Факултет техничких наука

Универзитет у Новом Саду

Рачунарски системи високих перформанси

Оптимизација алгоритма јата

Аутор:
Лука Иванишевић

Индекс:
E2 117/2025

1. фебруар 2026.

Сажетак

Алгоритам јата (Boids), који је развио Крејг Рејнолдс 1986. године, представља један од најпознатијих модела вештачког живота и користи се за симулацију колективног понашања јединки, попут јата птица или риба. Модел се заснива на једноставним локалним правилима – поравнању, кохезији и раздвајању – чијом применом настаје сложено глобално понашање система. Иако је концепт алгоритма релативно једноставан, његова рачунска сложеност значајно расте са повећањем броја јединки, што представља изазов за ефикасну реализацију у реалном времену.

Циљ овог рада је анализа и оптимизација алгоритма јата кроз различите приступе паралелизацији, са посебним освртом на употребу графичких процесора. Проблем који се разматра односи се на високу временску сложеност наивне имплементације алгоритма, где свака јединка интерагује са свим осталима, што резултује сложеношћу O(N2)

Решавању проблема приступљено је кроз више фаза. Најпре је реализована наивна секвенцијална имплементација алгоритма на централном процесору (CPU), као и њена директна паралелна верзија на графичком процесору (GPU). Након тога, алгоритам је паралелизован на CPU-у коришћењем OpenMP библиотеке. У завршној фази извршена је оптимизација GPU имплементације применом просторне деобе (uniform grid), чиме је значајно смањен број интеракција које је потребно израчунати за сваку јединку.

Добијени резултати показују очекивани поредак перформанси: најбоље резултате остварује оптимизована GPU имплементација, затим наивна GPU верзија, потом паралелна CPU имплементација са OpenMP-ом, док је најслабије перформансе показала секвенцијална CPU реализација.

Садржај

Списак изворних кодова

Списак табела

1 Увод

Симулација колективног понашања представља значајну област истраживања у рачунарским наукама, са применама у компјутерској графици, роботици, вештачкој интелигенцији и моделовању сложених система. Један од најпознатијих модела у овој области је алгоритам јата (Boids), који је предложио Крејг Рејнолдс средином осамдесетих година прошлог века. Овај алгоритам показује како се применом једноставних локалних правила на појединачне јединке може добити сложено и реалистично глобално понашање целог система. [1]

Основни принципи алгоритма заснивају се на три правила понашања која су уочена у природи: поравнање, кохезија и раздвајање [2]. Свако од ових понашања описује како појединачна јединка реагује на своје суседе у области видљивости. Иако су ова правила локална и једноставна, њихова примена на великом броју јединки доводи до захтевних рачунарских симулација.

Наивна имплементација алгоритма подразумева да свака јединка интерагује са свим осталима у систему, иако не утичу све на њено понашање. Оваква имплементација резултује квадратном временском сложеношћу у односу на број јединки. Оваква сложеност брзо постаје непрактична за веће системе и представља главну мотивацију за примену техника паралелног програмирања и алгоритамске оптимизације.

Циљ овог семинарског рада јесте да се анализирају различите имплементације и оптимизације алгоритма јата. У овом раду су описане имплементације алгоритма намењене за процесорско извршавање и њихова оптимизација коришћењем OpenMP библиотеке, као и имплементација намењена за извршавање на графичким картицама, специфично базирана на CUDA програмском моделу [5].

2 Алгоритам јата

2.1 Основни концепт алгоритма

Срж Boids модела јесте да симулира понашање птица или риба у јату. У природи је уочено да на сваку јединку важе три основна типа понашања: поравнање, кохезија и раздвајање. Свако од ових правила јединки саопштава информацију о смеру и интензитету којим треба да се креће. Сабирањем резултујућих вектора ова три понашања и њиховим множењем са одговарајућим тежинама правила добија се резултујући вектор који дефинише кретање јединке у наредном тренутку.

На понашање сваке јединке искључиво делују само локалне једнике које су јој у видокругу. Јединке ван те области не утичу на одлуку о њеном понашању. Свака јединка је независна од друге што значи да се може паралелно рачунати кретање сваке појединачне јединке.

