Microsoft

C++ AMP

Построение массивно параллельных программ с помощью Microsoft Visual C++

Кейт Грегори
Эйд Миллер

ДМК Пресс
издательство

УДК 004.438C++AMP
ББК 32.973.202-018.2
Г79

Г79 Кэйт Грегори, Эйд Миллер

C++ AMP: построение массивно параллельных программ с помощью Microsoft Visual C++. Пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2013. – 412с.: ил.

ISBN 978-5-94074-896-0

C++ Accelerated Massive Parallelism (C++ AMP) – разработанная корпорацией Microsoft технология ускорения написанных на C++ приложений за счет исполнения кода на оборудовании с распараллеливанием по данным, например, на графических процессорах. Модель программирования в C++ AMP основана на библиотеке, устроенной по образцу STL, и двух расширениях языка C++, интегрированных в компилятор Visual C++ 2012. Она в полной мере поддерживается инструментами Visual Studio, в том числе IntelliSense, отладчиком и профилировщиком. Благодаря C++ AMP свойственная гетерогенному оборудованию производительность становится доступна широким кругам программистов.

В книге показано, как воспользоваться всеми преимуществами C++ AMP в собственных приложениях. Помимо описания различных черт C++ AMP, приведены примеры различных подходов к реализации различных алгоритмов в реальных приложениях.

Издание предназначено для программистов, уже работающих на C++ и стремящихся повысить производительность существующих приложений.

УДК 004.438C++AMP
ББК 32.973.202-018.2

ISBN 978-0-7356-6473-9 (англ.)
ISBN 978-5-94074-896-0 (рус.)

ОГЛАВЛЕНИЕ

ГЛАВА 1. Общие сведения и подход C++ AMP

В этой главе:

- Что означает GPGPU? Что такое гетерогенные вычисления?
- Технологии распараллеливания вычислений на ЦП.
- Подход C++ AMP.

Что означает GPGPU? Что такое гетерогенные вычисления?

Разработчикам ПО привычно приспосабливаться к изменяющемуся миру. У нашей индустрии изменение мира стало уже почти рутиной. Мы изучаем новые языки, осваиваем новые методологии, начинаем использовать новые парадигмы человеко-машинного интерфейса и считаем само собой разумеющимся, что программу всегда можно улучшить. Когда на некотором пути мы упираемся в стену и уже не можем сделать версию n+1 лучше версии n, мы находим другой путь. Последним из таких путей, на который готовы встать некоторые разработчики, являются гетерогенные вычисления.

В этой главе мы рассмотрим, что делалось для повышения производительности раньше; это поможет понять, в какую стену мы уперлись сейчас. Вы узнаете об основных различиях между ЦП и ГП, двумя потенциальными компонентами гетерогенного решения, и о том, какие задачи поддаются ускорению с помощью этих приемов распаралле-

ливания. Затем мы рассмотрим применяемые ныне виды параллелизма на ЦП и ГП и познакомимся с концепциями технологии C++ AMP, подготовив почву для детального изучения в последующих главах.

История роста производительности

В середине 70-х годов прошлого века компьютеры, находящиеся в распоряжении одного человека, были в диковинку. Термин «персональный компьютер» появился в 1975 году. За прошедшие с тех пор десятилетия идея компьютера на каждом рабочем столе перестала восприниматься как амбициозная и, быть может, недостижимая цель, а превратилась в обыденность. Теперь на многих столах стоит даже *несколько* компьютеров, да не только в кабинетах, а и в гостиных. Многие носят в кармане смартфоны – тоже компьютеры, хоть и совсем маленькие. За первые 30 лет экстенсивного роста компьютеры не только стали дешевле и популярнее, но и быстрее. Каждый год производители выпускали кристаллы со все более высокой тактовой частотой, с большим объемом кэш-памяти и, как следствие, более производительные. У разработчиков вошло в привычку добавлять в программы всё новые и новые функции. И если из-за этого программа начинала работать медленнее, то разработчики не особо переживали; все равно через полгода-год появятся более быстрые машины, и программа снова станет «шустрой» и отзывчивой. Это было время так называемых «бесплатных завтраков», когда наращивание функциональности программ обеспечивалось повышением производительности оборудования. В конечном итоге производительность порядка гигафлопс – миллиардов операций над числами с плавающей точкой в секунду – стала досягаемой и экономически доступной.

