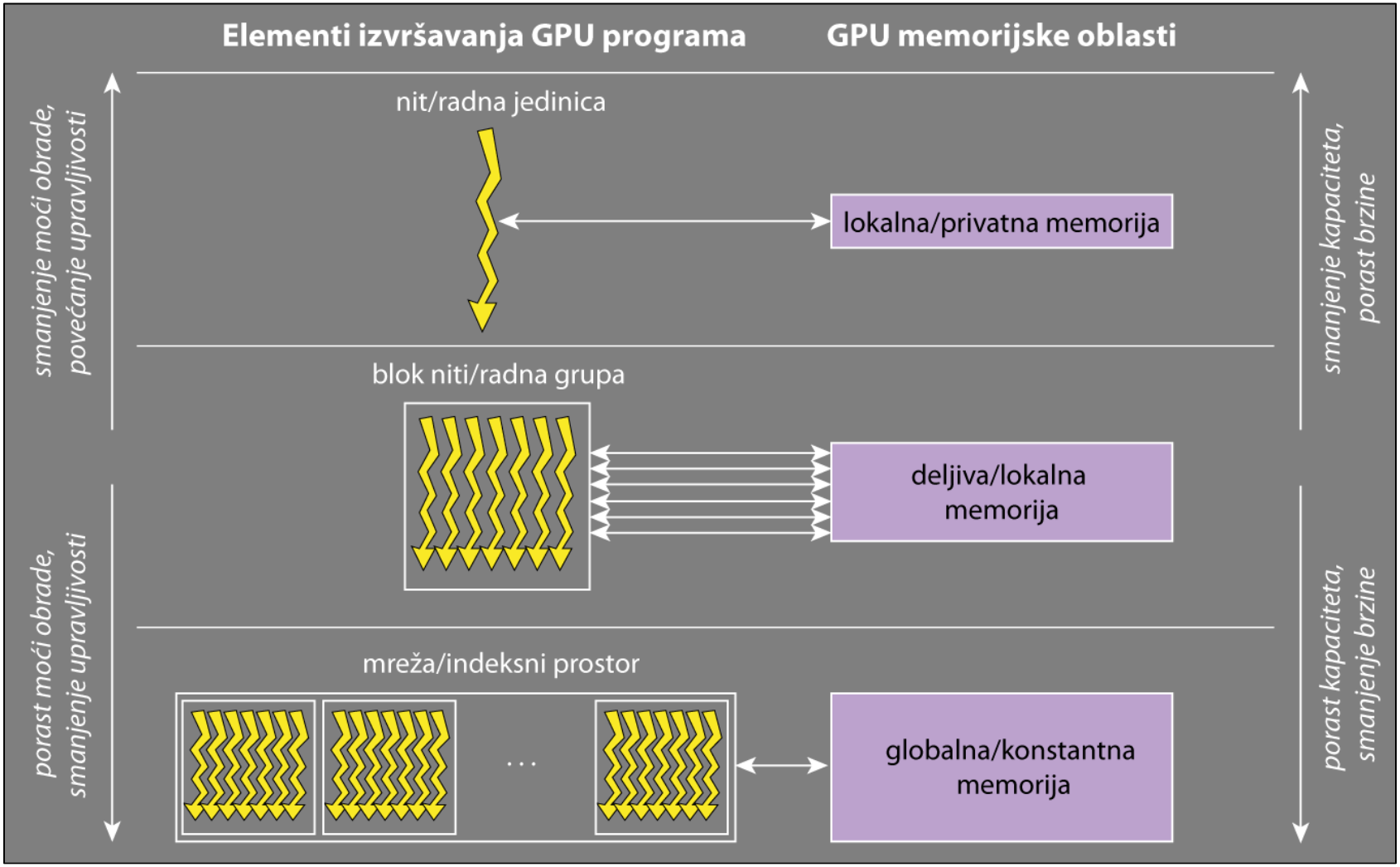
CUDA paralelno programiranje – optimizacije