Симулација се извршава у дискретним временским корацима, при чему се у сваком кораку најпре израчунавају утицаји суседних јединки, затим ажурира вектор брзине, а након тога и позиција јединке у простору. Да би се обезбедило реалистично кретање, брзина и убрзање јединки се најчешће ограничавају на одређене минималне и максималне вредности.

2.2 Правила понашања јединки

2.2.1 Поравнање

Правило поравнања има за циљ да јединка усклади свој смер кретања са просечним смером кретања суседних јединки. На овај начин се постиже уједначено кретање јата и смањују нагле промене смера. Вектор поравнања се добија као нормализована средња вредност вектора брзина свих суседа у области видљивости.

$$\vec{v}_{align} = \frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} \vec{v}_j \tag{1}$$

2.2.2 Кохезија

Правило кохезије усмерава јединку ка центру масе суседних јединки, чиме се обезбеђује да јато остане компактно и да се јединке не удаљавају превише једна од друге. Вектор кохезије представља разлику између позиције центра масе суседа и позиције посматране јединке.

$$\vec{v}_{coh} = \left(\frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} \vec{p}_j \right) - \vec{p}_i \tag{2}$$

2.2.3 Раздвајање

Правило раздвајања спречава сударање јединки тако што производи одбојни вектор који усмерава јединку да се удаљи од превише блиских суседа. Утицај сваког суседа је обрнуто пропорционалан растојању између јединки, чиме се наглашава избегавање блиских контаката.

$$\vec{v}_{sep} = \sum_{j \in \mathcal{N}_i} \frac{\vec{p}_i - \vec{p}_j}{\|\vec{p}_i - \vec{p}_j\|} \tag{3}$$

2.2.4 Коначни вектор кретања

Коначан вектор кретања јединке добија се као линеарна комбинација вектора поравнања, кохезије и раздвајања, при чему сваки од ових утицаја има придружен те-

жински коефицијент који омогућава подешавање њиховог значаја у укупном понашању јата.

$$\vec{v}_i^{new} = w_{align} \cdot \vec{v}_{align} + w_{coh} \cdot \vec{v}_{coh} + w_{sep} \cdot \vec{v}_{sep} \tag{4}$$

3 Имплементација алгоритма

3.1 Секвенцијална наивна имплементација

Наивна секвенцијална имплементација алгоритма јата подразумева да свака јединка интерагује са свим осталима у систему. За сваку јединку се пролази кроз све остале јединке и, уколико се налазе у области видљивости, израчунавају се вектори поравнања, кохезије и раздвајања. Таква реализација резултује временском сложеношћу $O(N^2)$.

Код се може приказати у сажетом облику:

```
for (int i = 0; i < N; i++) {
    Vector align = {0,0}, coh = {0,0}, sep = {0,0};
    int count = 0;

    for (int j = 0; j < N; j++) {
        if (i == j) continue;
        float dx = boids[j].position.x - boids[i].position.x;
        float dy = boids[j].position.y - boids[i].position.y;
        float dist2 = dx*dx + dy*dy;
        if (dist2 > perception_radius*perception_radius) continue;

        align.x += boids[j].velocity.x;
        align.y += boids[j].velocity.y;
        coh.x += boids[j].position.x;
        coh.y += boids[j].position.y;
        sep.x -= dx / dist2;
        sep.y -= dy / dist2;
        count++;
    }

    if (count) {
        align.x /= count; align.y /= count;
        coh.x = coh.x/count - boids[i].position.x;
        coh.y = coh.y/count - boids[i].position.y;
    }

    boids[i].velocity.x += w_align*align.x +
     w_coh*coh.x + w_sep*sep.x;
    boids[i].velocity.y += w_align*align.y +
     w_coh*coh.y + w_sep*sep.y;

    float speed = sqrtf(boids[i].velocity.x*boids[i].velocity.x +
                        boids[i].velocity.y*boids[i].velocity.y);
    if (speed > MAX_SPEED) {
        boids[i].velocity.x = (boids[i].velocity.x / speed)
         * MAX_SPEED;
        boids[i].velocity.y = (boids[i].velocity.y / speed)
         * MAX_SPEED;
    }

    boids[i].position.x += boids[i].velocity.x;
    boids[i].position.y += boids[i].velocity.y;
}
```


