Поддержка стандарта OpenMP 4.0 для архитектуры NVIDIA PTX в компиляторе GCC[1]

А.В. Монаков <amonakov@ispras.ru>
В.А. Иванишин <vlad@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В статье описывается реализация стандарта OpenMP версии 4.0 для акселераторов NVIDIA PTX в компиляторе GCC. Особое внимание уделяется вопросам генерации корректного и эффективного кода для прагм OpenMP с учетом ограничений архитектуры PTX. Поскольку реализация опирается на существующую в GCC трансляцию OpenMP и интеграцию с библиотекой libgomp, для PTX реализованы вторичные программные стеки, позволяющие организовать общий для синхронной группы стек в глобальной памяти и передавать адреса на данные в таких стеках между нитями. Описывается схема организации выполнения одной OpenMP-нити в 32 синхронных потоках выполнения в PTX вне OpenMP SIMD-регионов за счет легковесной инструментации некоторых инструкций. Представлены результаты тестирования на микротестах и сравнение с реализацией стандарта OpenACC.

Ключевые слова: компиляторы, GCC, OpenMP, CUDA, PTX.

DOI: 10.15514/ISPRAS-2016-28(4)-10

Для цитирования: А.В. Монаков, В.А. Иванишин. Поддержка стандарта OpenMP 4.0 для архитектуры NVIDIA PTX в компиляторе GCC. Труды ИСП РАН, том 28, вып. 4, 2016, стр. 169-182. DOI: 10.15514/ISPRAS-2016-28(4)-10

1. Введение

Интерфейс программирования OpenMP [1] предоставляет набор расширений для языков C, C++, Fortran в виде аннотаций-прагм, с помощью которых можно описывать параллельные алгоритмы на общей памяти. Важно отметить, что результатом удаления всех прагм из корректного OpenMP-кода будет корректный последовательный код с эквивалентной семантикой. Таким образом обеспечивается возможность инкрементального переноса последовательного кода для многоядерных архитектур и поддержки как последовательной, так и параллельной версий программы в рамках единого исходного кода.

[1] Работа поддержана грантом РФФИ 13-07-12102 офи_м.

В ревизии 4.0 стандарта OpenMP был добавлен ряд новых расширений, предназначенных для выполнения параллельных вычислений на специализированных программируемых акселераторах: x86-совместимых многоядерных процессорах Xeon Phi (Intel MIC) или графических акселераторах (для которых обычно используется интерфейс CUDA или OpenCL).

В настоящее время акселераторы, как правило, подключаются через интерфейс PCI-Express, и поэтому доступ к оперативной памяти основного процессора имеет большую задержку и меньшую пропускную способность; с другой стороны, сами акселераторы имеют специализированную оперативную память с очень высокой пропускной способностью. Чтобы дать программисту возможность эффективно ей распоряжаться, в интерфейсах CUDA и OpenCL есть ряд функций для выделения блоков памяти на акселераторе и копирования между акселератором и хост-процессором. В OpenMP порядок обмена данными между хост-системой и акселератором задается с помощью map-прагм.

Поскольку графические акселераторы не совместимы с хост-процессором по набору команд, компиляторы должны транслировать все участки кода, которые могут выполняться на акселераторе или хост-процессоре (это target-регионы, а также все функции, помеченные прагмой declare target) больше одного раза: не только для архитектуры команд хост-процессора, но и для всех поддерживаемых акселераторных архитектур.

2. Поддержка OpenMP 4.0 в компиляторе GCC

В компиляторе GCC поддержка OpenMP 4.0 была добавлена в версии 4.9 в 2014 году [2], но поддержка акселераторных платформ появилась только в версии 5 с добавлением запуска вычислений на Intel Xeon Phi через библиотеку liboffloadmic. Также в версии 5 была добавлена поддержка интерфейса OpenACC 2.0 для графических акселераторов NVIDIA; генерация кода для GPU выполняется самим GCC: для этого была добавлена новая целевая архитектура nvptx.

