А.В. Чистяков, И.С.Ислямова

МЕТОД И ТЕХНОЛОГИИ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ ПРИ РЕШЕНИИ ПРИКЛАДНЫХ ЗАДАЧ

Национальный авиационный университет

кафедра инженерии программного обеспечения

Научный руководитель Иванова Л.Н. (к.т.н., доцент)

Постановка задачи

На сегодняшний день ни одна отрасль народного хозяйства, а также любого производства не может эффективно работать без использования компьютерной техники, а тем более решать конструкторские задачи, задачи моделирования и создания новых образцов. Даже, на первый взгляд, несложные задачи, которые встречаются в повседневной жизни, не всегда могут быть решены эффективно классическими методами и средствами линейного или структурного программирования. Также и наличие мощной вычислительной техники не гарантирует быстрого выполнения поставленной задачи средствами среды разработки программного обеспечения (ПО). В качестве примера можно привести приложение, написанное на языке программирования C++ и скомпилированное в режиме многопоточного выполнения, однако общее время исполнения сократится незначительно. Поэтому возникает задача максимально эффективного использования вычислительных ресурсов ЭВМ как при разработке нового ПО, так и при модернизации существующего. Одним из решений этой задачи может быть использование технологий и методов параллельного программирования.

Анализ существующих топологий многоядерных компьютеров

На сегодняшний день в мире существует множество классов и типов компьютеров, их все можно классифицировать по количеству потоков команд и по количеству потоков данных, которые обрабатывает одновременно система (классификация Флинна):

- ОКОД — Вычислительная система с **о**диночным потоком **к**оманд и **о**диночным потоком **д**анных (SISD, **S**ingle **I**nstruction stream over a **S**ingle **D**ata stream).
- ОКМД — Вычислительная система с **о**диночным потоком **к**оманд и **м**ножественным потоком **д**анных (SIMD, **S**ingle **I**nstruction, **M**ultiple **D**ata).
- МКОД — Вычислительная система со **м**ножественным потоком **к**оманд и **о**диночным потоком **д**анных (MISD, **M**ultiple **I**nstruction **S**ingle **D**ata).
- МКМД — Вычислительная система со **м**ножественным потоком **к**оманд и **м**ножественным потоком **д**анных (MIMD, **M**ultiple **I**nstruction **M**ultiple **D**ata).

Самыми распространенными и эффективными промышленными системами являются MIMD-компьютеры. В состав такого компьютера входит несколько процессоров, которые функционируют асинхронно и независимо друг от друга. В любой момент времени различные процессоры могут выполнять различные команды над разными частями одних и тех же данных. MIMD-компьютеры могут быть как однородные, состоящие из одинаковых узлов, так и разнородные (полисистемы), состоящие из различных по аппаратной составляющей узлов. Разнородные системы часто называют Beowulf-системами или Beowulf-кластерами. Beowulf-кластер состоит из широко распространённого аппаратного обеспечения, работающие под управлением операционной системы с открытым исходным кодом (например, GNU/Linux или FreeBSD). Основным до-

стоинством таких систем является их дешевизна, высокая производительность и возможность комбинировать любые модели компьютеров в один кластер.

MIMD-компьютеры относятся к компьютерам с разделенной памятью. А персональные компьютеры и малые сервера, привычные обычному пользователю, относятся к SIMD-системам с общей памятью.

В последние годы очень активно развиваются гибридные системы и вычисления на графических ускорителях (видеокартах). Гибридная система представляет собой персональный компьютер (ПК), сервер или кластер, на котором установлены специализированные программно-аппаратные комплексы, позволяющие выполнять задачи не только на центральном процессоре (CPU), но и на процессоре видеокарты (graphics processing unit, GPU). В этом случае работу вычислительной системы можно организовать таким образом, что бы поставленная задача могла выполнятся параллельно как на отдельно взятом GPU, так и совместно с CPU. Совместное использование CPU с GPU позволяет повысить эффективность работы ПО в десятки раз. Это связано с тем, что процессор видеокарты имеет в своем составе большое количество арифметико-логических устройств, которые специализированы под обработку больших массивов данных. Но при решении прикладных задач (моделирования физики процессов, генома, фармацевтики, погоды, и т.д.) не всегда достаточно даже суммарной производительности CPU и GPU. Если установить в узлы кластера видеокарты и использовать несколько GPU параллельно, то можно повысить эффективность решения поставленной задачи на несколько порядков.

Графические процессоры можно рассматривать как мощные параллельные SIMD-процессоры (Single Instruction Multiple Data), способные выполнять одну и ту же операцию одновременно над несколькими значениями однородных данных. То есть SIMD-процессор получает на вход поток однородных данных и параллельно обрабатывает их, порождая тем самым выходной поток.

Анализ существующих технологий и средств параллельного программирования

На сегодняшний день можно выделить несколько основных технологий параллельного программирования для систем с разделенной памятью (Message Passing Interface, Parallel Virtual Machine), для систем с общей памятью (Open Multi-Processing), а так же для программирования видеокарт и GPU (CUDA, ATI Stream Technology).

Message Passing Interface (MPI, интерфейс передачи сообщений) [1] — программный интерфейс для передачи информации, который позволяет обмениваться сообщениями между процессорами, выполняющими одну задачу. Разработан Уильямом Гроуппом, Эвином Ласком и другими.

MPI является наиболее распространённым стандартом интерфейса обмена данными в параллельном программировании. Существуют его реализации для большого числа компьютерных платформ и для языков программирования Фортран 77/90, Си и Си++. Основным средством коммуникации между процессами в MPI является передача сообщений друг другу. Стандартизацией MPI занимается MPI Forum. В стандарте MPI описан интерфейс передачи сообщений, который должен поддерживаться как системой, на которой выполняется ПО, так и самим ПО пользователя. В настоящее время существует большое количество бесплатных и коммерческих дистрибутивов MPI: MPICH, LAM, HPVM, OpenMPI, WMPI и другие.

