Intel oneAPI z DPC++

Tomaž Hribernik, Lan Zukanović

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, Večna pot 113, 1000 Ljubljana, Slovenija

thXXXX@student.uni-lj.si, lz3523@student.uni-lj.si

**Povzetek.** Razvijanje zmogljivih, učinkovitih in skalabilnih algoritmov zna predstavljati težavo. Sploh, ko imamo opravka s sodobnimi računalniškimi sistemi, ki s seboj nosijo večjedrne procesorje, GPE-je (grafično procesne enote), pospeševalnike UI (umetne inteligence) ter razširitvene kartice FPGA (field-programmable gate array). Torej je potrebno uporabljati nabor različnih jezikov in orodij. V tem delu je predstavljena ena izmed rešitev tega, in sicer programski jezik DPC++ (Data Parallel C++), ki je bil razvit v okviru Intel oneAPI programskega modela. Sprva bo predstavljena osnovna ideja jezika, na koncu, pa še podrobnejši ogled dveh zahtevnejših algoritmov.

**Ključne besede:** Intel, oneAPI, DPC++, programski jezik

Prejet d. mesec, leto  
Odobren d. mesec, leto

**Intel oneAPI with DPC++**

The development of high performing and scalable algorithms proves to be difficult. Especially when dealing with modern computer systems who, under the hood, carry multiple core processors, GPUs (graphical processing units), AI (artificial intelligence) accelerators and FPGAs (field-programmable gate arrays). Because of this, developers are forced to use multiple programming languages and different tools, to make full use of the system. This paper will focus on one of the potential fixes. The DPC++ (Data Parallel C++) programming language developed by Intel for its oneAPI programming model. Firstly, the core idea of DPC++ will be presented by looking at the general building blocks of the language and at the end two more complex algorithms will be shown.

**Keywords:** Intel, oneAPI, DPC++, programming language

# Uvod

Doseganje zmogljivosti na sodobnih računalniških arhitekturah zahteva pisanje optimiziranih, energetsko učinkovitih in skalabilnih algoritmov. To zna biti težavno. Sploh, ko imamo opravka s sodobnimi računalniškimi sistemi, ki s seboj nosijo večjedrne procesorje, GPE-je (grafično procesne enote), pospeševalnike UI (umetne inteligence) ter razširitvene kartice FPGA (field-programmable gate array). Pri pisanju programske opreme, ki izkorišča vso to strojno opremo, potrebujemo različne jezike, knjižnice in orodja, kar precej zaplete in podaljša celoten proces.

Programski model oneAPI poenostavlja programiranje procesorjev in pospeševalnikov z uporabo dodatnih paralelnih konstruktov za jezik C++ in optimiziranih knjižnic iz specifikacije SYCL. Ta razširitev jezika C++ se imenuje Data Parallel C++ (DPC++). Jezik je bil razvit v okviru Intel oneAPI programskega modela in je trenutno v testni fazi (beta), pri čemer Intel ponuja vnaprej konfigurirano izvajalno okolje v njihovem oblaku, kjer je mogoče preizkusiti uporabo tega jezika na dosegljivih primerih za različne naprave [1][2]. Zato ker je še, v času pisanja tega, v beta fazi, je potrebno zaprositi dostop do izvajalnega okolja, lahko pa si prenesemo Docker sliko, na kateri je okolje že nameščeno. Najboljšo dokumentacijo trenutno najdemo na Intel-ovi spletni strani v pdf obliki, drugje na spletu (StackOverflow) pa je dokaj pomanjkljiva, saj je OneAPI še precej nov.

Tisti, ki ste že kdaj delali z OpenCL, boste lahko videli, da sta si okolji konceptualno zelo podobni. Ključna razlika je v številu vrstic izvorne kode.

# Kaj je dpc++

Data Parallel C++ (DPC++), kot ime pove, je navaden C++ z dodanimi elementi iz SYCL, ki je višje-nivojski programski model za OpenCL, katerega je razvil Khronos Group. Zaradi SYCL-a dobimo podporo paralelizma podatkov in enojno-izvornega (angl. single source), heterogenega programiranja. En izvor pomeni, da je vsa koda zapisana v eni datoteki, še bolj kot to je varna glede na tip (angl. type safe). Vsa paralelna koda se zanaša samo na klice knjižnice, kar pomeni da lahko navadni C++ prevajalniki (angl. compilers) to kodo prevedejo.

Ideja je da lahko hitro izdelamo funkcionalno kodo, potem pa se lahko spustimo v specifike in optimizacijo za določeno platformo.