В качестве стабильного набора команд, который мог бы использоваться компиляторами для GPU, NVIDIA предложила абстрактный ассемблеро-подобный интерфейс PTX [3]. Таким образом достигается компромисс между необходимостью формата для хранения кода, для которого требуется совместимость с будущими акселераторами, и, с другой стороны, возможностью изменения аппаратного набора инструкций и их кодирования. В отличие от PTX, аппаратный набор команд документирован производителем лишь минимально, в виде, достаточном, например, для чтения дизассемблированного кода (однако в проектах, напрямую выдающих машинный код для акселераторов, таких как Nouveau и maxas [4], за счет обратной разработки стали известны детали кодирования инструкций).

Аналогичный подход используется в таких проектах как PNaCl, где для распространения кода используется представление LLVM IR.

Как и LLVM IR, PTX структурирован: для каждой функции задан ее прототип. Машинные регистры общего назначения не доступны непосредственно: каждая функция декларирует имена и типы регистров, используемых в ней. Таким образом, регистры PTX аналогичны скалярным неадресуемым локальным переменным в Си; соответственно, регистры не теряют свои значения при вызовах (на архитектурах без регистровых окон бинарные соглашения о вызовах для каждого регистра определяют, какая сторона ответственна за его сохранение и восстановление, вызываемая или вызывающая). Явного указателя стека нет, и в связи с этим динамическое выделение стековой памяти (для alloca в Си) не возможно. Статическое выделение стековой памяти возможно за счет объявления локальных объектов в пространстве памяти .local.

При трансляции PTX в машинный код происходит распределение регистров и множество машинно-зависимых оптимизаций: планирование команд, разворачивание циклов и другие.

В наборе команд PTX присутствуют как специализированные команды, специфичные для графических акселераторов (например, инструкция барьерной синхронизации bar.sync), так и часто используемые команды общего назначения, которые не имеют соответствующих аппаратных инструкций и раскрываются в нетривиальную последовательность при трансляции PTX (например, инструкция целочисленного деления div). Почти все инструкции указывают тип обрабатываемых данных с помощью суффикса и поддерживают условное выполнение.

Архитектура графических акселераторов оптимизирована для максимизации производительности вычислений в множестве параллельно работающих нитей; производительность однопоточного кода, запущенного на GPU, была бы невысока. Параллельные контексты выполнения на GPU образуют иерархию. В первую очередь можно выделить синхронные группы (warps), составленные из 32 нитей. Все нити группы имеют общий счетчик команд, так что на каждом шаге все нити выполняют одну и ту же инструкцию; в случаях, когда предикат условного перехода имеет разные значения в нитях группы, выполнение противоположных веток сериализовано: сначала часть нитей с одинаковым значением предиката полностью выполняет свою ветку, затем нити с противоположным значением предиката — свою; это достигается за счет сокращения маски активных нитей. После выполнения обеих веток (в непосредственном постдоминаторе ветвления) маска активных веток восстанавливается и продолжается выполнение синхронной группы. Аналогично обрабатываются вызовы по указателю.

Поскольку у программиста нет возможности вручную определять точки слияния синхронных групп (соответствующие инструкции не присутствуют в PTX и вставляются неявно при трансляции PTX-кода), это приводит к тому, что некоторые программы будут иметь взаимоблокировки. В частности, выполнить

взаимное исключение за счет цикла, выполняющего активное ожидание, невозможно.

Отметим, что на разных уровнях иерархии параллельные нити на GPU имеют разные ограничения и возможности коммуникации. Так, в пределах синхронной группы нити могут выполнять быстрый обмен содержимым регистров с помощью инструкции shfl. На следующем уровне иерархии параллельных нитей находятся блоки. В пределах блока нити имеют доступ к общему блоку быстрой памяти в пространстве .shared и могут выполнять барьерную синхронизацию с помощью инструкции bar.sync.

PTX-код определяет работу одной нити. Для определения положения текущей нити в общей иерархии доступны специальные регистры, например %laneid (номер в синхронной группе).