Одной из основных особенностей MPI является то, что эта система лучше всего работает в однородных системах. Это связано с тем, что используя базовый пакет MPI, система будет работать до тех пор, пока задача не будет решена на самом маломощном узле разнородного кластера. Вследствие чего большая часть вычислительного ресурса будет простаивать, что не целесообразно. Средствами MPI решить задачу балансировки вычислительной нагрузки сложно, для этого требуется писать специальные алгоритмы решения этой задачи.

Parallel Virtual Machine [2] (PVM, параллельная виртуальная машина) – является основой вычислительной среды Beowulf-кластера, который представляет собой пакет программ и позволяет использовать связанный в локальную сеть набор разнородных компьютеров, работающих под операционной системой Unix, как один большой параллельный компьютер. Таким образом, проблема больших вычислений может быть весьма эффективно решена за счет использования совокупной мощности и памяти большого числа компьютеров. Пакет программ PVM легко переносится на любую платформу, начиная от laptop и до CRAY.

PVM можно определить как часть средств реального вычислительного комплекса (процессоры, память, периферийные устройства и т.д.), предназначенную для выполнения множества задач, участвующих в получении общего результата вычислений. В общем случае число задач может превосходить число процессоров, включенных в PVM. Кроме того, в состав PVM можно включать довольно разнородные вычислительные машины, несовместимые по архитектуре данных. Иначе говоря, параллельной виртуальной машиной может стать как отдельно взятый персональный компьютер (ПК), так и локальная сеть, включающая в себя суперкомпьютеры с параллельной архитектурой, универсальные ЭВМ, графические рабочие станции и все те же маломощные ПК. Важно лишь, чтобы о включаемых в PVM вычислительных средствах имелась информация в используемом программном обеспечении PVM. Благодаря этому программному обеспечению пользователь может считать, что он общается с одной вычислительной машиной, в которой возможно параллельное выполнение множества задач.

Использование PVM более желательно в кластерах типа Beowulf. Это связано с тем, что при разработке приложения программист должен учитывать аппаратную особенность каждого узла кластера, а это не всегда удобно. Но тем самым решается задача с простоем вычислительных мощностей.

Для программирования в системах с общей памятью используется технология OpenMP.

OpenMP (Open Multi-Processing) [3,4] - открытый стандарт для распараллеливания программ, написанных на языках программирования Си, Си++ и Фортран. Описывает набор директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных прило-жений на многопроцессорных системах с общей памятью.Разработку спецификации OpenMP ведут несколько крупных производителей вычислительной техники и ПО, чья работа регулируется некоммерческой организацией, называемой OpenMP Architecture Review Board (ARB).

OpenMP реализует параллельные вычисления с помощью многопоточности, в которой «главный» (master) поток создает набор подчиненных (slave) потоков и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами (количество процессоров не обязательно должно быть больше или равно количеству потоков).

MPI и OpenMP технологии могут быть использованы совместно для повышения эффективности функционирования многоядерных узлов в кластерах, так как MPI ориентирована на системы с разделенной памятью, то есть когда затраты на передачу данных велики, а OpenMP ориентирована на системы с общей памятью (многоядерные ПК).

Задачи, выполняемые потоками параллельно, также как и данные, требуемые для выполнения этих задач, описываются с помощью специальных директив препроцессора соответствующего языка — прагм.

Ключевыми элементами OpenMP являются:

- конструкции для создания потоков (директива parallel);
- конструкции распределения работы между потоками (директивы DO/for и section);
- конструкции для управления работой с данными (выражения shared и private);
- конструкции для синхронизации потоков (директивы critical, atomic и barrier);
- процедуры библиотеки поддержки времени выполнения (например, omp_get_thread_num);
- переменные окружения (например, OMP_NUM_THREADS).

OpenMP - это идеальное средство для модернизации существующего ПО функционирующего в однопроцессорных системах. Так как данная технология использует прагмы для управления потоками и поэтому нет необходимости масштабно редактировать исходный код.

CUDA (Compute Unified Device Architecture) — это технология компании NVIDIA, основанная на расширении языка Си и которая даёт возможность управления набором инструкций и памятью графического ускорителя для организации параллельных вычислений. CUDA может быть применена на графических процессорах видеокарт (видеоускорителей) GeForce восьмого поколения и старше (серии GeForce 8, GeForce 9, GeForce 200), а также Quadro и Tesla.

Трудоемкость программирования GPU при помощи CUDA довольно велика, однако она ниже, чем с ранними GPGPU (General-purpose graphics processing units — «GPU общего назначения») решениями. При создании ПО с использованием CUDA требуется учитывать его выполнение на нескольких мультипроцессорах, так же как и при MPI про-

граммировании, но без разделения данных, которые хранятся в общей видеопамяти. И так как CUDA программирование для каждого мультипроцессора подобно OpenMP программированию, оно требует хорошего понимания организации памяти. Но, конечно же, сложность разработки нового и переноса существующего ПО на CUDA в большой степени зависит от решаемой задачи.

Технология программирования CUDA используется для организации массивно-параллельных операций на GPU [5]. При этом последовательная часть программы выполняется на CPU, а массивно-параллельные вычисления организуются на GPU, как набор ниток, одновременно выполняющихся потоков (threads). При этом каждой нитке соответствует один элемент вычисляемых данных. Нити объединяются в сетки (grid) и блоки (block). Каждый блок может быть одномерным, двумерным или трехмерным. Таким образом, задача в CUDA разбивается на несколько отдельных подзадач. Каждой такой подзадаче соответствует свой блок нитей, и взаимодействие между нитями может происходить только в пределах одного блока на быстрой разделяемой памяти.

Технологию CUDA может использовать любой программист, знающий язык Си. Придётся только привыкнуть к иной парадигме программирования, присущей параллельным вычислениям. Но если алгоритм в принципе хорошо распараллеливается, то изучение и затраты времени на программирование на CUDA вернутся в многократном размере.

