Векторизация нахождения пересечения объемной и поверхностной сеток для микропроцессоров с поддержкой AVX-512

А.А. Рыбаков

МСЦ РАН – филиал ФГУ ФНЦ НИИСИ РАН, Москва, Россия, rybakov@jscc.ru

Аннотация. Векторизация программного кода является одной из наиболее важных низкоуровневых оптимизаций, позволяющих существенно повысить производительность суперкомпьютерных приложений. Эффект от векторизации кода тем заметнее, чем больше длина вектора. Набор инструкций AVX-512, поддержанный в современных микропроцессорах Intel, предназначен для работы с 512-битными векторами и позволяет проводить векторизацию предикатного кода. В данной работе рассмотрен практический подход к векторизации сложного программного контекста, реализующего нахождение пересечения объемной декартовой и поверхностной неструктурированной расчетных сеток. Эта задача используется в расчетах обтекания тел со сложной геометрией, выполняемых с помощью метода погруженной границы, при этом в качестве объемной декартовой сетки может быть использована как равномерная сетка с одинаковыми ячейками, так и локально-измельчающаяся сетка. Приведено математическое решение поставленной задачи, выполнена ее реализация, а также показано, что векторизация с помощью набора инструкций AVX-512 позволяет достичь ускорения на микропроцессоре Intel Xeon Phi KNL более чем в 6,5 раз по сравнению с невекторизованной версией на вещественных данных одинарной точности.

Ключевые слова. Векторизация, Intel Xeon Phi KNL, AVX-512, объемная декартова сетка, поверхностная неструктурированная сетка, плоский цикл, функции-интринсики.

Введение

При численном решении задач газовой динамики часто приходится сталкиваться с телами, обладающими сложной геометрией. Для таких тел построение согласованной расчетной сетки может быть крайне трудозатратной задачей. Альтернативой в данном случае является использование метода погруженной границы [1]. Данный метод позволяет использовать для расчетов несогласованную сетку и даже простую декартову сетку, что сильно упрощает выполнение расчетов. Единственным тонким моментом метода является выполнение граничных условий на сложной границе, которое достигается путем модификации решаемой системы уравнений. Можно выделить два подхода в методах погруженной границы, различающихся по способу выполнения расчетов на границе: задание граничных условий посредством внешних (или источниковых) членов [2] и методы фиктивных ячеек [3].

Рассмотрим немного подробнее метод погруженной границы с использованием фиктивных ячеек. Пусть в некоторой области пространства расположено тело, ограниченное сложной границей, данная граница, представлена неструктурированной поверхностной расчетной сеткой, ячейки которой являются треугольниками. Пусть также в охватывающей тело области пространства построена объемная расчетная сетка, ячейки которой могут быть разделены на следующие три основные класса. Внешними ячейками будем называть те ячейки, которые целиком лежат вне тела. Внутренние ячейки лежат целиком внутри тела, все остальные ячейки пересекают границу тела и являются граничными. В методе фиктивных ячеек из граничных ячеек выделяются ячейки, для которых меньшая часть находится вне тела, а большая – внутри тела. Такие ячейки называются фиктивными. На каждой итерации расчетов для фиктивных ячеек требуется выполнить аппроксимацию газодинамических величин (плотность, давление, вектор скорости), чтобы данные фиктивные ячейки могли быть использованы для определения потоков между ними и соседними с ними граничными и внешними ячейками [4]. Таким образом, классификация ячеек объемной сетки является неотъемлемой частью метода погруженной границы. В условиях изменяющейся геометрии обтекаемого тела задача классификации ячеек занимает существенную долю вычислений и должна быть оптимизирована.

Для получения качественного решения вблизи тела со сложной геометрией требуется использовать мелкие объемные

сетки, а вблизи границы тела выполнять также локальное измельчение граничных ячеек. На рис. 1 продемонстрированы варианты построения множества граничных и фиктивных ячеек для тела со сложной геометрией, в которых использованы объемные сетки с разным количеством ячеек. При использовании мелких объемных сеток количество граничных ячеек сильно возрастает и задача их классификации становится критичной по времени выполнения. Основным этапом задачи классификации ячеек объемной сетки является задача об определении пересечения множества ячеек объемной сетки с множеством ячеек поверхностной сетки. В данной работе в качестве объемной сетки рассматривается декартова сетка, ребра ячеек которой параллельны осям координат.

Для решения задачи классификации ячеек объемной сетки требуется решить частную задачу определения пересечения треугольника и прямоугольного параллелепипеда в пространстве.

