



НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ. Н.И. ЛОБАЧЕВСКОГО
ЦЕНТР КОМПЕТЕНЦИЙ ONEAPI В ННГУ

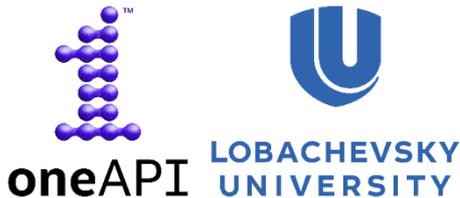
ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МАТЕМАТИКИ И МЕХАНИКИ



НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ. Н.И. ЛОБАЧЕВСКОГО
ЦЕНТР КОМПЕТЕНЦИЙ ONEAPI В ННГУ
ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МАТЕМАТИКИ И МЕХАНИКИ

***ВВЕДЕНИЕ В АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ
И ОПТИМИЗАЦИЮ ПРОГРАММ***

**Векторизация циклов.
Использование транслятора Intel
Compiler и помощника Intel Advisor**



Панова Е.А., Волокитин В.Д., Мееров И.Б.

Содержание

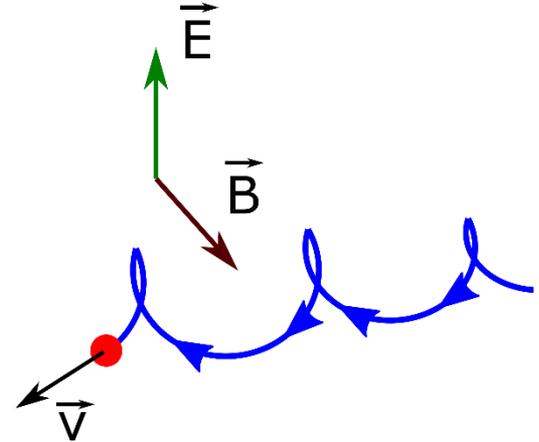
- План работы
- Задача и метод
- Описание кода
- Векторизация цикла
- Работа с Intel Advisor
- Другие оптимизации

План работы

- Существует простой код, который решает задачу движения заряженных частиц в электромагнитном поле. Предлагается ускорить время его выполнения.
 - Задача 1: проверить, векторизуется ли код, и если нет, то его векторизовать. Это должно ускорить время выполнения программы в несколько раз.
 - Задача 2: выяснить, с какой эффективностью векторизовался код и нельзя ли улучшить этот показатель. Для этого предлагается воспользоваться Intel Advisor.
 - Задача 3: попробовать оптимизировать данный код, в частности, на основе рекомендаций Intel Advisor.

Задача и метод

- Рассмотрим задачу движения заряженных частиц в электромагнитном поле.
- В несколько более сложной постановке задача актуальна, например, в физике плазмы.
- Входные данные:
 - Стартовый набор частиц определенного типа (например, пучок *электронов*); частица задается *положением в пространстве \vec{r} и скоростью \vec{v}* .
 - Электромагнитное поле:
 - *напряженность* электрического поля $\vec{E}(\vec{r}, t)$;
 - *индукция* магнитного поля $\vec{B}(\vec{r}, t)$.
 - В общем случае электромагнитное поле – функция координат и времени. Мы будем считать, что поле постоянное: $\vec{E}(\vec{r}, t) = \vec{E} = const, \vec{B}(\vec{r}, t) = \vec{B} = const$.



Задача и метод

- В электромагнитном поле на заряженную частицу действует *сила Лоренца*:

$$\vec{F} = q \left(\vec{E} + \frac{\vec{v}}{c} \times \vec{B} \right)$$

- Здесь q – заряд частицы, c – скорость света (СГС), \times – векторное произведение.
- Принимая во внимание второй закон Ньютона, получаем систему дифференциальных уравнений:

$$\frac{d\vec{r}}{dt} = \vec{v}, \quad \frac{d\vec{v}}{dt} = \frac{q}{m} \left(\vec{E} + \frac{\vec{v}}{c} \times \vec{B} \right)$$

- Решим систему численно классическим методом Рунге-Кутты 4 порядка.
- Замечание: число частиц может быть очень большим, поэтому на практике для решения системы используют специальные быстрые методы. Они, как правило, меньшего порядка, но при этом численное решение обладает хорошими свойствами.

