Elektrotehniški vestnik 77(5): 267–272, 2010
Electrotechnical Review, Ljubljana, Slovenija

Strojni in programski vidiki arhitekture CUDA

Tomaž Dobravec, Patricio Bulić

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, Tržaška 25, 1000 Ljubljana, Slovenija
E-pošta: tomaz.dobravec@fri.uni-lj.si

Povzetek. V tem članku predstavimo grafične procesne enote CUDA iz dveh zornih kotov: strojnega in programskega. Pri strojnem vidiku opišemo zgradbo grafične procesne enote, zgradbo posameznih procesnih enot ter opišemo način izvajanja ukazov v procesnih enotah. Programer arhitekturo CUDA uporablja kot dodatno računsko moč, in sicer predvsem za probleme, pri katerih je mogoča visoka stopnja paralelizacije. Kodo razbije v ščepce, ki se lahko izvajajo tudi v več tisoč zaporednih ali vzporednih nitih.

Ključne besede: grafične procesne enote, CUDA, paralelno programiranje

A Hardware/Software View of CUDA

Extended abstract. This paper presents a hardware/software view of CUDA. A Graphical Processing Unit (GPU) has up to 480 streaming processors (SP), organized as up to 30 streaming multiprocessors (SM). Each SM has 8 or 16 SPs [2, 3]. SM is an independent processing unit. SM consists of a Fetch/Issue unit (Figure 1) and an execution unit (Figure 2). SM uses two different clock domains for the Fetch/Issue unit and the execution unit, respectively. SM manages and executes threads in groups of 32 parallel threads called warps. An issued warp executes in the execution unit as a set of 32 threads over four processor cycles (Figure 3). Besides a shared memory and a register set for each SM, each SM has a uniform access to a global memory. The global memory (up to 4GB DDR3 SDRAM) is connected to SMs via a 384-bit wide bus and is used for comunication between CPU and GPU and for sharing data among threads in different SMs.

From the programmers point of view, CUDA is a device which can be used through a parallel programming model. The code is split into the so called kernels which are executed in up to several thousand parallel and/or asynchronous threads. Parallel threads can cooperate using synchronization mechanisms and shared memory, while asynchronous threads can only use a common global memory.

Key words: graphics processing units, CUDA, parallel programming

1 Uvod

Pri izvajanju zahtevnih grafičnih aplikacij je grafična kartica poleg centralne procesne enote (CPE) eden najbolj obremenjenih delov računalniškega sistema. Poleg tega gre v teh primerih za povečini visoko paralelne probleme, zato si že zelo dolgo močno prizadevajo povečati zmogljivosti grafične strojne opreme s povečanjem paralelizma na strojni ravni. Tako so grafične kartice v zadnjem desetletju doživele velik razvoj za povečanje procesnih jeder na čipu. Podjetje NVIDIA je v tej smeri verjetno (v tem trenutku) naredilo največ in tako imamo danes na trgu grafične kartice, ki imajo že 30 samostojnih procesnih enot, od katerih ima vsaka po 16 procesnih jeder za izvajanje operacij v plavajoči vejici [1, 2, 3, 4]. Tako grafične kartice danes niso več namenjene samo procesiranju grafike, temveč splošnemu računanju. Ta visoka stopnja strojnega paralelizma zahteva dobro podporo tudi na programski ravni. V ta namen so leta 2006 v podjetju NVIDIA razvili programsko platformo CUDA (*angl. Compute Unified Device Architecture*), ki omogoča razvoj programske opreme za večjedrne grafične kartice. CUDA programerju skrije dokaj zapleteno zgradbo strojne opreme in omogoča prijazen način pisanja programske opreme. CUDA vključuje razširitev programskega jezika C/C++, prevajalnik, knjižnice funkcij in abstrakcijo strojne opreme. CUDA programerju omogoča razvoj aplikacij na ravni niti, čeprav od njega ne zahteva eksplicitne tvorbe in upravljanja niti. Ker CUDA vključuje razširitev programskega jezika C/C++, programer lahko poljubno prepleta programsko kodo v C/C++, ki se bo izvajala na centralni procesni enoti, ter programsko kodo, ki se bo izvajala na večjedrni grafični kartici. CUDA je namenjena predvsem razvoju in izvajanju podatkovno intenzivnih aplikacij, kjer se uporablja veliko operacij v plavajoči vejici.

