AKCELERACIJA SIMULACIJE MOLEKULARNE DINAMIKE

Lazar Premović

2023/3020

ALGORITAM

```
printf(" Molecular Dynamics Simulation example program\n");
printf(" ------------------------------------------\n");
printf(" number of particles is ............ %6d\n", npart);
printf(" side length of the box is ......... %13.6f\n", side);
printf(" cut off is ........................ %13.6f\n", rcoff);
printf(" reduced temperature is ............ %13.6f\n", tref);
printf(" basic timestep is ................. %13.6f\n", h);
printf(" temperature scale interval ........ %6d\n", irep);
printf(" stop scaling at move .............. %6d\n", istop);
printf(" print interval .................... %6d\n", iprint);
printf(" total no. of steps ................ %6d\n", movemx);
printf("\n");
printf("    i       ke         pe            e         temp      pres      vel      rp \n");
printf("  -----  ----------  ----------  ----------  --------  --------  --------  ----\n");

clock_t begin = clock();

for (int move = 1; move <= movemx; move++) {
    double epot = 0;
    double vir = 0;
    double ekin = 0;
    comb(npart, x_x, x_y, x_z, vh_x, vh_y, vh_z, f_x, f_y, f_z,
        side, rcoff, hsq2, hsq, &epot, &vir, &ekin);

    double count = 0;
    double vel = velavg(npart, vh_x, vh_y, vh_z, vaver, h, &count);
    if (move < istop && move % irep == 0) {
        double sc = sqrt(tref / (tscale * ekin));
        dscal(npart, sc, vh_x, 1);
        dscal(npart, sc, vh_y, 1);
        dscal(npart, sc, vh_z, 1);
        ekin = tref / tscale;
    }

    if (move % iprint == 0) {
        prnout(move, ekin, epot, tscale, vir, vel, count, npart, den);
    }
}

clock_t end = clock();
double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

printf("Time: %lf\n", time_spent);
```

- Kod predstavlja simulaciju molekularne dinamike atoma argona u ograničenom prostoru sa periodičnim graničnim uslovima.
- Atomi se inicijalno nalaze raspoređeni u pravilnu mrežu, a zatim se tokom simulacije izračunavaju interakcije između njih.
- U svakom koraku simulacije u glavnoj petlji se dešava sledeće:
- Čestice se pomeraju zavisno od njihovih brzina i brzine se parcijalno ažuriraju.
- Izračunavaju se sile koje deluju na čestice, prosečna kinetička energija (virial) i potencijalna energija.
- Dobijene sile se skaliraju, brzine se ažuriraju i izračunava se kinetička energija.
- Izračunava se prosečna brzina i primenjuju granični uslovi.
- Ispisuju se rezultati.

STRATEGIJA AKCELERACIJE

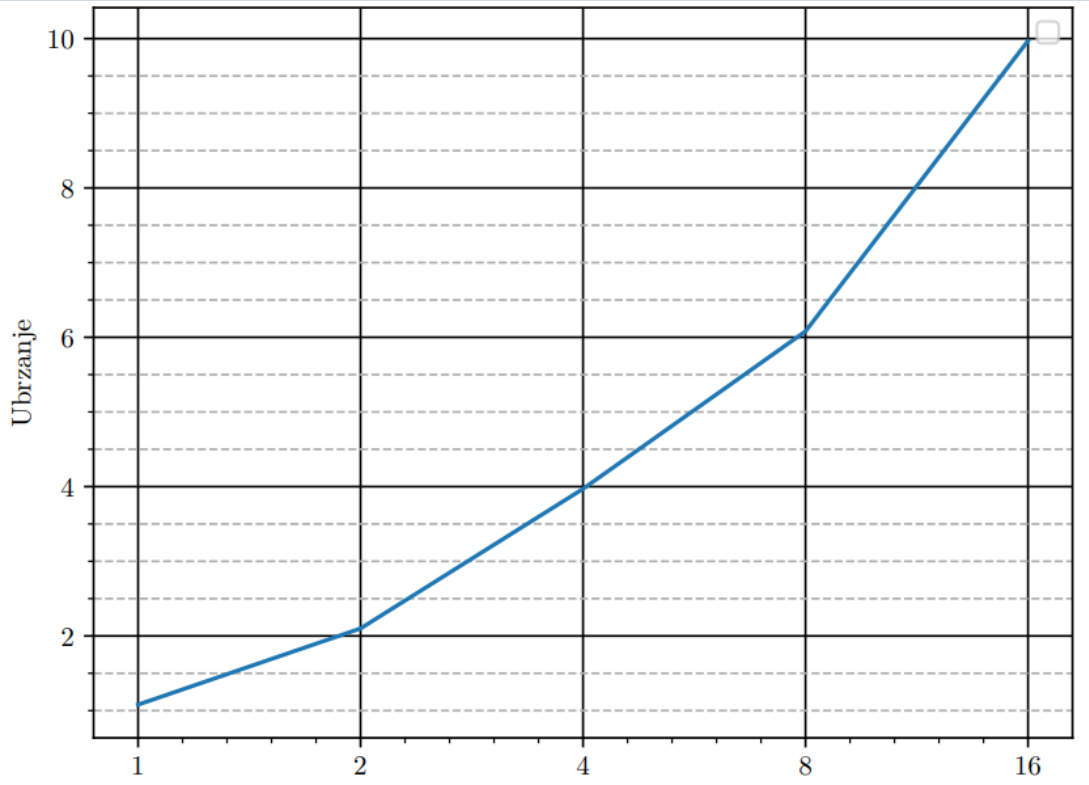
```
void comb(int npart, double x[], double vh[], double f[],
    double side, double rcoff, double hsq2, double hsq,
    double* epot, double *vir, double* ekin)
{
    for (int i = 0; i < npart * 3; i++) {
        x[i] += vh[i] + f[i];
        if (x[i] < 0) { x[i] += side; }
        if (x[i] > side) { x[i] -= side; }
        vh[i] += f[i];
        f[i] = 0;
    }

    for (int i = 0; i < npart * 3; i += 3) {
        for (int j = i + 3; j < npart * 3; j += 3) {
            double xx = x[i] - x[j];
            double yy = x[i + 1] - x[j + 1];
            double zz = x[i + 2] - x[j + 2];
            if (xx < -0.5 * side) { xx += side; }
            if (xx > 0.5 * side) { xx -= side; }
            if (yy < -0.5 * side) { yy += side; }
            if (yy > 0.5 * side) { yy -= side; }
            if (zz < -0.5 * side) { zz += side; }
            if (zz > 0.5 * side) { zz -= side; }
            double rd = xx * xx + yy * yy + zz * zz;

            if (rd <= rcoff * rcoff) {
                double rrd = 1 / rd;
                double rrd2 = rrd * rrd;
                double rrd3 = rrd2 * rrd;
                double rrd4 = rrd2 * rrd2;
                double rrd6 = rrd2 * rrd4;
                double rrd7 = rrd6 * rrd;
                double r148 = rrd7 - 0.5 * rrd4;
                double forcex = xx * r148;
                double forcey = yy * r148;
                double forcez = zz * r148;
                *epot += (rrd6 - rrd3);
                *vir -= rd * r148;
                f[i] += forcex;
                f[j] -= forcex;
                f[i + 1] += forcey;
                f[j + 1] -= forcey;
                f[i + 2] += forcez;
                f[j + 2] -= forcez;
            }
        }
    }

    double sum = 0;
    for (int i = 0; i < 3 * npart; i++) {
        f[i] *= hsq2;
        vh[i] += f[i];
        sum += vh[i] * vh[i];
    }
    *ekin = sum / hsq;
}
```

