasnije pristupiti i unaprijed ih zaključa.

Konačni problem grafičkih kartica vezan uz memoriju bio je povezanost (engl. *coherency*). Naslijeđenim sustavima nedostajala je mogućnost više-nitnosti (engl. *multiple threads)* kako bi dijelili memorijske podatke putem pravila o vidljivosti hardvera (engl. *hardware visiability rules*), umjesto da se oslanjaju na softver za ispiranje predmemorije (engl. *flush caches*) na granicama vidljivosti. Jezgre CPU-a u više-jezgrenom CPU-u ili preko SMP utičnica su međusobno usklađene putem predmemorijske povezanosti (engl. *cache coherent*). To znači da ako je jedan procesor napisao podatke na adresu A, a podaci su u njenoj predmemoriji, a ne u potpunosti u središtu memorije, te zatim neka druga jezgra procesora čita sa adrese A, hardver sustava će osigurati da se kod čitanja prikažu ažurirani podaci. Obično ovaj postupak provodi jedan CPU na način da ispituje/ sondira (engl. *probing*) predmemoriju drugih CPU jezgri kako bi provjerio postoje li ažurirani podaci prije preuzimanja podataka iz memorije. Kada je zamišljena HSA, računske jedinice (CU) GPU-a bile su sposobne sondirati predmemorije CPU-a, ali CPU nije bio sposoban za sondiranje predmemorija GPU-a (ne vrijedi obratno). U stvari, GPU računske jednice čak nisu bile u stanju međusobno sondirati predmemorije. Umjesto toga, softver je bio odgovoran za pokretanje poslova GPU-a u serijama (engl. *batches*) i pamtio je da između poslova ili serija isprazni predmemoriju

GPU-ova kako bi rezultati postali vidljivi u ostatku sustava. To je značilo da su GPU-i sposobni samo za vrlo grubu povezanost (engl. *coarse-grained coherency)*, što ih je ograničavalo po pitanju vrste poslova koje bi mogli obavljati. Ovo je bila vrlo teška paradigma za programere koji su navikli programirati višeprocesorske procesore, gdje se nikad nisu morali brinuti o predmemoriji za ispravnost.

Sljedeći problem s kojim su se suočili rani programeri GPU računala bio je režijski trošak (engl. *overhead inherent*) svojstven pokretanju GPU zadatka. Obično svi podaci koje GPU obrađuje dolaze od CPU-a. Svi međuspremnici za unos podataka moraju se kopirati iz sistemske memorije procesora u GPU memoriju. Zatim se naredbe za izvršavanje zadatka na GPU-u moraju sastaviti (engl. *assemble*) i pretvoriti (engl. *convert*) u oblik u memoriji koji GPU može dohvatiti (engl. *fetch*) i dekodirati. To se događa pomoću funkcije API-ja (engl. *application programming interface*[2]) za organizaciju redova (engl. *queuing API function*) u OpenCL-u ili CUDA-i koja pretvara generički naredbeni paket (engl. *generic command packet*) u konačni hardver, u vlasničkom obliku proizvođača GPU-a. Tada se poziv operacijskog sustava koristi za stavljanje naredbenog paketa u jedan red do hardvera koji dijele svi aplikacijski procesi. Konačno, hardver izvršava posao dovršetka i stvara prekid CPU-a. Nakon toga, svi generirani rezultati kopiraju se iz GPU memorije u CPU memoriju kako bi ih učinili dostupnima ostatku programa koji rade na CPU-u. Tipično je taj režijski iznos bio vrlo visok. To je značilo da čak i ako je GPU bio mnogo brži pri paralelnoj obradi (npr. 10 puta brži), prebacivanje na GPU bilo je samo neto dobit ako je iznos izračuna koji se izvrši bio vrlo velik. Ovaj nadzemni limit (engl. *overhead*) ograničava vrstu posla koji se može pretovariti (engl. *offloaded*).

Budući da su naslijeđeni GPU računarski sustavi bili toliko specijalizirani, njih nije bilo moguće izravno programirati koristeći C++, Python, Java-u ili bilo koji drugi popularni programski jezik koji koriste programeri aplikacija. Umjesto toga, stvoreni su novi modeli i modeli programa poput Brook+, CUDA i OpenCL. Navedeni modeli za programiranje obično su uključivali dva elementa: runtime knjižnicu s API-jem i GPU s "kernel jezgrom" temeljenom na vrlo ograničenom podskupu C programskog jezika. Većinu programera najčešće zanima programiranje koda u nekom od njima "omiljenih" programskih jezika koji je odabran za datu aplikaciju i prilično su otporni na pisanje modula u drugom jeziku unutar istog projekta. Te činjenice su nadalje ograničile prihvaćanje GPU računarstva na one koji su voljni učiti i uključiti nove jezike, pogotovo one kojima nedostaju mnoge potrebne značajke, poput pokazivača, predložaka, rekurzije, alokacije memorije i iznimki.

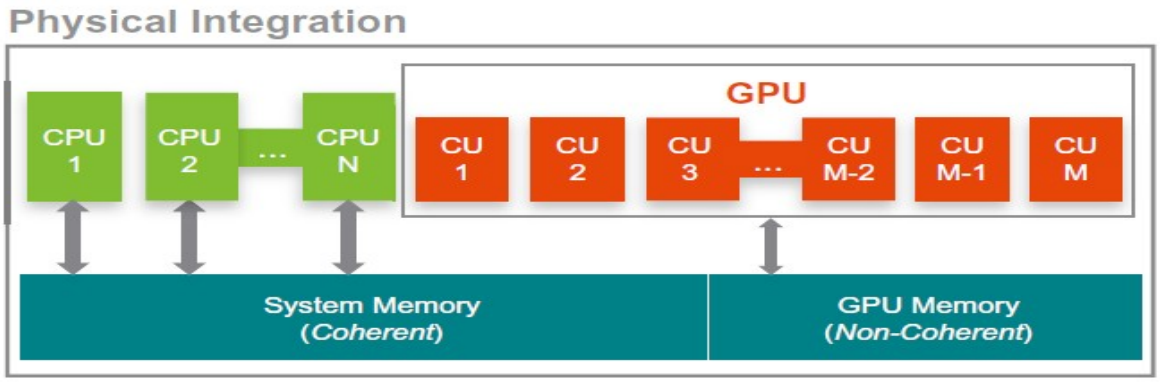
2 **API** - je sučelje ili komunikacijski protokol između različitih dijelova računalnog programa koji je namijenjen pojednostavljivanju implementacije i održavanja softvera. [5]

Suptilni, ali važan učinak zahtjeva specijaliziranog jezika za GPU računarstvo bio je taj što je vodio ka zahtjevu za "dvostruko-izvorskim programiranjem", što se pokazalo vrlo velikim problemom. Većina aplikacija koje potencijalno mogu ubrzati sa izvođenjem koristeći GPU računarstvo također se moraju moći pokretati na sustavima koji ne sadrže odgovarajući GPU. To znači da svi moduli koji mogu biti učitani na GPU moraju biti kodirani dva puta - jednom u glavnom jeziku aplikacije kako bi se izvodili na CPU-u (npr. C++), a zatim ponovno na jeziku jezgre GPU-a radi potencijalnog ubrzanja (npr. OpenCL). Ove dvije kopije izvornog koda obično se nalaze u različitim izvornim datotekama, ali moraju se održavati tako da sadrže istu funkcionalnost i uvijek daju iste rezultate za iste ulazne podatke. To znači da ako se pogreška pronađe i popravi u jednoj rutini, mora se provjeriti u ostalim rutinama i tamo također ispraviti, ukoliko greška postoji. Također, ukoliko se i najmanje promijeni funkcionalnost, mora se kodirati na oba mjesta. Takav razvoj dualnih izvora je ekspanzivan, naporan i teško je postići željeno.

Konačno, neki od naslijeđenih GPU računalnih programskih modela su vlasnički i pokreću se isključivo na GPU-u od samo određenog proizvođača. Većina programera aplikacija ne želi pisati različit kôd za različite proizvođače, čime se dodatno ograničava prihvaćanje računanja na GPU-ima.

Rana GPU računarska podloga bila je vrlo teško okruženje u kojem su uglavnom uspjevali samo oni profesionalni programeri koji su bili voljni nositi se s tim problemima. U HSA Fondaciji, njezini članovi su se prihvatili riješavanja svih tih problema, uklanjanja mnogih prepreka ka cilju da GPU-u postane pravi koprocesor procesne jedinice te omogućavanje "mainstream" programerima jednostavnije „single source” programiranje za GPU računarstvo.

Dokaz koliko je ubrzanje bilo dostupno putem GPU-a (kroz prvih 10 godina izračunavanja uz pomoć GPU-a) očituje se u stotinama komercijalnih GPU ubrzanih programa. Navedeni programi kodirani su od strane skupa desetaka tisuća programera širom svijeta koji su stručni u pisanju programa za naslijeđena aplikacijska programska sučelja za GPU računanje. Mnogi od tih komercijalnih programa nalaze se na područjima kao što su CAD, CAM, CAE, obrada slike, istraživanje nafte i plina, financijsko modeliranje, vremensko modeliranje, bioznanosti i druge HPC aplikacije. Zapravo, cijelo područje dubinske neuronske mreže (engl. *deep neural networks*, ili skraćeno DNN) za klasifikaciju slike, glasa i videa ne bi postalo financijski održivo bez računalne propusnosti GPU-a.



Slika 2: Izračunavanje naslijeđenog GPU-a na SOC uređajima [30]

Zapitajmo se što bi se dogodilo kad bismo mogli iskoristiti prednosti GPU izračunavanja jednostavnim poput programiranja za višejezgreni procesor? Ono što bi se dogodilo i ono što se događa je da donosimo prednosti računarskog ubrzanja GPU-a za 10 milijuna programera u svijetu koji već pišu kod na C-u, C++-u, Java-i, Python-u i OpenMP-ju. [4]

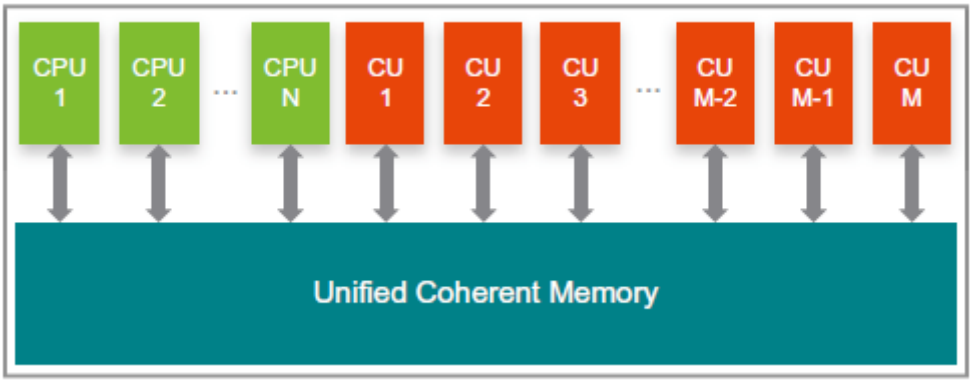
Sljedeća inovacija koja je olakšala programiranje GPU računarstva bilo je stvaranje SOC-ova ili APU-a, kao što je prikazano na Slici 2.

Fizička integracija GPU-a i CPU-a na isti silicij i memorijski kontroler bio je vrlo važan korak. Ovaj korak postavio je GPU na istu sistemsku memoriju kao i CPU, ali nažalost, time nije postignuto arhitektonsko objedinjenje memorijskog prostora CPU-a i GPU-a. U mnogim je slučajevima fizički neprekidno "isklesano područje" sistemske memorije u trenutku dizanja sustava (engl. *boot time*) bilo rezervirano za GPU memoriju - otprilike nekih 510 MB u sustavima od 4 GB. GPU jezgre često bi mogle raditi brže u ovoj izdvojenoj GPU memoriji nego u ostatku sistemske memorije, te aplikacija ne treba zaključavati stranicu prije njezine upotrebe. Ovaj GPU memorijski fond (engl. *pond*) nije koherentan/povezan sa CPU-om. GPU predmemorije ostaju nekoherentne, a sustav je i dalje imao potpuno odvojene virtualne adresne prostore za dva memorijska polja i različite propusne širine prema svakom memorijskom spremištu. Unatoč činjenici da je GPU bio na istom memorijskom kontroleru kao i CPU, NUMA arhitektura sustava još uvijek postoji. GPU je i dalje bio ograničen na pristup maloj količini memorije, a kopije su bile uobičajene između memorije sustava i GPU memorije, iako su memorijske ćelije bile u potpuno istim DRAM čipovima.

Bez potpune koherencije memorije, zajedničkog adresnog prostora i sposobnosti GPU-a da dereferenciraju pokazivače, programeri bi mogli programirati samo u OpenCL-u i CUDA-i,

ostavljajući ih u situaciji da vrlo nezgodno barataju algoritmima pisanim u dva izvorna koda. Također, u tim ranim SOC-ima GPU jezgre obično su imale samo jedan red naredbi kojim se i dalje upravljao pozivima operativnih sustava kako bi se osigurala atomska dostupnost čekanja. Dakle, unatoč vrlo dobrom koraku unaprijed u fizičkoj integraciji, programiranje GPU-a računarstva na tim uređajima ostalo je domena stručnjaka i do danas nije privuklo masovno usvajanje.

Slika 3 prikazuje kako je HSA koji podržava SOC konfiguriran za memoriju. Sada uistinu imamo jedinstveno memorijsko spremište gdje više nije potreban "iskidati" za GPU, a jezgre GPU-a dizajnirane su tako da omoguće pune performanse sistemske memorije. Nadalje, hardver GPU-a nadograđen je za: upotrebu istog adresnog prostora kao i CPU, da bi u potpunosti bio koherentan s CPU-om, djeluje u pagabilnoj memoriji, obrađuje greške na stranici i koristi stvarne adrese umjesto da nadoknađuje unutar memorijskog međuspremnika. Konačno, uspostavljen je odnos "pokazivač je pokazivač" na onom procesoru kojem se pristupa. Ovo je nevjerojatno moćno jer omogućava GPU-u da radi izravno na CPU strukturi podataka koja je nastala s jezika visoke razine. Zauzvrat, to dovodi do toga da se jezik visoke razine kompajliraju izravno i u CPU i GPU kôdu. Nema više potrebe za upotrebom jezika specifičnog za GPU – čime se postiže željeno jedno-izvorsko programiranje. [4]

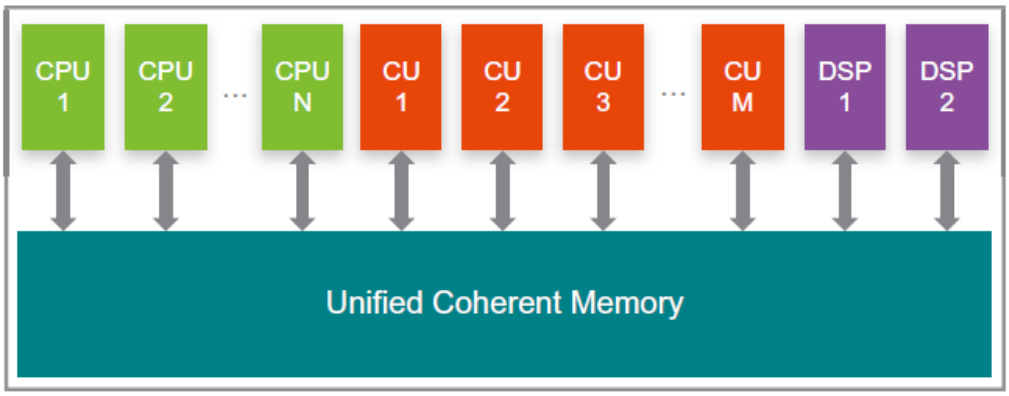


Slika 3: SOC odobren od strane HSA [30]

Slika 4 pokazuje da HSA nije izričito ograničena na računanje na GPU-ima. U budućnosti se očekuje da će DSP[3] (skraćeno od engl. *digital signal* processor) jezgre i drugi ubrzivači sudjelovati u HSA arhitekturi. Jedan od glavnih pozitivnih aspekata HSA je taj što je podrška za operacijski sustav (OS) općenita. To znači da nakon što se u OS doda HSA podrška za jedan tip procesora,

