**СРС**

**Выполнил студент**

**ИВТ-12М**

**Туринцев К.А.**

**Вариант - 4**

Задание 1. Запуск программы и метрики

#include <chrono>

#include <iostream>

#include <random>

#include <array>

#include <advisor-annotate.h>

#include <omp.h>

constexpr int STEP\_PRINT\_MOD{1};

constexpr float dt{0.1};

constexpr float T{1};

constexpr int N{static\_cast<int>(T / dt)};

constexpr int N\_BODYES{10000};

constexpr float MAX\_POS{0.5};

constexpr float MIN\_VEL{0.1};

constexpr float MAX\_VEL{1};

constexpr float MIN\_MASS{0.1};

constexpr float MAX\_MASS{100};

constexpr float G{6.67259e-11};         // Gravitational constant

constexpr float softeningSquared{1e-6}; // Softening parameter

struct Body

{

    float pos\_x, pos\_y, pos\_z;

    float vel\_x, vel\_y, vel\_z;

    float acc\_x, acc\_y, acc\_z;

    float mass;

};

float random\_float(float min, float max)

{

    std::random\_device rd;

    std::mt19937 gen(rd());

    std::uniform\_real\_distribution<> dis(min, max);

    return dis(gen);

}

class Simulation

{

private:

    std::array<Body, N\_BODYES> particles;

    const float step\_size;

public:

    Simulation(const float step\_size, const std::array<Body, N\_BODYES> &bodies) : particles{bodies}, step\_size(step\_size) {}

    void start(int n\_steps)

    {

        for (int step = 0; step < n\_steps; ++step)

        {

            auto start = std::chrono::high\_resolution\_clock::now();

            compute\_func();

            update\_data();

            float current\_time = step \* step\_size;

            float total\_energy = get\_energy();

            auto end = std::chrono::high\_resolution\_clock::now();

            if (step % STEP\_PRINT\_MOD == 0)

            {

                std::cout << "Step: " << step << ", Time: " << current\_time << ", Total Energy: " << total\_energy << std::endl;

                auto duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count();

                std::cout << "Step time is: " << duration << " microseconds" << std::endl;

            }

        }

    }

private:

    void compute\_func()

    {

        // Compute the acceleration for each body due to gravitational interactions

        // using Newton's law of gravitation

        ANNOTATE\_SITE\_BEGIN(for\_compute\_func);

        {

            for (int i = 0; i < particles.size(); ++i)

            {

                ANNOTATE\_ITERATION\_TASK(for\_compute\_func\_0);

                float acc\_x = 0, acc\_y = 0, acc\_z = 0;

                for (int j = 0; j < particles.size(); ++j)

                {

                    ANNOTATE\_ITERATION\_TASK(for\_compute\_func\_0\_0);

                    if (i != j)

                    {

                        float dx = particles[j].pos\_x - particles[i].pos\_x;

                        float dy = particles[j].pos\_y - particles[i].pos\_y;

                        float dz = particles[j].pos\_z - particles[i].pos\_z;

                        float rSquared = dx \* dx + dy \* dy + dz \* dz + softeningSquared;

                        float inv\_r = 1.0 / sqrt(rSquared);

                        float inv\_r3 = inv\_r \* inv\_r \* inv\_r;

                        acc\_x += G \* particles[j].mass \* dx \* inv\_r3;

                        acc\_y += G \* particles[j].mass \* dy \* inv\_r3;

                        acc\_z += G \* particles[j].mass \* dz \* inv\_r3;

                    }

                }

                particles[i].acc\_x = acc\_x;

                particles[i].acc\_y = acc\_y;

                particles[i].acc\_z = acc\_z;

            }

        }

    }

    void update\_data()

    {

        // Use the Runge-Kutta method to update positions and velocities

        ANNOTATE\_SITE\_BEGIN(for\_update\_data);

        {

            for (auto &particle : particles)

            {

                ANNOTATE\_ITERATION\_TASK(for\_update\_data\_0);

                // k1

                float k1vx = particle.acc\_x \* step\_size;

                float k1vy = particle.acc\_y \* step\_size;

                float k1vz = particle.acc\_z \* step\_size;

                float k1px = particle.vel\_x \* step\_size;

                float k1py = particle.vel\_y \* step\_size;

                float k1pz = particle.vel\_z \* step\_size;

                // k2

                float k2vx = (particle.acc\_x + k1px / 2) \* step\_size;

                float k2vy = (particle.acc\_y + k1py / 2) \* step\_size;

                float k2vz = (particle.acc\_z + k1pz / 2) \* step\_size;

                float k2px = (particle.vel\_x + k1vx / 2) \* step\_size;

                float k2py = (particle.vel\_y + k1vy / 2) \* step\_size;

                float k2pz = (particle.vel\_z + k1vz / 2) \* step\_size;

                // k3

                float k3vx = (particle.acc\_x + k2px / 2) \* step\_size;

                float k3vy = (particle.acc\_y + k2py / 2) \* step\_size;

                float k3vz = (particle.acc\_z + k2pz / 2) \* step\_size;

                float k3px = (particle.vel\_x + k2vx / 2) \* step\_size;

                float k3py = (particle.vel\_y + k2vy / 2) \* step\_size;

                float k3pz = (particle.vel\_z + k2vz / 2) \* step\_size;

                // k4

                float k4vx = (particle.acc\_x + k3px) \* step\_size;

                float k4vy = (particle.acc\_y + k3py) \* step\_size;

                float k4vz = (particle.acc\_z + k3pz) \* step\_size;

                float k4px = (particle.vel\_x + k3vx) \* step\_size;

                float k4py = (particle.vel\_y + k3vy) \* step\_size;

                float k4pz = (particle.vel\_z + k3vz) \* step\_size;

                // Update velocities

                particle.vel\_x += (k1vx + 2 \* k2vx + 2 \* k3vx + k4vx) / 6;

                particle.vel\_y += (k1vy + 2 \* k2vy + 2 \* k3vy + k4vy) / 6;

                particle.vel\_z += (k1vz + 2 \* k2vz + 2 \* k3vz + k4vz) / 6;

                // Update positions

                particle.pos\_x += (k1px + 2 \* k2px + 2 \* k3px + k4px) / 6;

                particle.pos\_y += (k1py + 2 \* k2py + 2 \* k3py + k4py) / 6;

                particle.pos\_z += (k1pz + 2 \* k2pz + 2 \* k3pz + k4pz) / 6;

            }

        }

    }

    float get\_energy()

    {

        // Calculate the total energy of the system

        float total\_energy = 0;

        ANNOTATE\_SITE\_BEGIN(get\_energy);

        {

            for (const auto &particle : particles)

            {

                ANNOTATE\_ITERATION\_TASK(get\_energy\_0);

                float kinetic\_energy = 0.5 \* particle.mass \* (particle.vel\_x \* particle.vel\_x + particle.vel\_y \* particle.vel\_y + particle.vel\_z \* particle.vel\_z);

                float potential\_energy = 0;

                for (const auto &other : particles)

                {

                    ANNOTATE\_ITERATION\_TASK(get\_energy\_0\_0);

                    if (&particle != &other)

                    {

                        float dx = other.pos\_x - particle.pos\_x;

                        float dy = other.pos\_y - particle.pos\_y;

                        float dz = other.pos\_z - particle.pos\_z;

                        float rSquared = dx \* dx + dy \* dy + dz \* dz + softeningSquared;

                        potential\_energy -= G \* particle.mass \* other.mass / sqrt(rSquared);

                    }

                }

                total\_energy += kinetic\_energy + potential\_energy;

            }

        }

        return total\_energy;

    }

};

int main()

{

    static std::array<Body, N\_BODYES> bodies;

    for (int i = 0; i < N\_BODYES; ++i)

    {

        float pos\_x{random\_float(-MAX\_POS, MAX\_POS)}, pos\_y{random\_float(-MAX\_POS, MAX\_POS)}, pos\_z{random\_float(-MAX\_POS, MAX\_POS)};

        float vel\_x{random\_float(MIN\_VEL, MAX\_VEL)}, vel\_y{random\_float(MIN\_VEL, MAX\_VEL)}, vel\_z{random\_float(MIN\_VEL, MAX\_VEL)};

        float acc\_x{0}, acc\_y{0}, acc\_z{0};

        float mass{random\_float(MIN\_MASS, MAX\_MASS)};

        bodies[i] = Body{pos\_x, pos\_y, pos\_z, vel\_x, vel\_y, vel\_z, acc\_x, acc\_y, acc\_z, mass};

    }

    // Initialize simulation with a step size

    static Simulation sim(dt, bodies); // Step size

    auto start = std::chrono::high\_resolution\_clock::now();

    sim.start(N);

    auto end = std::chrono::high\_resolution\_clock::now();

    auto duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count();

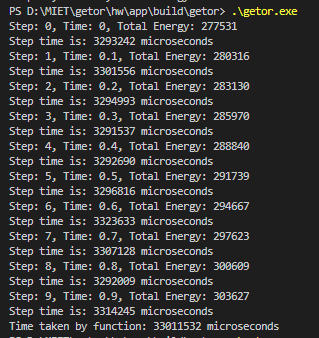
    std::cout << "Time taken by function: " << duration << " microseconds" << std::endl;

    return 0;

}

N\_BODYES = 10000

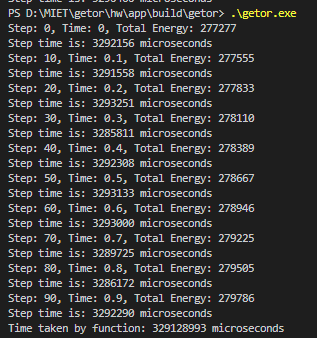
dt = 0.1 сек.



|  |  |
| --- | --- |
| Параметр | Значение параметра |
| Время одной итерации | 3620002 мкс (3.6 сек) |
| Полное время моделирования | 35489915 мкс (35 сек) |
| Накопленная ошибка | ~9.4% |

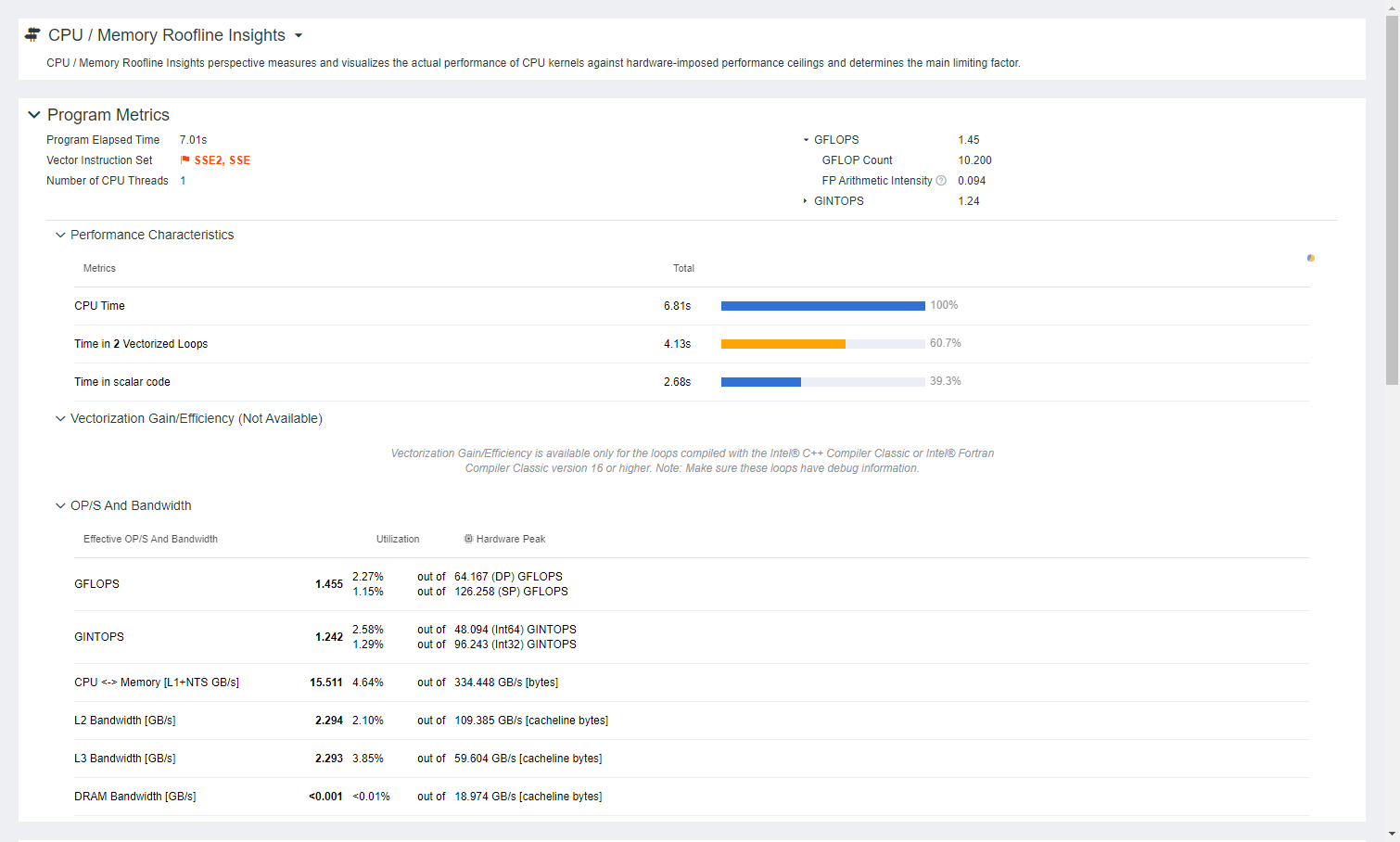
N\_BODYES = 10000

dt = 0.01 сек.