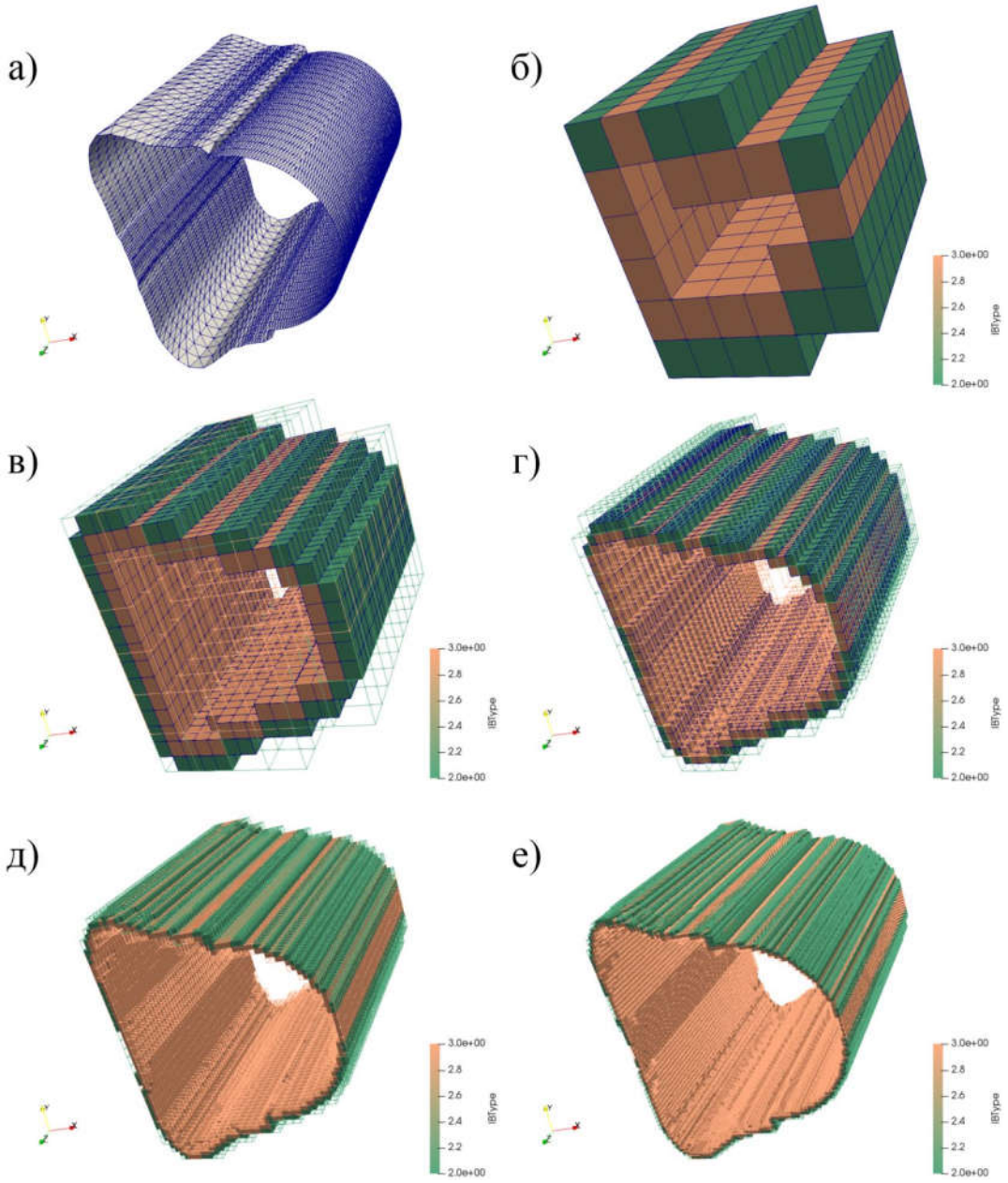

Рисунок 1. Иллюстрация пересечения поверхностной сетки и объемных сеток с разным количеством ячеек. На рисунках цветами обозначены только граничные и фиктивные ячейки объемной сетки (граничные – зеленые, фиктивные – бежевые). а) Исходная геометрия поверхностной сетки. б) Размер объемной сетки 10×10×10. в) Размер объемной сетки 25×25×25. г) Размер объемной сетки 50×50×50. д) Размер объемной сетки 100×100×100. е) Размер объемной сетки 150×150×150.

Задача о пересечении треугольника и прямоугольного параллелепипеда

В данном разделе рассмотрим математическую постановку и метод решения задачи об определении пересечения треугольника и прямоугольного параллелепипеда в пространстве.

Пусть треугольник задан тремя точками: $A(x_A, y_A, z_A)$, $B(x_B, y_B, z_B)$, $C(x_C, y_C, z_C)$. Тогда координаты любой точки P(x, y, z), находящейся внутри треугольника, можно представить следующим образом:

$$\begin{cases} x = x_A + (x_B - x_A)\alpha + (x_C - x_A)\beta \\ y = y_A + (y_B - y_A)\alpha + (y_C - y_A)\beta \\ z = z_A + (z_B - z_A)\alpha + (z_C - z_A)\beta \end{cases} \quad (1)$$

где $\alpha \geq 0$, $\beta \geq 0$, $\alpha + \beta \leq 1$.