К сожалению, примерно в 2005 году «бесплатные завтраки» кончились. Производители продолжали увеличивать количество транзисторов на одном кристалле, но физические ограничения – в частности, тепловыделение кристалла – уже не дают повышать тактовую частоту. Но рынок – как всегда – требовал всё более и более мощных компьютеров. Для удовлетворения спроса производители стали выпускать многоядерные машины – с двумя, четырьмя и более процессорами. Когда-то цель «каждому пользователю по процессору» считалась труднодостижимой, но по завершении эры бесплатных завтраков пользователям стало недостаточно одноядерного компьютера – сначала настольного, потом – ноутбука, а теперь уже и смартфона. В последние пять-шесть лет стало обычным делом иметь параллельный

суперкомпьютер на каждом рабочем столе, в каждой гостиной и в каждом кармане.

Но одно лишь добавление процессорных ядер ничего не ускоряет. Программы можно грубо разделить на две большие группы: поддерживающие и не поддерживающие параллелизм. Программа без поддержки параллелизма обычно задействует лишь половину, четверть или одну восьмую часть имеющихся ядер. Она ютится на единственном ядре, упуская возможность ускорить работу, когда пользователь приобретает новую машину с большим числом ядер. Разработчики же, научившиеся писать программы, работающие тем быстрее, чем больше имеется процессорных ядер, могут обеспечить почти линейный прирост быстродействия – вдвое на двухъядерных машинах, вчетверо на четырехъядерных и т. д. Информированные потребители недоумевают, почему некоторые разработчики игнорируют дополнительные возможности повысить производительность программ.

Гетерогенные платформы

В те же пять-шесть лет, на которые пришелся расцвет многоядерных компьютеров с несколькими процессорами, не стояли на месте и производители графических карт. Но вместо двух или четырех ядер, как в ЦП, графические процессоры (ГП) оснащались десятками, а то и сотнями ядер. Эти ядра сильно отличаются от имеющихся в ЦП. Первоначально они разрабатывались для повышения скорости вычислений, специфичных для машинной графики, например для определения цвета конкретного пикселя на экране. ГП справляется с этой работой гораздо быстрее ЦП, а поскольку на современной графической карте графических процессоров так много, открывается возможность массивного параллелизма. Разумеется, очень быстро возникло непреодолимое желание приспособить ГП для численных расчетов, *не относящихся* к графике. Машина, оснащенная многоядерными ЦП и ГП, на одном или на разных кристаллах, или даже кластер подобных машин называется гетерогенным суперкомпьютером. Очевидно, очень скоро гетерогенные суперкомпьютеры окажутся на каждом рабочем столе, в каждой гостиной и в каждом кармане.

В начале 2012 года типичный ЦП имел четыре ядра с двойной гиперпоточностью и насчитывал примерно миллиард транзисторов. Компьютеры высшего класса могут достигать при вычислениях с двойной точностью пиковой производительности 0,1 терафлопс (100 гигафлопс). Типичный ГП в начале 2012 года имел 32 ядра с 32 ни-

тями[1] в каждом и примерно вдвое больше транзисторов, чем ЦП. ГП высшего класса могут достигать при вычислениях с одинарной точностью производительности 30 терафлопс – в 30 раз больше пиковой производительности ЦП.

Примечание. Одни ГП поддерживают вычисления с двойной точностью, другие – нет, но данные о производительности обычно приводятся для вычислений с одинарной точностью.

Причина такой высокой производительности ГП не в количестве транзисторов или даже ядер. Дело в пропускной способности памяти – у ЦП она составляет около 20 гигабайт в секунду (ГБ/с), а у ГП – 150 ГБ/с. ЦП рассчитан на исполнение кода общего вида – с многозадачностью, вводом/выводом, виртуализацией, многоуровневым вычислительным конвейером и произвольной выборкой. Напротив, ГП проектируются для исполнения кода обработки графических данных, с распараллеливанием по данным, оснащаются программируемыми процессорами с фиксированными функциями, одноуровневым вычислительным конвейером и ориентированы на последовательную выборку. На самом деле, сверхвысокой производительности ГП достигают только на тех задачах, на которые рассчитаны, а не на задачах общего вида. У ГП есть еще одна особенность, быть может, даже более важная, чем быстродействие, – низкое энергопотребление. Для ЦП характерно энергопотребление порядка 1 Гфлопс/вт, а для ГП – примерно 10 Гфлопс/вт.