3. Трансляция OpenMP в GCC

Поддержка OpenMP в GCC разделена между собственно компилятором и библиотекой времени выполнения libgomp, поставляемой вместе с другими компонентами компилятора. Языковые фронт-энды C, C++, Fortran осуществляют парсинг прагм и сохранение их в составе AST-представления; далее одни из самых ранних фаз машинно-независимой трансляции производят анализ прагм при переходе к представлению GIMPLE и сведение их к универсальным конструкциям (например, вызовам функций, условным и циклическим блокам), которые могут обрабатываться последующими фазами компилятора.

В случаях, когда трансляция прагм приводит к добавлению специальных функций, либо когда на уровне исходного кода есть явные вызовы функций из OpenMP API, GCC не встраивает реализации этих функций в объектный код. Все подобные функции реализованы в рамках библиотеки libgomp. Таким образом, публичные функции библиотеки libgomp составляют два пространства имен: функции с префиксом omp_ реализуют функциональность OpenMP API, а функции с префиксом GOMP_ реализуют элементы внутреннего интерфейса между компилятором и libgomp. Например, вход в параллельный регион производится через функцию GOMP_parallel. Отдельно можно отметить функции в пространствах имен GOMP_PLUGIN_ и GOMP_OFFLOAD_, реализующие интерфейсы между основным кодом libgomp и ее модулями-плагинами для различных поддерживаемых акселераторных платформ.

Есть ровно три OpenMP-прагмы, для которых код полного выражения под прагмой переносится компилятором в отдельную функцию: это прагмы parallel, task и target, так как эти прагмы предписывают, что соответствующий код может выполняться не в рамках текущего контекста, а либо в нескольких параллельных нитях (прагма parallel), в любой нити (прагма task), или на акселераторном устройстве (прагма target). Эти функции вызываются через функции GOMP_parallel, GOMP_task и GOMP_target_ext; эти функции libgomp реализуют запуск новых нитей, распределение задач между нитями, запуск

вычислений на акселераторе. Новые функции, выделенные из пользовательского кода, принимают аргумент типа (struct omp_data_s*N* *) — указатель на структуру, содержащую указатели на переменные, которые объявлены вне блока, но используются внутри него (для переменных небольшого размера можно передавать непосредственно их значения вместо указателей, при условии, что внешняя переменная не модифицируется внутри блока: это заведомо так, например, для firstprivate-переменных).

#pragma omp parallel
 	
v = 0;
	omp_data_o.v = &v;
	GOMP_parallel(omp_fn1, &omp_data_o);
	
	void omp_fn1(omp_data_s *omp_data_i)
	{
	*(omp_data_i->v) = 0;
	}

Рис. 1. Выделение OpenMP-региона в отдельную функцию с передачей адреса разделяемой переменной.

Fig. 1. Outlining an OpenMP region into a separate function and passing the address of a shared variable.

4. Вторичные стеки для автоматических переменных в PTX

При добавлении поддержки PTX как акселераторной архитектуры в OpenMP нами было принято решение использовать, где это оправдано, имеющиеся в GCC подходы к трансляции и функциональность libgomp. Это позволяет переиспользовать существующий код, снизить сложность компилятора и поддержки в будущей разработке. Возможны и другие подходы, когда трансляция регионов кода для акселератора выполняется компилятором иначе, чем для хост-процессора: но в этом случае поддержка всего многообразия OpenMP-прагм в target-регионах затруднена.

Как было отмечено, параллельные регионы вырезаются компилятором в отдельные функции, которые получают указатели на общие данные в структуре omp_data_i, передающейся по ссылке. Эта структура располагается на стеке нити, вызвавшей GOMP_parallel. Поскольку в PTX стековые данные располагаются в пространстве памяти .local, это приводит к тому, что другие нити не могут обратиться к этим данным.