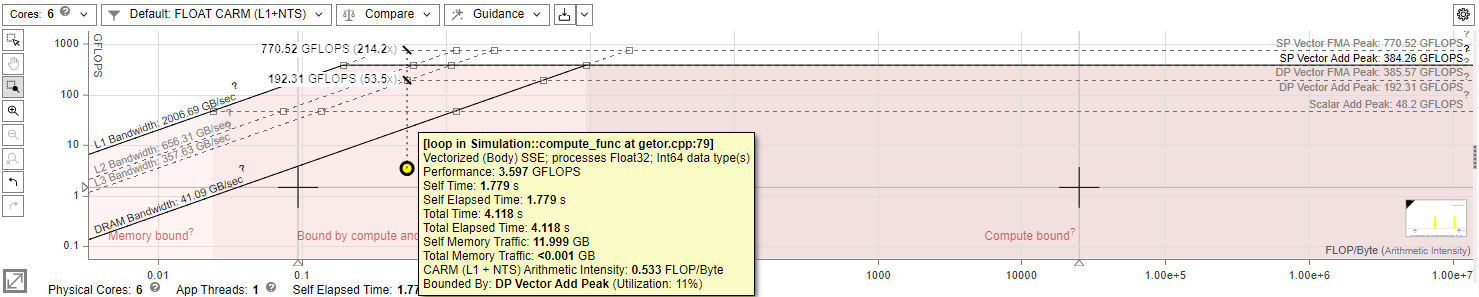


|  |  |
| --- | --- |
| Параметр | Значение параметра |
| Время одной итерации | 3323633 мкс (3.3 сек) |
| Полное время моделирования | 329128993 мкс (330 сек) |
| Накопленная ошибка | ~0.9% |

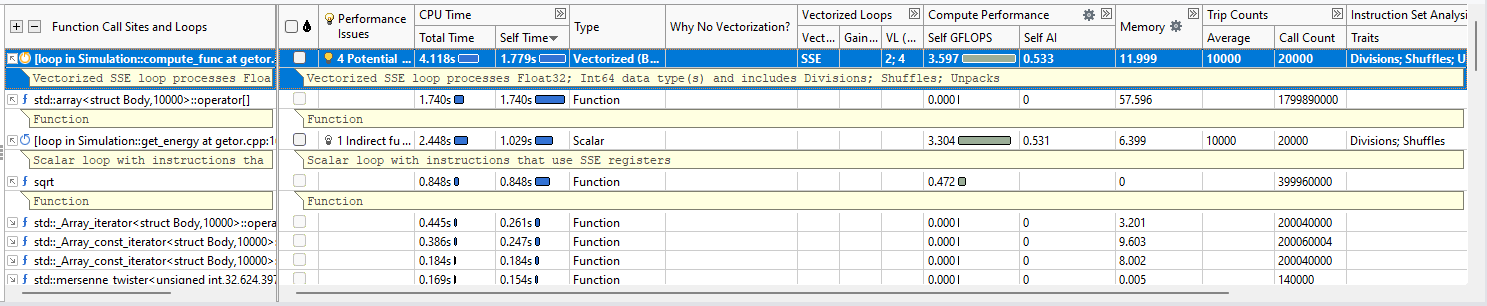
Advisor summary



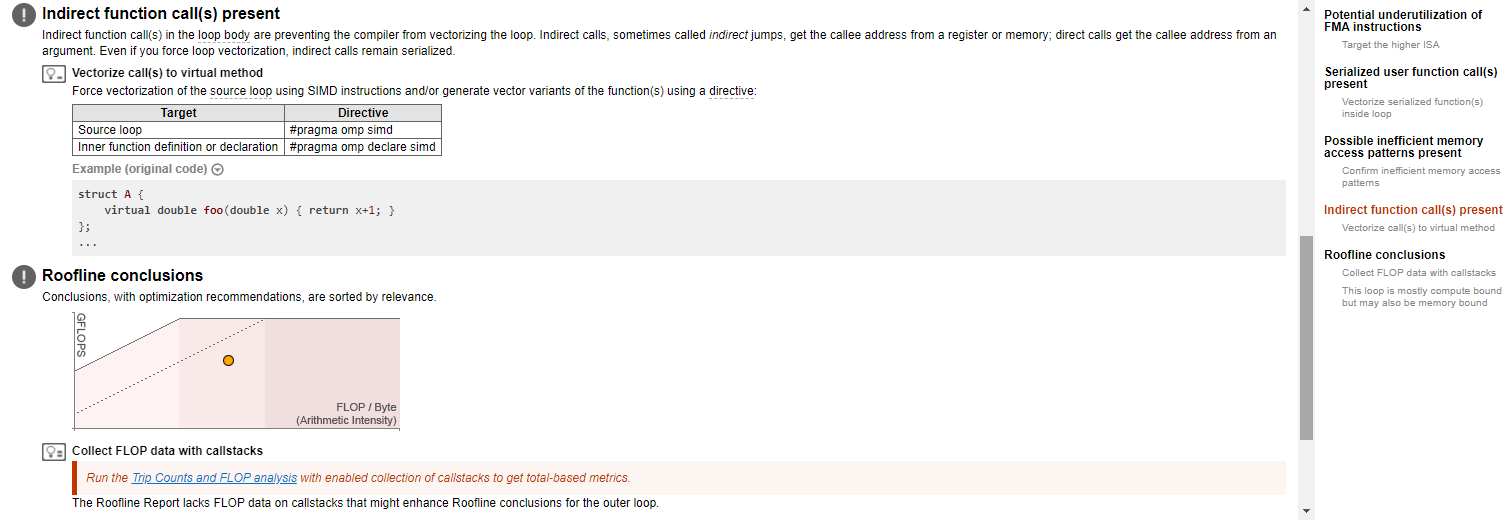
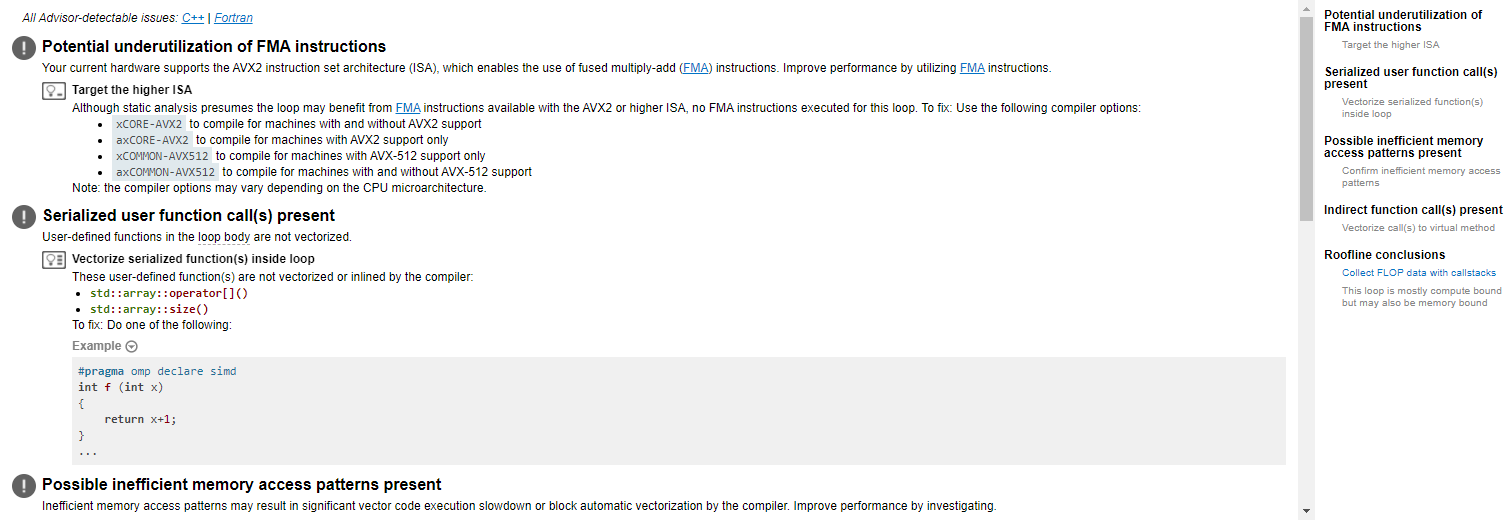
Roofline



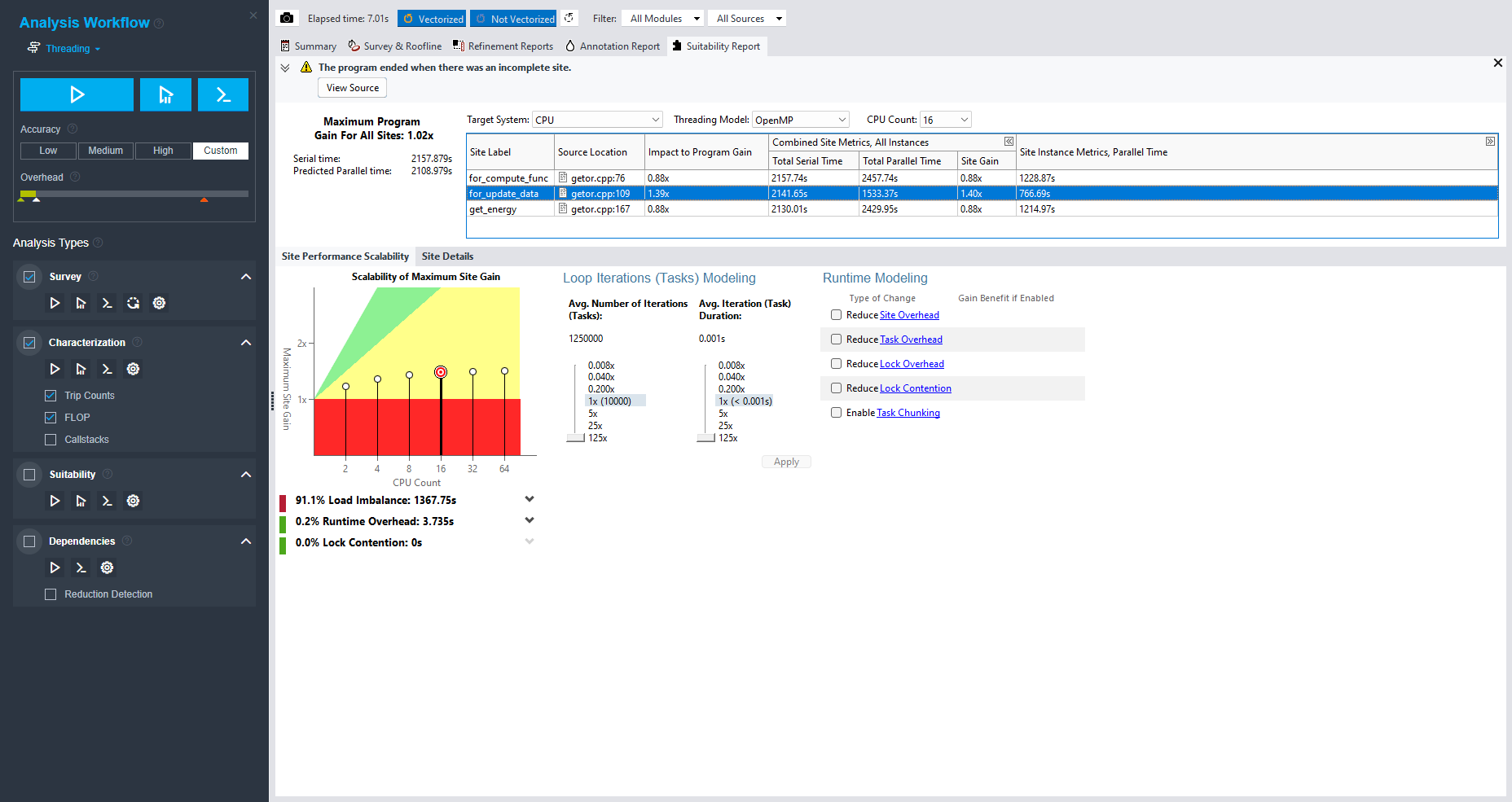
Survey



Recommendations

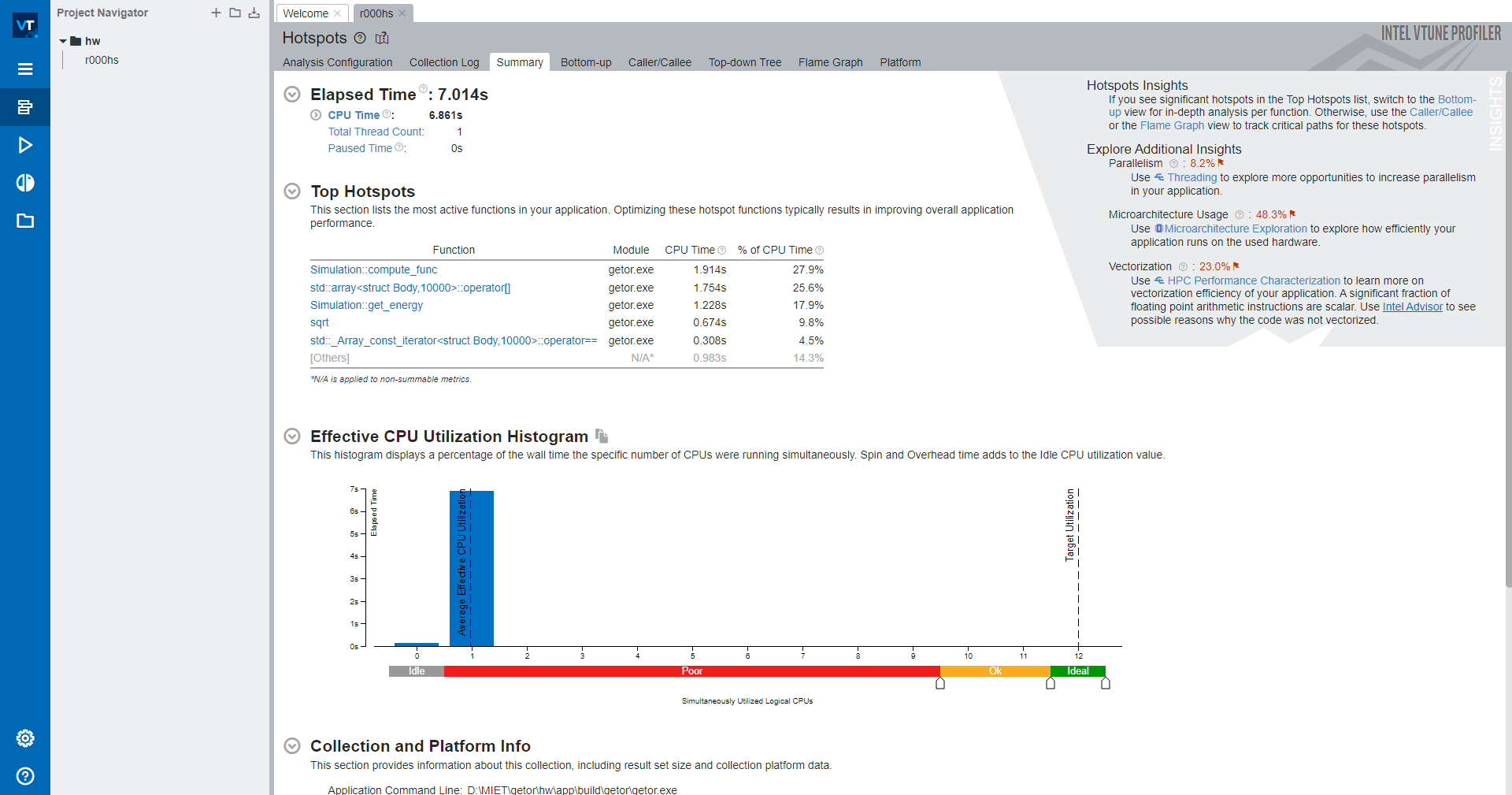


Advisor threading (suitability)