Описание кода

- В базовой версии кода используются следующие структуры данных:

```
using FP = double; // тип числа с плавающей запятой, может быть float
```

```
struct Vector3 { // вектор в трехмерном пространстве  
    FP x, y, z; // для него перегружены основные арифметические операторы  
};
```

```
struct Particle { // частица (электрон)  
    Vector3 r, v;  
};
```

```
class ParticleEnsemble { // ансамбль (набор) частиц  
    std::vector<Particle> particles;  
};
```

- Набор частиц организован в виде *массива структур* (Array of Structures, SoA). Эта структура данных довольно естественна и удобна для программирования, но не всегда оптимальна с точки зрения производительности.

Описание кода

□ Код, производящий одну итерацию решателя:

```
void update(const FP& dt) { // dt - шаг численного интегрирования
    const FP coeffR = dt / (FP)6, coeffV = coeffR / ELECTRON_MASS; // предвычисленные коэффициенты
    for (int i = 0; i < particles.size(); i++) { // цикл по частицам
        Vector3 v = particles[i].v;

        // метод Рунге-Кутты для нахождения положения частицы
        Vector3 k1 = v;
        Vector3 k2 = v + (FP)0.5 * dt * k1;
        Vector3 k3 = v + (FP)0.5 * dt * k2;
        Vector3 k4 = v + dt * k3;
        particles[i].r += coeffR * (k1 + (FP)2 * k2 + (FP)2 * k3 + k4);

        // метод Рунге-Кутты для нахождения скорости частицы
        k1 = getLorentzForce(v); // сила Лоренца
        k2 = getLorentzForce(v + (FP)0.5 * dt * k1);
        k3 = getLorentzForce(v + (FP)0.5 * dt * k2);
        k4 = getLorentzForce(v + dt * k3);
        particles[i].v += coeffV * (k1 + (FP)2 * k2 + (FP)2 * k3 + k4);
    }
}

Vector3 getLorentzForce(const Vector3& v) { return ELECTRON_CHARGE*(E + cross(v, B)/LIGHT_VELOCITY); }
```

Векторизация цикла

1. Скомпилируйте код с помощью Intel Compiler.
 - Ключ **-O2 (/O2)** включает оптимизацию (по умолчанию).
 - Убедитесь, что задействованы подходящие векторные инструкции (ключ **-xHost** или **/QxHost**). Если на машине поддерживаются инструкции AVX512, то рекомендуется дополнительно установить **-qopt-zmm-usage=high** или **/Qopt-zmm-usage=high**.
 - Ключ **-qopt-report=[n] (/Qopt-report[:n], n=1..5)** дает команду сгенерировать отчет компилятора.
2. Запустите программу, оцените время работы.
3. Векторизован ли цикл? Это можно понять из отчета компилятора. Если цикл скалярный, то определите причины, по которым компилятор не стал создавать векторный код. Исправьте код так, чтобы были задействованы векторные расширения.
4. Запустите две версии: скалярную и векторную. Сравните время работы.

Векторизация цикла

- ❑ По каким причинам код может быть скалярным:
 - Компилятор подозревает зависимости по данным (**#pragma ivdep** или **#pragma omp simd**).
 - В теле цикла есть вызовы функций:
 - какие-то из вызываемых функций не inline;
 - тело цикла слишком большое и вынесено компилятором в отдельную функцию.
 - Компилятор решил, что векторизация цикла для данной архитектуры неэффективна (**#pragma vector always** или **#pragma omp simd**).
 - ...
- ❑ Чтобы отключить векторизацию (например, для замеров времени скалярной версии), можно использовать **#pragma novector**.