Технология CUDA включает в себя и позволяет следующее:

- унифицированный программно-аппаратный комплекс для параллельных вычислений на видеочипах NVIDIA;
- набор инструментов, поддерживающих работу с GPU различных типов устройств;
- поддерживает программирование на языке программирования Си;
- стандартные библиотеки численного анализа FFT (быстрое преобразование Фурье) и BLAS (линейная алгебра);
- оптимизированный обмен данными между CPU и GPU;
- взаимодействие с графическими API (application programming interface, Интерфейс прикладного программирования) OpenGL и DirectX;
- поддержку 32- и 64-битных операционных систем: Windows XP, Windows Vista, Linux и MacOS X;
- возможность разработки ПО на низком уровне.

В дополнение к свойству поддержки операционных систем необходимо уточнить, что официально поддерживаются все основные дистрибутивы Linux (Red Hat Enterprise Linux 3.x/4.x/5.x, SUSE Linux 10.x), но CUDA прекрасно работает и на других сборках операционных систем (ОС): Fedora Core, Ubuntu, Gentoo и др.

Среда разработки CUDA (CUDA Toolkit) включает:

- компилятор nvcc;
- библиотеки FFT и BLAS;
- профилировщик;
- отладчик gdb для GPU;
- драйвер CUDA runtime в комплекте стандартных драйверов NVIDIA;
- руководство по программированию;
- CUDA Software Development Kit (SDK, комплект средств разработки ПО) (исходный код, утилиты и документация).

В состав CUDA SDK входят примеры: параллельная битонная сортировка (bitonic sort), транспонирование матриц, параллельное префиксное суммирование больших массивов, свёртка изображений, дискретное вейвлет-преобразование, пример взаимодействия с OpenGL и Direct3D, использование библиотек CUBLAS и CUFFT, вычисление цены опциона (формула Блэка-Шоулза, биномиальная модель, метод Монте-Карло), параллельный генератор случайных чисел Mersenne Twister, вычисление гистограммы большого массива, шумоподавление, фильтр Собеля (нахождение границ).

Технология Firestream - это программно-аппаратная вычислительная архитектура компании AMD. AMD FireStream (ранее ATI FireStream и AMD Stream Processor) - является потоковым процессором, разработанным компанией ATI [6,7].

Областями применения FireStream являются задачи, требующие большого вычислительного ресурса, такие как финансовый анализ или обработка сейсмических данных. Использование потокового процессора позволило увеличить скорость некоторых финансовых расчетов в 55 раз по сравнению с решением той же задачи силами только центрального процессора.

К системе программирования ATI Stream входит набор приложений и процессоры ATI Stream. На рис. 1 представлено взаимодействие всех компонентов системы программирования ATI Stream. Система программирования ATI Stream обеспечивает конечных пользователей и разработчиков гибкими средствами для пользования графическими процессорами. ПО ATI поддерживает открытые системы и открытые стандарты. Таким образом, открытая стратегия ATI дает возможность разработчикам внедрять программное обеспечение, разработанное для ATI Stream по лицензии GPL.

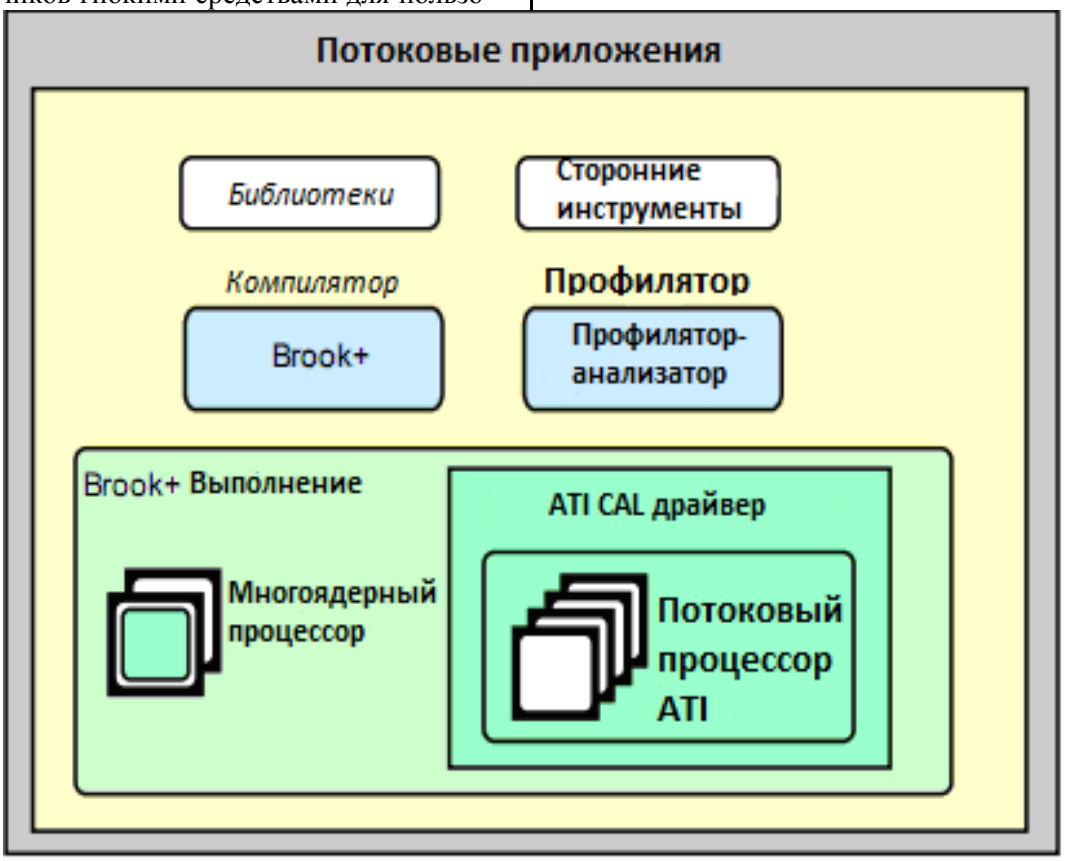

Рис. 1 Взаимодействие систем компонентов ATI Stream

В состав программного обеспечения входит:

• компилятор brook + с дополнениями для ATI устройств;

• драйвер устройства для ATI процессоров (ATI Compute Abstraction Layer – CAL);

• профилятор-анализатор быстродействия - Stream Kernel Analyzer;

• вычислительная библиотека - AMD Core Math Library (ACML);

Программирование графических процессоров ATI Stream проводится с помощью технологии программной модели шейдеров. Потоковые процессоры выполняют пользовательсие приложения, называемые потоковым ядром (stream kernel).Потоковые процессоры могут использоваться для неграфических вычислений, используя модель программирования SIMD. В этой модели программирования, которая называется потоковым программированием, массивы входных данных, хранящихся в общей памяти, разделяются между определенным количеством SIMD процессоров, которые в свою очередь выполняют поточные ядра и записывают результат в общую память.