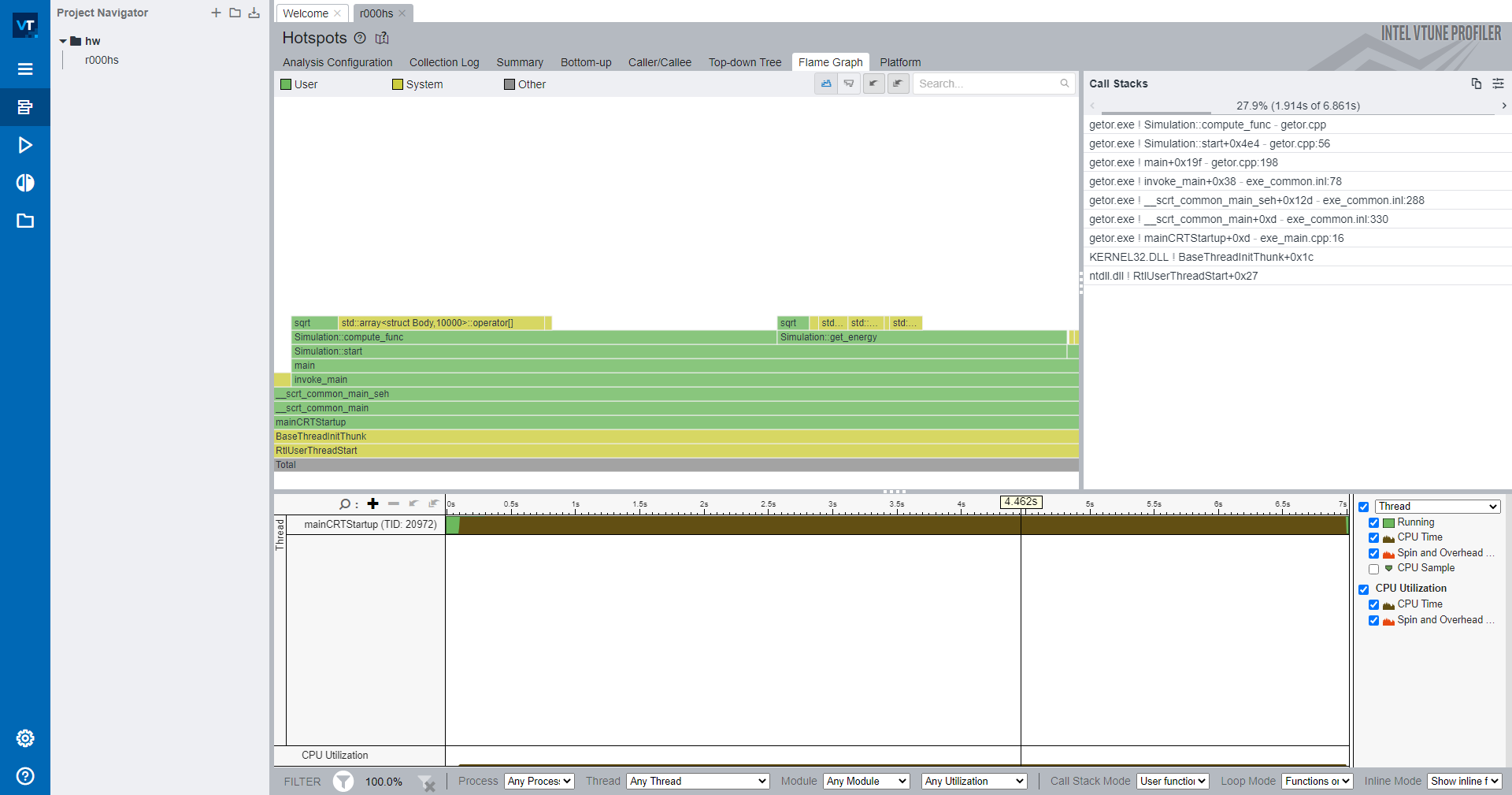


V-tune

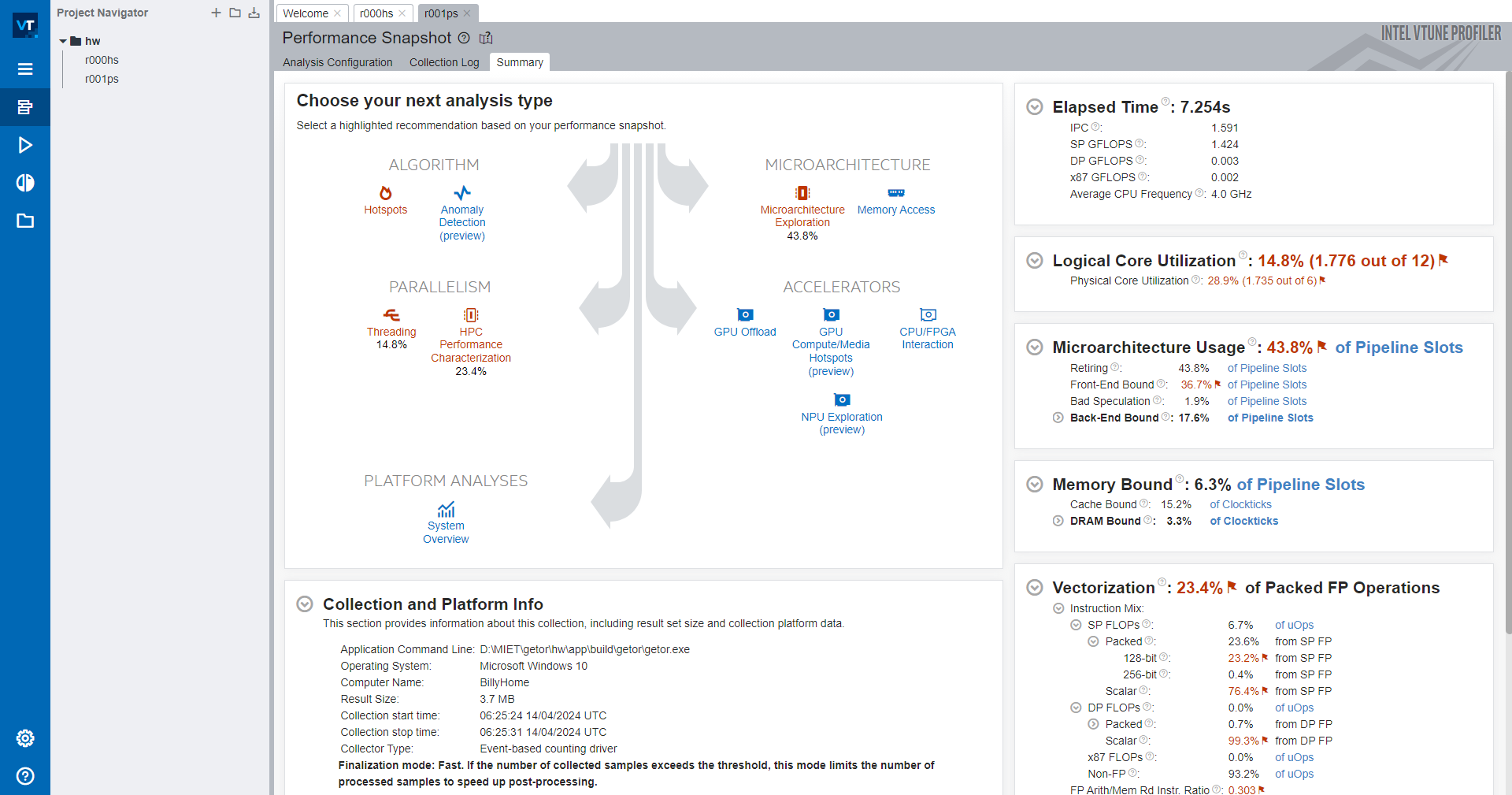
Summary



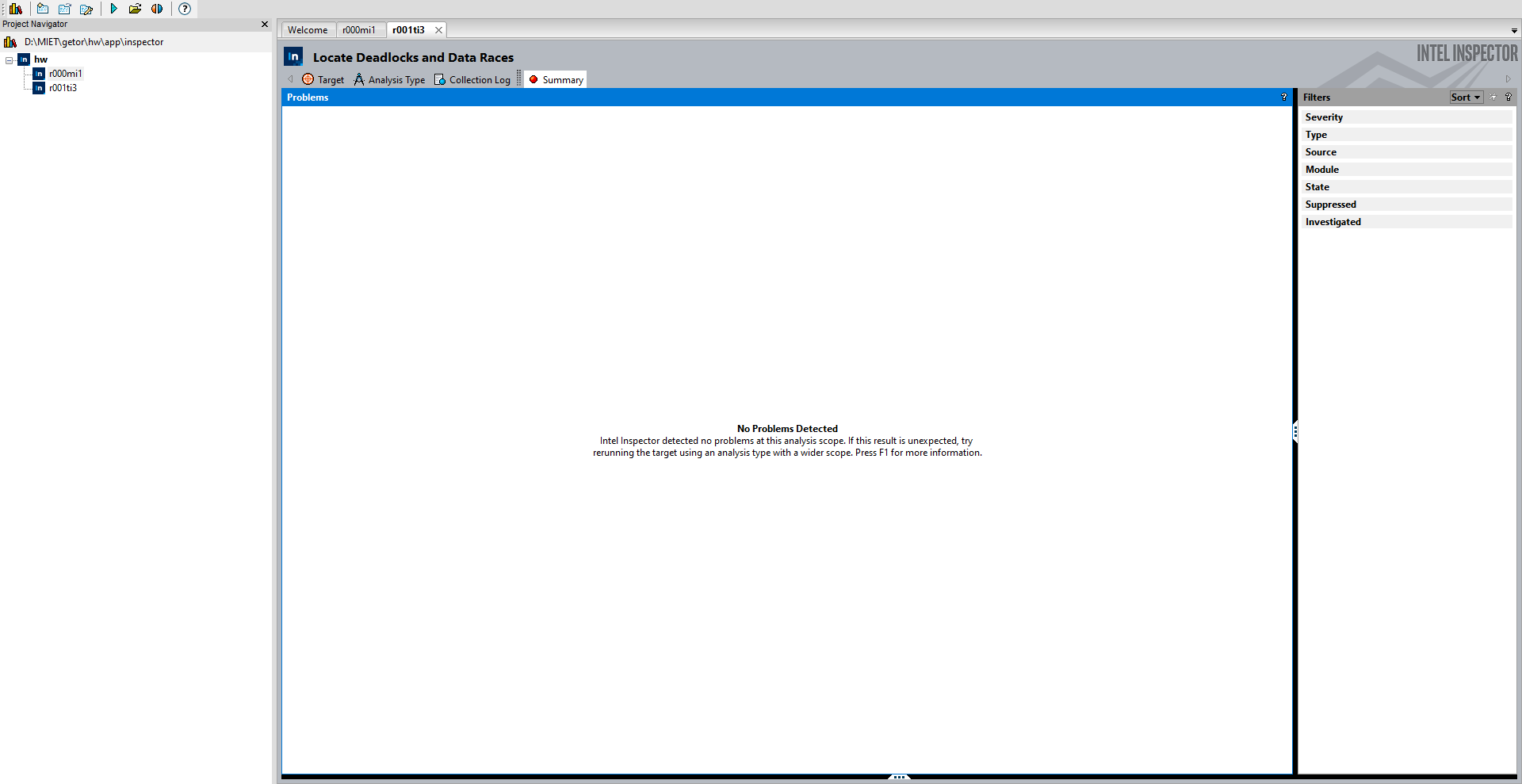
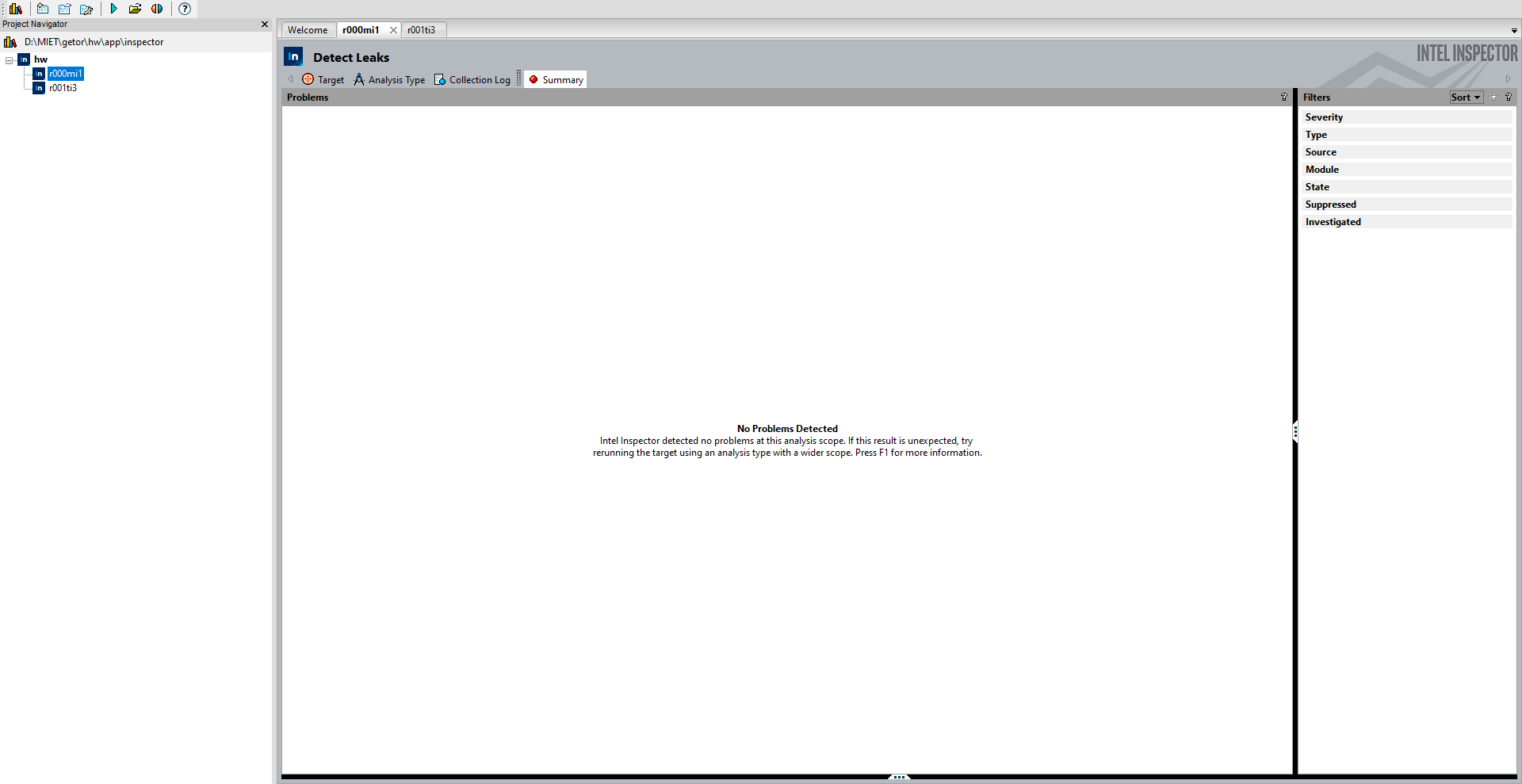
Hotspots



Performance snapshot



Inspector



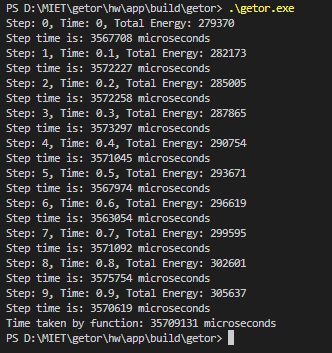
**Оптимизация**

**Через флаги компилятора**

add\_compile\_options(-O3 -Qopt-report=max -debug -Qopenmp /QxCORE-AVX2 ***/Qprec-sqrt- /Qprec-div-***)

N\_BODYES = 10000

dt = 0.1 сек.



|  |  |  |  |
| --- | --- | --- | --- |
| Параметр | Предыдущее значение параметра | Значение параметра | Изменение |
| Время одной итерации | 3620002 мкс (3.62 сек) | 3573297 мкс (3.57 сек) | +1.5% |
| Полное время моделирования | 35489915 мкс (35.5 сек) | 35709131 мкс (35.7 сек) | -0.5% |
| Накопленная ошибка | ~9.4% | ~9.4% | 0% |

**Векторизация/параллелизм update\_data()**

Ввиду независимости компонент в particles при подсчёте новых положения/скорости по методу Руге-Кутта возможно параллелизовать/векторизовать подсчёт компонент.