3 **DSP –** digitalni procesor signala je specijalizirani mikroprocesorski čip (ili SIP blok, skraćeno od engl. *system in package* ili *system-in-a-package*, broji nekolicinu integriranih krugova zatvorenih u jednom paketu nosača čipa), čija je arhitektura optimizirana za operativne potrebe digitalne obrade signala.

recimo onaj za GPU izračunavanje, daljnje promjene u OS-u nisu potrebne kako bi se dodale dodatne vrste procesora, poput DSP-a, kodeka, FPGA[4] (niz polja programibilnih vrata, skraćeno od engl. *field-programmable gate array*)-e i drugih ubrzivača. HSA je uistinu heterogena arhitektura koja se ne ograničava na samo dvije vrste procesora. [4]



Slika 4: HSA omogućen SOC s više procesora izvan CPU-a. [30]

2.2. Osnova heterogene sustavske arhitekture

HSA uključuje desetine značajki i stotine zahtjeva, dok sama osnova HSA arhitekture počiva na nekolicini principa koji ovu vrstu arhitekture razlikuju od njezinih prethodnica, omogučujući jedno-izvorski razvoj računalnih programa te po prvi put čini GPU računarstvo dostupnim njezinim osnovnim/vjernim (engl. *mainstream*) programerima. [4]

2.2.1. Memorijski model heterogene sustavske arhitekture

O nekim značajkama modela HSA memorije se već duže raspravlja, poput: objedinjenog adresiranja, pagabilnu (engl. *pageable*) memoriju koja se može pretraživati na stranici, koherencije potpune memorije te mogućnosti prosljeđivanja pokazivača između procesora različitih vrsta. U ovom se odjeljku govori o značajki sinkronizacije koja dodatno omogućava izravnu suradnju između procesora. [4]

HSA model uključuje definiciju atomičnih operacija platforme, što znači potreban skup operacija u zajedničkoj memoriji, koji je atomičan u odnosu na sve procesore koji pristupaju memoriji. Ova atomična platforma je ključna za omogućavanje interakcije između različitih vrsta procesora i to u

4 **FPGA** - je integrirani krug dizajniran nakon proizvodnje za konfiguriraciju od strane kupaca ili dizajnera - odatle pojam "programibilno polje" (engl. *field-programmable*).

operacijama: upravljanja redovima, sinkronizaciji rada/posla i implementaciji strukture podataka bez upotrebe zaključavanja (engl. *lock-free data structures*).

HSA memorijski model je memorijski model opuštene konzistencije (engl. *relaxed consistency*), koji kako bi omogućio paralelnu performansu visokih propusnosti, te je sinkroniziran (odnosi se na HSA memorijski model) sa semantikom, barijerama i ogradama koje se isporučuju/oslobađaju na tržište (engl. *store-released)* i prikupljaju teret (engl. *load-acquire*). Semantičko prikupljanje i oslobađanje su zapravo pravila koja utječu na vidljivost opterećenja i spremišta unutar niti izvršenja (engl. *execution threads*). Zajamčeno je da "preuzimanje tereta" u nizu/sekvenci opterećenja neće biti prepravljeno (engl. *re-ordered*) u odnosu na bilo koji (ili od strane bilo kojeg) teret koji ga prati unutar niti izvršenja. Zajamčeno je da se isporučivanje neće ponovno naručiti u odnosu na bilo koji dućan (engl. *store*) koji mu prethodi unutar niti izvršenja. To omogućuje da uobičajena isporučivanja i trgovine budu prepravljeni unutar niti izvršenja, od strane programa prevoditelja (engl. *compiler*) ili hardvera radi svrhe postizanja najboljeg učinka. Operacije prikupljanja i oslobađanja tereta omogućuju sinkronizaciju narudžbi, omogućuju programiranje bez zaključavanja (engl. *lock-free programming*) i omogućavaju pouzdane interakcije u memoriji između neovisnih niti.

HSA memorijski model je dizajnerski kompatibilan sa svim glavnim memorijskim modelima jezika visoke razine poput memorijskih modela programskih jezika C++14, Java, OpenMP i .Net.

2.2.2. HSA model za kreiranje redova čekanja (engl. queuing model)

HSA model za kreiranje reda čekanja dizajniran je kako bi omogućio slanje poslova sa niskom razinom čekanja između HSA agenata, to je ujedno i terminologija za **HSA-procesor**. Ovdje se spominju dva osnovna principa/izvedbe HSA-procesora. [4]

Prvi je k**orisnički način planiranja** (engl. user mode scheduling). To znači da su HSA agenti (poput GPU računarskih jezgri) u stanju dosezati/dohvaćati iz višestrukih zasebnih redova pod kontrolom programa za raspoređivanje (engl. *scheduler*, skraćeno na hrv. **PZR**). Dizajn redova korisničkog načina za kreiranja redova čekanja, omogućuje stvaranje zasebnog reda za svaku aplikaciju ili čak više redova po aplikaciji. Svaka aplikacija može smjestiti pakete za slanje izravno u svoj vlastiti red/ove, bez upotrebe **API-ja za HSA vrijeme izvođenja/izvršavanje** (engl. *runtime API*, u daljnjem tekstu koristiti ćemo se terminom HSA runtime kada mislimo na HSA API za vrijeme izvođenja/izvršavanja) ili usluge OS-a za upravljanje zajedničkim redom čekanja. Redovi korisničkog načina (za kreiranje redova čekanja) značajno smanjuju kašnjenje koje aplikacije

trebaju za predati posao. Nadalje, redovi korisničkog načina također bolje podržavaju napredne PZR radi provođenja različite kvalitete usluga (engl. *service polices*).

Primjenjujući ova dva koraka, tipična vremena za otpremu smanjila su se s nekoliko milisekundi na nekoliko mikrosekundi, čime je postignuto drastično povećanje učinkovitosti sustava te je omogućeno da manji poslovi u paralelnoj obradi podataka dodatno benificiraju samog ubrzanja.

2.2.3. HSAIL virtualna ISA

Intermedijarni (međusobni, srednji) jezik heterogene sustavske arhitekture (engl. *heterogeneous system architecture intermediate language*, skraćeno **HSAIL**[5]), je virtualni **ISA** (skraćeno od engl. *Internet security and acceleration*, prev. na hrv. Internetska sigurnost i ubrzanje) za paralelne računalne rutine ili jezgre. HSAIL je posredni prikaz niske razine, obično generiran od strane programa prevoditelja jezika visoke razine, koji je neovisan o tvrtki proizvođaču te vrsti ISA. Prije izvršavanja izvodi se finalizator (engl. *finalizer*), kako bi preveo HSAIL u ISA i to u formatu koji je razumljiv uređaju na kojem će se izvršavati. HSAIL je ključan za kreiranje HSA nedefiniranog ISA (engl. *ISA agnostic*) i kompatibilan je između različitih proizvođača. [4]

Treba imati na umu da se HSA finalizator može izvoditi u različito vrijeme, čime se omogućava prilagođavanje različitim korisničkim potrebama. Može se pokrenuti „upravo u ovom trenutku” (engl. *just in time*, skraćeno **JIT**), u trenutku prvog pokretanja aplikacije, u vremenu kada se aplikacija instalira ili čak u trenutku prevođenja. Izbor trenutka/vremena kada će se HSA finalizator pokrenuti prepušta se programeru aplikacije i ovisiti će o tome da li je aplikacija izrađena za otvorenu hardversku platformu koja podržava „plug and play” uređaje, poput PC računala ili za zatvorenu hardversku platformu, poput telefona ili HPC instalacija.

2.2.4. Priključivanje HSA konteksta

Da bi heterogeno računalstvo zaista postalo sveprisutno, HSA aplikacije moraju dobro funkcionirati u sustavima u kojima se istodobno pokreće više aplikacija ili procesa. Naslijeđeni GPU računalni sustavi pate od problema s kvalitetom usluge kada više od jednog programa planira (program „scheduler”) rad na GPU-u. Ova problem se posebno očitovao u situacijama ako jedan od tih programa koristi računarske jezgre koje ne popuštaju uređaj na duži vremenski period ili koriste velike količine prikvačene (engl. *pinned*) memorije. [4]

5 **HSAIL** - je jezik prevodilac niske razine, dizajniran tako da izražava (engl. *express*) paralelna područja koda te da bude prenosiv na više platformi proizvođača i hardverskih generacija. [6]

Baš iz tog razloga, HSA uključuje specifikacije: za pretpostavku (engl. *preemption*) i prebacivanje konteksta kako bi omogućio OS-u da predvidi dugotrajne poslove, spremanje stanja tih poslova radi kasnijeg nastavka i prebacivanje se na drugi posao, baš kao što to čini CPU. Zahtjevi za kontekstnu promjenu u HSA specifikacijama razlikuju kod različitih modela strojeva.

2.3. Specifikacije HSA

HSA se sastoji od tri specifikacije koje opisujemo u nastavku.

2.3.1. Specifikacija sustavske arhitekture HSA platforme

Ova specifikacija dokumentira hardverske zahtjeve za HSA, uključujući: zajedničku virtualnu memoriju, domene koherencije, signalizaciju i sinkronizaciju, atomske memorijske operacije, vremensko pečatiranje sustava (engl. *system time stamping*), kreiranje redova čekanja od strane korisnika, AQL (engl. *acceptable quality levels*, prev. na hrv. prihvatljiva razina kvalitete) raspoređivanje agensa (engl. *agent scheduling*), prebacivanje konteksta, rukovanje iznimkama, infrastrukturu uklanjanja pogrešaka (engl. *debug infrastructure*) i otkrivanje topologije (engl. *topology discovery*). [4]

2.3.2. Specifikacija HSA za vrijeme izvršavanja

Ova specifikacija dokumentira kratku biblioteku vremena izvršavanja (engl. *runtime*), na koju se HSA aplikacije povezuju za upotrebu platforme. HSA runtime uključuje API-je za inicijalizaciju i isključivanje, podatke o sustavu i agentu, upravljanje memorijom, signale i sinkronizaciju, arhitektonsku otpremu (engl. *architected dispatch*) i rukovanje pogreškama. [4]

Jedan od aspekata HSA-ovog vremena izvođenja koji ga značajno razlikuje od svih njegovih prethodnika po pitanju računarskih vremena izvođenja jest činjenica da API za otpremu posla zapravo nije potreban. Umjesto toga, runtime omogućuje aplikaciji da postavi svoje redove u korisničkom načinu rada, kojima može prosljeđivati rad po volji, s malim kašnjenjem. Po dizajnu, HSA runtime se uopće ne poziva tijekom paralelne obrade, te se stoga ne bi trebao pojaviti u tragovima profila, ako se koristi prema namjeri.

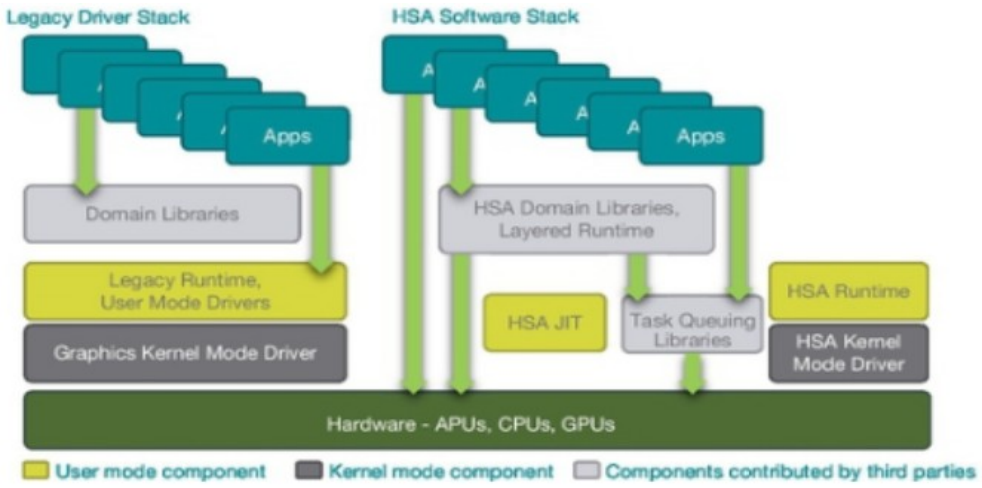
2.3.3. Priručnik s referencama za HSA programera („HSAIL spec”)

Ova specifikacija dokumentira HSA virtualni stroj i HSA međusobni jezik. Većinu HSAIL-a je generirano od strane generatora koda „open source” programa prevodioca, kao što su **LLVM** i **gcc**. Iako se to ne preporučuje za opće programere, u HSAIL-u je moguće ručno kodirati (engl. *hand-code*), te se na taj način neke biblioteke visokih performansi mogu dodatno optimizirati. [4]

Ova specifikacija također definira BRIG, objektni format za HSAIL. BRIG omogućuje stvaranje „ogromnih binarnih izvora” (engl. *„fat binaries”*) iz jedno-izvorskih programa. U tom slučaju, za paralelne rutine, program prevoditelj će istovremeno generirati CPU objektni kôd i HSAIL kod u istu binarnu mrežu (engl. *binary*). U trenutku podizanja sustava, korisnik se odlučuje koji put će se izvršiti i to na temelju rutine otkrivanja (engl. *dicovery routines*) u HSA trenvremenu izvršavanja. Mogućnost da se funkcija ili metoda „kompajlira” dvaput, jednom za CPU i jednom za HSAIL, ostvaruje obećanje o "jedno-izvorskom programiranju".

2.4. HSA softver

Arhitektura HSA platforme pruža učinkovitije izvođenja softvera, kao što je prikazano na Slici 5.



Slika 5: Izvođenje HSA softvera. [30]

Na ovoj slici široke okomite strelice predstavljaju često korištene putove naredbi i izvršenja. U naslijeđenim upravljačkim programima (engl. *driver stack*) svaka naredba od aplikacije prema hardveru, mora prije isporuke naredbenog paketa hardveru proći kroz nekoliko softverskih slojeva, uključujući: runtime, upravljačke programe korisnika, upravljačke programe kernela i operativni sustav. Nasuprot tome, u HSA softverskom paketu, aplikacije mogu izravno pisati naredbene pakete

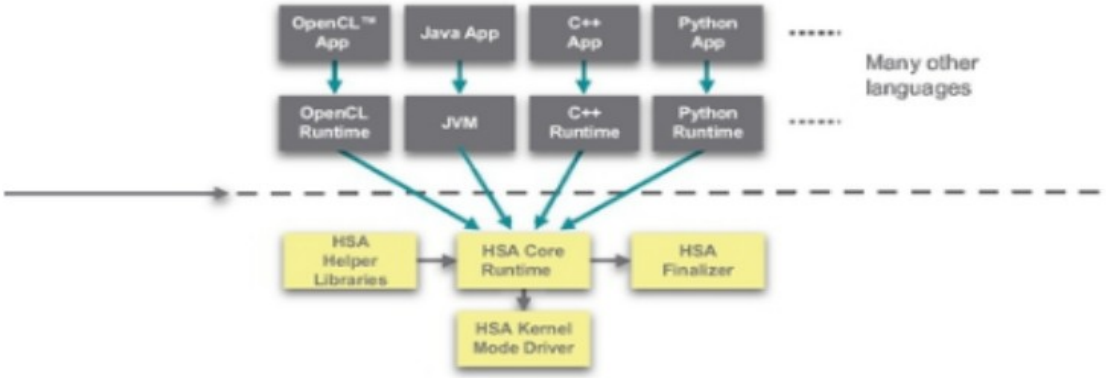
u hardverske redove, bez potrebe pozivanja kroz dodatne softverske slojeve. U ovoj arhitekturi i upravljačkom programu za kernel još uvijek postoji runtime, ali oni se nalaze po strani te se pozivaju mnogo rjeđe kako bi riješili stvari poput: inicijalizacije, kreiranje redova čekanja, alokaciju memorije i rukovanja iznimkama. HSA runtime ne mora biti pozvan za slanje GPU-ovog zadatka. [4]

Dvije su velike prednosti ovog softverskog snopa:

1. Troškovi za otpremu rada na GPU-u su mnogo manji.
2. Aplikacija ima veću kontrolu nad izvršavanjem i manje je osjetljiva na promjene nakon otkrića da se njezin profil performansi promijenio nakon ažuriranja upravljačkog programa.

