ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛЕНИЯ НА МНОГОЯДЕРНЫХ ПРОЦЕССОРАХ В РЕАЛЬНОМ ВРЕМЕНИ

АЛЁША ВРАНЧИЧ (ALJOŠA VRANČIČ), ДЖЕФ МЕЙЗЕЛ (JEFF MEISEL), National Instruments

В статье рассматриваются методы оптимизации многоядерных систем для выполнения высокопроизводительных вычислений в реальном времени на примере процессора Intel. Данные методы применяются во многих проектах, включая мониторинги здоровья, системы контроля зданий или объектов, «помощники водителя» и т.д. Статья представляет собой сокращенный перевод [1].

Высокопроизводительные вычисления (HPC — high-performance computing) ввиду их громоздкости нельзя провести на персональном компьютере, не говоря уже о встраиваемых системах. Однако с появлением многоядерных процессоров современные устройства могут решать все больше задач. Подобного уровня развития HPC-систем пять лет назад невозможно было представить.

Основной объем вычислений зачастую проделывается автономно, т.е. процессор получает исходные данные, затем обрабатывает их и выдает конечный результат. Такие системы нельзя назвать системами реального времени, как бы быстро они ни работали. Вычисления в режиме реального времени предполагают ограничения по времени, за которое должен быть выдан результат (детерминизм). Подход, применяющийся в обычных системах, когда задачи распределяются между узлами системы посредством обмена сообщениями по Ethernet (стандарты MPI, MPICH), для систем реального времени неприемлем. Во-первых, пересылка сообщений происходит с задержками, а во-вторых, такой подход не позволяет синхронизовать узлы с большой точностью (не больше 1 мс). Строго говоря, его можно адаптировать для работы в реальном времени, но он изначально был разработан для других целей.

Многоядерные системы оказываются более приспособленными к работе в режиме реального времени, поскольку связь между ядрами осуществляется не по сети, а по системной шине (см. рис. 1). Кроме того, многоядерные системы, как правило, поддерживают работу с симметричными ОС, которые автоматически распределяют задачи по доступным ресурсам. Это значительно облегчает проектирование. Разработчик лишь указывает временные ограничения и приоритетность выполнения задач, не думая о взаимодействиях между ядрами.

ВЫЧИСЛЕНИЯ В РЕАЛЬНОМ ВРЕМЕНИ

Для полноценного использования ресурсов системы необходимо предусмотреть параллельные алгоритмы вычисления. При выборе алгоритма следует учитывать архитектурные особенности системы и требования к приложению. Одним из наиболее удобных языков программирования является LabVIEW. Он поддерживает работу с ОСРВ, параллельные алгоритмы, а также содержит большой набор инструментов, в т.ч. для оптимизации и синхронизации работы многоядерных систем. Рассмотрим кратко три подхода, которые можно применить в высокопроизводительных системах реального времени: конвейерный алгоритм, разделение данных и N-мерная сетка.

Конвейер (Pipelining)

Этот подход обычно используется в потоковых приложениях, а также в тех случаях, когда алгоритм состоит из нескольких шагов, выполнение каждого из которых занимает много времени и ресурсов ЦП. Например, в блоках быстрого преобразования Фурье. Как следует из названия, принцип здесь такой же, что и у конвейера: каждая стадия выполняется на отдельном ядре, а результат передается следующему и так до конца (см. рис. 2). Шаги, или стадии, должны быть подобраны так, чтобы их выполнение занимало примерно одно и то же время. Алгоритм можно прокручивать много раз с разными наборами данных, как показано на рисунке 2. Главное — разбить алгоритм на равные по времени выполнения блоки.

Недостатками данного подхода являются, во-первых, незащищенность от потери данных из кэш-памяти, а во-вторых, снижение быстродействия при большом количестве пересылок между ядрами.

Разделение данных

Если требуется обработать большой объем данных, то применяется другой подход, когда массив данных разбивается на блоки, каждый из которых обрабатывается отдельно, а затем они снова соединяются, формируя конечный результат. Скорость обработки при этом увеличивается пропорционально количеству потоков.