Во многих приложениях мощность, необходимая для выполнения конкретного вычисления, важнее затрачиваемого времени. Например, портативные устройства –смартфоны и ноутбуки – работают от аккумулятора, поэтому пользователи зачастую выбирают не самые быстрые, а самые энергетически экономичные приложения. Это существенно и для ноутбуков, пользователи которых ожидают, что при эксплуатации приложений, выполняющих большой объем вычислений, заряда аккумулятора хватит на целый день. Становится нормой ожидать нескольких ЦП – и ГП тоже – даже на таких небольших устройствах, как смартфоны. Некоторые устройства умеют включать и отключать питание отдельных ядер для продления срока работы от аккумулятора. В таких условиях перенос части вычислений на ГП может означать разницу между «приложением, которое невозможно

1 В программировании общего назначения слово thread обычно переводится как «поток», но в контексте программирования ГП чаще употребляется термин «нить» во избежание конфликтов со словом stream. В дальнейшем мы будем придерживаться именно этой терминологии. *Прим. перев.*

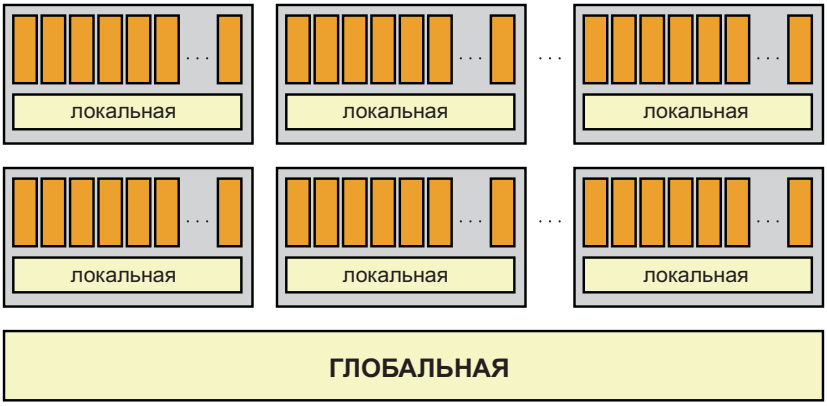
использовать вне офиса, потому что “жрет” батарейку» и «приложением, без которого я жизни себе не представляю». На другом конце спектра находятся центры обработки данных, для которых значительная доля эксплуатационных затрат приходится на оплату энергоснабжения. Двадцатипроцентная экономия энергии при выполнении сложного расчета в ЦОД или в облаке напрямую отражается на счете за электроэнергию.

Следует также учитывать доступ к памяти со стороны процессорных ядер. С точки зрения быстродействия, размер кэш-памяти может оказаться важнее тактовой частоты, поэтому ЦП оснащаются большими кэшами, чтобы у процессора всегда были данные для обработки и ядру как можно реже приходилось ждать завершения выборки данных из памяти. Для ЦП характерно повторное обращение к одним и тем же данным, что и позволяет получить от кэширования реальный выигрыш. Напротив, у типичного ГП кэш-память невелика, зато имеется много нитей, часть из которых всегда готова к работе. ГП может осуществлять предвыборку данных для компенсации задержки памяти, но поскольку данные, скорее всего, будут использоваться однократно, кэширование дает меньший выигрыш и не так необходимо. Чтобы от такого подхода была польза, в идеале должно быть очень много данных, над которыми производится относительно простое вычисление.

Но, пожалуй, самое важное различие заключается в технологии программирования. Для программирования ЦП существует много давно и прочно зарекомендовавших себя языков и инструментов. С точки зрения выразительной мощи и производительности, C++ стоит на первом месте, предлагая разработчику абстракции и развитые библиотеки, но не отнимая возможность низкоуровневого контроля. Выбор средств универсального программирования ГП (general-purpose GPU programming – GPGPU) куда более ограничен, и в большинстве случаев подразумевает нишевую или экзотическую модель программирования. Из-за этого ограничения лишь в немногих областях и задачах удавалось в полной мере задействовать способность ГП к «перемалыванию чисел». По той же причине большая часть разработчиков не стремилась изучать взаимодействие с ГП. Но разработчикам необходим способ повысить быстродействие приложений или сократить энергопотребление в конкретном вычислении. Сегодня таким способом может стать использование ГП. Идеальным было бы такое решение, которое позволило бы воспользоваться преимуществами ГП сегодня и другими формами гетерогенных вычислений – в будущем.

