МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА
(национальный исследовательский университет)»

РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ РЕШЕНИЯ ЗАДАЧ ВЫСОКОЙ ВЫЧИСЛИТЕЛЬНОЙ СЛОЖНОСТИ В СРЕДАХ MPI, OPEN MP И CUDA

Методические материалы
(описание методик и инструментальных средств)

САМАРА 2010

УДК 004.75

Составители: Никоноров Артем Владимирович,
Фурсов Владимир Алексеевич

В методическом пособии рассмотрены основы массивно-многопоточного программирования. В частности, рассматриваются вопросы многопоточного программирования для многоядерных и многопроцессорных систем с общей памятью с использованием библиотеки OpenMP, основы разработки программ для кластерных систем с распределенной памятью на основе парадигмы передачи сообщений, а также новое направление в многопоточном программировании – GPGPU (General Purpose computation on Graphic Processing Units) – разработка высокопроизводительных программ общего назначения на графических процессорах и технология разработки массивно многопоточных программ для графических процессоров Nvidia с использованием технологии CUDA.

Методические материалы предназначены для студентов специальностей и направлений «Прикладная математика и информатика», «Прикладные математика и физика».

УДК 004.75

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Среди специалистов, занимающихся параллельными вычислениями, популярна шутка “Параллельные вычисления - технология будущего: и так будет всегда”. Эта шутка не теряет актуальность уже несколько десятилетий. Аналогичные настроения были распространены в сообществе разработчиков архитектур компьютеров, обеспокоенном тем, что скоро будет достигнут предел тактовой частоты процессоров, однако частоты процессоров продолжают повышаться, хотя гораздо медленнее, чем раньше. Сплав оптимизма специалистов по параллельным вычислениям и пессимизма архитекторов систем способствовал появлению революционных многоядерных процессоров. Благодаря этому, аппаратное обеспечение современных вычислительных систем поддерживает параллельные вычисления (одновременное выполнение нескольких процессов). Параллелизм на уровне команд, в свою очередь, определяет производительность процессора. Современные кристаллы процессоров содержат несколько ядер, а в типовых серверах устанавливается несколько процессоров. Стремительно развивается использование серверных кластеров и систем распределенных вычислений. Одним словом, параллельные вычисления становятся неотъемлемой частью нашей жизни, поэтому разработчикам ПО стоит взяться за освоение новых парадигм программирования.

Для начала необходимо познакомиться с аппаратным обеспечением, поддерживающим параллельные вычисления. Четверть века разработчики суперкомпьютеров создают высокопроизводительные системы путем объединения отдельных вычислительных систем. Можно выделить две основные архитектуры таких систем - ***векторно-конвейерные ПК с распределенной памятью*** и ***векторно-конвейерные ПК с общей памятью***. Каждый процессорный элемент векторно-конвейерной ПК имеет собственные потоки команд и потоки данных. Термин "ПК с совместно используемой памятью" используется применительно к системе, процессорные элементы которой используют единое адресное пространство. Если же каждому процессорному элементу соответствует отдельная область памяти, с которой он общается по сети, речь идет о системе с распределенной памятью.

Обе архитектуры имеют определенные достоинства, а выбор между ними зависит от поставленной задачи, которую необходимо решить. Благодаря развитию многоядерных процессоров для персональных компьютеров и многопроцессорных систем, в ближайшие годы ожидается рост количества систем с совместно используемой памятью. В ближайшем будущем, даже в карманные ПК и телефоны будут устанавливаться многоядерные процессоры, а программисты получат возможность работы с платформами, оптимизированными для параллельных вычислений.

Какие преимущества дают разработчикам ПО системы параллельных вычислений? Здесь необходимо рассмотреть два аспекта, зависящие от конкретных потребностей. Во-первых, это возможность выполнения нескольких задач за меньшее время, то есть повышенная производительность системы. Объясняется она тем, что каждый процессор занимается обработкой отдельной задачи. В таком случае разработчику остается проконтролировать работу планировщика заданий (обычно уже встроенного в ОС), который занимается распределением нагрузки между процессорами. Во-вторых, это выполнение отдельных задач за меньшее время. Например, современные условия коммерческой деятельности обязывают иметь возможность быстрых вычислений для контроля экономической ситуации в режиме реального времени. Таким образом, решение конкретной задачи должно проводиться за меньшее время. Использование параллелизма в рамках единой задачи с целью исполнения ее за меньшее время называют "параллельным программированием".

Долгое время параллельным программированием могли заниматься только разработчики программного обеспечения для высокопроизводительных вычислительных систем. Времена изменились, и теперь каждому разработчику приложений стоит разобраться в принципах параллельных вычислений. Однако создание программ, поддерживающих параллельные вычисления, является непростой задачей.

Разработчики ПО для высокопроизводительных вычислительных систем имеют многолетний опыт параллельного программирования. Им хорошо знакомы параллельные алгоритмы, параллельное программирование прикладных программных интерфейсов, а также инструменты для разработчиков. Данные методические указания предназначены в помощь программистам, которые только начинают осваивать разработку приложений с параллельными потоками команд.

Настоящее пособие состоит из трех частей. В первой части рассмотрены основы программирования многопоточного программирования для многоядерных и многопроцессорных систем с общей памятью с использованием библиотеки OpenMP.

Во второй части рассмотрены основы разработки программ для кластерных систем с распределенной памятью на основе парадигмы передачи сообщений. Многолетним стандартом для такой разработки является разработка с использованием библиотеки MPI.

Наконец третья часть посвящена новому направлению в многопоточном программировании – GPGPU. General Purpose computation on Graphic Processing Units – разработка высокопроизводительных программ общего назначения на графических процессорах. Будет рассмотрена технология разработки массивно многопоточных программ для графических процессоров Nvidia с использованием технологии CUDA.

1. Многопоточное программирование в стандарте OpenMP

Развитие многоядерных процессоров впечатляет, однако, если ваше приложение не будет использовать несколько ядер, его быстродействие никак не изменится. Именно здесь и вступает в игру технология OpenMP, которая помогает программистам на C++ быстрее создавать многопоточные приложения.
Настоящий раздел является введением, где демонстрируется применение различных средств OpenMP для быстрого написания многопоточных программ. Если вам понадобится дополнительная информация по этой тематике, мы рекомендуем обратиться к спецификации, доступной на сайте OpenMP (www.openmp.org), - она на удивление легко читается.

Активизация OpenMP в Visual C++.

Стандарт OpenMP был разработан в 1997 г. как API, ориентированный на написание портируемых многопоточных приложений. Сначала он был основан на языке Fortran, но позднее включил в себя и C/C++. Последняя версия OpenMP - 2.0; ее полностью поддерживает Visual C++ 2005. Стандарт OpenMP поддерживается также и компилятором GCC начиная с 4 версии.
В VisualC++ реализованные в компиляторе средства OpenMP активизируются опцией компилятора /openmp. Можно активизировать директивы OpenMP на страницах свойств проекта, выбрав Configuration Properties, C/C++, Language и изменив значение свойства OpenMP Support. Встретив параметр /openmp, компилятор определяет символ _OPENMP, с помощью которого можно выяснить, включены ли средства OpenMP. Для этого достаточно написать #ifndef _OPENMP.
OpenMP связывается с приложениями через библиотеку импорта vcomp.lib. Соответствующая библиотека периода выполнения называется vcomp.dll. Отладочные версии библиотек импорта и периода выполнения (vcompd.lib и vcompd.dll соответственно) поддерживают дополнительные сообщения об ошибках, генерируемых при некоторых недопустимых операциях. Visual C++ не поддерживает статическое связывание с библиотекой OpenMP.

Параллельная обработка в OpenMP

Работа OpenMP-приложения начинается с единственного потока - основного. В приложении могут содержаться параллельные регионы, входя в которые, основной поток создает группы потоков (включающие основной поток). В конце параллельного региона группы потоков останавливаются, а выполнение основного потока продолжается. В параллельный регион могут быть вложены другие параллельные регионы, в которых каждый поток первоначального региона становится основным для своей группы потоков.

Вложенные регионы могут в свою очередь включать регионы более глубокого уровня вложенности.
Параллельную обработку в OpenMP иллюстрирует рис. 1. Самая левая стрелка представляет основной поток, который выполняется в одиночестве, пока не достигает первого параллельного региона в точке 1. В этой точке основной поток создает группу потоков, и теперь все они одновременно выполняются в параллельном регионе.

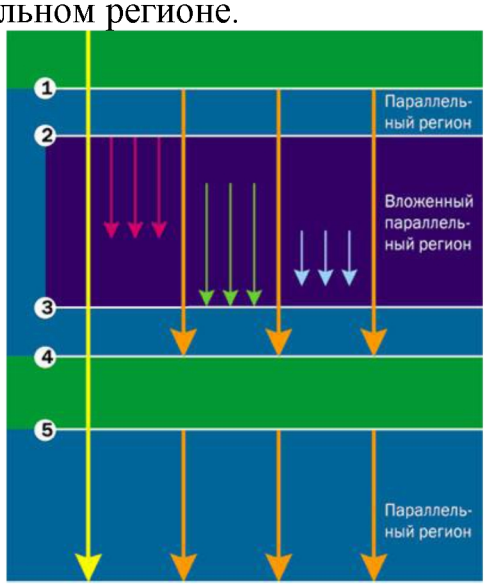

Рис. 1. Параллельные разделы OpenMP

В точке 2 три из этих четырех потоков, достигнув вложенного параллельного региона, создают новые группы потоков. Исходный основной и потоки, создавшие новые группы, становятся владельцами своих групп (основными в этих группах).
В точке 3 вложенный параллельный регион завершается. Каждый поток вложенного параллельного региона синхронизирует свое состояние с другими потоками в этом регионе, но синхронизация разных регионов между собой не выполняется. В точке 4 заканчивается первый параллельный регион, а в точке 5 начинается новый. Локальные данные каждого потока в промежутках между параллельными регионами сохраняются. Далее рассматривается, с чего начать разработку параллельного приложения.

1.1 Конструкции OpenMP

OpenMP прост в использовании и включает лишь два базовых типа конструкций: директивы pragma и функции исполняющей среды OpenMP. Директивы pragma, как правило, указывают компилятору реализовать параллельное выполнение блоков кода. Все эти директивы начинаются с #pragma omp. Как и любые другие директивы pragma, они игнорируются компилятором, не поддерживающим конкретную технологию - в данном случае OpenMP.

Функции OpenMP служат в основном для изменения и получения параметров среды. Кроме того, OpenMP включает API-функции для поддержки некоторых типов синхронизации. Чтобы задействовать эти функции библиотеки OpenMP периода выполнения (исполняющей среды), в программу нужно включить заголовочный файл omp.h. Если вы используете в приложении только OpenMP-директивы pragma, включать этот файл не требуется.
Для реализации параллельного выполнения блоков приложения нужно просто добавить в код директивы pragma и, если нужно, воспользоваться функциями библиотеки OpenMP периода выполнения. Директивы pragma имеют следующий формат:

```
#pragma omp <директива> [раздел [ [,] раздел]...]
```

OpenMP поддерживает директивы parallel, for, parallel for, section, sections, single, master, critical, flush, ordered и atomic, которые определяют или механизмы разделения работы или конструкции синхронизации. Далее мы обсудим большинство директив.

Раздел (clause) - это необязательный модификатор директивы, влияющий на ее поведение. Списки разделов, поддерживаемые каждой директивой, различаются, а пять директив (master, critical, flush, ordered и atomic) вообще не поддерживают разделы.

1.1.1 Реализация параллельной обработки

Хотя директив OpenMP много, все они сразу нам не понадобятся. Самая важная и распространенная директива - parallel. Она создает параллельный регион для следующего за ней структурированного блока, например:

```
#pragma omp parallel [раздел[ [,] раздел]...]
структурированный блок
```

Эта директива сообщает компилятору, что структурированный блок кода должен быть выполнен параллельно, в нескольких потоках. Каждый поток будет выполнять один и тот же поток команд, но не один и тот же набор команд - все зависит от операторов, управляющих логикой программы, таких как if-else.
В качестве примера рассмотрим классическую программу <Hello World>:

```
#pragma omp parallel
{
    printf("Hello World\n");
}
```

В двухпроцессорной системе вы, конечно же, рассчитывали бы получить следующее:

```
Hello World
Hello World
```

Тем не менее, результат мог бы оказаться и таким:

```
HellHell oo WorWlodrl
d
```

Второй вариант возможен из-за того, что два выполняемых параллельно потока могут попытаться вывести строку одновременно. Когда два или более потоков одновременно пытаются прочитать или изменить *общий ресурс* (в нашем случае им является окно консоли), возникает вероятность *гонок* (*race condition*). Это недетерминированные ошибки в коде программы, найти которые крайне трудно. За предотвращение гонок отвечает программист; как правило, для этого используют блокировки или сводят к минимуму обращения к общим ресурсам.

Давайте взглянем на более серьезный пример, который определяет средние значения двух соседних элементов массива и записывает результаты в другой массив. В этом примере используется новая OpenMP-конструкция #pragma omp for, которая относится к *директивам разделения работы* (*work-sharing directive*). Такие директивы применяются не для параллельного выполнения кода, а для логического распределения группы потоков, чтобы реализовать указанные конструкции управляющей логики. Директива #pragma omp for сообщает, что при выполнении цикла for в параллельном регионе итерации цикла должны быть распределены между потоками группы:

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

Если бы этот код выполнялся на четырехпроцессорном компьютере, а у переменной size было бы значение 100, то выполнение итераций 1-25 могло бы быть поручено первому процессору, 26-50 - второму, 51-75 - третьему, а 76-99 - четвертому. Это характерно для политики планирования, называемой статической. Политики планирования мы обсудим позднее.

Следует отметить, что в конце параллельного региона выполняется барьерная синхронизация (barrier synchronization). Иначе говоря, достигнув

конца региона, все потоки блокируются до тех пор, пока последний поток не завершит свою работу.
Если из только что приведенного примера исключить директиву #pragma omp for, каждый поток выполнит полный цикл for, проделав много лишней работы:

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

Так как циклы являются самыми распространенными конструкциями, где выполнение кода можно распараллелить, OpenMP поддерживает сокращенный способ записи комбинации директив #pragma omp parallel и #pragma omp for:

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

В этом цикле нет зависимостей, т. е. одна итерация цикла не зависит от результатов выполнения других итераций. А вот в двух следующих циклах есть два вида зависимости:

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

for(int i = 0; i < n; ++i)      // цикл 2
    x[i] = x[i+1] + b[i];
```

Распараллелить цикл 1 проблематично потому, что для выполнения итерации i нужно знать результат итерации i-1, т. е. итерация i зависит от итерации i-1. Распараллелить цикл 2 тоже проблематично, но по другой причине. В этом цикле вы можете вычислить значение x[i] до x[i-1], однако, сделав так, вы больше не сможете вычислить значение x[i-1]. Наблюдается зависимость итерации i-1 от итерации i.
При распараллеливании циклов вы должны убедиться в том, что итерации цикла не имеют зависимостей. Если цикл не содержит зависимостей, компилятор может выполнять цикл в любом порядке, даже параллельно. Соблюдение этого важного требования компилятор не проверяет - вы сами должны заботиться об этом. Если вы укажете компилятору распараллелить цикл, содержащий зависимости, компилятор подчинится, что приведет к ошибке.

Кроме того, OpenMP налагает ограничения на циклы for, которые могут быть включены в блок #pragma omp for или #pragma omp parallel for block. Циклы for должны соответствовать следующему формату:

```
for([целочисленный тип] i = инвариант цикла;
    i {<,>,=,<=,>=} инвариант цикла;
    i {+,-}= инвариант цикла)
```

Эти требования введены для того, чтобы OpenMP мог при входе в цикл определить число итераций.

1.1.2 Сравнение поддержки потоков в OpenMP и Win32

Сравним только что приведенный пример, включающий директиву #pragma omp parallel for, с кодом, который пришлось бы написать для решения той же задачи на основе Windows API. Как видно в листинге 1, для достижения того же результата требуется гораздо больше кода, а за кулисами в этом варианте выполняются еще кое-какие операции. Так, конструктор класса ThreadData определяет, какими должны быть значения start и stop при каждом вызове потока. OpenMP обрабатывает все эти детали сам и предоставляет программисту дополнительные средства конфигурирования параллельных регионов и кода.

Листинг 1. Многопоточность в Win32

```
class ThreadData {
public:
   // Конструктор инициализирует поля start и stop
   ThreadData(int threadNum);
   int start;
   int stop;
};

DWORD ThreadFn(void* passedInData)
{
   ThreadData *threadData = (ThreadData *)passedInData;
   for(int i = threadData->start; i < threadData->stop;
++i )
      x[i] = (y[i-1] + y[i+1]) / 2;
   return 0;
}

void ParallelFor()
{
```

```
   // Запуск групп потоков
   for(int i=0; i < nTeams; ++i)
      ResumeThread(hTeams[i]);

