УДК 004.42, 004.051

Рыбаков А.А.

Национальный исследовательский центр «Курчатовский институт», 123182, Москва, пл. Академика Курчатова, д. 1, Россия.
Межведомственный суперкомпьютерный центр Российской академии наук – филиал Федерального государственного учреждения «Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук», 119334, Москва, Ленинский проспект, 32а, Россия.

Векторизация циклов с условными операциями с помощью комбинирования векторных масок

Аннотация: Статья посвящена проблеме повышения эффективности векторизации вычислений на вещественных числах. При выполнении вычислений несколько одинаковых скалярных операций могут быть объединены в единую векторную команду, существенно повышая скорость выполнения программы. Данная оптимизация является критически важной для расчетных задач суперкомпьютерного моделирования. Основным объектом, на который нацелена векторизация вычислений, является цикл с независимыми итерациями. При относительно простом виде тела рассматриваемого цикла проблем с векторизацией, как правило, не возникает. При появлении в теле цикла сложного управления, вложенных циклов и вызовов функций оптимизирующий компилятор зачастую не справляется с векторизацией. Однако особенности набора векторных инструкций AVX-512 с поддержкой выборочной обработки элементов данных векторов позволяют векторизовать циклы с телом практически произвольного вида. В настоящей статье рассматривается подход к векторизации цикла, содержащего условия. Подход основан на оптимизации слияния путей исполнения программы под соответствующими предикатами. Векторизованный предикат представляет собой маску обработки элементов вектора. Такие маски используются в векторных инструкциях AVX-512. При векторизации циклов, тело которых содержит сложное управление, основной проблемой является низкая плотность масок векторизованного кода, что приводит к снижению производительности. В статье рассмотрены методы, позволяющие повысить плотность векторных масок и эффективность выполнения векторного кода. Разработанные методы апробированы на программном контексте газодинамического решателя. Данные по эффективности векторизации были получены в режиме эмуляции векторных инструкций и на реальной машине (микропроцессор Intel Xeon Phi Knights Landing). После применения оптимизаций векторного кода были достигнуты показатели эффективности векторизации до 0.75 в режиме эмуляции и до 0.47 на реальной машине.

Ключевые слова: векторизация, AVX-512, плоский цикл, векторизация условий, векторная маска.

Rybakov A. A.

National Research Centre «Kurchatov Institute», 123182, Moscow, 1 Academician Kurchatov Square, Russia.
Joint Supercomputer Center of the Russian Academy of Sciences – branch of Scientific Research Institute of System Analysis of the Russian Academy of Sciences, 119334, Moscow, Leninsky prospect, 32a, Russia.

Vectorization of loops with conditional operations by combining vector masks

Annotation: The article is devoted to the problem of increasing the efficiency of vectorization for calculations on real numbers. When performing calculations, several similar scalar operations can be combined into a single vector command, significantly increasing the speed of program execution. This optimization is crucial for computational tasks of supercomputer modeling. The main object that the vectorization of calculations is aimed at is a loop with independent iterations. With a relatively simple form of the body of the considered loop, problems with vectorization, as a rule, do not arise. When complex controls, nested loops, and function calls appear in the body of the loop, the optimizing compiler often fails to cope with vectorization. However, the features of the AVX-512 vector instruction set with support of selective processing of vector data elements make it possible to vectorize loops with a body of almost arbitrary structure. This article discusses an approach to vectorization of a loop which contains conditions. The approach is based on merging of program execution paths under the appropriate predicates. A vectorized predicate is a mask for processing vector elements. Such masks are used in AVX-512 vector instructions. When vectorizing loops whose body contains complex controls, the main problem is the low density of masks of the vectorized code, which leads to decrease in performance. The article discusses methods to increase the density of vector masks and the efficiency of vector code execution. The developed methods have been tested on the progamm context of a gas-dynamic solver. Data on vectorization efficiency were obtained in vector instruction emulation mode and on a real machine (Intel Xeon Phi Knights Landing microprocessor). After applying vector code optimizations, vectorization efficiency indicators were achieved up to 0.75 in emulation mode and up to 0.47 on a real machine.

Keywords: vectorization, AVX-512, flat loop, conditions vectorization, vector mask.

Введение

Повышение эффективности приложений суперкомпьютерного моделирования является критически важной задачей. Выполнение одного расчета по моделированию физического процесса может занимать несколько часов или дней. В условиях экспоненциального повышения количества различных расчетов, которые необходимо выполнить для проведения исследований или проектирования новых изделий, время работы суперкомпьютера становится крайне дефицитным ресурсом. В настоящее время для численного решения задач газовой динамики уже возникают потребности в использовании суперкомпьютерных ресурсов экзафлопсного диапазона производительности [1]. Можно отметить, что как сама задача, так и вычислительное поле суперкомпьютера зачастую имеют гетерогенную природу. По отношению к задаче это выражено в изменяющихся адаптивных сетках [2], блочно-структурированных сетках [3], индуцированных сетках [4], перекрывающихся или химерных сетках [5] и других особенностям. По отношению к вычислительному полю это выражено в совокупности отличающихся двух от друга вычислительных узлов, микропроцессоров и графических карт [6]. В таких условиях для организации счета требуются специальные подходы, учитывающие балансировку вычислительной нагрузки [7]. При проведении суперкомпьютерных расчетов широко используется распараллеливание вычислений между узлами суперкомпьютерного кластера с помощью MPI [8,9], распараллеливание внутри одного узла с помощью OpenMP [10,11] и распараллеливание на уровне команд с помощью векторизации.