- Odlučeno je da se akceleracija fokusira na prva tri koraka jer su oni odgovorni za većinu vremena izvršavanja.
- Ta tri koraka su potom spojena u jednu funkciju pod imenom comb.
- Prvi i treći deo se svode na trivijalnu obradu nad podacima jedne čestice i nemaju zavisnosti po podacima između iteracija petlje, osim redukcije po ukupnoj kinetičkoj energiji.
- Pri računanju sila se međutim prolazi kroz svaki par čestica (bez simetričnih i sopstvenih parova), izračunava se njihova udaljenost i ukoliko je unutar opsega od interesa, ažuriraju se sile koje deluju na obe čestice.
- Kako srednji deo i dalje dominira vremenom izvršavanja, složenost algoritma je približna $O(\frac{N^2}{2})$ za jednu iteraciju, pogotovo jer je broj čestica velik $(N = 13\,500)$.

CPU SINGLE/MULTI

- Referentna CPU implementacija u C programskom jeziku izvršava jednu iteraciju simulacije za **0.22s** i ta vrednost će biti korišćena kao referentna za izračunavanje ostalih ubrzanja.

- Još nekoliko izmerenih CPU vremena su:
  - C, bez optimizacije: **0.5s**
  - Python: **56.7s**
  - Python, Google Colab: **198s**
  - Python + TensorFlow: **7.4s**
  - Python + TensorFlow, Google Colab: **50.7s**
  - C, MI Sanu: **0.92s**



- Za evaluaciju multi-core performansi, iteracije spoljne petlje u drugoj sekciji su dinamički podeljene nitima na procesiranje korišćenjem OpenMP API-a.

- Algoritam je izvršavan na do **16** niti i izmereno je ubrzanje do **9.97** puta.

GPU MANY

- Za evaluaciju many-core performansi na grafičkoj kartici korišćena je slična strategija davanja jedne iteracije spoljne petlje svakom jezgru.
- Ovaj pristup je implementiran direktno korišćenjem CUDA API-a.
- Inicijalno je postignuto ubrzanje od samo **3.78** puta, što je znatno lošije i od multi-core implementacije.
- Međutim zamenom tipa nekoliko ključnih promenjivih sa **double** na **float** dobijeno je ubrzanje od čak **128** puta, bez značajnog gubitka preciznosti!
- Iz tog razloga će svi ostali pristupi akceleraciji koristiti **float** tip podataka.
- Takođe je zanimljivo napomenuti da ista zamena tipa negativno utiče na performanse CPU-a s obzirom da su oni već neko vreme optimizovani za podatke širine 64 bita.

Flex Data-flow

- Flex Data-flow kernel ima 9 ulaznih i izlaznih toka podataka, tokovi podataka pozicije sile i brzine su podeljeni na po tri toka, jedan za svaku osu.
- Ovim je smanjen potreban broj ciklusa, po cenu većeg iskorišćenja resursa i memorijskog protoka.
- Konstante i rezultujući virial, potencijalna, kinetička energija se prosleđuju kao skalarni podaci.
- Stanje kernela je reprezentovano kroz tri ulančana brojača:
  - Prvi vodi računa o agregacijama korišćenjem AutoLoop offseta.
  - Drugi prolazi kroz svaku česticu
    i koristi se kao brojač petlje u prvom i trećem koraku
    a u drugom koraku predstavlja brojač unutrašnje petlje
  - Treći brojač prolazi kroz svaku česticu plus još dva prolaza
    u kojima se izvršavaju prvi i treći korak.
- Kako se prvi korak svodi na trivijalnu manipulaciju tokovima, tokovi se u jednom ciklusu učitavaju modifikuju i upisuju u memoriju.
- Treći korak slično vrši obradu nad tokovima (koji se ovog puta čitaju iz memorije) ali dodatno vrši agregaciju kinetičke energije koristeći AutoLoop offsete.

```
DFEVar xx = io.input("xx", streamType, (outer === 0) & counterLoop.getWrap());
DFEVar xy = io.input("xy", streamType, (outer === 0) & counterLoop.getWrap());
DFEVar xz = io.input("xz", streamType, (outer === 0) & counterLoop.getWrap());
DFEVar fx = io.input("fx", streamType, (outer === 0) & counterLoop.getWrap());
DFEVar fy = io.input("fy", streamType, (outer === 0) & counterLoop.getWrap());
DFEVar fz = io.input("fz", streamType, (outer === 0) & counterLoop.getWrap());
DFEVar vx = io.input("vx", streamType, (outer === 0) & counterLoop.getWrap());
DFEVar vy = io.input("vy", streamType, (outer === 0) & counterLoop.getWrap());
DFEVar vz = io.input("vz", streamType, (outer === 0) & counterLoop.getWrap());

// In loop
xx = xx + vx + fx;
xy = xy + vy + fy;
xz = xz + vz + fz;
xx = xx < 0 ? xx + side : xx > side ? xx - side : xx;
xy = xy < 0 ? xy + side : xy > side ? xy - side : xy;
xz = xz < 0 ? xz + side : xz > side ? xz - side : xz;
vx = vx + fx;
vy = vy + fy;
vz = vz + fz;
fx = constant.var(streamType,0);
fy = constant.var(streamType,0);
fz = constant.var(streamType,0);

xxRam.write(addr, xx, (outer === 0) & counterLoop.getWrap());
xyRam.write(addr, xy, (outer === 0) & counterLoop.getWrap());
xzRam.write(addr, xz, (outer === 0) & counterLoop.getWrap());
vxRam.write(addr, vx, (outer === 0) & counterLoop.getWrap());
vyRam.write(addr, vy, (outer === 0) & counterLoop.getWrap());
vzRam.write(addr, vz, (outer === 0) & counterLoop.getWrap());
```