Apstrakcije u GPGPU programiranju



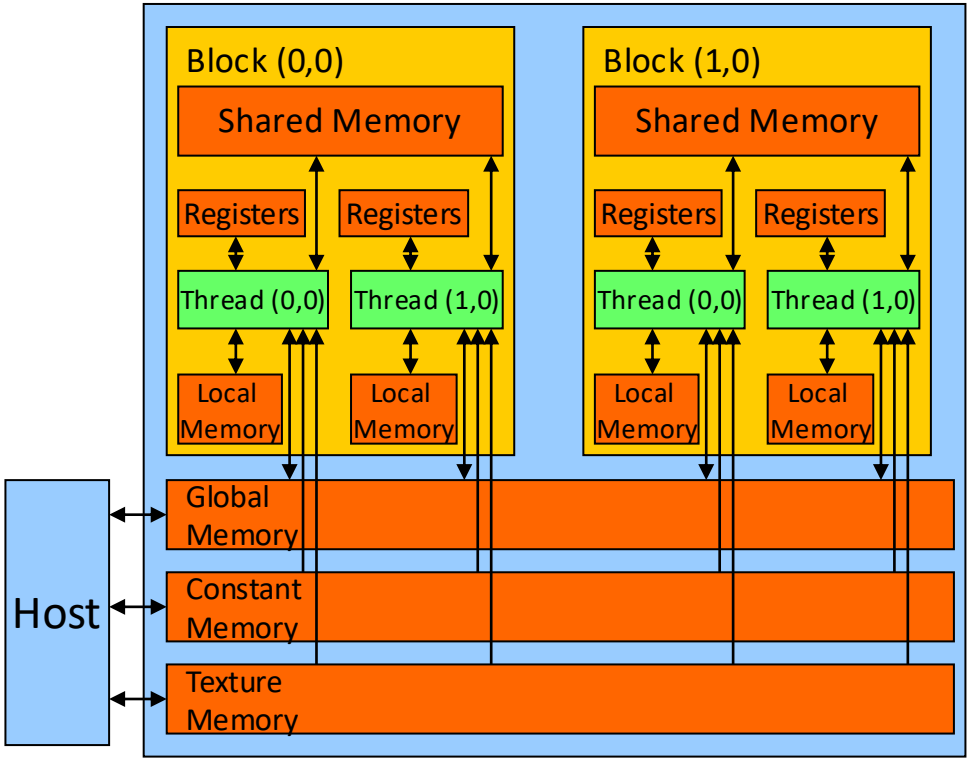
Optimizacija GPGPU programa

- **Visok propusni opseg** po cenu **produžene latencije** pokriva se **veoma velikim brojem niti**
- Fokus je na **memorijskim optimizacijama**
  - **memorijski transferi** – poravnati pristup (engl. *coalesced access*), memorija sa zaključanim stranicama (engl. *page-locked memory* ili *pinned memory*)
  - **eksplicitna deklaracija memorijskih regiona**
  - **broj registara po niti** – svi PP na PM dele isto registarsko polje (engl. *register field*)
- **Veličina mreže za izračunavanja** (engl. *computation grid*) – broj niti/bloku, broj bloka u mreži, …)
- **Zauzetost** (engl. *occupancy*) – broj konkurentnih osnova (warpova) po PM
- Problem **planiranja zadataka** (engl. *task scheduling*) postaje **složeniji – zadaci** se (optimalno) usmeravaju na **najpogodniji procesor**

CUDA memorijski model i regioni

- Memorijski regioni:
  - čitanje/upis po niti **registri (~1 ciklus)**
  - čitanje/upis po bloku **deljena memorija (~5 ciklusa)**
  - čitanje/upis po mreži **globalna memorija (~500 ciklusa)**
  - Samo čitanje po mreži **konstantna memorija (~5 ciklusa - keširanje)**

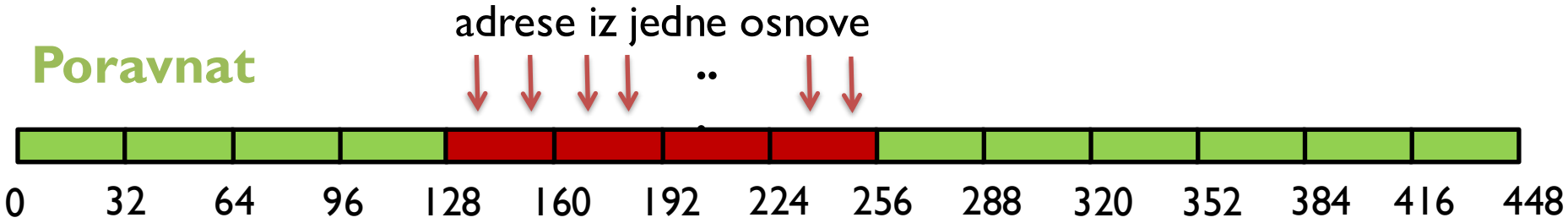
    __constant__

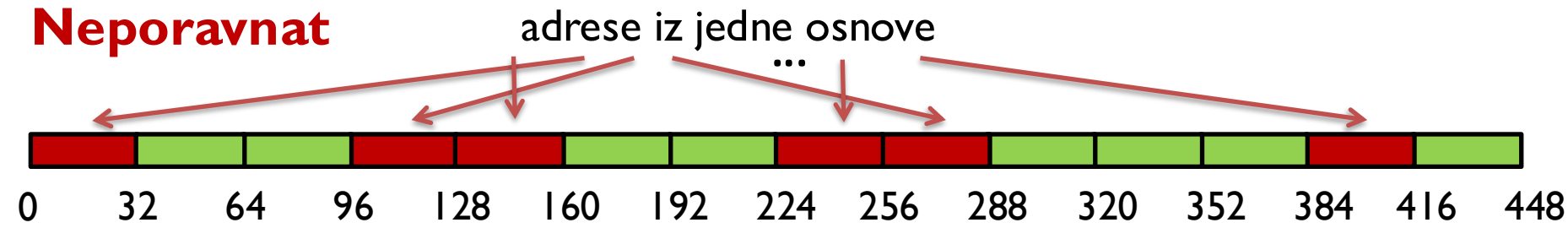


Izvor: Nvidia (https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)

Poravnat memorijski pristup

- **Operacije** sa **globalnom memorijom se izvršavaju na nivou osnove (warpa)**
  - 32 niti iz warpa šalju memorijske adrese
  - hardver određuje na koje linije te adrese dolaze