Таким образом, реализация стеков в .local-памяти не совместима с предположением GCC, что указатели на стековые данные валидны во всех нитях (стандарт C11 оставляет это как implementation-defined поведение).

Для решения этой проблемы в GCC была реализована поддержка управляемых компилятором стеков, которые могут размещаться в пространстве памяти .global и, таким образом, допускать обмен данными между нитями. Для поддержки указателя на вершину альтернативного стека возможны два подхода:

- хранение указателя на PTX-регистрах и передача как дополнительного аргумента при вызовах;
- хранение указателя в памяти.

Первое решение добавляет небольшие накладные расходы во все нелистовые функции, даже те, которые не обращаются к альтернативному стеку. Второе решение избегает дополнительных расходов в большинстве функций за счет повышения стоимости кода в тех, которые имеют стековый фрейм.

Для хранения указателей на стек используется .shared память. Это существенно упрощает ее резервирование и освобождает от необходимости учитывать номер блока нитей в вычислении адреса указателя текущей нити. Более того, адрес зависит только от номера синхронной группы в блоке: все нити в пределах одной синхронной группы используют один массив в глобальной памяти как стек, и, соответственно, имеют один и тот же указатель на его вершину.

При входе в simd-регион указатель на вершину стека переключается на небольшой регион, выделенный в .local-памяти. За счет этого вызываемые внутри simd-региона функции могут продолжать работать с альтернативным стеком таким образом, что разные нити в одной синхронной группе могут хранить на нем разные данные.

5. Выполнение кода вне SIMD-регионов в синхронных группах

Как отмечалось ранее, OpenMP-нити выполняются синхронными группами, а отдельные PTX-нити в составе синхронной группы могут обрабатывать различные данные только внутри simd-регионов. Соответственно, код вне simd-регионов необходимо выполнять, как если бы в каждой синхронной группе была активна только одна нить, но при этом обеспечить, чтобы при входе в simd-регион все нити синхронной группы могли быть активны и имели одинаковые значения живых регистров. PTX не предоставляет средств для управления маской активных нитей, так что явно активировать нити на входе в simd-регион нельзя: необходимо обеспечить, чтобы они выполнили все переходы от начала выполнения GPU-ядра до входа в регион.

Рассмотрим, что происходит в случае, если код GPU-ядра выполняется нитями синхронной группы, имеющими изначально одинаковое состояние. Инструкции, выполняющие вычисления на регистрах, очевидно, сохраняют этот инвариант: если до выполнения инструкции I нити имели одинаковые значения регистров, то и после выполнения очередной инструкции они будут

совпадать. Инструкции загрузки из памяти также сохраняют этот инвариант: поскольку инструкция выдается синхронно для всех нитей и значение регистра, задающего адрес, совпадает, то и результат загрузки будет одинаковый. Далее, для записи в память PTX гарантирует, что ровно одна из синхронных записей будет успешна, и поскольку все нити записывают одно и то же значение, изменение глобальной памяти будет таким, как если бы запись выполняла одна нить.

Среди инструкций PTX, выдаваемых компилятором, можно выделить два класса инструкций, которые не гарантируют нужной нам инвариантности выполнения: это все атомарные инструкции и команда вызова функции (call). Теоретически, пользовательский код может содержать ассемблерные вставки, которые также могли бы содержать нарушающие инвариантность инструкции, но в настоящее время они не могут возникать в OpenMP target-регионах: для этого синтаксис ассемблерной строки должен быть допустимым как для PTX-ассемблера, так и для хост-архитектуры (обычно x86-64), что невозможно.

Отметим, что сами по себе call-инструкции не нарушают инвариантность: проблема в том, что вызванный код в целом может иметь наблюдаемые эффекты, различные в зависимости от того, активны ли все нити синхронной группы или только одна. Так, вызов vprintf приведет к появлению 31 копии форматированных данных, а вызов malloc выделит 32 блока памяти (и вернет уникальный указатель в каждой нити синхронной группы).