   // Для каждого потока здесь неявно вызывается
   // метод ThreadFn

   // Ожидание завершения работы
   WaitForMultipleObjects(nTeams,     hTeams,     TRUE,
INFINITE);
}

int main(int argc, char* argv[])
{
   // Создание групп потоков
   for(int i=0; i < nTeams; ++i)
   {
      ThreadData *threadData = new ThreadData(i);
      hTeams[i]   =   CreateThread(NULL,   0,   ThreadFn,
threadData,
          CREATE_SUSPENDED, NULL);
   }

   ParallelFor();    //    имитация    OpenMP-конструкции
parallel for

   // Очистка
   for(int i=0; i < nTeams; ++i)
      CloseHandle(hTeams[i]);
}
```

1.1.3 Общие и частные данные

Разрабатывая параллельные программы, необходимо разделять общие (shared) данные и частные (private) данные, - от этого зависит не только производительность, но и корректная работа программы. В OpenMP это различие очевидно, и может быть настроено программистом.

Общие переменные доступны всем потокам из группы, поэтому изменения таких переменных в одном потоке видимы другим потокам в параллельном регионе. Что касается частных переменных, то каждый поток из группы располагает их отдельными экземплярами, поэтому изменения таких переменных в одном потоке никак не сказываются на их экземплярах, принадлежащих другим потокам.

По умолчанию все переменные в параллельном регионе - общие, но из этого правила есть три исключения. Во-первых, частными являются индексы параллельных циклов for. Например, это относится к переменной i в коде, показанном в листинге 2. Переменная j по умолчанию не является частной, но явно сделана таковой через раздел firstprivate.

Листинг 2. Разделы директив OpenMP и вложенный цикл for

```
float sum = 10.0f;
MatrixClass myMatrix;
int j = myMatrix.RowStart();
int i;
#pragma omp parallel
{
   #pragma omp for firstprivate(j) lastprivate(i)
      reduction(+: sum)

   for(i = 0; i < count; ++i)
   {
      int doubleI = 2 * i;
      for(; j < doubleI; ++j)
      {
         sum += myMatrix.GetElement(i, j);
      }
   }
}
```

Во-вторых, частными являются локальные переменные блоков параллельных регионов. На рис. 3 такова переменная doubleI, потому что она объявлена в параллельном регионе. Любые нестатические и не являющиеся членами класса MatrixClass переменные, объявленные в методе myMatrix::GetElement, будут частными.

В-третьих, частными будут любые переменные, указанные в разделах private, firstprivate, lastprivate и reduction. В листинге 2 переменные i, j и sum сделаны частными для каждого потока из группы, т. е. каждый поток будет располагать своей копией каждой из этих переменных.

Каждый из четырех названных разделов принимает список переменных, но семантика этих разделов различается. Раздел private говорит о том, что для каждого потока должна быть создана частная копия каждой переменной из списка. Частные копии будут инициализироваться значением по умолчанию (с применением конструктора по умолчанию, если это уместно). Например, переменные типа int имеют по умолчанию значение 0.

У раздела firstprivate такая же семантика, но перед выполнением параллельного региона он указывает копировать значение частной переменной в каждый поток, используя конструктор копий, если это уместно.

Семантика раздела lastprivate тоже совпадает с семантикой раздела private, но при выполнении последней итерации цикла или раздела конструкции распараллеливания значения переменных, указанных в разделе lastprivate, присваиваются переменным основного потока. Если это уместно, для копирования объектов применяется оператор присваивания посредством копирования (copy assignment operator).
Похожая семантика и у раздела reduction, но он принимает переменную и оператор. Поддерживаемые этим разделом операторы перечислены в табл. 1, а у переменной должен быть скалярный тип (например, float, int или long, но не std::vector, int [] и т. д.). Переменная раздела reduction инициализируется в каждом потоке значением, указанным в таблице. В конце блока кода оператор раздела reduction применяется к каждой частной копии переменной, а также к исходному значению переменной. Таким образом, reduction реализует одну из важнейших параллельных конструкций – операцию редукции.

Табл. 1. Операторы раздела reduction

Оператор раздела reduction	Значение по-умолчанию
+	0
*	1
-	0
&	~0 (все биты установлены в 1)
|	0
^	0
&&	1
||	0

В листинге 2 переменная sum неявно инициализируется в каждом потоке значением 0.0f (заметьте, что в таблице указано значение 0, но в данном случае оно принимает форму 0.0f, так как sum имеет тип float). После выполнения блока #pragma omp for над всеми частными значениями и исходным значением sum (которое в нашем случае равно 10.0f) выполняется операция +. Результат присваивается исходной общей переменной sum.

1.Параллельная обработка в конструкциях, отличных от циклов

Как правило, OpenMP используется для распараллеливания циклов, но поддерживает ся параллелизм и на уровне функций. Этот механизм называется *секциями* OpenMP (*OpenMP sections*). Он довольно прост и часто бывает полезен.

Рассмотрим один из самых важных алгоритмов в программировании - быструю сортировку (quicksort). В качестве примера мы реализовали рекурсивный метод быстрой сортировки списка целых чисел. Ради простоты

мы решили не создавать универсальную шаблонную версию метода, но суть дела от этого ничуть не меняется. Код нашего метода, реализованного с использованием секций OpenMP, показан в листинге 3 (код метода Partition опущен, чтобы не загромождать общую картину).

Листинг 3. Быстрая сортировка с использованием параллельных секций

```
void QuickSort (int numList[],  int nLower, int nUpper)
{
   if (nLower < nUpper)
   {
      // Разбиение интервала сортировки
      int nSplit = Partition (numList, nLower, nUpper);
      #pragma omp parallel sections
      {
         #pragma omp section
         QuickSort (numList, nLower, nSplit - 1);

         #pragma omp section
         QuickSort (numList, nSplit + 1, nUpper);
      }
   }
}
```

В данном примере первая директива #pragma создает параллельный регион секций. Каждая секция определяется директивой #pragma omp section. Каждой секции в параллельном регионе ставится в соответствие один поток из группы потоков, и все секции выполняются одновременно. В каждой секции рекурсивно вызывается метод QuickSort.

Как и в случае конструкции #pragma omp parallel for, вы сами должны убедиться в независимости секций друг от друга, чтобы они могли выполняться параллельно. Если в секциях изменяются общие ресурсы без синхронизации доступа к ним, результат может оказаться непредсказуемым.

Обратите внимание на то, что в этом примере используется сокращение #pragma omp parallel sections, аналогичное конструкции #pragma omp parallel for. По аналогии с #pragma omp for директиву #pragma omp sections можно использовать в параллельном регионе отдельно.

По поводу кода, показанного в листинге 3, следует сказать еще пару слов. Прежде всего заметьте, что параллельные секции вызываются рекурсивно. Рекурсивные вызовы поддерживаются и параллельными регионами, и (как в нашем примере) параллельными секциями. Если создание вложенных секций разрешено, по мере рекурсивных вызовов QuickSort будут создаваться все новые и новые потоки. Возможно, это не то, что нужно программисту, так как такой подход может привести к созданию большого числа потоков. Чтобы ограничить число потоков, в программе можно запретить вложение.

Тогда наше приложение будет рекурсивно вызывать метод QuickSort, используя только два потока.
При компиляции этого приложения без параметра /openmp будет сгенерирована корректная последовательная версия. Одно из преимуществ OpenMP в том, что эта технология совместима с компиляторами, не поддерживающими OpenMP.

Директивы pragma для синхронизации

При одновременном выполнении нескольких потоков часто возникает необходимость их синхронизации. OpenMP поддерживает несколько типов синхронизации, помогающих во многих ситуациях.
Один из типов - неявная *барьерная синхронизация*, которая выполняется в конце каждого параллельного региона для всех сопоставленных с ним потоков. Механизм барьерной синхронизации таков, что, пока все потоки не достигнут конца параллельного региона, ни один поток не сможет перейти его границу.
Неявная барьерная синхронизация выполняется также в конце каждого блока #pragma omp for, #pragma omp single и #pragma omp sections. Чтобы отключить неявную барьерную синхронизацию в каком-либо из этих трех блоков разделения работы, укажите раздел nowait:

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

Как видите, этот раздел директивы распараллеливания говорит о том, что синхронизировать потоки в конце цикла for не надо, хотя в конце параллельного региона они все же будут синхронизированы.
Второй тип - явная барьерная синхронизация. В некоторых ситуациях ее целесообразно выполнять наряду с неявной. Для этого включите в код директиву #pragma omp barrier.
В качестве барьеров можно использовать критические секции. В Win32 API для входа в критическую секцию и выхода из нее служат функции EnterCriticalSection и LeaveCriticalSection. В OpenMP для этого применяется директива #pragma omp critical [имя]. Она имеет такую же семантику, что и критическая секция Win32, и опирается на EnterCriticalSection. Вы можете использовать именованную критическую секцию, и тогда доступ к блоку кода является взаимоисключающим только для других критических секций с тем же именем (это справедливо для всего процесса). Если имя не указано, директива ставится в соответствие некоему имени, выбираемому системой.

Доступ ко всем неименованным критическим секциям является взаимоисключающим.
В параллельных регионах часто встречаются блоки кода, доступ к которым желательно предоставлять только одному потоку, - например, блоки кода, отвечающие за запись данных в файл. Во многих таких ситуациях не имеет значения, какой поток выполнит код, важно лишь, чтобы этот поток был единственным. Для этого в OpenMP служит директива #pragma omp single.
Иногда возможностей директивы single недостаточно. В ряде случаев требуется, чтобы блок кода был выполнен основным потоком, - например, если этот поток отвечает за обработку GUI и вам нужно, чтобы какую-то задачу выполнил именно он. Тогда применяется директива #pragma omp master. В отличие от директивы single при входе в блок master и выходе из него нет никакого неявного барьера.
Чтобы завершить все незавершенные операции над памятью перед началом следующей операции, используйте директиву #pragma omp flush, которая эквивалентна внутренней функции компилятора _ReadWriteBarrier.
Учтите, что OpenMP-директивы pragma должны обрабатываться всеми потоками из группы в одном порядке (или вообще не обрабатываться никакими потоками). Таким образом, следующий пример кода некорректен, а предсказать результаты его выполнения нельзя (вероятные варианты - сбой или зависание системы):

```
#pragma omp parallel
{
    if(omp_get_thread_num() > 3)
    {
        #pragma omp single    // код, доступный не всем
потокам
        x++;
    }
}
```

1.1.5 Функции исполняющей среды OpenMP

Помимо уже описанных директив OpenMP поддерживает ряд полезных функций. Они делятся на три обширных категории: функции исполняющей среды, блокировки/синхронизации и работы с таймерами (последние в настоящей работе не рассматриваются). Все эти функции имеют имена, начинающиеся с omp_, и определены в заголовочном файле omp.h.
Функции первой категории позволяют запрашивать и задавать различные параметры операционной среды OpenMP. Функции, имена которых начинаются на omp_set_, можно вызывать только вне параллельных регионов. Все остальные функции можно использовать как внутри параллельных регионов, так и вне таковых.

Чтобы узнать или задать число потоков в группе, используйте функции omp_get_num_threads и omp_set_num_threads. Первая возвращает число потоков, входящих в текущую группу потоков. Если вызывающий поток выполняется не в параллельном регионе, эта функция возвращает 1. Метод omp_set_num_thread задает число потоков для выполнения следующего параллельного региона, который встретится текущему выполняемому потоку. Кроме того, число потоков, используемых для выполнения параллельных регионов, зависит от двух других параметров среды OpenMP: поддержки динамического создания потоков и вложения регионов.

Поддержка динамического создания потоков определяется значением булевого свойства, которое по умолчанию равно false. Если при входе потока в параллельный регион это свойство имеет значение false, исполняющая среда OpenMP создает группу, число потоков в которой равно значению, возвращаемому функцией omp_get_max_threads. По умолчанию omp_get_max_threads возвращает количество потоков, поддерживаемых аппаратно, или значение переменной OMP_NUM_THREADS. Как правило, это количество равно числу реальных или виртуальных ядер в системе. Например, для процессора Core2Duo это количество равно 2, для процессора Core i7 с четырьмя ядрами и поддержкой HyperTreading это количество будет равно 8.

Если поддержка динамического создания потоков включена, исполняющая среда OpenMP создаст группу, которая может содержать переменное число потоков, не превышающее значение, которое возвращается функцией omp_get_max_threads.

Вложение параллельных регионов также определяется булевым свойством, которое по умолчанию установлено в false. Вложение параллельных регионов происходит, когда потоку, уже выполняющему параллельный регион, встречается другой параллельный регион. Если вложение разрешено, создается новая группа потоков, при этом соблюдаются правила, описанные ранее. А если вложение не разрешено, формируется группа, содержащая один поток.

Для установки и чтения свойств, определяющих возможность динамического создания потоков и вложения параллельных регионов, служат функции omp_set_dynamic, omp_get_dynamic, omp_set_nested и omp_get_nested. Кроме того, каждый поток может запросить информацию о своей среде. Чтобы узнать номер потока в группе потоков, вызовите omp_get_thread_num. Помните, что она возвращает не Windows-идентификатор потока, а число в диапазоне от 0 до omp_get_num_threads - 1.

Функция omp_in_parallel позволяет потоку узнать, выполняет ли он в настоящее время параллельный регион, а omp_get_num_procs возвращает число процессоров в компьютере.

Взаимосвязи различных функций исполняющей среды демонстрирует листинг 4. В этом примере реализованы четыре отдельных параллельных региона и два вложенных.

Листинг 4. Использование подпрограмм исполняющей среды OpenMP

```
#include <stdio.h>
#include <omp.h>

int main()
{
   omp_set_dynamic(1);
   omp_set_num_threads(10);
   #pragma omp parallel        // параллельный регион 1
   {
      #pragma omp single
      printf("Num threads in dynamic region is = %d\n",
         omp_get_num_threads());
   }
   printf("\n");
   omp_set_dynamic(0);
   omp_set_num_threads(10);
   #pragma omp parallel        // параллельный регион 2
   {
      #pragma omp single
      printf("Num threads in non-dynamic region is =
%d\n",
         omp_get_num_threads());
   }
   printf("\n");
   omp_set_dynamic(1);
   omp_set_num_threads(10);
   #pragma omp parallel        // параллельный регион 3
   {
      #pragma omp parallel
      {
         #pragma omp single
         printf(
            "Num threads in nesting disabled region is
= %d\n",
            omp_get_num_threads());
      }
   }
   printf("\n");
   omp_set_nested(1);
   #pragma omp parallel        // параллельный регион 4
   {
      #pragma omp parallel
      {
```

```
            #pragma omp single
            printf("Num threads in nested region is =
%d\n",
                omp_get_num_threads());
        }
    }
}
```

Скомпилировав этот код в Visual Studio 2005 и выполнив его на обычном двухъядерном компьютере, получим такой результат:

```
Num threads in dynamic region is = 2

Num threads in non-dynamic region is = 10

Num threads in nesting disabled region is = 1
Num threads in nesting disabled region is = 1