Архитектура ГП

Выше уже отмечалось, что ГП оснащен одноуровневым вычислительным конвейером, небольшим кэшем и большим числом нитей, выполняющих последовательную выборку из памяти. Нити не являются независимыми, а объединены в группы. В продуктах компании NVIDIA эти группы называются *канатами* (warps), а в продуктах AMD – *волновыми фронтами* (wavefront). В этой книге мы будем употреблять термин «канат». Канаты работают совместно, могут сообща обращаться к одной области памяти и взаимодействовать между собой. Для чтения локальной памяти требуется всего четыре такта, а для чтения более объемной (до 4 Гб) глобальной памяти – от 400 до 600 тактов. Когда одна группа нитей блокирована в ожидании результатов чтения, может исполняться другая группа. ГП способен очень быстро переключаться между группами нитей. Доступ к памяти организован таким образом, что чтение производится гораздо быстрее, когда соседние нити обращаются к соседним ячейкам. Если же отдельные нити в группе обращаются к ячейкам памяти, далеко отстоящим от тех, которые читают остальные нити в той же группе, то производительность резко падает.



Архитектуры ЦП и ГП существенно различаются. Программисты, работающие на языках высокого уровня, обычно не задумываются об архитектуре ЦП. Компоненты низкого уровня, операционные системы и оптимизирующие компиляторы обязаны принимать во внимание эти факторы, но при этом они ограждают «обычные» приложения от аппаратных деталей. Рекомендации и эвристические правила, которые вам, возможно, кажутся очевидными, иногда вовсе не являются

таковыми; даже для ЦП простое сложение целых чисел, приводящее к промаху кэша, может занимать гораздо больше времени, чем операция чтения с диска, удовлетворенная из буферизованного содержимого файла, которое находится в ближнем кэше.

Некоторые разработчики, создающие приложения с повышенными требованиями к производительности, должны учитывать, сколько команд можно выполнить за время, потерянное в случае промаха кэша, или сколько тактов требуется для чтения одного байта из файла (во многих случаях миллионы). В настоящее время от необходимости учитывать это никуда не деться, особенно при работе с архитектурами, отличными от ЦП (например, ГП). Пока еще не существует уровней защиты, сравнимых с теми, что компиляторы и операционные системы предоставляют при программировании ЦП. Например, иногда требуется знать, сколько нитей в канате или каков размер разделяемой ими кэш-памяти. Возможно, придется организовать вычисление, так чтобы на каждой итерации производились обращения к соседним ячейкам памяти, избегая произвольной выборки. Чтобы оценить, какого ускорения можно ожидать, необходимо знакомство с аппаратной архитектуртой – по крайней мере, на концептуальном уровне.

Кандидаты на повышение производительности за счет распараллеливания

ГП лучше приспособлен для задач, распараллеливаемых по данным. Иногда с первого взгляда очевидно, как разбить большую задачу на много мелких, которые можно решать параллельно и независимо. Взять, к примеру, сложение матриц: каждый элемент результирующей матрицы можно вычислять независимо от остальных. Для сложения двух матриц размером 100×100 потребуется 10 000 операций сложения, но если бы удалось распределить их между 10 000 нитей, то все операции можно было выполнять одновременно. Сложение матриц – естественно параллельная задача.

В других случаях необходимо специально придумывать алгоритм, который мог бы выполняться независимыми нитями. Рассмотрим задачу нахождения наибольшего значения в большом списке чисел. Можно было бы перебирать элементы по одному, сравнивая каждый с «текущим наибольшим» и обновляя «текущий наибольший» при обнаружении большего значения. Если список содержит 10 000

элементов, то потребуется 10 000 сравнений. Можно поступить иначе – создать несколько нитей и поручить каждой обрабатывать какую-то часть списка. Каждая из 100 нитей могла бы обработать по 100 элементов и найти среди них наибольший. Таким образом, для вычисления частичного максимума потребовалось бы столько времени, сколько занимают 100 операций сравнения. И наконец, 101-ая нить могла бы сравнить 100 найденных «частичных максимумов» и выбрать среди них наибольший. Изменяя количество нитей и, следовательно, количество операций сравнения, выполняемых каждой нитью, можно минимизировать время поиска наибольшего элемента в списке. Если сравнение обходится гораздо дороже, чем создание нитей, то можно пойти на крайнюю меру: создать 5000 нитей, каждая из которых будет сравнивать два числа, затем 2500 нитей для сравнения победителей первого круга, затем 1250 нитей для победителей второго круга и т. д. При таком подходе для нахождения наибольшего значения понадобится 14 кругов, а общее время равно времени 14 операций сравнения плюс накладные расходы. Такой «турнирный» подход можно применить и к другим операциям, например, сложению всех элементов коллекции, подсчету количества элементов в заданном диапазоне и т. д. В применении к классу задач, в которых для заданного большого набора данных ищется одно число (сумма, минимум, максимум и т. п.), часто употребляется термин *редукция*.