Векторизация вычислений это низкоуровневая оптимизация, правильное применение которой способно кратно повысить производительность наиболее горячих участков программного кода и сократить энергопотребление [12]. В мире постоянно проводятся исследования, направленные на повышение производительности векторного кода. В работе [13] путем векторизации безусловных операций с помощью функций-интринсиков было продемонстрировано повышение производительности газодинамического решателя на 200% при исполнении на микропроцессорах Intel Xeon Phi KNL и Intel Xeon Scalable. В работе [14] описывается сравнение реализации римановских решателей в применении к теории мелкой воды, одним из результатов работы является ускорение решателя с помощью инструкций AVX-512 в 16.7 раз при работе с вещественными числами одинарной точности. В исследовании [15] было достигнуто ускорение в 3.27 газодинамического решателя ADflow, работающего на структурированных расчетных сетках. Ускорение достигнуто путем декомпозиции сетки на вычислительные блоки, которые могут быть эффективно обработаны с точки зрения использования кэш и применения векторных инструкций AVX-512. В работе [16] рассмотрена реализация расчета гравитационного взаимодействия между N телами, из результатов видно, что использование набора инструкций AVX-512 позволило ускорить работу приложения в 2 и более раз. Также использование векторизации позволяет ускорить работу не только газодинамических решателей. В работе [17] описано успешное применение векторных инструкций в задаче поиска сходных участков в белковых последовательностях. Работы [18,19] посвящены ускорению алгоритмов, связанных с шифрованием. Однако, в данной работе мы будем больше фокусироваться на программном контексте, работающим с вещественными числами и наиболее характерно представленном в реализации решателей для расчета физических процессов.

В качестве объекта исследования будем рассматривать тип программного контекста, который наилучшим образом подходит для векторизации – плоский цикл. Под плоским циклом будем подразумевать обычный цикл for (int i = 0; i < n; ++i) который удовлетворяет следующим требованиям. Во-первых, все итерации цикла являются независимыми друг от друга, что позволяет выполнять их в любом порядке. Во-вторых, на i-ой итерации цикла все обращения в массивы данных имеют вид a[i]. И, в-третьих, все используемые массивы данных не пересекаются и выровнены в памяти для использования 512-битного чтения их блоков. При выполнении этих простых требований использование инструкций AVX-512 позволяет векторизовать плоский цикл с телом практически произвольного вида (включая сложное управление, вложенные циклы и вызовы функций) [20]. В данной работе будем рассматривать подходы к повышению эффективности векторизации плоского цикла, содержащего условия. Векторизация плоского цикла с шириной векторизации w будет представлять собой объединение w последовательных итераций цикла в одну широкую итерацию, внутри которой все скалярные операции заменяются на векторные аналоги, а предикаты заменяются на векторные маски. В дальнейшем при рассмотрении будем опускать сам плоский цикл и рассматривать только его тело.

Слияние путей исполнения по условию

Универсальным способом векторизации программного кода, содержащего условия, является слияние всех ветвей исполнения под соответствующими предикатами. Рассмотрим это действие на примере простого условия cond, по результату которого выполняется переход на один из блоков block A и block B. Пусть известны вероятности перехода на эти блоки – они равны p и $1-p$ соответственно. Длины рассматриваемых блоков подберем таким образом, чтобы в сумме они давали единицу, а отношение их длин задавалось параметром α. Таким образом, длины блоков будут равны $\frac{\alpha}{\alpha+1}$ и $\frac{1}{\alpha+1}$ соответственно (см. рис. 1). При этом условимся считать, что длина блока и время его исполнения это по сути одно и то же (то есть, время исполнения блока исчисляется количеством содержащихся в нем операций).

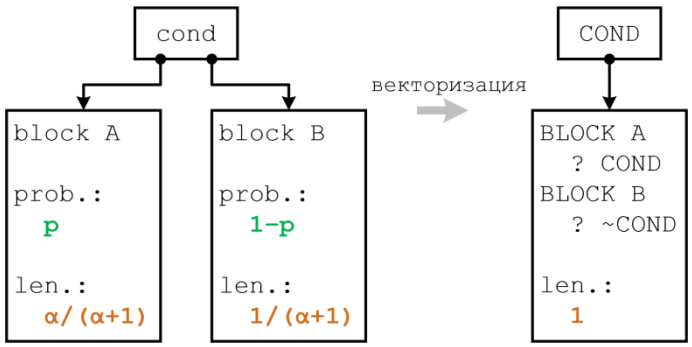

Рисунок 1. Схема векторизации участка программного кода, состоящего из одного условия и двух блоков, переход на которые осуществляется в соответствии с этим условием.

Согласно представленной схеме программного контекста математическое ожидание времени исполнения рассматриваемых блоков в зависимости от условия, будет равно $T_1 = \frac{p\alpha}{\alpha+1} + \frac{1-p}{\alpha+1} = p\left(\frac{\alpha-1}{\alpha+1}\right) + \left(\frac{1}{\alpha+1}\right)$. Время выполнения w таких участков кода в невекторизованном виде будет равно wT_1. При векторизации кода необходимо избавиться от операций перехода, вместо этого все операции блоков block A и block B должны быть поставлены под предикаты cond и ~cond соответственно. Далее выполняется объединение w участков кода, при котором скалярные операции под предикатами cond/~cond заменяются на векторные аналоги, выполняющиеся с использованием векторных масок COND/~COND. Так как длины блоков выбирались таким образом, чтобы в сумме они давали единицу, то время исполнения векторной версии кода в точности равно $T_w = 1$. Таким образом, эффективность векторизации рассмотренного фрагмента кода равна $e = \frac{T_1}{T_w} = p\left(\frac{\alpha-1}{\alpha+1}\right) + \left(\frac{1}{\alpha+1}\right)$. На рис. 2 представлены графики зависимостей эффективности векторизации для разных значений параметра α.