DPC++:

• Je rešitev za vse enojno-arhitekturne jezike: vsako podjetje razvije nekaj svojega za določeno arhitekturo, kar pa ni dovolj splošno za uporabo med različnimi napravami. Ta problem poskuša rešiti OneAPI

• Razvijalcem omogoča ponovno uporabo kode čez vrsto strojne opreme/naprav (CPE-ji in pospeševalniki kot so GPE-ji in FPGA-ji). Najprej osredotočimo na korektno delovanje prek večih platform, potem pa na optimizacijo specifične platforme

Poleg samega prevajalnika so del OneAPI platforme tudi razne knjižnice in orodja za analizo ter razhroščevanje aplikacij. Knjižnice naj bi omogočale hiter razvoj aplikacij, kjer bi programer uporabljal že v naprej testirane in optimizirane OneAPI knjižnice, po razvoju, pa bi analiziral delovanje aplikacij z uporabo raznih orodij.

# OneAPI programski model

Kot omenjeno je programski model definiran na podlagi SYCL specifikacije. Ta specifikacija predstavlja heterogeno delovanje s štirimi osnovnimi modeli, aspekti katerih se pojavijo v vsakem DPC++ programu:

* Model platforme – navede gostitelja in naprave,
* Model izvajanja – navede ukazne vrste in izdane ukaze za izvajanje na napravah,
* Model pomnilnika – definira kako bosta gostitelj in ščepec dostopala do pomnilnika,
* Model ščepca – definira kodo, ki se bo izvajala na napravi.

Za tiste, ki ste že imeli izkušnje z OpenCL boste opazili podobnosti v sledečih modelih.

## Model platforme

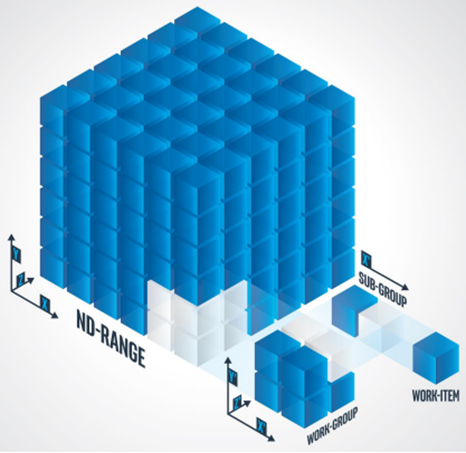
Navede gostitelja, ki bo upravljal z eno ali več napravami. Gostitelj, po navadi je to CPE, izvaja večji del programa, specifično aplikacijski obseg in obseg ukazne skupine. Torej usklajuje in upravlja z računskim delom, ki bo izveden na napravi. Naprava je specializirana komponenta zgrajena iz računskih virov, ki lahko hitro izvršijo podmnožico operacij. Tipično je to bolj učinkovito kot CPE-ji v sistemu. Vsaka naprava vsebuje eno ali več računskih enot, ki lahko izvršijo več operacij paralelno. Vsaka računska enota ima eno ali več procesnih enot (angl. Processing elements), ki služijo kot osnovna enota za izračune.

## Model izvajanja

Definira in navede kako so bo koda za ščepec, izvršila na gostitelju in napravah.

Izvajalni model za gostitelja usklajuje izvajanje in upravljanje s podatki med gostiteljem in napravo preko ukaznih skupin. Ukazne skupine so združevanja ukazov, kot so definicije ščepca in dostopnikov (angl. accessors), ki so potem poslani v vrsto za izvajanje. Na kratko, delo izvedemo na napravi, tako da ga pošljemo v vrsto. Vsaka vrsta pa je vezana na natanko eno napravo, ki jo lahko kot programer določimo sami ali pa prepustimo, da se ščepec izvede na poljubni napravi.

Izvajalni model za napravo pove, kako se bo računanje izvedlo na napravi. Računski problem, ki je lahko eno ali več dimenzionalen se alocira čez hierarhijo ND-obsegov (angl. ND-range), delovnih skupin (angl. Work-groups), podskupin (angl. Sub-groups) in delovnih enot (angl. Work-items), ki so vsi specificirani, ko delo pošljemo ukazni vrsti.



Slika 1: Relacije med različnimi hierarhijami v problemskem prostoru

## Model pomnilnika

Definira kako gostitelj in naprave dostopajo do pomnilnika. Usklajuje alokacijo in upravljanje pomnilnika med napravami in gostiteljem. Komunikacija poteka v dveh korakih:

1. Alociramo pomnilniški prostor kot bi ga normalno, potem pa še ustvarimo pomnilniški objekt, ki bo omogočal, da bo naš lokalno alociran pomnilnik bil poslan na napravo. Pomnilniški objekt je lahko tipa buffer ali pa tipa image,
2. Omogočimo interakcijo med gostiteljem in napravo preko dostopnika (angl. accessor), ki komunicira z ciljno lokacijo (gostitelj ali naprava) na izbran način dostopa (beri ali piši).

