

# Paralelizacija algoritama sortiranja: Optimizacija performansi kroz paralelno računarstvo

Harun Goralija  
*Elektrotehnički fakultet*  
*Univerzitet u Sarajevu*  
Sarajevo, Bosna i Hercegovina  
hgoralija2@etf.unsa.ba

Belmin Durmo  
*Elektrotehnički fakultet*  
*Univerzitet u Sarajevu*  
Sarajevo, Bosna i Hercegovina  
bdurmo1@etf.unsa.ba

Harun Mioc  
*Elektrotehnički fakultet*  
*Univerzitet u Sarajevu*  
Sarajevo, Bosna i Hercegovina  
hmioc1@etf.unsa.ba

Amar Hodžić  
*Elektrotehnički fakultet*  
*Univerzitet u Sarajevu*  
Sarajevo, Bosna i Hercegovina  
ahodzic13@etf.unsa.ba

Kenan Abadžić  
*Elektrotehnički fakultet*  
*Univerzitet u Sarajevu*  
Sarajevo, Bosna i Hercegovina  
kabadzic1@etf.unsa.ba

**Sažetak**—Ovaj rad se bavi paralelizacijom algoritama sortiranja s ciljem optimizacije performansi kroz paralelno računarstvo. U eri velikih podataka, efikasno sortiranje velikih skupova podataka postaje ključno za aplikacije poput baza podataka i mašinskog učenja. Tradicionalni sekvencijalni algoritmi postaju usko grlo za velike skupove podataka.

Rad istražuje implementacije na višejezgrenim CPU-ima koristeći OpenMP i na GPU-ima koristeći CUDA. Implementirani su algoritmi: bitonic sort, merge sort, quick sort, radix sort i `std::sort` u sekvencijalnim i paralelnim varijantama.

Rezultati na 134 miliona elemenata pokazuju značajna ubrzanja, posebno za regularne algoritme poput radix sorta ( $21,6\times$  na CPU, 386 Melem/s na GPU). Poređenje CPU i GPU implementacija ističe komplementarnu prirodu dvije platforme: GPU nadmašuje CPU za masovno paralelne workloade, dok CPU pruža konkurentne performanse za manje skupove podataka bez overhead-a memorijskih transfera.

Rad doprinosi razumijevanju primjene paralelnih tehnika na algoritme sortiranja i pruža uvide u optimizaciju performansi na modernom heterogenom hardveru.

**Index Terms**—paralelizacija, algoritmi sortiranja, paralelno računarstvo, CPU, GPU, OpenMP, CUDA

## I. INTRODUCTION

U eri velikih podataka, efikasno sortiranje postaje temelj mnogih računarskih aplikacija. Tradicionalni sekvencijalni algoritmi postižu složenost  $O(n \log n)$ , što je optimalno za sekvencijalne mašine, ali postaju usko grlo za velike skupove podataka.

Paralelizacija algoritama sortiranja predstavlja prirodno rješenje. Iskorištavanjem više procesorskih jezgara ili GPU-a, moguće je značajno ubrzati sortiranje. Ovaj rad fokusira se na paralelizaciju pet algoritama: bitonic sort, merge sort, quick sort, radix sort i `std::sort`, sa implementacijama na CPU (OpenMP) i GPU (CUDA). Ukupno je implementirano 25 varijanti ( $5$  algoritama  $\times 5$  nivoa: naivni sekvencijalni, optimizovani sekvencijalni, paralelni CPU, naivni GPU i paralelni GPU) sa preko 6.700 linija koda.

Tema je relevantna jer paralelno računarstvo postaje neizbježno u aplikacijama poput analize velikih podataka i real-time procesiranja. Ključni izazov je mapiranje inherentno sekvencijalnih dijelova algoritama na masovno paralelne arhitekture, uz minimiziranje overhead-a sinhronizacije i memorijskih transfera.