Геометрическим местом точек прямоугольного параллелепипеда является множество точек P(x, y, z), координаты которых удовлетворяют следующей системе неравенств:

$$\begin{cases} x_l \leq x \leq x_h \\ y_l \leq y \leq y_h \\ z_l \leq z \leq z_h \end{cases} \quad (2)$$

Для установления факта пересечения треугольника и прямоугольного параллелепипеда нужно определить, имеет ли решение приведенная ниже система неравенств относительно α и β:

$$\begin{cases} x_l \leq x_A + (x_B - x_A)\alpha + (x_C - x_A)\beta \leq x_h \\ y_l \leq y_A + (y_B - y_A)\alpha + (y_C - y_A)\beta \leq y_h \\ z_l \leq z_A + (z_B - z_A)\alpha + (z_C - z_A)\beta \leq z_h \\ \alpha \geq 0 \\ \beta \geq 0 \\ \alpha + \beta \leq 1 \end{cases} \quad (3)$$

Для подобных систем неравенств существует много различных способов решения, описание которых можно найти, например, в [5]. В нашем случае система довольно простая, содержит две переменные, и к ней может быть применен метод свертывания конечных систем линейных неравенств, описанный в [6].

Для решения методом свертывания преобразуем систему неравенств так, чтобы она содержала неравенства только вида $k_\alpha\alpha + k_\beta\beta + k \leq 0$. После выполнения всех преобразований система неравенств примет следующий вид:

$$\begin{cases} (x_B - x_A)\alpha + (x_C - x_A)\beta + (x_A - x_h) \leq 0 \\ (x_A - x_B)\alpha + (x_A - x_C)\beta + (x_l - x_A) \leq 0 \\ (y_B - y_A)\alpha + (y_C - y_A)\beta + (y_A - y_h) \leq 0 \\ (y_A - y_B)\alpha + (y_A - y_C)\beta + (y_l - y_A) \leq 0 \\ (z_B - z_A)\alpha + (z_C - z_A)\beta + (z_A - z_h) \leq 0 \\ (z_A - z_B)\alpha + (z_A - z_C)\beta + (z_l - z_A) \leq 0 \\ -1\cdot\alpha + 0\cdot\beta \leq 0 \\ 0\cdot\alpha + (-1)\beta \leq 0 \\ \alpha + \beta + (-1) \leq 0 \end{cases} \quad (4)$$

Так как система содержит всего две переменные, то после выполнения одного шага свертывания (или деформации системы) она превратится в систему неравенств относительно одной переменной, проверка разрешимости которой не представляет труда. Будем выполнять деформацию системы с целью исключить из нее переменную α. Для этого составим новую систему, в которую войдут все неравенства исходной системы вида $k_\beta\beta + k \leq 0$, а каждая пара неравенств

$$\begin{aligned} k^1_\alpha\alpha + k^1_\beta\beta + k^1 \leq 0 \\ k^2_\alpha\alpha + k^2_\beta\beta + k^2 \leq 0 \end{aligned} \quad (5)$$

где $k^1_\alpha < 0$, а $k^2_\alpha > 0$ войдет в деформированную систему в виде

$$(k^1_\beta k^2_\alpha - k^2_\beta k^1_\alpha)\beta + (k^1 k^2_\alpha - k^2 k^1_\alpha) \leq 0 \quad (6)$$

Так как исходная система содержит 9 уравнений, по крайней мере одно из которых имеет нулевой коэффициент при переменной α, а из оставшихся восьми половина коэффициентов при переменной α неотрицательно, а половина неположительны, то деформированная система будет содержать не более 17 уравнений.

Оптимизация поиска пересечения сеток

Ячейки объемной сетки являются прямоугольными параллелепипедами. Ячейки поверхностной сетки являются треугольниками. Для того, чтобы найти все ячейки объемной сетки, которые пересекаются с поверхностью, в общем

случае необходимо проверить факт пересечения каждой объемной ячейки с каждой поверхностной ячейкой. В общем случае данная процедура является очень затратной, так как сетки могут содержать большое количество ячеек. Вместо этого для каждой ячейки поверхностной сетки сначала будем определять диапазон ячеек объемной сетки, с которыми в принципе возможно пересечение.

Пусть изначально объемная сетка, описывающая область $[X_l, X_h]\times[Y_l, Y_h]\times[Z_l, Z_h]$, размера $S_x\times S_y\times S_z$, разделена на одинаковые базовые ячейки (которые впоследствии могут быть разделены на более мелкие) размера $s_x\times s_y\times s_z$, где

$$n_x = S_x / s_x$$
$$n_y = S_y / s_y \qquad (7)$$
$$n_z = S_z / s_z$$

(n_x – количество базовых ячеек по направлению x, аналогично для направлений y и z). Таким образом, объемная сетка представлена трехмерным массивом базовых ячеек, координаты которых (x_l, x_h, y_l, y_h, z_l, z_h) могут быть вычислены по индексам (i, j, k) в данном трехмерном массиве:

$$\begin{aligned} x_l(i) &= X_l + is_x \\ x_h(i) &= X_h + (i+1)s_x \\ y_l(j) &= Y_l + js_y \\ y_h(j) &= Y_h + (j+1)s_y \\ z_l(k) &= Z_l + ks_z \\ z_h(k) &= Z_h + (k+1)s_z \end{aligned} \qquad (8)$$