Работа с Intel Advisor

1. Скомпилируйте программу в режиме Release с отладочной информацией (ключ **-g** для Linux или **/debug** для Windows).
2. Запустите помощник Intel Advisor. Соберите профиль.
 - Пример запуска из командной строки:
`advisor --collect=survey --project-dir=<директория> -- <имя программы>`
 - Intel Advisor GUI для сборки профиля и/или просмотра результатов.
3. Проанализируйте результаты:
 - Какие векторные инструкции были задействованы (SSE, AVX, AVX2, ...)? Векторные регистры какой длины используются? Поддерживается ли текущей архитектурой более новый набор векторных инструкций?
 - Какова эффективность векторизации?
 - Дает ли Intel Advisor рекомендации для улучшения эффективности векторизации? Если да, то какие?

Возможные оптимизации

- ❑ Предлагается в первую очередь изменить структуру данных с *Array of Structures (AoS)* на *Structure of Arrays (SoA)*, которая больше подходит для векторных вычислений.
- ❑ Такие простые оптимизации, как замена деления на умножение или вынос инвариантов, могут дать прирост производительности:

```
Vector3 getLorentzForce(const Vector3& v) { return ELECTRON_CHARGE*(E + cross(v, B)*INV_LIGHT_VELOCITY); }
```

- ❑ Иногда компилятор справляется недостаточно эффективно с оптимизацией, если код написан достаточно сложно (ООП, последние стандарты C++ и т.д.). Можно попробовать переписать код в стиле языка C:

```
FP *vxArr, *vyArr, *vzArr, ...; // указатели на первый элемент соответствующих массивов
const int nParticles;          // число частиц

#pragma omp simd
for (int i = 0; i < nParticles; i++) {
    FP k1x = ELECTRON_CHARGE*(E.x + (vyArr[i]*B.z - vzArr[i]*B.y) * INV_LIGHT_VELOCITY);
    // FP k1y = ..., k1z = ...;
    // ...
}
```

Возможные оптимизации

- ❑ Подразбиение итерационного пространства на группы делает код cache-friendly.
- ❑ Если тело цикла получается слишком большим, то Intel Advisor может предложить разбить цикл на несколько независимых циклов. Это можно сделать вручную или с помощью **#pragma distribute_point**.

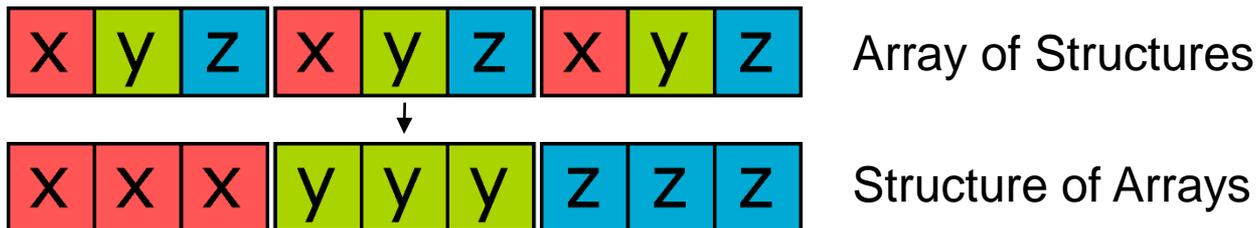
```
const int groupSize = 64; // оптимальным может быть другое значение, например, 16 или 32
const int nGroups = nParticles / groupSize; // остаток обрабатывается отдельно

for (int iGroup = 0; iGroup < nGroups; iGroup++) { // внешний цикл по группам
    int indexShift = iGroup * groupSize;
    FP *vxPtr = &(vxArr[indexShift]), ...; // указатели на начало элементов текущей группы
    FP k1x[groupSize], ...; // массивы, хранящие значения локальных переменных в рамках одной группы

#pragma omp simd
#pragma distribute_point
    for (int i = 0; i < groupSize; i++) { // цикл по элементам группы
        k1x[i] = ELECTRON_CHARGE*(E.x + (vyPtr[i]*B.z - vzPtr[i]*B.y)*INV_LIGHT_VELOCITY);
        // ...
    }
}
```

SIMD Data Layout Templates (SDLT)

- ❑ Почему структура массивов (SoA) больше подходит для векторных вычислений, чем массив структур (AoS)?