Оказывается, что любая задача, связанная с обработкой больших объемов данных, – естественный кандидат на распараллеливание. Раньше всего этот подход нашел применение в следующих областях.

- **Аналитическое и имитационное моделирование в различных науках.** В физике, биологии, биохимии и других отраслях знания очень сложные ситуации с гигантскими объемами данных описываются простыми уравнениями. Чем больше данных участвует в расчете, тем точнее результаты моделирования. Но проверка теории на модели возможна, только если моделирование удается завершить в разумное время.
- **Системы управления реального времени.** Сбор данных с многочисленных датчиков, определение параметров, вышедших за пределы допустимого диапазона, и восстановление оптимального режима работы с помощью управляющих воздействий – весьма ответственные процессы. Пожары, взрывы, дорогостоящие отключения и даже гибель людей – вот что пытаются предотвратить такого рода программы. Обычно количество

датчиков ограничено временем, необходимым для обработки полученных от них данных.

- **Мониторинг, моделирование и прогнозирование финансовой ситуации.** Часто для выявления трендов или открывающихся возможностей для извлечения прибыли требуются очень сложные вычисления с огромными объемами данных. Но обнаруживать возможности следует, пока они еще существуют, что налагает жесткие ограничения на максимальное время вычислений.
- **Компьютерные игры.** Большинство игр по существу представляют собой модель реального или тщательного продуманного иного мира с другими физическими законами. Чем больше данных удается включить в модель, тем правдоподобнее выглядит игра. Но при этом ни в коем случае нельзя жертвовать быстродействием.
- **Обработка изображений.** Обнаружение аномалий в медицинских изображениях, распознавание лиц на видеозаписи, отснятой камерой наблюдения, сопоставление отпечатков пальцев – во всех этих и многих аналогичных задачах требуется избежать ложноположительных и ложноотрицательных ответов при том, что время, отведенное для решения задачи, ограничено.

Во всех перечисленных случаях десятикратное увеличение скорости обработки числовых данных открывает одну из двух возможностей. Самое простое – увеличить объем данных, не увеличивая времени его обработки. Обычно это означает, что результаты окажутся точнее или что у конечного пользователя будет больше уверенности в правильности принимаемых решений. Интереснее, однако, ситуации, когда ускорение расчетов позволяет сделать нечто такое, что раньше было невозможно. Например, если финансовые расчеты, на которые раньше уходило 20 часов, реально завершить всего за два часа, то их можно выполнять ночью, когда биржи закрыты, а утром люди смогут предпринять те или иные действия на основе полученных результатов. А если бы удалось добиться стократного ускорения? Если некий расчет занимал 1000 часов (свыше 40 дней), то к моменту завершения исходные данные для него вполне могли устареть. Если же удастся выполнить его за 10 часов – ночью, то шансы на осмысленность результатов резко повысятся.

Наличие временных окон характерно отнюдь не только для финансовых программ. Аналогичные ограничения есть в системах охраны,

обработке медицинских изображений и многих других приложениях, в том числе предназначенных для взлома паролей и добычи данных. Если для подбора пароля прямым перебором требуется 40 дней, а вы меняете пароль каждые 30 дней, то ваш пароль в безопасности. Но что, если для взлома достаточно всего 10 часов?

Добиться десятикратного ускорения работы относительно просто, стократного – гораздо сложнее. Проблема не в том, что ГП на это не способен, а в том, что в любом приложении имеются части, не допускающие распараллеливания.

Возьмем три приложения. Каждому для решения некоторой задачи требуется 100 условных единиц времени. В первой не распараллеливаемая часть (скажем, отправка отчета на принтер) занимает 25 % общего времени. Во втором – только 1 %, а в третьем – 0,1 %. Что произойдет, если ускорить распараллеливаемую часть каждого приложения?