Для вызовов мы можем потребовать, чтобы все вызванные функции сохраняли инвариантность (это можно требовать для всех функций, компилируемых GCC; исключение составляют функции malloc, free, vprintf, на которые полагается реализация libc (newlib) для nvptx).

Для атомарных операций и перечисленных трех функций необходимо обеспечить, чтобы операция была выполнена ровно одной нитью в синхронной группе и после этого вычисленный регистр был распространен в другие нити этой группы (кроме функции free, которая не имеет возвращаемого значения). Для этого достаточно выполнить оригинальную команду под предикатом, истинным только в нулевой нити, и воспользоваться инструкцией shfl для распространения регистра с результатом. Например, если исходный код выполняет инструкцию атомарного обмена (atom.exch.b32 dst, [addr], src), то инструментированный код выглядит так:

```
[pred]  atom.exch.b32   dst, [addr], src
        shfl.idx.b32    dst, dst, 0
```

Инструкции shfl поддерживаются в PTX только для GPU архитектурного уровня sm_30 или выше (архитектура NVIDIA Kepler или более новая). Для акселераторов без поддержки shfl-инструкций можно воспользоваться .shared-

памятью (но это не реализовано: в настоящее время кодогенерация GCC рассчитана на уровень не ниже sm_30).

Предикатный регистр pred можно вычислять перед каждым использованием или в прологе функции, сравнивая индекс нити %laneid с 0. Для call-инструкций инструментация выполняется аналогично.

Отметим, что предложенная схема неявно предполагает, что инструментируемые инструкции сами не имеют контролирующего предиката. В настоящее время GCC не может генерировать инструкции с условным выполнением для nvptx (это может происходить только после распределения регистров, но для nvptx вся последовательность машинно-зависимых преобразований выполняется на псевдорегистрах). Хотя преобразование можно обобщить на случаи, когда исходная команда сама выполняется под предикатом, эффективнее будет запретить выдачу атомарных инструкций и вызовов под предикатом.

Заметим, что хотя предложенный выше подход решает проблему для кода вне simd-регионов, для кода внутри simd-регионов такое преобразование делать нельзя, так как в них побочные эффекты от атомарных инструкций должны происходить независимо во всех активных нитях (причем не все нити синхронной группы могут быть активны). Просто запретить инструментацию для инструкций в simd-регионах было бы недостаточно: если в них есть вызовы других функций, для вызываемой функции также нужно запретить инструментацию, и так далее. Решить это за счет клонирования функций нельзя, так как в общем случае оно невозможно.

Предлагается использовать следующее решение. Изменим инструкцию shfl так, чтобы индекс нити с распространяемым регистром мог быть динамическим:

```
[pred]  atom.op.b32     dst, ...
        shfl.idx.b32    dst, dst, master
```

Как отмечено ранее, вне simd-регионов pred = (%laneid == 0) и master = 0. Внутри simd-регионов необходимо обеспечить, чтобы исходная операция выполнилась во всех нитях, так что pred = 1. Далее, последующая инструкция shfl не должна изменить значение регистра: для этого достаточно master = %laneid.

Чтобы эффективно вычислять pred и master, будем для каждой синхронной группы хранить флаг, указывающий, находится ли выполнение внутри simd-региона. Положим, что вне simd-регионов значение этого флага равно 0, а внутри -1 (т. е. все биты в слове установлены). Тогда можно сначала вычислить master как %laneid & mask (побитовое «и» индекса нити со значением флага), и затем pred как %laneid == master.

Как и для указателя на вершину стека, флаги предлагается хранить в массиве в .shared памяти.