- ❑ Для того, чтобы перейти от массива структур к структуре массивов с минимальными изменениями в коде, можно использовать библиотеку SIMD Data Layout Templates (SDLT), предоставляемую с Intel Compiler.
- ❑ Когда имеет смысл использовать SDLT:
 - Если уже имеется какой-то старый сложный код, который работает медленно и требует оптимизации. Новый код лучше сразу проектировать правильно 😊
 - Стоит помнить о кроссплатформенности.

SIMD Data Layout Templates (SDLT)

- ❑ С SDLT наш код будет выглядеть так:

```
struct Vector3 { // вектор в трехмерном пространстве
    FP x, y, z;
};
SDLT_PRIMITIVE(Vector3, x, y, z) // предоставляем SDLT описание структуры
```

```
struct Particle { // частица
    Vector3 r, v;
};
SDLT_PRIMITIVE(Particle, r, v) // Vector3 уже описан в SDLT, так делать можно
```

```
sdl::soa1d_container<Particle> particles; // SoA-контейнер для частиц
```

- ❑ Доступ к элементам структуры осуществляется так:

```
auto accessor = particles.access();
Vector3 v = accessor[i].v();
FP rx = accessor[i].r().x();
```

- ❑ Таким образом, удобство не теряется, но эффективность возрастает.

Пример результатов оптимизации

Версия \ Время (сек) 2 [^] 20 = 1048576 частиц 256 итераций метода	Intel Core i7-11800H (Tiger Lake, AVX512), Windows 10, Intel oneAPI C++ Compiler Classic 2021.6		Intel Xeon Gold 6248R (Cascade Lake, AVX512), CentOS Linux 7, Intel oneAPI C++ Compiler Classic 2021.1*		Комментарий
	scalar	vector	scalar	vector	
Базовая, AoS	<u>7,452</u> →	<u>1,236</u> ↓	<u>10,050</u> →	<u>2,708</u> ↓	Переход от скалярного к векторному коду дал прирост в несколько раз
Базовая + SDLT	7,405	0,936	10,162	2,326	SoA лучше подходит для SIMD
SoA, ООП	7,435 ↓	0,928 ↓	10,070 ↓	2,278 ↓	
SoA, ООП, замена деления на умножение	5,975 ↓	<u>0,918</u> ↓	8,100 ↓	<u>1,205</u> ↓	Простейшая оптимизация в этом случае дала большой прирост производительности
SoA, C-style (без ООП)	6,868 ↓	1,010	8,770 ↓	1,470	Переписывание кода на язык C не дало положительного эффекта
SoA, C-style, cache- friendly	<u>5,377</u>	0,970	<u>7,124</u>	1,558	Однако cache-friendly код на языке C дал существенное ускорение для скалярной версии

*Узел суперкомпьютера МВС 10П МСЦ РАН

Вопросы для обсуждения

1. Почему AoS может быть немного быстрее для скалярного кода, чем SoA?
2. Почему при переходе к C-style коду мы получили замедление?
3. Почему cache-friendly вариант дает прирост только для скалярного кода?
4. Получили бы мы прирост производительности, если бы использовали вариант с делением на группы для C++ кода с ООП?

Литература

1. Birdsall, Charles K., and A. Bruce Langdon. *Plasma physics via computer simulation*. CRC press, 2018.
2. Sabahi, M. "A Guide to Vectorization with Intel C++ Compilers." (2010).

Авторский коллектив

- ❑ Мееров Иосиф Борисович, к.т.н., доцент, зам. зав. каф. МОСТ
- ❑ Сысоев Александр Владимирович, к.т.н., доцент каф. МОСТ
- ❑ Линев Алексей Владимирович, зав. лаб. интернета вещей, каф. ПРИН
- ❑ Волокитин Валентин Дмитриевич, программист лаборатории СТиВВ, каф. МОСТ
- ❑ Козинов Евгений Александрович, к.т.н., преподаватель каф. МОСТ
- ❑ Панова Елена Анатольевна, инженер лаборатории СТиВВ, каф. МОСТ

Контакты

Нижегородский государственный университет

<http://www.unn.ru>

Центр компетенций oneAPI в ННГУ

<http://hpc-education.unn.ru/ru/центр-компетенций-oneapi-в-ннгу>

Институт информационных технологий, математики и механики

<http://www.itmm.unn.ru>