Obje ove karakteristike vrlo su privlačne programerima aplikacija.

Pored učinkovitijeg izvršnog skupa, HSA pruža programerima mogućnost da koriste jezik prema vlastitiom nahođenju, za pisanje koda koji će se izvršiti na GPU računskim motorima, kao što je prikazano na Slici 6.



Slika 6: Izbor programskog jezika na HSA [30]

HSA softverski paket se uklapa u implementaciju otvorenog koda i velik dio HSA softverskog sklopa AMD dostavlja u obliku otvorenog koda, te je već javno dostupan. Dok su pokretači finalizatora i jezgra specifični za proizvođača hardvera, ostale komponente kao što su HSA runtime, HSA pomoćne biblioteke i LLVM prevoditelj za HSAIL generator koda neovisni su o hardveru i mogu se koristiti kod svih proizvođača hardvera.

Nekoliko je prednosti sklopa otvorenog koda za heterogeno računalstvo:

1. Jedna inačica otvorenog koda koja sadrži uobičajene komponente izbjegava divergentne/iskrivljene implementacije iste funkcionalnosti, stvarajući ujednačeno softversko okruženje za programere aplikacija.
2. Mnogi kupci više vole softver s otvorenim kodom tako da mogu doprinijeti optimizaciji performansi, izvršavati vlastito održavanje te postići da se ne osjećaju ograničenima niti jednim proizvođačem.
3. Softver otvorenog koda otvara vrata sveučilišnim istraživačima ka „istraživanju svijeta" ovog uzbudljivog novog područja te daju svoj doprinos.

3. Heterogeno računarstvo

Pod pojmom heterogene sistemske arhitekture, obuhvaćena je nova hardverska platforma i povezani softverski paket koji omogućava procesorima različitih vrsta da zajedno rade efikasno i kooperativno na zajedničkoj memoriji. Ova vrsta računalne arhitekture se primjenjuje kod pametnih telefona (engl. *smartphones*), tableta, PC računala, radnih stanica (engl. *workstations*) te sve češće na HPC čvorovima superračunala. HSA arhitektura svoje korijene vuče od arhitektura sutava zajedničke memorije koje su se u posljednjih 30 godina vezale zajedno uz jezgre CPU-a i CPU utičnica. HSA se razvila na mnogim saznanjima o homogenim multiprocesorskim sustavima te ta saznanja primjenjuje na današnje heterogene sustave na čipu (engl. *system-on-chips*, kraće SoC[6]) i ubrzane procesne jedinice (APU, skraćeno od engl. *accelerated processing unit),* koji su prevladavali prethodnih 15-ak godina. Ovi SOC-ovi i APU-ovi osim CPU jezgara sadrže i mnoge specijalizirane procesne jedinice poput GPU-a, DSP-a (skraćeno od engl. *digital signal processing*[7]), Codec-a[8] , DMA[9] (skraćeno od engl. *direct memory access*) motora i kripto-motora[10]. Može se ustvrditi da prije nastanka HSA nije bilo pokušaja kreiranja sličnih arhitektura za integraciju različitih vrsta procesnih jedinica radi njihova nesmetanog međusobnog rada.

HSA je izvorno bila usmjerena na rješavanje problema učinkovitog korištenja GPU-a kao paralelnog koprocesora za rad s CPU-om. Prilikom razvoja HSA, autori su shvatili da su arhitektonske karakteristike potrebne za učinkovit rad s GPU-om također primjenjive na mnogo različitih vrsta specijaliziranih procesnih jedinica, od kojih su mnoge već prisutne u SOC-ovima. Svaki GPU procesor za računanje zahtijeva vlastiti razvoj HSA kroz korake potrebne za arhitektonsku integraciju.

Izvorno, grafički procesori su bili priključeni na CPU poput ulazno-izlazni (engl. *input-output*) uređaji. U svojim početcima, GPU-i su se nalazili na odvojenoj silicijskoj površini dok im je

6 **System-on-Chips** – uređaji rastuće upotrebe i sve manjih dimenzija poput tableta, smartphone i drugih mobilnih uređaja [7]

7 **Digitalna obrada signala –** označuje upotrebu digitalne obrade u području računala ili specijaliziranih digitalnih procesora signala više namjene, radi izvršavanja širokog spektra operacija obrade signala. Signali obrađeni na ovaj način su niz brojeva koji predstavljaju uzorke kontinuirane varijable u domenama poput vremena, prostora ili frekvencije. [8]

8 **Codec** – uređaj ili računalni program za kodiranje ili dekodiranje (obrnuti postupak od kodiranja) digitalnog toka podataka ili signala. [9]

9 **DMA –** izravni pristup memoriji je značajka računalnih sustava koja omogućava određenim hardverskim podsustavima pristup glavnoj memoriji sustava (memorija sa slučajnim pristupom ili engl. *random-access memory*), neovisno o središnjem procesoru – CPU-u. [10]

10 **Sigurnosni kriptoprocesor –** jest namjensko računalo na čipu ili mikroprocesor za izvođenje kirptografskih operacija, ugrađeno je u „pakiranje" s više fizičkih sigurnosnih mjera, koji mu pružaju stupanj otpornosti na dodir. [11]

najčešća primjena bila grafički prikaz. Ovo nasljeđe ulazno-izlaznih uređaja, uzrokovalo je višestruke izazove za upotrebu GPU-a kao koprocesora pri izračunavanjima opće namjene (u početku se naziva „*GPGPU izračunavanje”*). Problem je nastao kada se započelo implementacijom grafičkih procesora na čipove u ranijim dizajnima SOC-ova. Programiranje uz pomoć GPU jezgre unutar SOC-a bio je pravi izazov, iz tog razloga osniva se Zaklada HSA (engl. *HSA Foundation*) 2012. godine, čija glavna zadaća postaje kreiranje arhitektonske integracije svih vrsta procesnih jedinica na SOC-u. [4]

Ideja heterogenog računarstva postoji već nego vrijeme, no tek 2012. godine i predstavljanjem superračunala Titan, (Nacionalni Laboratorij Oak Ridge) koje je kasnije postalo svjetsko najbrže superračunalo (zasnovano na HSA) dolazi do ogromnog zaokreta u konstrukciji računalnih čvorova temeljenih na HSA. Heterogenu arhitekturu danas koriste mnoge ključne tehnološke tvrtke koje se bave proizvodnjom računalnih čipova. Važnost HSA dokazana je osnivanjem ne-profitne inženjerske organizacije/zaklade pod nazivom HSA Fondacija, osnovnane zajedničkim naporima jednih od najvećih tehnoloških imena poput: AMD, ARM, Imagination Technologies, Qualcomm, MediaTek, Samsung i Texas Instruments. Cilj spomenute zaklade jest da se pomoću HSA arhitekture smanji vrijeme kašnjenja komunikacije između različitih aspekata SoC-a – poput komunikacije GPU – CPU procesora, čineći ova dva uređaja kompatibilnijima iz programerske perspektive. To se postiže tako što se programeru olakšava zadatak planiranja pomicanja podataka između memorija uređaja kako bi se povećala njihova sveukupna izvedba. Na primjer, dopuštajući programima da koriste resurse GPU kartice i da rade na istoj razini obrade kao i CPU sustava, u paraleli, na taj način se koristi znatno manje energije nego li u slučajevima kada se obrada prebacuje na serijsko izvođenje između GPU-a i CPU-a. Prema riječima GPU dizajnera (iz tvrtke Nvidia), HSA je ključna kako bi ova dva procesora (GPU i CPU) radila u paraleli te na taj način omogućili brži i učinkovitiji rad u računalnim sustavima. [7]

Poanta heterogenog računalnog modela jest da jedna veličina ne odgovara svima. Kombinacijom GPU-ova i CPU-ova u jedinstveni sustav moguće je u potpunosti iskoristiti prednosti obje arhitekture. Paralelni i serijski segmenti radnog opterećenja (engl. *workload*) izvršavaju se na najprikladnijem procesoru, CPU-u optimiziranome na kašnjenje (engl. l*atency-optimised*) ili GPU-a optimizirane propusnosti/protoka (engl. t*hroughput-optimised*), koji pružaju brže ukupne performanse, veću učinkovitost te niži energetski i trošak po jedinici (engl. *cost per unit*) izračun. [7]

3.1. Prednosti heterogenog računarstva

Heterogene platforme se sve više pojavljuju u svakoj od domena računarstva, računala i servera visokih performansi (engl. *high-performance computing*), do tableta, mobilnih telefona i ugrađenih uređaja (engl. *embedded devices*).

Povećanjem efikasnosti obrade i omogućujući nove mogućnosti obrade, ovi procesori odvažno ulaze u sljedeće razdoblje arhitekture računalnih sustava i inovacija. Maksimizacijom cjelokupnih računskih mogućnosti sustava s CPU-om i GPU-om, korisnici i programeri mogu vidjeti povećanja performansi i energetske učinkovitosti kroz različite primjene i na tržištima u rasponu od računalnog učenja (engl. *machine learning*) do molekularne dinamike te od istraživanja nafte i plina do vizualnih efekata i računalom-generiranog snimanja (engl. c*omputer-fenerated imaging*). [7]

3.2. Nedostaci heterogenog računarstva

Mnogi trenutni CPU-i i GPU-ovi dizajnirani su kao zasebni procesni elementi i ne rade zajedno učinkovito te su stoga nezgrapni za programiranje. Svaki ima zaseban memorijski prostor, zahtijevajući od aplikacije izričito kopiranje podataka sa CPU-a na GPU i obratno. [7]

3.3. Razlozi razvoja heterogenog računarstva

Na starijim ne-heterogenim postavkama, program koji se izvodio na CPU redovima (engl. *queues*), radio je za GPU-u koristeći se sistemskim pozivima kroz snop upravljačkih programa (engl. *driver stack*) kojima upravlja potpuno odvojeni planer (engl. *scheduler*). U ovakvoj postavci, zbog opisane CPU – GPU komunikacije, dolazi do značajnih kašnjenja u otpremanju (engl. *dispatch*) koja su „opravdana” samo u slučaju kada aplikacija zahtijeva jako velike količine paralelnih izračuna. Nadalje, ukoliko program koji se izvodi na GPU-u želi izravno generirati radne stavke, bilo za sebe ili za CPU, to je nemoguće.

Da bi u potpunosti iskoristili mogućnosti jedinica namijenjenih za paralelno izvršavanje, bitno je da dizajneri računalnih sustava razmišljaju drugačije. Potrebno je rekonstruirati računalne sustave kako bi se čvrsto integrirali različiti računski elementi na platformi i to u razvijeni (engl. *envolved*) središnji procesor, dok istovremeno pruža programsku stazu (engl. *programming path*) koja ne zahtijeva temeljne promjene za softverske programere. Sve navedeno je primarni cilj dizajna HSA (heterogene sistemske arhitekture). [7]

Heterogena arhitektura ubrzava/pojačava postavke „starih” (ne-heterogenih) procesora kreiranjem poboljšanog dizajna procesora koji otkriva prednosti i mogućnosti glavnih (engl. *mainstream*) programabilnih računalnih elemenata na način da ne ometa zajednički rad.

HSA arhitektura omogućava da aplikacije mogu kreirati strukture podataka u jednom jedinstvenom (engl. *unified*) adresnom prostoru i mogu pokrenuti radne stavke na hardveru koji je najprikladniji za određeni zadatak. Dijeljenje podataka između računarskih elementa jednostavno je poput slanja pokazivača. Višestruki računski zadaci mogu se izvršavati na istim povezanim (engl. *coherent*) memorijskim regijama, koristeći barijere i atomske memorijske operacije potrebne za održavanje sinkronizacije podataka, kao što to rade više-jezgreni (engl. *multi-core*) CPU-i danas.

4. HSA API za vrijeme izvođenja

HSA API za vrijeme izvođenja (engl. *runtime*) je korisnički API koji omogućava/pruža domaćinu sučelje potrebno za pokretanje računarskih jezgri odnosno zrana (engl. kernel) nad raspoloživim HSA agentima. HSA runtime može se klasificirati u dvije kategorije: baza i proširenje (engl. *core* i *extension*). Osnovni HSA runtime API-ja (engl. *core runtime API*) dizajnirano je za podršku operacijama zahtijevanima od strane specifikacija HSA sustavske platforme i moraju ga podržati svi sustavi usklađeni sa HSA (engl. *HSA-compliant*). Prošireni HSA runtime API može biti: odobren od strane HSA (engl. *HSA-approved*) ili svojstven proizvođač (engl. *vendor spcific*) te je opcionalano za HSA-kompatibilne sisteme. U ovom najprije je opisani su: API za HSA bazični runtime, uključujući inicijalizaciju i isključivanje, obavijesti, podatke o sustavu i HSA agentu, signale, redove, memoriju i kodne objekte (engl. code objects) i izvršne datoteke, nakon čega slijedi API za HSA-odobreno vrijeme izvođenja (ili HSA-odobreni runtime), zaključno sa HSAIL finalizacijom i slikama (engl. *HSAIL finalization and images*). [12]

4.1. Uvod u HSA vrijeme izvođenja

HSA standard integrira CPU, GPU i druge računalne ubrzivače (akceleratore) u jedinstvenu platformu s zajedničkim, širokopojasnim (engl. *high-bandwidth*) memorijskim sustavom za podršku velikog broja modela paralelnih podataka (enlg. *data-parallel*) i paralelnih zadataka (engl. *task-parallel*). Da bi se postigli navedeni ciljevi, HSA zaklada predlaže tri specifikacije: HSA platformsku arhitekturu sustava (engl. platform system architecture) HSA , HSAIL i HSA runtime. [12]

Gledano s hardverske perspektive, specifikacija HSA arhitektura sustavske platforme, definira skup arhitektonski zahtjeva sustava, kao što su otkrivanje topologije platforme HSA, signali i sinkronizacija, model čekanja, model postavljanja reda čekanja (engl. architecture queuing language, skraćeno AQL), memorijski model itd., za podršku HSA modelu programiranja i programske infrastrukture sustava. Programirajući na HSA programskom modelu, programeri mogu kreirati prijenosne HSA aplikacije koje iskorištavaju snagu i prednosti performansi namjenskih HSA agenata. Mnogi od tih HSA agenata, uključujući GPU i DSP, sposobni su i fleksibilni procesori prošireni posebnim hardverom radi ubrzanja izvođenja paralelnog koda. Povijesno gledano bilo je iznimno teško programirati ove uređaje zbog njihovog specijaliziranog ili vlastitog

programskog jezika. HSA želi donijeti prednosti ovih agenata u "mainstream" programskim jezicima, kao što su C ++ i Python. [12]

Specifikacija HSAIL-a definira prijenosni i niski nivo programa kompajlera srednjeg jezika koji predstavlja srednji/intermedijarni format računarskih jezgara GPU-a. Kompajlerski program visoke razine obrađuje većinu procesa optimizacije te generira HSAIL za paralelna područja koda. Kompajler niske razine i "*lightweight compiler-i*"[11], koji se još naziva finalizator, prevodi HSAIL u ciljnani strojni kod. Finalizator se može pozvati u vrijeme prevođenja, instalacije ili u trenutku izvršavanja. Svaki HSA agent osigurava vlastitu implementaciju finalizatora. [12]

Specifikacija HSA vremena izvođenja definira "user-mode" API niskih troškova, te pruža potrebno sučelje za "upper-level" vrijeme izvođenja jezika, kao što je OpenCL vrijeme izvršavanja za pokretanje računarskih kernela na raspoloživim HSA agentima. Krajnji cilj dizajna HSA vremena izvođenje jest osigurati mehanizam otpreme visokih performansi koji je prenosiv na što veći broj proizvođačkih implementacija HSA. Kako bi se postigle visoke performanse: slanja, postavke argumenata (engl. argument setting) te mehanizmi pokretanja kernela, definirani su na razini hardvera, dok specifikacijska razina se definira na HSA platformi arhitekture sustava. HSA runtime API standardiziran je na način da jezici izrađeni za HSA runtime mogu biti pokrenuti na raznim platformama različitih proizvođača koje podržavaju spomenuti API. [12]

HSA runtime API može se razvrstati u dvije kategorije: **bazični/osnovni runtime** (engl. *core runtime*) i **produženi runtime** (engl. *extended runtime*). Svrha HSA osnovnog runtime-a API-ja za je podrška operacijama koje zahtijevaju specifikacije arhitekture HSA platforme sustava (engl. *system platform architecture spcification*). HSA osnovni runtime API je potreban za svaku odgovarajuću HSA implementaciju.