		Прог1	**Прог2**	**Прог3**
	% последовательного кода	25 %	1 %	0,1 %
Исходное	Последовательная часть	25	1	0,1
	Параллельная часть	75	99	99,9
	Общее время	100	100	100
x 10	Последовательная часть	25	1	0,1
	Параллельная часть	7,5	9,9	9,99
	Общее время	32,5	10,9	10,09
	Ускорение	**3,08**	**9,17**	**9,91**
x 100	Последовательная часть	25	1	0,1
	Параллельная часть	0,75	0,99	0,999
	Общее время	25,75	1,99	1,099
	Ускорение	**3,88**	**50,25**	**90,99**
Бесконечно	Последовательная часть	25	1	0,1
	Параллельная часть	0	0	0
	Общее время	25	1	0,1
	Ускорение	**4**	**100**	**1000**

При десятикратном ускорении параллельной части первое приложение проводит гораздо больше времени в последовательной части, чем в параллельной. Общий коэффициент ускорения немного больше 3. Стократное ускорение параллельной части помогает не силь-

но – из-за существенного преобладания последовательной части. Даже при бесконечном ускорении, когда время выполнения параллельной части равно 0, снизить влияние последовательной части не удается и общий коэффициент ускорения составляет всего 4. Остальные две программы выигрывают от десятикратного ускорения больше, но даже при стократном ускорении общее время работы второго приложения уменьшается всего в 50 раз, а при бесконечном ускорении – только в 100 раз.

Этот кажущийся парадокс – тот факт, что вклад последовательной части в конечном итоге определяет максимальное общее ускорение, какой бы малой ни была его первоначальная доля, – известен под названием закона Амдала. Это не означает, что стократное ускорение невозможно, но говорит о том, насколько важен выбор алгоритма, минимизирующего время выполнения не распараллеливаемой части. Кроме того, использование алгоритма распараллеливания по данным, открывающего возможность применения GPGPU для ускорения работы приложения, может дать больший выигрыш, чем выбор очень быстрого и эффективного алгоритма, который по своей природе последователен и не допускает распараллеливания. Решение, подходящее для задачи с миллионом точек, может оказаться непригодным, когда число точек увеличится до 100 миллионов.

Технологии распараллеливания вычислений на ЦП

Один из способов уменьшить время, проводимое в последовательной части приложения, – сделать его менее последовательным, то есть перепроектировать, воспользовавшись параллелизмом как ЦП, так и ГП. Хотя на ГП могут одновременно работать тысячи нитей, а на ЦП гораздо меньше, задействование параллелизма ЦП все же вносит вклад в общее ускорение работы. В идеале технологии, применяемые для распараллеливания вычислений на ЦП и ГП, должны быть совместимы. И тут возможно несколько подходов.

Векторизация

Один из важных способов ускорить обработку состоит в том, чтобы применить технологию SIMD (Single Instruction, Multiple Data – одна команда, много данных). В типичном приложении команды вы-

бираются по одной за раз и исполняются в соответствии с потоком управления внутри приложения. Но при выполнении масштабной операции с распараллеливанием по данным, например сложения матриц, одни и те же команды (сложение элементов матриц, являющихся целыми числами или числами с плавающей точкой) повторяются снова и снова. Это означает, что стоимость выборки команд можно распределить между большим количеством операций благодаря применению одной и той же команды к разным данным (например, разным элементам матрицы). Тем самым мы сможем резко увеличить быстродействие либо сократить расход энергии на выполнение вычислений.

Под векторизацией понимается преобразование программы из формы, при которой обрабатывается по одному элементу данных за раз (и каждый раз новыми командами), в форму, при которой сразу обрабатывается вектор данных, причем к каждому его элементу применяются одни и те же команды. Некоторые компиляторы умеют автоматически применять такое преобразование к циклам определенного вида и к другим допускающим распараллеливание операциям.

Microsoft Visual Studio 2012 поддерживает ручную векторизацию с помощью встроенных функций SSE (Streaming SIMD Extensions). Встроенные функции выглядят как обычные функции, но на самом деле напрямую отображаются на последовательности ассемблерных команд, поэтому с ними не связаны накладные расходы на вызов функции. В отличие от встроенного ассемблерного кода, о встроенных функциях компилятор знает и может соответственно оптимизировать другие части кода. В смысле переносимости встроенные функции лучше, чем встроенный ассемблерный код, но все равно подвержены проблемам, так как зависят от наличия определенных команд в целевом процессоре. Разработчик должен быть уверен, что процессор машины, на которой будет исполняться программа, поддерживает используемые встроенные функции. Неудивительно, что для этой цели имеется встроенная функция: *__cpuid()* генерирует команды, помещающие в четыре целых числа информацию о возможностях процессора (имя начинается с двух знаков подчеркивания, чтобы показать, что это внутреннее средство компилятора). Чтобы проверить, поддерживается ли набор команд SSE3, нужно написать такой код:

```
int CPUInfo[4] = { -1 };
__cpuid(CPUInfo, 1);
bool bSSEInstructions = (CPUInfo[3] >> 24 & 0x1);
```