#pragma omp declare simd

    void update\_data()

    {

        // Use the Runge-Kutta method to update positions and velocities

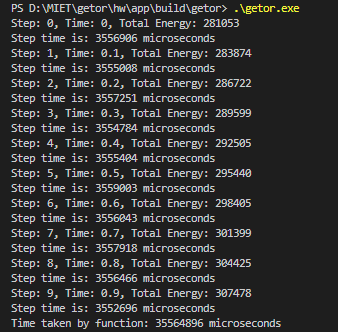
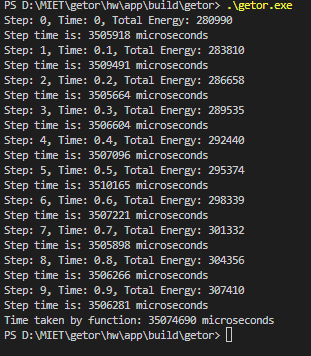
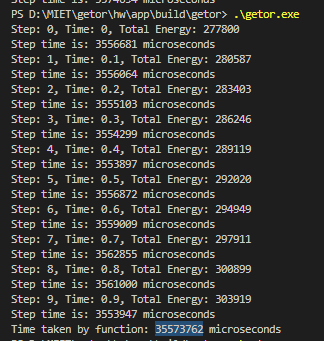
        ANNOTATE\_SITE\_BEGIN(for\_update\_data);

        {

#pragma omp parallel for simd

            for (auto &particle : particles)

            {…

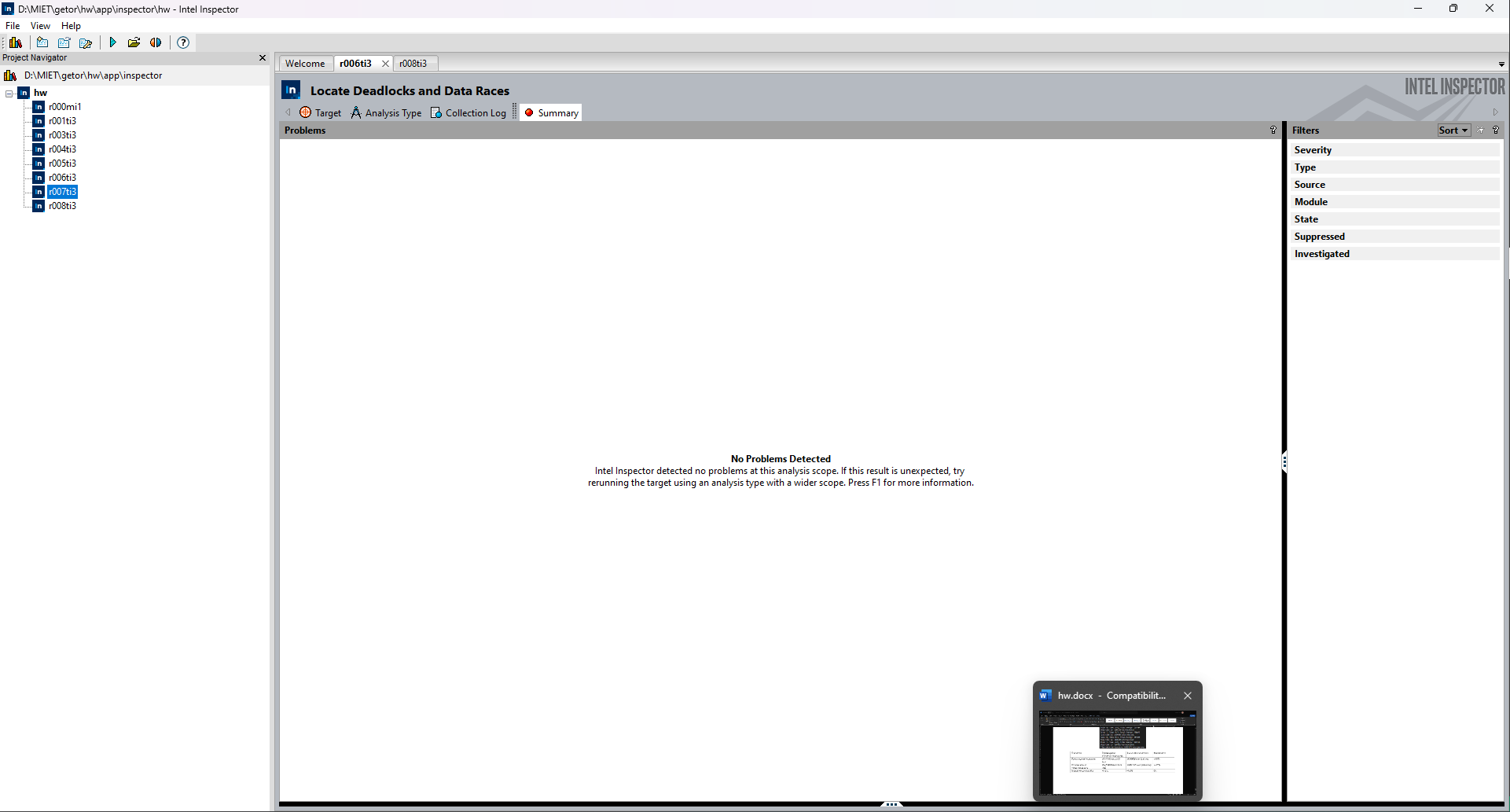


Parallel for only Simd only Parallel for + simd

Оставим только simd

|  |  |  |  |
| --- | --- | --- | --- |
| Параметр | Предыдущее значение параметра | Значение параметра | Изменение |
| Время одной итерации | 3573297 мкс (3.57 сек) | 3510165 мкс (3.51 сек) | +1.7% |
| Полное время моделирования | 35709131 мкс (35.7 сек) | 35074690 мкс (35.1 сек) | +1.7% |
| Накопленная ошибка | ~9.4% | ~9.4% | 0% |

Data races не появились по результатам анализа в inspector



**Параллелизм compute\_func ()**

Ввиду независимости присваивания particles[i] во всех итерациях внешнего цикла возможно параллелизовать подсчёт компонент. Использование simd в этой функции не обосновано и проверяться не будет.

    void compute\_func()

    {

        // Compute the acceleration for each body due to gravitational interactions

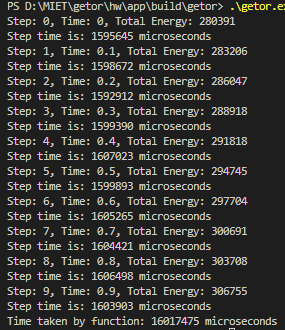
        // using Newton's law of gravitation

        ANNOTATE\_SITE\_BEGIN(for\_compute\_func);

        {

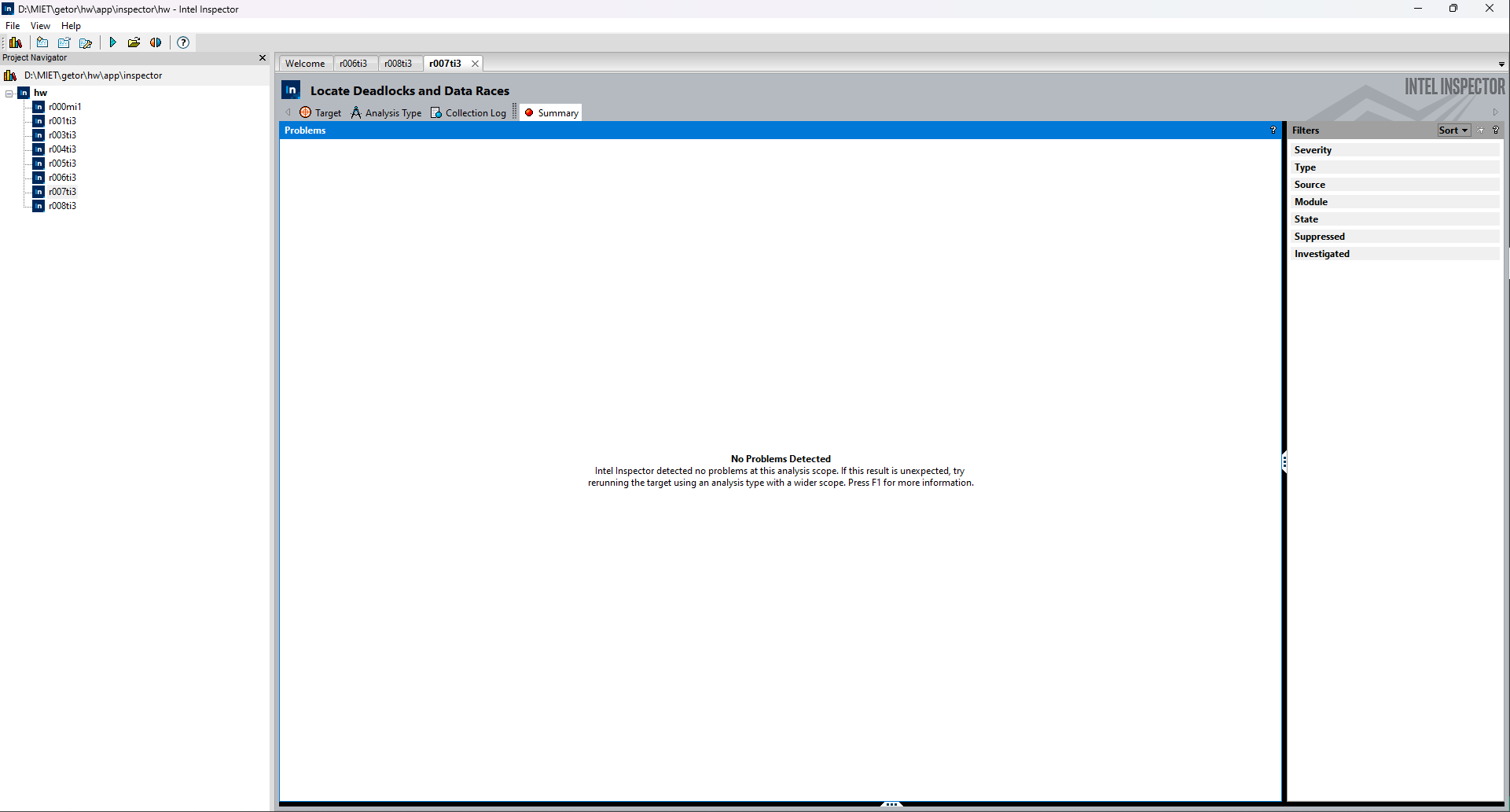
#pragma omp parallel for

            for (int …



|  |  |  |  |
| --- | --- | --- | --- |
| Параметр | Предыдущее значение параметра | Значение параметра | Изменение |
| Время одной итерации | 3510165 мкс (3.51 сек) | 1606498 мкс (1.6 сек) | +220% |
| Полное время моделирования | 35074690 мкс (35.1 сек) | 16017475 мкс (16.0 сек) | +220% |
| Накопленная ошибка | ~9.4% | ~9.4% | 0% |

Data races не появились по результатам анализа в inspector



**Параллелизм get\_energy()**

Ввиду независимости чтения particles[i] во всех итерациях внешнего цикла возможно параллелизовать подсчёт общей. Использование simd в этой функции не обосновано и проверяться не будет.

float get\_energy()

    {

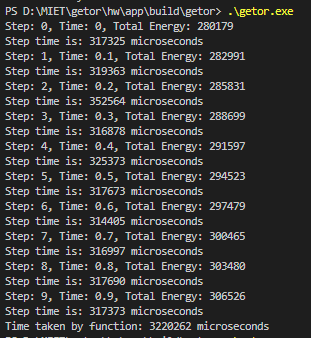
        // Calculate the total energy of the system

        float total\_energy = 0;

#pragma omp parallel for reduction(+ : total\_energy)

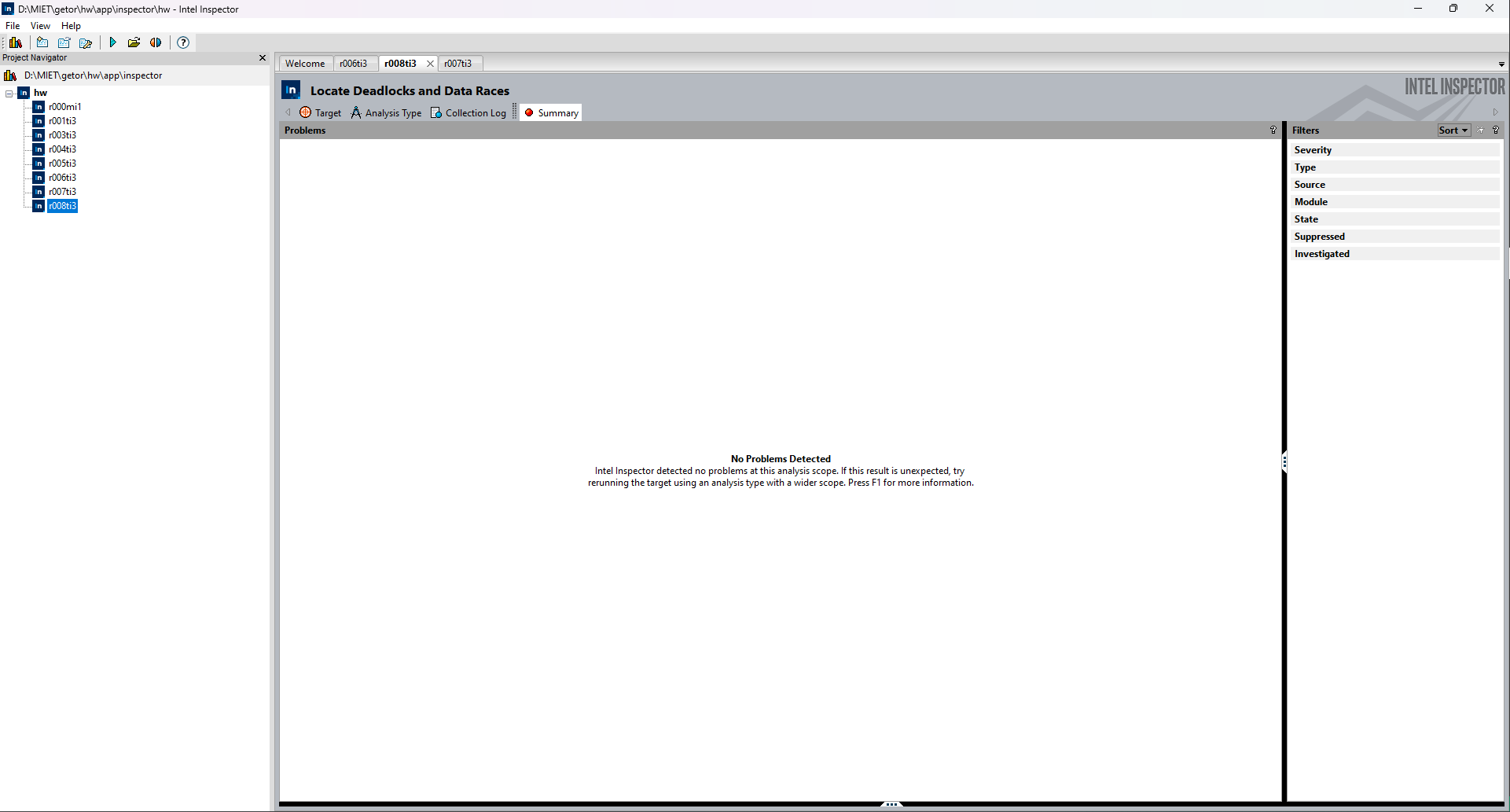
        for (int i = 0; i < particles.size(); ++i)