Flex Data-flow

- Flex Data-flow kernel ima 9 ulaznih i izlaznih toka podataka, tokovi podataka pozicije sile i brzine su podeljeni na po tri toka, jedan za svaku osu.
- Ovim je smanjen potreban broj ciklusa, po cenu većeg iskorišćenja resursa i memorijskog protoka.
- Konstante i rezultujući virial, potencijalna, kinetička energija se prosleđuju kao skalarni podaci.
- Stanje kernela je reprezentovano kroz tri ulančana brojača:
  - Prvi vodi računa o agregacijama korišćenjem AutoLoop offseta.
  - Drugi prolazi kroz svaku česticu
    i koristi se kao brojač petlje u prvom i trećem koraku
    a u drugom koraku predstavlja brojač unutrašnje petlje
  - Treći brojač prolazi kroz svaku česticu plus još dva prolaza
    u kojima se izvršavaju prvi i treći korak.
- Kako se prvi korak svodi na trivijalnu manipulaciju tokovima, tokovi se u jednom ciklusu učitavaju modifikuju i upisuju u memoriju.
- Treći korak slično vrši obradu nad tokovima (koji se ovog puta čitaju iz memorije) ali dodatno vrši agregaciju kinetičke energije koristeći AutoLoop offsete.

```
DFEVar oxx = xxRam.read(addr);
DFEVar oxy = xyRam.read(addr);
DFEVar oxz = xzRam.read(addr);
DFEVar ofx = fxRam.read(addr);
DFEVar ofy = fyRam.read(addr);
DFEVar ofz = fzRam.read(addr);
DFEVar ovx = vxRam.read(addr);
DFEVar ovy = vyRam.read(addr);
DFEVar ovz = vzRam.read(addr);

// Out loop
ofx = ofx * hsq2;
ofy = ofy * hsq2;
ofz = ofz * hsq2;
ovx = ovx + ofx;
ovy = ovy + ofy;
ovz = ovz + ofz;

DFEVar ekin = streamType.newInstance(this);
DFEVar lekin = (outer === n + 1) & (inner === 0) ?
    constant.var(streamType,0) : stream.offset(ekin, -loopLength);
ekin <== lekin + ovx * ovx + ovy * ovy + ovz * ovz;

io.output("oxx", oxx, streamType, (outer === n + 1) & counterLoop.getWrap());
io.output("oxy", oxy, streamType, (outer === n + 1) & counterLoop.getWrap());
io.output("oxz", oxz, streamType, (outer === n + 1) & counterLoop.getWrap());
io.output("ofx", ofx, streamType, (outer === n + 1) & counterLoop.getWrap());
io.output("ofy", ofy, streamType, (outer === n + 1) & counterLoop.getWrap());
io.output("ofz", ofz, streamType, (outer === n + 1) & counterLoop.getWrap());
io.output("ovx", ovx, streamType, (outer === n + 1) & counterLoop.getWrap());
io.output("ovy", ovy, streamType, (outer === n + 1) & counterLoop.getWrap());
io.output("ovz", ovz, streamType, (outer === n + 1) & counterLoop.getWrap());

io.scalarOutput("epot", epot, streamType, counterOuter.getWrap());
io.scalarOutput("vir", vir, streamType, counterOuter.getWrap());
io.scalarOutput("ekin", ekin / hsq, streamType, counterOuter.getWrap());
```

Flex Data-flow

- Drugi korak je ubedljivo najsloženiji i zahteva najviše ciklusa (N^2) i resursa.
- Radi smanjivanja kompleksnosti kernel iterira kroz sve parove čestica (uključujući simetrične i sopstvene parove) naknadno ignorišući nepoželjne parove za koje važi (i <= j), što povećava vreme izvršavanja samo dva puta a značajno smanjuje kompleksnost rešenja.
- Pored klasične obrade tokova (koji kao i kod trećeg dela dolaze iz memorije) i dve agregacije realizovane na isti način koji je korišćen i do sada, drugi deo vrši i neophodnu manipulaciju memorijom.
- Kako algoritam ima read-modify-write semantiku nad dva podatka u svakoj iteraciji potrebno je obratiti posebnu pažnju kako ne bi prekršili ograničenja memorijskih modula (jedno čitanje – jedan upis i čitanje lokacije na koju se vrši upis je nedefinisano)

```
DFEVar oxx = xxRam.read(addr);
DFEVar oxy = xyRam.read(addr);
DFEVar oxz = xzRam.read(addr);
DFEVar ofx = fxRam.read(addr);
DFEVar ofy = fyRam.read(addr);
DFEVar ofz = fzRam.read(addr);
DFEVar ovx = vxRam.read(addr);
DFEVar ovy = vyRam.read(addr);
DFEVar ovz = vzRam.read(addr);

DFEVar ouxx = xxRam.read(oaddr);
DFEVar ouxy = xyRam.read(oaddr);
DFEVar ouxz = xzRam.read(oaddr);
DFEVar oufx = stream.offset(fxRam.read(oaddr),-18);
DFEVar oufy = stream.offset(fyRam.read(oaddr),-18);
DFEVar oufz = stream.offset(fzRam.read(oaddr),-18);

// Mid loop

DFEVar dx = ouxx - oxx;
DFEVar dy = ouxy - oxy;
DFEVar dz = ouxz - oxz;
dx = dx < -0.5 * side ? dx + side : dx > 0.5 * side ? dx - side : dx;
dy = dy < -0.5 * side ? dy + side : dy > 0.5 * side ? dy - side : dy;
dz = dz < -0.5 * side ? dz + side : dz > 0.5 * side ? dz - side : dz;
DFEVar rd = dx * dx + dy * dy + dz * dz;

DFEVar rrd = 1 / rd;
rrd = addr > oaddr ? rrd : constant.var(streamType,0);
rrd = rd <= rcoff * rcoff ? rrd : constant.var(streamType,0);

DFEVar rrd2 = rrd * rrd;
DFEVar rrd3 = rrd2 * rrd;
DFEVar rrd4 = rrd2 * rrd2;
DFEVar rrd6 = rrd2 * rrd4;
DFEVar rrd7 = rrd6 * rrd;
DFEVar r148 = rrd7 - 0.5 * rrd4;
DFEVar forcex = dx * r148;
DFEVar forcey = dy * r148;
DFEVar forcez = dz * r148;

DFEVar epot = streamType.newInstance(this);
DFEVar lepot = (outer === 1) & (inner === 0) ? constant.var(streamType,0) : stream.offset(epot, -loopLength);
epot <== lepot + rrd6 - rrd3;

DFEVar vir = streamType.newInstance(this);
DFEVar lvir = (outer === 1) & (inner === 0) ? constant.var(streamType,0) : stream.offset(vir, -loopLength);
vir <== lvir - rd * r148;

DFEVar prefxj = stream.offset(ofx, -18);
DFEVar prefyj = stream.offset(ofy, -18);
DFEVar prefzj = stream.offset(ofz, -18);

DFEVar wfxi = oufx + forcex;
DFEVar wfyi = oufy + forcey;
DFEVar wfzi = oufz + forcez;
DFEVar wfxj = prefxj - forcex;
DFEVar wfyj = prefyj - forcey;
DFEVar wfzj = prefzj - forcez;

DFEVar fxa = ((outer === 0) & counterLoop.getWrap()) ? addr : ((outer !== 0 & outer !== n + 1) & counterLoop.getWrap()) ? addr : oaddr;
DFEVar fxw = ((outer === 0) & counterLoop.getWrap()) | ((outer !== 0 & outer !== n + 1) & (counterLoop.getWrap() | loop === loopLengthVal - 2));
DFEVar fxv = ((outer === 0) & counterLoop.getWrap()) ? fx : ((outer !== 0 & outer !== n + 1) & counterLoop.getWrap()) ? wfxj : wfxi;
DFEVar fyv = ((outer === 0) & counterLoop.getWrap()) ? fy : ((outer !== 0 & outer !== n + 1) & counterLoop.getWrap()) ? wfyj : wfyi;
DFEVar fzv = ((outer === 0) & counterLoop.getWrap()) ? fz : ((outer !== 0 & outer !== n + 1) & counterLoop.getWrap()) ? wfzj : wfzi;

fxRam.write(fxa, fxv, fxw);
fyRam.write(fxa, fyv, fxw);
fzRam.write(fxa, fzv, fxw);
```