Разделение потока данных применяется, например, в системах контроля для одновременного умножения больших матриц на векторы. Размер матрицы обычно жестко задан, поэтому ее можно разделить на части. Вектор описывает показания датчиков и принимает новые значения при каждом прогоне. По результату перемножения можно судить об изменениях в системе или контролировать какую-либо величину.

Программно этот подход реализуется следующим образом. Матрица разбивается на блоки, причем их количество равно количеству ядер, участвующих в вычислении. Затем каждый блок умножается на вектор, а результаты суммируются в один общий вектор.

Рис. 1. **Связь между ядрами в процессоре**

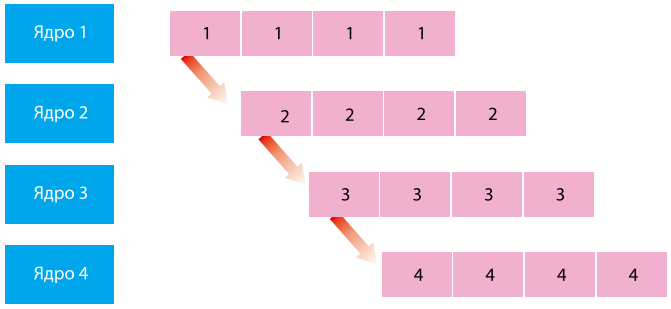

Рис. 2. **Конвейерный принцип реализации параллельных вычислений**

Структурная сетка

Этот алгоритм применяется обычно для проведения вычислений, связанных с физическими моделями. Примером может служить решение уравнения теплового баланса системы, в котором граничные условия постоянно меняются. Двумерная (в общем виде N-мерная) таблица вычисляется на каждой итерации, причем значение каждой ячейки зависит от соседних. Для многоядерных систем алгоритм изменяется следующим образом: сетка сначала разделяется на несколько блоков, каждый из которых вычисляется отдельно. Эффективность параллельных вычислений определяется отношением площади сетки к периметру.

Важно подчеркнуть, что какой бы ни был выбран подход, необходимо правильно организовать блок памяти.

КЭШ-ПАМЯТЬ

В традиционных встраиваемых системах обращение к кэш-памяти ЦП рассматривается как вынужденная мера. Дело в том, что время выполнения задачи зависит от объема занесенного в кэш кода (или данных), к которому будет происходить обращение во время выполнения задачи. Зависимость эта обратно пропорциональная — чем больше данных будет занесено в кэш, тем быстрее выполнится задача. Время выполнения недетерминировано.

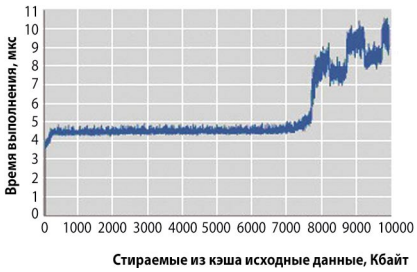

Рис. 3. **Время выполнения цикла заданной длительности в зависимости от объема кэшированного кода**

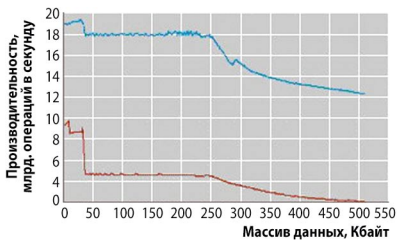

Рис. 4. **Производительность процессора в зависимости от объема кэшированных данных**

Для иллюстрации воспользуемся структурой, входящей в LabVIEW, которая называется «цикл заданной длительности» (timed loop). Она работает так же, как и цикл while, но учитывая аппаратные требования. К примеру, ее можно сконфигурировать так, чтобы она выполнялась на одном потоке. Длительность цикла задается с точностью до мкс. Циклу можно присвоить соответствующий приоритет для выполнения в ОСРВ. Кроме того, в цикле предусмотрена установка родственности процессоров, а также отклик на прерывания.

Хотя в рассмотренных алгоритмах не используется цикл заданной длительности, эта структура полезна при разработке HPC-приложений реального времени. Параллельные вычисления проводятся с помощью нескольких циклов и служебных структур очередности (queue structure), которые осуществляют пересылку данных между циклами.

Время выполнения цикла заданной длительности как функции от количества кэшированного кода или данных показано на рисунке 3. Для получения этой зависимости после каждого прогона цикла уменьшали массив данных, необходимых для выполнения критичной ко времени задачи. Это происходило за счет записи в кэш нового массива, к которому после каждой итерации прибавлялась единица. Из рисунка 3 видно, что чем меньше исходных данных записано в кэш, тем дольше выполняется код.

В высокопроизводительных системах реального времени кэш-память очень важна, поскольку она позволяет использовать вычислительные ресурсы ЦП. Простой пример. Допустим, мы хотим добавить 1 ко всем элементам массива с обычной точностью, используя 3-ГГц процессор, который может выполнять одну операцию с плавающей точкой за один цикл такта, т.е. требуется выполнить 3 млрд операций в секунду (3 GFLOP). Тогда пропускная способность блока памяти должна составлять по меньшей мере 24 Гбайт/с (по 12 Гбайт/с в каждом направлении). В современных системах эта планка пока не достигнута. Самая большая полоса у трехканального процессорного модуля i7, она составляет 18 Гбайт/с. Однако в нем содержится несколько блоков FPU (Floating Point Unit — блок арифметики с плавающей запятой), поэтому он может производить более одной операции за такт. Используя дополнительные инструкции SSE, в каждом такте можно прибавлять не одно, а четыре числа типа float с обычной точностью. Таким образом, для обработки массива потребовалась бы полоса 96 Гбайт/с, чтобы не было остановов.

На рисунке 4 показана зависимость количества операций в секунду от размера массива для процессора i7 с частотой 3,2 ГГц при выполнении двух функций с использованием команд SSE: x[i] = x[i] + 1 (красная кривая) и x[i] = A·x[i]2 + B·x[i] + c (синяя кривая). Аргументом для обеих функций служат элементы массива чисел типа float с обыкновенной точностью. На графиках четко видны три ступени кэша: L1 (32 Кбайт), L2 (256 Кбайт) и L3 (8 Мбайт). Когда все данные помещаются на уровне L1, производительность одного процессора достигает 8,5 GFLOP при пропускной способности памяти 72 Гбайт/с. Когда данные переполняют L2, процессор выполняет до 4,75 млрд операций при пропускной способности памяти 38 Гбайт/с. Как только данные начинают выходить за рамки кэш, быстродействие падает до 0,6 млрд операций в секунду, удерживаясь на уровне 4,8 Гбайт/с.

На рисунке 4 видна дополнительная ступень в начале красной кривой. Она может указывать на еще один уровень кэш — 8 Кбайт. Максимальное и минимальное быстродействие отличаются в 14 раз. Для процессора с четырьмя ядрами разница еще больше за счет параллельных вычислений. В лучшем случае четырехъядерный процессор может выполнить 36 млрд операций, поскольку блоки кэш-памяти независимы. В худшем — те же 0,6 млрд, поскольку память ОЗУ является общей для всех процессоров. Итак, разница в 56 раз.

Для верности запустим другую задачу, выполнение которой требует больше операций в секунду. Вместо прибавления единицы к элементу вычислим полином второго порядка. Это потребует примерно в 4 раза больше операций. Полученная зависимость изображена синим графиком на рисунке 4. Как и ожидалось, максимальное быстродействие увеличивается до 15 млрд операций в секунду, поскольку процессор обращается к памяти реже. По этой же причине уменьшается разница в быстродействии системы между слу-

чаями, когда данные полностью расположены в кэше уровня L1 или L2. Поскольку задержка и пропускная способность памяти являются ограничивающими факторами, то мы опять видим большую разницу в производительности, хотя и не такую ощутимую, как в первом случае, всего в 8 раз.

Рассмотренный пример показывает, что процессоры с несколькими ядрами могут работать быстрее, чем одноядерные, только в том случае, если хватает размера их кэш-памяти. Тем не менее из рисунка 4 следует, что лишняя пересылка данных между ядрами может существенно снизить быстродействие, особенно если данные перемещаются из основной памяти. Это необходимо учитывать при организации параллельных вычислений, поскольку при распараллеливании задачи часть данных из кэша стирается. В итоге можно не только потерять весь выигрыш в быстродействии, который был получен за счет использования нескольких ядер, но и получить систему с меньшей производительностью, чем одноядерная.