Изворни код 1: Пример C наивног алгоритма

3.2 Паралелна CPU имплементација (OpenMP)

4 Паралелизација помоћу OpenMP

Да би се побољшале перформансе на вишејезгреним процесорима, секвенцијална наивна имплементација је паралелизована коришћењем OpenMP библиотеке. Основна петља кроз све јединке је обележена `#pragma omp parallel for` директивом, што омогућава да више јединки буде обрађено истовремено на различитим језгрима процесора.

```
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < N; i++) {
    update_boid(&boids[i], boids, N,
                window_width, window_height,
                perception_radius, fov_deg,
                w_align, w_cohesion, w_separation);
}
```

Изворни код 2: Пример Omp паралелизације наивног алгоритма

У овој имплементацији, функција `update_boid` обавља исту логику као и унутрашња петља из секвенцијалне верзије, односно израчунава векторе поравнања, кохезије и раздвајања за сваку јединку, а затим ажурира брзину и позицију.

Кључна опција `schedule(dynamic)` омогућава динамичку расподелу посла између језгара, што је корисно јер различите јединке имају различит број суседа у области видљивости. На тај начин се избегава ситуација да нека језгра заврше раније, док друга остају оптерећена, чиме се постиже боља равнотежа оптерећења и већа ефикасност паралелизације.

4.1 Наивна GPU имплементација

Наивна GPU имплементација користи CUDA програмски модел да се обрада јединки паралелизује на графичкој картици. Модел је замишљен тако да један CUDA thread представља једну јединку. Thread-ови су организовани у једнодимензионалне блокове од по 256 нити, а свака нит приступа својој јединки и израчунава векторе поравнања, кохезије и раздвајања на основу осталих јединки у систему, на исти начин као у претходним секвенцијалним и OpenMP примерима.

Важно је напоменути да није могуће организовати блокове у дводимензионалну структуру кајi би моделовали сегментирање простора, јер број јединки који би

припао неком сегменту није унапред познат. Уколико бисмо хтели да блокови означавају конкретне регије простора, било би немогуће динамички распоредити јединке тако да свака нит унутар блока обрађује тачно једну јединку из тог региона. Због тога је једнодимензионални блок најприкладнији, јер свака нит обрађује појединачну јединку без потребе за претходним познавањем распореда јединки унутар простора.

Да би ова паралелизација функционисала, низ јединки мора бити алоциран и на CPU и на GPU. Пре извршавања CUDA kernel-а, подаци се копирају са CPU меморије у GPU меморију. Након што све ните заврше свој рад, потребно је синхронизовати извршење помоћу `cudaDeviceSynchronize()` како би се осигурало да су све јединке ажуриране. Пошто је потребно да CPU изврши визуелизацију, резултати се враћају из GPU меморије на CPU.

```
__global__ void update_boids_kernel(Boid* boids, int N,
          float perception_radius, float fov_deg,
          float w_align, float w_cohesion, float w_separation)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    Boid b = boids[i];
    if (i >= N) return;

    update_boid_position_device(&b, boids, N,
                                perception_radius, fov_deg,
                                w_align, w_cohesion, w_separation);
    boids[i] = b;
}

void cuda_update_boids( Boid* boids, int N,
            float perception_radius, float fov_deg,
            float w_align, float w_cohesion, float w_separation)
{
    int threads = 256;
    int blocks = (N + threads - 1) / threads;
    update_boids_kernel<<<blocks, threads>>>(
        d_boids, N,
        perception_radius, fov_deg,
        w_align, w_cohesion, w_separation
    );
    cudaGetLastError();
    cudaDeviceSynchronize();
    cudaMemcpy(boids, d_boids,
       N * sizeof(Boid), cudaMemcpyDeviceToHost);
}
```

Изворни код 3: Пример GPU паралелизације наивног алгоритма

4.2 Оптимизација алгоритма на GPU-у

Оптимизована GPU имплементација користи исту организацију нити и блокова као и наивна верзија, односно једнодимензионалне блокове од по 256 нити. Ово је неопходно због непознатог броја јединки унапред и немогућности динамичке расподеле 2D блокова по просторним регијама.