V delu predstavimo strojni in programski vidik modernih grafičnih procesnih enot podjetja NVIDIA. V 2. poglavju opišemo strojno zgradbo grafične procesne enote. Poglavje 3 predstavi programske vidike arhitekture CUDA. V poglavju 5 predstavimo sklepne ugotovitve.

Prejet 14. julij, 2010
Odobren 22. september 2010

2 Strojni vidik arhitekture CUDA

Z vidika strojne opreme gledamo na arhitekturo CUDA, kot na grafično procesno enoto (GPE). Grafična procesna enota je danes v sistem povezana prek vodila PCI Express in običajno zahteva 16-kanalno povezavo, ki zagotavlja hiter prenos podatkov med CPE (oz. sistemskim pomnilnikom) in GPE. GPE sestavlja 8-30 samostojnih centralnih procesnih enot (*angl. Streaming Multiprocessors*) (SM) [2, 3]. Enote SM so medseboj povezane prek skupnega vodila z zelo hitrim pomnilnikom (trenutno v tehnologiji DDR3 SDRAM). Na celotno GPE lahko gledamo kot na tesno sklopljeni večprocesorki sistem, pri katerem imajo vse SM enoten dostop do glavnega pomnilnika. SM je najmanjša nedvisna procesna enota, ki sestavlja GPE. SM je zelo podoben modernim mikroprocesorjem, le da je njegova zgradba delno prirejena izvajanju operacij nad slikovnimi elementi (piksli).

2.1 Zgrada neodvisne centralne procesne enote (SM)

Procesna enota SM je pravzaprav superskalarni procesor s cevovodno zgradbo, čeprav v podjetju NVIDIA pravijo (v bistvu gre za marketinško potezo), da je SM multiprocesor, sestavljen iz osmih (pri novejših GPE celo 16) nitnih procesorjev. Interno zgradbo procesne enote SM lahko razdelimo v dva dela: enoto za zajem, dekodiranje in izstavljanje ukazov (*angl. Fetch and Issue Unit*) (FIU) in izvajalno enoto (*angl. Execution Unit*) (EX) z več funkcijskimi enotami. Enota FIU zajema, dekodira in izstavlja ukaze v enoto EX, medtem ko enota EX izstavljene ukaze izvaja. Obe enoti sta zgrajeni cevovodno.

2.1.1 Enota za zajem, dekodiranje in izstavljanje ukazov (FIU)

Enoto FIU sestavljata dve stopnji: stopnja za zajem ukazov (*angl. Instruction Fetch*) (IF) in stopnja za dekodiranje ukazov (*angl. Instruction Decode*) (ID). Pri današnjih GPE deluje enota FIU tipično s frekvenco ure 700 MHz (imenujemo jo tudi *počasna ura*).

Zgradba enote FIU je prikazana na sliki 1. V vsaki urni periodi počasne ure se v stopnji IF zajame en ukaz iz ukaznega predpomnilnika. Vsi ukazi so skalarni. Enota FIU implementira dinamično razvrščanje ukazov. Za to uporablja tehniko *scoreboarding*, ki je bila za dinamično razvrščanje ukazov prvič uporabljena leta 1963 v računalnikih CDC 6600. Pri tej tehniki dinamičnega razvrščanja ukazov se ID stopnja razdeli v dva dela:

1. dekodiranje ukazov ter preverjanje strukturnih in WAW-nevarnosti,
2. branje operandov in preverjanje RAW-nevarnosti.

Operandi se lahko shranjujejo v osmih registrskih nizih, od katerih ima vsak niz 1024 32-bitnih registrov (v resnici gre za 8K 32-bitnih registrov, implementiranih v hitrem pomnilniku SRAM na istem čipu, kot SM), v pomnilniku konstant (gre za hiter pomnilniku SRAM na istem čipu, kot SM, do katerega SM lahko opravi le bralni dostop) ali pa v hitrem skupnem pomnilniku (*angl. Shared Memory*) velikosti 16 KB. Ko je ukaz pripravljen za izvajanje (je dekodiran, ima pripravljene operande ter nima WAW- in RAW- nevarnosti), se izstavi v izvajalno enoto.