Num threads in nested region is = 2
Num threads in nested region is = 2
```

Для первого региона мы включили динамическое создание потоков и установили число потоков в 10. По результатам работы программы видно, что при включенном динамическом создании потоков исполняющая среда OpenMP решила создать группу, включающую всего два потока, так как у компьютера два процессора. Для второго параллельного региона исполняющая среда OpenMP создала группу из 10 потоков, потому что динамическое создание потоков для этого региона было отключено.
Результаты выполнения третьего и четвертого параллельных регионов иллюстрируют следствия включения и отключения возможности вложения регионов. В третьем параллельном регионе вложение было отключено, поэтому для вложенного параллельного региона не было создано никаких новых потоков - и внешний, и вложенный параллельные регионы выполнялись двумя потоками. В четвертом параллельном регионе, где вложение было включено, для вложенного параллельного региона была создана группа из двух потоков (т. е. в общей сложности этот регион выполнялся четырьмя потоками). Процесс удвоения числа потоков для каждого вложенного параллельного региона может продолжаться, пока вы не исчерпаете пространство в стеке. На практике можно создать несколько сотен потоков, хотя связанные с этим издержки легко перевесят любые преимущества. Следует отметить, что реальный прирост производительности в системе возможен для числа потоков, сравнимых с числом ядер в системе. Если это число будет значительно выше, то вместо прироста мы получим снижение производительности.

1.2 Эффективное использование OpenMP

1.2.1 Методы синхронизации/блокировки

OpenMP включает и функции, предназначенные для синхронизации кода. В OpenMP два типа блокировок: простые и вкладываемые (nestable); блокировки обоих типов могут находиться в одном из трех состояний - неинициализированном, заблокированном и разблокированном.
Простые блокировки (omp_lock_t) не могут быть установлены более одного раза, даже тем же потоком. Вкладываемые блокировки (omp_nest_lock_t) идентичны простым с тем исключением, что, когда поток пытается установить уже принадлежащую ему вкладываемую блокировку, он не блокируется. Кроме того, OpenMP ведет учет ссылок на вкладываемые блокировки и следит за тем, сколько раз они были установлены.
OpenMP предоставляет подпрограммы, выполняющие операции над этими блокировками. Каждая такая функция имеет два варианта: для простых и для вкладываемых блокировок. Вы можете выполнить над блокировкой пять действий: инициализировать ее, установить (захватить), освободить, проверить и уничтожить. Все эти операции очень похожи на Win32-функции для работы с критическими секциями, и это не случайность: на самом деле технология OpenMP реализована как оболочка этих функций. Соответствие между функциями OpenMP и Win32 иллюстрирует табл. 2.

Табл. 2. Функции для работы с блокировками в OpenMP и Win32

Простая блокировка OpenMP	Вложенная блокировка OpenMP	Win32-функция
omp_lock_t	omp_nest_lock_t	CRITICAL_SECTION
omp_init_lock	omp_init_nest_lock	InitializeCriticalSection
omp_destroy_lock	omp_destroy_nest_lock	DeleteCriticalSection
omp_set_lock	omp_set_nest_lock	EnterCriticalSection
omp_unset_lock	omp_unset_nest_lock	LeaveCriticalSection
omp_test_lock	omp_test_nest_lock	TryEnterCriticalSection

Для синхронизации кода можно использовать и функции времени исполнения, и директивы синхронизации. Преимущество директив в том, что они прекрасно структурированы. Это делает их более понятными и облегчает поиск мест входа в синхронизированные регионы и выхода из них.
Преимущество функций - гибкость. Например, вы можете передать блокировку в другую функцию и установить/освободить ее в этой функции. При использовании директив это невозможно. Как правило, если вам не нужна гибкость, обеспечиваемая лишь подпрограммами исполняющей среды, лучше использовать директивы синхронизации.

1.2.2 Параллельная обработка структур данных

В листинге 5 показан код двух параллельно выполняемых циклов, в начале которых исполняющей среде неизвестно число их итераций. В первом примере выполняется перебор элементов STL-контейнера std::vector, а во втором - стандартного связанного списка.

Листинг 5. Выполнение заранее неизвестного числа итераций

```
#pragma omp parallel
{
   // Параллельная обработка вектора STL
   std::vector<int>::iterator iter;
   for(iter = xVect.begin(); iter != xVect.end();
++iter)
   {
      #pragma omp single nowait
      {
         process1(*iter);
      }
   }

   // Параллельная обработка стандартного связанного
списка
   for(LList *listWalk = listHead; listWalk != NULL;
         listWalk = listWalk->next)
   {
      #pragma omp single nowait
      {
         process2(listWalk);
      }
   }
}
```

В примере с вектором STL каждый поток из группы потоков выполняет цикл for и имеет собственный экземпляр итератора, но при каждой итерации лишь один поток входит в блок single (такова семантика директивы single). Все действия, гарантирующие однократное выполнение блока single при каждой итерации, берет на себя исполняющая среда OpenMP. Такой способ выполнения цикла сопряжен со значительными издержками, поэтому он полезен, только если в функции process1 выполняется много работы. В примере со связанным списком реализована та же логика.
Стоит отметить, что в примере с вектором STL мы можем до входа в цикл определить число его итераций по значению std::vector.size, что позволяет привести цикл к канонической форме для OpenMP:

```
#pragma omp parallel for
   for(int i = 0; i < xVect.size(); ++i)
      process(xVect[i]);
```

Это существенно уменьшает издержки в период выполнения, и именно такой подход мы рекомендуем применять для обработки массивов, векторов и любых других контейнеров, элементы которых можно перебрать в цикле for, соответствующем канонической форме для OpenMP.

Более сложные алгоритмы планирования

По умолчанию в OpenMP для планирования параллельного выполнения циклов for применяется алгоритм, называемый статическим планированием (static scheduling). Это означает, что все потоки из группы выполняют одинаковое число итераций цикла. Если n - число итераций цикла, а T - число потоков в группе, каждый поток выполнит n/T итераций (если n не делится на T без остатка, ничего страшного). Однако OpenMP поддерживает и другие механизмы планирования, оптимальные в разных ситуациях: динамическое планирование (dynamic scheduling), планирование времени выполнения (runtime scheduling) и управляемое планирование (guided scheduling).

Чтобы задать один из этих механизмов планирования, используйте раздел schedule в директиве #pragma omp for или #pragma omp parallel for. Формат этого раздела выглядит так:

```
schedule(алгоритм планирования[, число итераций])
```

Вот примеры этих директив:

```
#pragma omp parallel for schedule(dynamic, 15)
for(int i = 0; i < 100; ++i)
...
#pragma omp parallel
   #pragma omp for schedule(guided)
```

При динамическом планировании каждый поток выполняет указанное число итераций. Если это число не задано, по умолчанию оно равно 1. После того как поток завершит выполнение заданных итераций, он переходит к следующему набору итераций. Так продолжается, пока не будут пройдены все итерации. Последний набор итераций может быть меньше, чем изначально заданный.

При управляемом планировании число итераций, выполняемых каждым потоком, определяется по следующей формуле:

число_выполняемых_потоком_итераций =
  max(число_нераспределенных_итераций/omp_get_num_threads(),
  число итераций)

Завершив выполнение назначенных итераций, поток запрашивает выполнение другого набора итераций, число которых определяется по только что приведенной формуле. Таким образом, число итераций, назначаемых каждому потоку, со временем уменьшается. Последний набор итераций может быть меньше, чем значение, вычисленное по формуле.
Если указать директиву #pragma omp for schedule(dynamic, 15), цикл for из 100 итераций может быть выполнен четырьмя потоками следующим образом:

Поток 0 получает право на выполнение итераций 1-15
Поток 1 получает право на выполнение итераций 16-30
Поток 2 получает право на выполнение итераций 31-45
Поток 3 получает право на выполнение итераций 46-60
Поток 2 завершает выполнение итераций
Поток 2 получает право на выполнение итераций 61-75
Поток 3 завершает выполнение итераций
Поток 3 получает право на выполнение итераций 76-90
Поток 0 завершает выполнение итераций
Поток 0 получает право на выполнение итераций 91-100

А вот каким может оказаться результат выполнения того же цикла четырьмя потоками, если будет указана директива #pragma omp for schedule(guided, 15):

Поток 0 получает право на выполнение итераций 1-25
Поток 1 получает право на выполнение итераций 26-44
Поток 2 получает право на выполнение итераций 45-59
Поток 3 получает право на выполнение итераций 60-64
Поток 2 завершает выполнение итераций
Поток 2 получает право на выполнение итераций 65-79
Поток 3 завершает выполнение итераций
Поток 3 получает право на выполнение итераций 80-94
Поток 2 завершает выполнение итераций
Поток 2 получает право на выполнение итераций 95-100

Динамическое и управляемое планирование хорошо подходят, если при каждой итерации выполняются разные объемы работы или если одни процессоры более производительны, чем другие. При статическом планировании нет никакого способа, позволяющего сбалансировать нагрузку на разные потоки. При динамическом и управляемом планировании нагрузка распределяется автоматически - такова сама суть этих подходов. Как правило, при управляемом планировании код выполняется быстрее, чем при динамическом, вследствие меньших издержек на планирование.
Последний подход - планирование в период выполнения - это скорее даже не алгоритм планирования, а способ динамического выбора одного из трех

описанных алгоритмов. Если в разделе schedule указан параметр runtime, исполняющая среда OpenMP использует алгоритм планирования, заданный для конкретного цикла for при помощи переменной OMP_SCHEDULE. Она имеет формат <тип[,число итераций]>, например:

set OMP_SCHEDULE=dynamic,8
Планирование в период выполнения дает определенную гибкость в выборе типа планирования, при этом по умолчанию применяется статическое планирование.

1.2.3 Эффективность применения OpenMP

Знать, когда использовать технологию OpenMP, не менее важно, чем уметь с ней работать. Ниже дан ряд рекомендаций по полезному использованию OpenMP.
Целевая платформа является многопроцессорной или многоядерной. Если приложение полностью использует ресурсы одного ядра или процессора, то, сделав его многопоточным при помощи OpenMP, вы почти наверняка повысите его быстродействие.
Приложение должно быть кроссплатформенным. OpenMP - кроссплатформенный и широко поддерживаемый API. А так как он реализован на основе директив pragma, приложение можно скомпилировать даже при помощи компилятора, не поддерживающего стандарт OpenMP.
Выполнение циклов нужно распараллелить. Весь свой потенциал OpenMP демонстрирует при организации параллельного выполнения циклов. Если в приложении есть длительные циклы без зависимостей, OpenMP - идеальное решение.
Перед выпуском приложения нужно повысить его быстродействие. Так как технология OpenMP не требует переработки архитектуры приложения, она прекрасно подходит для внесения в код небольших изменений, позволяющих повысить его быстродействие.
В то же время следует признать, что OpenMP - не панацея от всех бед. Эта технология ориентирована в первую очередь на разработчиков высокопроизводительных вычислительных систем и наиболее эффективна, если код включает много циклов и работает с разделяемыми массивами данных.
Создание как обычных потоков, так и параллельных регионов OpenMP имеет свою цену. Чтобы применение OpenMP стало выгодным, выигрыш в скорости, обеспечиваемый параллельным регионом, должен превосходить издержки на создание группы потоков. В версии OpenMP, реализованной в Visual C++, группа потоков создается при входе в первый параллельный регион. После завершения региона группа потоков приостанавливается, пока не понадобится вновь. За кулисами OpenMP использует пул потоков Windows. Например, для двухпроцессорной системы, максимальный прирост

быстродействия составляет примерно 1.7 от исходного, что типично для двухпроцессорных систем. Причем достигается это значение при достаточно большом количестве параллельно выполняемых итераций. При небольшом количестве итераций многопоточная программа может работать медленнее однопоточной, т.к. для многопоточной версии часть времени расходуется на служебные процедуры обслуживания многопоточности. Само по себе применение OpenMP не гарантирует, что быстродействие вашего кода повысится.

OpenMP-директивы pragma просты в использовании, но не позволяют получать детальные сведения об ошибках. Если вы пишете критически важное приложение, которое должно определять ошибки и корректно восстанавливать нормальную работу, от OpenMP, пожалуй, следует отказаться (по крайней мере, пока). Например, если OpenMP не может создать потоки для параллельных регионов или критическую секцию, поведение программы становится неопределенным. В Visual C++ 2005 исполняющая среда OpenMP какое-то время продолжает пытаться выполнить нужную задачу, после чего сдается. В будущих версиях OpenMP помимо прочего планируется реализовать стандартный механизм уведомления об ошибках.

Еще одна ситуация, в которой следует сохранять бдительность, имеет место при использовании потоков Windows вместе с потоками OpenMP. Потоки OpenMP создаются на основе потоков Windows, поэтому они прекрасно работают в одном процессе. Увы, OpenMP ничего не знает о потоках Windows, созданных другими модулями. Из этого вытекают две проблемы: во-первых, исполняющая среда OpenMP не ведет учет других потоков Windows, а во-вторых, методы синхронизации OpenMP не синхронизируют потоки Windows, потому что они не входят в группы потоков.

Ловушки, в которые можно попасть при использовании OpenMP

Хотя использовать OpenMP совсем несложно, некоторые моменты все же требуют к себе повышенного внимания. Например, индексная переменная самого внешнего параллельного цикла for является частной (private), а индексные переменные вложенных циклов for по умолчанию общие. При работе с вложенными циклами обычно требуется, чтобы индексы внутренних циклов были частными. Используйте для этого раздел private.

Разрабатывая приложения OpenMP, следует быть осторожным при генерации исключений C++. Если приложение генерирует исключение в параллельном регионе, оно должно быть обработано в том же регионе тем же потоком. Иначе говоря, исключение не должно покинуть регион. Как правило, все исключения, которые могут быть сгенерированы в параллельном регионе, следует перехватывать. Если не перехватить исключение в том же параллельном регионе, приложение скорее всего потерпит крах.

Чтобы можно было открыть структурированный блок, выражение

#pragma omp <директива> [раздел]
должно завершаться символом новой строки, а не фигурной скобкой. Директива, заканчивающаяся фигурной скобкой, приведет к ошибке компиляции:

```
// Плохо
#pragma omp parallel
{
// Ошибка компиляции
}

// Хорошо
#pragma omp parallel
{
  // Код
}
```

Отлаживать приложения OpenMP в среде Visual Studio 2005 иногда трудно. В частности, определенные неудобства связаны со входом в параллельный регион и/или с выходом из него нажатием клавиши F10/F11. Это объясняется тем, что компилятор генерирует дополнительный код для вызова исполняющей среды и групп потоков. Отладчик об этом не знает, поэтому то, что вы увидите, может показаться вам странным. Мы рекомендуем установить точку прерывания в параллельном регионе и нажать F5, чтобы достичь ее. Чтобы выйти из параллельного региона, установите точку прерывания вне такого региона и нажмите F5.
При нахождении внутри параллельного региона в окне Threads Window отладчика будет отображаться информация о потоках, выполняемых в группе потоков. Идентификаторы этих потоков будут соответствовать не потокам OpenMP, а лежащим в их основе потокам Windows.
В настоящее время использовать с OpenMP оптимизацию, определяемую профилем (Profile Guided Optimization, PGO), нельзя. К счастью, технология OpenMP основана на директивах pragma, поэтому вы можете скомпилировать свое приложение с параметром /openmp и с PGO и узнать, какой подход более эффективен.

OpenMP и .NET
Высокопроизводительные вычисления мало у кого ассоциируются с .NET, но в Visual C++ 2005 эта ситуация улучшена. Особо стоит отметить то, что совместная работы OpenMP с управляемым C++-кодом возможна. Для этого обеспечена совместимость параметра /openmp с /clr и /clr:OldSyntax. То есть вы можете использовать OpenMP для параллельного выполнения методов .NET-типов, которые подлежат сбору мусора. Учтите, что сейчас параметр

/openmp не совместим ни с /clr:safe, ни с /clr:pure, но мы планируется исправить это.
Сущемтвкет важное ограничение, связанное с применением OpenMP в управляемом коде. Приложение, в котором задействованы средства OpenMP, следует использовать только в одном домене приложения. При загрузке другого AppDomain в процесс с уже загруженной исполняющей средой OpenMP приложение может потерпеть крах.
OpenMP - простая, но мощная технология распараллеливания приложений. Она позволяет реализовать параллельное выполнение как циклов, так и функциональных блоков кода. Она легко интегрируется в существующие приложения и включается/выключается одним параметром компилятора. OpenMP позволяет более полно использовать вычислительную мощь многоядерных процессоров.

2. Технологии параллельного программирования.

2.1 Общая информация о технологии параллельного программирования в стандарте MPI

В последние годы бурно развиваются аппаратные средства, позволяющие параллельно исполнять программный код. В 2002 году Intel выпустила первые процессоры, которые с использованием технологии Hyper-treading позволяли параллельно выполнять два процесса одновременно на одном процессоре. В 2006 был выпущен процессор Intel Core 2 Duo, а в 2007 – Core 2 Quad, число параллельных процессов увеличилось до 4, а в 2008 в процессорах i7 до 8 на процессор. В это же время NVIDIA и ATI запустило в массы технологию CUDA – технологию позволяющую использовать для вычислительных задач мощь графического ускорителя. На графических ускорителях с массивно-многопоточной архитектурой счет параллельных потоков идет на тысячи. В связи с этим, параллельное программирование, раньше используемое в небольшом числе дорогих суперпроектов, сегодня становится актуальным для прикладных программистов.

Параллельное программирование, с точки зрения прикладного программиста, использует несколько различных парадигм (моделей). Любая программа, согласно архитектуре Фон-Неймана, может быть представлена в виде кода, глобальных и локальных данных и ресурсов ввода-вывода. В зависимости от того, разделяют ли параллельно выполняющиеся потоки одни и те же данные и код или нет, получаются принципиально разные парадигмы параллельного программирования.

Модель общей памяти предполагает, что параллельные процессы могут работать с общими глобальными данными. Обмен данными между процессами происходит через глобальные переменные, для исключения коллизий при доступе к которым используется механизм блокировок. Такая модель, как правило, применима для работы в рамках одной многопроцессорной машины – многоядерного персонального компьютера или суперкомпьютера.

Для кластерных систем состоящих из отдельных узлов (возможно многоядерных) связанных сетью, общая память в общем случае не доступна, т.е. возможность доступа к общим переменным для процессов отсутствует. В таком случае используется модель параллельного программирования, основанная на сообщениях. Обмен данными между параллельными процессами выполняется посредством обмена сообщениями.

Параллельное программирование, основанное на событиях, достаточно простая и удобная абстракция, которая к тому же, может также функционировать и на системах с общей памятью. Более того, параллельное программирование, основанное на событиях, с успехом использует технологию удаленного прямого доступа к памяти (RDMA), ставшую в недавнем времени доступной для кластерных систем. Промышленным стандартом, реализующим модель передачи сообщений, является стандарт MPI.

MPI (*Message Passing Interface,* http://www.mpi-forum.org) является технологией создания параллельно выполняющихся программ, основанной на передаче сообщений между процессами (сами процессы могут выполняться как на одном, так и на различных вычислительных узлах). MPI-технология является

типичным примером императивной (основанной на полном управлении программистом последовательности вычислений и распределения данных) технологии создания программ для параллельного выполнения [1 - 3].

Формально MPI-подход основан на включении в программные модули вызовов функций специальной библиотеки (заголовочные и библиотечные файлы для языков C/C++ и Fortran) и загрузчика параллельно исполняемого кода в вычислительные узлы (ВУ). Подобные библиотеки имеются практически для всех платформ, поэтому написанные с использованием технологии MPI взаимодействия ветвей параллельного приложения программы независимы от машинной архитектуры (могут исполняться на однопроцессорных и многопроцессорных с общей или разделяемой памятью ЭВМ), от расположения ветвей (выполнение на одном или разных процессорах) и от API (*Application Program Interface*) конкретной операционной системы (ОС).

История MPI начинается в 1992 г. созданием стандарта эффективной и переносимой библиотеки передачи сообщений в *Oak Ridge National Laboratory (Rice University)*, окончательный вариант стандарта MPI 1.0 представлен в 1995 г., стандарт MPI-2 опубликован в 1997 г.

Из реализаций MPI известны MPICH (все UNIX-системы и Windows'NT, разработка *Argonne National Lab*., http://www.mcs.anl.gov), LAM (UNIX- подобные ОС, *Ohio Supercomputer Center*, http://www.osc.edu и *Notre-Damme University*, http://www.lsc.nd.edu), CHIMP/MPI (*Edinburgh Parallel Computing Centre*, http://www.epcc.ed.ac.uk), WMPI (все версии Windows, *разработка University of Coimbra*, *Portugal,* http://dsg.dei.uc.pt/wmpi; в 2002 г. права перешли к *Critical Software SA*, wmpi-sales@criticalsoftware.com) и др. (подробнее см. http://parallel.ru/tech/tech_dev/mpi.html).

Стандарт MPI постоянно развивается, так многие реализации включают такие современные технологии как RDMA – удаленный прямой доступ к памяти (http://www.open-mpi.org/papers/euro-pvmmpi-2006-hpc-protocols/euro-pvmmpi-2006-hpc-protocols.pdf). Эта технология позволяет с использованием высокоскоростных телекоммуникационных сред Myrinet и Infiniband получать скорость обмена между узлами сравнимую со скоростями обмена по внутренней шине компьютера.

Основными отличиями стандарта MPI-2 являются: динамическое порождение процессов, параллельный ввод/вывод, интерфейс для C++, расширенные коллективные операции, возможность одностороннего взаимодействия процессов (см. [1] и http://www.mpi-forum.org). В настоящее время MPI-2 практически не используется; существующие реализации поддерживают только MPI 1.1.

MPI является (довольно) низкоуровневым инструментом программиста; с помощью MPI созданы рассчитанные на численные методы специализированные библиотеки (например, включающая решение систем линейных уравнений, обращение матриц, ортогональные преобразования, поиск собственных значений и т.п. библиотеки ScaLAPACK, http://www.netlib.org/scalapack и AZTEC, http://www.cs.sandia.gov/CRF/aztec1.html для плотнозаполненных и разреженных матриц соответственно). MPI не обеспечивает механизмов задания начального размещения процессов по вычислительным узлам (процессорам); это должен явно задать программист или воспользоваться неким сторонним механизмом.

Разработчики MPI (справедливо) подвергаются жесткой критике за излишнюю громоздкость и сложность для прикладного программиста. Также интерфейс достаточно сложен для реализации.

Строго говоря, технология MPI подразумевает подход MPMD (*Multiple Program – Multiple Data*: *множество программ – множество данных*), при этом одновременно (и относительно независимо друг от друга) на различных ВУ выполняются несколько (базирующихся на различных исходных текстах) программных ветвей, в определенные промежутки времени обменивающиеся данными. Однако подобные программы слишком громоздки при написании (для каждого ВУ требуется отдельный исходной текст), поэтому на практике применяется SPMD-подход (*Single Program - Multiple Data: одна программа – множество данных*), предполагающий исполнение на различных ВУ логически выделенных условными операторами участков идентичного кода. Все ветви программы запускаются загрузчиком одновременно как процессы UNIX; количество ветвей фиксировано – согласно стандарту MPI 1.1 в ходе работы порождение новых ветвей невозможно.

Параллельно выполняющиеся программы обычно не используют привычного пользователям ОС Windows оконного интерфейса (вследствие как трудности связывания окон с конкретным экземпляром приложения, так и несущественности использования подобного интерфейса при решении серьезных счетных задач). Отладка параллельного приложения также существенно отличается от таковой обычного последовательного [1,3].

Полномасштабная отладка MPI-программ достаточно сложна вследствие одновременного исполнения нескольких программных ветвей (при этом традиционная отладки методом включения в исходный текст операторов печати затруднена вследствие смешивания в файле выдачи информации от различных ветвей MPI-программы); одним из наиболее мощных отладчиков считается TotalView (http://www.dolphinics.com). Также достаточно удобной для разработки и отладки считается среда, разработанная фирмой SUN. В функции MPI отладчиков входит трассировка программы и профилирование – выдача (часто в графическом виде) информации об обменах в параллельной программе.

В дальнейшем будем считать, что параллельное приложение состоит из нескольких ветвей (или процессов, или задач), выполняющихся одновременно на ВУ; при этом процессы обмениваются друг с другом данными в виде сообщений. Каждое сообщение имеет идентификатор, который позволяет программе и библиотеке связи отличать их друг от друга. В MPI существует понятие области связи; области связи имеют независимую друг от друга нумерацию процессов (коммуникатор как раз и определяет область связи).

В общем случае создание параллельной программы включает в себя (укрупненно) две основные стадии:

- Исходно-последовательный алгоритм подвергается декомпозиции (распараллеливанию), т.е. разбивается на независимо работающие ветви; для взаимодействия в ветви вводятся два типа дополнительных нематематических операции: прием (Send) и передача (Receive) данных.
- Распараллеленный алгоритм оформляется в виде программы, в которой операции приема и передачи записываются в терминах конкретной системы связи между ветвями (в нашем случае MPI).

Т.к. время обмена данными между ветвями намного (на порядки) больше времени доступа к собственной (локальной) памяти, распределение работы между процессами должно быть ‘крупнозернистым’ [1]. Типичный размер этого зерна (*гранулы*) составляет десятки-сотни тысяч машинных операций (что на порядки

превышает типичный размер оператора языков Fortran или C/C++, [3]). В зависимости от размеров гранул говорят о мелкозернистом и крупнозернистом параллелизме (*fine-grained parallelism* и *coarse-grained parallelism*).

На размер гранулы влияет и удобство программирования – в виде гранулы часто оформляют некий логически законченный фрагмент программы. Целесообразно стремиться к равномерной загрузке процессоров (как по количеству вычислительных операций, так и по загрузке оперативной памяти).

В MPI определены три категории функций - блокирующие, локальные, коллективные:

• Блокирующие функции останавливают (блокируют) выполнение процесса до тех пор, пока производимая ими операция не будет выполнена. В противовес этому неблокирующие функции возвращают управление вызвавшей их программе немедленно, а выполнение операции продолжается в фоновом режиме; за завершением операции надо проследить особо. Неблокирующие функции возвращают квитанции ('requests'), которые 'погашаются' при завершении; до погашения квитанции с переменными и массивами, которые были аргументами неблокирующей функции, ничего делать нельзя.

• Локальные функции не инициируют пересылок данных между ветвями.

Локальными являются большинство информационных функций (т.к. копии системных данных уже хранятся в каждой ветви). Функция передачи MPI_Send и функция синхронизации MPI_Barrier не являются локальными, поскольку производят пересылку. В то же время функция приема MPI_Recv (парная для MPI_Send) является локальной: она всего лишь пассивно ждет поступления данных, ничего не пытаясь сообщить другим ветвям.

• Коллективные функции должны быть вызваны всеми ветвями-абонентами коммуникатора (о понятии коммуникатора см. ниже), передаваемого им как аргумент. Несоблюдение этого правила приводит к ошибкам на стадии выполнения программы (обычно к 'повисанию' программы).

В MPI продумано объединение ветвей в коллективы, это позволяет настраивать логическую топологию параллельных процессов согласно требованиям прикладной задачи. Достигается это введением в функции MPI параметра типа 'коммуникатор', который можно рассматривать как дескриптор (номер) коллектива (он ограничивает область действия данной функции соответствующим коллективом). Коммуникатор коллектива, который включает в себя все ветви приложения, создается автоматически (при выполнении функции MPI_INIT) и называется MPI_COMM_WORLD; в дальнейшем имеется возможность (функция MPI_Comm_split) создавать новые коммуникаторы. Имеют место включающий только самого себя как процесс коммуникатор MPI_COMM_SELF и (заведомо неверный) коммуникатор MPI_COMM_NULL; в случае возможности (динамического) порождения новых процессов ситуация сложнее (подробнее см. [1 - 3]).

Наиболее простыми функциями коммуникаций являются функции 'точка-точка', в таких взаимодействиях участвуют два процесса, причем один процесс является отправителем сообщения, а другой - получателем.

Процесс-отправитель вызывает одну из процедур передачи данных и явно указывает номер в коммуникаторе процесса-получателя, а процесс получатель должен вызвать одну из процедур приема с указанием того же коммуникатора (в

некоторых случаях необязательно знать точный номер процесса-отправителя в заданном коммуникаторе).

Все процедуры этой группы делятся на два класса: *процедуры с блокировкой* (с синхронизацией) и *процедуры без блокировки* (асинхронные). Процедуры обмена с блокировкой приостанавливают работу процесса до выполнения определенного условия, а возврат из асинхронных процедур происходит немедленно после инициализации соответствующей коммуникационной операции. Неаккуратное использование процедур с блокировкой может привести к возникновению *тупиковой ситуации* (см. ниже), поэтому при этом требуется дополнительная осторожность; при использовании асинхронных операций тупиковых ситуаций не бывает, однако эти операции требуют более аккуратной работы с памятью.

При использовании процедур ‘точка-точка’ программист имеет возможность выбрать способ взаимодействия процессов и способ взаимодействия коммуникационного модуля MPI с вызывающим процессом.

По способу взаимодействия процессов - в случае неодновременного вызова двумя процессами парных функций приема и передачи могут быть произведены следующим образом:

• Буферизация данных осуществляется на передающей стороне (при этом функция передачи захватывает временный буфер, копирует в него сообщение и возвращает управление вызвавшему процессу; содержимое буфера передается принимающему процессу в фоновом режиме).

• Режим ожидания на стороне принимающего процесса (при этом возможно завершение с выдачей кода ошибки на передающей стороне).

• Режим ожидания на передающей стороне (завершение с кодом ошибки на стороне приема).

• Автоматический выбор одного из трех вышеприведенных вариантов (именно так действуют простейшие блокирующие функции MPI_Recv и MPI_Send).

Функций коллективных коммуникаций (при которых получателей и/или отправителей несколько) имеется несколько типов (подробнее о применении описанных функций см. в [1,2] и http://parallel.ru).

• **Broadcast**. Один – всем.

Широковещательная передача сообщений - один блок данных одного из процессов передается всем процессам группы MPI_Bcast(buffer, count, datatype, source, comm), где buffer – начальный адрес буфера данных, count – число элементов в буфере, datatype – тип элементов, source - номер передающего процесса, comm – коммуникатор

• **Scater**. Один-каждому.

Рассылка блока данных от одного процесса всем процессам группы, выполняется функцией MPI_Scatter (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, source, comm)

• **Gather**. Каждый-одному.

Сбор данных от всех процессов в группе в один из процессов группы MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, dest, comm). Каждый процесс группы (включая корневой) посылает содержимого своего передающего буфера корневому процессу.

• **Alltoall:** К*аждый-каждому.* Раздача/сборка БЛД от всех процессов во все процессы MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm).

• **Барьерная синхронизация группы процессов.** MPI_Barrier.

При вызове этой процедуры управление в вызывающую программу не возвращается до тех пор, пока *все процессы группы* не произведут вызовов MPI_Barrier(comm), где comm – коммуникатор.

• **Глобальные (собирающие данные со всех процессов) операции редукции**

Редукция позволяет выполнить некоторую операцию (сложение, вычисление максимума, минимума и т.д.) над данными всех процессов, результат передается одному или всем процессам (MPI_Reduce, MPI_Allreduce, MPI_Reduce_Scatter, MPI_Scan).

Восемь основных функций MPI.

MPI представляет достаточно много возможностей для прикладного программиста, однако для того чтобы начать программировать достаточно всего 8 функций (иногда это количество сокращают до 6). Перечислим эти восемь функций.

`MPI_Init` - инициализация MPI-библиотеки
`MPI_Finalize` - завершение программы MPI

`MPI_Comm_size` - определение числа процессов
`MPI_Comm_rank` - определение процессом собственного номера

`MPI_Send` - посылка сообщения
`MPI_Recv` - получение сообщения

`MPI_Bcast` - посылка сообщения группе процессов
`MPI_Reduce` – выполнение редукции

Среди функций основными являются функции MPI_Send/MPI_Recv для обмена сообщениями типа ‘точка-точка’. Подробнее эти функции рассмариваются далее при описании практических заданий.

Все функции библиотеки MPI описаны в заголовочном файле mpi.h, поэтому в раздел include MPI программы надо добавлять строку:

```
include <mpi.h> //подключение библиотеку MPI
```

Компиляция и запуск программы на кластере существенно отличается от компиляции и запуска обычной программы на персональном компьютере. В следующем разделе процесс работы с кластером рассматривается подробно.

2.2 Базовые приемы работы с вычислительным кластером

Стандарт MPI – это стандарт, допускающий различные прикладные реализации, существует, например, проприетарная реализация Intel MPI, оптимизированная под архитектуру Intel. Open source реализация MPI разрабатывается в университете Чикаго и носит название mpich. В настоящей методичке рассматривается mpich версии 2 <ссылка>.

Несмотря на то, что mpich 2 может работать под управлением различных ОС, и, в частности, windows, стандартом кластерной ОС является Linux. В нашем конкретном случае мы рассмотрим rpm-based систему CentOS.

Удаленный доступ к головной машине кластера осуществляется посредством SSH. Бесплатный SSH клиент для windows – putty (http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html). Для соединения с кластером нужно в строке Host Name указать имя хоста (или IP адрес) кластера. В нашем случае – cluster410.ssau.ru, далее свой логин и пароль. Для корректного отображения unicode кодировки в сеансе SSH необходимо в настройках putty в разделе Window->Translation указать кодировку UTF-8, вместо принятой по-умолчанию кодировки KOI8-R.

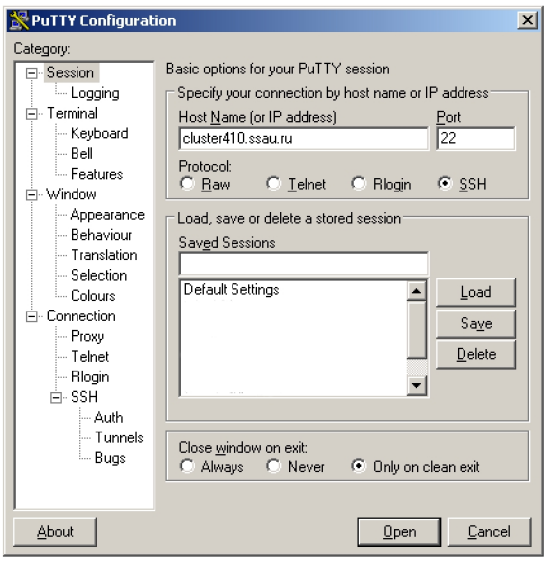

Рисунок 1. Окно ssh клиента - putty.

SSH не только обеспечивает удаленный доступ к системной консоли Linux машины, реализуя так называемый shell-доступ, но позволяет выполнять копирование файлов по шифрованному каналу. Для файлового обмена можно использовать консольный аналог ftp – sftp, а можно воспользоваться WinSCP, программой, которая позволяет работать с файлами удаленной Linux системой как с сетевым диском. Программа WinSCP бесплатная и может быть получена с сайта http://winscp.net/eng/docs/lang:ru.

Таким образом, разработка программы под mpi может быть выполнена в привычном для пользователя программном окружении, а потом скопирована на кластер посредством SSH. Однако с текстом программы можно работать и непосредственно в сессии удаленного доступа. В Linux существует множество текстовых редакторов, которые позволяют это сделать, например, nano или vim. На наш взгляд, наиболее удобным является редактор mcedit, редактор, встроенный в

файловый менеджер Midnight Commander или mc (<ссылка>). Работа с mc очень похожа на работу с Far Manager в Windows. Оба файловых менеджера имеют одного предка – Norton Commander for DOS. Так что знакомым с NC освоится в mc не составит труда. На рисунке 2 показан пример окна Midnight Commander открытого в ssh-сеансе.

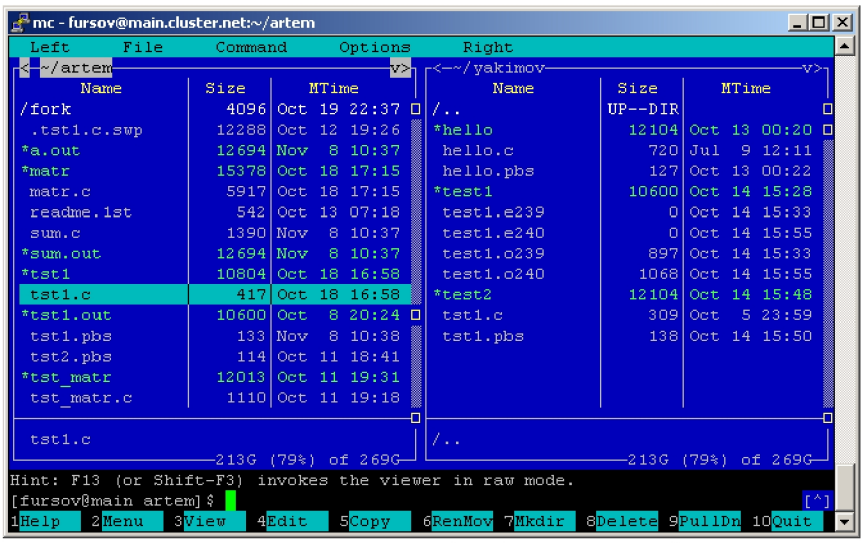

Рисунок 2. Окно Midnight Commander.

Перечислим некоторые, наиболее часто используемые операции. Переключение между отображением окна mc и командной строкой производится одновременным нажатием клавиш **Ctrl+O**. Для создания нового файла текста программы нажмите **Shift+F4**. В открывшемся окне редактирования можно набирать текст программы, а затем при сохранении файла будет выдан запрос на ввод имени нового файла. При редактировании можно вставлять текст из буфера обмена ОС MS Windows.

Оболочка mc позволяет провести все необходимые процедуры по написанию программы, запуску ее на кластере и анализ результатов исполнения. Прежде чем перейти к процессу написания и запуска программы на кластере, рассмотрим, каким образом выполняется запуск нашей программы на узлах кластера.

Система управления заданиями для высокопроизводительных вычислительных систем

Высокопроизводительные вычислительные кластеры обычно используются в многопользовательском режиме, в условиях запуска одновременно нескольких задач. Выполнение задач не всегда эффективно на всем кластере, чаще выгоднее выделять подмножество его ресурсов для задачи. Ситуация неконтролируемого разделения времени, когда несколько параллельных задач конкурируют за ресурсы – катастрофична для общей производительности; ручное управление заданиями (т.е. определение свободных и необходимых ресурсов, запуск и останов процессов задач, раздача входных данных и сбор результатов) трудоемко и неэффективно.

Поэтому, как правило, кластеры имеют систему управления заданиями. Она объединяет узлы, представляя пользователю кластер как некое единство, единый вычислитель у которого можно запрашивать ресурсы – процессоры, память, и т.д. на какое-то время. Затем сама заботится о выделении ресурсов так чтобы

исключить разделение времени, о справедливом планировании прохождения задач, автоматически прекращает выполнение зависших задач и т.д.

Существует несколько различных систем управления заданиями, старые проекты, такие как OpenPBS, и новые развивающиеся, такие как SLURM. Среди серьезных кластерных проектов, обычно используется планировщик Torgue (http://www.clusterresources.com/products/torque-resource-manager.php).

PBS – система управления заданиями для высокопроизводительных вычислительных кластеров. Она развилась эволюционно, из непараллельной NQS. Сначала PBS разрабатывалась в НАСА, потом компанией Veridian которая поддерживала две версии -- бесплатную OpenPBS и коммерческую PBS Pro. После того как Veridian была куплена компанией Altair, назначившей за PBS Pro большую стоимость для всех, в том числе и академических, лицензий, и остановившей разработку OpenPBS, последняя была поднята сообществом и получила название Torque. Она использует очень развитый планировщик Maui; изменен протокол опроса узлов, так что возможен запуск большого количества задач. Система более удобна для пользователя, так, сообщения об ошибках получили читаемый характер (больше нет ошибок: ‘error 15038 on command 42’). Система Torque установлена на кластерах СГАУ. Формирование заданий для этой системы будет рассмотрено ниже.

Этапы разработки программы на кластере

Работу с заданиями для кластера рассмотрим в процессе создания и запуска программы для кластера. Этапы этого процесса подробно описаны далее.

0. Создание файла с исходным текстом программы.

Программа может быть набрана непосредственно в сеансе работы на кластере в редакторе Midnight Commander. Для больших проектов удобнее воспользоваться привычным редактором, а уже готовый текст программы загрузить на кластер при помощи WinSCP.
Положим, что наша программа сохранена в файле:

```
Hello_mpi.c
```

1. Компиляция программы.

```
mpicc -o Hello_mpi Hello_mpi.c
```

Утилита mpicc это скрипт, выполняющий вызов стандартного компилятора gcc с передачей путей к заголовочным файлам и библиотекам пакета mpi. Приведенный пример создает исполняемый файл с именем Hello_mpi.

2. Запуск программы на кластере.

Запуск программы выполняет система управления заданиями. Наша задача сформировать задание для этой системы и поместить его в очередь. Поместить задание в очередь можно при помощи утилиты qsub. Эта утилита принимает

аргументы из командной строки, однако рекомендуется воспользоваться файлом описания задания:

```
qsub Hello_mpi.pbs
```

Приведенный пример регистрирует в очереди задание, описанное в файле Hello_mpi.pbs.

3. Проверка состояния очереди.
Проверить состояния задачи в очереди можно утилитой qstat.

```
[fursov@main mpi]$ qstat

Job id     Name         User             Time Use S Queue
---------- ------------ ---------------- --------- - -----
236.main   Hello_mpi    fursov           0         R  dque
```

В колонках таблицы выводимой утилитой присутствуют следующие поля. Job id – номер задачи, Name – название задачи, как оно задано в файле Hello_mpi.pbs, Time – используемое процессорное время, S – состояние задачи. В поле S может быть: R означает, что задача выполняется, E – задача в состоянии ошибки, C – исполнение задачи завершено.

4. Анализ результатов выполнения задачи.

После завершения выполнения задачи планировщиком в каталоге будут сохранены два файла:

```
Идентификатор_задачи.оНомер_задачи
Идентификатор_задачи.еНомер_задачи
```

в этих файлах сохранен поток стандартного вывода (STDOUT) и поток ошибок (STDERR) соответственно.

Работа с файлом .pbs

Рассмотрим пример .pbs файла.

```
#PBS -N Hello_mpi
#PBS -l nodes=2:ppn=4
#PBS -l walltime=00:01:00
cd $PBS_O_WORKDIR

mpiexec-pbs -np 8 /home/fursov/mpi/Hello_mpi
```

Определение идентификатора задачи:

```
#PBS -N Hello_mpi
```

Указание количества узлов и потоков для выполнения задачи, в данном случае, использовано 2 узла (nodes) и по 4 потока на каждом узле (ppn). Итого, наша задача будет выполняться в 8 потоках:

```
#PBS -l nodes=2:ppn=4
```

Определение максимального времени для выполнения задачи (walltime) в формате ЧЧ:ММ:СС, по истечению указанного интервала задача будет принудительно завершена планировщиком.

```
#PBS -l walltime=00:01:00
```

Механизм принудительное завершения или как его еще называют, сторожевой таймер, предохраняет систему от зависших задач. На этапе отладки, когда есть вероятность ошибок в программе приводящих к блокировке или бесконечному циклу не рекомендуется ставить walltime больше нескольких минут. Если значение walltime не указано то оно принимается равным значению по-умолчанию, как правило это 60 секунд.

Установка рабочего каталога. Кроме установки каталога, могут быть выполнены дополнительные команды инициализации, например, копирование необходимых для вычисления файлов.

```
cd $PBS_O_WORKDIR
```

Строка, указывающая путь к исполняемому файлу.

```
mpiexec-pbs -np 8 /home/fursov/mpi/Hello_mpi
```

Дополнительно можно указать опцию, которая позволит объединить файлы стандартного вывода и ошибок в один.

```
#PBS -j oe
```

Задача, описанная pbs файлом отправляется в очередь командой qsub, задача может быть удалена из очереди при помощи:

```
qdel Идентификатор_задачи.
```

2.3 Основные функции MPI

При использовании компактно расположенных кластеров исполняемый код приложения компилируется главной машиной, рассылается по вычислительным узлам (ВУ) и запускается на каждом средствами ОС.

Момент старта каждой ветви программы может отличаться от такового на других ВУ, также нельзя априори точно определить момент выполнения любого оператора внутри конкретной ветви, см. рисунок 3 (именно поэтому применяются различного типа приемы синхронизации при обмене сообщениями); Для практического осознания необходимости синхронизации процессов при параллельном программировании служит первая часть данной работы. Во второй части практически рассматривается формальный обмен данными между процессами (использование простейших функций передачи MPI_Send и приема данных MPI_Recv).

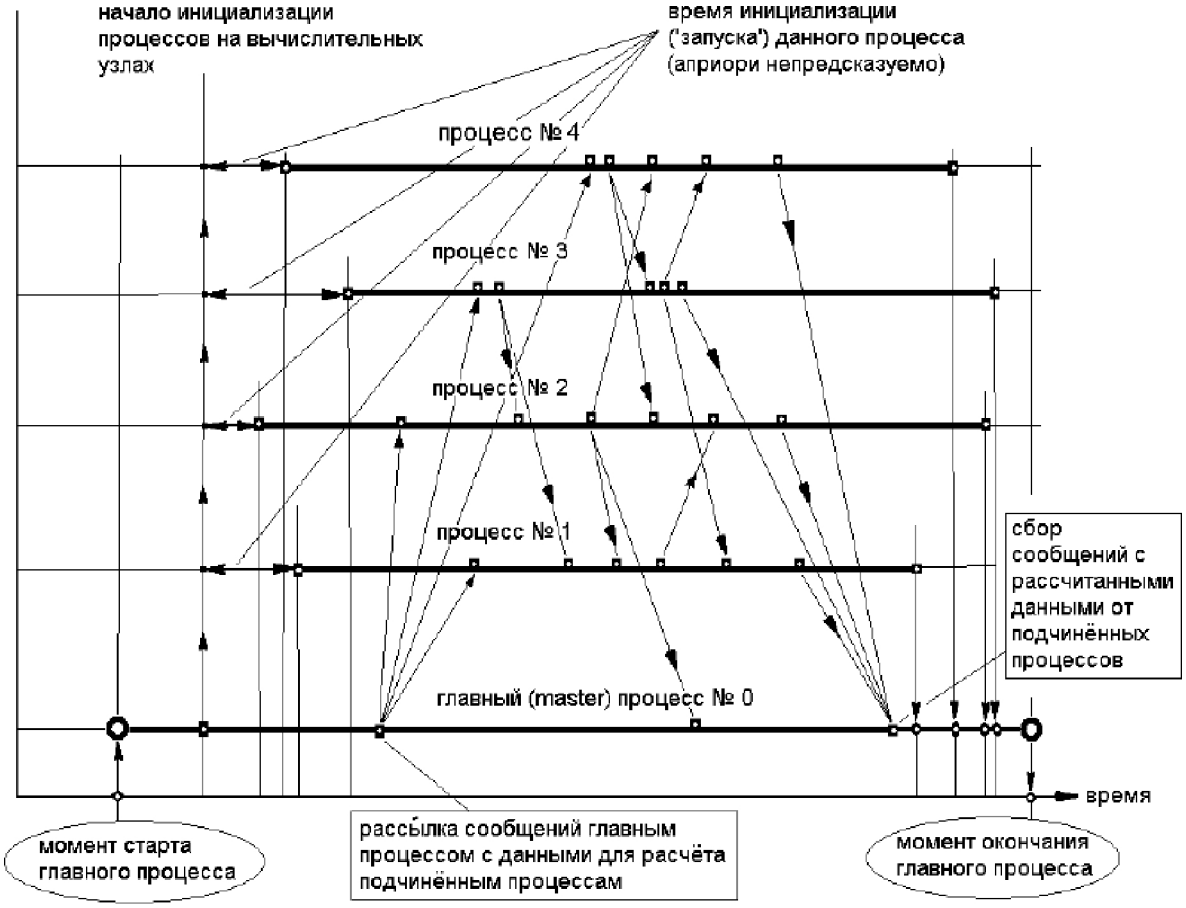

Рисунок. 3 Схема жизненного цикла MPI процесса.

Для этих (базовых) функций полезно привести прототипы (многие из приведенных формальных параметров используются и в других MPI-функциях):

```
int
MPI_Send(void* buf, int count, MPI_Datatype datatype, int
dest, int msgtag,
    MPI_Comm comm);
```

• buf - адрес начала буфера отправляемого сообщения
• count - число передаваемых в сообщении элементов (*не байт* – см. ниже)
• datatype - тип передаваемых элементов
• dest - номер процесса-получателя
• msgtag - идентификатор сообщения (0 ÷32767, выбирается пользователем)
• comm - идентификатор группы (MPI_COMM_WORLD для создаваемого по умолчанию коммуникатора)

Функция MPI_Send осуществляет *блокирующую* посылку сообщения с идентификатором msgtag процессу с номером dest; причем сообщение состоит из count элементов типа datatype (все элементы сообщения расположены подряд в буфере buf, значение count может быть и нулем). Тип передаваемых элементов datatype должен указываться с помощью предопределенных констант типа (см. ниже), разрешается передавать сообщение самому себе. При пересылке сообщений можно использовать специальное значение MPI_PROC_NULL для несуществующего процесса; операции с таким процессом немедленно успешно завершаются (код завершения MPI_SUCCESS).

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. Это достигается копированием в промежуточный буфер или непосредственной передачей процессу dest (определяется MPI). Важно отметить, что возврат из MPI_Send не означает ни того, что сообщение уже передано процессу dest, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший MPI_Send.

Имеются следующие модификации процедуры передачи данных с блокировкой MPI_Send:

• MPI_Bsend - передача сообщения *с буферизацией.* Если прием посылаемого сообщения еще не был инициализирован процессом получателем, то сообщение будет записано в специальный буфер, и произойдет немедленный возврат из процедуры. Выполнение этой процедуры никоим образом не зависит от соответствующего вызова процедуры приема сообщения. Тем не менее процедура может вернуть код ошибки, если места под буфер недостаточно (о выделении массива для буферизации должен озаботиться пользователь).

• MPI_Ssend - передача сообщения *с синхронизацией.* Выход из этой процедуры произойдет только тогда, когда прием посылаемого сообщения будет инициализирован процессом-получателем. Таким образом, завершение передачи с синхронизацией говорит не только о возможности повторного использования буфера посылки, но и о гарантированном достижении процессом-получателем точки приема сообщения в программе. Использование передачи сообщений с синхронизациейможет замедлить выполнение программы, но позволяет избежать наличия в системе большого количества не принятых буферизованных сообщений.

• MPI_Rsend - передача сообщения *по готовности.* Этой процедурой можно пользоваться только в том случае, если процесс-получатель уже инициировал прием сообщения. В противном случае вызов процедуры является ошибочным и результат ее выполнения не определен. Гарантировать инициализацию приема сообщения перед вызовом процедуры MPI_Rsend можно с помощью операций,

осуществляющих явную или неявную синхронизацию процессов (например, MPI_Barrier или MPI_Ssend). Во многих реализациях процедура MPI_Rsend сокращает протокол взаимодействия между отправителем и получателем, уменьшая накладные расходы на организацию передачи данных.

В MPI не используются привычные для C типы данных (int, char и др.), вместо них удобно применять определенные для данной платформы константы MPI_INT, MPI_CHAR и т.д. (см. табл.1).

Таблица 1.— Предопределенные в MPI константы типов данных.

Константы MPI	*Соответствующий тип в C*
MPI_INT	signed int
MPI_UNSIGNED	Unsigned int
MPI_SHORT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_SHORT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	long double
MPI_UNSIGNED_CHAR	unsigned char
MPI_CHAR	signed char

Пользователь может зарегистрировать в MPI свои собственные типы данных (например, структуры), после чего MPI сможет обрабатывать их наравне с базовыми.

Практически в каждой MPI-функции одним из параметров является коммуникатор (идентификатор группы процессов); в момент инициализации библиотеки MPI создается коммуникатор MPI_COMM_WORLD и в его пределах процессы нумеруются линейно от 0 до size. Однако с помощью коммуникатора для процессов можно определить и другие системы нумерации (*пользовательские топологии*). Дополнительных систем в MPI имеются две: декартова N-мерная решетка (с цикличностью и без) и ориентированный граф. Существуют функции для создания и тестирования нумераций (MPI_Cart_xxx, MPI_Graph_xxx, MPI_Topo_test) и для преобразования номеров из одной системы в другую. Этот механизм *чисто логический и не связан с аппаратной топологией*; при его применении автоматизируется пересчет адресов ветвей (например, при вычислении матриц иногда выгодно использовать картезианскую систему координат, где координаты вычислительной ветви совпадают с координатами вычисляемой ею подматрицы).