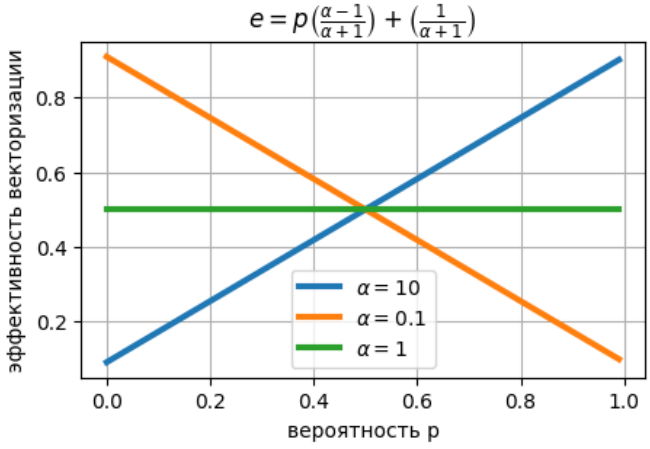

Рисунок 2. Графики зависимостей эффективности векторизации от вероятности перехода на block A при значениях отношения длин блоков block A и block B, равных 10.0, 0.1, 1.0 и при использовании простого слияния ветвей исполнения.

Из рис. 2 видно, что при $\alpha = 1$ (то есть при одинаковых длинах блоков block A и block B) эффективность векторизации постоянна и равна 0.5. В тех же случаях, когда длины блоков отличаются, эффективность векторизации возрастает, если вероятность перехода на более длинный блок выше, чем на более короткий блок. В любом случае, можно констатировать, что такой подход прямого слияния ветвей исполнения под соответствующими предикатами в единый линейный участок является крайне неэффективным. При возрастании количества условий эффективность векторизации таким способом падает экспоненциально. Это связано с появлением в программном коде большого количества векторных инструкций с практически пустыми масками. Для повышения эффективности векторизации контекста с условиями требуется рассмотрение других подходов, позволяющих повысить плотность масок исполняемых векторных команд.

В качестве примера, на котором мы будем проводить анализ методов повышения эффективности векторизации кода с условиями, возьмем одну из функций реализации газодинамического решателя – функцию

prefun[1]. Для удобства будем пользоваться реализацией данной функции на языке программирования C, как это представлено на листинге 1 (исходный код функции доступен в открытом репозитории[2]).

```
void scase_prefun_1(float& f,
                    float& fd,
                    float p,
                    float dk,
                    float pk,
                    float ck)
{
    if (p <= pk)
    {
        // Rarefaction wave.

        float prat = p / pk;

        f = riemann::sg4 * ck * (pow(prat, riemann::sg1) - 1.0f);
        fd = (1.0f / (dk * ck)) * pow(prat, -riemann::sg2);
    }
    else
    {
        // Shock wave.

        float ak = riemann::sg5 / dk;
        float bk = riemann::sg6 * pk;
        float qrt = sqrt(ak / (bk + p));

        f = (p - pk) * qrt;
        fd = (1.0f - 0.5f * (p - pk) / (bk + p)) * qrt;
    }
}
```

Листинг 1. Код функции prefun, обрабатывающей один набор скалярных данных.

В данном листинге мы видим реализацию функции prefun, обрабатывающей один набор скалярных данных. Функция принимает на входные аргументы p, dk, pk, ck и вычисляет выходные аргументы f, fd. Функция содержит одно условие в зависимости от которого выходные аргументы вычисляются с помощью той или иной последовательности операций. Все задействованные в реализации функции операции имеют векторные аналоги в наборе инструкций AVX-512 (точнее в наборе функций-интринсиков), поэтому приведенная функция может быть векторизована путем замены скалярных операций векторными аналогами и слияния ветвей исполнения под соответствующими предикатами, как это показано на листинге 2. В процессе векторизации умышленно не применялись никакие локальные оптимизации, все скалярные операции были строго заменены на векторные аналоги с сохранением порядка вычислений с точности до ассоциативности умножения. Для удобства код, относящийся к блокам block A и block B, заключен в фигурные скобки.

```
void vcase_prefun_1(__m512& f,
                    __m512& fd,
                    __m512& p,
                    __m512& dk,
                    __m512& pk,
                    __m512& ck,
                    __mmask16 m)
{
    // Conditions.
    __mmask16 cond = _mm512_kand(_mm512_cmple_ps_mask(p, pk), m);
    __mmask16 ncond = _mm512_kand(_mm512_knot(cond), m);

    // The first branch.
    {
        __m512 prat = _mm512_mask_div_ps(zero, cond, p, pk);

        f = _mm512_mask_mul_ps(f, cond,
                 _mm512_mask_mul_ps(zero, cond, riemann::g4, ck),
                 _mm512_mask_sub_ps(zero, cond,
                     _mm512_mask_pow_ps(zero, cond, prat, riemann::g1),
                     one));

        fd = _mm512_mask_mul_ps(fd, cond,
                  _mm512_mask_div_ps(zero, cond,
                      one,
                       mm512_mask_mul_ps(zero, cond, dk, ck)),
```

[1] E. F. Toro. Riemann Solvers and Numerical Methods for Fluid Dynamics. A Practical Introduction. // Springer, 2nd edition, 1999, p. 158.

[2] Открытый репозиторий https://github.com/r-aax/flatvec, исходный файл fvcases/cases.cpp

```
                _mm512_mask_pow_ps(zero, cond,
                    prat,
                    _mm512_mask_sub_ps(zero, cond, zero, riemann::g2)));
    }

    // The second branch.
    {
        __m512 ak = _mm512_mask_div_ps(zero, ncond, riemann::g5, dk);
        __m512 bk = _mm512_mask_mul_ps(zero, ncond, riemann::g6, pk);
        __m512 qrt = _mm512_mask_sqrt_ps(zero, ncond,
                        _mm512_mask_div_ps(zero, ncond,
                            ak,
                            _mm512_mask_add_ps(zero, ncond, bk, p)));

        f = _mm512_mask_mul_ps(f, ncond,
                _mm512_mask_sub_ps(zero, ncond, p, pk),
                qrt);

        fd = _mm512_mask_mul_ps(fd, ncond,
                 _mm512_mask_sub_ps(zero, ncond,
                     one,
                     _mm512_mask_mul_ps(zero, ncond,
                         half,
                         _mm512_mask_div_ps(zero, ncond,
                             _mm512_mask_sub_ps(zero, ncond, p, pk),
                             _mm512_mask_add_ps(zero, ncond, bk, p)))),
                 qrt);
    }
}
```

Листинг 2. Код функции prefun, обрабатывающей один набор векторных данных, представляющий собой объединение 16 наборов скалярных данных.