2.1.2 Izvajalna enota (EX)

Enoto EX sestavlja relativno veliko različnih funkcijskih enot, pri čemer vsaka funkcijska enota lahko deluje samostojno in neodvisno od preostalih. Posamezne funkcijske enote izvajajo operacije, ki jih zahtevajo izstavljeni ukazi. Zgradba izvajalne enote je prikazana na sliki 2. Izvajalno enoto sestavlja:

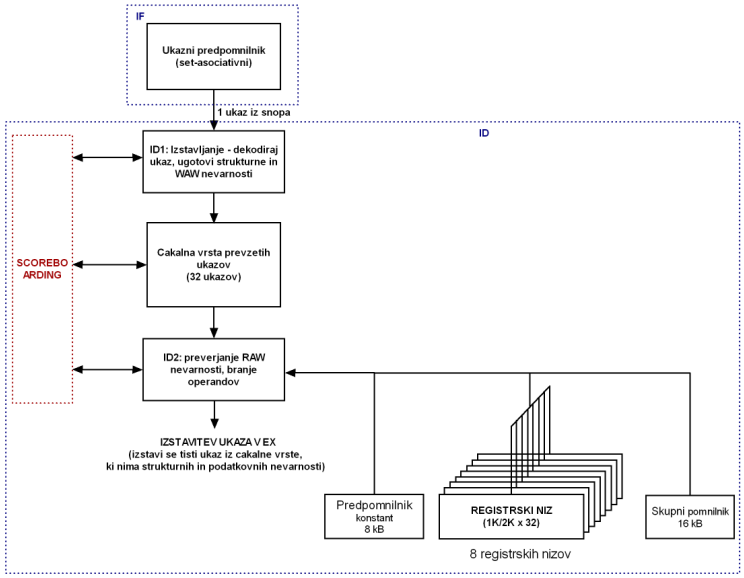
a. osem 32-bitnih celoštevilskih enot za računanje s celoštevilskimi operandi in naslovi, tj. izvajanje skočnih ukazov,
b. osem 32-bitnih enot ALE/MAD za računanje v plavajoči vejici in izvajanje ukazov MAD (*angl. Multiply and Add*),
c. dve enoti za množenje v plavajoči vejici in
d. dve enoti za transcedenčne operacije.

Vse funkcijske enote (razen enot za transcedenčne operacije) imajo cevovodno zgradbo, pri čemer ima cevovod štiri stopnje. Začetna latenca teh enot je 4 urne periode, medtem ko je latenca transcedenčne enote 16 ali več urnih period. Vsaka stopnja v cevovodu se izvede v eni urni periodi, pri čemer je frekvenca ure, s katero deluje izvajalna enota, enkrat večja od frekvence ure v enoti FIU (danes tipično 1400 MHz) in jo imenujemo *hitra ura*. V podjetju NVIDIA pravijo, da ima ena procesna enota SM osem jeder oz. nitnih procesorjev (*angl. Streaming Processor*) (SP), čeprav se moramo zavedati, da to niso samostojni procesorji. Eno jedro SP sestavljata po ena enota za celoštevilske operacije in ena enota za operacije v plavajoči vejici, medtem ko si štiri jedra SP delijo po eno enoto za množenje v plavajoči vejici in eno enoto za transcedenčne operacije.