Tabela 1: Tabela vseh možnih ciljnih naprav, ko definiramo dostopnike do pomnilnika

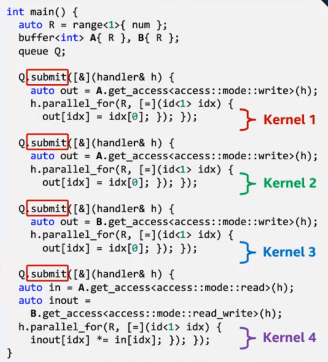
|  |  |
| --- | --- |
| **Ciljna naprava** | **Opis** |
| host\_buffer | Dostop do buffer-ja na gostitelju. |
| global\_buffer | Dostop do buffer-ja na napravi preko globalnega pomnilnika. |
| constant\_buffer | Dostop do buffer-ja iz konstantnega pomnilnika na napravi. |
| local | Dostop do buffer-ja iz konstantnega pomnilnika na napravi. |
| image | Dostop do slike. |
| image\_array | Dostop do polja slik. |
| host\_image | Dostop do slike na gostitelju. |

Tabela 2: Tabela vseh možnih dostopnih načinov

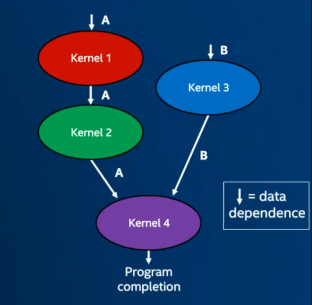
|  |  |
| --- | --- |
| **Način dostopa** | **Opis** |
| read | Samo branje. |
| write | Samo pisanje. |
| read\_write | Branje in pisanje. |
| discard\_write | Samo pisanje. Prejšnja vrednost je zavržena. |
| discard\_read\_write | Branje in pisanje. Prejšnja vrednost je zavržena. |
| atomic | Atomaren, eden naenkrat dostop. |

### Odvisnosti v dostopih do pomnilnika

Kadar imamo več ščepcev (angl. kernels), ki želijo dostopati do podatkov, ki so jih uporabljali ščepci pred njim, je potrebno čakati preden izvajanje nadaljuje naprej. Tako pride do odvisnosti podatkov.



Slika 2: Primer kode, kjer več ščepcev zahteva dostop do pomnilnika



Slika 3: Ustvarjen graf odvisnosti glede na zgornji program

Uporabljamo lahko dostopnike ali pa sami rezerviramo pomnilnik z malloc\_shared ter dostopamo kot do vsakega drugega rezerviranega pomnilnika (pointer\_na\_mem[i]), vendar moramo potem sami sinhronizirati dostop do pomnilnika. Če bi uporabljali dostopnike, bi se med izvajanjem zgradil graf odvisnosti in ustrezno poskrbel za sinhronizacijo in paralelno izvajanje ščepcev, ki ne potrebujejo istih podatkov.

## Model ščepca

Omogoča nedvoumen paralelizem med gostiteljem in napravo. Nedvomnost v smislu, da programer sam določi kateri del kode se bo izvajal na gostitelju in napravi, ni nekaj kar bi bilo avtomatizirano.

Razlikovanje kode za napravo in kode za gostitelja lahko storimo z uporabo enega izmed treh konstruktov:

* Lambda izraz – če želimo, da sta oba dela kode kot smiselna celota,
* Funktor – dela kode sta med seboj ločena, ampak sta še zmeraj v isti datoteki,
* Kernel class – bolj robustna ločitev med deloma kode.

Koda za napravo znotraj lambda izraza, funktorja ali kernel objekta lahko navede količino paralelizacije preko treh mehanizmov:

* Single\_task – izvede eno instanco ščepca na eni delovni enoti,
* Parallel\_for – paralelno izvede ščepec čez več procesnih enot,
* Parallel\_for\_work\_group – paralelno izvede ščepec čez več procesnih enot z uporabo lokalnega pomnilnika in pregrad.

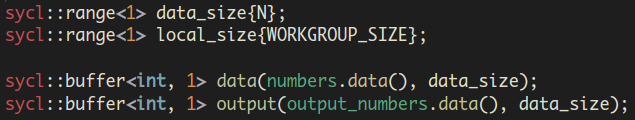
# Primeri kode

OneAPI je v končni fazi le »sintaktični sladkor« za programerja, saj temelji na istem konceptu kot OpenCL, le da nam prevajalnik sam poskrbi za »boilerplate« kodo. Še vedno imamo isti proces izvajanja: izbiro platforme, naprave, kreiranja izvajalne vrste, alokacijo pomnilnika in zagon ščepca.

