🧠 Pitanja i odgovori za obranu – Paralelno programiranje (OpenCL)

🔹 Zadatak 1 – Brojanje prim brojeva (OpenCL)

🔸 **Kako ste paralelizirali problem brojanja prim brojeva?**

Svaka dretva obrađuje dio niza (više elemenata) unutar petlje, temeljem svojeg globalnog ID-a. Dijeljenje se vrši s work_per_thread = (n + total_threads - 1) / total_threads.

🔸 **Zašto ste koristili atomic_add? Kada je potreban?**

Kada više dretvi pristupa i mijenja zajedničku varijablu (npr. brojač prim brojeva), potrebno je koristiti atomičku operaciju kako bi se izbjegle pogreške u pristupu („race condition“).

🔸 **Koja je razlika u performansama između običnog zbrajanja i atomičkog?**

Obična operacija += je brža, ali nije sigurna. atomic_add je sporiji, ali točan. Mjerenjem se pokazuje koliko utječe na izvedbu.

🔸 **Koji su najpovoljniji G i L parametri za najbrže izvođenje?**

To se određuje eksperimentalno – kombinacije kao (256, 16) ili (1024, 256) često daju dobre rezultate. Rezultati su ispisani u CSV datoteci.

🔸 **Kako ste provjerili točnost OpenCL rezultata?**

Rezultat OpenCL izvođenja uspoređuje se s referentnim brojem prim brojeva izračunatim sekvencijalno na CPU-u.

🔸 **Gdje se nalazi funkcija za provjeru je li broj prim (is_prime)?**

U kernelu i na hostu (isPrime(n)), koristi se poznati postupak testiranja prim brojeva (do √n).

🔸 **Gdje se koristi atomic_add, a gdje obični +=?**

U OpenCL kernelu count_primes, ovisno o use_atomic flagu, koristi se jedna od te dvije metode.

🔸 **Kako se računaju start, end i work_per_thread?**

Svaka dretva obrađuje N / G elemenata: start = gid * work_per_thread, end = min(start + work_per_thread, n).

🔸 **Kako se inicijaliziraju OpenCL uređaji i bira jedan za izvođenje?**

Pomoću clGetPlatformIDs i clGetDeviceIDs. Korisnik ručno bira uređaj nakon što se prikažu svi dostupni.

🔹 Zadatak 2 – Računanje broja Pi (OpenCL)

🔸 **Koji algoritam koristite za računanje Pi?**

Leibnizova formula: π ≈ 4 × Σ((-1)^n / (2n + 1)) za n = 0, 1, 2, ...

🔸 **Kako ste raspodijelili posao na dretve?**

Svaka dretva računa sumu M elemenata, počevši od start_index + gid * M.

🔸 **Što predstavljaju parametri N, M, L?**

N: ukupan broj elemenata reda

M: broj elemenata po dretvi

L: veličina radne grupe

🔸 **Zašto je potrebno koristiti višestruke pozive dretvi u petlji (batch processing)?**

Jer broj elemenata N može biti veći od broja radnih jedinica koje uređaj može podržati u jednom pozivu.

🔸 **Koliko ste ubrzanje postigli u odnosu na sekvencijalnu verziju?**

Ubrzanje (speedup) se računa kao time_seq / time_opencl, a rezultat se ispisuje.

🔸 **Kolika je bila greška aproksimacije Pi u odnosu na math.pi?**

Računa se relativna greška: abs(pi_opencl - math.pi) / math.pi * 100.

🔸 **Gdje se nalazi OpenCL kernel (kernel_source)?**

Na vrhu 2zad_OpenCL.py, definirana kao multi-line string varijabla kernel_source.

🔸 **Gdje se računaju global_size i local_size?**

U funkciji compute_pi_opencl, ovisno o N, M, L.

🔸 **Kako se upravlja bufferima i prebacuju rezultati natrag iz OpenCL-a?**

Korištenjem cl.Buffer, cl.enqueue_copy i clEnqueueNDRangeKernel.

🔸 **Gdje se računa ukupna suma i konačna vrijednost Pi?**

U compute_pi_opencl, nakon svakog batch-a, elementi iz rezultata se zbrajaju.

🔸 **Gdje se koristi argparse za prihvaćanje argumenata iz komandne linije?**

U main() funkciji, pomoću argparse.ArgumentParser.

Zadatak 3 – Simulacija dinamike fluida (CFD) – Jacobi metoda

Što predstavlja Jacobi metoda i gdje se koristi u ovom kontekstu?

Koristi se za iterativno rješavanje Laplaceove jednadžbe u 2D mreži – modeliranje potencijalnog toka fluida.

Zašto ste baš funkciju jacobistep paralelizirali?

Jer se svaka točka psinew[i,j] računa neovisno na temelju susjednih vrijednosti u psi.

Kako ste paralelizirali jacobistep pomoću OpenCL-a?

Jedan kernel poziv, gdje svaka dretva obrađuje jednu točku [i,j] unutar granica (1..m, 1..n).

Koji je oblik domene (global/local size) korišten u kernelu?

global = {m, n}; local = {16, 16} (ili manje ako su dimenzije manje od 16).

Koji su ulazni i izlazni OpenCL Buffer objekti u vašoj implementaciji?

psi_buf: ulazni raspored stanja

psitmp_buf: novi raspored nakon iteracije

Kako ste provjerili ispravnost rezultata OpenCL inačice?

Korištenjem fallback CPU verzije i mjerenjem greške (deltasq()), koja mora biti manja od zadane tolerancije.

Zašto su indeksi u kernelu i + 1, j + 1? Jer se računaju samo unutarnje točke matrice [1..m][1..n], rubovi su definirani granicama.

Kako ste računali pogrešku (error) i čemu ona služi? Kvadratna suma razlika između psi i psitmp, normalizirana s početnom energijom (bnorm). Koristi se za uvjet konvergencije.

Gdje je OpenCL kernel definiran?

U jacobi.cpp kao const char* kernel_source.

Gdje se alociraju i postavljaju OpenCL buffere?

U init_opencl() funkciji.

Gdje se izvršava OpenCL kernel?

U funkciji jacobistep(), kroz clEnqueueNDRangeKernel.

Kako se izračunava greška?

Funkcija deltasq(), razlika kvadrata elemenata između dvije iteracije.

Kako se obavlja fallback na CPU?

Ako OpenCL nije dostupan (!opencl_ready), koristi se petlja u jacobistep() koja ručno računa psinew.

Gdje se mjeri vrijeme izvođenja?

U cfd.cpp, pomoću gettime(), tstart, tstop.

Zašto koristiš (m+2) kao širinu mreže u alokaciji i indeksiranju?

Mreža uključuje rubne ćelije (ghost cells) – stvarna širina je (m+2)x(n+2).

Zašto se kernel kompajlira u runtime-u, a ne unaprijed?

OpenCL koristi clCreateProgramWithSource jer omogućava fleksibilnost pri pokretanju na različitim uređajima.

Zašto koristiš clSetKernelArg(..., &m) i &n, a ne direktno m, n?

Funkcija zahtijeva pokazivač na podatak koji se kopira u memoriju uređaja.