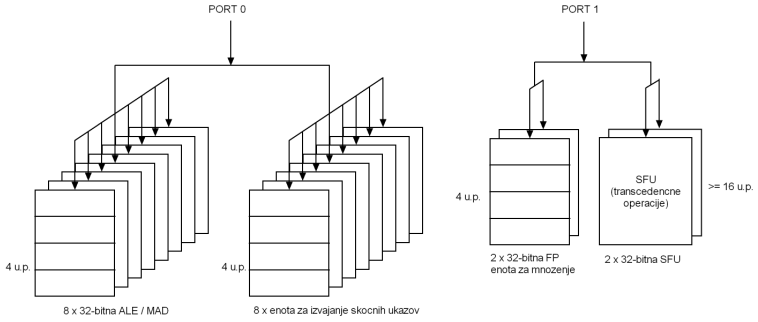
2.2 Izvajanje ukazov

Procesne enote SM so namenjene izvajanju večjega števila niti. Največjo stopnjo paralelizma dobimo, ko niti izvajajo iste ukaze. Seveda je mogoče, da imamo znotraj niti vejitvene ukaze in posamezne niti izvajajo različno kodo, vendar je v tem primeru izvajanje počasnejše.

Razvrščanje niti po enotah SM in njihov vrstni red izvajanja poteka na strojni ravni. Najmanjše programske enote, ki se razvrščajo se imenujejo snopi (*angl. warps*). Vsak snop sestavlja natanko 32 niti, pri čemer naj bi vse



Slika 1. Poenostavljena zgradba enote FIU
Figure 1. Fetch/Issue Unit in SM.



Slika 2. Izvajalna enota EX
Figure 2. Execution Unit in SM.

niti v snopu izvajale isto programsko kodo. Ena procesna enota SM izvaja večje število snopov, vendar največ 16. Moramo razumeti, da je dejansko izvajanje ukazov posamezne niti odvisno od vejitvenih ukazov, vendar si želimo, da bi vse niti enako vejile, oz. da bi bila divergenca niti minimalna.

Vse niti v snopu istočasno izvajajo isto množico ukazov. To pomeni, da v nekem trenutku nit iz snopa izvaja ukaz, ali pa ne izvaja nobenega ukaza. Slednje nastane pri vejitvah, ko določena nit ne izvaja iste veje ukazov kot preostale niti v snopu. S stališča strojne opreme to pomeni, da se v nekem trenutku v enoto EX izstavi en ukaz, ki pripada enemu snopu niti. Z drugimi besedami povedano, izvajalna enota izvaja isto operacijo za 32 različnih niti v snopu, pri čemer se za vsako nit v snopu operacija izvaja nad različnimi operandi. Gre torej za računalnik SIMD (po Flynnovi klasifikaciji).

Da bomo laže razumeli, kako izvajalna enota izvaja ukaze za posamezne niti v snopih, predpostavimo naslednji primer. Enota FIU v nekem trenutku izstavi enoti EX ukaz za seštevanje (ADD), kot prikazuje slika 3. Izstavljeni ukaz ADD pripada enemu snopu niti. Izvajalna enota začne izvajati ta ukaz v osmih funkcijskih enotah ALE/MAD za prvih osem niti v snopu. V naslednji urni periodi hitre ure prične z izvajanjem naslednjih 8 niti iz snopa in tako naprej. Po štirih urinih periodah hitre ure imamo v cevovodnih funkcijskih enotah ALE/MAD 32 ukazov ADD. Lahko rečemo, da se v tem trenutku v eni enoti SM sočasno izvaja 32 niti. Predpostavimo še, da se v naslednji urni periodi počasne ure izstavi ukaz za množenje MUL, ki pripada nekemu drugemu snopu 32 niti. Izvajalna enota za njegovo izvajanje ima na voljo dve funkcijski enoti za množenje v plavajoči vejici. Ob predpostavki, da sta nezasedeni, lahko začne izvajalna enota izvajati dve niti iz snopa in v vsaki naslednji urni periodi hitre ure začne izvajati naslednji dve niti iz snopa.

Vidimo, da s takšnim razvrščanjem snopov lahko ena procesna enota SM v nekem trenutku izvaja več kot 32 niti hkrati.