Ključni dijelovi HSA osnovnog runtime API-ja uključuju:

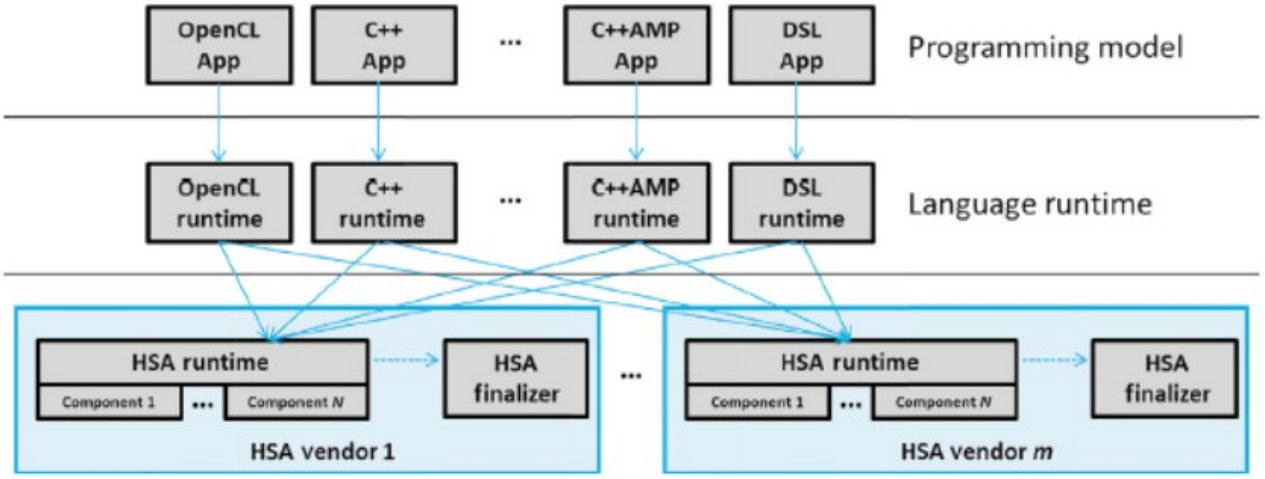
- Inicijaliziaciju i isključivanje HSA runtime-a
- Obavijesti o runtime-u
- Podatke o sustavu i HSA agentu
- Signale
- Redove
- AQL pakete
- Memoriju

11 **Lightweight compiler** – jest program kompajler osmišljen da ima vrlo mali otisak memorije (engl. *memory footprint*), lako ga je implementirati (što je važno za prijenos jezika na različite računalne sustave) i/ili ima minimalističku sintaksu i značajke. [13]

HSA API za produženi runtime može biti: HSA odobren ili specifičan obzirom na pojedinog proizvođača. Proširenja vremena izvođenja odobrenog od strane HSA su opcionalna tj. nisu obvezna s obzirom na odgovarajuću HSA implementaciju, no očekuje se da će uskoro biti široko dostupna. Trenutno postoje dva proširenja odobrena od HSA, a to su HSAIL API za finalizaciju i slikovni API. API za HSAIL finaliziaciju omogućava aplikaciji: kompajliranje skupa HSAIL modula u binarnom formatu (BRIG), generiranje kodnih objekata specifičnih za određenog proizvođača te dohvaćanje navedenih kodnih objekata. API za slike omogućava aplikaciji: specificiranje određenog prikaza koji se učita sa početne slike, zatim pohranjuje podatak o izvoru prikaza (engl. resource layout) i drugim svojstvima u memoriju. Pomoću slikovnog API-ja, aplikacija može kontrolirati alokaciju slikovnih podataka kao i učinkovitije upravljati memorijom. Proširenja odobrena od strane HSA Fondacije mogla bi se promovirati na temeljni API u budućim verzijama standarda. Kad se dogodi nova nadogradnja, specifikacija proširenja dodaje se u osnovnu specifikaciju. Prefiks proširenja funkcija, tipova i konstanti enumeracije odnosno nabrajanja (engl. enumeration constants) bivaju uklonjeni ukoliko proširenje postane dio nadogradnje. Međutim, u novijim verzijama HSA implementacija, također je i dalje potrebno nastaviti s deklaracijom te izlaganjem originalnih verzija funkcija, tipova i enumeracijskih konstanti kao pomoć pri migraciji sa starijih na novije verzije HSA implementacije. [12]

Implementacija HSA vremena izvođenja može uključivati komponente razine jezgre operacijskog sustava (potrebne za neke hardverske komponente) ili može uključivati samo komponente korisničkog prostora (npr. simulatore ili CPU implementacije). HSA runtime API standardiziran je za sve HSA proizvođače, to jest jezične implementacije koje koriste HSA runtime mogu se izvoditi na različitim platformama koje podržavaju API. Proizvođači su odgovorni za opskrbu vlastitim implementacijama HSA vremena izvođenja koje podržavaju sve HSA agente na njihovim platformama. HSA ne podržava mehanizam za kombiniranje vremena izvođenja različitih proizvođača. Slika 7 prikazuje položaj izvođenja HSA u nizu arhitekture HSA softvera. Na vrhu snopa je programski model ili jezici poput **OpenCL™**, **C++**, **C++AMP**, **OpenMP** ili jezik specifičnog za domenu (engl. *domain-specific language*, skraćeno **DSL**). Programski model mora sadržavati neki način navođenja paralelne regije koja se može ubrzati. Na primjer, OpenCL je poziva clEnqueueNDRangeKernel() s pridruženim kernelima te rasponima rešetki (engl. gird ranges). Za drugi primjer, C++AMP omogućuje parallel_for, konstruktor koji razgraničuje paralelne regije izvršavanja u funkcijama metoda. U sloju runtime jezika, svaki jezik uključuje svoj runtime; čija se implementacija može temeljiti na HSA runtime-u. U budućnosti će neki jezici moći izravno

otkriti (engl. expose) HSA runtime. Kada program kompajler generira kôd za paralelnu regiju, runtime jezika će postaviti i otpremiti paralelnu regiju do HSA agenata pozivanjem odgovarajuće HSA runtime rutine. Jezik runtime-a je također odgovoran za pozivanje odgovarajućih funkcija API-ja HSA vremena izvođenja za inicijalizaciju HSA runtime-a, odabir ciljanih uređaja, stvaranje redova za izvršavanje, upravljanje memorijom itd. Finalizator je neobavezna komponenta HSA vremena izvođenja. Aplikacija može pozvati finalizator putem HSAIL rutina finalizacije (tijekom izvođenja aplikacije) radi konvertiranja HSAIL modula u ciljni binarni oblik. [12]



Slika 7: Sklop softverske arhitekture s HSA runtime [12]

4.2. HSA API za bazično vrijeme izvođenja

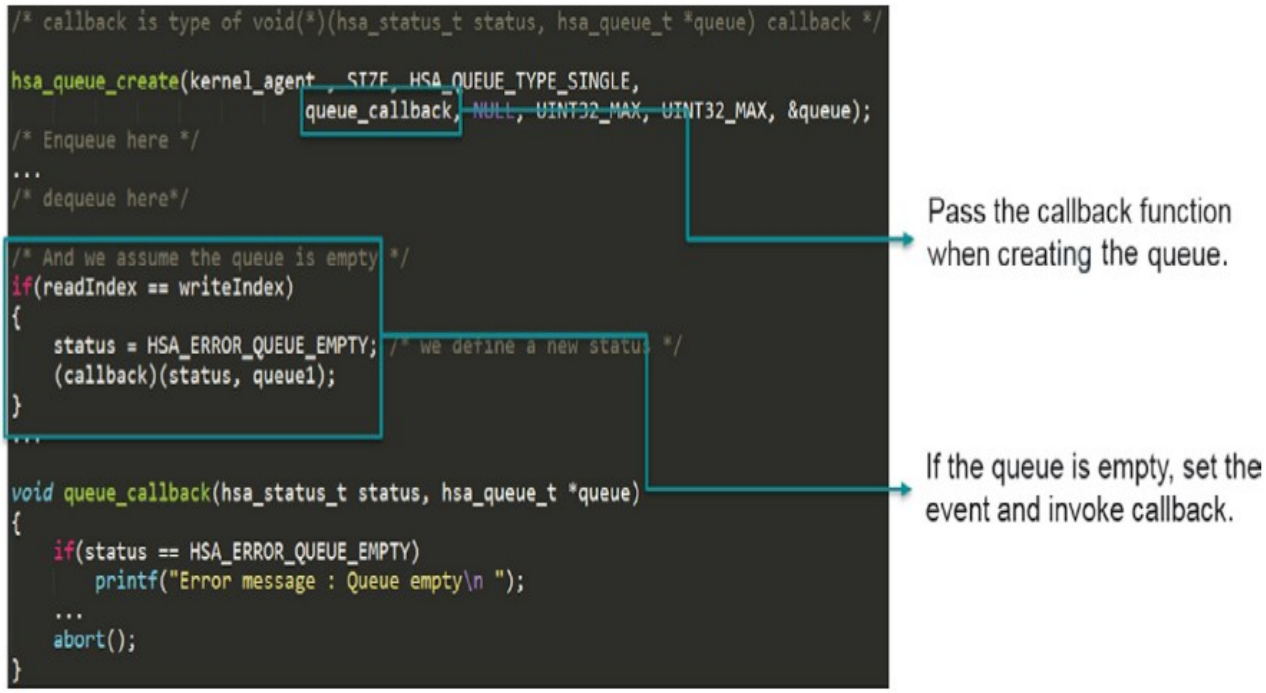
Kao sastavni dio HSA vremena izvršavanja, HSA bazični runtime API uključuje inicijalizaciju i isključivanje runtime-a, obavijesti o vremenu izvršavanja, podatke o HSA agentu i sustavu, signale, redove, AQL pakete i memoriju. Razumijevanje određenih funkcionalnosti ovih značajki pomaže kako hardverskim arhitektima tako i jezičnim implementatorima u njihovom radu. U nastavku ćemo detaljnije opisati svaki odjeljak. [12]

4.2.1. Inicijalizacija vremena izvođenja i isključivanje

Prije nego što se HSA runtime može koristiti aplikacijom, najprije mora biti inicijaliziran. Svrha inicijalizacije je kreiranje instance vremena izvođenja te alokacija resursa na kreiranoj instanci vremena izvršenja. Instanca vremena izvođenja je specifična obzirom na implementaciju. Tipična runtime instancija može sadržavati informacije o platformi, topologiji, referentnom broju (engl. reference count), redovima, signalima itd. Slika 8 prikazuje primjer vremena izvođenja koja se

koristi u HSAemu. Rutina inicijalizacije definirana u HSA vremenu izvođenja je **hsa_init**. Kada se prvi put u određenom procesu poziva **hsa_init**, stvara se instancija vremena izvođenja; referentni broj povezan s *runtime* instancom postavljen je na jedan. Nakon prvotne inicijalizacije spomenute rutine, svaki sljedeći put, referentni broj se povećava za jedan. Referentno brojanje koristi se za bilježenje broja priziva inicijalizacije API-ja u određenom procesu. Za određeni postupak postojati će samo jedna instanca vremena izvođenja. [12]

Kada instanca vremena izvođenja više nije potrebna, aplikacija koja stvara spomenutu instancu također poziva rutinu za isključivanje kako bi zatvorila instancu izvođenja. Rutina isključivanja definirana u HSA API-ju za vrijeme izvođenja je **hsa_shut_down**. Kada se pozove hsa_shut_down, broj referenci povezanih s instancom vremena izvođenja smanjuje se za jedan. Ako je referentni brojač trenutačne HSA instance vremena izvršavanja manji od jedan, tada se svi resursi povezani s vremenom izvođenja (redovi čekanja, signali, informacije o topologiji itd.) smatraju nevaljanim. Poziv bilo koje HSA runtime rutine osim **hsa_init** u sljedećim API pozivima rezultira nedefiniranim ponašanjem. Kad se hsa_shut_down naredba pozove više puta od hsa_init, HSA runtime će vratiti statusni kod **HSA_STATUS_ERROR_NOT_INITIALIZED** kako bi obavijestio korisnike da HSA runtime nije inicijaliziran. Ovisno o implementaciji, nevažeći resursi dodijeljeni vremenu izvođenja mogu se osloboditi ili čekati dok se ne pokrene još jedna hsa_init naredba, kako bi se ponovno aktivirala instanca HSA runtime-a. Na ovaj se način se postiže da uvijek jedna instanca HSA runtime-a bude zadržana za sve klijente u istom procesu dok svi ne pozovu HSA rutinu isključivanja. [12]

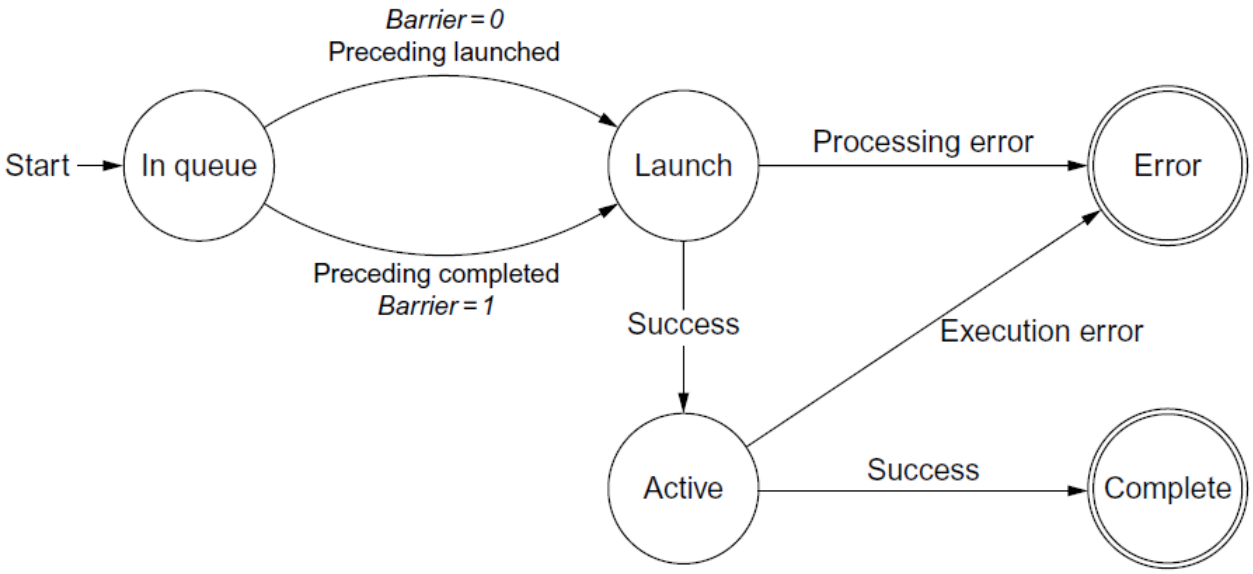


Slika 8: Primjer povezivanja korisnički definirane funkcije povratnog poziva s redom [12]