Сравнивая листинги 1 и 2, содержащие скалярный и векторный код, можно установить соответствие между участвующими в них объектами, как это показано в таблице 1.

Таблица 1. Соответствие между скалярными объектами реализации scase_prefun_1 и векторными объектами реализации vcase_prefun_1.

Объект скалярной версии scase_prefun_1	**Объект векторной версии vcase_prefun_1**
скалярные аргументы float f, fd, p, dk и т.д.	векторные аргументы __m512 f, fd, p, dk и т.д.
скалярные глобальные данные riemann::sg1, riemann::sg2 и т.д.	векторные глобальные данные riemann::g1, riemann::g2
скалярные операции +, -, *, /, pow	векторные команды, заданные функциями-интринсиками _mm512_mask_add_ps, _mm512_mask_sub_ps, _mm512_mask_mul_ps, _mm512_mask_div_ps, _mm512_mask_pow_ps
скалярная операция сравнения <=	векторные операции получения масок _mm512_cmple_ps_mask, _mm512_knot, _mm512_kand

Из приведенного листинга 2 векторного кода видно, что часть команд выполняется с использованием векторной маски cond, тогда как другая часть команд использует векторную маску ncond. При этом понятно, что если маска cond окажется нулевой, то нет необходимости выполнять инструкции, использующие эту маску. То же касается маски ncond. Для того, чтобы учесть это изменение достаточно перед выполнением блока операций, относящихся к block A, проверить маску cond на пустоту, в противном случае вообще не выполнять соответствующие инструкции. Аналогично следует поступить с маской ncond. Скорректируем зависимости эффективности векторизации для рассматриваемого примера с учетом этого условия. Величина T_1 остается неизменной, а вот T_w несколько изменится. Если считать, что в каждом наборе обрабатываемых данных переход на тот или иной блок является независимым событием, то вероятность пустой маски cond будет равна $(1-p)^w$, тогда как вероятность пустой маски ncond равна p^w. Тогда общая длина векторизованного кода может быть вычислена как $T_w = \left(1-(1-p)^w\right)\left(\frac{\alpha}{\alpha+1}\right)+\left(1-p^w\right)\left(\frac{1}{\alpha+1}\right)$, а эффективность векторизации примет следующий вид:

$$e = \frac{p\left(\frac{\alpha-1}{\alpha+1}\right)+\left(\frac{1}{\alpha+1}\right)}{\left(1-(1-p)^w\right)\left(\frac{\alpha}{\alpha+1}\right)+\left(1-p^w\right)\left(\frac{1}{\alpha+1}\right)}$$

На рис. 3 представлены зависимости эффективности векторизации при разных значениям параметра α с учетом проверок масок на пустоту для ширины векторизации 16, что соответствует использованию вещественного формата данных одинарной точности.

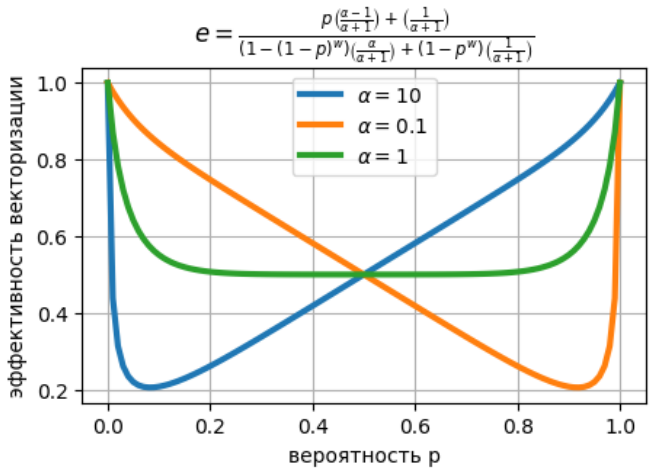

Рисунок 3. Графики зависимостей эффективности векторизации от вероятности перехода на block A при значениях отношения длин блоков block A и block B, равных 10.0, 0.1, 1.0 и при использовании слияния ветвей исполнения с проверкой масок на пустоту.

Из рис. 3 видно, что эффективность векторизации возрастает, если значение вероятности перехода на один из блоков близко к единице, однако в среднем вероятность векторизации остается невысокой. Когда мы вычисляли вероятность появления пустой маски, то принимали условное соглашение, что выполнение условий для разных наборов скалярных данных являются независимыми событиями. На самом деле это не так и существенным образом зависит от локальности размещения данных, участвующих в расчетах [21]. Рассмотрим более подробно наше условие p <= pk. Элементы данных p и pk, свои для каждой расчетной ячейки. Если речь идет о физических расчетах (а функция prefun относится к газодинамическому решателю), то значение элемента данных изменяется не слишком сильно при переходе от одной ячейки к соседней ячейке. Из этого следует, что и значение условия p <= pk при переходе от одной ячейке к соседней будет изменяться также не слишком быстро. Но условие это дискретная величина, а это значит, что часто значение условия будет сохраняться при переходе к соседней ячейке. Рассмотри теоретический краевой случай, когда во время обработки n наборов скалярных данных условие выполняется для первых np из них и не выполняется для оставшихся $n(1-p)$. Для простоты будем считать, что все числа np, $n(1-p)$ кратны ширине векторизации w. В этом случае очевидно, что во время выполнения векторной версии кода первые $\frac{np}{w}$ масок cond будут полные, а остальные $\frac{n(1-p)}{w}$ масок cond будут пустыми (с масками ncond ситуация будет обратной). В таком случае, время исполнения векторизованной версии кода с проверкой обеих масок cond и ncond на пустоту в точности совпадет со временем T_1. Таким образом, эффективность векторизации в этом теоретически идеальном случае будет ровно единица с поправкой на дополнительные операции проверки масок на пустоту.