Основна идеја оптимизације је смањење временске сложености кроз просторно

индексирање јединки. Поступак се састоји из следећих корака [3]:

1. **Додела сегмената:** За сваку јединку се одређује којем сегменту (grid cell) припада на основу њене позиције. Величина сегмента је једнака радијусу видљивости јединке. Ово се обавља функцијом `calculateGridPosition`.

2. **Сортирање јединки:** Јединке се сортирају растуће у односу на сегмент којем припадају. На овај начин се јединке истог сегмента налазе континуирано у низу, што омогућава ефикасно обрађивање суседних јединки. Сортирање се обавља помоћу `thrust::sort_by_key`.

3. **Одређивање почетка и краја сегмента:** За сваки сегмент се израчунава индекс прве и последње јединке које припадају том сегменту, што ради функција `calculateStartEndIndices`. Ово омогућава да се касније обрађују само јединке из сегмента и суседних сегмената.

4. **Ажурирање јединки:** На крају се позива kernel функција слична као у наивној верзији, која узима једну јединку и рачуна векторе само за јединке из њеног сегмента и суседних сегмената (у најгорем случају 8 суседних сегмената). Будући да се увек знају границе сваког сегмента и које јединке припадају ком сегменту, више није потребно пролазити кроз све јединке у систему, што значајно смањује сложеност алгоритма.

```
void cuda_update_optimized_boids(Boid* boids, int N,
                float perception_radius, float fov_deg,
              float w_align, float w_cohesion, float w_separation)
{
    int threads = 256;
    int blocks = (N + threads - 1) / threads;

    calculateGridPosition<<<blocks, threads>>>(d_boids, N,
      d_boidArrayIndices,d_boidGridIndices );

    thrust::sort_by_key(dev_thrust_particleGridIndices,
       dev_thrust_particleGridIndices + N,
        dev_thrust_particleArrayIndices);

    calculateStartEndIndices<<<blocks, threads>>>(
       N,
       d_boidGridIndices,
       d_gridCellStartIndices, d_gridCellEndIndices);

    update_boids_optimized<<<blocks, threads>>>(
        d_boids, N,
        perception_radius, fov_deg,
        w_align, w_cohesion, w_separation,
        d_boidArrayIndices,
        d_boidGridIndices,
        d_gridCellStartIndices,
        d_gridCellEndIndices
    );
    cudaGetLastError();

    cudaDeviceSynchronize();

    cudaMemcpy(boids, d_boids,
      N * sizeof(Boid), cudaMemcpyDeviceToHost);
}
```

Изворни код 4: Пример GPU паралелизације применом просторног сегментирања

Величина сегмента је намерно изабрана тако да одговара радијусу видљивости јединке. На тај начин, ако је јединка на самој ивици сегмента, она ће видети све

јединке до краја суседног сегмента, осигуравајући коректно израчунавање свих релевантних суседа.

5 Резултати и анализа

5.1 Опис тест окружења

Симулације су извршаване на хардверу са следећим карактеристикама: CPU је *11th Gen Intel(R) Core(TM) i7-11800H* са 16 хардверских нити, а GPU је *NVIDIA GeForce RTX 3060*. Графички приказ симулације је остварен коришћењем `SDL2` [4] библиотеке.

Параметри симулације су следећи: величина прозора 1900 x 1000 пиксела, радијус видљивости јединке (*perception radius*) 50, угао видног поља (*FOV*) 70 степени, тежине понашања: *alignment* 1.0, *cohesion* 0.5, *separation* 8.0. Број јединки се разликовао у различитим тестовима: 500, 1000, 5000, 10000, 30000, 100000, 200000 и 300000.

Кључни параметар који је праћен током извршавања је број кадрова у секунди (FPS), како би се оцењивала ефикасност различитих имплементација алгоритма.

5.2 Поређење перформанси

Табела 1 приказује број кадрова у секунди (FPS) за различите имплементације алгоритма јата при различитом броју јединки. Тестови су обављани у два режима: са рендеровањем графике и без рендеровања.