Для треугольника, являющегося ячейкой поверхностной сетки, вершинами которого являются точки $A(x_A, y_A, z_A)$, $B(x_B, y_B, z_B)$, $C(x_C, y_C, z_C)$, можно найти охватывающий прямоугольный параллелепипед, координаты которого равны

$$\begin{aligned} \tilde{x}_l &= \min(x_A, x_B, x_C) \\ \tilde{x}_h &= \max(x_A, x_B, x_C) \\ \tilde{y}_l &= \min(y_A, y_B, y_C) \\ \tilde{y}_h &= \max(y_A, y_B, y_C) \\ \tilde{z}_l &= \min(z_A, z_B, z_C) \\ \tilde{z}_h &= \max(z_A, z_B, z_C) \end{aligned} \qquad (9)$$

Если треугольник пересекает некоторую объемную ячейку, то его охватывающий прямоугольный параллелепипед также пересекает эту ячейку. То есть для определения пересечения поверхностной ячейки со всеми ячейками объемной сетки достаточно проверить только те ячейки, с которыми пересекается охватывающий прямоугольный параллелепипед рассматриваемого треугольника. Так как координаты анализируемых параллелепипедов могут быть записаны явно, то можно вычислить диапазоны индексов базовых ячеек, которые требуется проверить на вопрос пересечения с треугольником. Факт пересечения по координате x двух отрезков $[x_l(i), x_h(i)]$ и $[\tilde{x}_l, \tilde{x}_h]$ можно записать в виде системы из двух неравенств

$$\begin{cases} x_l(i) \le \tilde{x}_h \\ x_h(i) \ge \tilde{x}_l \end{cases} \qquad (10)$$

Преобразовав данную систему, а также выполнив аналогичные действия для координат по y и z, получим итоговые диапазоны индексов базовых ячеек

$$\begin{cases} \dfrac{\tilde{x}_l - X_l}{s_x} - 1 \le i \le \dfrac{\tilde{x}_h - X_l}{s_x} \\ \dfrac{\tilde{y}_l - Y_l}{s_y} - 1 \le j \le \dfrac{\tilde{y}_h - Y_l}{s_y} \\ \dfrac{\tilde{z}_l - Z_l}{s_z} - 1 \le k \le \dfrac{\tilde{z}_h - Z_l}{s_z} \end{cases} \qquad (11)$$

Данный диапазон индексов содержит малую долю всех ячеек объемной сетки. В данном случае не требуется проводить анализ ускорения от этого преобразования, так как при его отсутствии поиск пересечения сеток просто нежизнеспособен.

Заметим, что при измельчении объемной сетки алгоритм поиска пересечения изменяется не сильно, так как диапазоны индексов базовых ячеек не меняются, однако кроме базовых ячеек обязательными для проверки становятся также все их дочерние ячейки.

Набор инструкций AVX-512

Современные микропроцессоры Intel начиная с Intel Xeon Phi KNL [7] и Intel Xeon Skylake поддерживают набор инструкций AVX-512 – 512-битное расширение AVX-инструкций. Однако основное отличие набора инструкций AVX-512 от AVX не в длине вектора, благодаря которому пиковая

производительность микропроцессора возрастает вдвое. Набор инструкций AVX-512 поддерживает выборочное применение операций к элементам векторов, которое обеспечивается с помощью специальных масок. Маски являются объединением битов, каждый из которых управляет записью результатов векторных операций в регистр назначения. Благодаря использованию масочных операций стало возможным векторизовать широкий спектр программного кода, записанного в предикатном виде. Предикатный режим исполнения аппаратно поддержан, например, в архитектуре «Эльбрус» [8], и с его помощью возможно проведение множества оптимизаций, позволяющих генерировать параллельный на уровне отдельных инструкций код. Помимо наличия предикатных команд набор инструкций AVX-512 обладает другими особенностями, позволяющими создавать эффективный параллельный код. Это операции множественного обращения в память с произвольными смещениями от базового адреса, комбинированные операции (позволяющие за одну операцию производить поэлементное вычисление выражений вида ±a·b±c), огромное разнообразие операций перестановок элементов векторов (perm, shuf и другие), реализация поэлементных логических функций от трех переменных и многие другие.

Для упрощения использования инструкций из набора AVX-512 для компилятора icc реализована библиотека функций-интринсиков [9, 10], которые определены в заголовочном файле immintrin.h. Данные функции являются обертками над инструкциями AVX-512, оперируют встроенными типами векторных данных (__m512, __m512i, __m512d и другими), и могут быть использованы в программах, написанных на языке C. Таким образом, предоставлена возможность напрямую использовать инструкции AVX-512 в коде, не прибегая к ассемблерным вставкам.

Появление набора инструкций AVX-512 вызвало большой интерес со стороны исследователей и разработчиков программного обеспечения, в настоящее время ведутся работы по их использованию для оптимизации приложений из разных научных областей [11]. Можно отметить, например, работы по векторизации ядра программного кода LAMMPS [12], операций с разреженными матрицами [13] и матрицами специального вида [14]. В работе [15] описан подход векторизации гнезда циклов на примере построения множества Мандельброта. Можно встретить работы, в которых описывается применение векторизации с использованием набора инструкций AVX-512 для других приложений из широкого круга, начиная от задач ядерной физики [16], и численного решения уравнений мелкой воды [17,18] и заканчивая задачами дискретной математики, включая векторизацию задач сортировки [19] или генератора псевдослучайных чисел [20].