Для рассматриваемой функции prefun были собраны расчетные данные распределения плотности маски условия p <= pk, чтобы оценить вероятность появления пустых масок cond и ncond. На рис. 4 представлено распределение плотности масок в случае независимости условий для разных наборов скалярных данных (нетрудно видеть, что распределение является нормальным). Результаты распределения плотности маски cond, собранные по настоящему профилю исполнения векторного кода на реальных данных, представлены на рис. 5.

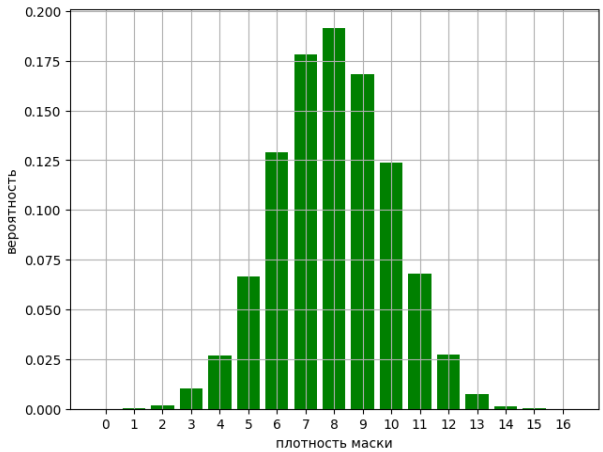

Рисунок 4. Гистограмма распределения плотностей маски cond при условии, что все условия p <= pk для наборов скалярных данных являются независимыми.

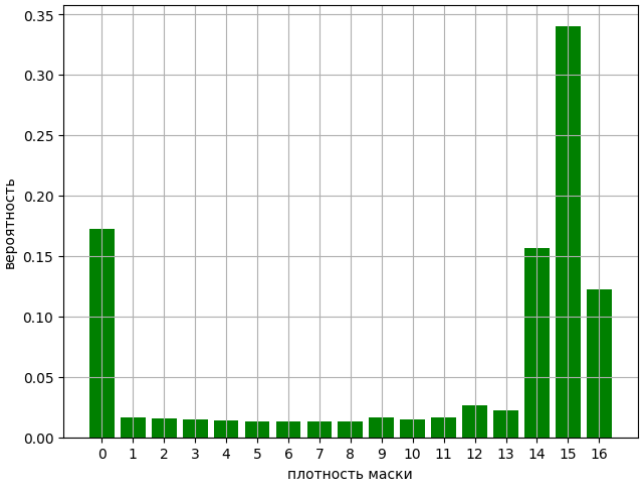

Рисунок 5. Гистограмма распределения плотностей маски cond на реальном профиле исполнения.

Из рис. 4 и рис. 5 видно, что распределение плотностей масок на реальных данных совершенно не похоже на распределение, вычисленное в предположении о независимости условий переходов. Можно заметить, что в реальном коде более четверти всех масок cond являются либо пустыми, либо полными (в этом случае пустой является маска ncond), а значит использование проверок масок на пустоту обосновано.

Комбинирование масок при исполнении векторного кода

В общем случае можно считать, что в результате слияния под соответствующими предикатами ветвей исполнения внутри тела плоского цикла мы получим совокупность векторных блоков, обрабатывающихся сходим образом: загрузка входных данных in_data под маской векторного блока, выполнение вычислений block под маской блока, сохранение результатов out_data под маской блока (см. рис. 6).

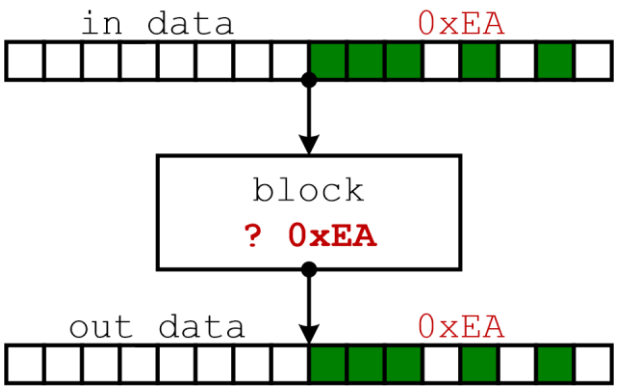

Рисунок 6. Схема вычислений векторизованного блока команд с входными данными in_data, выходными данные out_data и маской исполнения 0xEA.

Проверка маски блока на пустоту может повысить эффективность кода, если маски часто оказываются пустыми. Однако этот никак не поможет в том случае, если в маске выставлено несколько битов. В некоторых случаях достичь повышения производительности можно путем объединения двух соседних векторных блоков. Рассмотрим простейший случай такого объединения. Если у нас есть два соседних векторных блока in_data_1 → block → out_data_1 и in_data_2 → block → out_data_2, которые должны выполняться под разными векторными масками mask_1 и mask_2, и в дополнение к этому для этих масок выполнено условие (mask_1 & mask_2) == 0x0, то вычисление этих двух соседних блоков можно объединить. Вместо последовательного выполнения двух векторных блоков можно объединить входные данные с помощью слияния векторов по условию in_data = _mm512_mask_blend_ps(mask_1, in_data_2, in_data_1), после чего выполнить тот же блок вычислений под маской mask_1 | mask_2. Ввиду отсутствия пересечения векторных масок в результирующем выходном векторе out_data будут содержаться как необходимые элементы вектора out_data_1, так и необходимые элементы вектора out_data_2. Последним действием, которое нужно выполнить является извлечение из объединенного результата out_data векторов out_data_1 и out_data_2 (см. рис. 7). В результате такого преобразования в случае отсутствия пересечения векторных масок количество вычислений рассматриваемого блока block сокращается вдвое, а плотность векторных масок внутри блока повышается. Однако вместе с этим появляются накладные расходы, связанные с проверками масок, а также операции слияния векторов до вычислений блока и выделения нужных данных после вычислений. Заметим, что эту технику можно применять для объединения трех и более соседних блоков, однако это связано с еще большим возрастанием накладных расходов.