Табела 1: Број FPS за различите имплементације и број јединки

Имплементација	500	1000	5000	10000	30000	100000	200000	300000
Са рендеровањем								
Naive CPU	720	650	0	0	0	0	0	0
OpenMP	680	640	200	10	0	0	0	0
GPU Naive	700	670	480	220	60	1	0	0
GPU Optimized	710	670	450	270	80	50	24	0
Без рендеровања								
Naive CPU	730	650	30	0	0	0	0	0
OpenMP	750	700	500	10	0	0	0	0
GPU Naive	730	720	650	630	570	30	0	0
GPU Optimized	730	720	660	630	600	600	320	0

5.3 Дискусија резултата

Резултати добијени тестирањем различитих имплементација алгоритма јата су у складу са очекивањима. Наивна секвенцијална CPU имплементација брзо губи

перформансе са растом броја јединки због квадратне временске сложености, што се огледа у значајном паду FPS-а при већем броју јединки.

OpenMP имплементација показује значајно побољшање у односу на наивни CPU код. Увођење само једне `#pragma omp parallel for` директиве омогућава ефективну паралелизацију и равномерну расподелу рада између 16 доступних нити, што резултује стабилнијим FPS-ом и јасно демонстрира како минималне измене могу донети значајну оптимизацију.

GPU имплементације, како наивна тако и оптимизована, показују очигледну и очекивану предност у поређењу са CPU верзијама. У почетку, перформансе GPU Naive и GPU Optimized могу бити сличне, а у неким тренуцима оптимизована верзија је чак спорија због додатних корака и overhead-а, као што су израчунавање сегмената, сортирање и одређивање граница сегмената. Међутим, са растом броја јединки, GPU Optimized показује значајну предност и успева да одржи висок FPS без обзира на број јединки, јер се за сваку јединку рачунају само суседи из сегмента и суседних сегмената. Ово доводи до јасне разлике у односу на наивну GPU верзију, која и даље пролази кроз већи број јединки и тиме троши више ресурса.

Вредно је напоменути да је рендеровање симулације представља потенцијално уско грло при мерењу FPS-а. Због тога резултати у режиму са рендеровањем могу бити мање оптимистични у поређењу са резултатима без рендеровања.

6 Закључак

У овом раду анализирана је оптимизација алгоритма јата (Boids) за различите имплементације: секвенцијалну, паралелну на CPU-у коришћењем OpenMP-а, као и наивну и оптимизовану верзију на GPU-у коришћењем CUDA програмског модела. Резултати показују да наивна секвенцијална имплементација брзо губи перформансе са повећањем броја јединки, док OpenMP верзија представља једноставну али ефективну оптимизацију. GPU имплементације, нарочито оптимизована верзија, успешно одржавају висок FPS и ефикасно се носе са великим бројем јединки захваљујући смањеној сложености обрачуна суседа.

Могућа будућа побољшања укључују даљу оптимизацију приступа меморији на GPU-у кроз коришћење сортиране структуре података за јединке. Наиме, приступ глобалној меморији на GPU-у организован је у трансакције величине 32 бајта. То значи да се приликом једног приступа не учитава појединачни елемент, већ читав блок меморије. Уколико се подаци суседних јединки (boids) налазе на суседним меморијским локацијама, више нити у истом warp-у може да искористи једну исту меморијску трансакцију. Овакво коалесирано приступање меморији значајно смањује број приступа глобалној меморији и директно утиче на убрзање извршавања програма. Супротно томе, неконзистентан распоред података доводи до већег броја меморијских трансакција и лошијих перформанси.

Библиографија

[1] Boids. https://en.wikipedia.org/wiki/Boids. Приступано: 2026-01--20.

[2] Boids — craig reynolds. https://www.red3d.com/cwr/boids/. Приступано: 2026-01-17.

[3] Cuda boids: Grid looping optimization. https://charleszw.com/projects/cuda-boids#grid-looping-optimization. Приступано: 2026-01-22.

[4] Sdl3 api by category. https://wiki.libsdl.org/SDL3/APIByCategory. Приступано: 2026-01-25.

[5] Writing cuda kernels. https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html. Приступано: 2026-01-18.