4.2.2. Obavijesti o vremenu izvođenja

HSA aplikacija izvještava o pogreškama i događajima pomoću obavijesti o vremenu izvođenja. HSA runtime definira dvije vrste obavijesti: **sinkronu** i **asinkronu**. Sinkrona obavijest koristi se radi indikacije da li je uspješno pozvana HSA runtime rutina izvršena. HSA runtime koristi povratne vrijednosti HSA runtime rutine za sinkrono prosljeđivanje obavijesti. HSA runtime definira statusni kôd kao enumeriranje (naredba **hsa_status_t)** kako bi se snimile povratne vrijednosti izvršene od strane HSA runtime rutine, izuzvši nekih API-ija indeksa reda ili vrijednosti signala. Obavijest je statusni kôd koji ukazuje na uspjeh ili pogrešku. **HSA_STATUS_SUCCESS** predstavlja status uspjeha, što je ekvivalentno nuli. Statusu pogreške dodijeljen je pozitivni integer (cijeli) broj, a njegov identifikator počinje s prefiksom **HSA_STATUS_ERROR**. Statusni kôd može pomoći kod utvrđivanja uzroka neuspješnog izvršenja. Pored naredbe hsa_status_t, HSA runtime također definira rutinu, **hsa_status_string**, za aplikacije koje traže dodatne informacije o objašnjenju povezane s kodom statusa. [12]

Kada HSA runtime otkrije pojavu asinkronih događaja ili pogrešaka, prosljeđuje asinkrone obavijesti pozivanjem odgovarajućih funkcija povratnog poziva koje omogućavaju aplikacije. Primjer je prikazan na Slici 9. [12]



Slika 9: Dijagram stanja paketa [12]

Slika 8 prikazuje zadatke koje aplikacija postavlja u red, asinkrono su konzumirane od strane procesora paketa. Kada je procesor paketa upućen da preuzme pakete iz queue1, ali ustanovi da je queue1 prazan, runtime otkriva ovu pogrešku i poziva funkciju povratnog poziva, **queue_callback**, sa statusnim kodom i pokazivačem na pogrešan red. Funkcija povratnog poziva queue_callback povezana je s queue1 kada se poziva **hsa_queue_create**. HSA runtime ne provodi nijedan osnovni/ defaultni povratni poziv. To jest, svi povratni pozivi su definirani od strane korisnika. Treba biti oprezan pri korištenju funkcija blokiranja u implementaciji povratnog poziva. Na primjer, povratni poziv koji se ne vraća može učiniti da se vrijeme izvođenja ne definira. [12]

4.2.3. Informacije o sustavu i HSA agentu

Prema specifikacijama sustava HSA platforme, HSA sustav može se implementirati kao **little-endian** ili **big-endian**. Model stroja može se implementirati kao **32-bitni sustav** ili **64-bitni sustav**. Zajednička virtualna memorija može se implementirati kao **osnovna** ili **memorija punog profila**. Da bi aplikacija mogla zatražiti implementacijski tip datog HSA sustava, HSA runtime definira **enumerator**, **hsa_system_info_t**, za sve atribute HSA sustava. HSA runtime također definira **rutinu**, **hsa_system_get_info**, kako bi aplikacija prikupila vrijednosti određenog atributa sustava. Važna osobina HSA agenta je da li je to **kernel agent ili ne**. Kako bi aplikacija mogla pristupiti HSA agentima u određenom sustavu, HSA runtime definira neprozirnu držač tipa **hsa_agent_t** koja predstavlja HSA agent i enumerator, **hsa_agent_info_t**, za atribute agenta. HSA runtime također definira rutinu, **hsa_iterate_agents**, koja aplikaciji omogućava prolazak kroz popis HSA agenata

koji su dostupni u sustavu. Pored toga, definira rutinu, **hsa_agent_get_info**, koja aplikaciji omogućava ispitivanje atributa specifičnih za HSA agente. Primjeri atributa specifičnih za agente: **ime, vrstu sigurnosnog uređaja** (CPU, GPU) i **podržane vrste čekanja**. Implementacija naredbe **hsa_iterate_agents** nužna je minimalno zbog prijavljivanje agenta glavnog računala (CPU). Aplikacije mogu pregledati/razmotriti atribut **HSA_AGENT_INFO_FEATURE** kako bi utvrdile da li je agent kernelskog tipa. HSA agent je kernelski agent ukoliko podržava skup HSAIL uputstava i podržava izvršavanje formata AQL kernelskog otpremnika paketa. Kernelski agent može otpremati naredbe bilo kojem drugom kernelskom agentu (uključujući i sam sebe) pomoću memorijskih operacija za izgradnju i pridruživanje AQL paketima. Kernelski agent sastoji se od jedne ili više računalnih jedinica (engl. *compute units*) i otkriva bogat niz atributa povezanih s odašiljanjem kernela, poput veličine valne fronte (engl. *wavefront size*) ili maksimalnog broja radnih predmeta (engl. *items*) u mreži. Moguće je da sustav uključuje agente koji nisu ni kernel agenti niti "domaćinski" CPU-ovi. Namjenski enkoderi/dekoderi i specijalizirani uređaji za enkripciju primjeri su ne-kernelskih agenata. [12]

4.2.4. Signali

HSA agenti mogu međusobno komunicirati pomoću koherentne globalne memorije ili pomoću signala. U specifikaciji sustava HSA platforme, svojstva i operacije signala su vrlo dobro definirani. Vrijednošću HSA signala moraju manipulirati samo kernel agenti koristeći posebne HSAIL mehanizme, a domaćinski CPU koristeći HSA mehanizme rada.

Potrebni mehanizmi za HSAIL i HSA runtime su:

• Alokacija HSA signala

• Uništenje HSA signal

• Čitanje trenutne vrijednosti HSA signala

• Slanje vrijednosti HSA signala

• Atomsko čitanje-modificiranje-pisanje na vrijednost HSA signala

• Čekanje HSA signala da zadovolji zadani uvjet

• Čekanje HSA signala da zadovolji zadani uvjet, uz zatraživanje maksimalnog trajanja čekanja

HSA runtime koristi signal neprozirnog držača **hsa_signal_t** koji predstavlja signalizaciju, a **hsa_signal_value_t** signal koji predstavlja pravu vrijednost. Pomoću mehanizma signalizacije neprozirnim držačem, vrijednošću signala može upravljati samo HSA runtime rutina ili HSAIL instrukcija, čime se udovolja ograničenjima njegove uporabe. HSA runtime definira rutinu, **hsa_signal_create**, za stvaranje HSA signala i rutinu, **hsa_signal_destroy**, za uništavanje HSA

signala. Budući da vrijednošću signala istovremeno može manipulirati više agenata, svaka operacija signala čitanja ili pisanja mora podržavati semantiku redoslijeda memorije. Moguća semantika određivanja memorije uključuje naredbe: **relaxed**, **release**, **acquire**, and **acquire-release**. HSA runtime definira rutine **hsa_signal_load_acquire** i **hsa_signal_load_relaxed** za atomično (engl. *atomically*) čitanje trenutne vrijednosti signala HSA. Za operaciju slanja vrijednosti HSA signala, ažuriranje vrijednosti signala ekvivalentno je slanju vrijednosti signala. HSA runtime definira rutine **hsa_signal_store_release** i **hsa_signal_store_release relaxed** za postavljanje vrijednosti HSA signala na atomičan način. Operacije atomičnog **read-modified-write** uključuju **AND**, **OR**, **XOR**, **Add**, **Subtract**, **Exchange**, and **CAS**. HSA runtime definira rutine za kombiniranje atomične read-modified-write operacije ažuriranja i narudžbe (engl. *ordering*) memorije za signale. Na primjer, **hsa_signal_add_release** je rutina inkrementiranja vrijednosti signala za određenu vrijednost/količinu s oslobađanjem ograde atomične memorije (engl. *atomic memory fence release*). Kombinacije akcija i naredbi memorije definirane u HSA runtime-u odgovaraju potrebnim HSAIL uputama. Kod operacije čekanja na HSA signal da ispuni određeni uvjet sa ili bez maksimalnog trajanja čekanja definirane su rutine **hsa_signal_wait_acquire** i **hsa_signal_wait_relaxed**. [12]

4.2.5. Redovi čekanja

Značajke, atributi, operacije i vste ili tipovi redova u specifikaciji sustava arhitekture HSA platforme dobro su opisani. Platforma koja podržava HSA također podržava i alokaciju višestrukih redova na razini korisnika. Redove čekanja na razini korisnika (ili kraće red) karakterizira alokacijsko vrijeme izvođenja, virtualna memorija određene veličine koja je dostupna (engl. *accessible*) na korisničkoj razini, te koja sadrži pakete definirane u arhitektonskom jeziku čekanja (engl. *architected queuing language*, skraćeno **AQL**), koji se nazivaju **AQL paketi**. Red čekanja povezan je s određenim (jednim) HSA agentom, ali HSA agent može biti povezan s više od jednog reda. HSA softver manipulira strukturama temeljenim na memoriji radi konfiguriranja hardverskih redova. To omogućuje učinkovito upravljanje softverom hardverskih redova HSA agenata. Memoriju reda obrađuje procesor paketa kao zvuk buffera, s odvojenim memorijskim lokacijama koje definiraju informacije o stanju čekanja i pisanja (engl. *write and read state information*) u tom redu. Paketni procesor je logično odvojeni agent. Njegova je glavna odgovornost učinkovito upravljanje redovima u ime odgovarajućeg kernel agenta. Poredak redova je definiran kao polu-neproziran objekt koji sadrži **vidljivi** i **nevidljivi dio**. Vidljivi dio uključuje vrstu, značajku i atribute reda čekanja. Nevidljivi dio sadrži indikatore čitanja/pisanja. Vrsta reda čekanja može biti **pojedinačno** ili **višestruko proizvedeni**. HSA vrijeme izvođenja definira enumerator,

hsa_queue_type_t, za sve vrste redova čekanja. Redovi čekanja mogu biti poslani od strane kernela ili agenta. **Kernelski redovi** koriste se za slanje kernela agentu, dok se **redovi agenata** koriste za slanje ugrađenih funkcija agentu. HSA vrijeme izvođenja definira enumerator, **hsa_queue_feature_t**, za značajke/potrebe redova. Atributi reda uključuju vrstu, značajku, bazičnu adresu, signal zvona na vratima (engl. *doorbell signal*), veličinu i identifikator. HSA runtime definira strukturu podataka, **hsa_queue_t**, za atribute reda. Redovi su *read-only* struktura podataka što se tiče korisničkog koda. Pisanje vrijednosti izravno u strukturu reda pomoću korisničkog koda rezultira nedefiniranim ponašanjem. Međutim, HSA agenti mogu izravno modificirati sadržaj *buffera* naznačenog osnovnom adresom. HSA agenti također mogu koristiti HSA runtime rutine za pristup *doorbell signalu* ili redu čekanja za slanje agenata. [12]

Operacije definirane za redove uključuju: **alokaciju redova**, **uništavanje reda**, **deaktivaciju reda** i **upravljanje read/write indeksom reda**. HSA runtime definira rutinu, **sa_queue_create**, za alokaciju HSA reda; rutina za uništavanje HSA reda, **hsa_queue_destroy**; i rutinu, **hsa_queue_inactive**, za postavljanje reda u neaktivno stanje. Razlika između neaktivnog stanja i uništenja je u tome što je svaka operacija uništenog reda nevaljana, ali neaktivna čekanja su valjana. Indeksi čitanja i pisanja reda čekanja ne mogu biti izravno izloženi korisničkom kodu. Umjesto toga, korisničkom kodu može se pristupiti putem indeksa čitanja/pisanja redova ali samo uz upotrebu namjenskih HSA runtime rutina. HSA runtime definira rutine za: **kombiniranje operacija indeksa čitanja/pisanja**, **uključujući učitavanje**, **pohranu**, **dodavanje** i **CAS** te **memorijske narudžbe**. Na primjer, **hsa_queue_store_write_index_release** je rutina dodjeljivanja zadane vrijednosti indeksu pisanja s redoslijedom prosljeđivanja memorije (**store-write-index-release**). [12]

4.2.6. Architected queuing language

Architected queuing language (skraćeno **AQL**) pruža standardno binarno sučelje za slanje naredbi agenta. AQL omogućava HSA agentima da izrade i izvršavaju vlastite pakete naredbi, omogučavajući brzu *low-power* otpremu. AQL također pruža podršku za slanje reda čekanja kernelskih agenata. AQL paket je međuspremnik (engl. *buffer*) korisničkog načina s određenim formatom koji kodira jednu naredbu. HSA runtime ne pruža rutine za stvaranje, uništavanje ili manipuliranje AQL paketima. Umjesto toga, aplikacija koristi alokatore na razini korisnika (npr. **malloc**) za stvaranje paketa i izvodi redovitu operaciju memorije za pristup sadržaju paketa. Aplikacijama nije potrebno izričito rezervirati prostor za pohranu paketa, jer redovi čekanja već sadrže međuspremnik naredbi u koji se mogu upisati AQL paketi. [12]

U specifikaciji sustava HSA platforme postoji šest vrsta AQL-a paketa:

- Kernel otpremni paketi
- Paketi agenta za otpremu
- Barrier-AND paketi
- Barrier-OR paketi
- Paketi specifičan za proizvođača
- Nevažeći paketi

HSA runtime koristi enumerator, **hsa_packet_type_t**, za enumeriranje svih vrsta AQL paketa; struktura podataka, **hsa_kernel_dispatch_packet_t**, koristi se za definiranje formata kernelskih otpremnih paketa; struktura podataka, **hsa_agent_dispatch_packet_t**, služi za definiranje formata otpreme paketa agenata; struktura podataka, **hsa_barrier_add_packet_t**, za definiranje formata **barrier-AND** paketa; i struktura podataka, **hsa_barrier_or_packet_t**, za format paketa **barrier-OR**. Svi formati paketa imaju zajedničko zaglavlje, **hsa_packet_header_t**, koje opisuje njihov tip, **barrier bit** (koji prisiljava procesor paketa na redoslijedno dovršavanje paketa) i druga svojstva. Aplikacija koristi kernelski paket za slanje kako bi poslala kernel kernelskom agentu; aplikacija koristi dispečerski paket agenata za pokretanje ugrađenih funkcija u HSA agentu. Barrier-AND paket omogućuje aplikaciji specifikaciju do pet zavisnosti o signalu (engl. *signal dependencies*) i zahtijeva da procesor paketa riješi te odnose zavisnosti prije nego što nastavi dalje sa izvođenjem. Paketni procesor neće pokretati ijedan sljedeći paket u tom redu sve dok barrier-AND paket nije dovršen. Barrier-AND paket je dovršen kada su svi ovisni signali sa vrijednošću 0 primijećeni (nakon pokretanja barrier-AND paketa). Barrier-OR paket je sličan barrier-AND paketu, ali postaje potpun tek kada procesor paketa primijeti da bilo koji od ovisnih signala ima vrijednost 0. Format paketa za pakete koji se odnose na pojedinog proizvođača je definiran od strane proizvođača. Ponašanje procesora paketa za paket specifičan za proizvođača je specifično za određenu implementaciju, ali ne smije uzrokovati eskalaciju privilegija ili ispadanje iz konteksta procesa. Format paketa za sve unose u redove je postavljen na nevažeći (engl. *invalid*) kada se red inicijalizira. Kad god se učita unos reda čekanja, njegov format paketa je postavljen na **nevaljan** i **indeks čitanja** (engl. *read index*) **reda se povećava**. [12]

Nakon slanja paketa, paket može biti u jednom od sljedećih pet stanja:

- **čekanja u redu** (engl. *in queue*),
- **pokretanja** (engl. *launch*),
- **aktivnom** (engl. *active*),

- **potpunom/kompletiranom** (engl. *complete*)
- **ili stanju pogreške** (engl. *error*).

Paket je u "*in queue*" stanju ukoliko procesor paketa nije započeo s raščlanjivanjem (engl. started to parse) paketa.

Paket je u "*launch*" stanju ukoliko se paket analizira, ali nije započeo s izvršavanjem.

Paket je u "*active*" stanju ukoliko je pokrenuto izvršavanje paketa.

Paket je u "*complete*" stanju ukoliko se primijeni ograda za oslobađanje memorije s opsegom naznačenim poljem opsega ograde u zaglavlju, a "*complete*" signal (signal završetka, ukoliko postoji) se umanjuje.

