ВЕКТОРИЗАЦИЯ ЦИКЛОВ В ОТКРЫТЫХ КОМПИЛЯТОРАХ ДЛЯ АРХИТЕКТУР С КОРОТКИМИ ВЕКТОРНЫМИ РЕГИСТРАМИ

О. В. Молдованова[1,2], И. И. Кулагин[1,2], М. Г. Курносов[1,2]

[1] Сибирский государственный университет телекоммуникаций и информатики,
[2] Институт физики полупроводников им. А.В. Ржанова СО РАН

УДК 004.272.25

В работе выполнен экспериментальный анализ эффективности подсистем автоматической векторизации циклов в открытых компиляторах GCC C/C++ и LLVM/Clang, выявлены классы трудновекторизуемых циклов на архитектуре Intel 64. В качестве тестового набора циклов использован пакет Extended Test Suite for Vectorizing Compilers, содержащий основные виды циклов, встречающихся в научных приложениях на языке C. Особое внимание в работе уделено гнезду из трех циклов, в котором выполняются арифметические операции над элементами двумерных массивов. Такие виды циклов характерны для подпрограмм библиотек линейной алгебры BLAS уровня 3. В работе рассмотрено отображение такого гнезда циклов на архитектуру с короткими векторными регистрами Intel AVX. В качестве представителя целевого гнезда циклов использован алгоритм умножения матриц по определению (DGEMM). Исследовано 6 способов векторизации указанного алгоритма. С целью нахождения границ применимости предложенных векторизованных версий в автоматическом векторизаторе открытых компиляторов проведено экспериментальное исследование эффективности реализованных алгоритмов на процессорах Intel с микроархитектурами Haswell и Broadwell.

Ключевые слова: векторизация, AVX, компиляторы, циклы, умножение матриц, DGEMM.

Введение. Современные процессоры вычислительных систем (ВС) характеризуются поддержкой и активным развитием трех основных форм параллельной обработки информации: параллелизм потоков на уровне процессорных ядер, параллелизм инструкций на уровне суперскалярного конвейера ядра, а также параллелизм обработки данных векторными арифметико-логическими устройствами. Значительное внимание со стороны разработчиков процессоров уделяется также развитию векторных наборов инструкций (Intel AVX, IBM AltiVec, ARM NEON SIMD). В частности компания Fujitsu анонсировала переход в будущей версии эксафлопсной системы K Computer на процессоры с архитектурой ARMv8.2-A, реализующей широкие векторные регистры переменной длины (scalable vector extension), а компания Intel активно развивает набор инструкций AVX-512. По этой причине постановки задач и работы по автоматической векторизации программ в последние десятилетия получили новый виток развития: компиляторная автоматическая векторизация; SIMD-директивы OpenMP и Cilk Plus; языковые расширения и библиотеки [1, 2, 5].

Цель данной работы заключается в выполнении экспериментального анализа эффективности подсистем автоматической векторизации циклов в открытых компиляторах GCC C/C++ и LLVM/Clang и определении классов трудновекторизуемых циклов. В качестве

Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (проекты 16-07-00992, 15-07-00653) и Программы Президиума РАН №27 «Фундаментальные проблемы решения сложных практических задач с помощью суперкомпьютеров».

тестового набора циклов использован пакет Extended Test Suite for Vectorizing Compilers [2-5]. Особое внимание в работе уделено гнезду из трех циклов, в котором выполняются арифметические операции над элементами двумерных массивов. Такие виды циклов характерны для подпрограмм библиотек линейной алгебры BLAS уровня 3 [6]. В работе рассматривается отображение такого гнезда циклов на архитектуру с короткими векторными регистрами Intel AVX. В качестве представителя целевого гнезда циклов использован алгоритм умножения матриц по определению (DGEMM). Исследовано 6 способов векторизации этого алгоритма. С целью нахождения границ применимости указанных векторизованных версий в автоматическом векторизаторе открытых компиляторов проведено экспериментальное исследование эффективности реализованных алгоритмов на процессорах Intel с микроархитектурами Haswell и Broadwell.