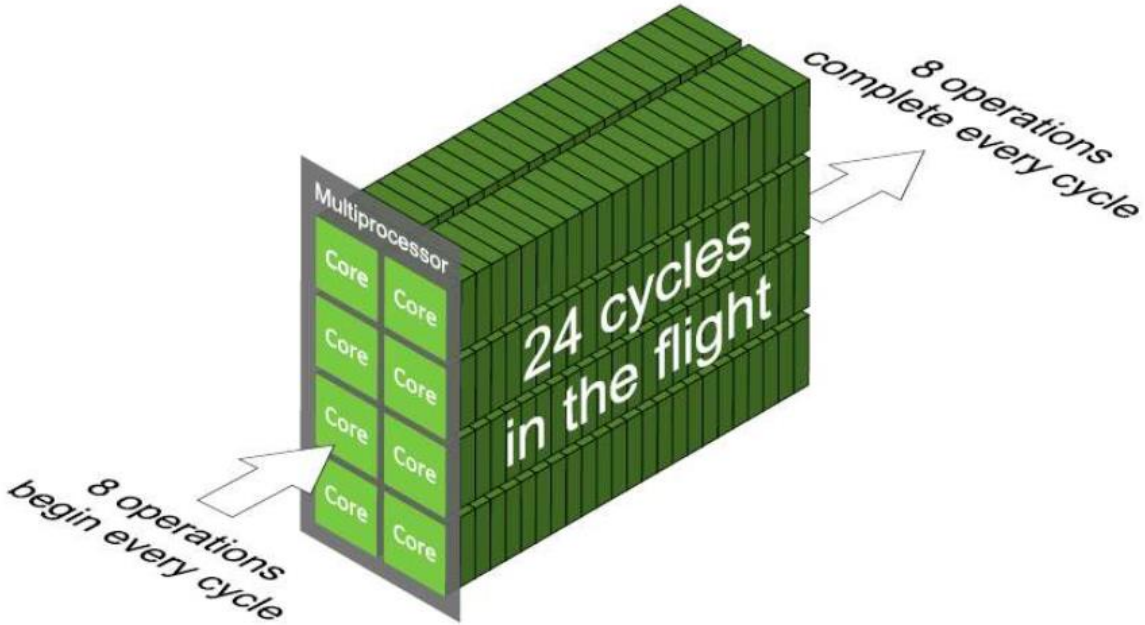




Memorija sa zaključanim stranicama

- **Memorija sa zaključanim stranicama** (engl. *page-locked/pinned memory*) – GPU zahteva transfere ka i iz memorije domaćina bez posredstva CPU
- Za podatke u virtuelnoj memoriji domaćina sa zaključanim stranicama je **garantovano da ne mogu biti izbačeni na disk**, mora se koristiti sa pažnjom, najviše 50% memorije domaćina na ovaj način
- CUDA funkcija cudaMallocHost
- Korišćenje memorije sa zaključanim stranicama omogućava višestruko brži prenos preko PCIe magistrale

Skrivanje latencije – Littleova jednačina



potreban paralelizam = latencija × propusni opseg

Primer za aritmetičke operacije na Nvidia Kepler:
20 ciklusa x 192 operacije po ciklusu po PM = potrebno 3840 operacija po PM tj. 120 osnova (manje operacija znači prazne (engl. *idle*) cikluse)

Izvor: Nvidia (https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)

Skrivanje latencije – zauzetost

- **Zauzetost** (engl. *occupancy*) prestavlja **odnosa broj aktivnih osnova i maksimalnog mogućeg broja osnova na protočnom multiprocesoru** (PM)
  - izražava se ili kao broj niti tj. osnova ili kao procenat od maksimalnog mogućeg broja niti (osnova)
  - CUDA Occupancy Calculator (deo Nsight Compute https://developer.nvidia.com/nsight-compute)
- Zavisi od više faktora:
  - broj registara po niti
  - količina deljene memorije po bloku
  - broj niti po bloku

Zauzetost i performanse

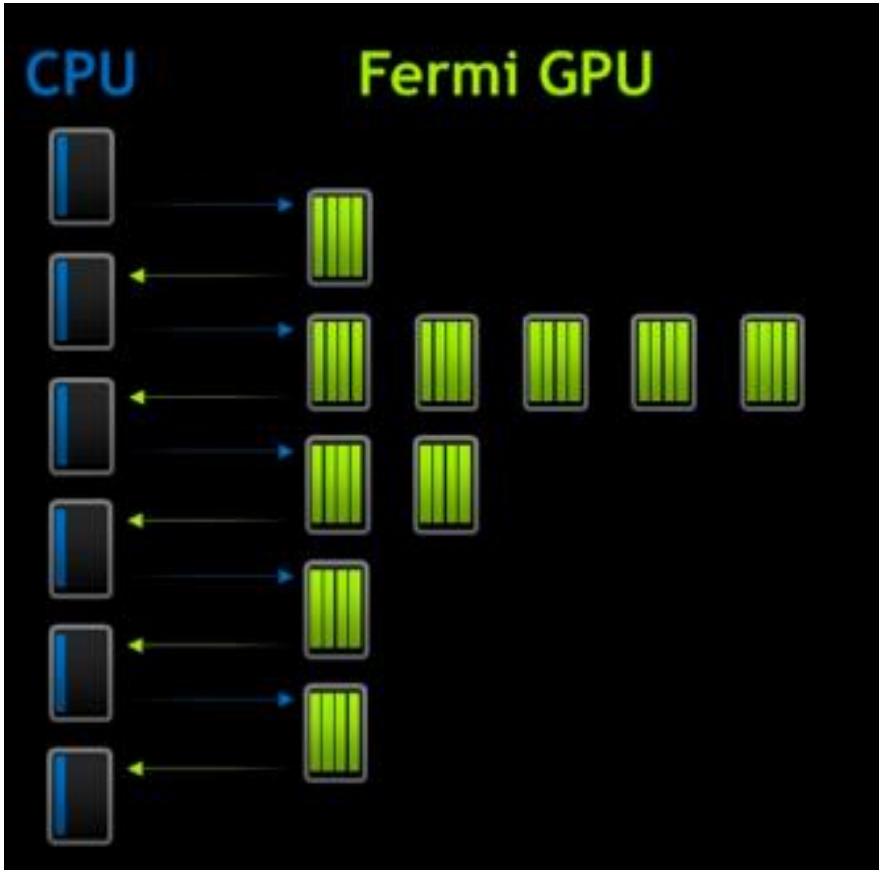
- **Puna zauzetost** (100%) nije neophodna kako bi se postigle maksimalne performanse
  - kada se neophodna zauzetost postigne, dalje povećanje ne povećava i performanse
- **Neophodna zauzetost** zavisi od problema koji se rešava
  - više nezavisnog posla po niti vodi do niže neophodne zauzetosti
  - programi ograničeni memorijskim pristupima zahtevaju višu zauzetost zbog pokrivanja latencija

Dinamički paralelizam

- Rekurzija nije bila moguća na GPU pre Kepler arhitekture (2012)
- Dinamički paralelizam uveden u CUDA 3.5
- Omogućava pokretanje novih funkcija na uređaju pozivom iz programa uređaja – GPU sam kreira novi posao za sebe
- Oslobađa CPU za rad na drugim zadacima, štedi energiju
- Omogućava različitu "rezoluciju" na različitim delovima podataka prilikom obrade na GPU

Dinamički paralelizam

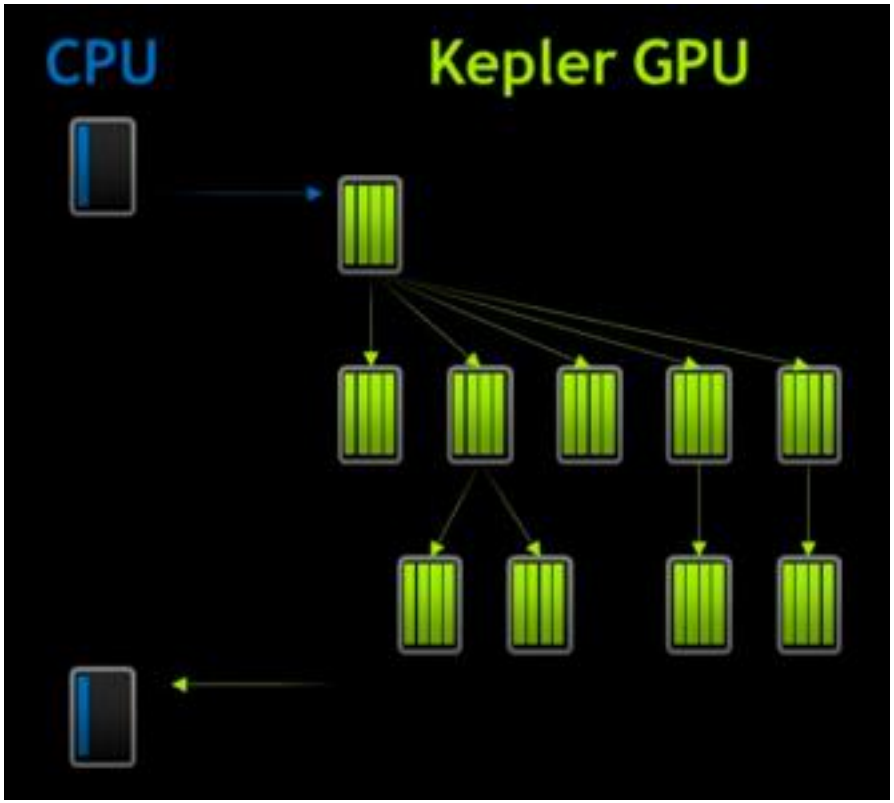
- Bez dinamičkog paralelizma:
  - Podaci putuju između CPU i GPU mnogo puta
  - Uzrok je nemogućnost GPU da sam kreira dodatne poslove zavisno od podataka



Izvor: Nvidia

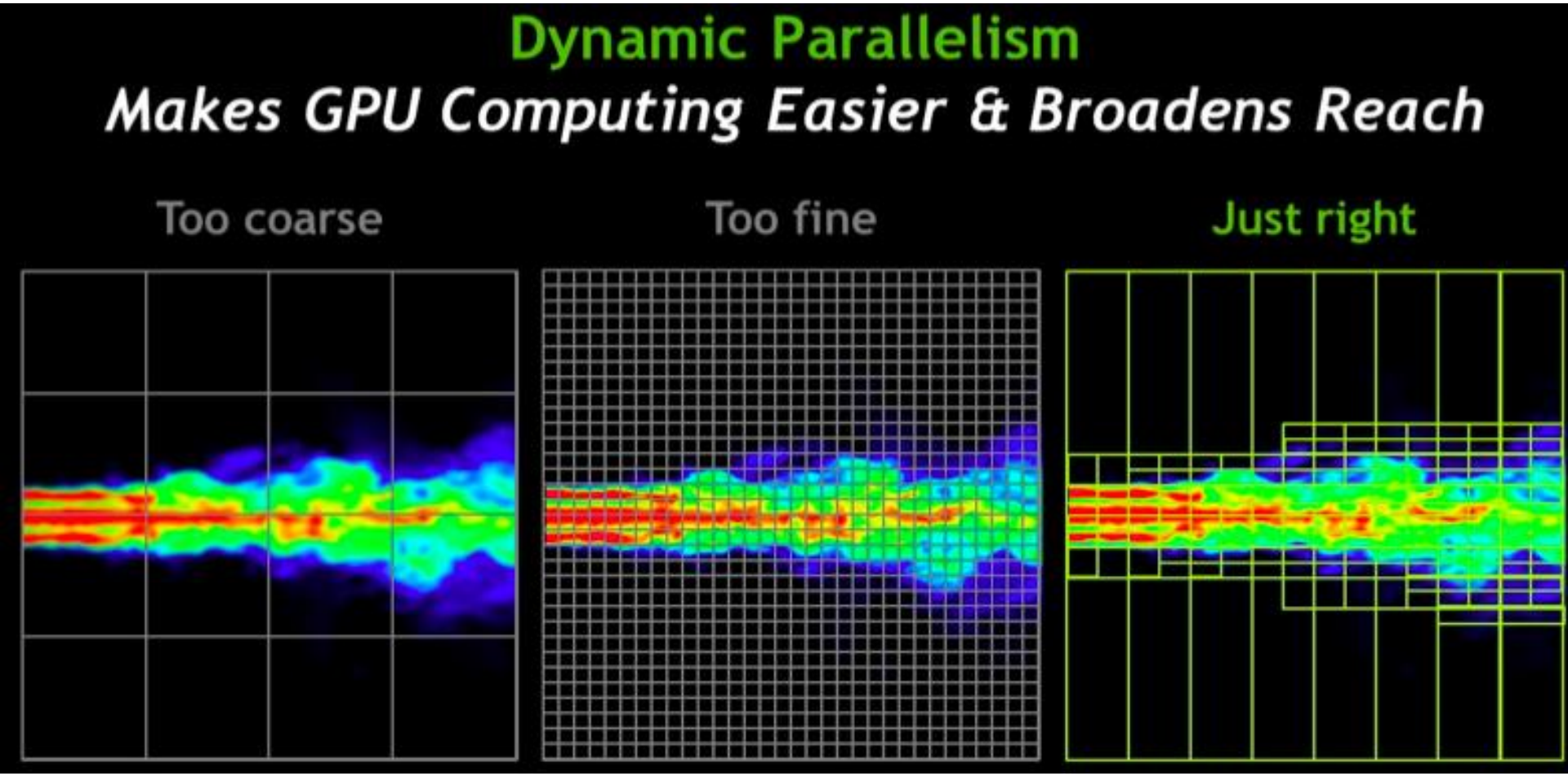
Dinamički paralelizam

- Sa dinamičkim paralelizmom:
  - Omogućava kodu uređaja da pozove kod uređaja, specifikator **__device__**
  - GPU može da sam sebi generiše posao na osnovu međurezultata, bez uključivanja CPU
  - Omogućava dinamičke odluke u vreme izvršavanja programa



Izvor: Nvidia

Primer – simulacija sa adaptivnom mrežom



Izvor: Nvidia

Primeri optimizacije CUDA paralelnih programa

Primer 1 – Hibridno CPU/GPU izračunavanje GFT

Izrazi na Galois poljima su generalizacija **Reed-Muller izraza** sa **binarne** na **višeznačnu logiku** (engl. *multiple-valued logic*)

$$f:\{0,1,\ldots,p-1\}^n \rightarrow \{0,1,\ldots,p-1\} \Rightarrow \mathbf{F}=[f(0),f(1),\ldots,f(p^n-1)]^T$$

$$\mathbf{S}_f=[s_f(0),s_f(1),\ldots,s_f(p^n-1)]^T \Rightarrow \mathbf{S}_f=\begin{bmatrix} transform \\ matrix \end{bmatrix}\cdot\mathbf{F} \Rightarrow O(N^2)$$

Brzi algoritmi se zanivaju na **faktorizaciji transfomacione matrice** na **retko-posednute matrice** $\Rightarrow O(N\log N)$

Osnovna ideja: Različiti delovi istog zadatka na **različitim procesorima?**

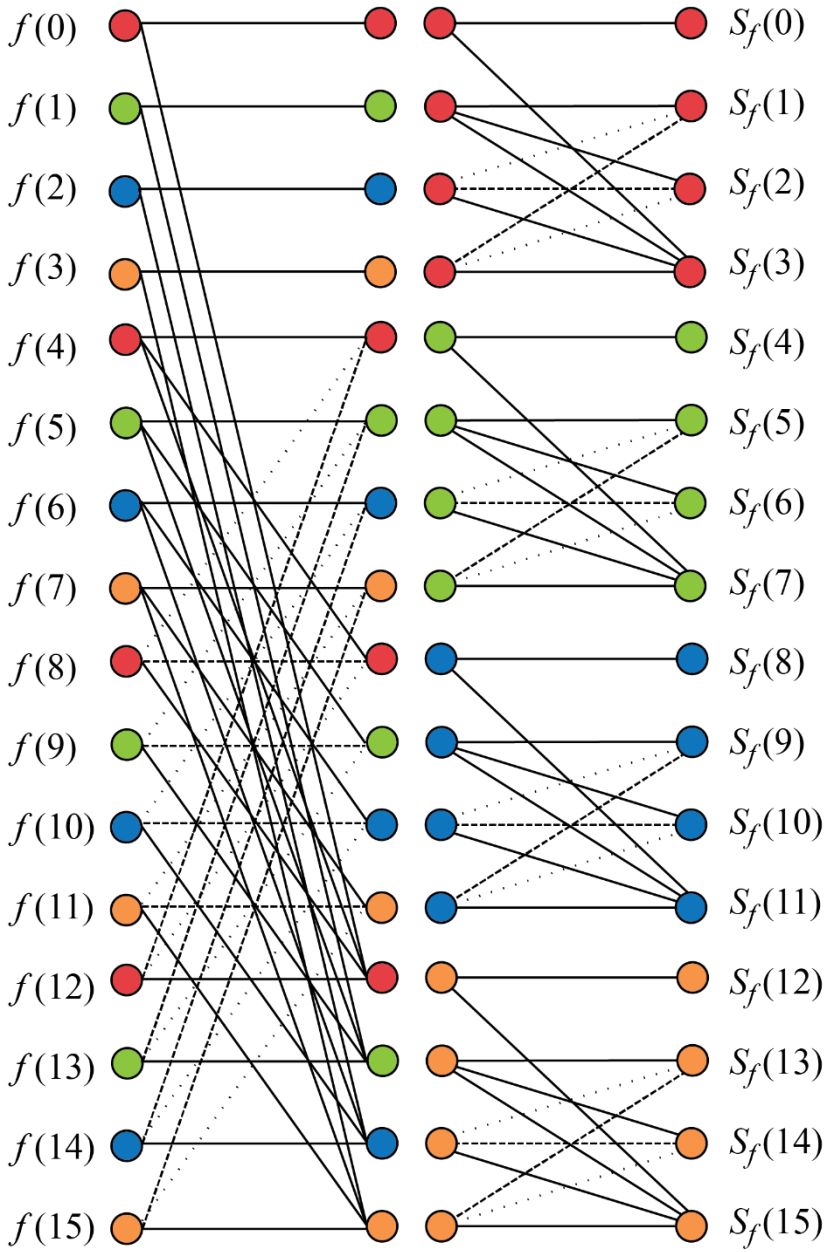
Izvršiti **izračunavanje nad različitim delovima vektora paralelno na CPU i GPU**

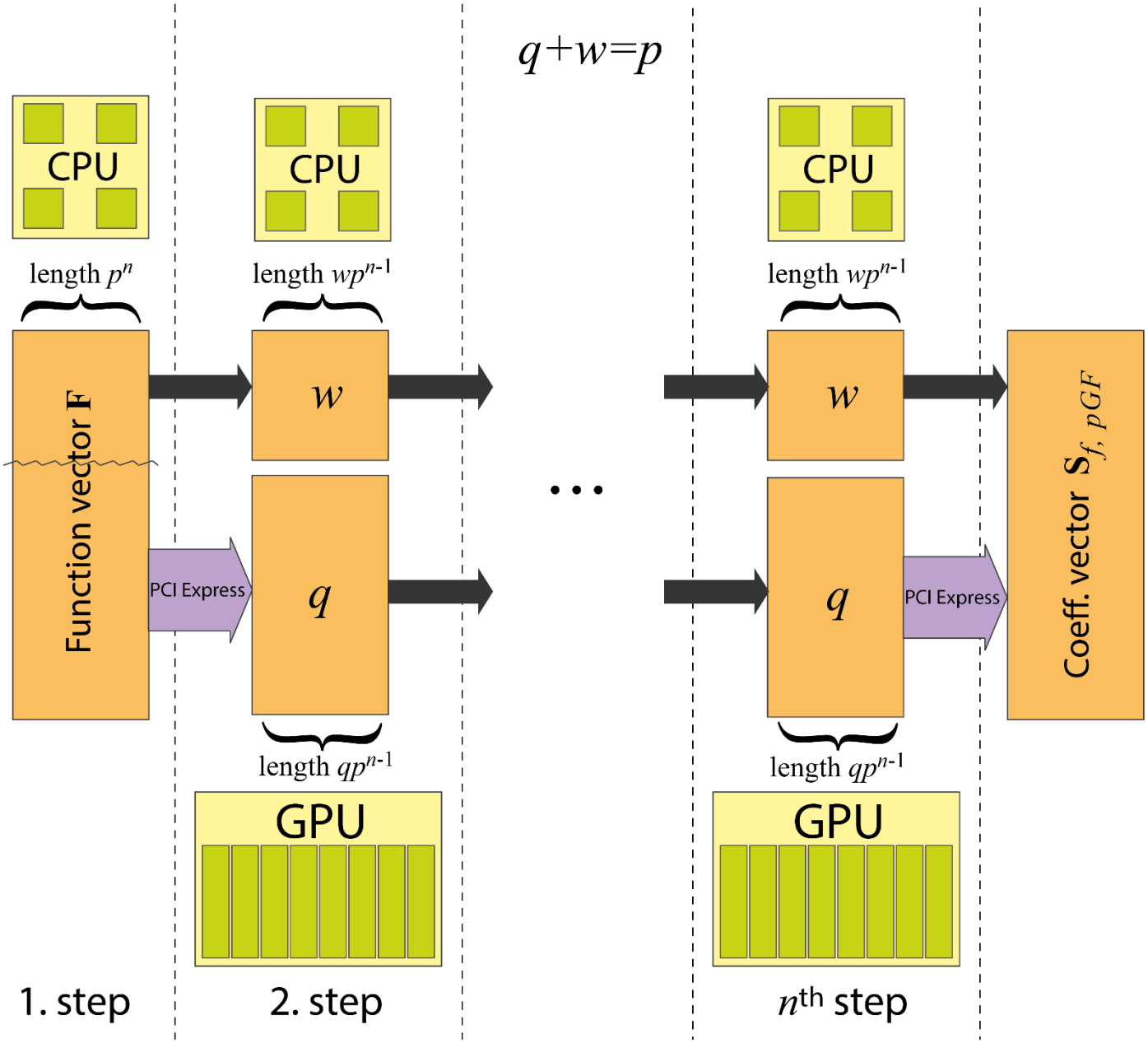
Transformaciona matrica za GF(4):

$$\mathbf{G}_{4GF}(1) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 3 & 2 \\ 0 & 1 & 2 & 3 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Cooley-Tukey faktorizacija:

$$\mathbf{C}_1 = \mathbf{G}_{4GF}(1) \otimes \mathbf{I}$$
$$\mathbf{C}_2 = \mathbf{I} \otimes \mathbf{G}_{4GF}(1)$$




$q+w=p$
CPU
CPU
CPU
length p^n
length wp^{n-1}
length wp^{n-1}
Function vector **F**
w
w
PCI Express
q
q
PCI Express
Coeff. vector $\mathbf{S}_{f,\,pGF}$
length qp^{n-1}
length qp^{n-1}
GPU
GPU
1. step
2. step
n^{th} step

Primer 2 – Gibbsovi izvodi na GPU

Gibbsovi izvodi su linearni diferencijalni **operatori** čije **sopstvene funkcije** (engl. *eigenfunctions*) su **karakteri** odgovarajuće **konačne grupe** na kojoj su definisani

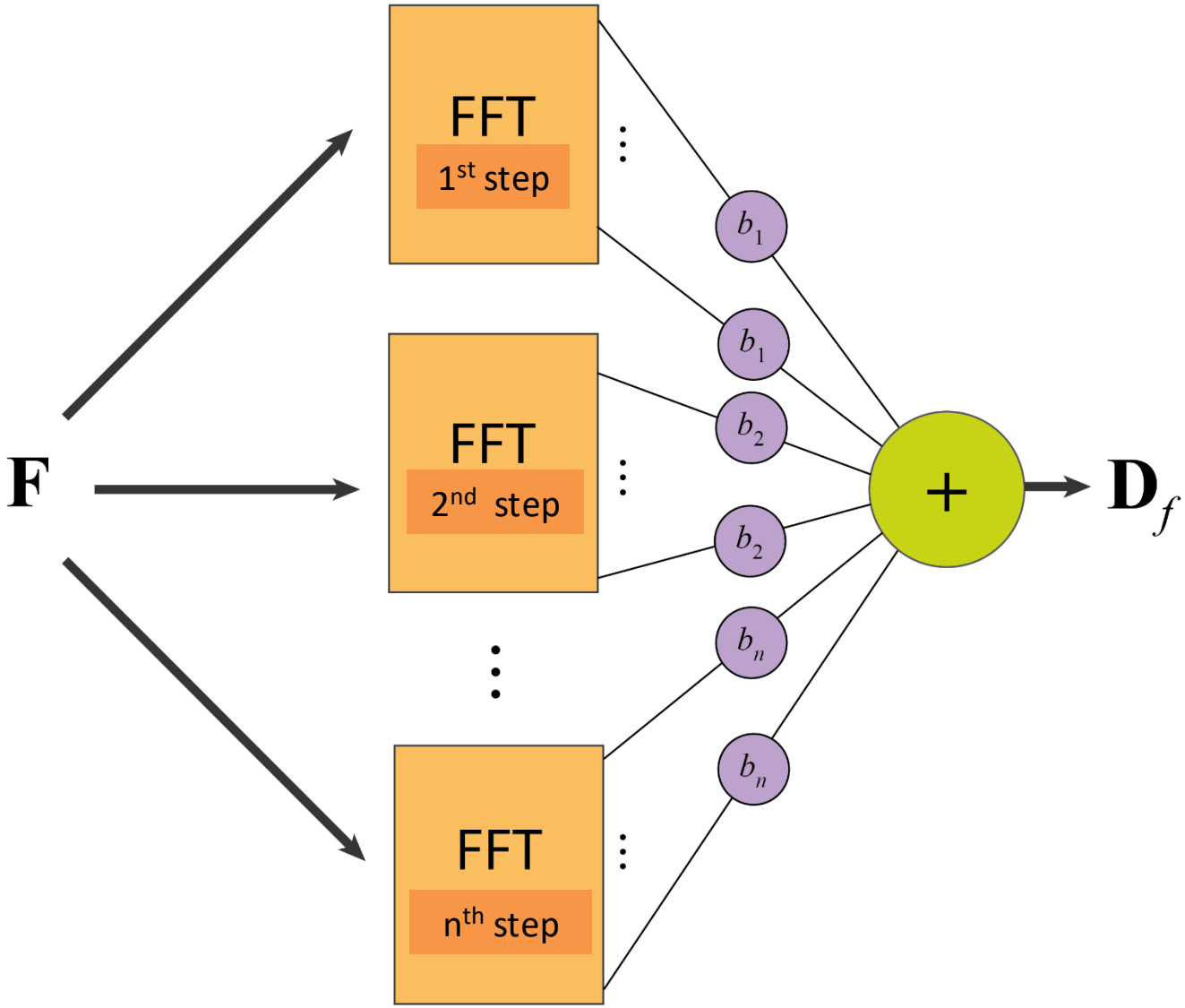
Odgovaraju **Newton-Leibnitz izvodu** na grupi realnih brojeva

Gibbs (1967) – izvod na konačnim dijadičkim grupama – operator čije su sopstvene funkcije Walshove funkcije

Primene: logičko projektovanje i **VLSI** (otkrivanje grešaka, funkcionalna dekompozicija, …), **modelovanje disipativnih dinamičkih sistema**, **fraktalna analiza**

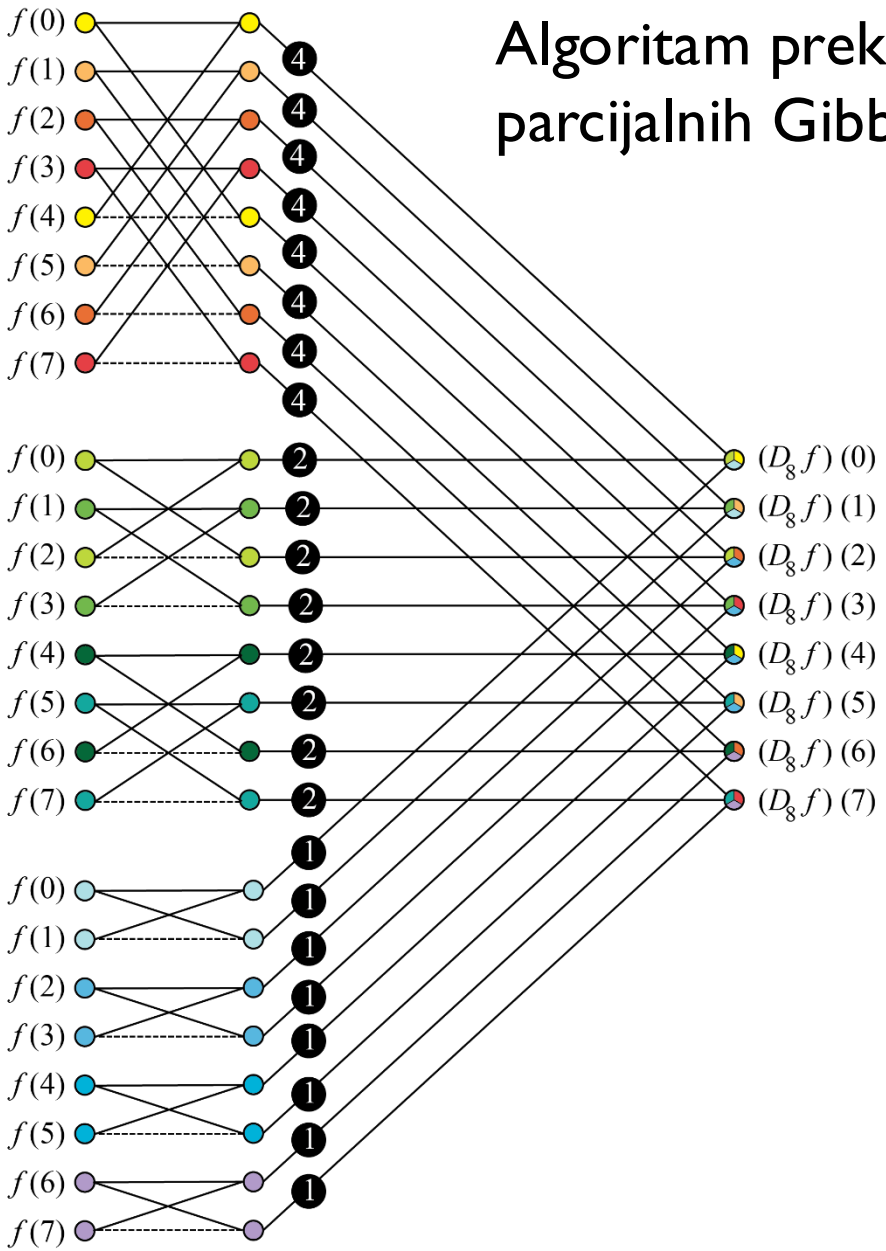
Konvolucioni algoritam i
algoritam preko parcijalnih Gibbs-ovih izvoda

Algoritam preko parcijalnih Gibbsovih izvoda



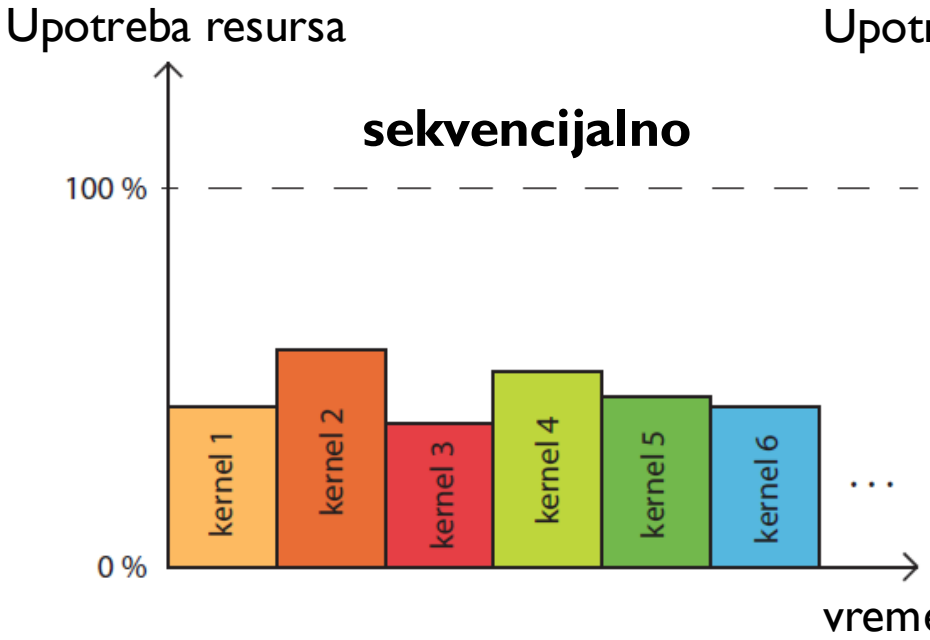
Gibbsov izvod $n = 3$

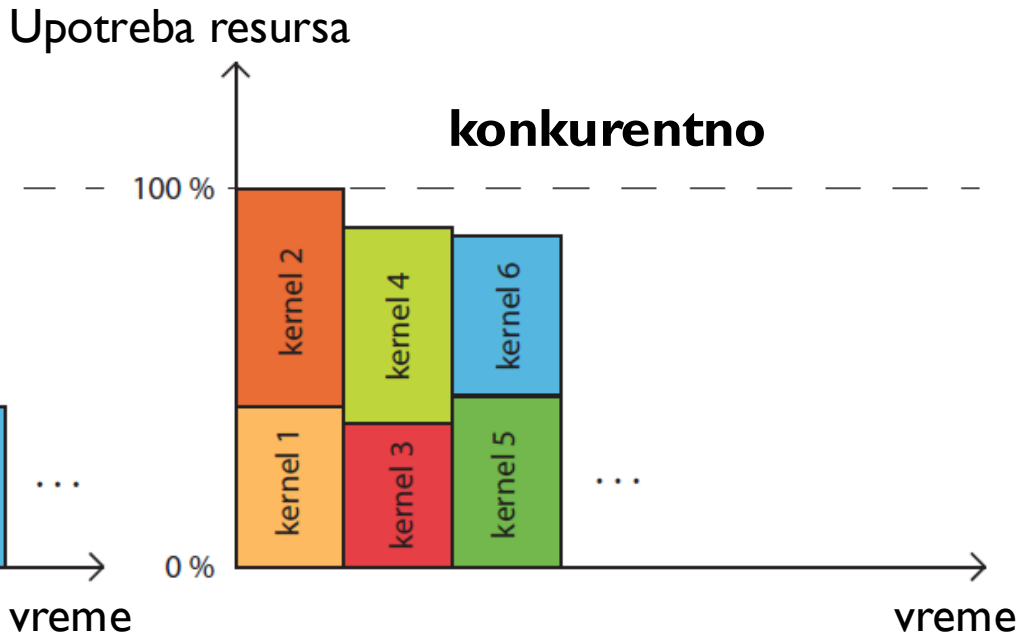
Algoritam preko parcijalnih Gibbsovih izvoda



Primer 2 – Gibbsovi izvodi na GPU

- Iskoristiti paralelizam i na nivou podataka i na nivou zadataka na GPU
- Konkurentno izvršavanje kernela primenom mehanizma CUDA tokova
- Upotreba atomičnih operacija za čitanje-izmenu-upis (engl. *read-modify-write*)
- Upotreba deljene memorije za međurezultate izračunavanja





Eksperimentalni rezultati

p	n	vreme izračunavanja [μs]							
		Nvidia Fermi GPU				Nvidia Maxwell GPU			
		data+task		data		data+task		data	
		veličina bloka [niti]							
		256	1024	256	1024	256	1024	256	1024
2	14	**24**	31	44	55	**32**	42	43	116
	18	590	652	**554**	852	**651**	653	761	755
	22	11173	12480	**9793**	15346	13018	**12927**	13194	13208
3	8	54	157	**30**	62	**31**	32	60	56
	10	502	502	**170**	181	**153**	155	155	159
	11	1622	1636	**493**	504	558	**557**	659	746

data+task = algoritam preko parcijalnih Gibbsovih izvoda
data = konvolucioni algoritam