Paket se nalazi u "*error*" ukoliko je došlo do pogreške tijekom faza pokretanja ili aktivne faze. Kada paket uđe u stanje pogreške, daljnji paketi neće se pokretati iz reda čekanja. Red čekanja se ne može povratiti. Red čekanja može se samo deaktivirati (ukoliko ga dijele višestruki procesi) ili uništiti (ako ga ne dijele drugi procesi). Na Slici 9 prikazan je dijagram stanja paketa. [12]

HSA runtime definira dva koda pogreške, jedan za fazu pokretanja i drugi za aktivnu faza. U fazi pokretanja, kod pogreške **HSA_STATUS_ERROR_INVALID_PACKET_FORMAT** koristi se za obavještavanje korisnika da je AQL paket nepravilno oblikovan. Kôd pogreške **HSA_STAUS_ERROR_OUT_OF_RESOURCES**, koristi se kako bi obavijestio korisnike da sustav ne može dodijeliti/alocirati dovoljno resursa za paket. Tijekom aktivne faze koristi se kôd pogreške **HSA_STATUS_ERROR_EXECPTION**, koji informira korisnike da je tijekom izvršenja kernela pokrenut HSAIL izuzetak. Dijeljenje s nulom je dobar primjer navedene situacije. Postoji pet vrsta izuzetaka. Za više informacija preporučljivo je pogledati odjeljak **PRM unutar HSA dokumentaicje o iznimkama hardvera**. [12]

4.2.7. Memorija

Važna funkcionalnost HSA vremena izvođenja je mogućnost usluga upravljanja memorijom HSA agenata. Područje HSA memorije (engl. *HSA memory region*, ili skraćeno na hrv. područje) predstavlja blok virtualne memorije koji je izravno dostupan putem HSA agenata. Prethodno spomenuto HSA područje izlaže svojstva o bloku virtualne memorije i kako joj se pristupa od strane određenog HSA agenta. Područje HSA memorije podijeljeno je u četiri različita segmenta: **globalni**, **read-only**, **grupni** i **privatni**. HSA runtime definira regiju objekata, **hsa_region_t**, za deklaraciju regije, enumerator, **hsa_region_segment_t**, za sve segmente kojima je moguće pristupiti pomoću HSA regije, **hsa_region_info_t**, za sve atribute povezane s regijom. Regija može biti povezana s više od jednog agenta. HSA runtime definira rutinu, **hsa_agent_iterate_regions**, za aplikaciju koja

vrši inspekciju skupa regija povezanih s HSA agentom. Također definira rutinu, **hsa_region_get_info**, za aplikaciju koja ukazuje na trenutne vrijednosti atributa HSA regije. [12]

Registracija je performabilni naputak koji omogućava implementaciji HSA runtime-a da sazna kojim međuspremnicima će pristupiti neki od kernelskih agenata te time omogućava pravovremeno izvršavanje odgovarajuće optimizacije. Kada se kerneli izvršavaju (na kernelskim agentima s podrškom *Potpunih profila)* oni mogu pristupiti bilo kojem redovitom pokazivaču domaćina (engl. *regular host pointer*), registrirani međuspremnik može rezultirati poboljšanim performansama pristupa. U agentima koji podržavaju samo *Base profil*, fino-zrnata semantika ograničena je na međuspremnike alocirane korištenjem HSA runtime rutime **hsa_memory_allocate**, koja se koristi za dodjelu memorije u globalnim i read-only segmentima. Korisnik može koristiti samo memoriju alociranu iz fino-zrnate regije za prosljeđivanje argumenata kernelu. Fino-zrnatoj memoriji mogu istovremeno pristupiti mnogi HSA agenti u sustavu (pod uvjetima definiranim HSA memorijskim modelom). Grubo-zrnatoj memoriji također mogu pristupiti mnogi HSA agenti u određenom vremenskom intervalu, ali samo jedan agent u datom trenutku. HSA runtime definira rutinu, **hsa_memory_assign_agent**, koja omogućava aplikaciji da eksplicitno dodijeli vlasništvo nad međuspremnikom određenom agentu. [12]

Read-only segment koristi se za pohranu konstantnih/stalnih informacija. Sadržaj read-only međuspremnika može se inicijalizirati ili izmijeniti koristeći HSA runtime rutinu **hsa_memory_copy**. Regije povezane s read-only segmentima privatne su za agente kernela. Prolazak kroz read-only međuspremnik povezan s jednim agentom u kernelskom paketu za slanje, koji je poslan na izvršavanje od strane različitih agenata rezultira nedefiniranim ponašanjem. Kernelski agenti mogu obavljati samo operacije čitanja na adresama varijabli koje se nalaze u njihovim vlastitim read-only segmentima. Sadržaj read-only segmenata postojan je kroz cijeli životni vijek aplikacija. [12]

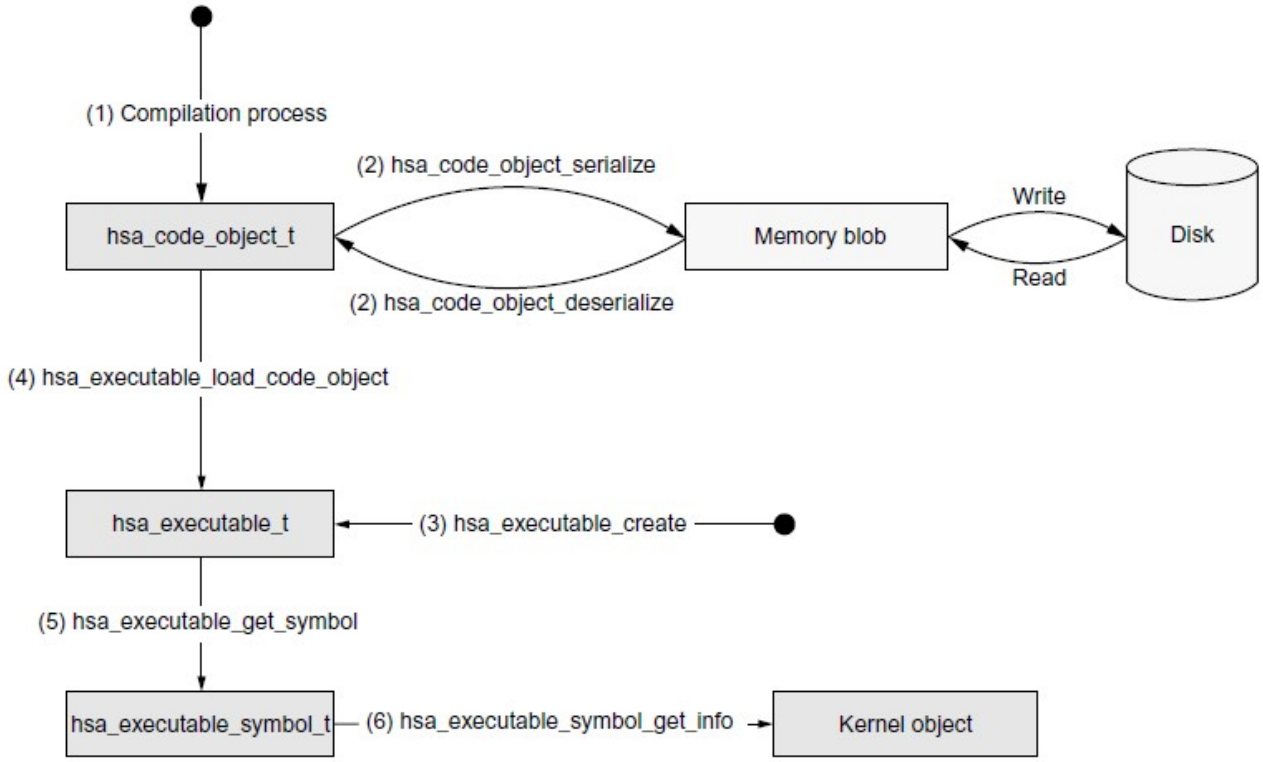
Segment grupe koristi se za pohranu podataka dijeljenih između svih radnih jedinica (engl. *work items*) koje se nalaze u istoj radnoj skupini. Varijabilnost u grupnoj memoriji očituje se u operacijama čitatanja i pisanja koje može izvršiti bilo koja radna jedinica unutar iste radne skupine s kojom je grupna memorija povezana, ali ne i od strane radnih jedinica unutar drugih radnih skupina ili drugih agenata. Grupna memorija je "živa" tijekom izvršavanja radnih jedinica u radnoj skupini kernelskog dispečera s kojom je povezana, a postaje neinicijalizirana kada radna skupina započne sa svojim izvršenjem. [12]

Privatni segment koristi se za lokalnu pohranu podataka u radnu jedinicu. Privatna memorija je vidljiva samo jednoj radnoj jedinici kernelskog dispečera. Varijabilnost u privatnoj memoriji očituje se u operacijama čitanja i pisanja koje može izvršiti samo radna jedinica s kojom je povezana, ali ne i bilo koja druga radna jedinica ili agent. Privatna memorija je postojana tokom cijelog postupka izvođenja radne jedinice s kojom je povezana, a postaje neinicijalizirana kada radna jedinica započne sa svojim izvršenjem. [12]

Upotreba memorije u grupnim i privatnim segmentima slična je onoj u globalnim te *read-only* segmentima. Svaki kernelski agent razotkriva grupnu i privatnu regiju. Međutim, korisniku nije dopušteno eksplicitno/izričito raspoređivanje memorije u ovim regijama pomoću **hsa_memory_allocate**, niti može kopirati bilo koji sadržaj u nju koristeći **hsa_memory_copy**. S druge strane, korisnik mora odrediti količinu grupne i privatne memorije koja se mora dodijeliti za određeno izvršenje kernela popunjavanjem polja **group_segment_size** i **private_segment_size** u dispečerskom paketu kernela. Stvarna alokacija grupne i privatne memorije događa se automatski prije nego li se započne s pokretanjem kernela. [12]

4.2.8. Kodni ciljevi i izvršenja

Kada se aktivira dispečerski paket kernela, kernelski objekt mora biti specificiran. Kernelski objekt je "premosnica" strojnom kodu koji se izvršava. Izrada kernelskog objekta sastoji se od dvije faze. U prvoj fazi se **kernelski izvor sastavlja ili finalizira do reprezentativnog ciljanog stroja koji se naziva kodni objekt**. U drugoj fazi, **kodni objekt se učitava u HSA runtime objekt koji se naziva "izvršni"**. HSA vrijeme izvođenja definira strukture podataka **hsa_code_object_t** i **hsa_executable_t** koji predstavljaju premosnicu objekta koda odnosno izvršnu ručicu (engl. *executable handle*). Kernelski objkt može se dobiti izvođenjem upita na izvršnom objektu pomoću HSA runtime rutine **hsa_executable_get_symbol**, **hsa_executable_iterate_symbols** i **hsa_executable_symbol_get_info**. Na Slici 10 prikazan je tok dohvaćanja držača kernelskog objekta iz kodnog objekta. [12]



Slika 10: Dohvaćanje držača kernelskog objekta iz objekta koda [12]

Tok se sastoji od sljedećih koraka:

1) **Korak generiranja kodnog objekta:** U ovom koraku, program se sastavlja u jedan ili više HSAIL modula koji sadrže kernel koji će se izvršavati. HSA runtime pruža proširenja HSAIL rutine finalizacije za korisnike kako bi mogli stvarati i finalizirati HSAIL module u ciljni kod strojnog objekta. Drugim riječima, to omogućuje korisnicima stvaranje objektnog koda. Ručke kodnog objekta (**hsa_code_object_t**) su reprezentativni primjerci ciljanih specifičnih uređaja (engl. *target-machine-spscific representations*) koji sadrže kôd za skup kernela i neizravnih funkcija.

2) **Korak serializacije/deserializacije kodnog objekta:** Ovaj korak omogućuje aplikaciji čitanje kodnih objekata generiranih u izvanmrežnom procesu kompilacije (engl. *offline compilation process*) ili zapisivanje kodnih objekata generiranih u 1. koraku za kasniju upotrebu. HSA runtime pruža rutine, **hsa_code_object_serialize** i **hsa_code_object_deserialize**, za omogućavanje aplikacijama da obave operacije serializacije i deserializacije. Svaki je kodni objekt povezan s više simbola (**hsa_code_symbol_t**), od kojih svaki predstavlja varijablu, kernel ili neizravnu funkciju u originalnom izvornom programu.

3) **Korak generiranja izvršnog držača** (engl. *executable handle generation step*): U ovom se koraku stvara prazni izvršni držač tipa **hsa_executable_t** pozivanjem HSA runtime rutine **hsa_executable_create**. HSA runtime će u sljedećem koraku upotrijebiti izvršni držač za učitavanje skupa držača kodnog objekta, eventualno povezanih s različitim ciljanim strojevima.

4) **Korak učitavanja kodnog objekta:** U ovom koraku, kod objekta može se dodati (ili učitati) u izvršnu obradu pozivanjem HSA runtime rutine **hsa_executable_load_code_object**.

5) **Korak dohvaćanja izvršnog simbola:** U ovom koraku se izvršni simbol koji odgovara datom kernelu i agentu može dohvatiti pozivanjem HSA runtime rutine **hsa_executable_get_symbol**. Izvršni simboli koji predstavljaju kernele otkrivaju atribut **HSA_EXECUTABLE_SYMBOL_INFO_KERNEL_OBJECT**, što je držač strojnog koda koji se u konačnici koristi za pokretanje kernela.

6) **Korak dohvaćanja kernelskog objkta:** U ovom koraku, rukohvat kernelskog objekta povezan s izvršnim simbolom može se dohvatiti pozivanjem HSA runtime rutine **hsa_executable_symbol_get_info**. [12]

4.3. Produženo HSA vemena izvođenja

Iako proširenje API-ja za HSA runtime nije obvezan dio kompatibilih implementacije, proizvođači će vjerojatno podržati proširenja odobrena od strane HSA Fondacije. U ovom poglavlju će se detaljno opisati dva proširenja odobrena od strane HSA, a to su **HSAIL finalizacija** i **slike**. [12]

4.3.1. HSAIL finalizacija

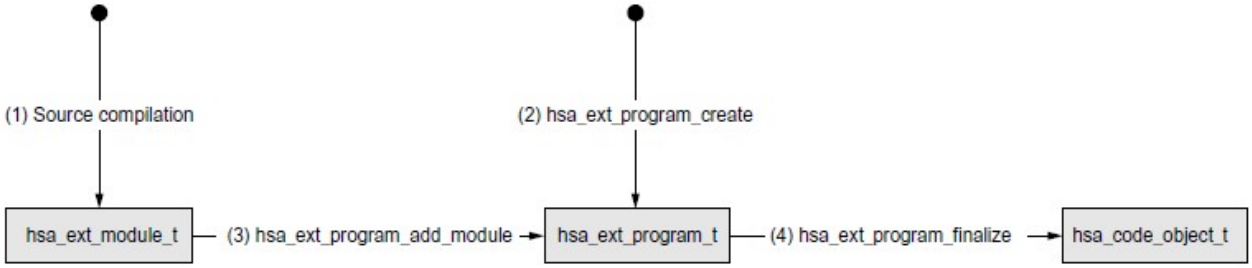
Svrha HSAIL finalizacijskih rutina je finaliziranje skupa (za vrijeme izvođenja) HSAIL modula u binarnom formatu (BRIG) u kod za ciljane specifične uređaje. Ciljani strojni kod predstavljen je kao kodni objekt. Na Slici 6 prikazan je takav finalizacijski tijek koji se sastoji od sljedećih koraka:

1. **Korak sastavljanja izvora:** U ovom se koraku program sastavlja u jedan ili više HSAIL modula. Jedan od HSAIL modula sadrži kernel od interesa. Ovaj se korak izvodi izvan HSA vremena izvođenja.

2. **Korak kreiranja HSAIL-ovog programa:** U ovom koraku, prazna HSAIL programska ručica **hsa_ext_program_t** kreirana je pozivanjem HSAIL finalizacijske rutine **hsa_ext_program_create**.

