

ИССЛЕДОВАНИЕ НЕВЕКТОРИЗУЕМЫХ КЛАССОВ ЦИКЛОВ НА SIMD-АРХИТЕКТУРАХ С КОРОТКИМИ ВЕКТОРНЫМИ РЕГИСТРАМИ

О. В. Молдованова, М. Г. Курносов

Сибирский государственный университет телекоммуникаций и информатики, 630102, Новосибирск

УДК 004.272.25

Одной из значимых техник оптимизации циклов является их автоматическая векторизация компилятором — преобразование выполнения итераций обработки массива векторными инструкциями (Intel SSE/AVX/AVX-512, ARM NEON SIMD, IBM Altivec). В данной работе определены основные виды циклов, автоматическая векторизация которых современными компиляторами Intel C/C++, PGI C/C++, GCC C/C++, LLVM/Clang на архитектурах Intel 64 и Intel Xeon Phi затруднена. Для каждого цикла установлен класс его автоматической векторизуемости при заданных типе данных массива и компиляторе. Для выявленных неавтоматически векторизуемых циклов предложены способы их трансформации, обеспечивающие их последующую успешную автоматическую векторизацию компиляторами. В качестве целевого набора циклов использован пакет Extended Test Suite for Vectorizing Compilers.

Ключевые слова: векторизация, компиляторы, циклы, архитектура Intel 64, архитектура Intel Xeon Phi.

Введение

Современные процессоры высокопроизводительных вычислительных систем (ВС) характеризуются поддержкой и активным развитием трех основных форм параллельной обработки информации: параллелизм потоков на уровне процессорных ядер (thread level parallelism), параллелизм инструкций на уровне суперскалярного конвейера ядра (instruction level parallelism), а также параллелизм обработки данных векторными арифметико-логическими устройствами (data level parallelism). Развитию векторных наборов инструкций (Intel AVX, IBM Altivec, ARM NEON SIMD) уделяется значительное внимание со стороны разработчиков процессоров. 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 (полу)автоматической векторизации программ компиляторами в последние десятилетия получили новый виток развития: компиляторная автовекторизация; SIMD-директивы OpenMP и Cilk Plus; языковые расширения Intel ISPC, Sierra; библиотеки C++17 SIMD Types, Boost.SIMD, gSIMD, Cyme.

Цель данной работы — выполнить экспериментальный анализ эффективности подсистем автоматической векторизации циклов в современных компиляторах Intel C/C++ Compiler, GCC C/C++, LLVM/Clang, PGI C/C++, выявить циклы, не векторизованные ни одним из компиляторов, и предложить способы их трансформации, обеспечивающие их последующую успешную автоматическую векторизацию.

Учитывая отсутствие информации о реализуемых коммерческими компиляторами методах векторизации, анализ выполнен методом «черного ящика» — на тестовом наборе циклов из пакета Extended Test Suite for Vectorizing Compilers [1, 2, 3, 4]. Определены классы типовых циклов, автоматическая векторизация которых указанными компиляторами затруднена, и выполнен анализ таких циклов.

1 Наборы векторных инструкций

Наборы команд практически всех архитектур современных процессоров включают поддержку векторных инструкций: MMX/SSE/AVX в архитектурах IA-32 и Intel 64, набор AltiVec в архитектуре Power, NEON SIMD в семействе архитектур ARM, MSA в MIPS. Процессоры, реализующие поддержку векторных 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. 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. В числе важнейших — выравнивание на заданную границу начальных адресов массивов, загружаемых в векторные регистры (32 байта для AVX и 64 байта для AVX-512). Чтение и запись по адресам, не выровненным на заданную границу, выполняется медленнее. Снижение эффективности также может быть обусловлено смешанным использованием 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) [5].

При использовании векторных инструкций возможна ситуация, когда полученное ускорение будет превышать ожидаемое. Например, после векторизации цикла, в котором выполнялось поэлементное суммирование двух массивов, накладные расходы процессора на его выполнение уменьшаются за счет сокращения числа загрузок процессором инструкции сложения из памяти и ее декодирования; числа обращений к памяти за операндами операции сложения; количества вычислений условия завершения цикла (меньше обращений к модулю предсказания переходов процессора).

Кроме этого, причиной дополнительного ускорения может являться параллельное выполнение векторных инструкций несколькими векторными АЛУ. Таким образом, эффективно векторизованная версия программы в меньшей степени загружает ряд подсистем суперскалярного конвейера процессора. Последнее является причиной меньшего энергопотребления процессора при выполнении векторизованной программы по сравнению с ее скалярной версией [6].

Разработчикам прикладных программ доступны следующие способы использования векторных инструкций: ассемблерные вставки; интринсики; SIMD-директивы компиляторов, стандартов OpenMP, OpenACC; автоматическая векторизация компилятором.

В данной работе внимание уделено последнему подходу — автоматической векторизации компилятором. Такой способ векторизации не требует значительной модификации прикладных программ и обеспечивает их переносимость на уровне исходного кода между разными архитектурами процессоров.

2 Набор тестовых циклов

В качестве тестового набора в данной работе использован пакет Extended Test Suite for Vectorizing Compilers (ETSVC) [2], содержащий основные классы циклов, встречающихся в научных приложениях на языке C. 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 BC Cray, NEC, IBM, DEC, Fujitsu и Hitachi [3, 4]. В 2011 году группа Д. Падуа транслировала пакет TSVC на язык программирования C и дополнила его новыми циклами [1]. Расширенная версия пакета содержит 151 цикл. Циклы разделены на категории: анализ зависимостей по данным (36 циклов), анализ потока управления и трансформация цик-

лов (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 $125 \cdot 1024 / \text{sizeof}(\text{TYPE})$ 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.

3 Результаты экспериментов

Эксперименты проводились на двух системах. Первая система — 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). Вторая система — установленный в сервер сопроцессор Intel Xeon Phi 3120A (микроархитектура Knights Corner, 57 ядер, поддержка AVX-512), 6 Гбайт оперативной памяти, MPSS 3.8.

Анализировалась работа следующих компиляторов: Intel C/C++ Compiler 17.0; GCC C/C++ 6.3.0; LLVM/Clang 3.9.1; PGI C/C++ 16.10.

Компиляция векторизованной версии пакета ETSVC выполнялась с опциями, указанными в табл. 1 (столбец 2). Для генерации скалярной версии теста опции оптимизации сохранялись, но отключался векторизатор компилятора (столбец 3, табл. 1).

Таблица 1: Опции компиляции

Компиляторы	Опции компиляции	Отключение векторизации
Intel C/C++ 17.0	-O3 -xHost -qopt-report3 -qopt-report-phase=vec,loop -qopt-report-embed	-no-vec
GCC C/C++ 6.3.0	-O3 -ffast-math -fivopts -march=native -fopt-info-vec -fopt-info-vec-missed	-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
PGI C/C++ 16.10	-O3 -Mvect -Minfo=loop,vect -Mneginfo=loop,vect	-Mnovect

Глобальные массивы в пакете ETSVC были выравнены на границу 32 байта для процессора Intel Xeon и 64 байта — для Intel Xeon Phi. Эксперименты выполнены для массивов с элементами типов `double`, `float`, `int` и `short`.

Для данных типа `double` на архитектуре Intel 64 (процессор Intel Xeon Broadwell) были получены следующие результаты. Компилятор Intel C/C++ Compiler в общей сложности векторизовал 95 циклов, из них 7 смог векторизовать только он. Для GCC C/C++ общее количество векторизованных циклов составляет 79. При этом не было ни одного такого цикла, который был бы векторизован только им. Компилятор PGI C/C++ векторизовал наибольшее количество циклов — 100, из них 13 векторизованы только этим компилятором. Наименьшее количество циклов было векторизовано LLVM/Clang — 52, 4 из которых смог векторизовать только он. Количество не векторизованных циклов ни одним из компиляторов составило 28.

Результаты векторизации для массивов с элементами типов `float` и `int` аналогичны `double` для всех компиляторов. Сопоставимые данные были получены и для типа `short int` в случае использования Intel C/C++ Compiler, GCC C/C++ и LLVM/Clang. Исключение составляет PGI C/C++, который не векторизовал ни одного цикла, оперирующего данными этого типа.

На рис. 1 показаны результаты векторизации циклов компиляторами для типа данных `double` на архитектуре Intel 64. В ячейках таблицы указаны сокращенные обозначения результатов векторизации, которые были сформированы на основании анализа отчетов компиляторов о ходе векторизации для каждого из 151

цикла. В табл. 2 приведена расшифровка этих обозначений. Для других типов данных были получены аналогичные результаты.

ICC	V	IF	V	IF	V	V	D	V	V	V	V	D	RV	M	V	D	D	V	V	D	V	V	V	D	V	D	D	V	D	D	V	D	D	V	D	D	V	D	D	V	D	D	V	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
-----	---	----	---	----	---	---	---	---	---	---	---	---	----	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Рис. 1: Результаты векторизации циклов (архитектура Intel 64, тип данных double)

Таблица 2: Сокращенные обозначения результатов векторизации

V	Loop is vectorized
PV	Partial loop is vectorized (loop fission with succeeding vectorization of obtained loops)
RV	Remainder is not vectorized
IF	Vectorization is possible but seems inefficient
D	Vector dependence prevents vectorization (supposed linear or non-linear data dependence in a loop)
M	Loop is multiversed (multiple loop versions are generated, unvectorized version is selected in runtime)
BO	Bad operation or unsupported loop bound (e.g., <code>sinf</code> or <code>cosf</code> function is used)
AP	Complicated access pattern (e.g., value of iteration count is more than 1)
R	Value that could not be identified as function is used outside the loop (induction variables are present in a loop)
IL	Inner-loop count not invariant (e.g., iteration count of inner loop depends on iteration count of outer loop)
NI	Number of iterations cannot be computed (lower and/or upper loop bounds are set by function's arguments)
CF	Control flow cannot be substituted for a select (conditional branches inside loop)
SS	Loop is not suitable for scatter store (e.g., in case of packing a two-dimensional array into a one-dimensional array)
ME	Loop with multiple exits cannot be vectorized (<code>break</code> or <code>exit</code> are present inside a loop)
FC	Loop contains function calls or data references that cannot be analyzed
OL	Value cannot be used outside the loop (scalar expansion or mixed usage of one- and two-dimensional arrays in one loop)
UV	Loop control flow is not understood by vectorizer (conditional branches inside a loop)
SW	Loop contains a <code>switch</code> statement
US	Unsupported use in statement (scalar expansion, wraparound variables recognition)
GS	No grouped stores in basic block (unrolled scalar product)

В категории «Анализ потока управления и трансформация циклов» сложными для векторизации оказались 11 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 [7]) и растягивание массивов. Среди причин неудач компиляторов: зависимость значений переменных-счетчиков итераций вложенных циклов друг от друга; линейные зависимости в теле цикла (рекуррентные отношения 1-го порядка); условные и безусловные переходы в теле цикла.

Следующие идиоматические конструкции из категории «Распознавание идиоматических конструкций» оказались проблемными при векторизации 6 циклов: рекуррентные отношения 1-го и 2-го порядков, поиск элемента в массиве, свертка цикла и редукция с вызовом функции. Причина не векторизации 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 [8]. 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 неэффективности является наличие вызовов функции `test`, вычисляющей сумму 4-х элементов, начиная с того, который был ей передан в качестве аргумента. Компилятор Intel C/C++ Compiler сообщил в отчете о векторизации о том, что векторизация возможна, но не эффективна. Остальные исследуемые компиляторы указали вызов функции в качестве причины невозможности выполнения векторизации.

Категория «Полнота понимания языка программирования» содержит 2 неэффективизованных ни одним из компиляторов циклов. Проблема обоих циклов — прерывание вычислений в цикле (вызов функции `exit` в первом случае и `break` во втором). Векторизаторы, реализованные в компиляторах, не смогли выполнить анализ потока управления.

На рис. 2 показано суммарное время выполнения теста (всех циклов) для каждого типа данных и компилятора. На рис. 3 и 4 показаны медиана и максимальные значения ускорения выполнения циклов после их векторизации компиляторами. Максимальное ускорение на архитектуре Intel 64 (процессор Intel Xeon Broadwell), полученное при векторизации компилятором Intel C/C++ Compiler, составило 6.96 для данных типа `double`, 13.89 для данных типа `float`, 12.39 для `int` и 25.21 для `short int`. Для GCC C/C++ величина максимального ускорения равна 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`). А для PGI C/C++ эти значения составили 14.6, 22.74, 34.0 и 68.0, соответственно. Ускорение измерялось как отношение времени выполнения скалярного кода к времени выполнения векторизованного.

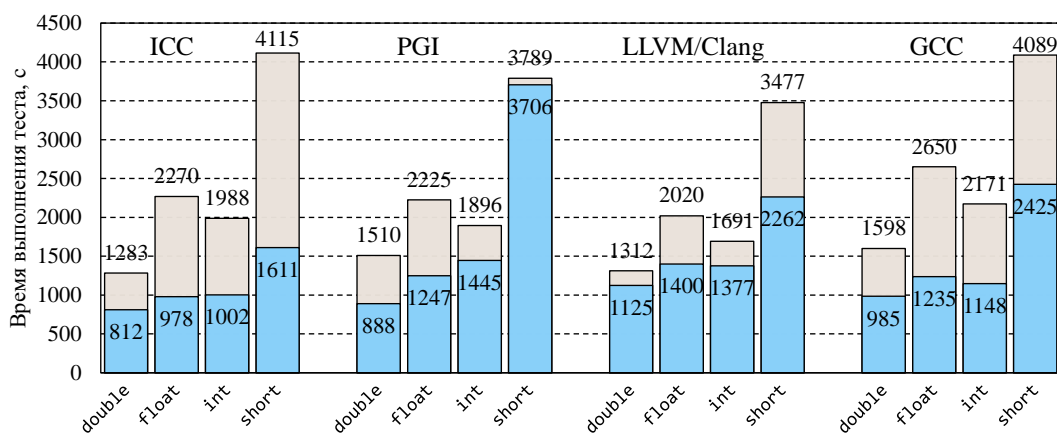


Рис. 2: Время выполнения теста (всех циклов) на процессоре Intel Xeon E5-2620 v4: *внешние столбцы* — неэффективизованная версия теста; *внутренние столбцы* — векторизованная версия теста

Как показал анализ, значения максимального ускорения для компиляторов Intel C/C++ Compiler, GCC C/C++ и LLVM/Clang 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». Для компилятора PGI C/C++ максимальные ускорения для типов `double` и `float` были достигнуты для цикла, формирующего единичную матрицу (категория «Анализ потока управления и трансформация циклов»), а для `int` и `short int` — в цикле из категории «Распознавание идиоматических конструкций», выполняющем операцию редукции (вычисление произведения). Однако, полученное увеличение производительности не всегда является результатом векторизации. Для компилятора PGI C/C++ ускорение 68.0 для типа данных `short int` объясняется тем, что в результате оптимизации вычисления в цикле не производились.

На архитектуре Intel Xeon Phi эксперименты проводились для компилятора Intel C/C++ Compiler 17.0. Вместо опции `-xHost` при компиляции использовалась `-mmic`. Результаты проведенных экспериментов для

В этом случае максимальное ускорение для типа `double` составило 13.7, для типа `float` — 19.43, для `int` — 30.84, а для `short int` — 46.3. Для типов данных `float` и `short int` максимальные значения ускорения были получены для циклов, выполняющих операции редукции над элементами одномерных массивов. Для типа `double` в цикле использовались функции `sinf` и `cosf`. В случае с типом `int` речь идет о контрольном цикле `vbor`, вычисляющем скалярное произведение шести одномерных массивов.

Заключение

Выполнен анализ эффективности подсистем автоматической векторизации циклов в современных компиляторах Intel C/C++ Compiler, GCC C/C++, LLVM/Clang, PGI C/C++ на архитектурах Intel 64 и Intel Xeon Phi. Установлено, что исследуемые компиляторы способны векторизовать от 39 % до 77 % циклов от общего числа в пакете ETSVC. Наилучшие результаты показал Intel C/C++ Compiler, а наихудшие — LLVM/Clang.

Наиболее проблемными оказались циклы, содержащие условные и безусловные переходы, вызовы функций, индуктивные переменные, 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.

Направление дальнейших работ — анализа и разработка методов эффективной векторизации установленного класса проблемных циклов, анализ возможностей применения JIT-компиляции [9] и оптимизации по результатам профилирования (profile-guided optimization).

Список литературы

- [1] 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.
- [2] Extended Test Suite for Vectorizing Compilers URL: <http://polaris.cs.uiuc.edu/~maleki1/TSVC.tar.gz> (дата обращения 29.05.2017).
- [3] 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.
- [4] Levine D., Callahan D., Dongarra J. A Comparative Study of Automatic Vectorizing Compilers // Journal of Parallel Computing. 1991. Vol. 17. pp. 1223–1244.
- [5] Konsor P. Avoiding AVX-SSE Transition Penalties // URL: <https://software.intel.com/en-us/articles/avoiding-avx-sse-transition-penalties> (дата обращения 29.05.2017).
- [6] Jibaja I., Jensen P., Hu N., Haghighat M., McCutchan J., Gohman D., Blackburn S., McKinley K. Vector Parallelism in JavaScript: Language and Compiler Support for SIMD // Proc. of the International Conference on Parallel Architecture and Compilation (PACT-2015). 2015. pp. 407–418.
- [7] Векторизация программ: теория, методы, реализация. Сб. статей: Пер. с англ. и нем. М.: Мир, 1991. 275 с.
- [8] Metzger R.C., Wen Zh. Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization. MIT Press. 2000. 219 p.
- [9] Rohou E., Williams K., Yuste D. Vectorization Technology To Improve Interpreter Performance // ACM Transactions on Architecture and Code Optimization. 2013. 9 (4). pp. 26:1–26:22.

Ольга Владимировна Молдованова — к.т.н., доцент, доцент Кафедры вычислительных систем Сибирского государственного университета телекоммуникаций и информатики;
e-mail: ovm@sibgu.ru;

Михаил Георгиевич Курносов — д.т.н., доцент, заведующий Кафедрой вычислительных систем Сибирского государственного университета телекоммуникаций и информатики;
e-mail: mkurnosov@sibgu.ru.

Дата поступления — 29 мая 2017 г.