```
int
MPI_Recv(void* buf, int count, MPI_Datatype datatype, int
source, int msgtag,
MPI_Comm comm, MPI_Status *status);
```

- buf - адрес начала буфера приема сообщения (возвращаемое значение)
- count - максимальное число элементов в принимаемом сообщении

• datatype - тип элементов принимаемого сообщения
• source - номер процесса-отправителя
• msgtag - идентификатор принимаемого сообщения
• comm - идентификатор группы
• status - параметры принятого сообщения (возвращаемое значение)

Функция MPI_Recv осуществляет прием сообщения с идентификатором msgtag от процесса source с блокировкой (блокировка гарантирует, что после возврата из подпрограммы все элементы сообщения приняты и расположены в буфере buf). Число элементов в принимаемом сообщении не должно превосходить значения count (если число принятых элементов меньше count, то гарантируется, что в буфере buf изменятся только элементы, соответствующие элементам принятого сообщения). Если процесс посылает два сообщения другому процессу и оба эти сообщения соответствуют одному и тому же вызову MPI_Recv, первым будет принято то сообщение, которое было отправлено раньше.

Т.о. с помощью пары функций MPI_Send/MPI_Recv осуществляется надежная (но не слишком эффективная) передача данных между процессами. Однако в некоторых случаях (например, когда принимающая сторона ожидает приема сообщений, но априори не знает длины и типа их) удобно использовать блокирующую функцию MPI_Probe, позволяющую определить характеристики сообщения *до того*, как оно будет помещено в приемный пользовательский буфер (гарантируется, что следующая вызванная функция MPI_Recv прочитает именно протестированное MPI_Probe сообщение):

```
int
MPI_Probe( int source, int msgtag, MPI_Comm comm,
MPI_Status *status);
```

• source - номер процесса-отправителя (или MPI_ANY_SOURCE)
• msgtag - идентификатор ожидаемого сообщения (или MPI_ANY_TAG)
• comm - идентификатор группы
• status - параметры обнаруженного сообщения (возвращаемое значение)

В структуре status (тип MPI_Status) содержится информация о сообщении – его идентификатор (поле MPI_TAG), идентификатор процесса-отправителя (поле MPI_SOURCE), фактическую длину сообщения можно узнать посредством вызовов

```
MPI_Status status;
int count;
MPI_Recv ( ... , MPI_INT, ... , &status );
MPI_Get_count (&status, MPI_INT, &count); /* тип элементов
тот же, что у MPI_Recv теперь в count содержится количество
принятых элементов типа MPI_INT */
```

Удобно сортировать сообщения с помощью механизма ‘джокеров’; в этом случае вместо явного указания номера задачи-отправителя используется ‘джокер’

MPI_ANY_SOURCE (‘принимай от кого угодно’) и MPI_ANY_TAG вместо идентификатора получаемого сообщения (‘принимай что угодно’).

Достоинство ‘джокеров’ в том, что приходящие сообщения извлекаются по мере поступления, а не по мере вызова MPI_Recv с нужными идентификаторами задач и/или сообщений (что экономит память и увеличивает скорость работы). Пользоваться ‘джокерами’ рекомендуется с осторожностью, т.к. возможна ошибка в приеме сообщения ‘не того’ или ‘не от того’.

При обмене данными в некоторых случаях возможны *вызванные взаимной блокировкой* т.н. тупиковые ситуации (используются также термины ‘deadlock’, ‘клинч’); в этом случае функции посылки и приема данных мешают друг другу и обмен не может состояться. Ниже рассмотрена deadlock-ситуация при использовании для пересылок разделяемой памяти.

Вариант 1.

Ветвь 1 :
Recv (из ветви 2)
Send (в ветвь 2)
Ветвь 2 :
Recv (из ветви 1)
Send (в ветвь 1)

Вариант 1 приведет к deadlock’у при любом используемом инструментарии, т.к. функция приема не вернет управления до тех пор, пока не получит данные; из-за этого функция передачи не может приступить к отправке данных, поэтому функция приема не вернет управление... и т.д. до бесконечности.

Вариант 2.

Ветвь 1 :
Send (в ветвь 2)
Recv (из ветви 2)
Ветвь 2 :
Send (в ветвь 1)
Recv (из ветви 1)

Казалось бы, что (если функция передачи возвращает управление толькопосле того, как данные попали в пользовательский буфер на стороне приема) и здесь deadlock неизбежен. Однако при использовании MPI зависания во втором варианте не произойдет: функция MPI_Send, если на приемной стороне нет готовности (не вызвана MPI_Recv), не станет дожидаться ее вызова, а скопирует данные во временный буфер и *немедленно вернет управление основной программе*. Вызванный далее MPI_Recv данные получит *не напрямую из пользовательского буфера, а из промежуточного системного* буфера (т.о. используемый в MPI способ буферизации повышает надежность – делает программу более устойчивой к возможным ошибкам программиста). Т.о. наряду с полезными качествами (см. выше) свойство блокировки может служить причиной возникновения (трудно

локализуемых и избегаемых для непрофессионалов) тупиковых ситуаций при обмене сообщениями между процессами.

Очень часто функции MPI_Send/MPI_Recv используются совместно и именно в таком порядке, поэтому в MPI специально введены две функции, осуществляющие одновременно посылку одних данных и прием других. Первая из них - MPI_Sendrecv (у нее первые 5 формальных параметров такие же, как у MPI_Send, остальные 7 параметров аналогичны MPI_Recv). Следует учесть, что:

• как при приеме, так и при передаче используется один и тот же коммуникатор,

• порядок приема и передачи данных MPI_Sendrecv выбирает автоматически; при этом гарантировано отсутствие deadlock'а,

• MPI_Sendrecv совместима с MPI_Send и MPI_Recv

Функция MPI_Sendrecv_replace помимо общего коммуникатора использует еще и общий для приема-передачи буфер. Применять MPI_Sendrecv_replace удобно с учетом:

• принимаемые данные должны быть заведомо *не длиннее* отправляемых

• принимаемые и отправляе мые данные должны иметь одинаковый тип

• принимаемые данные записываются на место отправляемых

• MPI_Sendrecv_replace так же гарантированно не вызывает deadlock'а

В MPI предусмотрен набор процедур для осуществления *асинхронной передачи данных.* В отличие от блокирующих процедур, возврат из процедур данной группы происходит сразу после вызова без какой-либо остановки работы процессов; на фоне дальнейшего выполнения программы одновременно происходит и обработка асинхронно запущенной операции.

Обычно эта возможность исключительно полезна для создания эффективных программ - во многих случаях совершенно не обязательно дожидаться окончания посылки сообщения для выполнения последующих вычислений. Асинхронным аналогом процедуры MPI_Send является MPI_Isend (для MPI_Recv, естественно, MPI_Irecv). Аналогично трем модификациям процедуры MPI_Send предусмотрены три дополнительных варианта процедуры

MPI_Isend:

• MPI_Ibsend - неблокирующая передача сообщения с буферизацией

• MPI_Issend - неблокирующая передача сообщения с синхронизацией

• MPI_Irsend - неблокирующая передача сообщения по готовности

Рассмотрим также операцию редукции данных всех процессов.

```
int
MPI_Reduce(void*   sendbuf,   void*   recvbuf,   int   count,
    MPI_Datatype  datatype,  int  op,  int  dest,  int  msgtag,
    MPI_Comm comm);
```

• **sendbuf** - адрес начала буфера отправляемого сообщения

• **recvbuf** - адрес начала буфера принимаемого сообщения

• **count** - число передаваемых в сообщении элементов (*не байт* – см. ниже)

• **datatype** - тип передаваемых элементов

• **op** – дескриптор операции, которую требуется применить к данным

• **dest** - номер процесса-получателя

• **msgtag -** идентификатор сообщения (0 ÷32767, выбирается пользователем)
• **comm -** идентификатор группы (MPI_COMM_WORLD для создаваемого по умолчанию коммуникатора).

Функция MPI_Reduce объединяет элементы входного буфера каждого процесса в группе, используя операцию op , и возвращает объединенное значение в выходной буфер процесса с номером root. Буфер ввода определен аргументами sendbuf, count и datatype; буфер вывода определен параметрами recvbuf, count и datatype; оба буфера имеют одинаковое число элементов одинакового типа. Функция вызывается всеми членами группы с одинаковыми аргументами count, datatype, op, root и comm. Таким образом, все процессы имеют входные и выходные буферы одинаковой длины и с элементами одного типа. Каждый процесс может содержать либо один элемент, либо последовательность элементов, в последнем случае операция выполняется над всеми элементами в этой последовательности. Например, если выполняется операция MPI_MAX , и посылающий буфер содержит два элемента - числа с плавающей точкой (count = 2, datatype = MPI_FLOAT), то recvbuf(1) = sendbuf(1) и recvbuf(2) = sendbuf(2).

3. Программирование для массивно-многопоточных систем с использованием технологии NVIDIA CUDA

3.1 История возникновения CUDA.

Графические процессоры (GPU), также как многоядерные процессоры и кластерные системы, являются параллельными архитектурами. Каждый GPU обладает своей памятью (DRAM), объем которой уже достигает 1,5 Гбайт для некоторых моделей. Также GPU содержит ряд потоковых мультипроцессоров (SM, Streaming Multiprocessor), каждый из которых способен одновременно выполнять 768 (1024 - для более поздних моделей) нитей. При этом количество потоковых мультипроцессоров зависит от модели GPU. Так, GTX 280 содержит 30 потоковых мультипроцессоров. Каждый мультипроцессор работает независимо от остальных.

Термин GPU (Graphics Processing Unit) был впервые использован корпорацией Nvidia для обозначения того, что графический ускоритель, первоначально используемый только для ускорения трехмерной графики, стал мощным программируемым устройством (процессором), пригодным для решения широкого класса задач, никак не связанных с графикой. Современные GPU представляют из себя массивно-параллельные вычислительные устройства с очень высоким быстродействием (свыше одного терафлопа) и большим объемом собственной памяти (DRAM).

Однако начиналось все более чем скромно - первые графические ускорители Voodoo компании 3DFx представляли из себя фактически просто растеризаторы (переводящие треугольники в массивы пикселов) с поддержкой буфера глубины, наложения текстур и альфа-блендинга. При этом вся обработка вершин проводилась центральным процессором, и ускоритель получал на вход уже отображенные на экран (то есть спроектированные) вершины. Однако именно эту очень простую задачу Voodoo умел делать достаточно быстро, легко обгоняя универсальный центральный процессор, что, собственно, и привело к широкому распространению графических ускорителей 3D-графики. Причина этого заключалась в значительной степени в том, что графический ускоритель мог одновременно обрабатывать сразу много отдельных пикселов, пусть и выполняя для них очень простые операции.

Вообще, традиционные задачи рендеринга очень хорошо подходят для параллельной обработки - все вершины можно обрабатывать независимо друг от друга, точно так же отдельные фрагменты, получающиеся при растеризации треугольников, тоже могут быть обработаны совершенно независимо друг от друга.

После своего появления ускорители трехмерной графики быстро эволюционировали, при этом, помимо увеличения быстродействия, также росла и их функциональность. Так, графические ускорители следующего поколения (например, Riva TNT) уже могли самостоятельно обрабатывать вершины, разгружая тем самым CPU, и одновременно накладывать несколько текстур.

Следующим шагом было увеличение гибкости при обработке отдельных фрагментов (пикселов) с целью реализации ряда эффектов, например попиксельного освещения. На том этапе развития сделать полноценно программируемый обработчик фрагментов было нереально, однако были созданы блоки, способные реализовывать довольно простые операции (например, вычисление скалярного произведения). При этом эти блоки можно быстро настраивать и соединять между собой их входы и выходы.

Следующим шагом было появление вершинных программ (GeForce 2) - можно было вместо фиксированных шагов по обработке вершин задать программу, написанную на специальном ассемблере. Данная программа выполнялась параллельно для каждой вершины, и вся работа шла над числами типа float (размером 32 бита). Следующим принципиальным шагом стало появление подобной функциональности уже на уровне отдельных фрагментов - возможности задания обработки отдельных фрагментов при помощи специальных программ. Подобная возможность появилась на ускорителях серии GeForce FX. Используемый для задания программ обработки отдельных фрагментов ассемблер был очень близок к ассемблеру для задания вершинных программ и также проводил все операции при помощи чисел типа float.

При этом как вершинные, так и фрагментные программы выполнялись параллельно (графический ускоритель содержал отдельно вершинные процессоры и отдельно - фрагментные процессоры). Поскольку количество обрабатываемых в секунду пикселов было очень велико, то получаемое в результате быстродействие в Flop'ах также было очень большим. Фактически, графические ускорители на тот момент стали представлять собой мощные SIMD-процессоры. Термин SIMD (Single Instruction Multiple Data) обозначает параллельный процессор, способный одновременно выполнять одну и ту же операцию над

многими данными. Фактически SIMD-процессор получает на вход поток однородных данных и параллельно обрабатывает их, порождая тем самым выходной поток.

Сам модуль, осуществляющий подобное преобразование входных потоков в выходные, принято называть **ядром (kernel)**. Отдельные ядра могут соединяться между собой, приводя к довольно сложным схемам обработки входных потоков. Добавление поддержки текстур со значениями в форматах с плавающей точкой (16- и 32-битовые типы float вместо используемых ранее 8-битовых беззнаковых целых чисел) позволило применять фрагментные процессоры для обработкибольших массивов данных. При этом сами такие массивы передавались GPU как текстуры, со значениями компонент типа float, и результат обработки также сохранялся в текстурах такого же типа.

Фактически мы получили мощный параллельный процессор, на вход которому можно передать большой массив данных и программу для их обработки и на выходе получить массив результатов. Появление высокоуровневых языков для написания программ для GPU, таких как Cg, GLSL и HLSL, заметно облегчило создание подобных программ обработки данных.

В результате возникло такое направление, как GPGPU *(General-Purpose computing on Graphics Processing Units)* - использование графических процессоров для решения неграфических задач. За счет использования высокой степени параллелизма довольно быстро удалось получить очень высокую производительность -ускорение по сравнению с центральным процессором часто достигало 10 раз. Оказалось, что многие ресурсоемкие вычислительные задачи достаточно хорошо ложатся на архитектуру GPU, позволяя заметно ускорить их численное решение. Так, многие игры используют довольно сложные модели для расчета волн в воде, решая при этом дифференциальные уравнения на GPU в реальном времени.

Всего за несколько лет за счет использования GPU удалось заметно ускорить решение ряда сложных вычислительных задач, достигая ускорения в 10 и более раз. Ряд процедур обработки изображения и видео с переносом на GPU стали работать в реальном времени (вместо нескольких секунд на один кадр). При этом разработчики использовали один из распространенных графических API (OpenGL и Direct3D) для доступа к графическому процессору. Используя такой API, подготавливались текстуры, содержащие необходимые входные данные, и через операцию рендеринга (обычно просто прямоугольника) на графическом процессоре запускалась программа для обработки этих данных. Результат

получался также в виде текстур, которые потом считывались в память центрального процессора.

Фактически программа писалась сразу на двух языках - на традиционном языке программирования, например C++, и на языке для написания шейдеров. Часть программы (написанная на традиционном языке программирования) отвечала за подготовку и передачу данных, а также за запуск на GPU программ, написанных на шейдерных языках.

Однако традиционный GPGPU обладает и рядом недостатков, затрудняющих его распространение. Все эти ограничения непосредственно связаны с тем, что использование возможностей GPU происходит через API, ориентированный на работу с графикой (OpenGL или Direct3D).

И в результате все ограничения, изначально присущие данным API (и вполне естественные с точки зрения графики), влияют на реализацию расчетных задач (где подобные ограничения явно избыточны).

Так, в графических API полностью отсутствует возможность какого-либо взаимодействия между параллельно обрабатываемыми пикселами, что в графике действительно не нужно, но для вычислительных задач оказывается довольно желательным. Еще одним ограничением, свойственным графическим API, является отсутствие поддержки операции типа scatter. Простейшим примером такой операции служит построение гистограмм по входным данным, когда очередной элемент входных данных приводит к изменению заранее неизвестного элемента (или элементов) гистограммы. Это связано с тем, что в графических API шейдер может осуществлять запись лишь в заранее определенное место, поскольку для фрагментного шейдера заранее определяется, какой фрагмент он будет обрабатывать, и он может записать только значение для данного фрагмента

Еще одним унаследованным недостатком является то, что разработка ведется сразу на двух языках программирования: один - традиционный, соответствующий коду, выполняемому на CPU, и другой - шейдерный, соответствующий коду, выполняемому на GPU. Все эти обстоятельства усложняют использование GPGPU и накладывают серьезные ограничения на используемые алгоритмы. Поэтому вполне естественно возникла потребность в средствах разработки GPGPU-приложений, свободных от этих ограничений и ориентированных на решение сложных вычислительных задач.

3.2 Модель программирования CUDA

3.2.1 Основные понятия CUDA

Предложенная компанией Nvidia технология CUDA (Compute Unified Device Architecture) заметно облегчает написание GPGPU-приложений. Она не использует графических API и свободна от ограничений, свойственных этим API.

Данная технология предназначена для разработки приложений для массивно-параллельных вычислительных устройств. На сегодняшний момент поддерживаемыми устройствами являются все GPU компании Nvidia, начиная с серии GeForce8, а также специализированные для решения расчетных задач GPU семейства Tesla.

Основными преимуществами технологии CUDA являются ее простота - все программы пишутся на «расширенном» языке C, наличие хорошей документации, набор готовых инструментов, включающих профайлер, набор готовых библиотек, кроссплатформенность (поддерживаются Microsoft Windows, Linux и Mac OS X).

CUDA является полностью бесплатной, SDK, документацию и примеры можно скачать с сайта developer.nvidia.com. На данный момент последней версией является CUDA 3.2.

CUDA строится на концепции, что GPU (называемый устройством, *device)* выступает в роли массивно-параллельного сопроцессора к CPU (называемому *host).* Программа на CUDA задействует как CPU, так и GPU. При этом обычный (последовательный, то есть непараллельный) код выполняется на CPU, а для массивно-параллельных вычислений соответствующий код выполняется на GPU как набор одновременно выполняющихся нитей (потоков, *threads).*

Таким образом, GPU рассматривается как специализированное вычислительное устройство, которое:

- является сопроцессором к CPU;
- обладает собственной памятью;
- обладает возможностью параллельного выполнения огромного количества отдельных нитей.

При этом очень важно понимать, что между нитями на CPU и нитями на GPU есть принципиальные различия:

- нити на GPU обладают крайне небольшой стоимостью создания, управления и уничтожения (контекст нити минимален, все регистры распределены заранее);

- для эффективной загрузки GPU необходимо использовать много тысяч отдельных нитей, в то время как для CPU обычно достаточно 10-20 нитей.

За счет того, что программы в CUDA пишутся фактически на обычном языке C (на самом деле для частей, выполняющихся на CPU, можно использовать язык C++), в который добавлено небольшое число новых конструкций (спецификаторы типа, встроенные переменные и типы, директива запуска ядра), написание программ с использованием технологии CUDA оказывается заметно проще, чем при использовании традиционного GPGPU (то есть использующего графические API для доступа GPU). Кроме того, в распоряжении программиста оказывается гораздо больше контроля и возможностей по работе с GPU.

В следующем листинге приводится простейший пример - фрагмент кода, использующий GPU для поэлементного сложения двух одномерных массивов.