2.3 Pomnilnik

Poleg predpomnilnika konstant, registrskega bloka in skupnega pomnilnika, ki pripadajo eni enoti SM, imamo na GPE še globalni pomnilnik, do katerega imajo vse enote SM enoten dostop prek skupnega vodila (*angl. Uniform Memory Acess*). S stališča procesnih enot gre za zelo počasen pomnilnik z latenco dostopov nekaj 10 urnih period [2, 3]. Glavni pomnilnik je na boljših GPE velik 4GB v tehnologiji DDR3 SDRAM. Z enotami SM je povezan prek skupnega 384-bitnega vodila s prepusnostjo do 177 GB/s. Sistemska CPE dostopa le do globalnega pomnilnika. Njegov osnovni namen je, da zagotavlja izmenjavo podatkov med CPE in GPE ter komunikacijo med nitmi, ki se ne izvajajo na isti enoti SM. Niti, ki se izvajajo na isti enoti SM, komunicirajo prek hitrega skupnega pomnilnika v SM.

3 Programski vidik arhitekture CUDA

CUDA je namenjena reševanju problemov, pri katerih je mogoče doseči veliko stopnjo paralelizacije. Programer uporablja razširljiv programski model [6], ki omogoča pisanje programov za različne arhitekture – v času pisanja programa ni pomembno, na koliko jedrih SP se bo ta izvajal – dejansko število jeder ima potencialni vpliv samo na učinkovitost, ne pa tudi na pravilnost izvajanja. CUDA ni samostojen računalnik, temveč je naprava, nameščena v osnovnem računalniku kot pomožna računska enota. Osnovni računalnik se po navadi imenuje gostitelj (*angl. host*), CUDA pa naprava (*angl. device*).

Program, ki uporablja arhitekturo CUDA, se lahko hkrati izvaja na gostitelju in na napravi. Pri določanju izvajalne enote se navadno uporablja naslednji pristop: problem se razbije na podprobleme – za tiste podprobleme, ki imajo nizko stopnjo paralelizacije, je smiselno, da se izvajajo na gostitelju (ta je za neparalelno izvajanje običajno zmogljivejši od naprave), podproblemi z visoko stopnjo paralelizacije pa naj se izvajajo na napravi.

Programski model predvideva dvojno stopnjo paralelizacije: groba paralelizacija razbije problem na podatkovno med seboj neodvisne *bloke*, znotraj katerih nadaljnje razbijanje problema privede do množice odvisnih in (po potrebi) sinhroniziranih *niti*, ki lahko upravljajo isto množico podatkov. Posamezni bloki se izvajajo na različnih enotah SM (lahko tudi zaporedno na isti enoti – odvisno pač od razpoložljivosti), zato je sodelovanje med njimi mogoče le prek globalnega pomnilnika. Množica niti, ki se izvajajo v bloku, pa hkrati teče na isti enoti SM, zato imajo vse dostop do istih virov in lahko med seboj sodelujejo.

Osnovni programski jezik za arhitekturo CUDA je razširjen C/C++: jeziku C/C++ so dodane funkcije za prenos podatkov med gostiteljem in napravo (v obe smeri), za upravljanje pomnilnika in izvajanje kode na napravi ter nekatera dodatna orodja (kot na primer merjenje časa, upravljanje sporočil o napakah in podobno). Obstajajo tudi razširitve za druge jezike (Java, Python, .NET, MathLab, Fortran ...).

Programer mora prevajalniku sporočiti, katera koda (funkcija) se bo izvajala na gostitelju in katera na napravi in od kod (iz naprave ali iz gostitelja) se posamezne funkcije lahko kličejo. To stori s pomočjo dodatnih oznak (del razširitve jezika) *__global__*, *__device__* in *__host__*. S prvo in drugo označujemo funkcije, ki se bodo izvajale na napravi, s tretjo pa funkcije, ki se bodo izvajale na gostitelju, pri čemer tiste z oznako *__global__* lahko kličemo iz gostitelja, tiste z oznako *__device__* pa samo znotraj naprave. Oznaka *__host__* je privzeta oznaka in je zato običajno ne pišemo.

Program, napisan v programskem jeziku C za arhitekturo CUDA, ima po navadi končnico *cu* in se prevaja s prevajalnikom *nvcc*. Ta loči programsko kodo na dva dela: na kodo, ki se bo izvajala na napravi, in na kodo, ki se bo izvajala na gostitelju. Prvo prevede v vmesno kodo PTX (ta je primerna za direktno izvajanje na napravi), drugo pa s pomočjo sistemskega prevajalnika (na primer *gcc*) v izvršljivo datoteko.