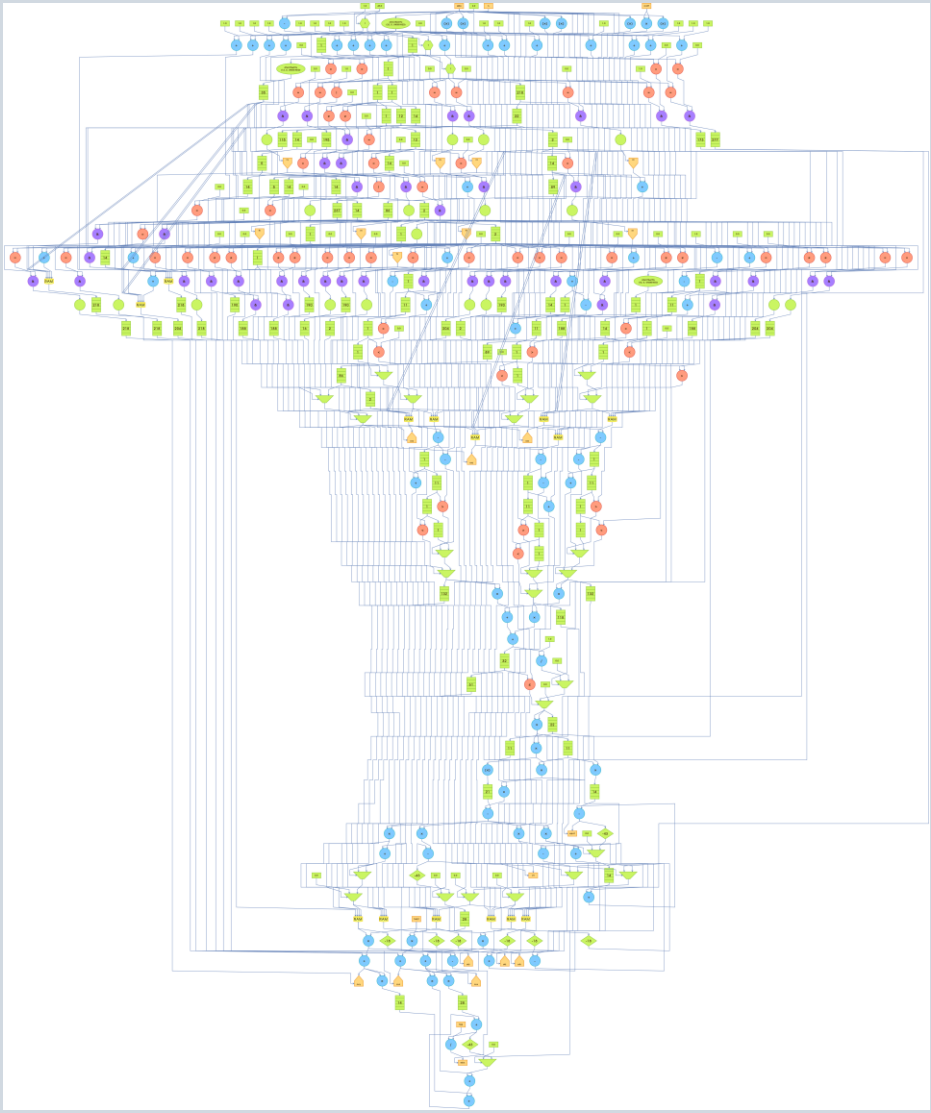
Flex Data-flow

- To je realizovano razdvajanjem dva upisa u jednoj iteraciji u dva ciklusa (ovo je omogućeno usporenjem koja zahteva agregacija korišćenjem AutoLoop offseta).
- Problem sa čitanjem adrese u koju se upisuje je rešen čitanjem podatka unapred i potom njegovim kašnjenjem za 18 ciklusa (što je opet zamaskirano AutoLoop offsetom)
- AutoLoop offset neophodan za agregacije viriala i potencijalne energije je čak 43 ciklusa zbok kompleksne zavisnosti između prethodne i nove vrednosti.

```
DFEVar oxx = xxRam.read(addr);
DFEVar oxy = xyRam.read(addr);
DFEVar oxz = xzRam.read(addr);
DFEVar ofx = fxRam.read(addr);
DFEVar ofy = fyRam.read(addr);
DFEVar ofz = fzRam.read(addr);
DFEVar ovx = vxRam.read(addr);
DFEVar ovy = vyRam.read(addr);
DFEVar ovz = vzRam.read(addr);

DFEVar ouxx = xxRam.read(oaddr);
DFEVar ouxy = xyRam.read(oaddr);
DFEVar ouxz = xzRam.read(oaddr);
DFEVar oufx = stream.offset(fxRam.read(oaddr),-18);
DFEVar oufy = stream.offset(fyRam.read(oaddr),-18);
DFEVar oufz = stream.offset(fzRam.read(oaddr),-18);

// Mid loop

DFEVar dx = ouxx - oxx;
DFEVar dy = ouxy - oxy;
DFEVar dz = ouxz - oxz;
dx = dx < -0.5 * side ? dx + side : dx > 0.5 * side ? dx - side : dx;
dy = dy < -0.5 * side ? dy + side : dy > 0.5 * side ? dy - side : dy;
dz = dz < -0.5 * side ? dz + side : dz > 0.5 * side ? dz - side : dz;
DFEVar rd = dx * dx + dy * dy + dz * dz;

DFEVar rrd = 1 / rd;
rrd = addr > oaddr ? rrd : constant.var(streamType,0);
rrd = rd <= rcoff * rcoff ? rrd : constant.var(streamType,0);

DFEVar rrd2 = rrd * rrd;
DFEVar rrd3 = rrd2 * rrd;
DFEVar rrd4 = rrd2 * rrd2;
DFEVar rrd6 = rrd2 * rrd4;
DFEVar rrd7 = rrd6 * rrd;
DFEVar r148 = rrd7 - 0.5 * rrd4;
DFEVar forcex = dx * r148;
DFEVar forcey = dy * r148;
DFEVar forcez = dz * r148;

DFEVar epot = streamType.newInstance(this);
DFEVar lepot = (outer === 1) & (inner === 0) ? constant.var(streamType,0) : stream.offset(epot, -loopLength);
epot <== lepot + rrd6 - rrd3;

DFEVar vir = streamType.newInstance(this);
DFEVar lvir = (outer === 1) & (inner === 0) ? constant.var(streamType,0) : stream.offset(vir, -loopLength);
vir <== lvir - rd * r148;

DFEVar prefxj = stream.offset(ofx, -18);
DFEVar prefyj = stream.offset(ofy, -18);
DFEVar prefzj = stream.offset(ofz, -18);

DFEVar wfxi = oufx + forcex;
DFEVar wfyi = oufy + forcey;
DFEVar wfzi = oufz + forcez;
DFEVar wfxj = prefxj - forcex;
DFEVar wfyj = prefyj - forcey;
DFEVar wfzj = prefzj - forcez;

DFEVar fxa = ((outer === 0) & counterLoop.getWrap()) ? addr : ((outer !== 0 & outer !== n + 1) & counterLoop.getWrap()) ? addr : oaddr;
DFEVar fxw = ((outer === 0) & counterLoop.getWrap()) | ((outer !== 0 & outer !== n + 1) & (counterLoop.getWrap() | loop === loopLengthVal - 2));
DFEVar fxv = ((outer === 0) & counterLoop.getWrap()) ? fx : ((outer !== 0 & outer !== n + 1) & counterLoop.getWrap()) ? wfxj : wfxi;
DFEVar fyv = ((outer === 0) & counterLoop.getWrap()) ? fy : ((outer !== 0 & outer !== n + 1) & counterLoop.getWrap()) ? wfyj : wfyi;
DFEVar fzv = ((outer === 0) & counterLoop.getWrap()) ? fz : ((outer !== 0 & outer !== n + 1) & counterLoop.getWrap()) ? wfzj : wfzi;

fxRam.write(fxa, fxv, fxw);
fyRam.write(fxa, fyv, fxw);
fzRam.write(fxa, fzv, fxw);
```