```
    // Ядро, выполняется параллельно на большом числе нитей.
 global   void sumKernel( float * a, float * b, float * c )
    // Глобальный индекс нити. int idx = threadIdx.x +
//blockIdx.x ♦ blockDim.x;
    // Выполнить обработку соответствующих данной нити данных. c
[idx] = a (idx] + b [idx]; )
    void sum ( float * a, float * b, float * c, int n )
    (
    int numBytes = n * sizeof ( float );
    float * aDev = NULL;
    float * bDev = NULL;
    float * cDev = NULL;
    // Выделить память на GPU.
    cudaMalloc ( (void**)&aDev, numBytes );
    cudaMalloc ( (void**)&bDev, numBytes );
    cudaMalloc ( (void**)&cDev, numBytes );
    // Задать конфигурацию запуска n нитей. dim3 threads =
dim3(512, 1); dim3 blocks = dim3(n / threads.x, 1);
```

```
    // Скопировать входные ванные из памяти CPU в память GPU.
cudaMemcpy ( aDev, a, numBytes, cudaMemcpyHostToDevice );
cudaMemcpy ( bDev, b, numBytes, cudaMemcpyHostToDevice );
    // Вызвать ядро с заданной конфигурацией для обработки
данных. sumKernel<<<blocks, threads>>> (aDev, bDev, cDev);
    // Скопировать результаты в память CPU.
    cudaMemcpy(c, cDev, numBytes, cudaMemcpyDeviceToHost );
    // Освободить выделенную память GPU. cudaFree ( aDev );
cudaFree ( bDev );
    cudaFree ( cDev ) ;
```

В приведенном выше листинге первая функция (sumKernel) является ядром -она будет параллельно выполняться

для каждого набора элементов a [i], b [i] и c[i]. Спецификатор global используется для обозначения того, что это ядро, то есть функция, которая работает на GPU и которая может быть вызвана (точнее, запущена сразу на большом количестве нитей) только с CPU.

Вначале функция sumKernel при помощи встроенных переменных вычисляет соответствующий данной нити глобальный индекс, для которого необходимо произвести сложение соответствующих элементов и записать результат.

Выполняемая на CPU функция sum осуществляет выделение памяти на GPU (поскольку GPU может непосредственно работать только со своей памятью), копирует входные данные из памяти CPU в выделенную память GPU, осуществляет запуск ядра (функции sumKernel), после чего копирует результат обратно в память CPU и освобождает выделенную память GPU.

Хотя для массивов, расположенных в памяти GPU, мы и используем обычные указатели, так же как и для данных, расположенных в памяти CPU, важно помнить о том, что CPU не может напрямую обращаться к памяти GPU по таким указателям. Вся работа с памятью GPU ведется CPU при помощи специальных функций.

Рассмотренный выше пример очень хорошо иллюстрирует использование CUDA

- выделяем память на GPU;
- копируем данные из памяти CPU в выделенную память GPU;
- осуществляем запуск ядра (или последовательно запускаем несколько

ядер);

□ копируем результаты вычислений обратно в память CPU;

□ освобождаем выделенную память GPU.

Обратите внимание, что мы фактически для каждого допустимого индекса входных массивов запускаем отдельную нить для осуществления нужных вычислений. Все эти нити выполняются параллельно, и каждая нить может получить информацию о себе через встроенные переменные.

Важным моментом является то, что хотя подобный подход очень похож на работу с SIMD-моделью, есть и принципиальные отличия (компания Nvidia использует термин *SIMT- Single Instruction, Multiple Thread).* Нити разбиваются на группы по 32 нити, называемые *warp'*ами. Только нити в пределах одного *warp'*а выполняются физически одновременно. Нити из разных *warp'*ов могут находиться на разных стадиях выполнения программы. При этом управление *warp'*ами прозрачно осуществляет сам GPU.

Для решения задач CUDA использует очень большое количество параллельно выполняемых нитей, при этом обычно каждой нити соответствует один элемент вычисляемых данных. Все запущенные на выполнение нити организованы в следующую иерархию (рис. 3.1).

Верхний уровень иерархии - сетка *(grid)* - соответствует всем нитям, выполняющим данное ядро. Верхний уровень представляет из себя одномерный или Двухмерный массив блоков *(block).* Каждый блок - это одномерный, двухмерный

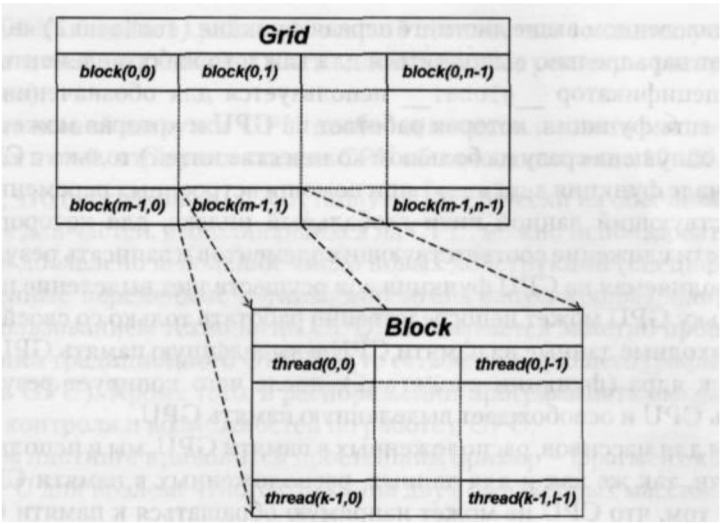

Рис. 3.1. Иерархия нитей в CUDA

или трехмерный массив нитей *(thread).* При этом все блоки, образующие сетку, имеют одинаковую размерность и размер.

Каждый блок в сетке имеет свой адрес, состоящий из одного или двух неотрицательных целых чисел (индекс блока в сетке). Аналогично каждая нить внутри блока также имеет свой адрес - одно, два или три неотрицательных целых числа, задающих индекс нити внутри блока.

Поскольку одно и то же ядро выполняется одновременно очень большим числом нитей, то для того, чтобы ядро могло однозначно определить номер нити (а значит, и элемент данных, который нужно обрабатывать), используются встроенные переменные threadIdx и blockIdx. Каждая из этих переменных является трехмерным целочисленным вектором. Обратите внимание, что они доступны только для функций, выполняемых на GPU, - для функций, выполняющихся на CPU, они не имеют смысла.

Также ядро может получить размеры сетки и блока через встроенные переменные gridDim и blockDim.

Подобное разделение всех нитей является еще одним общим приемом использования CUDA - исходная задача разбивается на набор отдельных подзадач, решаемых независимо друг от друга (рис. 2.2). Каждой такой подзадаче соответствует свой блок нитей.

При этом каждая подзадача совместно решается всеми нитями своего блока. Разбиение нитей на *warp*'ы происходит отдельно для каждого блока; таким образом, все нити одного *warp*'а всегда принадлежат одному блоку. При этом нити могут взаимодействовать между собой только в пределах блока. Нити разных блоков взаимодействовать между собой не могут.

Подобный подход является удачным компромиссом между необходимостью обеспечить взаимодействие нитей между собой и стоимостью обеспечения подоб ного взаимодействия - обеспечить возможность взаимодействия каждой нити с каждой было бы слишком сложно и дорого.

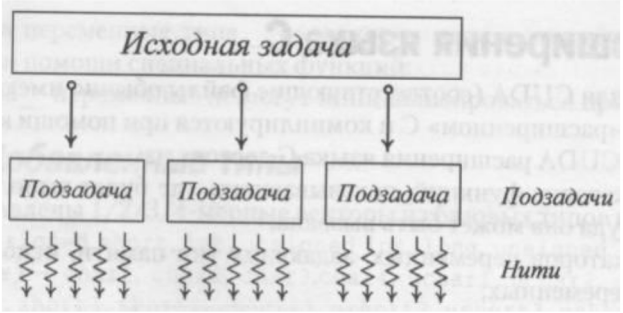

Рис. 3.2. Разбиение исходной задачи на набор независимо решаемых подзадач

Существуют всего два механизма, при помощи которых нити внутри блока могут взаимодействовать друг с другом:

□ разделяемая *(shared)* память;

□ барьерная синхронизация.

Каждый блок получает в свое распоряжение определенный объем быстрой разделяемой памяти, которую все нити блока могут совместно использовать. Поскольку нити блока не обязательно выполняются физически параллельно (то есть мы имеем дело не с «чистой» SIMD-архитектурой, а имеет место прозрачное управление нитями), то для того, чтобы не возникало проблем с одновременной работой с shared-памятью, необходим некоторый механизм синхронизации нитей блока.

CUDA предлагает довольно простой способ синхронизации - так называемая барьерная синхронизация. Для ее осуществления используется вызов встроенной функции *syncthreads ()*, которая блокирует вызывающие нити блока до тех пор, пока все нити блока не войдут в эту функцию. Таким образом, при помощи *_syncthreads ()* мы можем организовать «барьеры» внутри ядра, гарантирующие, что если хотя бы одна нить прошла такой барьер, то не осталось ни одной за барьером (не

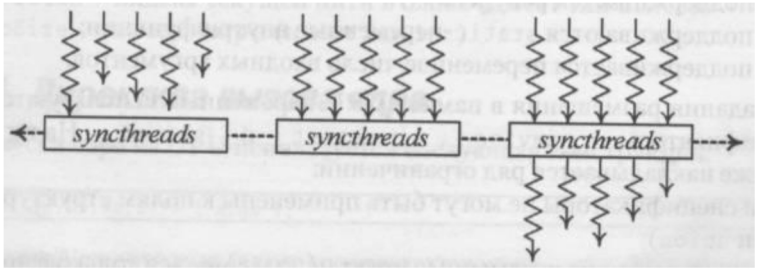

Рис. 3.3. Барьерная синхронизация

3.2.2 Расширения языка C

Программы для CUDA (соответствующие файлы обычно имеют расширение .cu) пишутся на «расширенном» C и компилируются при помощи команды nvcc.

Вводимые в CUDA расширения языка C состоят из:

- спецификаторов функций, показывающих, где будет выполняться функция и откуда она может быть вызвана;

- спецификаторов переменных, задающих тип памяти, используемый для данных переменных;

- директивы, служащей для запуска ядра, задающей как данные, так и иерар хию нитей;

- встроенных переменных, содержащих информацию о текущей нити;

- *runtime,* включающей в себя дополнительные типы данных.

3.2.1. Спецификаторы функций и переменных

В CUDA используется следующие спецификаторы функций (табл. 2.1).

Таблица 3.1. Спецификаторы функций в CUDA

Спецификатор	Функция	Функция может вызываться из
device	device (GPU)	device (GPU)
global	device (GPU)	host (CPU)
host	host (CPU)	host (CPU)

Спецификаторы _ host и ___ device могут быть использованы вместе

(это значит, что соответствующая функция может выполняться как на GPU, так ина CPU - соответствующий код для обеих платформ будет автоматически сгенерирован компилятором). Спецификаторы ___ global _ и _ host не могут быть использованы вместе.

Спецификатор ____global обозначает ядро, и соответствующая функция должна возвращать значение типа void.

На функции, выполняемые на GPU (_ device ____ и global), накладываются следующие ограничения:

- нельзя брать их адрес (за исключением _ global функций);

- не поддерживается рекурсия;

- не поддерживаются static-переменные внутри функции; - не поддерживается переменное число входных аргументов.

Для задания размещения в памяти GPU переменных используются следую щие спецификаторы - _____ device ____ , __constant и shared. На их использо вание также накладывается ряд ограничений:

- эти спецификаторы не могут быть применены к полям структуры (struct или union);

- соответствующие переменные могут использоваться только в пределах одного файла, их нельзя объявлять как extern;

- запись в переменные типа constant может осуществляться только

CPU при помощи специальных функций;

- __shared переменные не могут инициализироваться при объявлении.

Добавленные типы

В язык добавлены 1/2/3/4-мерные векторы из базовых типов (char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, longlong, float и double) - charl, char2, char3,char4, ucharl, uchar2, uchar3, uchar4, shortl, short2, short3, short4, ushortl, ushort2, ushort3, ushort4, intl, int2, int3, int4, uintl, uint2, uint3, uint4, long1, long2, long3, long4, ulong1, ulong2, ulong3, ulong4, floatl, float2, float3, float4, longlong1, longlong2, doublel и double2.

Обращение к компонентам вектора идет по именам - x, y, z и w. Для создания значений-векторов заданного типа служит конструкция вида make_<typeName>.

int2 a - make_int2 (1, 7); // Создает вектор (1, 7).

float3 u - make float3 (1, 2, 3, 4f); // Создает вектор (1.0f, 2.0f, 3.4f).

Обратите внимание, что для этих типов (в отличие от шейдерных языков GLSL, Cg и HLSL) не поддерживаются векторные покомпонентные операции, то есть нельзя просто сложить два вектора при помощи оператора «+» - это необходимо явно делать для каждой компоненты.

Также добавлен тип dim3, используемый для задания размерности. Этот тип основан на типе uint 3, но обладает при этом нормальным конструктором, инициализирующим все незаданные компоненты единицами.