        {…



|  |  |  |  |
| --- | --- | --- | --- |
| Параметр | Предыдущее значение параметра | Значение параметра | Изменение |
| Время одной итерации | 1606498 мкс (1.6 сек) | 352564 мкс (0.4 сек) | +400% |
| Полное время моделирования | 16017475 мкс (16.0 сек) | 3220262 мкс (3.2 сек) | +500% |
| Накопленная ошибка | ~9.4% | ~9.4% | 0% |

Data races не появились по результатам анализа в inspector



**Final listing**

#include <chrono>

#include <iostream>

#include <random>

#include <array>

#include <advisor-annotate.h>

#include <omp.h>

constexpr int STEP\_PRINT\_MOD{1};

constexpr float dt{0.1};

constexpr float T{1};

constexpr int N{static\_cast<int>(T / dt)};

constexpr int N\_BODYES{10000};

constexpr float MAX\_POS{0.5};

constexpr float MIN\_VEL{0.1};

constexpr float MAX\_VEL{1};

constexpr float MIN\_MASS{0.1};

constexpr float MAX\_MASS{100};

constexpr float G{6.67259e-11};         // Gravitational constant

constexpr float softeningSquared{1e-6}; // Softening parameter

struct Body

{

    float pos\_x, pos\_y, pos\_z;

    float vel\_x, vel\_y, vel\_z;

    float acc\_x, acc\_y, acc\_z;

    float mass;

};

float random\_float(float min, float max)

{

    std::random\_device rd;

    std::mt19937 gen(rd());

    std::uniform\_real\_distribution<> dis(min, max);

    return dis(gen);

}

class Simulation

{

private:

    std::array<Body, N\_BODYES> particles;

    const float step\_size;

public:

    Simulation(const float step\_size, const std::array<Body, N\_BODYES> &bodies) : particles{bodies}, step\_size(step\_size) {}

    void start(int n\_steps)

    {

        for (int step = 0; step < n\_steps; ++step)

        {

            auto start = std::chrono::high\_resolution\_clock::now();

            compute\_func();

            update\_data();

            float current\_time = step \* step\_size;

            float total\_energy = get\_energy();

            auto end = std::chrono::high\_resolution\_clock::now();

            if (step % STEP\_PRINT\_MOD == 0)

            {

                std::cout << "Step: " << step << ", Time: " << current\_time << ", Total Energy: " << total\_energy << std::endl;

                auto duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count();

                std::cout << "Step time is: " << duration << " microseconds" << std::endl;

            }

        }

    }

private:

#pragma omp declare simd

    void compute\_func()

    {

        // Compute the acceleration for each body due to gravitational interactions

        // using Newton's law of gravitation

        ANNOTATE\_SITE\_BEGIN(for\_compute\_func);

        {

#pragma omp parallel for

            for (int i = 0; i < particles.size(); ++i)

            {

                ANNOTATE\_ITERATION\_TASK(for\_compute\_func\_0);

                float acc\_x = 0, acc\_y = 0, acc\_z = 0;

                for (int j = 0; j < particles.size(); ++j)

                {

                    ANNOTATE\_ITERATION\_TASK(for\_compute\_func\_0\_0);

                    if (i != j)

                    {

                        float dx = particles[j].pos\_x - particles[i].pos\_x;

                        float dy = particles[j].pos\_y - particles[i].pos\_y;

                        float dz = particles[j].pos\_z - particles[i].pos\_z;

                        float rSquared = dx \* dx + dy \* dy + dz \* dz + softeningSquared;

                        float inv\_r = 1.0 / sqrt(rSquared);

                        float inv\_r3 = inv\_r \* inv\_r \* inv\_r;

                        acc\_x += G \* particles[j].mass \* dx \* inv\_r3;

                        acc\_y += G \* particles[j].mass \* dy \* inv\_r3;

                        acc\_z += G \* particles[j].mass \* dz \* inv\_r3;

                    }

                }

                particles[i].acc\_x = acc\_x;

                particles[i].acc\_y = acc\_y;

                particles[i].acc\_z = acc\_z;

            }

        }

    }

#pragma omp declare simd

    void update\_data()

    {

        // Use the Runge-Kutta method to update positions and velocities

        ANNOTATE\_SITE\_BEGIN(for\_update\_data);

        {

#pragma omp simd

            for (auto &particle : particles)

            {

                ANNOTATE\_ITERATION\_TASK(for\_update\_data\_0);

                // k1

                float k1vx = particle.acc\_x \* step\_size;

                float k1vy = particle.acc\_y \* step\_size;

                float k1vz = particle.acc\_z \* step\_size;

                float k1px = particle.vel\_x \* step\_size;

                float k1py = particle.vel\_y \* step\_size;

                float k1pz = particle.vel\_z \* step\_size;

                // k2

                float k2vx = (particle.acc\_x + k1px / 2) \* step\_size;

                float k2vy = (particle.acc\_y + k1py / 2) \* step\_size;

                float k2vz = (particle.acc\_z + k1pz / 2) \* step\_size;

                float k2px = (particle.vel\_x + k1vx / 2) \* step\_size;

                float k2py = (particle.vel\_y + k1vy / 2) \* step\_size;

                float k2pz = (particle.vel\_z + k1vz / 2) \* step\_size;

                // k3

                float k3vx = (particle.acc\_x + k2px / 2) \* step\_size;

                float k3vy = (particle.acc\_y + k2py / 2) \* step\_size;

                float k3vz = (particle.acc\_z + k2pz / 2) \* step\_size;

                float k3px = (particle.vel\_x + k2vx / 2) \* step\_size;

                float k3py = (particle.vel\_y + k2vy / 2) \* step\_size;

                float k3pz = (particle.vel\_z + k2vz / 2) \* step\_size;

                // k4

                float k4vx = (particle.acc\_x + k3px) \* step\_size;

                float k4vy = (particle.acc\_y + k3py) \* step\_size;

                float k4vz = (particle.acc\_z + k3pz) \* step\_size;

                float k4px = (particle.vel\_x + k3vx) \* step\_size;

                float k4py = (particle.vel\_y + k3vy) \* step\_size;

                float k4pz = (particle.vel\_z + k3vz) \* step\_size;

                // Update velocities

                particle.vel\_x += (k1vx + 2 \* k2vx + 2 \* k3vx + k4vx) / 6;

                particle.vel\_y += (k1vy + 2 \* k2vy + 2 \* k3vy + k4vy) / 6;

                particle.vel\_z += (k1vz + 2 \* k2vz + 2 \* k3vz + k4vz) / 6;

                // Update positions

                particle.pos\_x += (k1px + 2 \* k2px + 2 \* k3px + k4px) / 6;

                particle.pos\_y += (k1py + 2 \* k2py + 2 \* k3py + k4py) / 6;

                particle.pos\_z += (k1pz + 2 \* k2pz + 2 \* k3pz + k4pz) / 6;

            }

        }

    }

#pragma omp declare simd

    float get\_energy()

    {

        // Calculate the total energy of the system

        float total\_energy = 0;

#pragma omp parallel for reduction(+ : total\_energy)

        for (int i = 0; i < particles.size(); ++i)

        {

            const auto &particle = particles[i];

            float kinetic\_energy = 0.5 \* particle.mass \* (particle.vel\_x \* particle.vel\_x + particle.vel\_y \* particle.vel\_y + particle.vel\_z \* particle.vel\_z);

            float potential\_energy = 0;

            for (int j = 0; j < particles.size(); ++j)

            {

                if (i != j)

                {

                    const auto &other = particles[j];

                    float dx = other.pos\_x - particle.pos\_x;

                    float dy = other.pos\_y - particle.pos\_y;

                    float dz = other.pos\_z - particle.pos\_z;

                    float rSquared = dx \* dx + dy \* dy + dz \* dz + softeningSquared;

                    potential\_energy -= G \* particle.mass \* other.mass / sqrt(rSquared);

                }

            }

            total\_energy += kinetic\_energy + potential\_energy;

        }

        return total\_energy;

    }

};

int main()

{

    static std::array<Body, N\_BODYES> bodies;

    for (int i = 0; i < N\_BODYES; ++i)

    {

        float pos\_x{random\_float(-MAX\_POS, MAX\_POS)}, pos\_y{random\_float(-MAX\_POS, MAX\_POS)}, pos\_z{random\_float(-MAX\_POS, MAX\_POS)};

        float vel\_x{random\_float(MIN\_VEL, MAX\_VEL)}, vel\_y{random\_float(MIN\_VEL, MAX\_VEL)}, vel\_z{random\_float(MIN\_VEL, MAX\_VEL)};

        float acc\_x{0}, acc\_y{0}, acc\_z{0};

        float mass{random\_float(MIN\_MASS, MAX\_MASS)};

        bodies[i] = Body{pos\_x, pos\_y, pos\_z, vel\_x, vel\_y, vel\_z, acc\_x, acc\_y, acc\_z, mass};

    }

    // Initialize simulation with a step size

    static Simulation sim(dt, bodies); // Step size

    auto start = std::chrono::high\_resolution\_clock::now();

    sim.start(N);

    auto end = std::chrono::high\_resolution\_clock::now();

    auto duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count();

    std::cout << "Time taken by function: " << duration << " microseconds" << std::endl;

    return 0;

}

**Aos to soa**

typedef struct Body

{

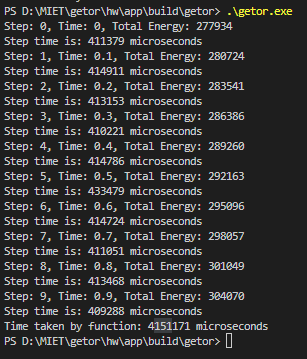
    std::array<float, N\_BODYES> pos\_x, pos\_y, pos\_z;

    std::array<float, N\_BODYES> vel\_x, vel\_y, vel\_z;

    std::array<float, N\_BODYES> acc\_x, acc\_y, acc\_z;

    std::array<float, N\_BODYES> mass;

} Body;



|  |  |  |  |
| --- | --- | --- | --- |
| Параметр | Предыдущее значение параметра | Значение параметра | Изменение |
| Время одной итерации | 352564 мкс (0.4 сек) | 433479 мкс (0.4 сек) | -19% |
| Полное время моделирования | 3220262 мкс (3.2 сек) | 4151171 мкс (4.2 сек) | -22% |
| Накопленная ошибка | ~9.4% | ~9.4% | 0% |

**Aos to soa listing**

#include <chrono>

#include <iostream>

#include <random>

#include <array>

#include <advisor-annotate.h>

#include <omp.h>

constexpr int STEP\_PRINT\_MOD{1};

constexpr float dt{0.1};

constexpr float T{1};

constexpr int N{static\_cast<int>(T / dt)};

constexpr int N\_BODYES{10000};

constexpr float MAX\_POS{0.5};

constexpr float MIN\_VEL{0.1};

constexpr float MAX\_VEL{1};

constexpr float MIN\_MASS{0.1};

constexpr float MAX\_MASS{100};

constexpr float G{6.67259e-11};         // Gravitational constant

constexpr float softeningSquared{1e-6}; // Softening parameter

typedef struct Body

{

    std::array<float, N\_BODYES> pos\_x, pos\_y, pos\_z;

    std::array<float, N\_BODYES> vel\_x, vel\_y, vel\_z;

    std::array<float, N\_BODYES> acc\_x, acc\_y, acc\_z;

    std::array<float, N\_BODYES> mass;

} Body;

float random\_float(float min, float max)

{

    std::random\_device rd;

    std::mt19937 gen(rd());

    std::uniform\_real\_distribution<> dis(min, max);

    return dis(gen);

}

class Simulation

{

private:

    Body particles;

    const float step\_size;

public:

    Simulation(const float step\_size, const Body &bodies) : particles{bodies}, step\_size(step\_size) {}

    void start(int n\_steps)

    {

        for (int step = 0; step < n\_steps; ++step)

        {

            auto start = std::chrono::high\_resolution\_clock::now();

            compute\_func();

            update\_data();

            float current\_time = step \* step\_size;

            float total\_energy = get\_energy();

            auto end = std::chrono::high\_resolution\_clock::now();

            if (step % STEP\_PRINT\_MOD == 0)

            {

                std::cout << "Step: " << step << ", Time: " << current\_time << ", Total Energy: " << total\_energy << std::endl;

                auto duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count();

                std::cout << "Step time is: " << duration << " microseconds" << std::endl;

            }

        }

    }

private:

#pragma omp declare simd

    void compute\_func()

    {

        // Compute the acceleration for each body due to gravitational interactions

        // using Newton's law of gravitation

        ANNOTATE\_SITE\_BEGIN(for\_compute\_func);

        {

#pragma omp parallel for

            for (int i = 0; i < N\_BODYES; ++i)

            {

                ANNOTATE\_ITERATION\_TASK(for\_compute\_func\_0);

                float acc\_x = 0, acc\_y = 0, acc\_z = 0;

                for (int j = 0; j < N\_BODYES; ++j)

                {

                    ANNOTATE\_ITERATION\_TASK(for\_compute\_func\_0\_0);

                    if (i != j)

                    {

                        float dx = particles.pos\_x[j] - particles.pos\_x[i];

                        float dy = particles.pos\_y[j] - particles.pos\_y[i];

                        float dz = particles.pos\_z[j] - particles.pos\_z[i];

                        float rSquared = dx \* dx + dy \* dy + dz \* dz + softeningSquared;

                        float inv\_r = 1.0 / sqrt(rSquared);

                        float inv\_r3 = inv\_r \* inv\_r \* inv\_r;

                        acc\_x += G \* particles.mass[j] \* dx \* inv\_r3;

                        acc\_y += G \* particles.mass[j] \* dy \* inv\_r3;

                        acc\_z += G \* particles.mass[j] \* dz \* inv\_r3;

                    }

                }

                particles.acc\_x[i] = acc\_x;

                particles.acc\_y[i] = acc\_y;

                particles.acc\_z[i] = acc\_z;

            }

        }

    }

#pragma omp declare simd

    void update\_data()

    {

        // Use the Runge-Kutta method to update positions and velocities

        ANNOTATE\_SITE\_BEGIN(for\_update\_data);

        {

#pragma omp simd

            for (int i = 0; i < N\_BODYES; ++i)