3. **Korak umetanja HSAIL modula:** U ovom koraku, HSAIL moduli dodaju se na držač HSAIL programa koji je stvoren u koraku 2 pomoću HSAIL finalizacijske rutine **hsa_ext_program_add_module**.

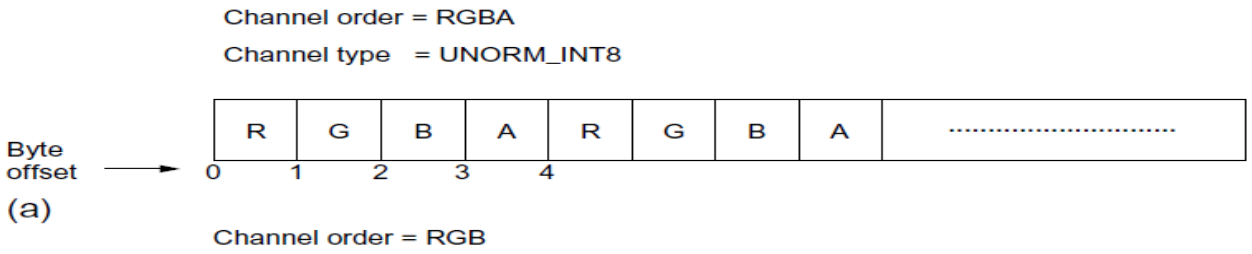
4. **HSAIL-ov finalizacijski korak:** Nakon što su svi HSAIL moduli dodani na odgovarajući držač HSAIL programa, HSAIL program može se dovršiti uporabom HSAIL finalizacijske rutine, **hsa_ext_program_finalize**, koja stvara kvaku kodnog objekta. Ručka objektnog koda može se serializirati na disk (izvanmrežna kompilacija) ili dalje obraditi u svrhu pokretanja (mrežna kompilacija). Daljnji scenarij prikazan je na Slici 11. [12]



Slika 11: Od izvora do kodnog objekta. [12]

4.3.2. Slike i uzorci

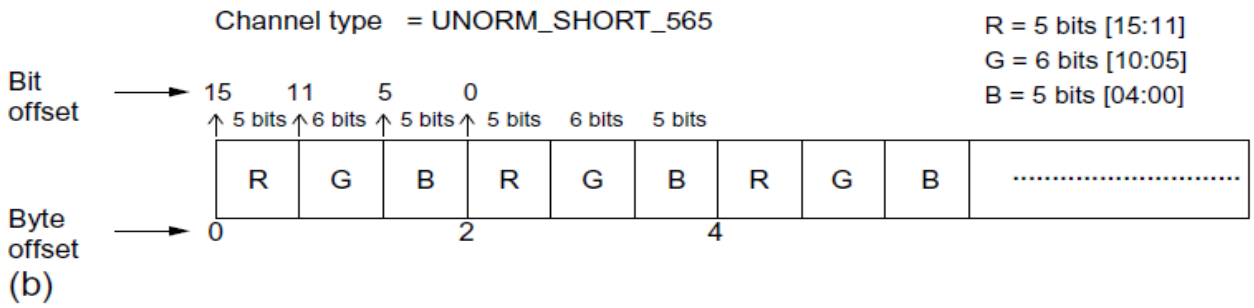
Slikovna slika (engl. *picture image*, ili skraćeno na hrv. slika) je značajka koja se proizvoljno pruža na HSA podržanim platformama. Ova značajka omogućuje HSA aplikacijskom softveru definiranje i korištenje slikovnih objekata. U HSA runtime-u, slika je objekt definiran kao neprozirni 64-bitni držač slike (**hsa_ext_image_t**), koji se može kreirati i biti uništen od strane HSA runtime proširenih rutina **hsa_ext_image_create** i **hsa_ext_image_destory**. Okvir slike odnosi se na slikovne podatke u memoriji i pohranjuje informacije o rasporedu resursa i drugim svojstvima. Primjeri rasporeda slika prikazani su na naredne dvije slike. [12]



Slika 12: a) primjer izgleda slike u memoriji [12]

Na Slici 12 prikazan je izgled slikovnog resursa tipa **UNORM_INT_8** (normalizirani 8-bitni nepotpisani tip) i redoslijedom **RGBA kanala**. [12]

Na Slici 13 prikazan je izgled slikovnog resursa s kanalom tipa **UNORM_SHORT_565** i redoslijedom RGB kanala. Na Slici 12 vrijednost R, G, B ili A boja je predstavljena jednim bajtom (8 bita). Za vrstu kanala prikazanu na Slici 13, vrijednost R, G ili B u boji predstavljena je sa 5, 6 ili 5 bita. HSA razdvaja pohranu slikovnih podataka i njihov interpretacijski opis. To omogućuje aplikaciji kontrolu nad mjestom pohrane slikovnih podataka i učinkovitije upravljanje memorijom. [12]



Slika 13: B) primjer izgleda slike u memoriji [12]

Da bi se opisao slikovni objekt, HSA produženo vrijeme izovđenja definira:

- enumerator, **hsa_ext_geometry_t**, za dimenzije slike dopuštene u HSA;
- strukturu podataka tipa **hsa_ext_image_format_t** koji predstavlja format slike za opisivanje redoslijeda kanala (npr. *RGB, sRGBA, ARGB*)
- vrstu kanala (npr. **UNORM_SHORT_565**, **UNORM_INT8**, **UNORM_INT16**);
- te podatkovnu strukturu tipa **hsa_ext_image_descriptor_t** za opis dimenzija slike, veličine podataka i formata slike.

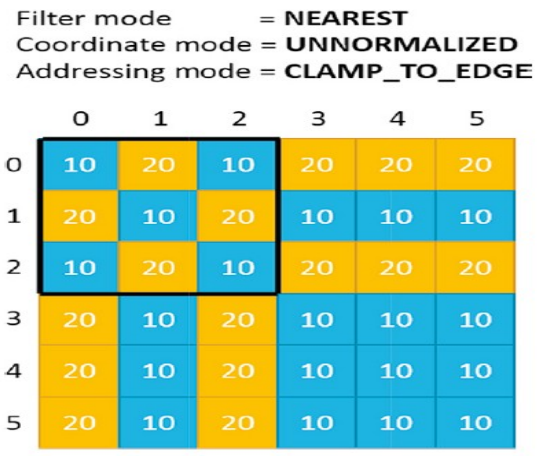
Proširenje HSA vremena izvođenja sadrži rutinu, **hsa_ext_image_get_info**, koja aplikaciji omogućava određivanje specifične veličine podataka i poravnanje slike. Aplikacija može ostvariti mogućnost formatiranja slike pomoću proširenja HSA rutine **hsa_ext_image_get_capability**. Ova rutina omogućava sposobnost kombiniranja određene geometrije i slikovnog formata definiranih pomoću **hsa_ext_image_capability_t**. Slikovne operacije, za izvoz i uvoz slikovnih podataka s linearnim rasporedom u memoriji definirane su proširenim HSA runtime rutinama **hsa_ext_image_export** i **hsa_ext_image_import**. Proširenje HSA runtime-a također definira **rutinu hsa_ext_image_copy** za kopiranje dijela slike na drugu sliku i rutinu **hsa_ext_image_clear** za uklanjanje slike. [12]

Proširenje HSA vremena izvođenja koristi:

- strukturu podataka tipa, **hsa_ext_sampler_descriptor_t**, za definiranje uzorka;
- enumerator, **hsa_ext_sampler_coordinate_mode_**t, za nabrajanje načina koordinata uzorkovanja;
- enumerator, **hsa_ext_sampler_addressing_mode_t**, za nabrajanje načina adresiranja uzorka;
- enumerator, **hsa_ext_sampler_filter_mode_t**, za nabrajanje načina filtriranja uzorka;
- rutinu, **hsa_ext_sampler_create** za izradu držač uzorka;
- i rutinu, **hsa_ext_sampler uništiti**, kako bi uništio držač uzorka.

Držači slike i uzorka opisani su u slikovnim uputama, poput **rdimage**, **ldimage** i **stimage**, koje su definirane u HSA priručniku za programere (engl. *HSA programmer's reference manual*). Naredba rdimage koristi koordinate, držač slike i držač uzorkovača za referenciranje na sliku i učitavanje vrijednosti podataka kanala (engl. *load the channel data value*), kao što su crvena, zelena, plava i alfa, s referencirane slike. Kada je koordinata predviđena za rdimage naredbu izvan granica, rdimage se odnosi na držač uzorka i određuje kako definirati vrijednost podataka (vezanog uz kanal) prema postavkama koordinatnog načina, koordinatnog načina adresiranja te postavkama filtriranog načina koordiniranja. Naredba ldimage također koristi držač slike za referenciranje slike i učitavanje vrijednosti podataka o kanalu. Međutim, izvorni operand ldimage naredbe ne sadrži držač uzorka. Kada je koordinata predviđena za ldimage izvan granice, ponašanje ldimage naredbe nije definirano. Naredba stimage koristi se držačem slike, zajedno s koordinatom, za referenciranje slike i ponovno spremanje vrijednosti podataka o kanalu na sliku. Budući da izvorni operand stimage naredbe ne sadrži držač uzorkovača, ponašanje ove naredbe je također nedefinirano kada je koordinata predviđena za stimage izvan granica. Dodatne specifikacije povezane s uputama za slike mogu se pronaći u "HSA Programmer's Reference Manual". [12]

Na Slici 14 dan je primjer interpretacije uzorkovača s normiranim načinom



Slika 14: Primjer interpretacije slikovnih podataka pomoću uzorkovača [12]

koordinata, načinom adresiranja **clamp_to_edge** i zadanim najbližim načinom filtriranja. U primjeru, pretpostavljamo da je granica slike (crna zadebljana linija) između indeksa 2 i indeksa 3. Uz učinak clamp_to_edge načina adresiranja, slika će ponoviti vrijednost vanjskog ruba (20 ili 10) za vanjske koordinate.

4.4. HSA API za vrijeme izvođenja – zaključno

U ovom smo poglavlju opisali neke od najvažnijih HSA runtime rutina kao i tipove podataka koji su osmišljeni za podršku operacijama koje zahtijevaju specifikacije arhitekture HSA platforme i pokretanje izvođenja kernela na odgovarajućim HSA agentima. Opisali smo HSA-odobrene, rutine za produženo vrijeme izvođenja povezanih s HSAIL finalizacijom i slikama. Svrha ovog poglavlja bila je pružiti čitatelju idejni/konceptualni temelj za razumijevanje HSA runtime API-ja. Čitatelju se preporuča pretraživanje HSA runtime specifikacije radi detaljnihih pojašnjenja o HSA bazičnim značajkama i/ili značajkama HSA proširenja. Također, proizvočima HSA platformi je dozvoljeno da u svojim sustavima omoguće proširenja HSA vremena izvođenja. [12]

5. Radeon Open Compute (ROC)

ROCm je računalna platforma otvorenog koda (engl. *open source*) koja se koristi za razvoj, otkivanje i edukaciju temeljenu na računanju na GPU-ima. Ova platforma je snažan temelj naprednog računanja koji se bazira na integraciji CPU i GPU jedinice s ciljem rješavanja stvarnih, svakodnevnih problema. Dana 25. travnja 2016. godine, isporučena je prva verzija platforme ROCm 1.0 koja se zasniva na tri osnovna preduvjeta:

1) **Heterogenoj računalnoj platformi otvorenog koda** (Linux upravljački program i runtime – engl. *Linux driver and runtime stack*) koja je optimizirana za HPC i računarstvo zasnovano na izrazito skalabilnim sustavima (engl. *ultra-scale class computing*[12]);

2) **Heterogenom jedno-izvorskom programu kompajleru programskih jezika C i C++** (engl. *heterogeneous C and C++ single source compiler*); kako bi se računarstvu pristupilo na holistički, cjeloviti način, na razini sustava, umjesto pristupa GPU-u kao diskretnom, odvojenom artefaktu;

3) **HIP** (skraćeno od engl. *heterogeneous-compute interface for portability*[13]), imajući na umu potrebu slobode izbora kada je riječ o platformama i API-jima za GPU računarstvo.

Upotrebom ROCm 1.0 platforme uz potrebno znanje o HSA standardima te naravno uz znanje o HSA vremenu izvođenja (engl. r*untime*), njezini autori su postigli proširenje podrške za dGPU-u (skraćeno od engl. d*iscrete GPU*) s kritičnim značajkama za ubrzanje NUMA (skraćeno od engl. *non-uniform memory access*[14]) izračunavanja. Kao rezultat ostvarenog, ROCK upravljački program (engl. *driver*) sastoji se od nekoliko komponenti koje se temelje na naporima njegovih autora (ROCm razvojnog tima) na razvoju HSA za **APU**-e[15], uključujući: novi upravljački program **AMDGPU**, **KFD** (skraćeno od engl. *kernel*[16] *fusion driver*[17]), **HSA+ Runtime** i **LLVM** (skraćeno

12 **Ultra-scale class computing** - jest izraz koji se koristi u područjima vezanim uz računalnu znanost (engl. c*omputer science*), programsko/softversko inženjerstvo (engl. s*oftware engineering*) i sistemski inženjering (engl. s*ystem engineering*), a odnosi se na softverski intenzivne sustave koji podržavaju neviđenu količinu hardvera, linija izvornog koda, broja korisnika i količinu podataka. [14]

13 **HIP** – C++ runtime API (skraćeno od engl. *Application Programming Interface*) i kernelski jezik koji programerima omogućava izradu prijenosnih aplikacija koje se mogu izvodniti na AMD-ovim i ostalim GPU-ima. Upotrebom HIP pristupa, programerima se omogućava korištenje osnovnih hardverskih značajki i maksimizacija performansi aplikacija koje se izvode na GPU hardveru. [15]

14 **NUMA** – jest dizajn računalne memorije namijenjene više-procesnoj (enlg. *multiprocessing*) obradi pri čemu vrijeme pristupa memoriji ovisi o relativnoj udaljenosti memorijske lokacije naspram procesora. [16]

15 **APU** – procesna jedinica (APU), još ranije poznata pod nazivom Fusion, marketinški je termin za seriju 64-bitnih mikroprocesora (engl. m*icroprocessors*) tvrtke AMD (skraćeno od engl. *Advanced Micro Devices*), dizajniranih na način da se CPU i GPU ponašaju kao jedinstvena jedinica/sustav. [17]

16 **Kernel** – računalni program koji čini jezgru operacijskog sustava računala, koji ima potpunu kontrolu nad svime u sustavu. [18]

17 **AMDKFD** – AMD *kernel fusion driver*, generalno gledano jest AMD-ova HSA računalna platforma upravljačkog programa sa kernel programom. AMDKFD je potreban za rad s računarskim komponentama ROCm/OpenCL

od engl. *low level virtual machine*[18]) kompilacijski skup koji pruža podršku za ključne programske jezike. Ova podrška započela je AMD-ovom „Fiji" obitelji dGPU-ova, a zatim se proširila uključivši „Hawaii" dGPU familiju sa ROCm 1.2. i ROCm 1.3. te nadalje podrška uključuje „Polaris" obitelj ASIC-ova (skraćeno od engl. *application-specific intergrated circuit*[19]). ROCm platforma se neprestano nadograđuje i proširuje kako bi obuhvatila nove AMD ASIC-ove i pružala veću funkcionalnost usprkos velikom porastu niza osnovnih upravljački programa (engl. b*ase drivers*), alata i knjižnica usmjerenih na računanje. [22]

5.1. Biblioteka rocRAND

5.1.1. Općenito o biblioteci rocRAND

Biblioteka rocRAND je projekt koji korisniku omogućuje funkcije za generiranje "**pseudo-slučajnih**" i "**kvazi-slučajnih**" **brojeva** (engl. *pseudo-random and quasi-random numbers*). Biblioteka rocRAND implementirana je u programskom jeziku HIP i optimizirana je za **najnovije AMD-ove diskretne** (engl. *discrete*) **GPU-ove**. Biblioteka je dizajnirana za rad uz pomoć AMD Radeon Open Compute ROCm runtime-a, ali isto tako biblioteka se može izvoditi i na GPU-ima koji podržavaju CUDA platforme. Dodatno, projekt uključuje i „omotnu biblioteku" (engl. *wrapper library*) zvanu hipRAND koja korisniku omogućava jednostavan prijenos CUDA aplikacija (koje koriste cuRAND biblioteku) u sloj HIP. U okruženju ROCm biblioteka hipRAND koristi rocRAND, dok se u CUDA okruženju koristi cuRAND biblioteka. [23]