Rad je organizovan na sljedeći način: Sekcija II daje pregled relevantne literature. Sekcija III opisuje implementaciju svih varijanti algoritama, uključujući sekvencijalne, optimizovane i paralelne verzije za CPU i GPU. Sekcija IV prezentira eksperimentalne rezultate sa detaljnim mjerenjima performansi. Sekcija V diskutuje rezultate i ograničenja, a Sekcija VI zaključuje rad.

## II. LITERATURE REVIEW

Paralelno sortiranje je aktivno istraživano područje od 1960-ih, kada je Batcher uveo bitonic sort sa  $O(\log^2 n)$  složenosti [1]. Moderna istraživanja fokusiraju se na GPU implementacije, gdje algoritmi poput bitonic sorta postižu značajna ubrzanja zahvaljujući SIMD arhitekturi [5]. Merge sort se dobro paralelizuje divide-and-conquer pristupom na višejezgrenim CPU-ima [3], dok quicksort izazove predstavlja zbog nebalansiranih particija, ali pokazuje dobre rezultate na GPU-ima za velike skupove podataka [2]. Radix sort, kao nekomparativni algoritam, postiže linearnu složenost i izuzetne performanse na GPU-ima [4].

Literatura naglašava prednosti heterogenog računarstva, gdje GPU implementacije nadmašuju CPU za masovno paralelne workload-e, ali CPU pruža bolju skalabilnost za manje podatke. NVIDIA Thrust biblioteka [5] služi kao de facto standard za GPU sortiranje i pruža visokooptimizovane implementacije koje iskorištavaju specifične karakteristike svake GPU arhitekture. Ovaj rad doprinosi polju pružajući direktne implementacije i poređenja performansi pet algoritama na CPU (OpenMP) i GPU (CUDA) platformama, adresirajući praznine u sveobuhvatnim studijama paralelizacije.

### III. IMPLEMENTATION

Projekat implementira pet algoritama sortiranja: bitonic sort, merge sort, quick sort, radix sort i `std::sort`. Svaki algoritam je realizovan u tri varijante: sekvencijalna (naivna), sekvencijalna optimizovana i paralelna CPU verzija koristeći OpenMP sa SIMD instrukcijama.

#### A. Arhitektura projekta

Implementacija je organizovana u modularnoj strukturi sa zajedničkim header fajlovima (`main_template.hpp`, `timers.hpp`, `utils.hpp`) koji pružaju infrastrukturu za testiranje, mjerenje performansi i generisanje testnih podataka. Svaki algoritam ima svoj wrapper koji se integriše sa sistemom za benchmarking. Build sistem koristi CMake sa Ninja generatorom, a CPU kod se kompajlira sa GCC 13.2 uz `-O3 -march=native -mavx2` zastavice za maksimalnu optimizaciju.

#### B. Sekvencijalne implementacije

Sekvencijalne verzije služe kao baseline za poređenje performansi. Naivne implementacije koriste standardne rekurzivne pristupe: quick sort sa Lomuto particioniranjem, merge sort sa rekurzivnim spajanjem, bitonic sort sa rekurzivnim compare-and-swap mrežama, radix sort sa base-10 counting sortom po cifri, i standardni `std::sort`.

Optimizovane verzije uvode niz poboljšanja. Quick sort koristi Hoare particioniranje sa median-of-three pivotom i tail rekurzijom, sa prelaskom na insertion sort za nizove manje od 24 elementa. Merge sort prelazi na iterativni pristup sa ping-pong double-buffering tehnikom koja eliminira rekurziju i nepotrebno kopiranje. Radix sort koristi base-256 shemu (8-bitni prolazi) umjesto base-10, uz preskakanje prolaza kada su svi elementi jednaki na datom byte-u, što značajno ubrzava sortirane i blizu-sortirane nizove.

#### C. Paralelne CPU implementacije

Paralelne verzije koriste OpenMP za višezvezgenu paralelizaciju. Ključne optimizacije uključuju:

**Bitonic sort** kombinuje SIMD instrukcije (AVX2) za vektorizovane compare-and-swap operacije (`_mm256_min_epi32`, `_mm256_max_epi32`) sa OpenMP task paralelizmom za velike nizove. Procesira 8 parova elemenata po SIMD instrukciji i koristi insertion sort za sekvence manje od 128 elemenata.