            {

                ANNOTATE\_ITERATION\_TASK(for\_update\_data\_0);

                // k1

                float k1vx = particles.acc\_x[i] \* step\_size;

                float k1vy = particles.acc\_y[i] \* step\_size;

                float k1vz = particles.acc\_z[i] \* step\_size;

                float k1px = particles.vel\_x[i] \* step\_size;

                float k1py = particles.vel\_y[i] \* step\_size;

                float k1pz = particles.vel\_z[i] \* step\_size;

                // k2

                float k2vx = (particles.acc\_x[i] + k1px / 2) \* step\_size;

                float k2vy = (particles.acc\_y[i] + k1py / 2) \* step\_size;

                float k2vz = (particles.acc\_z[i] + k1pz / 2) \* step\_size;

                float k2px = (particles.vel\_x[i] + k1vx / 2) \* step\_size;

                float k2py = (particles.vel\_y[i] + k1vy / 2) \* step\_size;

                float k2pz = (particles.vel\_z[i] + k1vz / 2) \* step\_size;

                // k3

                float k3vx = (particles.acc\_x[i] + k2px / 2) \* step\_size;

                float k3vy = (particles.acc\_y[i] + k2py / 2) \* step\_size;

                float k3vz = (particles.acc\_z[i] + k2pz / 2) \* step\_size;

                float k3px = (particles.vel\_x[i] + k2vx / 2) \* step\_size;

                float k3py = (particles.vel\_y[i] + k2vy / 2) \* step\_size;

                float k3pz = (particles.vel\_z[i] + k2vz / 2) \* step\_size;

                // k4

                float k4vx = (particles.acc\_x[i] + k3px) \* step\_size;

                float k4vy = (particles.acc\_y[i] + k3py) \* step\_size;

                float k4vz = (particles.acc\_z[i] + k3pz) \* step\_size;

                float k4px = (particles.vel\_x[i] + k3vx) \* step\_size;

                float k4py = (particles.vel\_y[i] + k3vy) \* step\_size;

                float k4pz = (particles.vel\_z[i] + k3vz) \* step\_size;

                // Update velocities

                particles.vel\_x[i] += (k1vx + 2 \* k2vx + 2 \* k3vx + k4vx) / 6;

                particles.vel\_y[i] += (k1vy + 2 \* k2vy + 2 \* k3vy + k4vy) / 6;

                particles.vel\_z[i] += (k1vz + 2 \* k2vz + 2 \* k3vz + k4vz) / 6;

                // Update positions

                particles.pos\_x[i] += (k1px + 2 \* k2px + 2 \* k3px + k4px) / 6;

                particles.pos\_y[i] += (k1py + 2 \* k2py + 2 \* k3py + k4py) / 6;

                particles.pos\_z[i] += (k1pz + 2 \* k2pz + 2 \* k3pz + k4pz) / 6;

            }

        }

    }

#pragma omp declare simd

    float get\_energy()

    {

        // Calculate the total energy of the system

        float total\_energy = 0;

#pragma omp parallel for reduction(+ : total\_energy)

        for (int i = 0; i < N\_BODYES; ++i)

        {

            // const auto &particle = particles[i];

            float kinetic\_energy = 0.5 \* particles.mass[i] \* (particles.vel\_x[i] \* particles.vel\_x[i] + particles.vel\_y[i] \* particles.vel\_y[i] + particles.vel\_z[i] \* particles.vel\_z[i]);

            float potential\_energy = 0;

            for (int j = 0; j < N\_BODYES; ++j)

            {

                if (i != j)

                {

                    // const auto &other = particles[j];

                    float dx = particles.pos\_x[j] - particles.pos\_x[i];

                    float dy = particles.pos\_y[j] - particles.pos\_y[i];

                    float dz = particles.pos\_z[j] - particles.pos\_z[i];

                    float rSquared = dx \* dx + dy \* dy + dz \* dz + softeningSquared;

                    potential\_energy -= G \* particles.mass[i] \* particles.mass[j] / sqrt(rSquared);

                }

            }

            total\_energy += kinetic\_energy + potential\_energy;

        }

        return total\_energy;

    }

};

int main()

{

    static Body bodies;

    for (int i = 0; i < N\_BODYES; ++i)

    {

        float pos\_x{random\_float(-MAX\_POS, MAX\_POS)}, pos\_y{random\_float(-MAX\_POS, MAX\_POS)}, pos\_z{random\_float(-MAX\_POS, MAX\_POS)};

        float vel\_x{random\_float(MIN\_VEL, MAX\_VEL)}, vel\_y{random\_float(MIN\_VEL, MAX\_VEL)}, vel\_z{random\_float(MIN\_VEL, MAX\_VEL)};

        float acc\_x{0}, acc\_y{0}, acc\_z{0};

        float mass{random\_float(MIN\_MASS, MAX\_MASS)};

        bodies.pos\_x[i] = pos\_x;

        bodies.pos\_y[i] = pos\_y;

        bodies.pos\_z[i] = pos\_z;

        bodies.vel\_x[i] = vel\_x;

        bodies.vel\_y[i] = vel\_y;

        bodies.vel\_z[i] = vel\_z;

        bodies.acc\_x[i] = acc\_x;

        bodies.acc\_y[i] = acc\_y;

        bodies.acc\_z[i] = acc\_z;

        bodies.mass[i] = mass;

    };

    // Initialize simulation with a step size

    static Simulation sim(dt, bodies); // Step size

    auto start = std::chrono::high\_resolution\_clock::now();

    sim.start(N);

    auto end = std::chrono::high\_resolution\_clock::now();

    auto duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count();

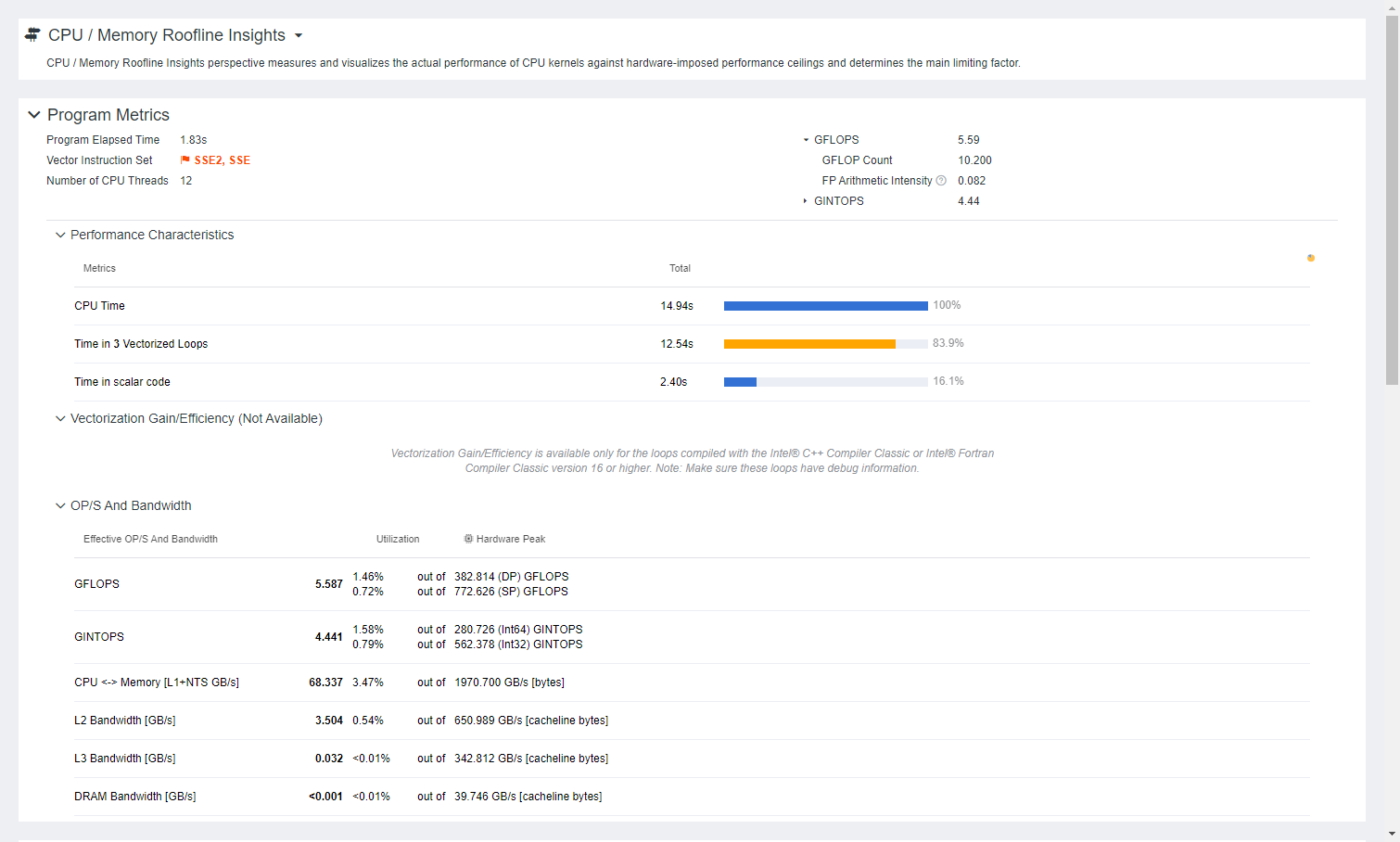
    std::cout << "Time taken by function: " << duration << " microseconds" << std::endl;

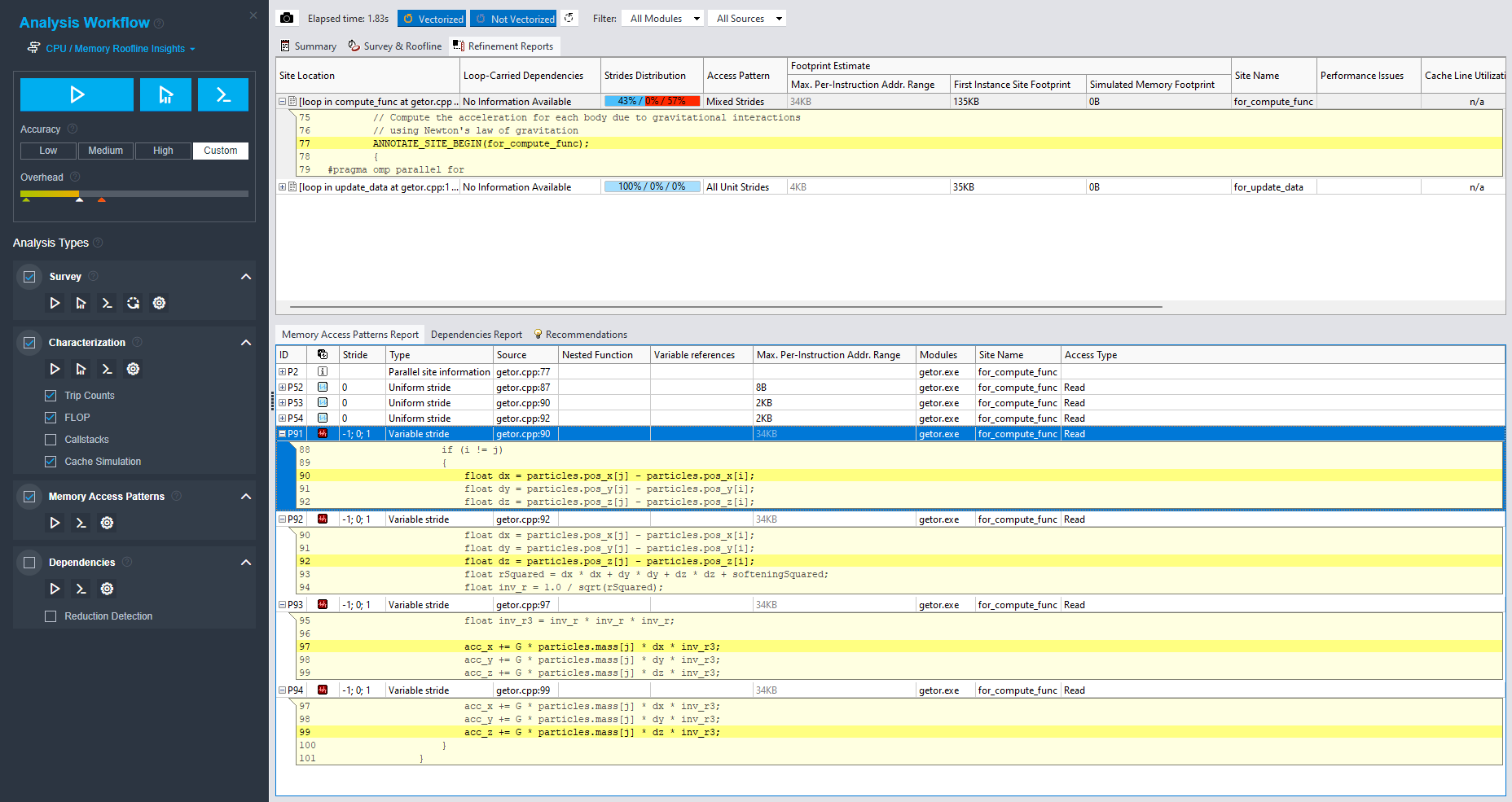
    return 0;