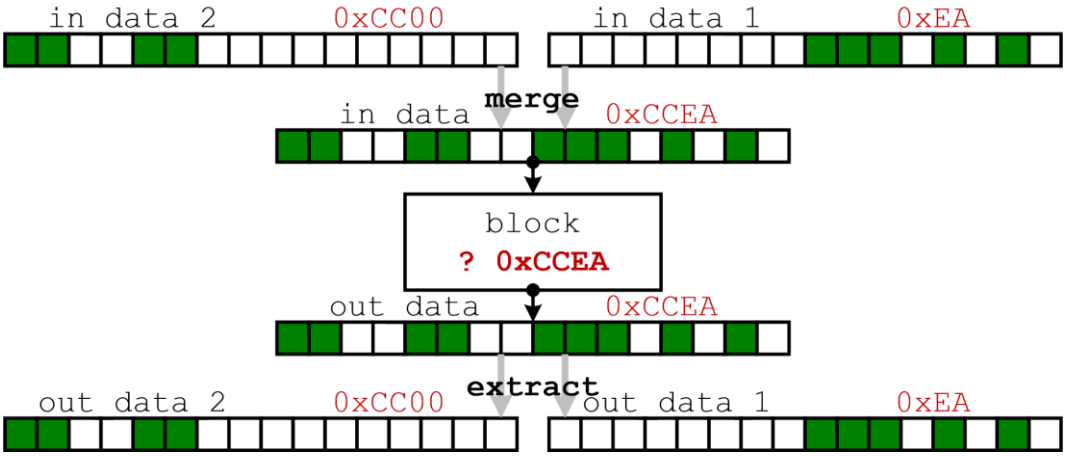

Рисунок 7. Схема вычислений с объединением двух векторизованных блоков in_data_1 → block → out_data_1, in_data_2 → block → out_data_2. Объединение допустимо, так как векторные маски 0xCC00 и 0xEA не пересекаются, объединенный блок выполняется под маской 0xCCEA.

Еще один подход, о котором стоит упомянуть, но который не тестировался в рамках данной работы, связан с объединением соседних блоков, для которых (mask_1 & mask_2) != 0x0. Если мы имеем дело с двумя масками низкой плотности, которые пересекаются, но для которых выполнено условие popcnt(mask_1) + popcnt(mask_2) <= w, то такие блоки также можно объединить. Для этого необходимо найти такое преобразование одной из масок (например, mask_1) perm_to, что оно будет иметь обратное преобразование perm_from(perm_to(mask_1)) == mask_1 и будет выполнено условие (perm_to(mask_1) & mask_2) == 0x0. В этом случае элементы входных данных переставляются местами в соответствии с преобразованием perm_to, применяется описанная выше техника объединения блоков, а для выходных данных выполняется перестановка элементов в соответствии с преобразованием perfm_from (см. рис. 8).

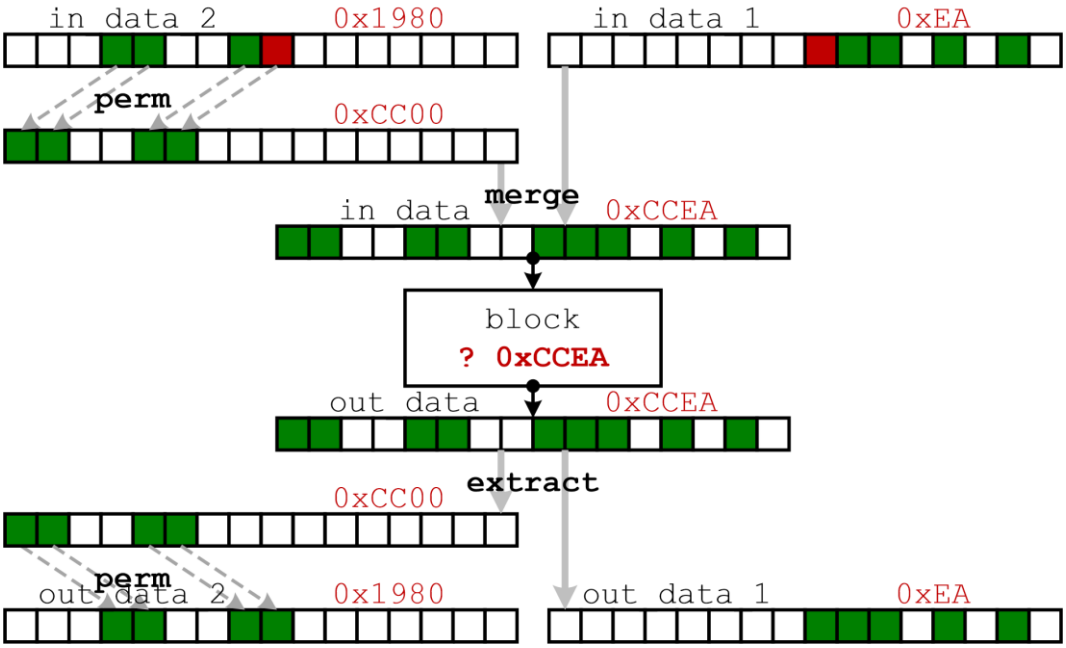

Рисунок 8. Схема вычислений с объединением двух векторизованных блоков при условии пересечения их масок. Для объединения применяется техника изменения порядка элементов в данных одного из блоков.

В описанном подходе следует отметить следующие моменты. Во-первых, объединять можно не только два соседние блока, но также три и более, хотя это существенно усложняет программный код и увеличивает количество накладных расходов. Во вторых, следует принимать во внимание доступные операции по изменению порядка расположения элементов векторов, так как таких операций достаточно много и они отличаются по времени выполнения (SHUF, UNPCK, VPERM, VPERMIL)[3].