```
dim3 blocks ( 16, 16 );
// Эквивалентно blocks ( 16, 16, 1 ).
dim3 grid  ( 256 );
// Эквивалентно grid ( 256, 1,1).
```

Добавленные переменные

В язык добавлены следующие специальные переменные:

□ gridDim - размер сетки (имеет тип dim3);

□ blockDim - размер блока (имеет тип dim3);

□ blockldx - индекс текущего блока в сетке (имеет тип uint3);

□ threadldx - индекс текущей нити в блоке (имеет тип uint3);

□ warpSize - размер warp'a (имееттип int).

Директива вызова ядра

Для запуска ядра на GPU используется следующая конструкция:

```
kernelName <<<Dg,Db,Ns,S>>>   (  args  );
```

Здесь *kernelName* это имя (адрес) соответствующей global функции. Через Dg обозначена переменная (или значение) типа d im3, задающая размерность и размер сетки (в блоках). Переменная (или значение) *Db* - типа dim3, задает размерность и размер блока (в нитях).

Необязательная переменная (или значение) *Ns* типа size_t задает дополнительный объем разделяемой памяти в байтах, которая должна быть динамически выделена каждому блоку (к уже статически выделенной разделяемой памяти), если не задано, то используется значение 0.

Переменная (или значение) S типа cudaStream_t задает поток *(CUDA stream),* в котором должен произойти вызов, по умолчанию используется поток 0.

Через *args* обозначены аргументы вызова функции *kemelName* (их может быть несколько).

Следующий пример запускает ядро с именем myKernel параллельно на n нитях, используя одномерный массив из двумерных (16x16) блоков нитей, и передает на вход ядру два параметра - a и n. При этом каждому блоку дополнительно выделяется 512 байт разделяемой памяти и запуск, производится на потоке my Stream.

```
myKernel<<<dim3(n/256) ,dim3(16,16) , 512,mystrean>>> 1 a, n ) ;
```

Добавленные функции

CUDA поддерживает все математические функции из стандартной библиотеки языка C. Однако при этом следует иметь в виду, что большинство стандартных математических функций используют числа с двойной точностью (double). Однако поскольку для современных GPU операции с double-числами выполняются медленнее, чем операции с float-числами, то предпочтительнее там, где это возможно, использовать float-аналоги стандартных функций. Так, float-аналогом функции sin является функция sinf.

Кроме того, CUDA предоставляет также специальный набор функций пониженной точности, но обеспечивающих еще большее быстродействие. Таким аналогом для функции вычисления синуса является функция sinf.

В табл. 2.2 приведены основные float-функции и их оптимизированные версии пониженной точности.

Для ряда функций можно задать требуемый способ округления. Используемый способ задается при помощи одного из следующих суффиксов:

- in - округление к ближайшему;
- r z - округление к нулю;
- ru - округление вверх;
- rd - округление вниз.

Таблица 3.2. Математические float-функции в CUDA

Функция	Значение
fadd_[rn, rz, ru.rd] (x, y)	Сложение, никогда не переводимое в команду FMAD
fmul_[rn, rz, ru.rd] (x, y)	Умножение, никогда не переводимое в команду FMAD
fmaf[rn, rz, ru.rd] (x, y, z)	*(x x y)* + *z*
frcp[rn, rz, ru.rd] (x)	*1/x*
fsqrt[rn, rz, ru,rd] (x)	*Vx*
fdiv[rn, rz, ru.rd] (x, y)	*x/y*
fdividef (x, y)	*x/y,* но если $2^{126} < y < 2^{128}$, то 0
_expf (x)	e^x
_logf (x)\	logr
log2f (x)	
_log10f(x)	log,(rr

sinf (x)	siar
_cosf (x)	co&дг
_sincosf (x, sptr, cptr)	*sptr = sin(jr); *cptr - cos(x)
_tanf (x)	taar
_powf (x, y)	***X"***
_int_as_float (x)	32 бита, образующие целочисленное значение, интерпретируются как float-значение. Так, значение OxCOOOOOO будет переведено в -2.Of
_float_as_int (x)	32 бита, образующие float-значение, интерпретируются как целочисленное значение. Так, значение 1 .Of будет переведено в -0x3F80000
_saturate (x)	min(0,max(I^))
_float_to_int_[rn, rz, ru.rd] (x)	Приведение float-значения к целочисленному значению с заданным округлением
_float_to_uint_[rn, rz, ru,rd] (x)	Приведение float-значения к беззнаковому целочисленному значению с заданным округлением
_jnt_to_float_[rn, rz, ru.rd] (x)	Приведение целочисленного значения к float-значению с заданным округлением
_uint_to_float_[rn, rz, ru,rd] (x)	Приведение беззнакового целочисленного значения к float-значению с заданным округлением
_float_tojl_[rn, rz, ru.rdj (x)	Приведение float-значения к 64-битовому целочисленному значению с заданным округлением
_floatjo_ull_[rn, rz, ru,rd] (x)	Приведение float-значения к 64-битовому беззнаковому целочисленному значению с заданным округлением

Кроме ряда оптимизированных функций для работы с числами с плавающей точкой, также есть ряд «быстрых» функций для работы с целыми числами, приводимых в табл. 2.3.

Таблица 3.3. Целочисленные функции в CUDA

Функция	Значение
_[u]mul24 (x, y)	Вычисляет произведение младших 24 бит целочисленных параметров x и y, возвращает младшие 32 бита результата. Старшие 8 бит аргументов игнорируются
[u]mulhi (x, y)	Возвращает старшие 32 бита произведения целочисленных операндов x и y
__[u]mul64hi (x, y)	Вычисляет произведение 64-битовых целых чисел и возвращает младшие 64 бита этого произведения
[u]sad (x, y, z)	Возвращает *z* + \x - y\
_clz (x)	Возвращает целое число от 0 до 32 включительно последовательных нулевых битов для целочисленного параметра x, начиная со старших бит
clzll (x)	Возвращает целое число от 0 до 64 включительно последовательных нулевых битов для целочисленного 64-битового параметра x, начиная со старших бит
__ffs **(x)**	Возвращает позицию первого (наименее значимого) единичного бита для аргумента x. Если x равен нулю, то возвращается нуль
ffsll (x)	Возвращает позицию первого (наименее значимого) единичного бита для целочисленного 64-битового аргумента x. Если x равен нулю, то возвращается нуль
_popc (x)	Возвращает число бит, которые равны единице в двоичном представлении 32-битового целочисленного аргумента x
popcll (x)	Возвращает число бит, которые равны единице в двоичном представлении 64-битового целочисленного аргументах
brev (x)	Возвращает число, полученное перестановкой (то есть биты в позициях к и 31-к меняются местами для всех к от 0 до 31) битов исходного 32-битового целочисленного

	аргументах
_brevll (x)	Возвращает число, полученное перестановкой (то есть биты в позициях к и 63-к меняются местами для всех к от 0 до 63) битов исходного 64-битового целочисленного аргумента х

Использование этих функций может заметно поднять быстродействие про граммы, однако при этом следует иметь в виду, что выигрыш от использовании некоторых функций (mul24) может исчезнуть для GPU следующих поколений.

3.3 Основы CUDA API

CUDA предоставляет в распоряжение программиста ряд функций, которые могут быть использованы только CPU (так называемый *CUDA host API).* Эти фун кции отвечают за:

- управление GPU;
- работу с контекстом; - работу с памятью;
- работу с модулями;
- управление выполнением кода;
- работу с текстурами;

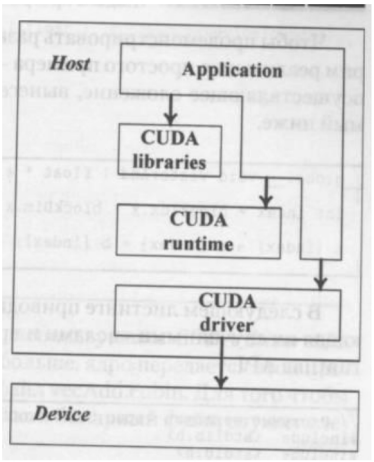

Рис. 2.4. Программный стек CUDA

- взаимодействие с OpenGL и Direct3D.

CUDA API для CPU *(host API)* выступает в двух формах:

□ низкоуровневый CUDA driver API;

□ высокоуровневый CUDA runtime API (реализованный через *CUDA driver API).*

Эти API являются взаимоисключающими - в своей программе вы можете работать только с одним из них.

На рис. 2.4 приведены различные уровни программно-аппаратного стека CUDA Как видно, все взаимодействие с GPU происходит только через драйвер устройства. Над ним находятся *CUDA driver API, CUDA runtime API* и CUDA-библиотеки.

Программа может взаимодействовать с верхними уровнями - есть CUDA driver API, CUDA runtime API и библиотеки (на данный момент это библиотеки CUFFT, CUBLAS и CUDPP), имеющие свои API.

3.3.1 CUDA driver API

Низкоуровневый API, дающий больше возможностей программисту, но и требующий большего объема кода. Данный API реализован в динамической библиотеке nvcuda, и все имена в нем начинаются с префикса cu.

Следует иметь в виду, что у каждой функции *CUDA runtime API* есть прямой аналог в *CUDA driver API,* то есть переход с *CUDA runtime API* на *CUDA driver API* не очень сложен, обратное в общем случае не верно.

CUDA driver API обладает обратной совместимостью с более ранними версиями.

К числу недостатков этого API относятся больший объем кода и необходимость явных настроек, требование явной инициализации и отсутствие поддержки режима эмуляции (позволяющего компилировать, запускать и отлаживать коды на CUDA с CPU).

3.3.2 CUDA runtime API

Это высокоуровневый API, к тому же *CUDA runtime API не* требует явной инициализации - она происходит автоматически при первом вызове какой-либо его функции.

Данный API поддерживает эмуляцию, реализован в динамической библиотеке cudart, все имена начинаются с префикса cuda.

Одним из плюсов данного API является возможность использования дополнительных библиотек (CUFFT, CUBLAS и CUDPP).

Чтобы продемонстрировать разницу в использовании этих двух API, рассмотрим реализацию простого примера - сложение двух векторов длины n. Само ядро, осуществляющее сложение, вынесено в отдельный файл (vecAdd.cu), приводимый ниже.

```
_global_void vectorAdd ( float * a, float * b, float * c){
    int index = blockldx.x * blockDim.x + threadldx.x;
    c [index] = a [index] + b [index];
}
```

В следующем листинге приводится программа, создающая векторы, заполня ющая их случайными числами и производящая их сложение при помощи CUDA runtime API.

```
    // Сложение векторов через CUDA runtime API. ((include
<stdlib.h> ((include <stdio.h> ((include "vecAdd.cu"
    «define EPS O.OOOOlf
    // Заполнить массив случайными числами. void randomlnit (
float * a, int n ) (
    fo: ( int i = 0; i < n; i*+ )
    a [i] * rand O / (float) RAND_MAX; }
    int main ( int argc,    char * argv   [] )
f
    const unsigned int blocksize =   512;
    const unsigned int numBlocks =   3;
    const unsigned int numltems =    numBlocks * blockSize;
    // Выбрать первый GPU Оля работы. cudaSetDevicel O ); // pick
first device
    // Выделить память CPU. float * a = new float [numltems];
float * b = new float [numltems]; float * c = new float
[numltems];
    // Проинициализировать входные массивы. randomlnit ( a,
numltems ); randomlnit ( b, numltems );
    // Выбелить память GPU. float * aDev, * bDev, * cDev,-
```

```
    cudaMalloc ( (void **)&aDev, numltems * sizeof ( float ) );
cudaMalloc ( (void **)&bDev, numltems * sizeof ( float ) );
cudaMalloc ( (void **)&cDev, numltems * sizeof ( float ) );
    // Скопировать банные из памяти CPU в память GPU.
    cudaMemcpy ( aDev, a, numltems * sizeof ( float ),
cudaMemcpyHostToDevice ); cudaMemcpy ( bDev, b, numltems * sizeof
( float ), cudaMemcpyHostToDevice );
    // Запустить ядро. vectorAdd<<<numBlocks, blockSize>>> (
aDev, bDev, cDev );
    // Скопировать результат в память CPU.
    cudaMemcpy ( (void *) c, cDev, numltems * sizeof ( float ),
cudaMemcpyDeviceToHost ) ;
    // Проверить результат. for ( int i = 0; i < numltems; i++ )
if ( fabs ( a [il + b [i] - c [i] ) > EPS ) printf ( "Error at
index %d\n", i );
    // Освободить выбеленную память, delete [] a; delete [] b;
delete [] c;
    cudaFiee ( aDev ); cudaFree ( bDev ); cudaFree ( cDev );
```

Далее мы будем в основном использовать именно этот API как более простой. При необходимости код, основанный на нем, может быть переписан на *CUDA driver API,* поскольку последний содержит полные аналоги всех функций ***CUD A runtime API.***

Важным моментом работы с *CUDA,* на который следует сразу же обратить внимание, является то, что многие функции API - асинхронные, то есть управление возвращается еще до реального завершения требуемой операции.

К числу асинхронных операций относятся:

□ запуск ядра;

□ функции копирования памяти, имена которых оканчиваются на Async;

□ функции копирования памяти device <-> device;

□ функции инициализации памяти.

Для синхронизации текущей нити на CPU с GPU используется функция cudaThreadSynchronize, которая дожидается завершения выполнения всех операций CUDA ранее вызванных с данной нити CPU.

cudaError_t cudaThreadSynchronize (void) ;

CUDA поддерживает синхронизацию через потоки *(streams)* - каждый поток задает последовательность операций, выполняемых в строго определенном порядке. При этом порядок выполнения операций между разными потоками не является строго определенным и может изменяться.

Каждая функция *CUDA runtime API* (кроме запуска ядра) возвращает значение типа cudaError_t. При успешном выполнении функции возвращается значение cudaSuccess, в противном случае возвращается код ошибки.

Получить описание ошибки в виде строки по ее коду можно при помощи функции cudaGetErrorString:

char * cudaGetErrorString (cudaError t code);

Также можно получить код последней ошибки при помощи функции cudaGetLastError:

cudaError t cudaGetLastError ();

3.3.3 Получение информации об имеющихся GPU

Прежде чем начать работу с GPU, очень важно получить максимально полную информацию обо всех имеющихся GPU (CUDA может работать сразу с несколькими GPU, если только они не соединены через SLI) и об их возможностях.Для этого можно воспользоваться *CUD A runtime API,* который предоставляет! простой способ получить информацию об имеющихся GPU, которые могут быть использованы CUDA, и обо всех их возможностях. Информация о возможностях GPU возвращается в виде структуры cudaDeviceProp.

```
struct cudaDeviceProp (
char name[256];     // Название устройства.
size_t totalGlobalMem;      // Полный объем глобальной памяти в байтах.
size_t sharedMemPerBlock;    // Объем разбеляемой памяти в блоке в
байтах,
int regsPerBlock;   // Количество 32-битовых резистров в блоке,
int warpSize;       // Размер warp'a.
size_t memPitch,-   // Максимальный pitch в байтах, для функций
```

```
    // копирования памяти, выделенной через
    // cudaMallocPitch
int maxThreadsPerBlock;    // Макс. число активных нитей в блоке,
int maxThreadsDim [31j     // Макс. размер блока по каждому измерению,
int maxGridSize  13];      // Макс. размер сетки по каждому измерению.
size_t totalConstMem;      // Объем константной памяти в байтах,
int major;          // Compute Capability, старший номер,
int minor;          // Compute Capability, младший номер,
int clockRate;      // Частота в килогерцах.
size_t textureAlignment;    // Выравнивание памяти для текстур,
int deviceOverlap; // Можно ли осуществлять копирование параллельно
    //с вычислениями.
int multiProcessorCount;     // Количество мультипроцессоров в GPU.
int kernelExecTimeoutEnables; // 1- есть ограничение на время
    //выполнения ядра
int integrated;     // 1, если GPU встроено в материнскую плату
int canMapHostMemory;       // 1, можно отображать память CPU в память
    // CUDA.для использования функциями cudaHostAlloc,
    // cudaHostGetDevicePointer
int computeMode;    // