# Računarski sistemi visokih performansi

Nikola Vukić, Veljko Petrović

Računarske vežbe
Zimski semestar 2025/26.

# Kokkos

# Sadržaj

- Zašto Kokkos?
- Kokkos programski model.
- Kokkos pogledi.
- Paralelna obrada podataka.
- Primer obrade 2D niza.
- Literatura.

# Zašto Kokkos?

# Zašto Kokkos?

- MPI kao jedini izvor paralelizma nije dovoljan.
- MPI ostaje kao odličan način komunikacije između čvorova, ali kada šta kada je u pitanju paralelizacija na nivou svakog čvora?
- Među čvorovima se pojavljuju široke klase uređaja:
  - Multicore CPU - jake serijske performanse, optimizacija za **latency**.
  - Many-core CPU - mnogo jezgara, slabije performanse na nivou pojedinačnog, optimizacija za **throughput**.
  - GP-GPU - postiže performanse kroz ogromne količine paralelizma.

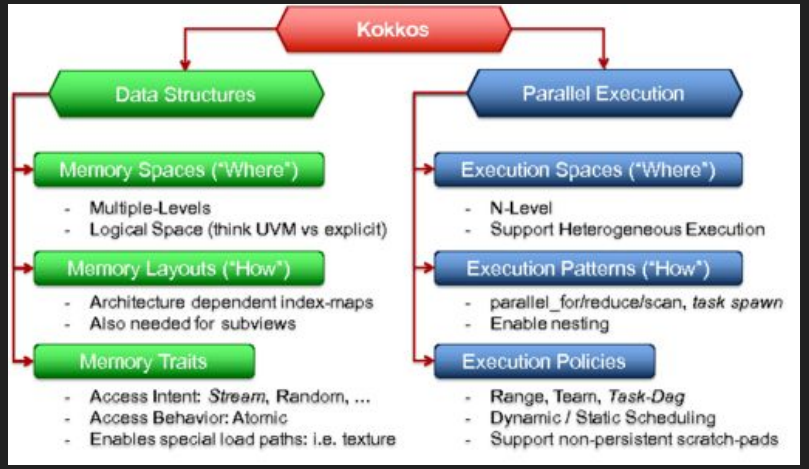
# Zašto Kokkos?

- Na jednom čvoru se mogu očekivati i kombinacije opcija.
- Sistemi više **nisu homogeni**.
- Programski modeli su mnogobrojni: OpenMP, pthreads, OpenACC, OpenCL, CUDA…
- Kako programirati za ovakve sisteme?
- Glavna misija Kokkos programskog modela je rešenje portabilnosti: isti kod, samo buildovan za različite uređaje, postiže **visoke performanse** na tim uređajima.

# Kokkos programski model

# Kokkos programski model

- 6 ključnih apstrakcija:
  - Execution Spaces
  - Execution Patterns
  - Execution Policies
  - Memory Spaces
  - Memory Layout
  - Memory Traits



# Kokkos programski model

- Execution Spaces (polja izvršavanja) - gde će se kod izvršavati. CPU ili GPU. Kako na CPU (OMP, threads, serijski), kako na GPU (CUPA, HIP, SYCL…).
- Execution Patterns (šablon izvršavanja) - način na koji će se postizati paralelizacija: *parallel_for*, *parallel_reduce*, *parallel_scan*, *task*.
- Execution Policies (polise izvršavanja) - način na koji će se izvršavati određeni šablon: range ili team, više na narednom slajdu.
- Memory Spaces (polje memorije) - gde se alocira memorija.
- Memory Layout (raspored memorije) - kako će se podaci organizovati u memoriji. više o ovome kasnije.
- Memory Traits (osobine memorije) - kako će se memoriji pristupati: nasumično, atomično, omogućava kompajlerske optimizacije.

# Kokkos programski model

- **Range** execution policy niti dobijaju određeni opseg elemenata koje obrađuju.
- **Team** execution policy:
  - omogućava hijerarski paralelizam, više niti čini tim. Svi timovi čine *league size*, nešto kao grid. Broj niti u timu, *team size*, ograničen hardverskim mogućnostima.
  - niti iz istog tima je moguće sinhronizovati pomoću barijere.
  - niti iz istog tima imaju pristup *scratch* memoriji koja je brza, ekvivalent *shared* memoriji u CUDA programskom modelu.
  - Za više informacija pročitati docs.

# Kokkos pogledi

# Kokkos pogledi

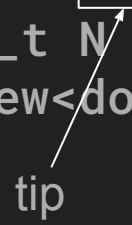
- Miks karakteristika jezika niskog nivoa (organizacija memorije, da bi se izbeglo lažno deljenje ili neporavnat pristup) i algoritama pisanih u jezicima visokog nivoa čine kod teško portabilnim.
- Zbog ovoga Kokkos uvodi poglede, **multi-dimenzionalne nizove** koje sam optimizuje za specifične arhitekture pri *build*-ovanju.
- Koriste reference counting mehanizam za automatsku dealokaciju.