6. Подходы, использованные в реализации OpenACC

Реализация OpenACC в GCC не использует libgomp для организации выполнения параллельных регионов на акселераторе: вместо этого компилятор непосредственно генерирует код, выполняющийся на границах параллельных и векторных регионов. Тем не менее, похожие задачи также решаются и для OpenACC. Описанная в предыдущей секции проблема решается следующим образом. Код вне векторных регионов выполняется только нитью 0 в каждой синхронной группе. Все остальные нити выполняют только переходы на конец очередного базового блока; при этом, если ветвление в конце базового блока является условным, с помощью инструкции shfl выполняется распространение предикатного регистра из нити 0 в остальные. Таким образом достигается, что все нити каждой синхронной группы выполнят переходы между базовыми блоками до входа в векторный регион в той же последовательности, что и нить 0. При достижении векторного региона состояние побочных нитей не синхронизировано с основной нитью, так как они пропускали все базовые блоки. Для синхронизации состояния выполняется копирование содержимого регистров и стекового фрейма. При этом на входе в векторный регион можно скопировать только текущий стековый фрейм, хотя содержимое фреймов вызывающих функций также может использоваться внутри векторного региона.

7. Тестирование реализации на модельных примерах

Для оценки производительности генерируемого с помощью GCC кода для пользовательских программ на OpenMP для акселераторов NVIDIA PTX был создан набор тестов, содержащий модельные примеры, пригодные для предварительной оценки производительности реализации OpenMP. Он включает в себя умножение матрицы на вектор (mul-matr-vec), извлечение квадратного корня методом Ньютона (newton-sqrt), вычисление скалярного произведения двух векторов (scalar-prod) и сложение двух векторов (vector-add). Каждый из этих тестов был реализован с использованием интерфейсов программирования OpenMP, OpenACC и CUDA (являясь более низкоуровневым интерфейсом, чем два других, он представляет для них ориентировочную верхнюю границу производительности).

Ниже приведены графики зависимости времени выполнения от различных параметров запуска (число нитей, блоков, наличие прагмы simd, размер входных данных). В случаях, когда графики для всех четырех тестов имеют типичный вид, приведен один график. Измерения времени, проводимого управлением внутри GPU-ядер, выполнялись с помощью утилиты nvprof, поставляемой в составе CUDA toolkit.

Как видно из графика на рис. 2, накладные расходы на старт параллельного региона составляют около 100 мкс. Для небольших входных данных (размеры вектора до 2^{11} элементов) выгоднее запускать меньшее количество нитей. С

увеличением размеров матрицы прирост производительности от дополнительных нитей начинает перевешивать накладные расходы на их запуск.

Распараллеливание выполнения на 32 контекста (прагма simd) дает ожидаемый прирост производительности, близкий к 32-кратному, для теста newton-sqrt. Это ожидаемо, так как этот тест имеет высокое соотношение числа арифметических опер аций к числу операций с памятью. График справа на рис. 3 типичен для остальных тестов из набора. Сверхлинейное ускорение, вероятно, объясняется увеличением эффективности использования кеша на акселераторе. С другой стороны, на тестах с большой интенсивностью обращений к памяти масштабирование может быть ограничено пропускной способностью памяти, что объясняет снижение достигнутого ускорения до величин, существенно меньших 32, при росте количества блоков и нитей.

На рис. 4 каждая из линий на графике слева отображает зависимость времени выполнения от числа блоков нитей (прагма teams) при фиксированном числе нитей в блоке. Каждая из линий на графике справа показывает зависимость времени выполнения от числа нитей в каждом блоке при фиксированном числе блоков.

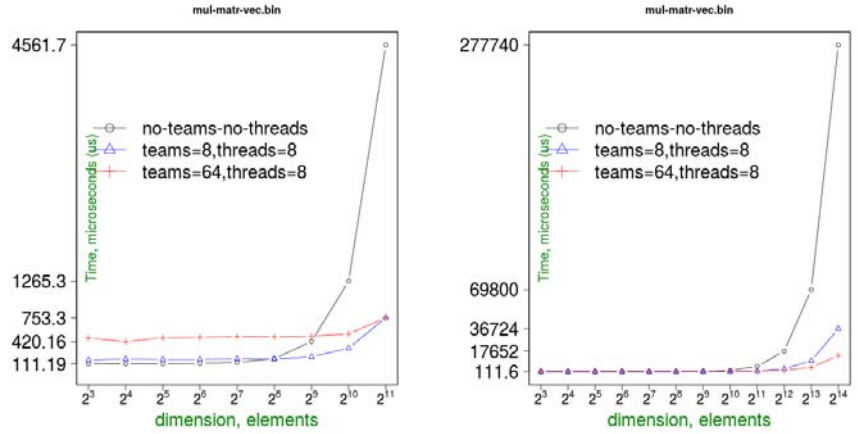

Рис. 2. Зависимости времени выполнения параллельного региона от размерности матрицы для различного числа нитей и блоков.