**Merge sort** koristi Merge-Path algoritam koji binarnom pretragom dijeli posao ravnomjerno među nitima, čime se izbjegava problem nebalansiranog spajanja kod standardnog pristupa. Implementira 64-byte poravnanu alokaciju i AVX2-ubrzani `simd_memcpy` za kopiranje podataka.

**Quick sort** implementira Dutch National Flag 3-way particioniranje za efikasno rukovanje duplikatima, sa AVX2 vektorskim prefetch-om i OpenMP task-ovima koji se dinamički kreiraju samo za particije veće od praga.

**Radix sort** koristi thread-local histograme za eliminaciju false sharing-a između niti. Svaka nit akumulira lokalni histogram za svoj dio niza, nakon čega se histogrami kombinuju u

globalne prefiksne sume. Base-256 shema sa 4 prolaza (po 8 bitova) minimizira broj sinhronizacija, dok SIMD instrukcije ubrzavaju scatter fazu. Ova kombinacija čini radix sort najbrži paralelni algoritam na CPU-u.

**std::sort** paralelizuje postojeću C++ standardnu biblioteku koristeći `std::execution::par_unseq` execution policy iz C++17 standarda, što kompajleru omogućava automatsku vektorizaciju i paralelizaciju bez eksplicitnog upravljanja nitima.

Implementacije koriste pragme kao `#pragma omp parallel for` i `#pragma omp task` za eksplicitnu paralelizaciju, sa pažljivim balansiranjem opterećenja i minimizacijom overhead-a sinhronizacije. Prag za kreiranje novih task-ova je ključan parametar: premali prag uzrokuje prevelik overhead kreiranja zadataka, dok preveliki prag rezultira nedovoljnom paralelizacijom.

#### D. GPU implementacije

GPU verzije koriste NVIDIA CUDA platformu za masivnu paralelizaciju na grafičkom procesoru RTX 3050 Ti Laptop GPU sa 20 streaming multiprocссора (SM) i compute capability 8.6. CUDA programski model organizuje niti u hijerarhiju: niti se grupišu u warp-ove (32 niti), blokove i grid, što omogućava fino upravljanje paralelizmom. Ključni optimizacijski ciljevi su koalescentan pristup memoriji, korištenje shared memorije i maksimalan occupancy.

**Bitonic sort** koristi dvofazni pristup. U prvoj fazi, shared memory kernel sortira blokove od 512 elemenata koristeći `__shared__` memoriju za brze compare-and-swap operacije unutar bloka. U drugoj fazi, za korake  $k > 512$ , koristi se global memory kernel sa grid-stride petljom koji sakriva latenciju pristupa globalnoj memoriji. Ulazni niz se dopunjava do sljedeće potencije broja 2 sa `INT_MAX` vrijednostima.

**Merge sort** kombinuje bitonic sort kao bazni slučaj sa paralelnim merge-path algoritmom. Početno sortiranje blokova od 512 elemenata vrši se istim shared memory bitonic kernelom. Zatim se sortirani blokovi spajaju paralelnim merge kernelom koji koristi binarnu pretragu za određivanje pozicije svakog elementa u spajanju, čime se postiže ravnomjerna raspodjela posla. Implementirane su tri varijante: `PerElement` (jedan element po niti), `Coarsened` (više elemenata po niti za sakrivanje latencije) i `BlockMerge`. Double-buffering tehnikom se izbjegava nepotrebno kopiranje.

**Quick sort** koristi hibridni CPU-GPU pristup (Iterative V2). CPU vrši bootstrap fazu koristeći BFS particioniranje dok se ne kreira oko 65.536 zadataka, čime se izbjegava sekvencijalno usko grlo prvog nivoa rekurzije. GPU kernel zatim procesira zadatke koristeći warp-agregiranu task queue sa `__ballot_sync` i `__shfl_sync` primitivama za efikasnu distribuciju posla. Mali segmenti ( $\leq 16$  elemenata) se sortiraju insertion sortom u registrima, a pivot se bira median-of-9 tehnikom za velike segmente.