Результаты

Для анализа полученных результатов были рассмотрены следующие три подхода к векторизации плоского цикла с условием. В качестве базового метода векторизация принималось простое слияние путей исполнения под соответствующими предикатами с последующим объединением w последовательных скалярных итераций в одну векторную (простое слияние). Данный базовый метод сравнивался с двумя рассмотренными выше улучшениями: проверка масок блоков на пустоту (проверка масок) и слияние двух соседних блоков при условии отсутствия пересечения их масок (комбинирование масок). Анализ эффективности применения преобразований рассматривался на приведенной в листинге 1 функции prefun из реализации газодинамического римановского решателя. Профиль исполнения функции собирался на задачах моделирования распада разрыва при различных начальных условиях [22,23]. Собранный профиль исполнения функций римановского решателя доступен в открытом репозитории[4]. Эффективность векторизации при выбранных подходах измерялась двумя способами. В качестве первого способа использовался режим эмуляции векторных инструкций. В настоящее время используются различные эмуляторы AVX-512 с помощью которых можно оценить эффективность векторного кода [24]. В данной работе мы ограничились инструментом, позволяющим отследить плотность используемых в коде масок и общее количество скалярных и векторных операций [25]. Вторым способом сравнения был замер производительности результирующего векторного кода на микропроцессоре Intel Xeon Phi Knights Landing 7290[5]. Результаты сравнения представлены на рис. 9.

[3] Intel 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4.
[4] Открытый репозиторий https://github.com/r-aax/riemann_vec
[5] Jeffers J., Reinders J., Sodani A. Intel Xeon Phi processor high performance programming. Knights Landing edition. Morgan Kaufmann. 2016.

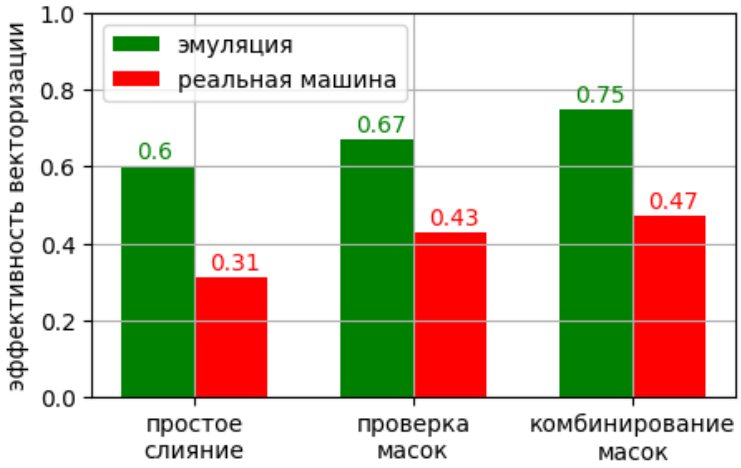

Рисунок 9. Результаты сравнения эффективности векторизации при простом слиянии, с проверкой масок и с комбинированием масок в режимах эмуляции и на микропроцессоре Intel Xeon Phi Knights Landing 7290.

Эксперименты показали, что в режиме эмуляции слияние ветвей исполнения привело к эффективности векторизации 0.6. Использование дополнительных методов повышения плотности векторных масок в коде – проверки масок и комбинирования масок – привело к повышению эффективности векторизации до показателей 0.67 и 0.75 соответственно. Эффективность векторизации при проведении замеров на реальной машине оказалась скромнее. Простое слияние путей исполнения позволило достичь эффективности 0.31, а использование проверок масок и комбинирования масок позволило повысить эффективность векторизации до 0.43 и 0.47 соответственно.

Заключение

Исследования, проведенные в рамках данной работы, показали, что векторизация плоских циклов с условиями может быть выполнена и при этом могут достигаться высокие показатели эффективности векторизации. Так на рассмотренном отдельном примере из реализации газодинамического решателя простое слияние путей исполнения привело к эффективности векторизации с показателем 0.31 на реальной машине, а использование дополнительных стратегий повышения плотности векторных масок в результирующем коде позволило повысить этот показатель до значения 0.47. При этом в работе были рассмотрены только самые простые подходы к проверке и комбинированию масок. Не затрагивались методы комбинирования масок для трех и более векторных блоков, не рассматривались методы комбинирования для пересекающихся масок. Так как возможности применения стратегий повышения плотности масок связаны с динамическими проверками масок, то это сопряжено с возникновением серьезных накладных расходов. Интересным видится подход к выбору стратегии повышения плотности масок, основанный на предварительном анализе профиля исполнения того или иного участка кода (анализ масок и вероятностей переходов). Такие эксперименты планируется выполнить в последующих исследованиях.

Работа выполнена в рамках государственного задания НИЦ «Курчатовский институт» по теме FNEF-2024-0016.

Литература:

1. Trebotich D. Exascale CFD in heterogenous systems. *Journal of Fluids Engineering.* 2024:146(4):1-19. https://doi.org/10.1115/1.4064534
2. Menshov I., Pavlukhin P. GPU-native gas dynamic solver on octree-based AMR grids. *Journal of Physics Conference Series.* 2020:1640(1):012017. https://doi.org/10.1088/1742-6596/1640/1/012017
3. Bosnjak D., Pepe A., Schussnig et al. Higher-order block-structured hex meshing of tubular structures. *Engineering with Computers.* 2023:40(2):931-951. https://doi.org/10.1007/s00366-023-01834-7
4. Carr G.E., Biocca N., Urquiza S.A. A biologically-inspired mesh moving method for cyclic motions mesh fatigue. *Computational Mechanics.* 2024. https://doi.org/10.1007/s00466-024-02514-z
5. Park Y.M., Jee S. Numerical study on interactional aerodynamics of a quadcopter in hover with overset mesh in OpenFOAM. *Physics of Fluids.* 2023:35(8):085138. https://10.1063/5.0160689
6. Wan Y., Zhao Z., Liu J. et al. Large-scale homo- and heterogeneous parallel paradigm design based on CFD application PHengLEI. *Concurrency and Computation Practice and Experiment.* 2024:36:e7933. https://doi.org/10.1002/cpe.7933
7. Jakobs T., Klöckner O., Rünger G. Parallelization with load balancing of the weather scheme WSM7 for heterogeneous CPU-GPU platforms. *The Journal of Supercomputing.* 2024:80:14645-14665. https://doi.org/10.1007/s11227-024-06009-9
8. Lai J., Yu H., Tian Z. et al. Hybrid MPI and CUDA parallelization for CFD applications on multi-GPU HPC clusters. *Scientific Programming.* 2020:1-15. https://doi.org/10.1155/2020/8862123