Fig. 2. Execution time of a parallel region againO9st matrix size, for different thread and block counts.

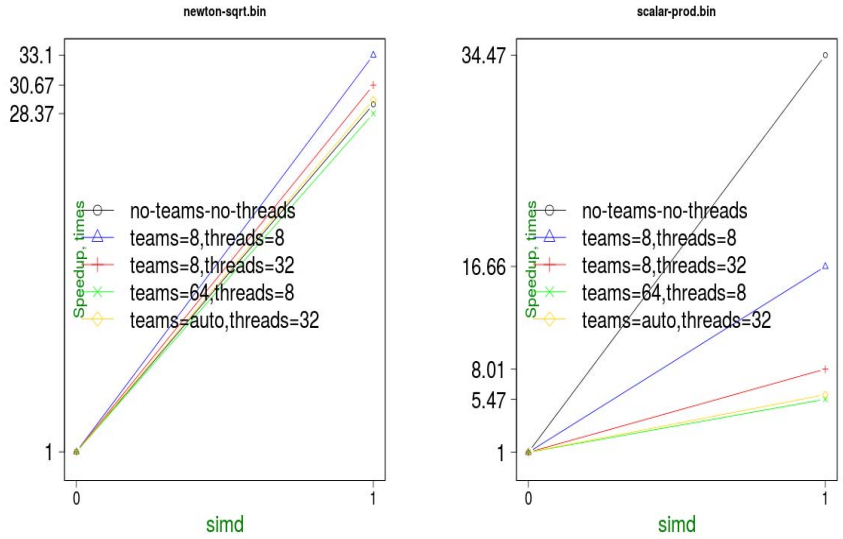

Рис. 3. Ускорение, полученное при включении прагмы simd для различных значений чисел нитей и блоков на тесте newton-sqrt (слева) и scalar-prod (справа).

Fig. 3. Speedup from enabling simd pragma for various numbers of teams and threads per team on newton-sqrt test (left) and the scalar-prod test (right).

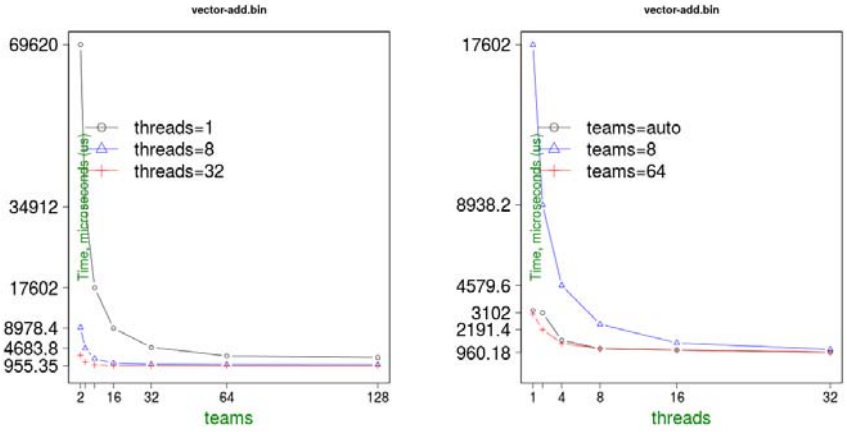

Рис. 4. Зависимости времени выполнения параллельного региона от числа блоков при различных значениях числа нитей и от числа нитей в блоке для различных значений числа блоков.