В реальных приложениях компилятор зачастую не способен определить пригодный для векторизации контекст или отказывается от векторизации ввиду низкой ожидаемой эффективности. Причиной тому может служить невозможность доказать отсутствие зависимостей в коде, обилие условий и команд передачи управления, использование невыровненных данных. Однако небольшое участие программиста может привести к векторизации сложного контекста, используя предикатное представление кода и замену скалярных операций векторными. В следующем разделе описано применение данного подхода для векторизации задачи пересечения расчетных сеток.

Векторизация вычислений

Рассмотрим реализацию функции tri_box_intersect(xa ya, za, xb, yb, zb, xc, yc, zc, xl, xh, yl, yh, zl, zh) → int, анализирующую наличие пересечения треугольника и прямоугольного параллелепипеда. Функция возвращает 1, если пересечение есть, и 0, если пересечения нет. Логика работы функции следующая. Сначала коэффициенты системы неравенств (4) заносятся в двумерный массив коэффициентов b[bec][3], где bec (basic equations count) – количество исходных неравенств системы (в нашем случае 9). Затем выполняется один шаг свертывания системы с одновременным поиском множества решения для переменной β. Перед началом свертывания множество допустимых значений для переменной β принимается в виде отрезка [0, 1] (lo = 0, hi = 1). По мере свертывания системы неравенств (4) происходит сокращение данного множеств решений. Если на каком-то этапе свертывания множество решений обращается в пустое (lo > hi), то функция

заканчивает работу и возвращает 0. Если после выполнения всех действий свертывания множество решений осталось ненулевым, то это означает наличие пересечения, и функция возвращает 1. Реализация свертывания системы уравнений с помощью метода из [6] представлена на рис. 2.

```
01 for (i = 0; i < bec; i++)
02 {
03     bi0 = b[i][0];
04
05     if (bi0 == 0.0)
06     {
07         if (!upgrade(b[i][1], b[i][2], &lo, &hi))
08         {
09             return 0;
10         }
11     }
12     else
13     {
14         for (j = i + 1; j < bec; j++)
15         {
16             if (bi0 * b[j][0] < 0.0)
17             {
18                 f0 = bi0 * b[j][1] - b[j][0] * b[i][1];
19                 f1 = bi0 * b[j][2] - b[j][0] * b[i][2];
20
21                 if (bi0 < 0.0)
22                 {
23                     f0 = -f0;
24                     f1 = -f1;
25                 }
26
27                 if (!upgrade(f0, f1, &lo, &hi))
28                 {
29                     return 0;
30                 }
31             }
32         }
33     }
34 }
35
36 return 1;
```

Рисунок 2. Исходная реализация свертывания системы линейных неравенств для определения пересечения треугольника и прямоугольного параллелепипеда.

Получившийся код можно охарактеризовать как имеющий сложное управление, уровень вложенности конструкций в нем достигает 5 (for-if-for-if-if), к тому же участок содержит 3 выхода из функции.

Функция upgrade, которая вызывается из кода на рис. 2, предназначена для обновления текущего множества допустимых значений для переменной β с учетом нового полученного ограничения вида $k_\beta \beta + k \le 0$, коэффициенты которого передаются в первом и втором параметрах.

Текущее множество решений является отрезком с границами, хранящимися в переменных lo и hi, и в зависимости от знака коэффициента k_β одна из этих границ внутри вызова функции upgrade может измениться (граница lo может увеличиться, либо граница hi может уменьшиться).

Если после обновления множества решений оно оказывается пустым (нижняя граница становится больше верхней), то функция upgrade возвращает 0, в противном случае она возвращает 1.

После рассмотрения реализации функции tri_box_intersect можно перейти к векторизации вычислений. Для анализа пересечения двух сеток необходимо вызывать функцию tri_box_intersect многократно с разными наборами входных параметров. Для оптимизации этого процесса реализуем функцию tri_box_intersect_16, объединяющую внутри себя обработку 16 вызовов функции tri_box_intersect (используется объединение 16 вызовов, так как это совпадает с количеством элементов вещественных данных одинарной точности в одном векторе __m512, VEC_WIDTH = 16). В первом приближении реализация функции tri_box_intersect_16 представлена на рис. 3. Конечно в реальной задаче требуется обрабатывать миллионы вызовов, однако они могут быть разбиты на группы по 16 и обработаны с помощью функции tri_box_intersect_16, поэтому остановимся подробнее на векторизации данной функции.

```
01 void tri_box_intersect_16(float *xa, float *ya, float *za,
02                           float *xb, float *yb, float *zb,
03                           float *xc, float *yc, float *zc,
04                           float *xl, float *xh,
05                           float *yl, float *yh,
06                           float *zl, float *zh,
07                           int *r)
08 {
09     for (int i = 0; i < 16; i++)
10     {
11         r[i] = tri_box_intersect(xa[i], ya[i], za[i],
12                                  xb[i], yb[i], zb[i],
13                                  xc[i], yc[i], zc[i],
14                                  xl[i], xh[i], yl[i], yh[i], zl[i], zh[i]);
15     }
16 }
```

Рисунок 3. Исходная реализация функции, объединяющей 16 вызовов функции tri_box_intersect.

Для векторизации функции tri_box_intersect_16 следует выполнить подстановку тела функции tri_box_intersect в место ее вызова. После выполнения данного преобразования получаем программный контекст, содержащий сложное управление и в частности гнездо из трех вложенных циклов, не считая условий. Данный контекст не может быть векторизован компилятором автоматически, поэтому будем проводить его векторизацию в ручном режиме с использованием функций-интринсиков, которые позволяют напрямую использовать инструкции AVX-512 в синтаксисе языка программирования C без использования ассемблера.

Заметим, что между итерациями внешнего цикла (рис. 3, строка 09) отсутствуют зависимости, то есть цикл является плоским, и его итерации могут быть выполнены в любом порядке, в том числе и одновременно. Такие циклы поддаются векторизации путем перевода тела в предикатное представление и заменой скалярных инструкций на векторные. Наиболее тонким местом при векторизации тела плоского цикла являются условия, то есть наличие конструкций if-else [21]. Альтернативные ветви таких конструкций должны быть объединены в предикатном коде под противоположными предикатами. Наличие большого количества условных операторов в исходном коде порождает множество инструкций под нулевыми предикатами в результирующем коде, что негативно сказывается на производительности. Для уменьшения количества условных операторов, можно использовать математические тождества, заменяющие условные конструкции на команды, имеющие векторные аналоги в наборе инструкций AVX-512. Для данных целей хорошо подходят такие векторные команды как abs, min, max, blend, avg и другие. Например в рассматриваемом коде вычисление значений f0 и f1 в строках 18-25 на рис. 2 с учетом условия b[i][0] * b[j][0] < 0 может быть заменено на следующее:

```
f0 = fabs(bi0) * b[j][1]
     + fabs(b[j][0]) * b[i][1]
f1 = fabs(bi0) * b[j][2]
     + fabs(b[j][0]) * b[i][2]
```

Похожие трудности вызывает условие if в строке 05 на рис 2. Данное условие имеет альтернативную ветку, содержащую цикл. Слияние данных двух ветвей снижает производительность результирующего кода, поэтому в данном случае выгодно применить расщепление внешнего цикла по конструкции if-else (в [22] данное преобразование фигурирует под названием loop distribution). При этом образуются два гнезда циклов, каждое из которых может быть векторизовано независимо. Заметим, что описанное преобразование в общем смысле не является эквивалентным, так как обе ветки условия if-else, а значит и тела образовавшихся циклов содержат выходы из функции, выполнение расщепления цикла может изменить условие, провоцирующее выход из функции. По этой причине компилятор не способен выполнить данное преобразование автоматически. Однако с точки зрения результата функции данное преобразование корректно, поэтому мы его и применяем.

Отметим еще один крайне положительный момент в рассматриваемом программном контексте. Условием выхода из вложенных циклов является достижение индуктивной переменной значения bec. Данное значение является константой, а значит не зависит от номера итерации, поэтому в векторный код это условие может быть перенесено без изменений. В случае зависимости условия выхода из цикла от

номера итерации само условие должно быть также векторизовано, что может привести к потерям производительности, причины которых описаны в [23] на примере векторизации сортировки Шелла. Однако в рассматриваемом коде проблем векторизации циклов с нерегулярным количеством итераций нет. На рис. 4 приведен получившийся предикатный код для функции tri_box_intersect_16, а также схематично показана его транформации в векторный аналог.

последовательный код

```
float b[bec][3][VEC_WIDTH];
<инициализация b>;

for (w = 0; w < VEC_WIDTH; w++) r[w] = 1;

for (w = 0; w < VEC_WIDTH; w++)
{
    lo = 0.0;
    hi = 1.0;

    for (i = 0; i < bec; i++)
    {
        upgrade(b[i][0][w] == 0.0,
                b[i][1][w], b[i][2][w], &lo, &hi);
        if (lo > hi) break;
    }

    for (i = 0; i < bec; i++)
    {
        bi0 = b[i][0][w];
        abi0 = fabs(bi0);

        for (j = i + 1; j < bec; j++)
        {
            bj0 = b[j][0][w];
            abj0 = fabs(bj0);

            upgrade(bi0 * bj0 < 0.0,
                    abi0 * b[j][1][w] + abj0 * b[i][1][w],
                    abi0 * b[j][2][w] + abj0 * b[i][2][w],
                    &lo, &hi);
            if (lo > hi) break;
        }

        if (lo > hi) break;
    }

    if (lo > hi) r[w] = 0;
}
```