А что делать, если данные не помещаются в кэш? В этом случае необходимо проанализировать поток данных, использующийся критичной ко времени задачей, и выделить среди них наиболее важные. Например, данные, использующиеся в алгоритмах контроля, должны быть доступны в любое время, поэтому они хранятся в кэше. Данные, которые используются для интерфейса пользователя, не являются критичными ко времени и их можно поместить в ОЗУ.

Другой способ хранить данные в кэше — не позволять другим потокам выполняться на данном ядре. В этом случае очень полезна возможность ОСРВ резервировать определенный процессор для выполнения выбранной задачи. Следует иметь в виду, что в некоторых процессорах кэш-память расположенных на одном физическом кристалле ядер объединена (например, в i7 кэш L3 используется 8 ядрами). В этом случае резервирование становится крайне неэффективным методом.

Если построить алгоритм так, чтобы обращение к данным поочередно производилось в двух направлениях (от первого к последнему и от последнего к первому), то в кэш можно поместить больше данных. Дело в том, что если обращение к данным производится в одном направлении, то те данные, которые уже не помещаются в кэш, непременно теряются. Если же обращение происходит в двух направлениях, то те данные, к которым происходило последнее обращение в предыдущем цикле, могут оказаться первыми, к которым обратится процессор в последующем цикле. И они уже находятся в кэше. Если объем данных не намного превышает емкость кэш-памяти, то этот прием поможет ускорить выполнение задачи. Однако выигрыш будет тем меньше, чем больше данных не помещается в кэш.

На рисунке 5 показаны результаты, иллюстрирующие применение указанных методик управления кэшем. Задача заключалась в умножении матрицы размером 3000×6000 на вектор из 6000 элементов и решалась для системы контроля самой большой в мире Европейской южной обсерватории. Система раз в миллисекунду считывает показания датчиков, чтобы определить расположение элементов зеркала. Если обнаруживаются отклонения, зеркала подстраиваются. Всего необходимо проверить 3000 параметров, которые определяются показаниями 6000 датчиков.

Изначально были использованы стандартные библиотеки для умножения (синяя кривая на графике), однако желаемого быстродействия достичь не удалось. Алгоритмы были самыми эффективными, однако они не были оптимизированы под HPC-системы реального времени. Поэтому был разработан новый алгоритм умножения матрицы на вектор, при котором использовались следующие приемы.

1. Матрица итераций, обратная к которой используется для вычислений параметров механизма, меняется не часто. Следовательно, ее можно предварительно преобразовать так, чтобы она помещалась в кэш L1 и L2 и подходила для инструкций SSE.

2. Если назначить процессору выполнение только одной задачи, то контрольная матрица всегда будет находиться в кэше, что повысит производительность системы.

3. Разделение процедуры умножения на несколько параллельных задач увеличивает объем кэш-памяти, доступной для работы.

4. Новый алгоритм предусматривает обращение к матрице в обоих направлениях.

На рисунке 5 показана зависимость времени умножения матрицы на вектор от величины матрицы. Данные были получены на двух четырехъядерных процессорах Intel Xeon с тактовой частотой 2,6 ГГц и 12 Мбайт кэш-памяти в каждом. Задействовано было всего четыре ядра, по два на каждом процессоре.

Вывод: рассмотренные приемы управления кэш-памятью помогают значительно ускорить вычисления.

ЛИТЕРАТУРА

1. A. Vrančič, J. Meisel. A real-time HPC-approach for optimizing Intel multi-core architectures//www.industrialcontroldesignline.com/howto/218100688

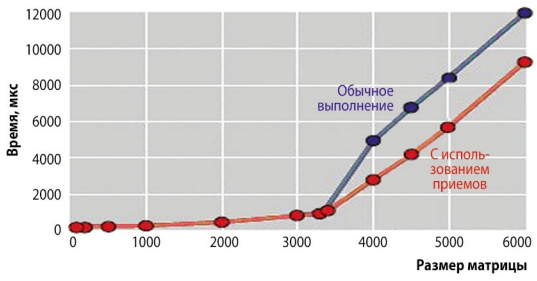

Рис. 5. **Время, затрачиваемое на умножение матрицы на вектор, как функция размера матрицы**