Каждый экземпляр ядра в на текущем процессоре, называется нитью (thread). Существует определенный квадратный регион, в котором отражаются все нити, который в свою очередь называется домен исполнения (domain of execution).

Потоковый процессор формирует очередь нитей для определенной группы ниточных процессоров (thread proccessors) до того, пока применение не закончит работу. Упрощенную модель вычислений ATI Stream можно увидеть на рис. 2.

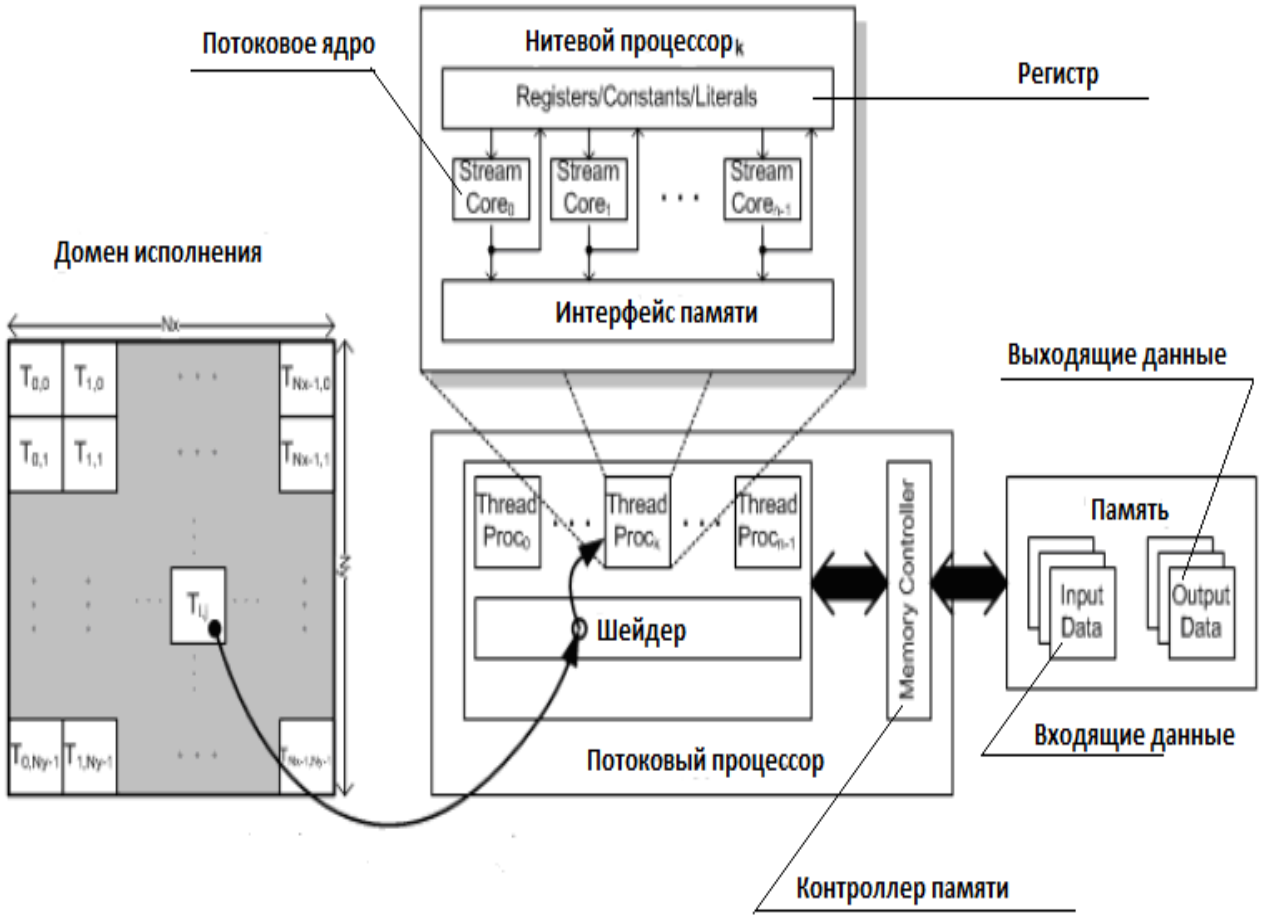

Рис. 2 Упрощенная модель вычислений ATI Stream

Открытый параллельный компилятор Brook +. Brook + дает возможность явного распараллеливания, используя расширения языка ANSI C. Модель вычислений Brook +, так называемая потоковая модель, разрабатывается параллельно с традиционными методами параллельного программирования и позволяет:

• организовать параллелизм по данных., таким образом позволяет использовать свойство SIMD-архитектуры.

• использование повышенной арифметической точности, таким образом позволяет разработку компьютерных алгоритмов с максимальной точностью результата и сокращением времени коммуникации между ядрами.

В языке программирования Brook + существует два ключевых элемента:

• Поток.

• Ядро.

Пример:

```
kernel void sum (float <> a, float <> b, float <> c)
{
c = a + b
}
```

Такой пример кода, написанный на языке Brook +, добавляет два потока и записывает их в выходной поток.

На рис. 3 изображены элементы языка программирования Brook +, в который входят [7]:

- brcc - компилятор программ Brook + с расширением. br, что превращает код Си в код, понятный графическим процессорам (у этого компилятора есть функция создания виртуальной среды, позволяющей отлаживать программы);

brt - библиотека, которая дает возможность пользоваться функциями для работы с графическими процессорами.

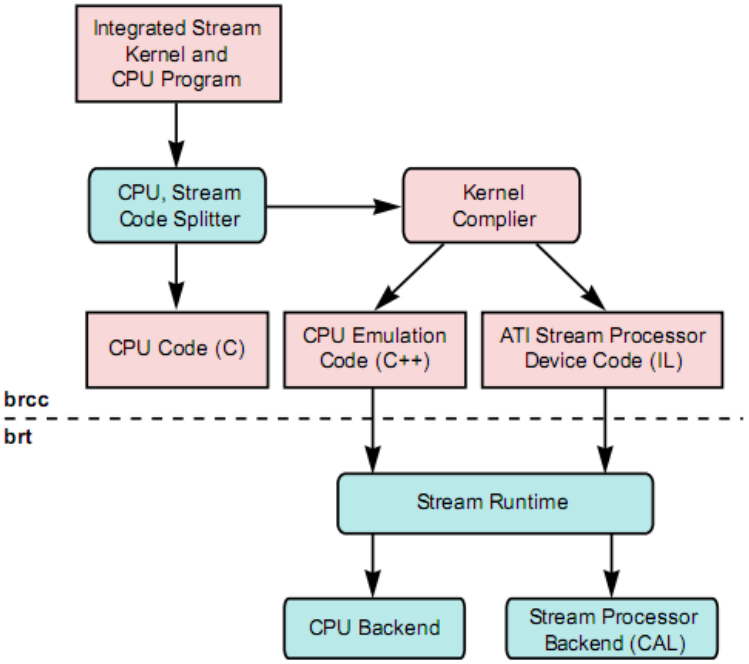

Рис. 3 Элементы яызка программирования Brook+

Абстрактный уровень вычислений (ATI Computation Abstraction Layer – CAL).

Как показано на рис. 4 существует абстрактный уровень вычислений, позволяющий пользоваться функциями для управления ядрами. Такой абстрактный уровень представлен драйверами и библиотекой. Таким образом, такая модель позволяет пользователю управлять процессом выполнения программы и тем самым повысить ее производительность.

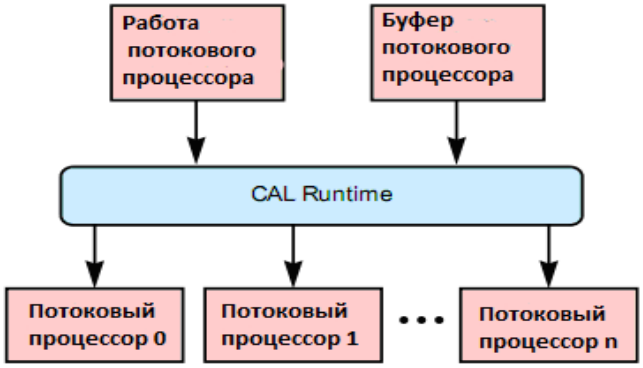

Рис. 4 Модель коммуникации среды и процессоров

CAL позволяет:

• генерировать инструкции для управления аппаратным обеспечением;
• управлять аппаратным обеспечением;
• управлять ресурсами;
• выполнять инструкции;
• поддерживать работу многих устройств;
• интероперабельность с разными 3D платформами.

К CAL входят функции языка Си и наборы типов данных, позволяющих высокоуровневым приложениям управлять процессами и количеством памяти на устройстве, где выполняется параллельное приложение.

AMD Core Math Library (ACML). В состав среды разработки входит математическая библиотека линейной алгебры ACML, которая имеет функции BLAS. Такая библиотека вклю-

чает основные математические функции. Она оптимизирована для платформы ATI.

В состав этой библиотеки входят следующие модули:

- весь набор функций BLAS;
- набор функций LAPACK;
- функции быстрого преобразования Фурье;
- функции генератора случайных чисел.

Потоковые процессоры. На рис. 5 показана упрощенная структура потокового процессора. Существует большое количество потоковых процессоров, но большинство из них имеют следующие модули:

- SIMD двигатель;
- нитевые процессоры;
- Потоковые ядра.

В состав потокового процессора входят группы SIMD двигателей, которые в свою очередь имеют определенное количество нитевых процессоров, отвечающих за выполнение ядер / модулей, каждое в отдельном потоке. Ниточный процессор включает большое количество потоковых ядер, отвечающих за расчеты разной точности. Каждый потоковый процессор выполняет одну инструкцию, но с разным набором данных, который показывает свойственную им SIMD-архитектуру.

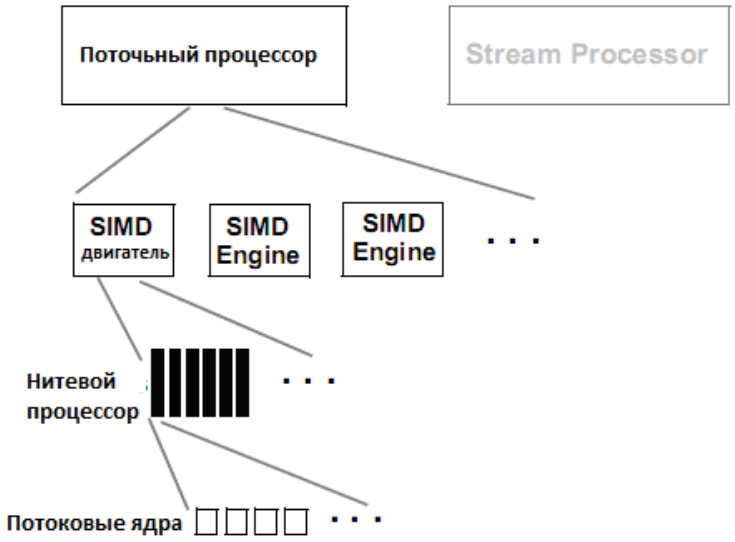

Рис. 5 Спрощена структура потокового процесора

Существуют некоторые рекомендации (правила) по переносу ПО на многопоточные технологии, например, планирование цикла и соответственно баланс нагрузки. Эти правила одинаково должны применяться для любой из выше приведенных технологий.