векторный код

```
__m512 b[bec][3];
<инициализация b>;

_mm512_store_epi32(r, _mm512_set1_epi32(1));


__m512 lo = z0;
__m512 hi = z1;

for (i = 0; i < bec; i++)
{
    upgrade(_mm512_cmpeq_ps_mask(b[i][0], z0),
            b[i][1], b[i][2], &lo, &hi);
    if (!_mm512_cmplt_ps_mask(lo, hi)) break;
}

for (i = 0; i < bec; i++)
{
    bi0 = b[i][0];
    abi0 = ABS(bi0);

    for (j = i + 1; j < bec; j++)
    {
        bj0 = b[j][0];
        abj0 = ABS(bj0);

        upgrade(_mm512_cmplt_ps_mask(MUL(bi0, bj0), z0),
                FMADD(abi0, b[j][1], MUL(abj0, b[i][1])),
                FMADD(abi0, b[j][2], MUL(abj0, b[i][2])),
                &lo, &hi);
        if (!_mm512_cmplt_ps_mask(lo, hi)) break;
    }

    if (!_mm512_cmplt_ps_mask(lo, hi)) break;
}

_mm512_mask_store_epi32(r,
                        _mm512_cmplt_ps_mask(hi, lo),
                        _mm512_set1_epi32(0));
```

Рисунок 4. Схема перевода последовательного кода в векторный аналог для ядра функции tri_box_intersect_16.

Из рис. 4 видно, что правильно составленный предикатный код (на рисунке слева) может быть довольно просто переведен в векторный аналог. Для этого должны соблюдаться некоторые простые требования. Во-первых, необходимо удалить все else ветки в условных операторах. Этого можно добиться, например, путем расщепления оператора if-else на два противоположных условия. После этого любое условие if легко трансформируется в предикат на весь блок кода, находящийся под условием. Во-вторых, вызовы функции не должны находиться под предикатами. Вместо этого в нашем случае скалярное условие вызова функции upgrade трансформировалось в аргумент данной функции, и после этого код легко поддается векторизации. В остальном все вещественные скалярные инструкции были просто заменены на векторные аналоги. Также была выполнена векторизация функции upgrade с помощью слияния всех веток выполнения под их предикатами, результирующий код данной функции представлен на рис. 5.

```
01 void upgrade(__mmask16 m, __m512 f0, __m512 f1,
02              __m512 *lo, __m512 *hi)
03 {
04     __mmask16 c_f0z = _mm512_cmpeq_ps_mask(f0, z0);
05     __mmask16 c_f0n = _mm512_cmplt_ps_mask(f0, z0);
06     __mmask16 c_f0p = ~(c_f0z | c_f0n);
07     __mmask16 c_f1p = _mm512_cmplt_ps_mask(z0, f1);
08     __m512 k = _mm512_mask_div_ps(k, ~(m & c_f0z), f1, f0);
09
10     k = SUB(z0, k);
11     *lo = _mm512_mask_add_ps(*lo, m & c_f0z & c_f1p, *hi, z1);
12     *hi = _mm512_mask_min_ps(*hi, m & c_f0p, *hi, k);
13     *lo = _mm512_mask_max_ps(*lo, m & c_f0n, *lo, k);
14 }
```

Рисунок 5. Векторная реализация функции upgrade с пропагированным условием вызова внутрь функции

Заключение

В работе был выполнен анализ задачи об определении пересечения объемной декартовой сетки и поверхностной неструктурированной сетки, состоящей из треугольников. Эффективное решение данной задачи напрямую влияет на производительность расчетных кодов обтекания тел со сложной и изменяющейся геометрией. Была сформулирована математическая постановка задачи, предложены пути оптимизации ее реализации и выполнен анализ векторизации кода для микропроцессоров с поддержкой набора инструкций AVX-512. В результате оптимизации была реализована векторная функция определения попарного пересечения 16 треугольников и 16 прямоугольных параллелепипедов. Полученные программные коды были протестированы на суперкомпьютере МВС-10П, находящемся в МСЦ РАН, на сегменте, оборудованном микропроцессорами Intel Xeon Phi 7290 Knights Landing. Векторизованный программный код продемонстрировал ускорение 6,7 раз по сравнению со скалярной версией, скомпилированной компилятором icc с полным набором оптимизаций. Полученные результаты подтверждают пригодность описанного подхода к векторизации программного кода.

Работа выполнена в рамках проекта «Исследование с использованием суперкомпьютера физических особенностей нестационарных турбулентных течений и газодинамического управления этими течениями в элементах силовой установки гиперзвукового летательного аппарата» по программе Фундаментальных исследований президиума РАН «Фундаментальные основы прорывных технологий в интересах национальной безопасности». В работе был использован суперкомпьютер МВС-10П, находящийся в МСЦ-РАН.

Vectorization of finding the intersection of volume grid and surface grid for microprocessors with AVX-512 support

A.A. Rybakov

Abstract. Vectorization of program code is one of the most important low-level optimizations that can significantly improve the performance of supercomputer applications. The effect of the vectorization of the code is the more noticeable, the greater the length of the vector. The AVX-512 instruction set supported in modern Intel microprocessors is designed to work with 512-bit vectors and allows vectorize predicated code. In this paper, we consider a practical approach to vectorizing a complex software context that implements the intersection of volume cartesian and surface unstructured computational grids. This problem is used in the calculations of the flow around bodies with complex geometry, performed using the immersed boundary method, while a uniform cartesian grid with cells of the same size and a local-refinement grid can be used as a cartesian volume grid. A mathematical solution of the problem is given, its implementation is carried out, and it is also shown that vectorization using the AVX-512 instruction set allows acceleration of the code on the Intel Xeon Phi KNL microprocessor to be more than 6.5 times compared to the non-vectorized version with single precision float data.