Примечание. Полная документация по *__cpuid()*, где в частности объясняется, почему второй параметр равен 1 и какой бит нужно проверять, чтобы узнать о поддержке SSE3, имеется в разделе MSDN по адресу *http://msdn.microsoft.com/en-us/library/hskdteyh(v=vs.100).aspx*.

Какую встроенную функцию использовать, зависит от того, каким образом вы собираетесь распараллелить программу. Рассмотрим случай, когда требуется сложить много пар чисел. Одна встроенная функция *mm_hadd_epi32* складывает за раз четыре пары 32-разрядных чисел. Вы должны поместить входные данные в два выровненных в памяти 128-разрядных числа, после чего вызвать функцию. В результате получится 128-разрядное число, которое можно разбить на четыре 32-разрядных, представляющих суммы каждой пары. Вот пример кода, взятый из MSDN:

```
#include <stdio.h>
#include <tmmintrin.h>
int main ()
{
 __m128i a, b;
 a.m128i_i32[0] = -1;
 a.m128i_i32[1] = 1;
 a.m128i_i32[2] = 0;
 a.m128i_i32[3] = 65535;
 b.m128i_i32[0] = -65535;
 b.m128i_i32[1] = 0;
 b.m128i_i32[2] = 128;
 b.m128i_i32[3] = -32;

 __m128i res = _mm_hadd_epi32(a, b);

 std::wcout << "Original a: " <<
 a.m128i_i32[0] << "\t" << a.m128i_i32[1] << "\t" <<
 a.m128i_i32[2] << "\t" << a.m128i_i32[3] << "\t" << std::endl;
 std::wcout << "Original b: " <<
 b.m128i_i32[0] << "\t" << b.m128i_i32[1] << "\t" <<
 b.m128i_i32[2] << "\t" << b.m128i_i32[3] << std::endl;
 std::wcout << "Result res: " <<
 res.m128i_i32[0] << "\t" << res.m128i_i32[1] << "\t" <<
 res.m128i_i32[2] << "\t" << res.m128i_i32[3] <<std::endl;

 return 0;
}
```

Первый элемент результата содержит $a0 + a1$, второй – $a2 + a3$, третий – $b0 + b1$, четвертый – $b2 + b3$. Если программу удастся перепроектировать, так чтобы операции сложения выполнялись парами и сгруппировать пары по четыре, то эта встроенная функция даст

возможность распараллелить код. Существуют встроенные функции и для других операций (сложения, вычитания, вычисления абсолютного и противоположного значения, даже для вычисления скалярного произведения с использованием матрицы 8-разрядных целых размером 16×16) над числами разной «ширины», а также ряда других вычислений.

Один из недостатков векторизации с помощью встроенных функций состоит в том, что код становится гораздо труднее читать и сопровождать. Обычно сначала пишется «естественный» код, проверяется его правильность, а затем, если профилирование выявит узкие места и они поддаются векторизации, то производится преобразование к менее читаемому виду.

Помимо этого, в Visual Studio 2012 реализованы автовекторизация и автораспараллеливание кода. Компилятор автоматически векторизует циклы там, где это возможно. В ходе этой процедуры цикл (например, суммирования) преобразуется, так чтобы ЦП мог одновременно выполнить несколько итераций. За счет этого иногда удается добиться восьмикратного ускорения работы цикла на процессорах, поддерживающих SIMD-команды. Например, большинство современных процессоров поддерживают набор команд SSE2, который позволяет компилятору сгенерировать код выполнения арифметических операций сразу над четырьмя числами. Ускорение при этом достигается даже на одноядерных машинах, причем в программу не нужно вносить никаких изменений.

В процессе автораспараллеливания цикл преобразуется так, что его можно было одновременно выполнять в нескольких потоках, задействуя тем самым возможность многоядерных и многопроцесорных машин распределять части работы всем имеющимся процессорам. В отличие от автовекторизации, для автораспараллеливания вы должны сами сказать компилятору, какие циклы распараллеливать, воспользовавшись директивой *#pragma parallelize*. Оба механизма могут работать одновременно, так что векторизованный цикл затем распараллеливается на несколько процессоров.