3.1 Ščepci

Osnovni gradnik paralelnega programa, ki se izvaja na arhitekturi CUDA, se imenuje ščepec (*angl. kernel*). Ta se ob klicu izvede N-krat v N vzporednih nitih. V nasprotju z navadnimi nitmi, ki tečejo na enoprocesorskem računalniku (in se zato izvajajo samo navidezno vzporedno), se niti, ki tečejo na arhitekturi CUDA, dejansko izvajajo vzporedno. Skupno število niti, ki se lahko izvajajo hkrati, je odvisno od lastnosti naprave (število in zmogljivost jeder) in teoretično lahko doseže vrednosti od nekaj sto do nekaj tisoč.

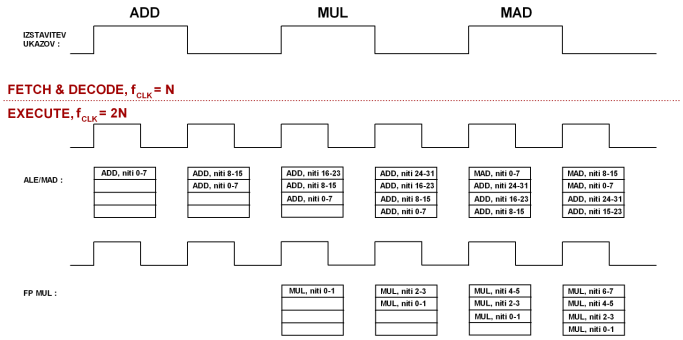
Niti istega ščepca se med seboj razlikujejo samo po zaporedni številki. Gre za zelo preprost in hkrati zelo učinkovit mehanizem, s pomočjo katerega lahko nadzorujemo in upravljamo delovanje posamezne niti.

```
// Scepec za "povecanje" vektorja
// (nit poveca vrednost v svojem polju)

__global__ void AddToVector(int *A) {
  // zaporedna stevilka niti
  int i = threadIdx.x;

  A[i] += i;
}

int main(int argc, char **argv){
  ...
  // klic scepca v N nitih
  InitVector<<<1,N>>>(A);
  ...
}
```



Slika 3. Potek izstavljanja in izvajanja ukazov
Figure 3. Instruction issue and execution.

Zgornji primer prikazuje, kako lahko ta mehanizem uporabimo za reševanje preprostega problema: vektor velikost N želimo "povečati" tako, da bomo i-temu polju tega vektorja prišteli i. V nasprotju s klasičnim (zaporednim) pristopom, s katerim bi problem rešili s pomočjo zanke, bomo na arhitekturi CUDA problem rešili s pomočjo N vzporednih niti – vsaka bo spremenila le "svoje" polje (ker ima vsaka nit svojo zaporedno številko, lahko uporabimo naravno preslikavo med nitmi in polji tabele: i-ti niti pripada i-to polje; vanj bo nit prištela svojo zaporedno številko).

3.2 Prenos podatkov med gostiteljem in napravo

Gostitelj ima dostop do skupnega pomnilnika na napravi prek funkcij, s katerimi lahko rezervira in sprosti pomnilniški prostor ter prenaša podatke iz gostitelja v napravo in nasprotno. Funkcije za delo s pomnilnikom so

- `cudaMalloc()` in `cudaFree()`: rezervacija in sproščanje pomnilnika na napravi.
- `cudaMemcpy`: prenos podatkov med gostiteljem in napravo ali med različnimi lokacijami na napravi sami. Smer prenosa se določi s parametrom, ki ima lahko eno od naslednjih vrednosti: `cudaMemcpyDeviceToHost`, `cudaMemcpyHostToDevice` ali `cudaMemcpyDeviceToDevice`.