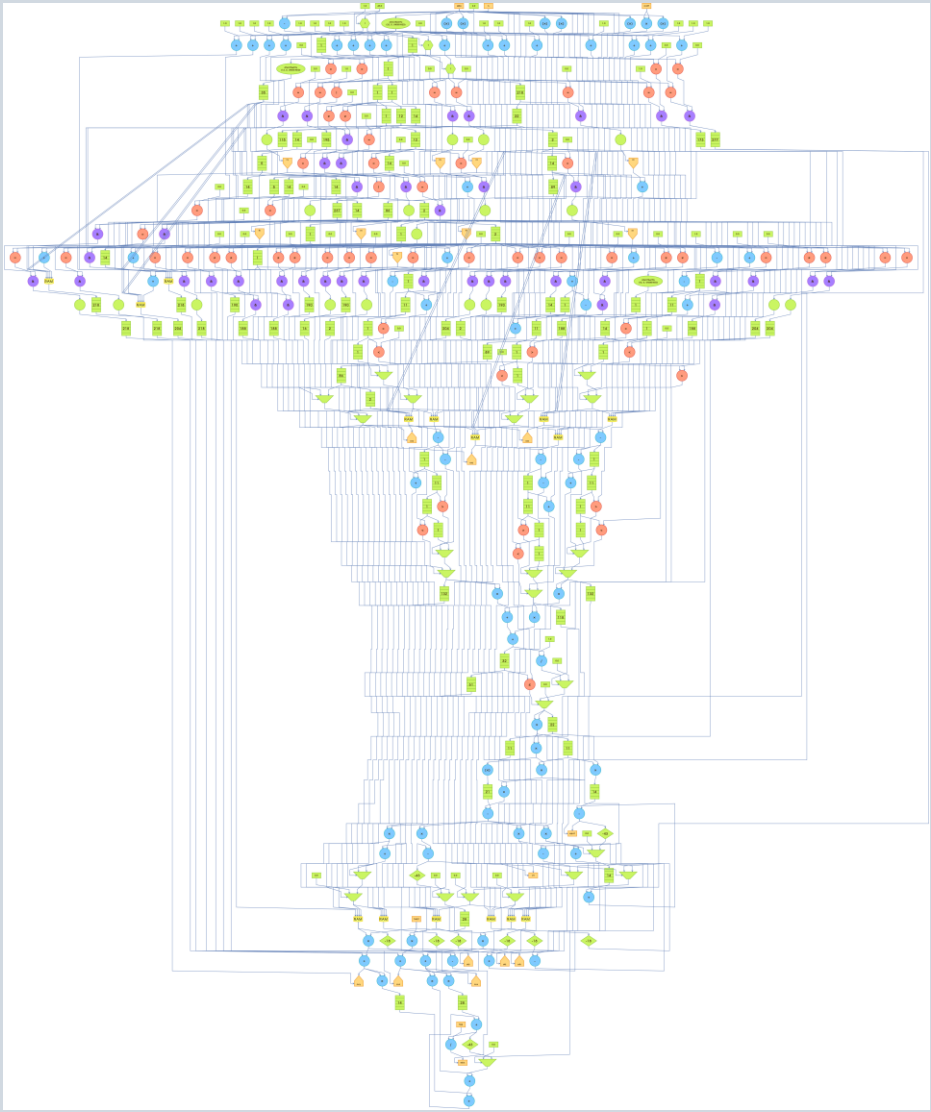
Performanse FlexDF-a

- Data implementacija se uprkos kompleksnosti izvršava **N * (N + 2) * loopLength** ciklusa što za prethodno dato ($N = 13\,500$) i frekvenciju signala takta od 200MHz daje vreme izvršavanja od **39s** što je značajno sporije od osnovne CPU implementacije.ž
- Međutim ovo rešenje ne iskorišćava pun potencijal Flex Data-flow paradigme iz čistog razloga praktičnosti razvoja i izvodljivosti simulacije.
- Iz tog razloga će biti data projekcija performansi direktne sinteze ovog algoritma na „top of the line“ Intel Agilex 7 FPGA čipu.
- Kako bi takva analiza bila moguća, potrebno je znati koliko resursa algoritam zahteva u svakom trenutku.
- Resursi koji će biti razmatrani su operacione jedinice za množenje i sabiranje
- Pretpostavka je da ostalih resursi neće predstavljati ograničavajući faktor



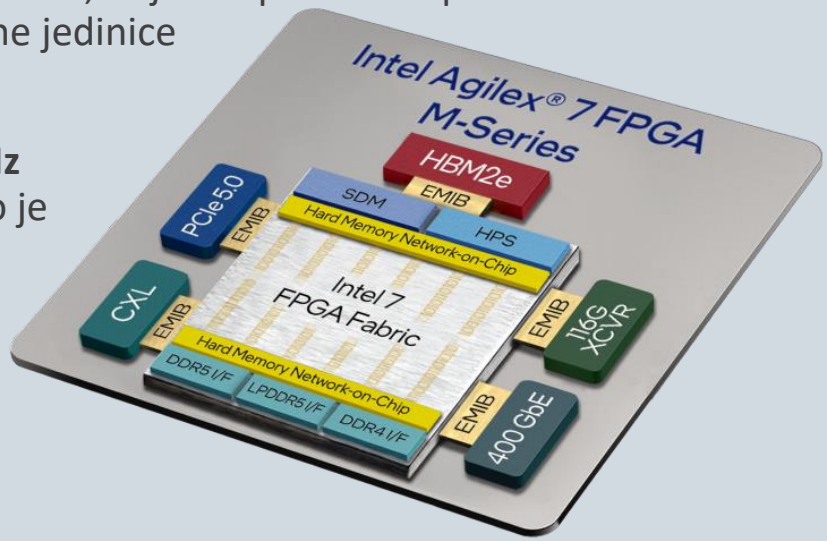
Performanse FlexDF-a

- Tehnike koje će se koristiti su:
  - Multipleksiranje resursa:
    Kako svaki od tri dela algoritma mora u potpunosti da se izvrši kako bi sledeći mogao da počne, u svakom trenutku su resursi potrebni samo jednom delu te resursi koje je neki drugi deo koristio mogu biti dodeljeni delu koji se trenutno izvršava.
  - Optimizacija algoritma:
    Ovo uključuje iteriranje samo kroz validne parove korišćenjem kompleksnije logike brojača.
  - Uklanjanje AutoLoop offseta:
    Umesto usporavanja kernela za broj ciklusa potrebnih za propagaciju rezultata, možemo koristiti kompleksnije strukture za agregaciju koje nemaju ovaj negativan efekat
  - Paralelizacija:
    Ukoliko nam resursi dozvoljavanju, umesto jednog podatka po ciklusu možemo obraditi više podataka prostim instanciranjem više kopija kernela.



Performanse FlexDF-a

- Intel Agilex 7 M 039 poseduje 12300 operacionih jedinica.
- Jedna iteracija prvog dela zahteva **15** operacionih jedinica, što omogućava **820** instanci koje obrađuju celokupan tok u **17** ciklusa.
- Jedna iteracija trećeg dela zahteva **13** operacionih jedinica i poseduje jednu redukciju, omogućavajući **906** instanci koje obrađuju celokupan tok u **15** ciklusa.
- Jedna iteracija drugog dela zahteva **35** operacionih jedinica i poseduje dve redukcije, omogućavajući **336** instanci koje obrađuju celokupan tok u **271186** ciklusa.
- Iako je moguće smanjiti usporenje koje proizilazi iz nekoliko redukcija, drugi deo efektivno zahteva redukciju nad svakim članom niza, što nije izvodljivo zbog dostupnih resursa, te je neophodno usporiti obradu u drugom delu onoliko puta koliko je kašnjenje jedne operacione jedinice (5 ciklusa) čineći da ukupan broj ciklusa bude oko **1355K**.
- Maksimalna radna frekvencija operacionih jedinica je **750MHz** rezultujući u očekivanom vremenu izvršavanja od **0.0018s** što je ubrzanje od **122** puta.
- Iako je ubrzanje znatno bolje nego kod inicijalne implementacije, GPU i dalje ima najveće ubrzanje.

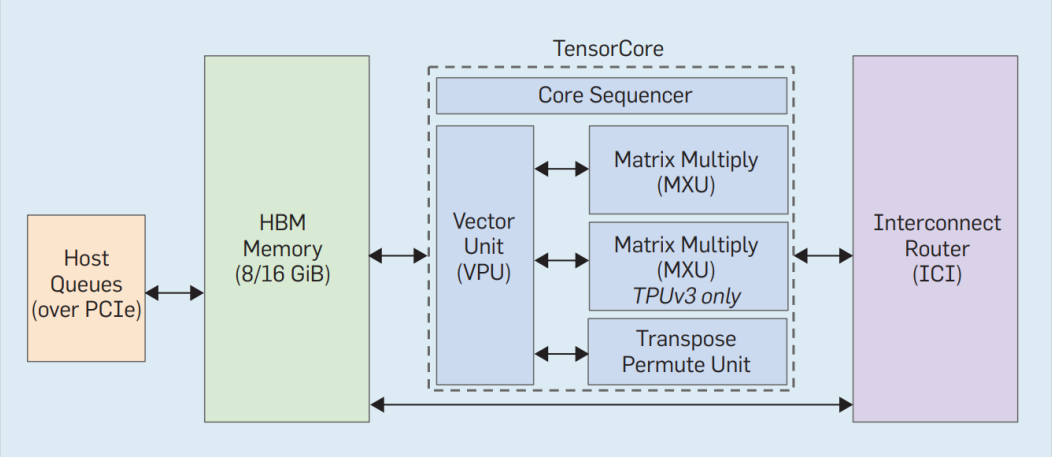


Speedup, Power, Volume, Precision

	Speedup	Power	Area	Precision
CPU	1x	20.5J	8mm2	Double
CPU – Multi	9.97x	32.9J	128mm2	Double
GPU	**128x**	0.55J	628mm2	Single
FlexDF – MPC	0.005x	110J	~435mm2	Single
FlexDF - Agilex	**122x**	**0.27J**	~500mm2	Single