OpenMP

OpenMP (MP – сокращение «multiprocessing») – это кросс-языковой, кросс-платформенный интерфейс прикладного программирования (API) для организации параллелизма на ЦП. Он существует с 1997 года, поддерживает языки Fortran, C и C++ и реализован в Windows и на ряде других платформ. Visual C++ поддерживает OpenMP

с помощью набора директив компилятора. OpenMP определяет, сколько имеется ядер, создает потоки и распределяет между ними работу. Ниже приведен пример:

```
// size – константа времени компиляции
double* x = new double[size];
double* y = new double[size + 1];

// поместить значения в x
#pragma omp parallel for
for (int i = 1; i < size; ++i)
{
  x[i] = (y[i - 1] + y[i + 1]) / 2;
}
```

Здесь мы обходим все элементы вектора *y* и строим из них вектор *x*. Добавление директивы *#pragma* и перекомпиляция программы с флагом /openmp – вот и всё, что нужно для распределения работы между несколькими потоками – по одному для каждого ядра. Например, если имеется четыре ядра и вектор *x* состоит из 10 000 элементов, то первому потоку может быть поручена обработка значений *i* от 1 до 2500, второму – от 2501 до 5000 и т. д. По завершении цикла вектор *x* будет корректно заполнен. Разумеется, программист должен позаботиться о том, чтобы цикл допускал распараллеливание, и в этом и состоит трудная часть работы. Например, следующий цикл не может быть распараллелен:

```
for (int i = 1; i <= n; ++i)
  a[i] = a[i - 1] + b[i];
```

В этом коде каждая итерация зависит от исхода предыдущей. Так, чтобы вычислить *a*[2502], поток должен иметь доступ к значению *a*[2501], а значит, второй поток не может начаться, пока не завершится первый. Если включить в этот код прагму, то компилятор не предупредит о наличии проблемы, но результат окажется неверен.

Одно из основных ограничений OpenMP является прямым следствием его простоты. Цикл от 1 до *size*, где *size* известно в начале цикла, легко распределить между потоками. Но OpenMP умеет обрабатывать только циклы *for*, в которых во всех трех частях используется одна и та же переменная (в данном примере *i*), и только в случае, когда в проверке и приращении фигурируют значения, известные на момент начала цикла.

Цикл

```
for (int i = 1; (i * i) <= n; ++i)
```

невозможно распараллелить с помощью директивы *#pragma omp parallel for*, потому что проверяется квадрат *i*, а не просто *i*.

Цикл

```
for (int i = 1; i <= n; i += Foo(abc))
```

также не распараллеливается, поскольку величина приращения *i* заранее неизвестна.

По той же причине невозможно таким способом распараллелить цикл, который «читает все строки файла» или обходит коллекцию с помощью итератора. В таком случае имеет смысл сначала последовательно прочитать все строки в какую-то структуру данных, а потом обработать ее в цикле, совместимом с OpenMP.

Система Concurrency Runtime (ConcRT) и библиотека Parallel Patterns Library

Microsoft Concurrency Runtime – это система, которая расположена между приложениями и операционной системой. Она состоит из четырех частей.

- **Библиотека PPL (Parallel Patterns Library).** Включает обобщенные типобезопасные контейнеры и алгоритмы.
- **Библиотека асинхронных агентов (Asynchronous Agents Library).** Предоставляет основанную на акторах модель программирования и механизм внутрипроцессной передачи сообщений; в совокупности они позволяют реализовать выполнение нескольких асинхронно взаимодействующих операций без блокировок.
- **Планировщик задач.** Координирует совместное выполнение задач с заниманием работы.
- **Диспетчер ресурсов.** Используется планировщиком задач для динамического выделения таких ресурсов, как процессорное ядро или память.

Библиотека PPL похожа на стандартную библиотеку Standard Library в том смысле, что для упрощения таких конструкций, как параллельные циклы, применяются шаблоны. Ее использование существенно облегчается за счет лямбда-выражений, добавленных в стандарт C++11 (хотя в компиляторе Microsoft Visual C++ они присутствуют с версии, вошедшей в Visual Studio 2010).

Например, последовательный цикл:

```
for (int i = 1; i < size; ++i)
```