**Radix sort** implementira LSB (Least Significant Bit) pristup procesiranjem po jednog bita u 32 prolaza. Count kernel koristi warp-level redukciju sa `__shfl_down_sync` za brojanje nula po bloku. Scan kernel računa prefix sume za globalne

offsete, a scatter kernel koristi `__ballot_sync` za efikasno raspoređivanje elemenata na izlazne pozicije. Implementacija se poredi sa NVIDIA Thrust bibliotekom kao baznom linijom.

Dodatno, implementirane su i naivne GPU verzije sa konfiguracijom `<<1, 1>>` (jedna nit) koje služe za demonstraciju overhead-a GPU kernel pokretanja i memorijskih transfera bez koristi od paralelizacije. Ove verzije su očekivano višestruko sporije od CPU naivnih implementacija, jer jedna GPU nit radi na nižoj frekvenciji (1,5 GHz) u poređenju sa CPU jezgrom (4,4 GHz), bez mogućnosti branch prediction-a i spekulativnog izvršavanja.

#### IV. EKSPERIMENTALNI REZULTATI

##### A. Testno okruženje

Eksperimenti su izvršeni na sistemu sa AMD Ryzen 7 5800H procesorom (8 jezgara, 16 niti) i NVIDIA GeForce RTX 3050 Ti Laptop GPU (20 SM-ova, compute capability 8.6, 4 GB GDDR6). Operativni sistem je Windows, CPU kod je kompajliran sa GCC 13.2 uz optimizacije `-O3 -march=native -mavx2` i OpenMP 4.5, dok su GPU kerneli kompajlirani sa NVCC za `sm_86` arhitekturu. Svi testovi koriste nasumično generirane nizove cijelih brojeva sa seed vrijednošću 12345 radi reproducibilnosti. CPU benchmarki su pokrenuti za veličine nizova od 16K do 128M elemenata, dok su GPU benchmarki izvršeni na 128M elemenata radi evaluacije performansi na velikim skupovima podataka.

##### B. CPU rezultati

Tabela I prikazuje vremena izvršavanja za 134.217.728 (128M) elemenata na CPU-u. Za bitonic sort naivnu verziju prikazano je vrijeme na 8M elemenata jer je  $O(n \log^2 n)$  složenost čini nepraktičnom za 128M.

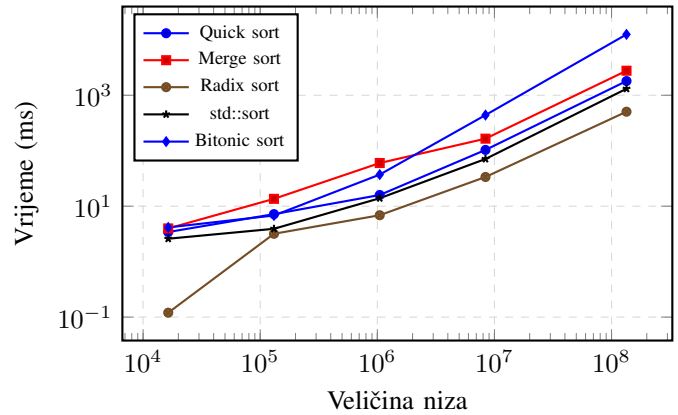
Tablica I  
CPU PERFORMANSE SORTIRANJA – 128M RANDOM ELEMENTI (MS)

Algoritam	Naivni	Opt.	Par. CPU	Ubrzanje
Bitonic sort	2026*	33608	12511	1,0×
Merge sort	26266	11386	2768	9,5×
Quick sort	12168	9647	1803	6,7×
Radix sort	10885	1507	505	21,6×
std::sort	8520	8342	1310	6,5×