1. Архитектуры с короткими векторными регистрами. Наборы команд практически всех архитектур современных процессоров включают поддержку векторных инструкций: Intel SSE/AVX/AVX-512, ARM NEON SIMD, IBM AltiVec, MIPS MSA. Процессоры, реализующие поддержку таких инструкций, содержат одно или несколько параллельно работающих векторных арифметико-логических устройств (АЛУ) и совокупность векторных регистров. В отличие от векторных систем 1990-х годов, современные процессоры поддерживают выполнение операций с векторами относительно небольшой длины (64 – 512 бит), предварительно загруженными из оперативной памяти в векторные регистры (класс векторных систем «регистр-регистр»).

Основная сфера применения векторных инструкций – сокращение времени работы с одномерными массивами. При векторизации происходит трансформация выполнения итераций обработки массивов данных в векторные инструкции, выполняющиеся одновременно над несколькими экземплярами данных. Как правило, ускорение, достигаемое при использовании векторных инструкций, в первую очередь определяется количеством элементов массива, помещающихся в векторный регистр. Например, каждый из 16 векторных регистров AVX имеет ширину 256 бит, что позволяет загружать в них 16 элементов типа **`short int`** (16 бит), 8 элементов типа **`int`** или **`float`** (32 бита) и 4 элемента типа **`double`** (64 бита). Соответственно, при использовании AVX ожидаемое ускорение выполнения операций с массивами типа **`short int`** – 16 раз, **`int`** и **`float`** – 8 раз, **`double`** – 4 раза.

Процессоры Intel Xeon Phi поддерживают набор векторных инструкций AVX-512 и содержат 32 векторных регистра шириной 512 бит. Каждое ядро процессора с микроархитектурой Knights Corner содержит одно векторное АЛУ шириной 512 бит, а ядра процессора с микроархитектурой Knights Landing – два АЛУ.

Достижение максимального ускорения при векторной обработке требует учета микроархитектурных параметров процессоров. Например, таких как выравнивание на заданную границу начальных адресов массивов, загружаемых в векторные регистры (32 байта для AVX и 64 байта для AVX-512). А также смешанное использование SSE- и AVX-инструкций. В этом случае при переходе от выполнения команд одного векторного расширения к другому процессор сохраняет (при переходе от AVX к SSE) или восстанавливает (в противоположном случае) старшие 128 бит векторных регистров **`YMM`** (AVX-SSE transition penalties) [7].

Причиной дополнительного ускорения может являться параллельное выполнение векторных инструкций несколькими векторными АЛУ. Таким образом, эффективно векторизо-

ванная версия программы в меньшей степени загружает ряд подсистем суперскалярного конвейера процессора.

Разработчикам прикладных программ доступны следующие способы использования векторных инструкций: ассемблерные вставки; интринсики (intrinsics); SIMD-директивы компиляторов, стандартов OpenMP, OpenACC; языковые расширения и библиотеки; автоматическая векторизация компилятором.

В данной работе внимание уделено последнему подходу. Автоматическая векторизация циклов компиляторами является одной из наиболее значимых методик их оптимизации. Такой способ векторизации не требует значительной модификации прикладных программ и обеспечивает их переносимость на уровне исходного кода между разными архитектурами процессоров.

2. Анализ эффективности подсистем автоматической векторизации циклов в открытых компиляторах

Для оценки эффективности подсистем векторизации в компиляторах GCC C/C++ и LLVM/Clang в работе использовался пакет Extended Test Suite for Vectorizing Compilers (ETSVC) [3], содержащий основные классы циклов, встречающихся в научных приложениях на языке С. Исходная версия пакета была разработана в конце 1980-х годов группой Дж. Донгарры и содержала 122 цикла на языке Fortran для оценки эффективности компиляторов векторных ВС [4, 5]. В 2011 году группа Д. Падуа транслировала пакет TSVC на язык программирования С и дополнила его новыми циклами [2]. Расширенная версия пакета содержит 151 цикл. Циклы разделены на категории: анализ зависимостей по данным (36 циклов), анализ потока управления и трансформация циклов (52 цикла), распознавание идиоматических конструкций (редукции, рекуррентности и т.п., 27 циклов), полнота понимания языка программирования (23 цикла). Кроме этого, в набор включены 13 контрольных циклов – тривиальные циклы, с векторизацией которых должен справиться каждый векторизующий компилятор.