Fixed Data-flow

- Google TPU v2 je predstavnik fixed data-flow paradigme
- Njegova računska moć se sastoji od:
  - Jedinice za množenje matrica koja je realizovana kao sistolni niz i može da množi matrice do 128x128
  - Jedinica za vektorska izračunavanja koja podržava SIMD paradigmu i može izvršiti do 8 operacija nad vektorima širine 128 elemenata
  - Iako ne vrši računanje direktno, efikasna jedinica za transponovanje i permutovanje je ključna za efikasan rad ovog akceleratora
- Google TPU se koristi pomoću TensorFlow API-a koji igra ulogu sličnu MaxJ kompajleru
  - Korisnik opisuje svoj algoritam u Python programskom jeziku, koristeći TensorFlow funkcije koje vrše operacije nad tenzorima.
  - TensorFlow potom pretvara tu funkciju u graf izvršavanja i priprema je za optimizovano izvršavanje na CPU, GPU ili TPU.



Akceleracija FixedDF-a

- Prva dva dela je trivijalno adaptirati jer njihove petlje upravo iteriraju kroz niz i ručno vrše vektorske operacije nad elementima nizova.
- Kako bi se izbegle petlje, drugi deo je adaptiran tako da se operacije vrše nad matricom gde koordinate elementa predstavljaju par čestica na koje se taj element odnosi.
- I ovde se obrada vrši i nad neželjenim parovima, ali to ovde dovodi do poboljšanja performansi jer TPU benefituje od regularnosti operacija.
- Treba napomenuti da iako se u drugom delu vrše operacije nad matricama, one se izvršavaju korišćenjem vektorske jedinice jer se nigde zapravo ne vrši množenje matrica.

```python
@tf.function
def comb(npart, x, vh, f, side, rcoff, hsq2, hsq):
    epot = tf.constant(0.0, dtype=tf.float32)
    vir = tf.constant(0.0, dtype=tf.float32)
    ekin = tf.constant(0.0, dtype=tf.float32)

    x.assign_add(vh + f)
    x.assign(tf.math.floormod(x + side, side))
    vh.assign_add(f)
    f.assign(tf.zeros_like(f))

    idx = tf.range(npart)
    i_idx = tf.reshape(tf.tile(idx, [npart]), [npart, npart])
    j_idx = tf.transpose(i_idx)
    mask = tf.less(i_idx, j_idx)
    i_idx = tf.boolean_mask(i_idx, mask)
    j_idx = tf.boolean_mask(j_idx, mask)
    i_idx3 = i_idx * 3
    j_idx3 = j_idx * 3

    xi = tf.gather(x, i_idx3)
    xj = tf.gather(x, j_idx3)
    yi = tf.gather(x, i_idx3 + 1)
    yj = tf.gather(x, j_idx3 + 1)
    zi = tf.gather(x, i_idx3 + 2)
    zj = tf.gather(x, j_idx3 + 2)
    xx = xi - xj
    yy = yi - yj
    zz = zi - zj
    xx = tf.math.floormod(xx + 1.5 * side, side) - 0.5 * side
    yy = tf.math.floormod(yy + 1.5 * side, side) - 0.5 * side
    zz = tf.math.floormod(zz + 1.5 * side, side) - 0.5 * side
    rd = xx * xx + yy * yy + zz * zz

    mask = tf.math.less_equal(rd, rcoff * rcoff)
    rrd = tf.where(mask, 1.0 / rd, tf.zeros_like(rd))
    rrd2 = rrd * rrd
    rrd3 = rrd2 * rrd
    rrd4 = rrd2 * rrd2
    rrd6 = rrd2 * rrd4
    rrd7 = rrd6 * rrd
    r148 = rrd7 - 0.5 * rrd4

    epot += tf.reduce_sum(rrd6 - rrd3)
    vir -= tf.reduce_sum(rd * r148)

    forcex = xx * r148
    forcey = yy * r148
    forcez = zz * r148

    f = tf.tensor_scatter_nd_add(f, tf.expand_dims(i_idx3, axis=-1), forcex)
    f = tf.tensor_scatter_nd_add(f, tf.expand_dims(j_idx3, axis=-1), -forcex)
    f = tf.tensor_scatter_nd_add(f, tf.expand_dims(i_idx3 + 1, axis=-1), forcey)
    f = tf.tensor_scatter_nd_add(f, tf.expand_dims(j_idx3 + 1, axis=-1), -forcey)
    f = tf.tensor_scatter_nd_add(f, tf.expand_dims(i_idx3 + 2, axis=-1), forcez)
    f = tf.tensor_scatter_nd_add(f, tf.expand_dims(j_idx3 + 2, axis=-1), -forcez)

    f = (f * hsq2)
    vh.assign_add(f)
    sum_vh2 = tf.reduce_sum(vh * vh)
    ekin = sum_vh2 / hsq

    return epot, vir, ekin, x, vh, f
```