}

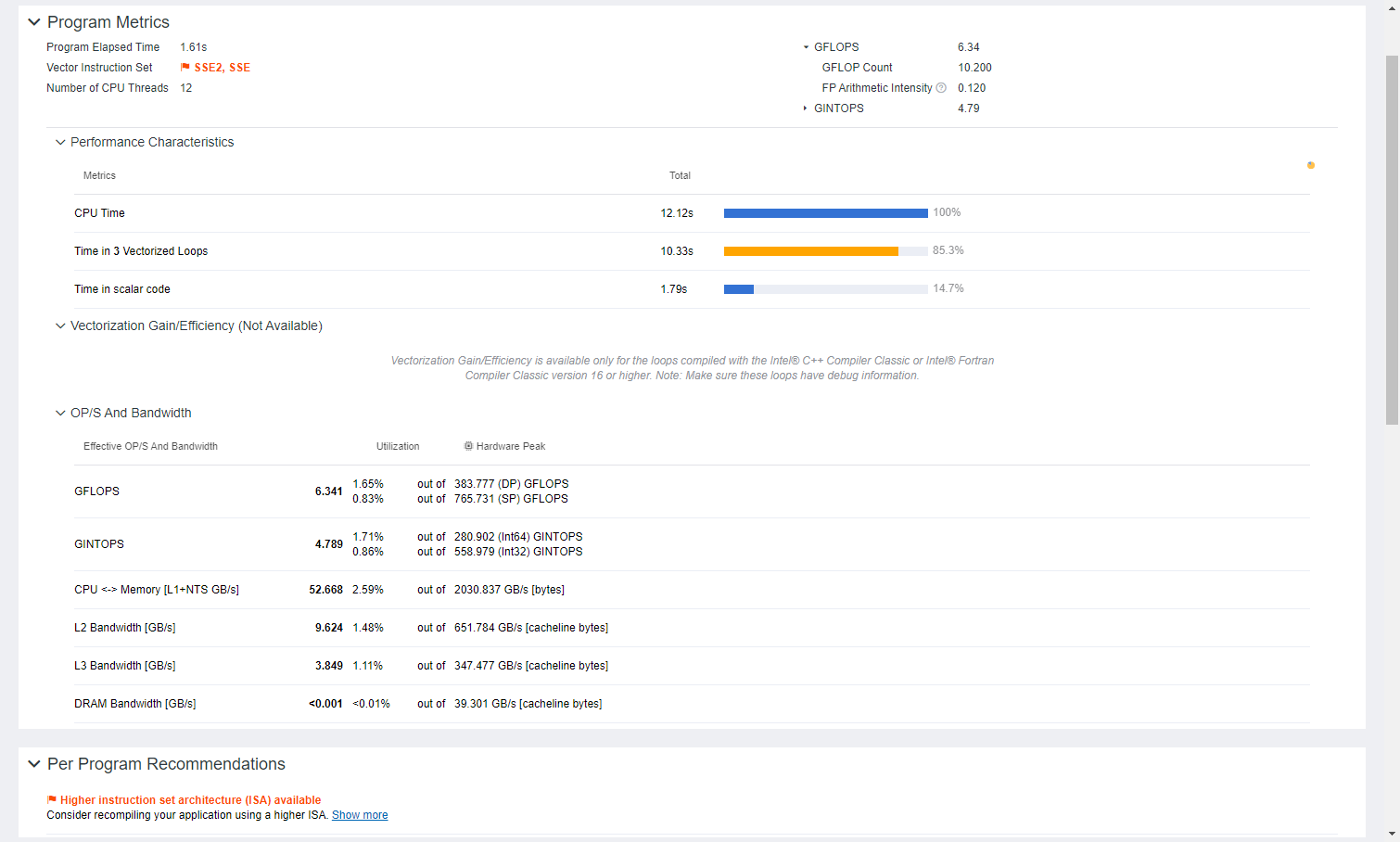
**Итоговые метрики**

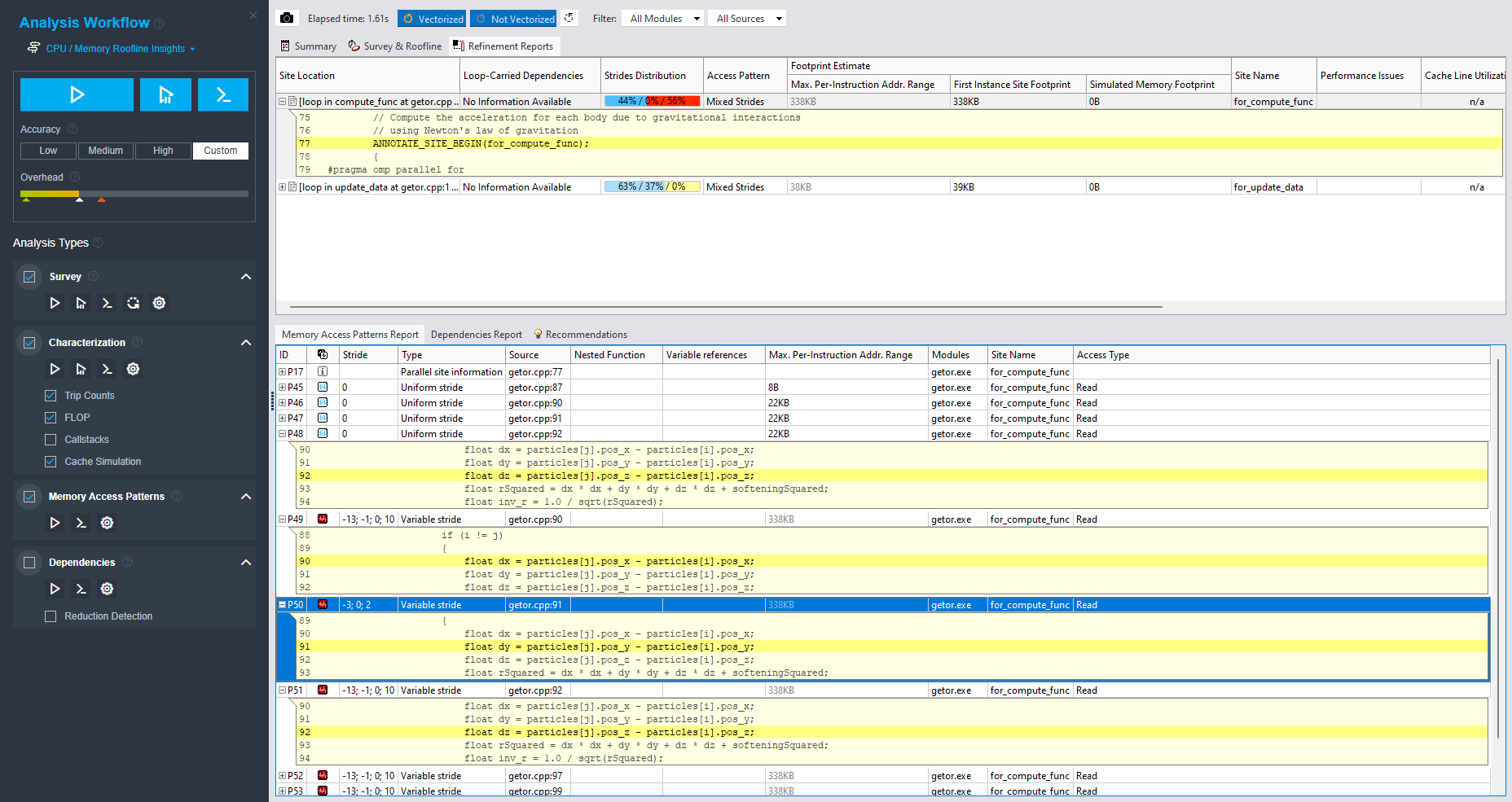
**Advisor summary (SOA)**



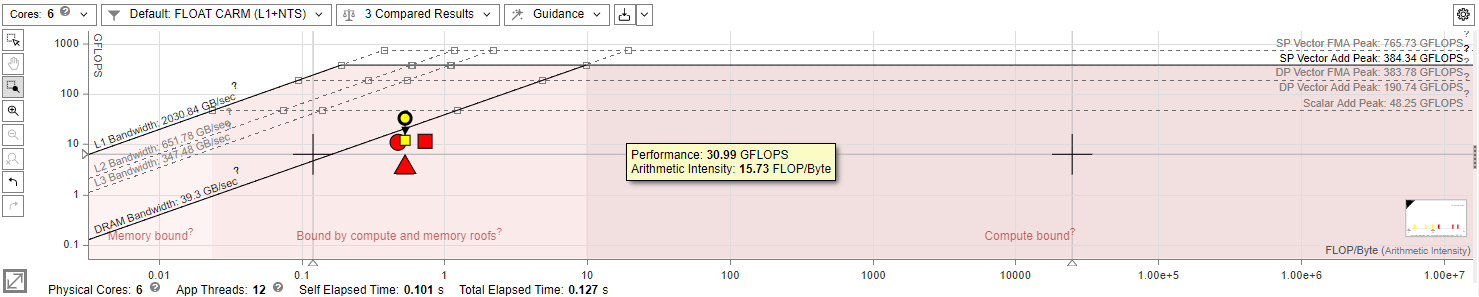


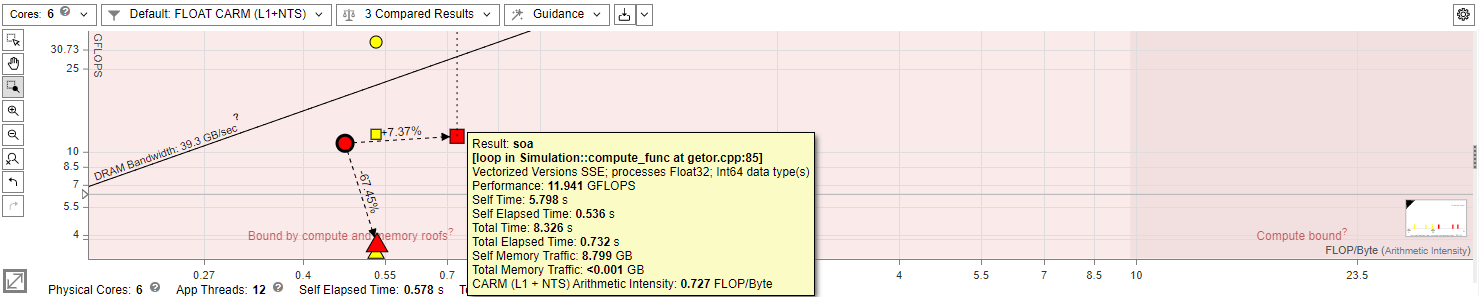
**Advisor summary (AOS)**



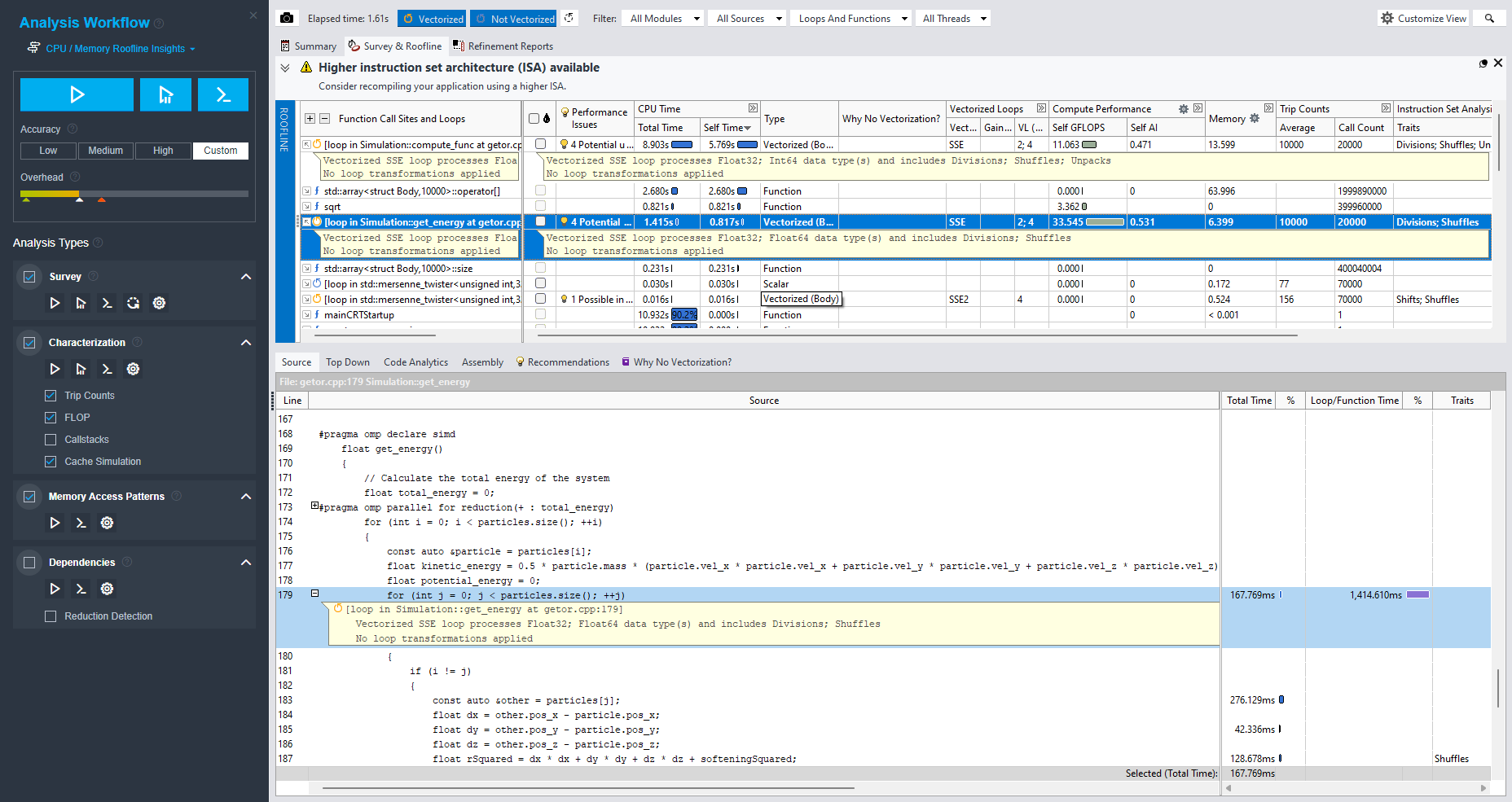


**Advisor roofline**

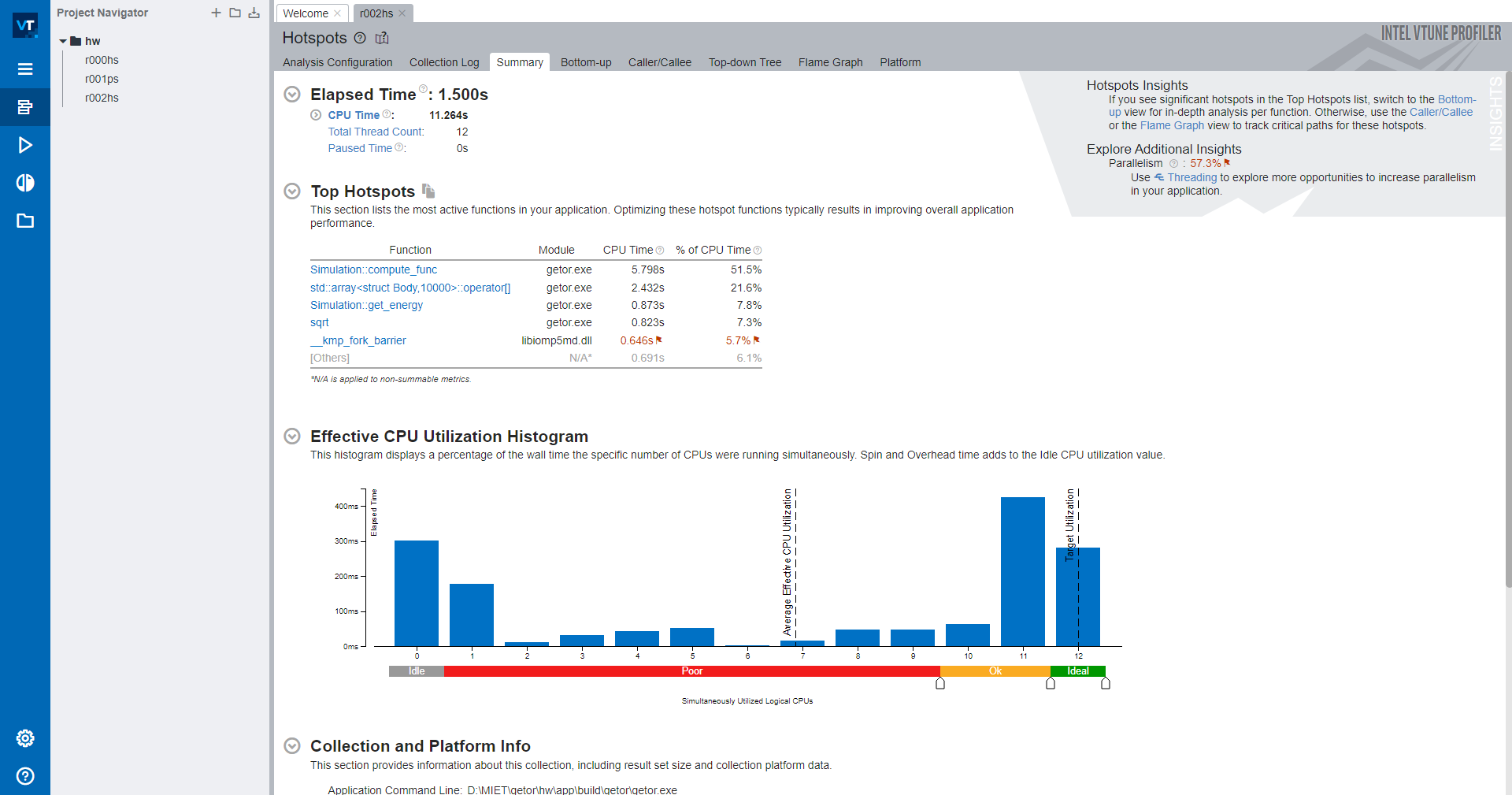




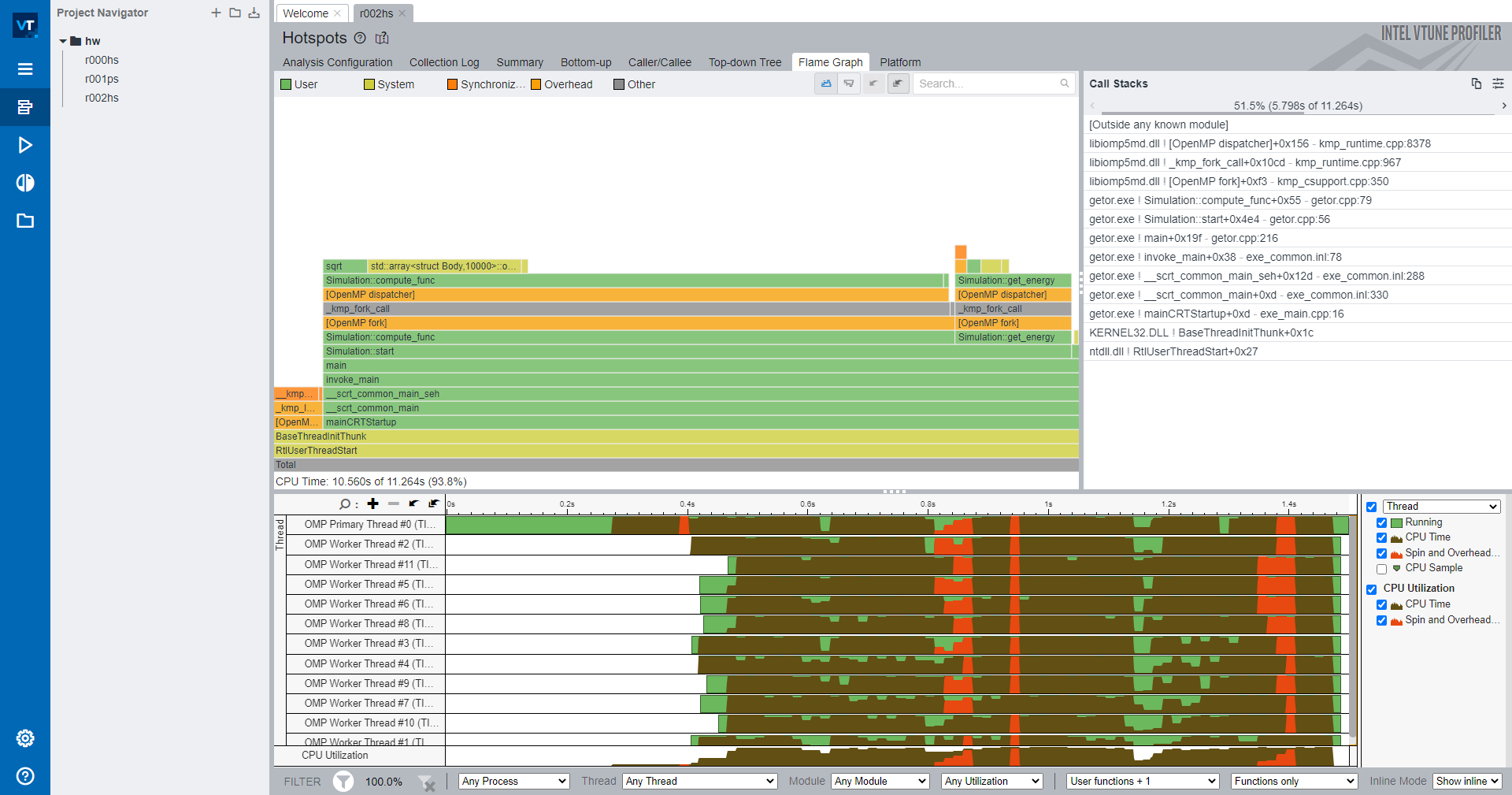
**Advisor survey**



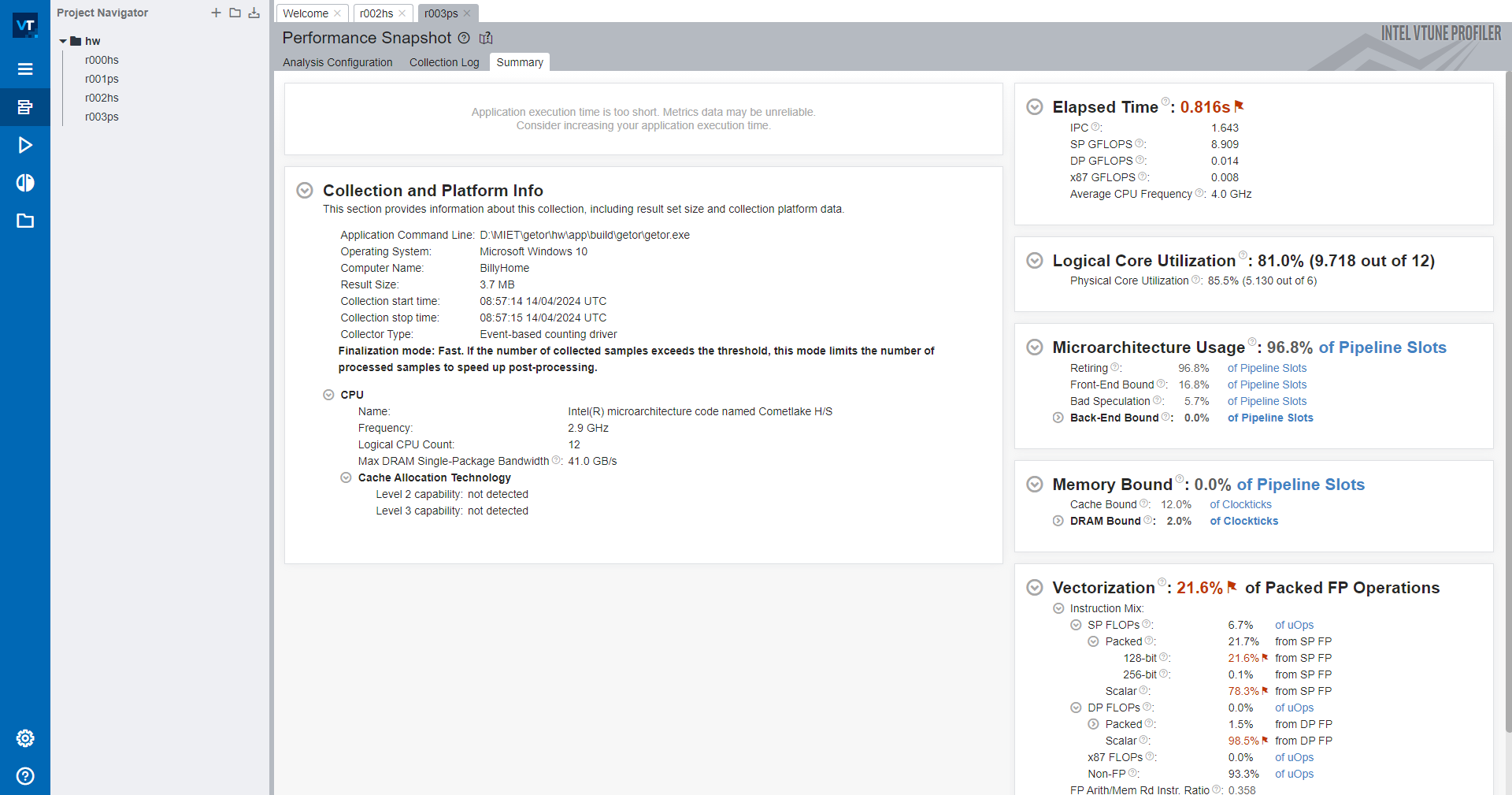
**V-tune summary**



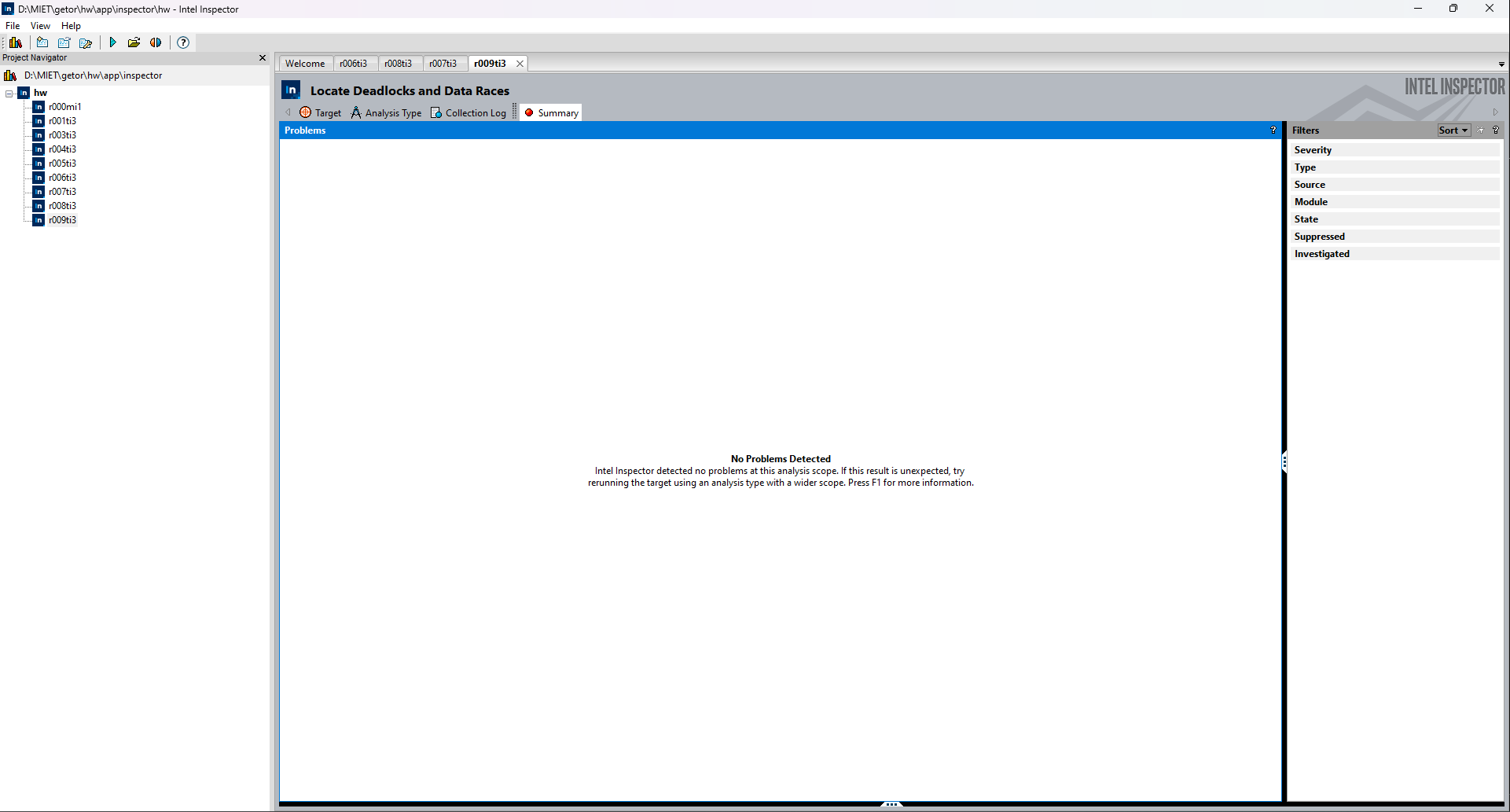
**V-tune hotspots**



**V-tune performance snap**



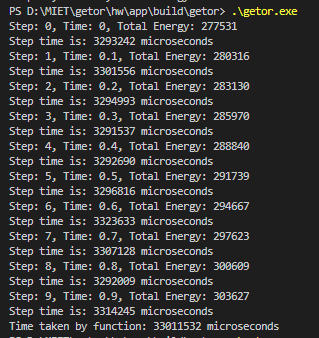
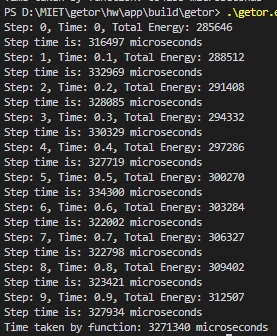
**Inspector**



**Сравнение с первоначальной реализацией**

N\_BODYES = 10000

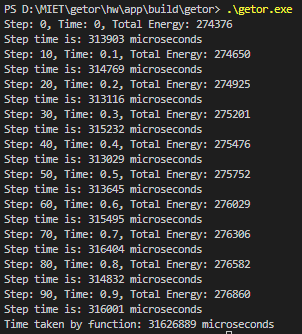