Циклы оперируют с одномерными и двумерными глобальными массивами, начальные адреса которых выравнены на заданную границу (по умолчанию 16 байт). Одномерные массивы содержат **`125·1024/sizeof(TYPE)`** элементов заданного типа **`TYPE`**, а двумерные – 256 элементов по каждому измерению.

Каждый цикл размещен в отдельной функции. Перед выполнением цикла в функции **`init`** выполняется инициализация массивов значениями, характерными для данного теста. Внешний цикл используется для увеличения времени выполнения теста (формирования статистики). Вызов пустой функций **`dummy`** предотвращает нежелательную оптимизацию внешнего цикла (трансформацию и вынесение внутреннего цикла за пределы внешнего, как инвариантного по отношению к внешнему). После выполнения циклов происходит вычисление и вывод на экран контрольной суммы элементов результирующего массива.

Эксперименты проводились на системе, представляющей собой сервер на базе двух процессоров Intel Xeon E5-2620 v4 (архитектура Intel 64, микроархитектура Broadwell, 8 ядер, Hyper-Threading включен, поддержка набора векторных инструкций AVX 2.0), 64 Гбайта оперативной памяти DDR4, операционная система GNU/Linux CentOS 7.3 x86-64 (ядро linux 3.10.0-514.2.2.el7).

Анализировалась работа следующих открытых компиляторов: GCC C/C++ 6.3.0, LLVM/Clang 3.9.1. Компиляция векторизованной версии пакета ETSVC выполнялась с оп-

циями, указанными в табл. 1 (столбец 2). Для генерации скалярной версии теста опции оптимизации сохранялись, но отключался векторизатор компилятора (столбец 3, табл. 1).

Глобальные массивы в пакете ETSVC были выравнены на границу 32 байта. Эксперименты выполнены для массивов с элементами типов **`double`**, **`float`**, **`int`** и **`short int`**.

Для данных типа **`double`** были получены следующие результаты. Для GCC C/C++ общее количество векторизованных циклов составляет 79. При этом 34 из них были векторизованы только им. LLVM/Clang векторизовал 51 цикл, 6 из которых смог векторизовать только он. Количество невекторизованных циклов ни одним из компиляторов составило 66. Результаты векторизации для массивов с элементами типов **`float`**, **`int`** и **`short int`** аналогичны **`double`** для обоих компиляторов.

В категории «Анализ зависимостей по данным» для типа данных **`double`** не были векторизованы ни одним из компиляторов 14 циклов. Проблемными в этой категории оказались циклы, содержащие линейные зависимости (рекуррентности 1-го порядка), непоследовательный доступ к элементам массива, индуктивные переменные в сочетании с условными и безусловными (**`goto`**) переходами внутри цикла, вложенностью циклов и переменными значениями нижней и(или) верхней границы цикла и(или) шага выполнения итераций. В последнем случае на этапе статической компиляции ни один из компиляторов не может принять однозначного решения о наличии зависимости по данным и принимает пессимистическое решение о том, что зависимость существует.

Таблица 1 – Опции, используемые при компиляции

Компилятор	Опции компиляции	Отключение векторизатора
GCC C/C++ 6.3.0	-O3 -ffast-math -fivopts -march=native -fopt-info-vec -fopt-info-vec-missed -fno-tree-vectorize	-fno-tree-vectorize
LLVM/Clang 3.9.1	-O3 -ffast-math -fvectorize -Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize	-fno-vectorize

В категории «Анализ потока управления и трансформация циклов» сложными для векторизации оказались 29 циклов, требующие выполнения следующих преобразований: расщепление тела цикла, перестановка циклов, расщепление вершин в графе зависимостей по данным (для устранения контура в графе и как следствие исключения выходных зависимостей и антизависимостей в цикле [8]) и растягивание скаляров и массивов. Среди причин неудач компиляторов: зависимость значений переменных-счетчиков итераций вложенных циклов друг от друга; линейные зависимости в теле цикла (рекуррентности 1-го порядка); условные и безусловные переходы в теле цикла; охватывающие (wraparound) переменные.

Следующие идиоматические конструкции из категории «Распознавание идиоматических конструкций» оказались проблемными при векторизации 15 циклов: рекуррентности 1-го и 2-го порядков, поиск элемента в массиве, свертка цикла и редукция с вызовом функции. Причина невекторизации циклов, содержащих рекуррентные отношения, заключается в наличии линейной зависимости по данным. В цикле, осуществлявшем поиск первого элемента

в массиве, удовлетворяющего заданному условию, проблема возникла из-за прерывания вычислений в цикле безусловным переходом **goto**.

Над циклами, для которых была выполнена раскрутка (unrolling) вручную, компиляторы выполняют операцию свертки (rerolling) прежде, чем приступить к векторизации [9]. Исследуемые компиляторы приняли решение, что векторизация таких циклов возможна, но будет неэффективной. Причиной этого является использование косвенной адресации при обращении к элементам массива: $X[Y[i]]$, где X – одномерный массив типа **float**, Y – указатель на целочисленный одномерный массив, i – переменная-счетчик итераций цикла.

Еще одна идиоматическая конструкция, вызвавшая проблемы с векторизацией – редукция, а именно нахождение суммы элементов одномерного массива. Здесь причиной невекторизации является наличие вызовов функции **test**, вычисляющей сумму 4-х элементов, начиная с того, который был ей передан в качестве аргумента.

Категория «Полнота понимания языка программирования» содержит 6 невекторизованных ни одним из компиляторов цикла. Проблемы в циклах: прерывание вычислений (вызов функции **exit** или **break**), использование оператора **switch**, условные переходы и косвенная адресация при доступе к элементам массивов. Векторизаторы, реализованные в компиляторах, не смогли выполнить анализ потока управления.

Среди контрольных циклов не были векторизованы ни одним из компиляторов 2, в которых реализовано поэлементное копирование двух одномерных массивов с использованием косвенной адресации.

Максимальное ускорение, полученное при векторизации компилятором GCC C/C++, составило 4.06, 8.1, 12.01 и 24.48 для типов **double**, **float**, **int** и **short int**, соответственно. Компилятором LLVM/Clang получены следующие значения максимального ускорения: 5.12 (**double**), 10.22 (**float**), 4.55 (**int**) и 14.57 (**short int**). Ускорение измерялось как отношение времени выполнения скалярного кода к времени выполнения векторизованного. При этом учитывались только значения ускорения, большие 1.15. Как показал анализ, значения максимального ускорения соответствуют циклам, выполняющим операции редукции (сумма, произведение, поиск минимального или максимального элемента) над элементами одномерных массивов всех типов данных. Эти циклы относятся в ETSVC к категории «Распознавание идиоматических конструкций».

3. Векторизация алгоритма умножения матриц. Распространенным шаблоном вычислений в библиотеках линейной алгебры является гнездо из трех циклов ($i = 0 \dots M - 1$; $j = 0 \dots N - 1$; $k = 0 \dots K - 1$), в котором выполняются арифметические операции над элементами двумерных массивов. Здесь i, j, k – индуктивные переменные, являющиеся счетчиками циклов. Такое гнездо характеризуется наличием инструкций обработки элементов массивов только в самом внутреннем вложенном цикле и пространством итераций, образующим прямоугольный параллелепипед с размерностями M, N, K.

Типичным представителем гнезда из трех циклов является алгоритм умножения матриц GEMM, выполняющий вычисления вида: $A = \alpha BC + \beta A$, где A, B и C – двумерные массивы с размерностями $M \times N$, $M \times K$ и $K \times M$, соответственно, а α и β – скалярные коэффициенты. Для алгоритма умножения матриц по определению $\alpha = \beta = 1$. Если $M = N = K$, A, B и C – квадратные матрицы. Частным случаем GEMM является алгоритм DGEMM, оперирующий матрицами, элементами которых являются числа с плавающей запятой двойной точности (тип данных **double**).

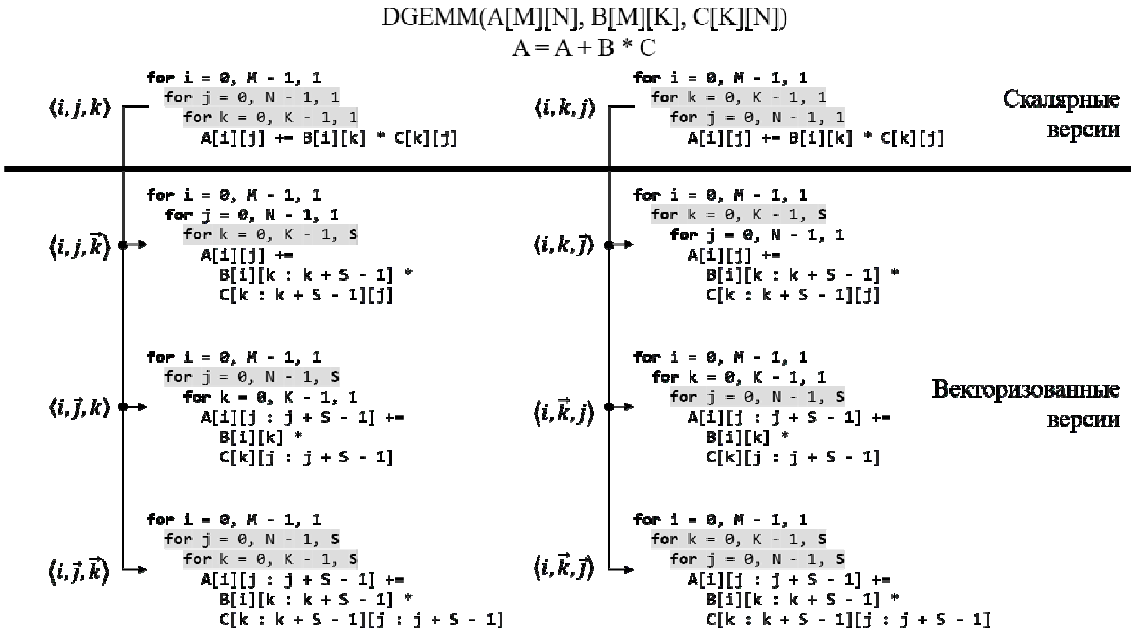

Рис. 1. Скалярные и векторизованные версии алгоритма умножения матриц DGEMM по определению для двумерных массивов **A**[*M*][*N*], **B**[*M*][*K*] и **C**[*K*][*N*] (*S* – количество элементов массивов, помещающихся в векторный регистр)

Алгоритм умножения матриц DGEMM можно реализовать в виде двух последовательных версий (см. рис. 1). В первой версии циклы выполняются в порядке $i \rightarrow j \rightarrow k$, а во второй – в порядке $i \rightarrow k \rightarrow j$. На рис. 1 порядок следования циклов представлен в виде кортежей из трех индуктивных переменных <*i*, *j*, *k*> и <*i*, *k*, *j*>. Каждая из этих версий может быть векторизована тремя способами: 1) только по самому внутреннему вложенному циклу; 2) по среднему вложенному циклу и 3) по обоим этим циклам. На рис. 1 в кортежах индуктивная переменная векторизуемого цикла помечена символом «→». Шаг *S* выполнения итераций для векторизуемых циклов равен количеству элементов типа данных **`double`**, помещающихся в векторный регистр целевой архитектуры вычислительной системы. Например, для архитектуры с короткими векторными регистрами, поддерживающей набор векторных инструкций Intel AVX, $S = 4$.

4. Результаты экспериментов. Проведено экспериментальное исследование эффективности предложенных версий алгоритма умножения матриц. Исследование проводилось на двух ВС с общей памятью. Первая ВС укомплектована двумя процессорами Intel Xeon CPU E5-2620 v3 (микроархитектура Haswell), а вторая – двумя процессорами Intel Xeon E5-2620 v4 (микроархитектура Broadwell). Обе ВС имеют 64 Гбайта ОЗУ.

Для сокращения влияния сторонних факторов на выполнение тестов в ходе экспериментов учитывались особенности NUMA-архитектуры. Запуск процесса, выполняющего тест, производился на том же самом NUMA-узле, на котором происходило выделение памяти.

Графики зависимости ускорения различных реализаций теста DGEMM от количества строк и столбцов *N* используемых матриц приведены на рис. 2 для микроархитектуры Broadwell и на рис. 3 для Haswell. В работе [10] представлены реализации приведенных версий алгоритма DGEMM. Наибольшее ускорение получено версией 1 (рис. 2*a*, 3*a*), достигнув

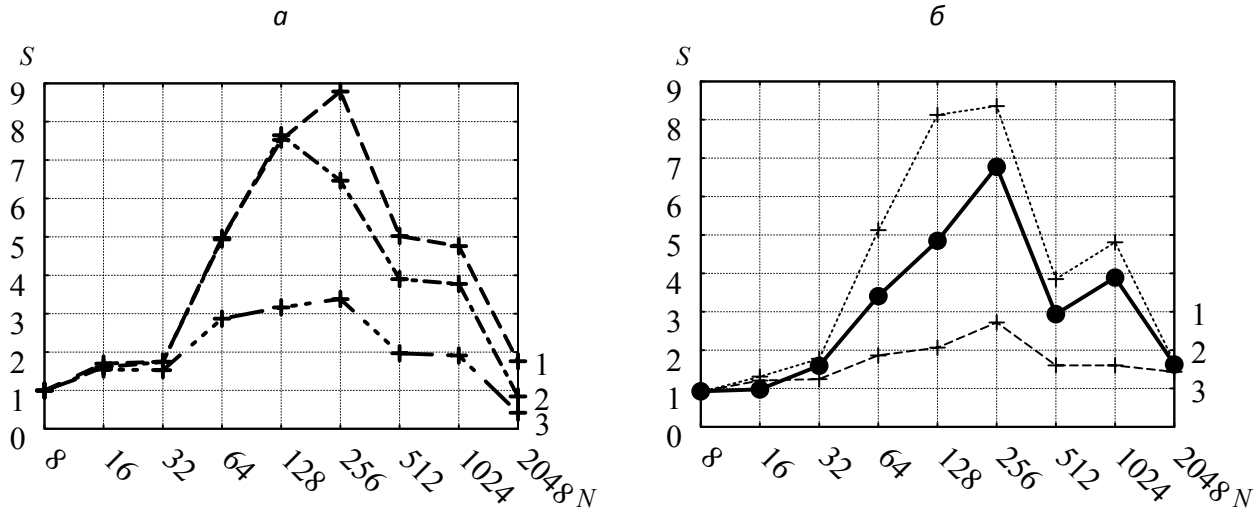

Рис. 2. Ускорение выполнения теста на микроархитектуре Broadwell:

а) версия $< i, j, k >$: 1 – $< i, \vec{j}, \vec{k} >$ с непоследовательным доступом к элементам матрицы C;

2 – $< i, \vec{j}, \vec{k} >$ с использованием команды **`vbroadcastsd`**; *3* – $< i, \vec{j}, k >$;

б) версия $< i, k, j >$: 1 – $< i, \vec{k}, \vec{j} >$ с использованием команды **`vbroadcastsd`**;

2 – $< i, k, \vec{j} >$; 3 – $< i, \vec{k}, \vec{j} >$ с непоследовательным доступом к элементам матрицы C

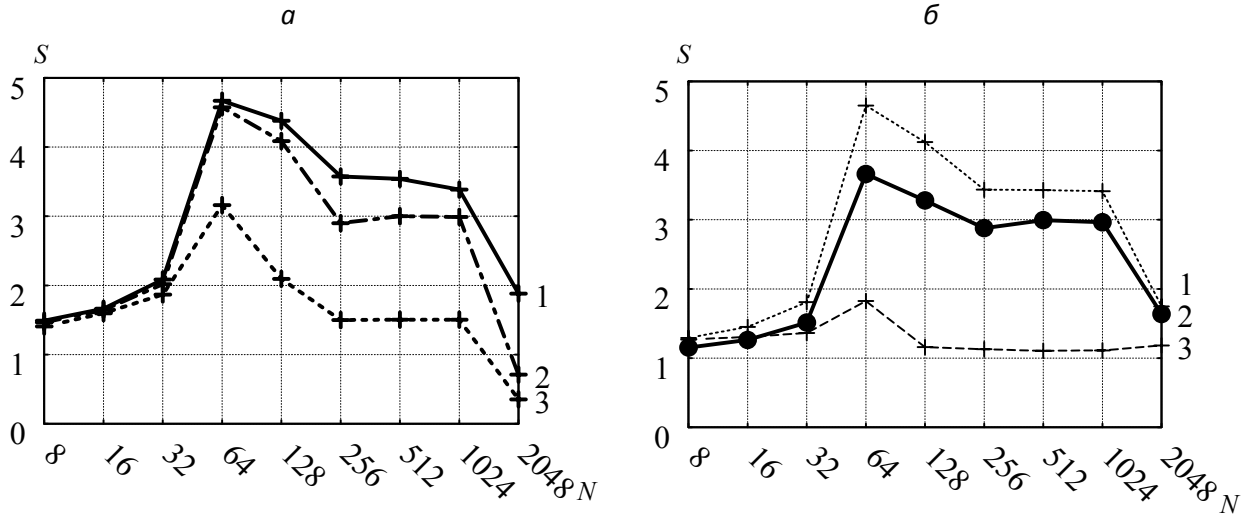

Рис. 3. Ускорение выполнения теста на микроархитектуре Haswell:

а – версия $< i, j, k >$: *1* – $< i, \vec{j}, \vec{k} >$ с непоследовательным доступом к элементам матрицы C;

2 – $< i, \vec{j}, \vec{k} >$ с использованием команды **`vbroadcastsd`**; 3 – $< i, \vec{j}, k >$;

б – версия $< i, k, j >$: *1* – $< i, \vec{k}, \vec{j} >$ с использованием команды **`vbroadcastsd`**;

2 – $< i, k, \vec{j} >$; 3 – $< i, \vec{k}, \vec{j} >$ с непоследовательным доступом к элементам матрицы C

максимального значения при размере массивов 64 × 64 элемента типа **`double`** для микроархитектуры Haswell (S = 4.67), и при 256 × 256 элементов для Broadwell (S = 8.79). В этой версии $< i, \vec{j}, \vec{k} >$ выполнена векторизация циклов j и k. Отличительной особенностью данной реализации является работа с транспонированной матрицей C. Это позволило уменьшить количество кэш-промахов при выполнении загрузки данных из L1 кэша (8684 кэш-промахов при N = 64, микроархитектура Haswell, и 2115816 кэш-промахов при N = 256, микроархитектура Broadwell), а также сократить число операций загрузки и сохранения данных из/в па-

мять (86038/1034 load/store операций при $N = 64$, микроархитектура Haswell, и 5308466/16396 load/store операций при $N = 256$, микроархитектура Broadwell). Детальный отчет, содержащий значения счетчиков производительности основных версий алгоритма умножения матриц, приведен в табл. 2 для микроархитектуры Haswell и в табл. 3 для Broadwell.

Таблица 2 – Значения счетчиков производительности для микроархитектуры Haswell ($N = 64$)

Версия DGEMM	Ускорение	L1d - load-misses	LLC-load-misses	mem-loads	mem-stores	cycles	instructions	CPI	branch-instructions	branch - misses
<*i*, *j*, *k*>: 1 (рис. 4,*а*)	4.67	8684	0	86038	1034	93663	240100	0.39	17501	72
<*i*, *j*, *k*>: 2 (рис. 4,*а*)	4.57	9884	0	100368	1034	97662	203236	0.48	8797	71
<*i*, *j*, *k*>: 3 (рис. 4,*а*)	3.16	19336	0	198678	2060	213945	600612	0.36	66653	1095
<*i*, *k*, *j*>: 1 (рис. 4,*б*)	4.65	8730	0	86038	16394	91136	222693	0.41	17501	72
<*i*, *k*, *j*>: 2 (рис. 4,*б*)	3.66	9643	0	135190	65546	144023	549349	0.26	69725	72
<*i*, *k*, *j*>: 3 (рис. 4,*б*)	1.83	9282	0	328728	65547	483118	1169893	0.41	83037	78

Таблица 3 – Значения счетчиков производительности для микроархитектуры Broadwell ($N = 256$)

Версия DGEMM	Ускорение	L1d - load-misses	LLC-load-misses	mem-loads	mem-stores	cycles	instructions	CPI	branch-instructions	branch-misses
<*i*, *j*, *k*>: 1 (рис. 3,*а*)	8.79	2115816	0	5308466	16396	9850108	14059112	0.7	1065249	267
<*i*, *j*, *k*>: 2 (рис. 3,*а*)	6.46	2143096	0	6324281	16396	13456733	12682856	1.06	532769	275
<*i*, *j*, *k*>: 3 (рис. 3,*а*)	3.38	4263691	3456	12615758	32780	25802907	37914476	0.68	4210981	16901
<*i*, *k*, *j*>: 1 (рис. 3,*б*)	8.36	2118641	0	5308466	1048588	10355924	13780585	0.75	1065249	270
<*i*, *k*, *j*>: 2 (рис. 3,*б*)	6.77	2098595	1826	8454194	4194316	12790121	33949289	0.38	4260129	520
<*i*, *k*, *j*>: 3 (рис. 3,*б*)	2.72	2178667	185	20987997	4194317	32114941	74548847	0.43	5259559	16657

Заключение. Выполнен анализ эффективности подсистем автоматической векторизации циклов в открытых компиляторах GCC C/C++ и LLVM/Clang на архитектуре Intel 64. Установлено, что исследуемые компиляторы способны векторизовать от 34 % до 52 % циклов от общего числа в пакете ETSVC.

Наиболее проблемными оказались циклы, содержащие условные и безусловные переходы, вызовы функций, индуктивные переменные, охватывающие переменные в границах цикла и шаге выполнения итераций, а также такие идиоматические конструкции, как рекуррентности 1-го или 2-го порядка, поиск первого подходящего элемента в массиве и свертка цикла.

Рассмотрено отображение гнезда из трех циклов, в котором выполняются арифметические операции над элементами двумерных массивов, на архитектуру с короткими векторными регистрами Intel AVX. В качестве представителя целевого гнезда циклов использован алгоритм умножения матриц по определению. Исследовано 6 способов векторизации этого алгоритма. С целью нахождения границ применимости указанных векторизованных версий в автоматическом векторизаторе открытых компиляторов проведено экспериментальное исследование эффективности реализованных алгоритмов на процессорах Intel с микроархитектурами Haswell и Broadwell.

Список литературы

1. Вальковский В.А. Распараллеливание алгоритмов и программ. Структурный подход. М.: Радио и связь, 1989. 176 с.

2. Maleki S., Gao Ya. Garzarán M.J., Wong T., Padua D.A. An Evaluation of Vectorizing Compilers // Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT'11), 2011. pp. 372–382.

3. Extended Test Suite for Vectorizing Compilers. [Электрон. ресурс]. URL: http://polaris.cs.uiuc.edu/~maleki1/TSVC.tar.gz (дата обращения 13.07.2017).

4. Callahan D., Dongarra J., Levine D. Vectorizing Compilers: A Test Suite and Results // Proc. of the ACM/IEEE conference on Supercomputing (Supercomputing'88), 1988. pp. 98–105.

5. Levine D., Callahan D., Dongarra J. A Comparative Study of Automatic Vectorizing Compilers // Journal of Parallel Computing. 1991. Vol. 17. pp. 1223–1244.

6. Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard // URL: http://www.netlib.org/blas/blast-forum/blas-report.pdf (дата обращения 15.07.2017).

7. Konsor P. Avoiding AVX-SSE Transition Penalties // URL: https://software.intel.com/en-us/articles/avoiding-avx-sse-transition-penalties (дата обращения 13.07.2017).

8. Векторизация программ: теория, методы, реализация. Сб. работ: пер. с англ. и нем. М.: Мир, 1991. 275 с.

9. Metzger R.C., Wen Zh. Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization. MIT Press. 2000. 219 p.

10. Исходный код разработанных тестов DGEMM [Электрон. ресурс]. URL: https://github.com/Ikulagin/vect_bench (дата обращения: 20.07.2017).

Молдованова Ольга Владимировна – канд. техн. наук, доц. Сибирского государственного университета телекоммуникаций и информатики; ведущ. инженер-программист Института физики полупроводников им. А. В. Ржанова СО РАН;
630102, Новосибирск; e-mail: ovm@sibguti.ru;
Иван Иванович Кулагин – ст. препод. Сибирского государственного университета телекоммуникаций и информатики;
мл. науч. сотр. Института физики полупроводников им. А. В. Ржанова СО РАН;
630090, Новосибирск; e-mail: ivan.i.kulagin@gmail.com;
Михаил Георгиевич Курносов – д-р техн. наук, доц. Сибирского государственного университета телекоммуникаций и информатики;
науч. сотр. Института физики полупроводников им. А.В. Ржанова СО РАН;
630090, Новосибирск; e-mail: mkurnosov@isp.nsc.ru