Performanse FixedDF-a

- Dati kod je izvršen nad Google TPU v2 čipom korišćenjem Google Colab cloud okruženja.
- Kod se izvršavao **25.6s** što je ubrzanje od **2** puta u odnosu na izvršavanje istog koda na Google Colab okruženju bez TPU-a
- Međutim izvršavanje istog koda na Google Colab okruženju sa GPU-om je rezultovalo u vremenu izvršavanja od samo **1.28s**, što je ubzanje od **39.6** puta.
- Nažalost oba rešenja su i dalje daleko od performansi modernog CPU-a sa kodom napisanim u C programskom jeziku ili GPU algoritmom koji direktno koristi CUDA API.
- U slučaju TPU-a, ovo se može pripisati njegovoj strogoj optimizaciji za samo jednu klasu problema, a to je mašinsko učenje koje ekstenzivno koristi operaciju množenja matrica, dok to nije slučaj sa prikazanim algoritmom.

```
@tf.function
def comb(npart, x, vh, f, side, rcoff, hsq2, hsq):
    epot = tf.constant(0.0, dtype=tf.float32)
    vir = tf.constant(0.0, dtype=tf.float32)
    ekin = tf.constant(0.0, dtype=tf.float32)

    x.assign_add(vh + f)
    x.assign(tf.math.floormod(x + side, side))
    vh.assign_add(f)
    f.assign(tf.zeros_like(f))

    idx = tf.range(npart)
    i_idx = tf.reshape(tf.tile(idx, [npart]), [npart, npart])
    j_idx = tf.transpose(i_idx)
    mask = tf.less(i_idx, j_idx)
    i_idx = tf.boolean_mask(i_idx, mask)
    j_idx = tf.boolean_mask(j_idx, mask)
    i_idx3 = i_idx * 3
    j_idx3 = j_idx * 3

    xi = tf.gather(x, i_idx3)
    xj = tf.gather(x, j_idx3)
    yi = tf.gather(x, i_idx3 + 1)
    yj = tf.gather(x, j_idx3 + 1)
    zi = tf.gather(x, i_idx3 + 2)
    zj = tf.gather(x, j_idx3 + 2)
    xx = xi - xj
    yy = yi - yj
    zz = zi - zj
    xx = tf.math.floormod(xx + 1.5 * side, side) - 0.5 * side
    yy = tf.math.floormod(yy + 1.5 * side, side) - 0.5 * side
    zz = tf.math.floormod(zz + 1.5 * side, side) - 0.5 * side
    rd = xx * xx + yy * yy + zz * zz

    mask = tf.math.less_equal(rd, rcoff * rcoff)
    rrd = tf.where(mask, 1.0 / rd, tf.zeros_like(rd))
    rrd2 = rrd * rrd
    rrd3 = rrd2 * rrd
    rrd4 = rrd2 * rrd2
    rrd6 = rrd2 * rrd4
    rrd7 = rrd6 * rrd
    r148 = rrd7 - 0.5 * rrd4

    epot += tf.reduce_sum(rrd6 - rrd3)
    vir -= tf.reduce_sum(rd * r148)

    forcex = xx * r148
    forcey = yy * r148
    forcez = zz * r148

    f = tf.tensor_scatter_nd_add(f, tf.expand_dims(i_idx3, axis=-1), forcex)
    f = tf.tensor_scatter_nd_add(f, tf.expand_dims(j_idx3, axis=-1), -forcex)
    f = tf.tensor_scatter_nd_add(f, tf.expand_dims(i_idx3 + 1, axis=-1), forcey)
    f = tf.tensor_scatter_nd_add(f, tf.expand_dims(j_idx3 + 1, axis=-1), -forcey)
    f = tf.tensor_scatter_nd_add(f, tf.expand_dims(i_idx3 + 2, axis=-1), forcez)
    f = tf.tensor_scatter_nd_add(f, tf.expand_dims(j_idx3 + 2, axis=-1), -forcez)

    f = (f * hsq2)
    vh.assign_add(f)
    sum_vh2 = tf.reduce_sum(vh * vh)
    ekin = sum_vh2 / hsq

    return epot, vir, ekin, x, vh, f
```

Speedup, Power, Volume, Precision

	Speedup	Power	Area	Precision
CPU	1x	20.5J	8mm2	Double
CPU – Multi	9.97x	32.9J	128mm2	Double
GPU	**128x**	0.55J	628mm2	Single
FlexDF – MPC	0.005x	110J	~435mm2	Single
FlexDF - Agilex	**122x**	**0.27J**	~500mm2	Single
TPU	0.0086x	~7162J	611mm2	Single
GPU - TensorFLow	0.172x	~89.5J	545mm2	Single

Kvantni računari

- Kvantni računari koriste principe kvantne mehanike i superpozicije kako bi efikasno izvršavali određene algoritme.
- Operacije se vrše nad qbitima koji su reprezentovani u kompleksnom domenu verovatnoćama da se nađu u stanju 0 ili 1 prilikom merenja.
- I ako je moguće reprezentovati klasične operacije bulove algebre u kvantnim računarima, i time izvršavati proizvoljni kod, njihova efikasnost nije dovoljna da bi bila isplativa.
- Stoga je primena kvantnih računara primarno u algoritmima koji mogu eksplicitno da iskoriste benefite ove nekonvencionalne paradigme.
- Takvi algoritmi su retki i danas postoji samo nekoliko algoritama koji imaju velike benefite od izvršavanja na kvantnim računarima.
- Nažalost simulacija molekularne dinamike nije jedan od njih.

Optički računari

- Razvoj optičkih računara danas teče u dva pravca:
  - Optički računari bazirani na bulovoj algebri koji koriste konvencionalne paradigme
  - Analogni računari koji koriste specifična fizička svojstva svetlosti
- Prvi pristup ne zahteva nikakvo eksplicitno prilagođavanje i dati algoritam automatski benefituje od bilo kog unapređenja na ovom polju.
- Analogni računari su koncept koji je relativno skoro opet postao od interesa zahvaljujući sve zahtevnijim modelima veštačke inteligencije.
- Neki od primera upotrebe fizičkih svojstava su optičke memorije, optički množači i korišćenje prizmi za konverziju u fekvencijski domen.
- Kako prikazan algoritam ne koristi ni jednu od operacija koje su posebno efikasne u optičkim računarima, jedini pravac za akceleraciju optičkim računarima je da oni preteknu konvencionalne računare u nekom od kriterijuma (Speed, Power, Area, Precision) u operacijama bulove algebre.

Biološki računari

- Biološki računari se oslanjaju na korišćenje DNK molekula za vršenje računanja.
- Još uvek ne postoje operacije koje su svojstvene biološkim računarima, te je za sada fokus na izvršavanju operacija bulove algebre bolje od konvencionalnih računara.
- Kriterijumi po kojima Biološki računari mogu najlakše da prevaziđu konvencionalne računare su: Power i Area.

Molekularni računari

- Molekularni računari se trude da modeluju potrebno izračunavanje kroz hemijske reakcije.
- Neki od problema koji se najlakše mogu modelovati hemijskim računarima su upravo fizičke i hemijske simulacije.
- I ako je ovaj tip računara tek u začeću i postoji samo nekoliko njihovih primera koji su ništa više nego prototipi, njihova paradigma obećava najbolje performanse za ovaj konkretan algoritam.