1. Izbira platforme, naprave in ustvaritev izvajalne vrste

Najprej izberemo platformo in napravo. SYCL (na katerem je DPC++ zasnovan) nam omogoča, da prevajalnik sam izbere najbolj primerno naprav (defualt\_selector). Zatem na izbrani napravi ustvarimo izvajalno vrsto.

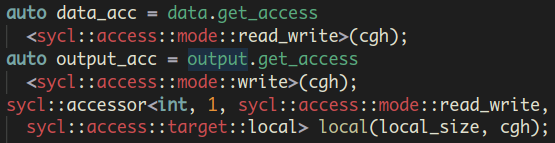
1. Alokacija pomnilnika

Dimenzije specificiramo na statično tipiziran način, kar zmanjša število napak. Tu že vnaprej določimo dimenzije vhodnih podatkov in velikost lokalne delovne skupine na pospeševalniku. Nato določimo še pomnilnik za vhodne in izhodne podatko, kjer sta podana tip in dimenzija (int, 1). Numbers in output\_numbers sta tipa std::array.

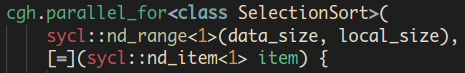
1. Ščepec dodamo v izvajalno vrsto

Ščepec dodamo v izvajalno vrsto, kot argument pa podamo lambda funkcijo, v kateri dobimo »handler« objekt. Preko njega sedaj komuniciramo z OpenCL »runtime« okoljem.

1. Ustvarjanje dostopnikov

Dostopniki do pomnilnika so zelo ugoden način dostopanja do pomnilnika tako na gostu kot na pospeševalniku. Prevajalnik lahko prek njih sam ugotovi zaporedje dostopov do pomnilnika in tudi sam sinhronizira ščepce. Ko ustvarjamo dostopnik podamo način dostopa do pomnilnika, če pa želimo dostop do lokalnega pomnilnika pa imamo kanček več dela, saj prevajalnik ne more iz že obstoječega objekta (npr. data/output) razbrati tipa podatkov in dimenzij, moramo to določiti sami.

1. Zagon ščepca

DPC++ ščepci so definirani kot lambda funkcije. Preko podanega »handlerja« poženemo ščepec, podati pa je potrebno število dimenzij ščepca, globalno velikost ščepca in velikost lokalne delovne skupine. Kot argument v lambda funkciji dobimo objekt, ki je istih dimenzij kot ščepec, preko katerega lahko kličemo vse OpenCL funkcije, ki so podprte znotraj ščepca. Npr.:

1. Izvajanje v ščepcu

Čeprav je ščepec lambda funkcija, ga lahko obravnavamo kot navaden C++ program, kjer pa se moramo seveda zavedati omejitev OpenCL. Če smo uporabljali dostopnike, je dostop do katerega koli pomnilnika enak kot dostop do pomnilnika na CPU. Težavo nam lahko predstavlja sinhronizacija, ki pa jo prav tako lahko zagotovimo z klicem funkcije barrier().

Po koncu ščepca so nam zapisani podatki vidni šele ko gredo sycl::buffer izven »scope«. Če pa do njih želimo dostopati že prej, moramo to storiti preko dostopnikov.

1. Header datoteke za »autocomplete«

Razvoj je seveda občutno lažji, če naš urejevalnik kode omogoča »autocomplete«. Vendar pa imamo lahko težavo najdi header datoteke, ki bi to omogočale. Najlažji način je, da ko prevajamo program, ali v Docker vsebniku ali direktno, poženemo prevajalnik z zastavico -H oz. -M. Le te namreč izpišeta pot do vseh header datotek, ki so vključene med prevajanjem. Sedaj lahko v urejevalniku te header datoteke podamo kot vir.

# Zaključek

Naši avtorji citiranje EV člankov izpuščajo – samo-zavora v svetu, ki tega ne pozna. So tudi nekatere svetle izjeme; z njihovim citiranjem objav v EV se omenja tudi revija EV. Z angleško izdajo bo to lažje in čez čas bomo ponovno povprašali za SCI indeks. Odgovor na vaše pogosto vprašanje “zakaj EV nima SCI”, je seveda na vaši strani. Sporočite citate in imeli boste krajši pripravljalni čas do objave.

# Literatura

1. Stran za Intel oneAPI projekt, https://software.intel.com/en-us/oneapi (20. 2. 2020).
2. Dokument za projektne naloge iz spletne učilnice FRI 19/20, https://ucilnica.fri.uni-lj.si/pluginfile.php/93827/mod\_resource/content/3/ProjektneNaloge2019.pdf