# Kokkos pogledi

```
const size_t N0 = ...;
const size_t N1 = ...;
const size_t N2 = ...;
const size_t N3 = ...;
Kokkos::View<int****> a ("ime pogleda", N0, N1, N2, N3);
const size_t N = ...;
Kokkos::View<double*[3]> b ("neko drugo ime", N);
```

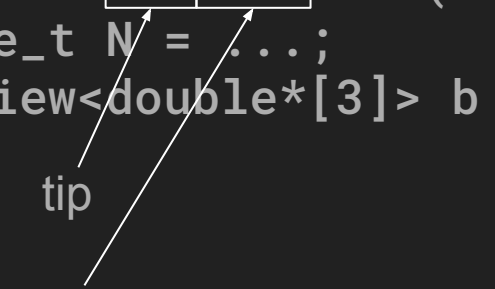
# Kokkos pogledi

```
const size_t N0 = ...;
const size_t N1 = ...;
const size_t N2 = ...;
const size_t N3 = ...;
Kokkos::View<int****> a ("ime pogleda", N0, N1, N2, N3);
const size_t N = ...;
Kokkos::View<double*[3]> b ("neko drugo ime", N);
```



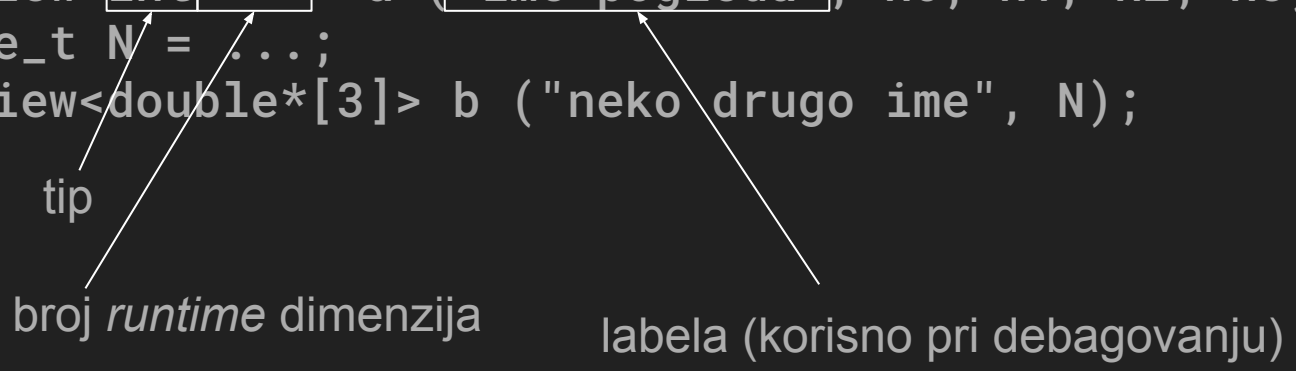
# Kokkos pogledi

```
const size_t N0 = ...;
const size_t N1 = ...;
const size_t N2 = ...;
const size_t N3 = ...;
Kokkos::View<int****> a ("ime pogleda", N0, N1, N2, N3);
const size_t N = ...;
Kokkos::View<double*[3]> b ("neko drugo ime", N);
```



# Kokkos pogledi

```
const size_t N0 = ...;
const size_t N1 = ...;
const size_t N2 = ...;
const size_t N3 = ...;
Kokkos::View<int****> a ("ime pogleda", N0, N1, N2, N3);
const size_t N = ...;
Kokkos::View<double*[3]> b ("neko drugo ime", N);
```



# Kokkos pogledi

```
const size_t N0 = ...;
const size_t N1 = ...;
const size_t N2 = ...;
const size_t N3 = ...;
Kokkos::View<int****> a ("ime pogleda", N0, N1, N2, N3);
const size_t N = ...;
Kokkos::View<double*[3]> b ("neko drugo ime", N);
```

veličine pojedinačnih dimenzija

tip

broj *runtime* dimenzija

labela (korisno pri debagovanju)

# Kokkos pogledi

```
const size_t N0 = ...;
const size_t N1 = ...;
const size_t N2 = ...;
const size_t N3 = ...;
Kokkos::View<int****> a ("ime pogleda", N0, N1, N2, N3);
const size_t N = ...;
Kokkos::View<double*[3]> b ("neko drugo ime", N);
```

broj *compile time* dimenzija

# Kokkos pogledi

- Kokkos nikada ne radi skrivenu deep kopiju, sve kopije su shallow, osim ukoliko se eksplicitno ne pozove funkcija koja ne radi deep kopiju.
- Znači da u code-snippetu ispod, nakon **a = b**, broj referenci na parče memorije na koje pokazuje **a** je 2, dok je broj referenci na koje pokazuje gde je pokazivalo **b** pre **a = b** pao na nula, pa se to parče memorije automatski dealocira.

```
Kokkos::View<int*> a ("a", 10);
Kokkos::View<int*> b ("b", 10);
a = b; // assignment does shallow copy
```

# Kokkos pogledi

- Pristup elementima ide preko običnih zagrada: A(0, 0) bi bio prvi element matrice A.
- Nekonstantni pogledi se mogu konvertovati u const, što omogućuje dodatne kompajlerske optimizacije.
- Lifetime počinje nakon konstruktora, završava nakon što broj referenci padne na nula, odnosno pogled se dealocira.
- Resize moguć pomoću Kokkos::resize, ovo oslobađa staru memoriju, ukoliko je pogled imao jednu referencu, ukoliko je imao više referenci, stare reference i dalje pokazuju na staru memoriju.
- Maksimalan broj dimenzija pogleda je 8.

# Kokkos pogledi — layout i dimenzije

- Uzmimo za primer dvodimenzionalni pogled:
  `Kokkos::View<int[3][3]> A ("A");`
- Recimo da ga popunimo brojevima od 1 do 9:

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

- U memoriji ga možemo organizovati po kolonama (column-major), u Kokkos-u se ovaj način organizacije naziva **LayoutLeft** (jer to stride-a dolazi pri promeni “levog indeksa” kada gledamo pristup), ili po redovima (row-major), odnosno **LayoutRight** (znači da elementi sa uzastopnim “desnim indeksom” su uzastopni i u memoriji).

# Kokkos pogledi — layout i dimenzije

- Treći tip layouta bi bio **LayoutStride** gde su uzastopni elementi razmaknuti u memoriji.
- I levi i desni layout se mogu generalizovati u strided layout (jer su oni specifični slučajevi strided layouta, sa korakom jedan).
- Za pristup dimenzijama se koristi **extent** polje pogleda, sa indeksom dimenzije za koju nas zanima dimenzija int dim1 = A.extent (0); vraća nultu dimenziju.
- Podrazumevani layout zavisi od execution space-a, tako *OpenMP* execution space kao podrazumevani layout ima **LayoutRight**, dok *Cuda* ima **LayoutLeft** kao podrazumevani layout. Zašto?

# Kokkos pogledi — layout i dimenzije

- Treći tip layouta bi bio **LayoutStride** gde su uzastopni elementi razmaknuti u memoriji.
- I levi i desni layout se mogu generalizovati u strided layout (jer su oni specifični slučajevi strided layouta, sa korakom jedan).
- Za pristup dimenzijama se koristi **extent** polje pogleda, sa indeksom dimenzije za koju nas zanima dimenzija int dim1 = A.extent (0); vraća nultu dimenziju.
- Podrazumevani layout zavisi od execution space-a, tako *OpenMP* execution space kao podrazumevani layout ima **LayoutRight**, dok *Cuda* ima **LayoutLeft** kao podrazumevani layout. Zašto?
- Layout je, naravno, moguće i eksplicitno zadati:
  Kokkos::View<int[3][3], Kokkos::LayoutLeft> A ("A");

# Paralelna obrada podataka

# Paralelna obrada podataka

- 3 vida paralelnih operacija:
  - *parallel_for* - paralelna for petlja sa nezavisnim iteracijama.
  - *parallel_reduce* - paralelna for petlja sa redukcijom.
  - parallel_scan - paralelni prefix scan.
- 2 opcije za definisanje tela za izvršavanje paralelnih petlji:
  - Lambde - pogodne za kraće petlje (koriste makro **KOKKOS_LAMBDA**).
  - Funktori - pogodni kada bi tela petlji bila duga (koriste makro **KOKKOS_INLINE_FUNCTION**).
- Funktor - struktura + const override operatora *()*:
  KOKKOS_INLINE_FUNCTION void operator() (...) const;
- Opciono se mogu definisati i *join* i *init* funkcije.
- Primer na iz docsa na linku.

# Paralelna obrada podataka

- Lambda (closure), uvedene u C++ 11 standardu.
- Anonimne funkcije svesne svog okruženja.
- Primer korišćenja redukcije sa lambdom:

```
const size_t N = ...;
View<double*> x ("x", N);
// ... fill x with some numbers ...
double sum = 0.0;
parallel_reduce ("Reduction", N, KOKKOS_LAMBDA (const int i, double&
update) {
  update += x(i);
}, sum);
// sum is now equal to sum of all the elements of x
```

# Primer obrade 2D niza

# Primer obrade 2D niza

```
1   #include <Kokkos_Core.hpp>
2
3   int main(int argc, char **argv) {
4       Kokkos::initialize(argc, argv);
5       {
6           Kokkos::View<int[3][3]> A("A");
7
8           for (int i = 0; i < 3; ++i) {
9               for (int j = 0; j < 3; ++j) {
10                  A(i, j) = i * 3 + j + 1;
11              }
12          }
13
14          for (int i = 0; i < 3; ++i) {
15              for (int j = 0; j < 3; ++j) {
16                  A(i, j) *= 2;
17              }
18          }
19      }
20      Kokkos::finalize();
21      return 0;
22  }
```

# Literatura

- Github: https://github.com/kokkos/kokkos
  - Repozitorijum sadrži *sve*.
  - Video lekcije, i prateće prezentacije.
  - Zadatke.
  - Primere korišćenja.
- Guide za programiranje (generalno upoznavanje): https://kokkos.org/kokkos-core-wiki/programmingguide.html
- API docs: https://kokkos.org/kokkos-core-wiki/api-references.html
- Oficijelna stranica: https://kokkos.org/