Prenos podatkov predstavimo na preprostem primeru: vektor na gostitelju bomo napolnili z N ničlami, vrednosti vektorja bomo prenesli v vektor na napravo, ga tam povečali (z uporabo funkcije AddToVector) in nato spremenjenega prenesli nazaj na gostitelja.

```
int main(int argc, char **argv){
  int i; // pomozna spremenljivka

  int *h_A;  // vektor na gostitelju
  int *d_A;  // vektor na napravi

  int N = 512; // stevilo podatkov v vektorju
  size_t size = N * sizeof(int);

  // rezerviram prostor na gostitelju...
  h_A = (int*) malloc(size);
  // ... in na napravi
  cudaMalloc((void**)&d_A, size);

  // vrednosti v h_A postavim na 0
  for(i=0; i<N; i++) h_A[i] = 0;

  // vsebino vektorja h_A prenesem na napravo ...
  cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice);
  // ... i-ti komponenti (i=1...N) pristejem i ...
  AddToVector<<<1,N>>>(d_A);
  // ... in vektor prenesem nazaj na gostitelja
  cudaMemcpy(hA, dA, size, cudaMemcpyDeviceToHost);

  // Se test pravilnosti izvajanja:
  for(i=0; i<N; i++)
    if (h_A[i] != i)
      printf("Napaka: %d\n", i);

  // Obvezno sproscanje pomnilnika na napravi ...
  cudaFree(d_A);
  // ... in na gostitelju
  free(h_A);
}
```

3.3 Organizacija niti

Množica niti, ki nastane pri izvajanju enega ščepca, je zaradi lažje organizacije in nadzora ter v želji po čim večji prilagodljivosti programa na različne arhitekture hierarhično urejena v bloke (*angl. thread blocks*), ti pa so razporejeni v mrežo (*angl. grid of blocks*). Niti istega bloka se izvajajo sočasno na enem jedru, delijo si del skupnega pomnilnika in lahko med seboj sodelujejo tudi s pomočjo sinhronizacije.

Niti znotraj istega bloka lahko označujemo z eno-, dvo- ali tridimenzionalnim indeksom, to je odvisno od tipa problema, ki ga rešujemo. Če na primer ščepci obdelujejo dvodimenzionalno tabelo, je smiselno uporabiti dvodimenzionalno označevanje, saj s tem dosežemo naravno preslikavo (i, j)-te niti na (i, j)-to polje tabele. Znotraj posameznega bloka CUDA omogoča izvajanje največ 512 niti. Če število niti preseže to mejo, jih moramo razporediti v več blokov, vendar se moramo pri tem zavedati, da sodelovanje med nitmi različnih blokov ni mogoče (niti

prek sinhronizacije niti prek uporabe skupnega pomnilnika). Namreč, dva bloka se lahko izvedeta vzporedno ali zaporedno, in to na istem ali na dveh različnih jedrih, odvisno pač od razpoložljivosti virov (na arhitekturi z osmimi enotami SM se bo osem blokov izvedlo hkrati, na taki z dvema enotama SM pa se bosta izvedla po dva in dva bloka v štirih zaporednih fazah). Programerju je podatek o tem, kje in kdaj se izvaja posamezen blok, skrit.

Tudi bloke lahko označujemo z eno- ali dvodimenzionalnim indeksom. Za označevanje blokov znotraj mreže in niti znotraj blokov je smiselna uporaba indeksov iste dimenzije. Pri uporabi enodimenzionalne tabele za izračun naravnega indeksa uporabimo formulo

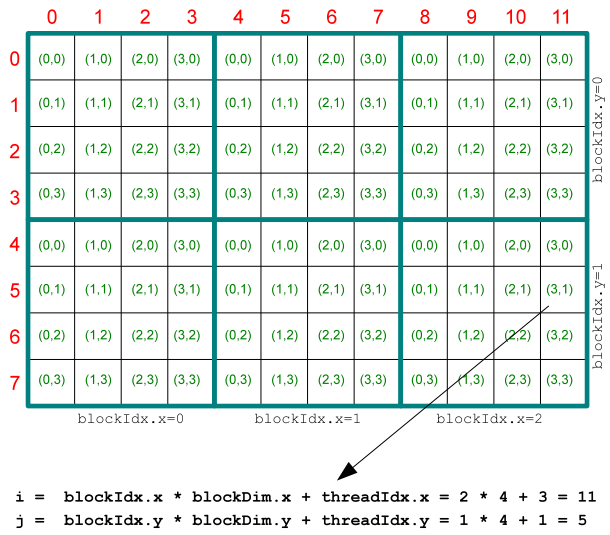
```
int i= blockIdx.x * blockDim.x + threadIdx.x;
```

pri dvodimenzionalni tabeli pa

```
int i= blockIdx.x * blockDim.x + threadIdx.x;
int j= blockIdx.y * blockDim.y + threadIdx.y;
```