\* Bitonic naivni mjereno na 8M elemenata

Rezultati pokazuju da optimizirane sekvencijalne verzije donose značajna poboljšanja, posebno za radix sort koji koristi base-256 umjesto base-10 i preskakanje uniformnih prolaza, čime postiže ubrzanje od 7,2× u odnosu na naivnu verziju. Optimizovani merge sort koristi iterativni pristup sa double-buffering tehnikom umjesto rekurzije, što smanjuje overhead i rezultira ubrzanjem od 2,3×. Paralelizacija sa OpenMP i AVX2 instrukcijama pruža dodatno značajno ubrzanje: radix sort postiže impresivnih 21,6× u odnosu na naivnu verziju, dok merge sort postiže 9,5×. Standardni `std::sort` sa `par_unseq` policy-em postiže 6,5× ubrzanje uz minimalan napor implementacije, što ga čini atraktivnom opcijom za aplikacije gdje je jednostavnost prioritet.

Slika 1 prikazuje skalabilnost paralelnih CPU implementacija u zavisnosti od veličine niza. Svi algoritmi pokazuju očekivano linearitmičko ponašanje na log-log skali, pri čemu radix sort konzistentno nadmašuje ostale algoritme na svim veličinama zahvaljujući linearnoj složenosti  $O(nk)$ . Za male nizove ( $< 128K$ ), overhead paralelizacije je vidljiv kod svih algoritama osim radix sorta, gdje thread-local histogrami minimiziraju sinhronizacijski trošak.



Slika 1. Skalabilnost paralelnih CPU implementacija (log-log skala)

##### C. GPU rezultati

Tabela II prikazuje performanse GPU implementacija za 128M elemenata. Prikazane su najbolje konfiguracije za svaki algoritam nakon opsežnog testiranja različitih broja niti po bloku (64–1024) i varijanti kernela.

Tablica II  
GPU PERFORMANSE SORTIRANJA – 128M RANDOM ELEMENTI

Algoritam	Varijanta	Kernel (ms)	Ukupno (ms)	Melem/s
Bitonic	Shared+Stride	34	50	86
Merge	PerElement E=8	383	430	312
Quick	Iterative V2	811	1021	131
Radix	Custom 256 niti	187	347	386
Thrust	Baseline	49	309	434

Thrust biblioteka postiže najbolji ukupni throughput od 434 Melem/s zahvaljujući visokooptimiziranim NVIDIA implementacijama. Među vlastitim implementacijama, radix sort postiže najbolje kernel vrijeme (187 ms) zahvaljujući regularnim pristupima memoriji, dok bitonic sort ima najkraće ukupno vrijeme (50 ms) jer ima mali broj kernel poziva i efikasan grid-stride pristup. Quick sort na GPU-u pokazuje najslabije rezultate (1021 ms) zbog neregularnog workloada, ali hibridni pristup sa CPU bootstrap fazom značajno poboljšava performanse u odnosu na prethodne varijante (Batched: 5187 ms, Fat: 4481 ms). Evolucija quick sort implementacije pokazuje da je izbor arhitekture kernela kritičan: Batched varijanta koja pokreće jedan kernel po nivou rekurzije pati od overhead-a pokretanja kernela, Fat varijanta sa velikom shared

memorijom gubi na occupancy-u, dok Iterative V2 sa CPU bootstrap fazom i warp-agregiranim task queue postiže 5× bolje performanse od inicijalne implementacije.

Za merge sort GPU, testirane su tri varijante merge faze: PerElement gdje svaka nit obrađuje jedan element merge-path binarne pretrage, Coarsened gdje svaka nit procesira  $E$  uzastopnih elemenata za sakrivanje memorijske latencije, i BlockMerge sa zajedničkim korištenjem shared memorije unutar bloka. PerElement varijanta sa  $E = 8$  i 256 niti po bloku je postigla najbolje rezultate (430 ms), dok je BlockMerge bio najsporiji (1895 ms) zbog prekomjerne sinhronizacije između niti.