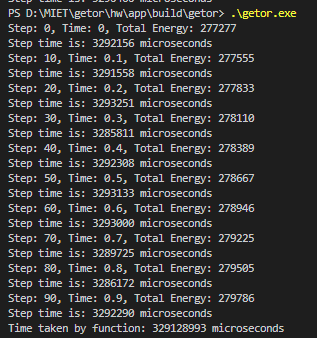
dt = 0.1 сек.

|  |  |  |  |
| --- | --- | --- | --- |
| Параметр | Предыдущее значение параметра | Значение параметра | Изменение |
| Время одной итерации | 3620002 мкс (3.6 сек) | 334300 мкс (0.3 сек) | +983% (x10.82) |
| Полное время моделирования | 35489915 мкс (35 сек) | 3271340 мкс (3.3 сек) | +985% (x10.85) |
| Накопленная ошибка | ~9.4% | ~9.4% | 0% |

N\_BODYES = 10000

dt = 0.01 сек.



|  |  |  |  |
| --- | --- | --- | --- |
| Параметр | Предыдущее значение параметра | Значение параметра | Изменение |
| Время одной итерации | 3323633 мкс (3.3 сек) | 316001 мкс (0.3 сек) | +952% (x10.52) |
| Полное время моделирования | 329128993 мкс (330 сек) | 31626889 мкс (31.6 сек) | +941% (x10.41) |
| Накопленная ошибка | ~0.9% | ~0.9% | 0% |