Fig. 4. Execution time of a parallel region against number of teams, for different threads per team counts (left), and against threads per team, for different team counts (right)

Список литературы

[1]. OpenMP Application Program Interface, 2013. (http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf)
[2]. GCC 4.9 Release Series, 2014. (https://gcc.gnu.org/gcc-4.9/changes.html)
[3]. Parallel Thread Execution ISA, 2016. (http://docs.nvidia.com/cuda/parallel-thread-execution/)
[4]. S. Gray. Assembler for NVIDIA Maxwell architecture, 2016 (https://github.com/NervanaSystems/maxas)

Implementing OpenMP 4.0 for the NVIDIA PTX architecture in GCC compiler

A.V. Monakov <amonakov@ispras.ru>
V.A. Ivanishin <vlad@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. The paper describes the approach used in implementing OpenMP offloading to NVIDIA accelerators in GCC. Offloading refers to a new capability in OpenMP 4.0 specification update that allows the programmer to specify regions of code that should be executed on an accelerator device that potentially has its own memory space and has an architecture tuned towards highly parallel execution. NVIDIA provides a specification of the abstract PTX architecture for the purpose of low-level, and yet portable, programming of their GPU accelerators. PTX code usually does not use explicit vector (SIMD) computation; instead, vector parallelism is expressed via SIMT (single instruction – multiple threads) execution, where groups of 32 threads are executed in lockstep fashion, with support on hardware level for divergent branching. However, some control flow constructs such as spinlock acquisition can lead to deadlocks, since reconvergence points after branches are inserted implicitly. Thus, our implementation maps logical OpenMP threads to PTX warps (synchronous groups of 32 threads). Individual PTX execution contexts are therefore mapped to logical OpenMP SIMD lanes (this is similar to the mapping used in OpenACC). To implement execution of one logical OpenMP thread by a group of PTX threads we developed a new code generation model that allows to keep all PTX threads active, have their local state (register contents) mirrored, and have side effects from atomic instructions and system calls such as malloc happen only once per warp. This is achieved by executing the original atomic or call instruction under a predicate, and then propagating the register holding the result using the shuffle exchange (shfl) instruction. Furthermore, it is possible to setup the predicate and the source lane index in the shuffle instruction in a way that this sequence has the same effect as just the original instruction inside of SIMD regions. We also describe our implementation of compiler-defined per-warp stacks, which is required to have per-warp automatic storage outside of SIMD regions that allows cross-warp references (normally automatic storage in PTX is implemented via .local memory space which is visible only in the PTX thread that owns it). This is motivated by our use of unmodified OpenMP lowering in GCC where possible, and thus using libgomp routines

for entering parallel regions, distribution of loop iterations, etc. We tested our implementation on a set of micro-benchmarks, and observed that there is a fixed overhead of about 100 microseconds when entering a target region, mostly due to startup procedures in libgomp (and notably due to calls to malloc), but for long-running regions where that overhead is small we achieve performance similar to analogous OpenACC and CUDA code.

Keywords: compilers; GCC; OpenMP; CUDA; PTX.

DOI: 10.15514/ISPRAS-2016-28(4)-10

For citation: A.V. Monakov, V.A Ivanishin. Implementing OpenMP 4.0 for the NVIDIA PTX architecture in GCC compiler. *Trudy ISP RAN/Proc. ISP RAS]*, vol. 28, issue 4, 2016. pp. 169-182 (in Russian). DOI: 10.15514/ISPRAS-2016-28(4)-10

References

[1]. OpenMP Application Program Interface, 2013. (http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf)
[2]. GCC 4.9 Release Series, 2014. (https://gcc.gnu.org/gcc-4.9/changes.html)
[3]. Parallel Thread Execution ISA, 2016. (http://docs.nvidia.com/cuda/parallel-thread-execution/)
[4]. S. Gray. Assembler for NVIDIA Maxwell architecture, 2016 (https://github.com/NervanaSystems/maxas)