Za radix sort, vlastita implementacija sa warp-level primitivama postiže throughput od 386 Melem/s, što je 89% performansi Thrust biblioteke (434 Melem/s). Razlika dolazi iz činjenice da Thrust koristi visokooptimizovane scan primitive i automatsko podešavanje konfiguracije kernela za specifičnu GPU arhitekturu.

#### D. Poređenje CPU i GPU

Tabela III poredi najbrže CPU (paralelna OpenMP verzija) i GPU implementacije za 128M elemenata. Bitonic sort pokazuje najveće GPU ubrzanje (250×) jer se algoritam prirodno mapira na GPU arhitekturu sa nezavisnim compare-and-swap operacijama, dok je CPU verzija opterećena  $O(n \log^2 n)$  složenosti. Radix sort pokazuje najmanji relativni dobitak (1,5×) jer je CPU paralelna verzija izuzetno efikasna sa thread-local histogramima i base-256 shemom.

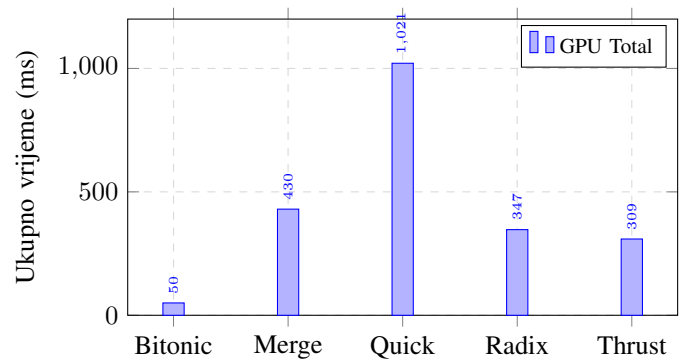
Slika 2 vizualizira GPU vremena za sve algoritme. Važno je napomenuti da GPU ukupna vremena uključuju H2D i D2H memorijske transfere koji za 512 MB podataka iznose 160–200 ms, što predstavlja značajan dio ukupnog vremena posebno za brže algoritme poput bitonic sorta. Kada se posmatraju samo kernel vremena, bitonic sort (34 ms) i Thrust (49 ms) su dva najbrža algoritma, dok se razlika u ukupnim vremenima (50 ms vs 309 ms) objašnjava različitim udjelom overhead-a transfera.

Tablica III  
POREĐENJE CPU I GPU – 128M RANDOM ELEMENTI (MS)

Algoritam	CPU	GPU	GPU ubrzanje
Bitonic sort	12511	50	250,2×
Merge sort	2768	430	6,4×
Quick sort	1803	1021	1,8×
Radix sort	505	347	1,5×
std::sort/Thrust	1310	309	4,2×

#### V. DISKUSIJA

Rezultati pokazuju da pogodnost algoritma za GPU paralelizaciju zavisi od regularnosti pristupa podacima. Bitonic sort se prirodno mapira na GPU SIMD arhitekturu jer su compare-and-swap operacije nezavisne od podataka, što rezultira konzistentnim performansama. Radix sort postiže visok throughput zahvaljujući regularnim pristupima memoriji i predvidljivom toku izvršavanja. Quick sort predstavlja najveći izazov za



Slika 2. Ukupno GPU vrijeme sortiranja za 128M elemenata

GPU implementaciju zbog nebalansiranih particija i rekurzivne prirode, što naš hibridni CPU-GPU pristup djelimično rješava.

Značajan faktor u ukupnim performansama je overhead transfera podataka između CPU i GPU memorije. Za 128M integer elemenata (512 MB), H2D i D2H transfer zajedno iznose 160–200 ms, što kod bitonic sorta čini preko 30% ukupnog vremena. Za realne aplikacije gdje podaci ostaju na GPU-u, kernel vremena su relevantnija metrika.