Velikost in število blokov se določi ob klicu ščepca v špičastih oklepajih (podata se dva parametra tipa bodisi `int` (pri enodimenzionalni razporeditvi) bodisi `dim3` za dvo- ali tridimenzionalno razporeditev). Slika 4 prikazuje primer razporeditve niti ščepca `racunaj()` in računanje dvodimenzionalnega indeksa posamezne niti pri uporabi šestih blokov (3 x 2) po 16 niti (4 x 4) ob klicu

```
dim3 threads(4, 4);
dim3 grid(3, 2);
racunaj<<< grid, threads >>>();
```



Slika 4. Računanje indeksov na mreži velikosti 3 x 2 in bloku velikosti 4 x 4

Figure 4. Calculating the thread indeces on a grid of size 3 x 2 and block size of 4 x 4.

4 Uporaba pomnilnika

Posamezni niti je na voljo pomnilnik treh tipov: lokalni (*angl. per-thread local*), skupen (*angl. per-block shared*) in globalni (*angl. global*). Lokalni pomnilnik je namenjen predvsem lokalnim spremenljivkam in ni viden drugim nitim. Skupen pomnilnik je viden vsem nitim istega bloka in je veliko hitrejši od globalnega pomnilnika, ki je viden vsem (tudi gostitelju). Prav zaradi velike razlike v hitrosti je priporočljiva uporaba skupnega pomnilnika povsod, kjer je to le mogoče.

5 Sklep

NVIDIA je z razširjanjem arhitekture CUDA v sklopu grafičnih kartic pripeljala pravo paralelno računanje v velik del sodobnih računalnikov. Čeprav se večina uporabnikov osebnih računalnikov niti ne zaveda, da skupaj z grafično kartico “brezplačno” prejme tudi izredno zmogljivo računsko enoto, je prisotnost arhitekture CUDA zelo dobrodošla pri uporabi namenskih programov, ki jo znajo dobro izkoristiti.

V tem članku smo predstavili grafične procesne enote CUDA iz dveh zornih kotov: iz strojnega in programskega. Če želimo namreč arhitekturo CUDA dobro izkoristiti, je poznavanje obeh nujno potrebno. Upamo, da smo bralcu podali dovolj informacij, ki jih potrebuje za začetek dela. Podrobnejše informacije pa bo, ko jih bo potreboval, našel v literaturi, ki je na tem področju res ne manjka.

6 Literatura

[1] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Yao Zhang, V. Volkov, Parallel Computing Experiences with CUDA, *IEEE Computer*, Vol. 28, No. 4, September 2008, pp. 13-27.

[2] T. R. Halfhill , Parallel Processing with CUDA, *Microprocessor Report*, January 28, 2008, pp. 1-8.

[3] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA TESLA: A Unified Graphics and Computing Architecture, *IEEE Micro*, Vol. 28, No. 2, March-April 2008, pp. 39-55.

[4] J. Nickolls, W. J. Dally, The GPU Computing Era, *IEEE Micro*, Vol. 30, No. 2, 2010, pp. 56-69.

[5] J. D. Owens, M. Houston, D. Luebke, S. Green, John E. Stone, J. C. Phillips, GPU Computing, *Proceedings of the IEEE*, Vol. 96, No. 5, May 2008, pp. 879-899.

[6] NVIDIA. *Programming guide (version 3.0)*, *http://developer.nvidia.com*, 20 February 2010.

Tomaž Dobravec je docent na Fakulteti za računalništvo in informatiko v Ljubljani.

Patricio Bulić je docent na Fakulteti za računalništvo in informatiko v Ljubljani.