Keywords. Vectorization, Intel Xeon Phi KNL, AVX-512, volume cartesian grid, surface unstructured grid, flat loop, intrinsic functions.

Литература

1. И.В. Абалакин, Н.С. Жданова, Т.К. Козубская. Метод погруженных границ для численного моделирования невязких сжимаемых течений. // Журнал вычислительной математики и математической физики, 2018, Т. 58, № 9, 1462–1471.
2. R. Mittal, G. Iaccarino. Immersed boundary methods. // Annual. Rev. Fluid Mech., 2005, V. 37, P. 239–261.
3. Y.-H. Tseng, J.H. Ferziger. A ghost-cell immersed boundary method for flow in complex geometry. // Journal of Computational Physics, 2003, V. 192, P. 593–623.

4. В.В. Винников, Д.Л. Ревизников. Метод погруженной границы для расчета сверхзвукового обтекания затупленных тел на прямоугольных сетках. // Труды МАИ, 2007, № 27.
5. В.И. Зоркальцев, М.А. Киселева. Системы линейных неравенств. // Учебное пособие, Иркутск, Издательство Иркутского государственного университета, 2007.
6. С.Н. Черников. Свертывание конечных систем линейных неравенств. // Доклады АН СССР, 1963, Т. 152, № 5, 1075–1078.
7. J. Jeffers, J. Reinders, A. Sodani. Intel Xeon Phi processor high performance programming. Knights Landing edition. // Morgan Kaufmann, 2016.
8. В.Ю. Волконский, С.К. Окунев. Предикатное представление как основа оптимизации программы для архитектур с явно выраженной параллельностью. // Информационные технологии, 2003, № 4, С. 36–45.
9. Intel C++ compiler 16.0 user and reference guide. // Intel Corporation, 2016.
10. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide // Дата обращения 15.10.1019.
11. B.M. Shabanov, A.A. Rybakov, S.S. Shumilin. Vectorization of High-performance Scientific Calculations Using AVX-512 Instruction Set. // Lobachevskii Journal of Mathematics, 2019, V. 40, № 5, P. 580–598.
12. W. McDoniel, M. Hohnerbach, R. Canales. et al. LAMMPS' PPPM Long-Range Solver for the Second Generation Xeon Phi. // J.M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS, 2017, V. 10266, P. 61–78.
13. T. Malas, T. Kurth, J. Deslippe. Optimization of the Sparse Matrix-Vector Products of an IDR Krylov Iterative Solver in EMGeo for the Intel KNL Manycore Processor. // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS, 2016, V. 9945. P. 378–389.
14. Л.А. Бендерский, А.А. Рыбаков, С.С. Шумилин. Векторизация перемножения малоразмерных матриц специального вида с использованием инструкций AVX-512. // Международный научный журнал «Современные информационные технологии и ИТ-образование», 2018, Т. 14, № 3, С. 594–602.
15. O. Krzikalla, F. Wende, M. Hohnerbach. Dynamic SIMD Vector Lane Scheduling. // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS, 2016, V. 9945, P. 354–365.
16. B. Cook, P. Maris, M. Shao. High Performance Optimizations for Nuclear Physics Code MFDn on KNL. // M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS, 2016, V. 9945, P. 366–377.
17. M. Bader, A. Breuer, W. Holtz, S. Rettenberger. Vectorization of an augmented Riemann solver for the shallow water equations. // Proceedings of the 2014 International Conference on High Performance Computing and Simulation, HPCS 2014, 2014, P. 193–201.
18. C.R. Ferreira, K.T. Mandli, M. Bader. Vectorization of Riemann solvers for the single- ans multi-layer shallow water equations. // Proceedings of the 2018 International Conference on High Performance Computing and Simulation, HPCS 2018, 2018, P. 415–422.
19. B. Bramas. Fast Sorting Algorithms using AVX-512 on Intel Knights Landing. // International Journal of Advanced Computer Science and Applications, 2017, V. 8, № 10, P. 337–344.
20. М.С. Гуськова, Л.Ю. Бараш, Л.Н. Щур. Применение AVX512-векторизации для увеличения производительности генератора псевдослучайных чисел. // Труды ИСП РАН, 2018, Т. 30, № 1, С. 115–126.
21. А.А. Рыбаков, С.С. Шумилин. Векторизация сильно разветвленного управления с помощью инструкций AVX-512. // Труды НИИСИ РАН, 2018, Т. 8, № 4, С. 114–126.
22. R. Allen, K. Kennedy. Optimizing Compilers for Modern Architectures. // Morgan Kaufmann, 2001.
23. А.А. Рыбаков, П.Н. Телегин, Б.М. Шабанов. Проблемы векторизации гнезд циклов с использованием инструкций AVX-512. // Электронный научный журнал: Программные продукты, системы и алгоритмы, 2018, № 3, С. 1–11.