9. Bashir S., Usman A., Mumtaz Y. et al. Parallelization of lattice Bolzmann method for CFD using message passing interface. *Thermal Science.* 2022:26(spec. issue 1):211-218. https://doi.org/10.2298/TSCI22S1211B
10. Lin H., Yan L., Chang Q. et al. O2ath: an OpenMP offloading toolkit for the sunway heterogeneous manycore platform. *CCF Transactions on High Performance Computing.* 2024:6(7). https://doi.org/10.1007/s42514-024-00191-1
11. He X., Wang K., Feng Y. et al. An implementation of MPI and hybrid OpenMP/MPI parallelization strategies for an implicit 3D DDG solver. *Computers & Fluids.* 2022:241(4):105455. https://doi.org/10.1016/j.compfluid.2022.105455
12. Cebrian J.M., Natvig L., Jahre M. Scalability analysis of AVX-512 extensions. *The Journal of Supercomputing.* 2020:76:2082-2097. https://doi.org/10.1007/s11227-019-02840-7
13. Kulikov I., Chernykh I., Tutukov A. A new hydrodynamic code with explicit vectorization instructions optimizations that is dedicated to the numerical simulation of astrophysical gas flow. I. Numerical method, tests, and model problem. *The Astrophysical Journal. Supplement Series.* 2019;243:4;15pp. https://doi.org/10.3847/1538-4365/ab2237
14. Glinting B.M., Mundani R.-P. Comparison of shallow water solvers: applications for dam-break and tsunami cases with reordering strategy for efficient vectorization on modern hardware. *Water.* 2019:11(4):639. https://doi.org/10.3390/w11040639
15. Yildirim A., Mader C., Martins J.R.R.A. Accelerating parallel CFD codes on modern vector processors using blockettes. *PASC'21: Proceedings of the Platform for Advanced Scientific Computing Conference.* 2021. https://doi.org/10.1145/3468267.3470615
16. Rucci E., Moreno E., Pousa A. et al. Optimization of the N-body simulation on Intel's architectures based on AVX-512 instruction set. *In book: Computer Science – CACIC 2019.* 2020. https://doi.org/10.1007/978-3-030-48325-8_3
17. Rucci E. Garcia C., Botella G. et al. SWIMM 2.0: Enhanced Smith-Waterman on Intel's multicore and manycore architectures based on AVX-512 vector extensions. *International Journal of Parallel Programming.* 2019;47(17). https://doi.org/10.1007/s10766-018-0585-7
18. Choi Y., Choi H., Chung S. AVX512Crypto: parallel implementations of Korean block ciphers using AVX-512. *IEEE Access.* 2023. https://doi.org/10.1109/ACCESS.2023.3278993
19. Cheng H., Fotiadis G., Großschädl J. et al. Batching CSIDH group actions using AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems.* 2021:4:618-649. https://doi.org/10.46586/tches.v2021.i4.618-649
20. Savin G.I., Shabanov B.M., Rybakov A.A., Shumilin S.S. Vectorization of flat loops of arbitrary structure using instructions AVX-512. *Lobachevskii Journal of Mathematics.* 2020:41:12:2575-2592. https://doi.org/10.1134/S1995080220120331
21. Rybakov A., Chopornyak A. Improving vector code performance by monitoring masks density in vector instructions. *Trudy NIISI RAN.* 2020:10:40-47. (in Russ.) https://doi.org/10.25682/NIISI.2020.4.0006
22. Toh Y.H. Efficient non-iterative multi-point method for solving the Riemann problem. *Nonlinear Dynnamics.* 2024:112(1):1-13. https://doi.org/10.1007/s11071-023-09229-5
23. Zeng Z., Feng C., Yu C. et al. Linearized double-shock approximate Riemann solver for augmented linear elastic solid. *Numerical Mathematics Theory Methods and Applications.* 2021:15(1). https://doi.org/10.4208/nmtma.OA-2021-0021
24. Lee S., Kim Y., Nam D. et al. Gem5-AVX: Extension of the Gem5 simulator to support AVX instruction sets. *IEEE Access.* 2024. https://doi.org/10.1109/ACCESS.2024.3359296
25. Rybakov A.A., Shwindt A.N. Tools for vectorizing a flat loop body using AVX-512 vector instructions. Software & Systems. 2023:36(4):561-572. (in Russ.) https://doi.org/10.15827/0236-235X.142.561-572

Об авторе:

Рыбаков Алексей Анатольевич, к. ф.-м. н., начальник отдела суперкомпьютерных технологий и систем, отделения суперкомпьютерных систем и параллельных вычислений Национального исследовательского центра «Курчатовский институт», 123182, Москва, пл. Академика Курчатова, д. 1, ведущий научный сотрудник МСЦ РАН – филиала ФГУ ФНЦ НИИСИ РАН, 119334, Москва, Ленинский проспект, 32а. Область научных интересов: компьютерное моделирование, теория графов, параллельное программирование, функциональное программирование. email: rybakov.aax@gmail.com. Телефон: +7 903 138-88-77. ORCID: 0000-0002-9755-8830.

Note on the author:

Rybakov Alexey, Candidate of Physical and Mathematical Sciences, head of the department of supercomputer technologies and systems, division of supercomputer systems and parallel calculations of National Research Centre «Kurchatov Institute», 123182, Moscow, 1 Academician Kurchatov Square, lead researcher in Joint Supercomputer Center of the Russian Academy of Sciences – branch of Scientific Research Institute of System Analysis of the Russian Academy of Sciences, 119334, Moscow, Leninsky prospect, 32a Scopes of interest: computer modeling, graph theory,

parallel programming, functional programming. email: rybakov.aax@gmail.com. Phone number: +7 903 138-88-77. ORCID: 0000-0002-9755-8830.