Метод инкрементного программирования и правила параллельного программирования

Создание эффективного алгоритма – самая сложная часть создания любого ПО, а особенно параллельного. Поэтому для упрощения написания и модернизации уже существующего ПО часто используют метод инкрементного программирования. Инкрементное программирование – это процесс подробного разбора алгоритма или ПО на составляющие части таким образом, чтобы получить независящие друг от друга блоки данных и\или команд. По результату анализа ПО программист дописывает требующиеся блоки кода, отвечающие за распараллеливание, руководствуясь правилами, приведенными ниже. Для простоты изложения, будем считать, что мы модернизируем некое ПО при помощи технологии OpenMP [8].

Первым правилом параллельного программирования является правило о балансе нагрузки на узлы системы. Баланс нагрузки (распределение рабочей нагрузки поровну между потоками), является одним из наиболее важных атрибутов параллельного выполнения приложения. Данное правило имеет большое значение, поскольку гарантирует работу всех процессоров большую часть времени. Без баланса нагрузки некоторые потоки могут завершить работу значительно раньше остальных, что приводит к простою вычислительных ресурсов и потере производительности.

В циклах отсутствие баланса нагрузки обычно является следствием различия времени вычисления в различных итерациях цикла. Разброс времен вычисления в итерациях цикла обычно легко определить, изучив исходный код. В большинстве случаев мы увидим, что

итерации цикла занимают одинаковое время. Если это не так, то можно найти наборы итераций, занимающие одинаковое время. Например, иногда набор всех четных итераций занимает примерно столько же времени, как и набор всех нечетных итераций. Аналогично, первая половина итераций цикла может занимать примерно столько же времени, как и вторая половина. С другой стороны, может оказаться невозможным найти наборы итераций, имеющие одинаковое время выполнения. Независимо от того, какой из этих случаев имеет место в конкретном приложении, необходимо предоставить OpenMP эту дополнительную информацию о планировании цикла, чтобы он мог правильно распределить итерации цикла между потоками (и, следовательно, между процессорами) для оптимизации распределения нагрузки.

По умолчанию, OpenMP предполагает, что все итерации цикла занимают одинаковое время. В результате OpenMP распределяет итерации цикла между потоками примерно поровну таким образом, чтобы минимизировать вероятность возникновения конфликтов памяти вследствие ее неправильного совместного использования. Это возможно, поскольку итерации цикла обычно обращаются к памяти последовательно. Поэтому при разделении цикла на две большие части (например, на первую и вторую половины) при использовании двух потоков вероятность наложения памяти оказывается наименьшей. Однако, хотя это и может быть наилучшим вариантом во избежание конфликтов памяти, с точки зрения баланса нагрузки это может быть плохим выбором. К сожалению, обратное тоже справедливо. То, что хорошо для баланса нагрузки, может быть плохо для работы с памятью. Поэтому инженерам по производительности необходимо найти баланс между оптимальным использованием памяти и оптимальным распределением нагрузки, измеряя производительность, чтобы определить, какие методы дают наилучшие результаты.

Как можно было понять из изложенного выше, вторым правилом параллельного программирования является то, что программу или функцию невозможно эффективно распараллелить если она обращается к общим ресурсам. Например, если нам надо чтобы несколько потоков в процессе решения одной задачи выводили на консоль, допустим, контрольные суммы в определенном порядке. Так как управление консолью будет передаваться каждому потоку в порядке очереди, которая формируется по мере готовности каждого результата контрольной суммы, в итоге мы получим последовательность чисел которая будет полностью лишена какого-либо смысла. Вместо результата первого потока мы сможем увидеть результат любого другого потока, только потому, что он вывел контрольную сумму на долю секунды раньше первого. А вторым примером общего доступа к ресурсам может стать индекс массива. Если потоки будут иметь общий доступ к индексам при умножении или любой другой операции над матрицами, мы получим неконтролируемый инкремент или дикремент, что приведет к неправильной работе алгоритма, и соответственно к неправильному результату.

Примеры применения технологии инкрементного программирования при создании параллельного ПО

Рассмотрим на примере умножения двух матриц $C=A\times B$ как можно применить в конкретной программе, написанной на языке программирования C++, средства OpenMP с целью распараллеливания выполнения отдельных её частей.

Как известно, умножение двух матриц осуществляется по формуле:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} .$$

Программа на языке программирования C++ будет иметь вид:

```
void m_mult_m (double *a, double *b, double *c, int n)
{
   int i, j, k;
    for(i = 0; i < n; i++)
        for(j = 0; j < n; j++)
            c[i][j]= 0;
     for (i = 0; i < n; i++){
       for (j = 0; j < n; j++){
        double s =0;
        for (k = 0; k < n; k++)
        s += a[i][k] * b[k][j];
```

```
            c[i][j]=s;
           }
        }
    }
```

Из формулы умножения матриц мы видим, что вычисления промежуточных произведений являются независимыми, поскольку каждое из них записывает (и читает) свой элемент c_{ij}. Таким образом, их можно выполнять параллельно. Для этого в исходный код программы вставляем соответствующие директивы OpenMP для параллельной реализации промежуточных произведений. Ниже приводится фрагмент текста программы на C++ с некоторыми вставками из OpenMP.

```
#include <omp.h> …
…
void m_mult_m_OpenMP (double *a, double *b, double *c, int n)
      { int i, j, k;
// запретить изменять количество потоков во время исполнения
//программы
#pragma omp_set_dynamic(0);
// установить количество потоков равным 2
#pragma omp_set_num_threads(2);
    for(i = 0; i < n; i++)
        for(j = 0; j < n; j++)
            c[i][j]= 0;
// описываем массивы, которые содержат  элементы матриц, //глобально
видимыми, а индексы – локально видимыми
#pragma omp parallel shared(a, b, c) private(i,j,k)
 #pragma omp for
 for (i = 0; i < n; i++){
   for (j = 0; j < n; j++){
    double s =0;
        for (k = 0; k < n; k++)
        s += a[i][k] * b[k][j];
      c[i][j]=s;
   }
 }
}
```

Результаты работы программы без распараллеливания и с распараллеливанием вычислений, т.е. с применением OpenMP, приведено в табл. 1. Программа выполнялась на 2-х ядерном процессоре. Конфигурация системы: DualCoe E6300 с 3.49ГБ ОЗУ, Windows XP.