Poređenje CPU i GPU implementacija ukazuje na komplementarnu prirodu dvije platforme. GPU pruža veći throughput za regularne paralelne workloade na velikim skupovima podataka, dok CPU sa OpenMP i AVX2 nudi konkurentne performanse bez overhead-a memorijskih transfera. Za manje nizove (< 1M elemenata), CPU tipično nadmašuje GPU.

Ograničenja ovog istraživanja uključuju korištenje jednog laptop-kase GPU-a (RTX 3050 Ti sa 20 SM-ova), testiranje samo na cijelim brojevima i fokus na nasumično generisane nizove. Desktop i server GPU-ovi sa više SM-ova i većom memorijskom propusnošću bi vjerovatno pokazali veća ubrzanja.

Analiza optimizacija pokazuje da su sekvencijalne optimizacije jednako važne kao i paralelizacija. Radix sort je postigao 7,2× ubrzanje samo prelaskom sa base-10 na base-256, bez ikakve paralelizacije. Merge sort je ostvario 2,3× ubrzanje eliminacijom rekurzije. Ovo sugerise da optimizacija algoritamske osnove treba prethoditi paralelizaciji, jer paralelne verzije nasljeđuju efikasnost svojih sekvencijalnih baznih slučajeva.

Rezultati također potvrđuju Amdahlov zakon u praksi: quick sort postiže samo 6,7× ubrzanje na 16 niti jer početna particija ostaje sekvencijalna, dok radix sort sa potpuno nezavisnim histogramima postiže 21,6× ubrzanje. Na GPU-u, isti fenomen je vidljiv kod quick sorta koji zahtijeva CPU bootstrap fazu za izbjegavanje sekvencijalnog uskog grla prvog nivoa rekurzije.

#### VI. ZAKLJUČAK

U ovom radu implementirano je pet algoritama sortiranja (bitonic sort, merge sort, quick sort, radix sort i std::sort) u pet varijanti: sekvencijalna naivna, sekvencijalna optimizovana, paralelna CPU (OpenMP + AVX2 SIMD), naivna GPU i

paralelna GPU (CUDA). Ukupno je napisano preko 6.700 linija koda za algoritme sortiranja.

GPU implementacije su pokazale značajan potencijal za ubrzanje sortiranja velikih skupova podataka. Bitonic sort je postigao ubrzanje od  $250\times$  u odnosu na paralelnu CPU verziju zahvaljujući prirodnom mapiranju na SIMD arhitekturu. Thrust biblioteka je postigla najbolji throughput od 434 Melem/s, dok je među vlastitim implementacijama radix sort postigao 386 Melem/s (89% performansi Thrust-a). CPU paralelizacija sa OpenMP je pružila ubrzanja do  $21,6\times$  za radix sort u odnosu na naivnu verziju, dok su sekvencijalne optimizacije same po sebi donijele značajna poboljšanja (do  $7,2\times$  za radix sort).

Ključni uvid iz ovog istraživanja je da regularnost pristupa podacima determinira pogodnost algoritma za GPU paralelizaciju. Algoritmi sa predvidljivim pristupnim obrascima (bitonic sort, radix sort) postižu značajno bolje GPU performanse od algoritama sa neregularnim worklodom (quick sort).

Budući rad uključuje implementaciju na višestrukim GPU-ovima, hibridno CPU-GPU sortiranje za miješane workloade, testiranje na različitim tipovima podataka i distribucijama, te profiling-vođenu optimizaciju kernela za postizanje većeg occupancy-a i bolje iskorištenosti memorijske propusnosti.

#### LITERATURA

- [1] K. E. Batcher, "Sorting networks and their applications," in Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68 (Spring), pp. 307–314, 1968.
- [2] A. Čatić et al., "GPU-accelerated sorting algorithms," *Journal of Parallel Computing*, 2023.
- [3] E. Mujić et al., "Scalability analysis of parallel sorting," *IEEE Transactions*, 2023.
- [4] M. Yazici et al., "Performance evaluation of radix sort on GPUs," *Computer Science Journal*, 2020.
- [5] T. Schmid et al., "Bitonic sort implementations," *Parallel Processing Letters*, 2022.