5.1.2. Podržani ROCm generatori slučajnih brojeva

- **XORWOW**
- **MRG32k3a**
- **Mersenne Twister** za grafičke procesore (**MTGP32**)
- **Philox** (4x32, 10 krugova)
- **Sobol32** [23]

korisničkog prostora (engl. *user-space*), a u novijim izdanjima kernela pokazalo se da pored AMD APU-ija, AMDKFD dobro funkcionira i u slučaju diskretnih grafičkih kartica (engl. dGPU). [19]

18 **LLVM** – računalna biblioteka koja se koristi za konstrukciju, optimizaciju i izgradnju srednjeg (engl. *intermediate*) i/ili binarnog računalnog koda. [20]

19 **ASIC** – je integrirani krug prilagođen određenoj uporabi umjesto da je namijenjen općoj namjeni (npr., čip dizajniran za pokretanje digitalnog diktafona ili rudar/miner bitcoina visoke učinkovitosti jesu vrste ASIC-a). [21]

5.1.3. Primjer: benchmark_rocrand_generate.cpp

Kako bi se koristio domaćinski API, korisnički kôd treba sadržavati datoteku zaglavlja biblioteke **<rocrand.h>** i dinamički se povezivati s rocRAND bibliotekom. Biblioteka rocRAND koristi se ROCm vremenom izvršavanja, pa korisnički kôd također mora koristiti ROCm vrijeme izvođenja. API za rocRAND ne podržava upravljački program rocRAND-a. Nasumični (engl. random) brojevi kreiraju se uz pomoć generatora. rocRAND generator objedinjuje sva unutarnja stanja neophodna za stvaranje niza pseudo slučajnih ili kvazi slučajnih brojeva. Uobičajeni redoslijed operacija pri genereiranju slučajnih brojeva opisan je u nastavku na primjeru rocRAND koda.
Kod **benchmark_rocrand_generate.cpp** programa: [24]

Po želji generiramo više slučajnih brojeva s više poziva funkcije **generate_func(generator, data, size):**

```
template<typename T>
using generate_func_type = std::function<rocrand_status(rocrand_generator, T *, size_t)>;

template<typename T>
void run_benchmark(const cli::Parser& parser,
                   const rng_type_t rng_type,
                   generate_func_type<T> generate_func)
{
        const size_t size0 = parser.get<size_t>("size");
        const size_t trials = parser.get<size_t>("trials");
        const size_t dimensions = parser.get<size_t>("dimensions");
        const size_t size = (size0 / dimensions) * dimensions;
        T * data;
```

Dodijeljivanje memorije na uređaj uz pomoću naredbe **hipMalloc():**

```
                HIP_CHECK(hipMalloc((void **)&data, size * sizeof(T)));
```

Stvaranje novog generatora željenog tipa uz pomoć naredbe **rocrand_create_generator:**

```
                rocrand_generator generator;
                ROCRAND_CHECK(rocrand_create_generator(&generator, rng_type));
```

Postavljanje opcije generatora na seed, offset ili order vrijednost; npr. koristimo naredbu za generiranje kvazi nasumičnog broja **rocrand_set_quasi_random_generator_dimensions:**

```
rocrand_status status = rocrand_set_quasi_random_generator_dimensions(generator, dimensions);
```

Upotreba rezultata:

```
        if (status != ROCRAND_STATUS_TYPE_ERROR) // If the RNG is not quasi-random
{
        ROCRAND_CHECK(status);
}
// Warm-up
for (size_t i = 0; i < 5; i++)
{
        ROCRAND_CHECK(generate_func(generator, data, size));
}
HIP_CHECK(hipDeviceSynchronize());
// Measurement
auto start = std::chrono::high_resolution_clock::now();
for (size_t i = 0; i < trials; i++)
{
        ROCRAND_CHECK(generate_func(generator, data, size));
}
HIP_CHECK(hipDeviceSynchronize());
auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double, std::milli> elapsed = end - start;

std::cout << std::fixed << std::setprecision(3)
          << "       "
          << "Throughput = "
          << std::setw(8) << (trials * size * sizeof(T)) /
              (elapsed.count() / 1e3 * (1 << 30))
          << " GB/s, Samples = "
          << std::setw(8) << (trials * size) /
              (elapsed.count() / 1e3 * (1 << 30))
          << " GSample/s, AvgTime (1 trial) = "
          << std::setw(8) << elapsed.count() / trials
          << " ms, Time (all) = "
          << std::setw(8) << elapsed.count()
          << " ms, Size = " << size
          << std::endl;
```

Pozivamo funkciju **rocrand_destroy_generator(generator)** koja očisti generator na način da ga se uništi, te obavimo provjeru da li se memorija generatora zaista oslobodila pomoću funkcije **hipFree(data):**

```
        ROCRAND_CHECK(rocrand_destroy_generator(generator));
        HIP_CHECK(hipFree(data));
}  [23]
```

Da biste generirali slučajne brojeve na domaćinskom CPU-u, najprije je potrebno pozvati **rocrand_generator**, a zatim se pomoću naredbe **hipMalloc()** dodijeli memorija na domaćinskom međuspremniku kako bi mogao pospremiti rezultate. Svi ostali pozivi djeluju identično bilo da se generiraju slučajni brojevi na uređaju ili na glavnom CPU-u. Dozvoljeno je kreirati nekolicinu generatora istovremeno. Svaki generator obuhvaća zasebno stanje te je neovisan o svim ostalim generatorima. Slijedovi brojeva proizvedenih od strane svakog generatora su unaprijed određeni tj. determinirani. S obzirom na iste parametre postavljene na ulazu, uvijek će se generirati isti izlazni slijed pri svakom pokretanju programa. Slučajno generiranje brojeva na uređaju rezultirati će istim redoslijedom kao i njihovo generiranje na domaćinskom CPU-u. Primjetimo kako funkcija **generator_func()** u ranije prikazanom primjeru pokreće kernel i vraća se asinkrono. Ukoliko pokrenemo drugi kernel u nekom drugom protoku tj. tijeku (engl. *stream*), a taj drugi kernel treba koristiti rezultate funkcije generate_func(), potrebno je ili pozvati funkciju **hipDeviceSynchronize()** ili koristiti rutine za upravljanje tijekom ili upravljanje događajima kako bi smo osigurali da je kernel za slučajno generiranje dovršio izvršavanje prije pokretanja novog kernela. Trebamo paziti kako ne bi došlo do situacije usmjeravanja pokazivača memorije domaćina prema generatoru koji se izvodi na uređaju te također je potrebno paziti da se ne usmjeri pokazivač memorije uređaja prema generatoru koji se izvodi na CPU-u (dakle suprotna situacija) jer taj način rada nije ispravan. Ponašanje u tim slučajevima nije definirano. [24]

5.2. Biblioteka rocBLAS

Biblioteka rocBLAS je AMD-ova biblioteka koja sadrži osnovne funkcije za linearnu algebru (engl. *basic linear algebra subroutines*, kraće BLAS) na ROCm-u koja se izvodi u programskom jeziku HIP i optimizirana je za AMD-ove GPU-e. BLAS je implementacija povrh AMD-ovog Radeon Open Compute ROCm runtime-a i skupine alata. rocBLAS je implementiran u programskom jeziku HIP i optimiziran je za najnovije diskretne GPU-e AMD-a. [25] [26]

5.3. Biblioteka rocSPARSE

Biblioteka rocSPARSE služi kao zajedničko sučelje koje pruža osnovne linearne algebarske potprograme za rijetko izračunanje koje je implementirano povrh AMD-ovog Radeon Open Compute ROCm runtime-a i skupine alata. rocSPARSE je implementiran u programskom jeziku HIP i optimiziran je za najnovije diskretne GPU-e AMD-a. [27]

6. Zaključak

Zaključno navest ćemo neka od razmišljanja Timothyja Pricketta Morgana, novinara na The Next Platform i „proricatelja" budućnosti, na području HSA računarstva kako bi smo čitatelju približili vlastito viđenje budućih zbivanja u svijetu HSA.

Spomenuti stručnjak u prvom članku koji ćemo navesti, na temu „The still expanding market for GPU compute" objavljenom (21.12.2019.) na već spomenutom portatlu The Next Platform (Stackhouse Publishing Inc u suradnji sa The Register, utjecajnim britanskim publicistima na području tehnologije) ističe:

„Zamislite kako brzo Xeon ili Power ili Epyc ili ThunderX procesor bi mogao izvršavati ukoliko bi se fokus pomaknuo prema maksimiziranju performansi CPU procesora s jednom niti. Glavni razlog zašto se poslovanje Nvidia-inog podatakovnog centra povećava iznimno velikom brzinom je iz razloga što se već više od desetljeća njihovo poslovanje prebacilo od proizvođača GPU-a koji su uglavnom usmjereni na klijentske uređaje, na proizvođača platformi. Pojavom CUDA-e i ubrzanjima koja su se izvodila na HPC sustavima područje primjene proširio se na radno opterećenje (engl. workloads) umjetne inteligencije (engl. artificial intelligence, skraćeno AI) dovelo je do toga da novo tržište usmjereno na GPU ubrzane baze podataka ima veliki potencijal za širenje. Nvidia sada stvara platforme za strujanje videozapisa i mrežnih igara, omogućuje pokretanje različitih opterećenja podataka u domeni znanosti te na taj način uključujuje tradicionalnne algoritme za statističko strojno učenje." [28]

U svojem drugom članku objavljenom 07.05.2019. a na koji ćemo se pozvati, gospodin Prickett Morgan piše na temu „Cray, AMD tag team on 1.5 exaflops „Frontier" supercomputer" te ističe:

„Ništa nije bolji dokaz pouzdanja u buduće planove AMD-ovih CPU-a i GPU-a kao i domaće povezivanje „Slingshot" i Cray-jev „Shasta" dizajna sustava od činjenice da su ova dva proizvođačka giganta sklopila posao kako bi srušili „Frontier-ov" exascale sustav. Frontier superračunalo biti će instalirano u „Oak Ridge National Laboratory-ju" kao nasljednik trenutno postojećeg „Summit" sustava, koji je IBM izgradio u suradnji s Nvidia proizvođačem GPU-a i „InfiniBand Mellanox Technologies" proizvođačem preklopnika." [29]

Uzimajući oba članka u obzir, možemo reći da je vrijeme HSA tek dolazi.

Literatura

[1] „Heterogeneous System Architecture“, *Wikipedia*. 24-srp-2019.
[2] „FPGAs in Heterogeneous Systems | IEEE Computer Society“. [Na internetu]. Dostupno na: https://www.computer.org/publications/tech-news/fpgas-in-the-world-of-heterogeneous-systems. [Pristupljeno: 25-ruj-2019].
[3] „PCI (Peripheral Component Interconnect) Definition“. [Na internetu]. Dostupno na: https://techterms.com/definition/pci. [Pristupljeno: 13-lis-2019].
[4] S. B. Online, „Chapter 2: HSA Overview - Heterogeneous System Architecture“. [Na internetu]. Dostupno na: https://learning.oreilly.com/library/view/heterogeneous-system-architecture/9780128008010/B9780128003862000018.xhtml. [Pristupljeno: 03-lis-2019].
[5] „Application programming interface“, *Wikipedia*. 02-pros-2019.
[6] B. Sander i T. Tye, „Chapter 3 - HSAIL - Virtual Parallel ISA“, u *Heterogeneous System Architecture*, W. W. Hwu, Ur. Boston: Morgan Kaufmann, 2016, str. 19–34.
[7] „Heterogeneous system architecture helps AMD and ARM deal with mammoth compute demands | TheINQUIRER“, *http://www.theinquirer.net*, 23-stu-2015. [Na internetu]. Dostupno na: https://www.theinquirer.net/inquirer/feature/2435169/heterogeneous-system-architecture-helps-amd-and-arm-deal-with-mammoth-compute-demands. [Pristupljeno: 25-ruj-2019].
[8] „Digital signal processing“, *Wikipedia*. 07-ruj-2019.
[9] „Codec“, *Wikipedia*. 20-ruj-2019.
[10] „Direct memory access“, *Wikipedia*. 09-kol-2019.
[11] „Secure cryptoprocessor“, *Wikipedia*. 30-ruj-2019.
[12] „Hwu - 2016 - Heterogeneous system architecture a new compute p.pdf“. .
[13] „Lightweight programming language“, *Wikipedia*. 08-lis-2019.
[14] „Ultra-large-scale systems“, *Wikipedia*. 12-ruj-2019.
[15] B. S i er, „News from the HIP part of the world“, *GPUOpen*, 12-ruj-2016. [Na internetu]. Dostupno na: https://gpuopen.com/news-hip-part-world/. [Pristupljeno: 26-ruj-2019].
[16] „Non-uniform memory access“, *Wikipedia*. 18-ruj-2019.
[17] „AMD Accelerated Processing Unit“, *Wikipedia*. 11-ruj-2019.
[18] „Kernel (operating system)“, *Wikipedia*. 21-ruj-2019.
[19] „AMDKFD Looking To Be Merged Into AMDGPU Linux DRM Kernel Driver - Phoronix“. [Na internetu]. Dostupno na: https://www.phoronix.com/scan.php?page=news_item&px=AMDKFD-Merge-Into-AMDGPU. [Pristupljeno: 26-ruj-2019].
[20] „What exactly is LLVM?“, *Stack Overflow*. [Na internetu]. Dostupno na: https://stackoverflow.com/questions/2354725/what-exactly-is-llvm. [Pristupljeno: 26-ruj-2019].
[21] „Application-specific integrated circuit“, *Wikipedia*. 02-ruj-2019.
[22] „ROCm: Open Platform For Development, Discovery and Education around GPU Computing“, *GPUOpen*. [Na internetu]. Dostupno na: https://gpuopen.com/compute-product/rocm/. [Pristupljeno: 19-stu-2019].
[23] *ROCmSoftwarePlatform/rocRAND*. ROCm Software Platform, 2019.
[24] „Host API Overview“. [Na internetu]. Dostupno na: http://docs.nvidia.com/cuda/curand/index.html. [Pristupljeno: 10-pros-2019].
[25] „ROCmSoftwarePlatform/rocRAND“, *GitHub*. [Na internetu]. Dostupno na: https://github.com/ROCmSoftwarePlatform/rocRAND. [Pristupljeno: 09-pros-2019].
[26] „ROCmSoftwarePlatform/rocBLAS“, *GitHub*. [Na internetu]. Dostupno na: https://github.com/ROCmSoftwarePlatform/rocBLAS. [Pristupljeno: 09-pros-2019].
[27] *ROCmSoftwarePlatform/rocSPARSE*. ROCm Software Platform, 2019.

[28] T. P. Morgan, „The Still Expanding Market For GPU Compute“, *The Next Platform*, 21-ožu-2019. [Na internetu]. Dostupno na: https://www.nextplatform.com/2019/03/21/the-still-expanding-market-for-gpu-compute/. [Pristupljeno: 10-pros-2019].
[29] T. P. Morgan, „Cray, AMD Tag Team On 1.5 Exaflops “Frontier” Supercomputer“, *The Next Platform*, 07-svi-2019. [Na internetu]. Dostupno na: https://www.nextplatform.com/2019/05/07/cray-amd-tag-team-on-1-5-exaflops-frontier-supercomputer/. [Pristupljeno: 10-pros-2019].
[30] S. B. Online, „Chapter 2: HSA Overview - Heterogeneous System Architecture“. [Na internetu]. Dostupno na: https://learning.oreilly.com/library/view/heterogeneous-system-architecture/9780128008010/B9780128003862000018.xhtml. [Pristupljeno: 03-lis-2019].