Таблица 1. Время выполнения программы на 2-х ядерном процессоре

Размер матриц	Программа с OpenMP, мс	Программа без OpenMP, мс
200x200	32	63
500x500	890	1764
1000x1000	8766	15703
2000x2000	71469	124469
3000x3000	264328	425687

Из табл. 1 видно, что время решения задачи, использующей параллельные вычисления, в среднем на 50% меньше чем без.

В табл. 2 приведено время выполнения программы на одно и двух ядерном процессоре для матриц размером 3000x3000.

Таблица 2.Сравнение времени выполнения программы на одно и двух ядерных процессорах

Компьютер	Программа с OpenMP, мс	Программа без OpenMP, мс
Pentium4 3,0GHz +HT 2 ГБ ОЗУ	748156	730857
E6300 2x2,8GHz 3,49ГБ ОЗУ	320428	463157

Примечание: Pentium4- одноядерный процессор с реализацией технологии Hyper-threading (HT).

Из табл.2 видно, что время решения задачи на Pentium4 с применением OpenMP отличается незначительно от времени решения без ее применения, поскольку здесь не используется технология HT. Для применения этой технологии в полной мере необходимо использовать специальные API, описание которых можно найти на сайтах компаний Intel и Microsoft.

Для повышения эффективности программы умножения двух матриц, можно учитывать наличие кэш-памяти компьютера (кэш первого уровня). Когда при выполнении программы необходимо оперировать данными и они не находятся в кэш-памяти (кэш-промах), то процессор должен обращаться к оперативной памяти (ОП), что увеличивает временные затраты на выполнение задачи.

При проведении эксперимента было выявлено, что с ростом размера матриц теряется эффективность использования кеш-памяти, поскольку длина строк и столбцов матриц превосходят размеры кеш-памяти. Однако, если построить блочный алгоритм умножения матриц, то можно получить кратчайшее время выполнения этой операции: матрицы, которые умножаются, могут быть представлены как составленные из блоков, и результирующая матрица будет получена путем выполнения умножения над блоками. Если размер блока соизмерим с размером кэш-памяти, то время вычисления произведения матриц уменьшится.

Рассмотрим еще один пример применения метода инкрементного программирования при создании ПО с использованием CUDA на графических процессорах для операции умножения двух матриц $C=A\times B$.

Как и в случае программирования с помощью технологии OpenMP, сначала необходимо выделить в алгоритме математические операции над массивами, которые могут быть распараллелены на графических процессорах, а затем шаг за шагом добавить новые команды и функции CUDA, описывающие параллельные конструкции.

В программах, написанных на языке Си с использованием технологии CUDA, помимо основной памяти CPU используется глобальная память (global), разделяемая память (shared) и другие.

Глобальная память – это память, которая выделяется на DRAM GPU. Для использования этой памяти программа должна с помощью функций CUDA SDK выделять (захватывать) память, выполнять двухстороннее копирование между CPU и GPU.

CUDA-функции выделения глобальной памяти под массивы имеют вид:

```
cudaMalloc ( (void**)&adev, numBytes );
cudaMalloc ( (void**)&bdev, numBytes );
cudaMalloc ( (void**)&cdev, numBytes );
```

CUDA-функции копирования массивов данных из CPU на GPU имеют вид:

```
cudaMemcpy( adev, a, numBytes, cudaMemcpyHostToDevice );
cudaMemcpy( bdev, b, numBytes, cudaMemcpyHostToDevice );
```

CUDA- функции копирования массивов данных из GPU на CPU имеют вид:

```
cudaMemcpy( c, cdev, numBytes, cudaMemcpyDeviceToHost );
```

Пример программы на языке Си с приведенными выше функциями CUDA которая функционирует на CPU, имеет вид:

```
...
#include <cuda_runtime.h>
#define BLOCK_SIZE 16
#define N 1024   //порядок матриц

int main ( int argc, char *  argv [] )
{
    int numBytes = N * N * sizeof ( float );
        // выделение памяти под массиви на CPU
    float * a = new float [N*N];
    float * b = new float [N*N];
```

```
        float * c = new float [N*N];
        // инициализация массивов
        for ( int i = 0; i < N; i++ )
            for ( int j = 0; j < N; j++ )
            {
                a [i] = 2.0f;
                b [i] = 1.0f;
            }
           // выделение памяти под массиви на GPU
        float * adev = NULL;
        float * bdev = NULL;
        float * cdev = NULL;
        cudaMalloc ( (void**)&adev, numBytes );
        cudaMalloc ( (void**)&bdev, numBytes );
        cudaMalloc ( (void**)&cdev, numBytes );
          // конфигурация сетки
        dim3 threads ( BLOCK_SIZE, BLOCK_SIZE );
        dim3 blocks  ( N / threads.x, N / threads.y);
   //старт cudaEvent функций, с помощью которых отслеживается
//завершение работы программ на GPU и определяется время //выполнения за-
дачи на GPU
        cudaEvent_t start, stop;
        float gpuTime = 0.0f;
        cudaEventCreate ( &start );
        cudaEventCreate ( &stop );
        cudaEventRecord ( start, 0 );
        // копирование массивов из CPU на GPU
        cudaMemcpy      ( adev, a, numBytes, cudaMemcpyHostToDevice );
        cudaMemcpy      ( bdev, b, numBytes, cudaMemcpyHostToDevice );
          // Операция умножения подматриц выполняется
          // в global-функции matrixMult
        matrixMult <<<blocks, threads>>> ( adev, bdev, cdev, N );
               // копирование массивов из CPU на GPU
        cudaMemcpy      ( c, cdev, numBytes, cudaMemcpyDeviceToHost );
             //прекращение работы cudaEvent функций,
        cudaEventRecord ( stop, 0 );
        cudaEventSynchronize ( stop );
        cudaEventElapsedTime ( &gpuTime, start, stop );
             // печать времени выполнения задачи на GPU
        printf("time spent executing by the GPU: %.2f millseconds\n", gpu-
Time );
             // освобождение ресурсов
        cudaEventDestroy ( start );
        cudaEventDestroy ( stop  );
        cudaFree         ( adev  );
        cudaFree         ( bdev  );
        cudaFree         ( cdev  );
        delete a;
        delete b;
        delete c;
        return 0;
   }
```

Пример global-функции matrixMul, которая реализует массивно-параллельные операции умножения матриц с использованием global-памяти имеет следующий вид :

```
global__ void matrixMult(float *a,float *b,float *c)
{
    int bx = blockIdx.x; //индексы блока
    int by = blockIdx.y;
  int tx= threadIdx.x; //индексы нити внутри блока
  int ty= threadIdx.y;
  float sum =0.0f;  //накопление результата

                    // смещение  для  a[i][0]
  int ia = n * blockSize * by + n * ty;
                    // смещение для  b[0][j]
     int ib = blockSize * bx + tx;
                    // умножаем и вычисляем сумму
     for (int  k = 0; k < n; k++)
    sum += a[ia + k] * b[ib + k * n];
        // сохраняем смещение вычисленного элемента
 // в глобальной памяти
         int ic = n * blockSize * by + blockSize * bx;
    c[ic + n * ty + tx] = sum;
 }
}
```

Однако global-память не имеет высокого быстродействия, чаще всего она используется для сохранения больших массивов данных, доступ к которым осуществляется как можно реже.

Для нахождения одного элемента результирующей матрицы нам необходимо выполнить чтение $2\times N$ значений из global-памяти и исполнить $2\times N$ арифметических операций, что является очень затратным по времени. Для уменьшения времени выполнения можно внести изменения в исходный код global-функции, применяя shared-память.

Shared-память представлена в виде блоков, которые находятся непосредственно в потоковом мультипроцессоре. Каждому блоку выделяется 16 Кбайт shared-памяти, доступ к которой и операции над массивами имеют очень высокое быстродействие. В этом случае результирующая подматрица C^* является суммой произведений подматриц [6]:

$$C^* = A_1^* \times B_1^* + A_2^* \times B_2^* + \ldots + A_{N/16}^* \times B_{N/16}^*$$

В таком случае global-функции matrixMul, которая реализует этот алгоритм массивно-параллельных операций умножения матриц на shared-памяти имеет вид :

```
__global__ void matrixMult( float* a, float* b,  float* c, int n)
{
    // Номер блока
    int bx = blockIdx.x;
    int by = blockIdx.y;
    // Номер нити
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    // Индекс начала первой подматрицы A, которая обрабатывается блоком
    int aBegin = n * BLOCK_SIZE * by;
     // Индекс конца первой подматрицы A, которая обрабатывается блоком

   int aEnd   = aBegin + n - 1;
    // Шаг перебора подматриц A
    int aStep  = BLOCK_SIZE;
    // Индекс начала первой подматрицы B, которая обрабатывается блоком
    int bBegin = BLOCK_SIZE * bx;
    // Шаг перебора подматриц B
    int bStep  = BLOCK_SIZE * n;
float sum=0.0f;  //Элемент, который вычисляется
    // Цикл по 16X16  подматрицам A и B
```

```
for (int ia = aBegin, ib = bBegin; ia <= aEnd;  ia += aStep, ib += bStep)
    {
        // Очередная подматрица A  в shared-памяти
        __shared__ float as[BLOCK_SIZE][BLOCK_SIZE];
        // Очередная подматрица B  в shared-памяти
        __shared__ float bs[BLOCK_SIZE][BLOCK_SIZE];
        // Загрузить по одному элементу из A и B в shared-память
        as[ty][tx] = a[ia + n * ty + tx];
        bs[ty][tx] = b[ib + n * ty + tx];
        // Дождаться, когда обе подматрицы будут подностью загружены
        __syncthreads();
       // Вычисляем элемент произведения загруженных подматриц
            for (int k = 0; k < BLOCK_SIZE; k++ )
                sum += as[ty][k] * bs[k][tx];
        // Дождаться, когда все нити блока закончат вычисления
        __syncthreads();
     }
     // Записать результат в массив, который содержит матрицу c
       int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    c[ic + n * ty + tx] = sum;
}
```

Исходя из выше приведеного исходного кода вычисления произведения матриц, для определения одного элемента результирующей матрици необходимо выполнить $2\times N/16$ чтений из глобальной памяти и исполнить $2\times N$ арифметических операций. Используя на компьютере графические процессоры, можно решить задачу в десятки раз быстрее. Можно сделать вывод, что использование shared-памяти существенно уменьшает сумарное время обращения к global-памяти.

Вывод

Время решения одной задачи на многоядерных ПК значительно сокращается при распараллеливании ПО с помощью технологии OpenMP или других средств параллельного программирования. Параллельное программирование – это технология будущего. Применение технологии инкрементного программирования упрощает работу пользователя как при написании параллельных программ на OpenMP или CUDA, так и при их отладке. Кроме того, это дает возможность лучше понять суть параллельного программирования и подготовиться к работе на параллельных компьютерах кластерного типа.

Литература

1. http://www.mpiforum.org/
2. http:// www.epm.ornl.gov/pvm/
3. http://ru.wikipedia.org/wiki/openmp
4. http://www.openmp.org/
5. «Основы работы с технологией CUDA» А.В. Боресков, А.А. Харламов, –М.: ДМК Пресс, 2010 год.
6. http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx
7. http://ati.amd.com/products/streamprocessor/specs.html
8. Антонов А.С. Параллельное программирование с использованием технологии OpenMP, издательство МГУ, 2009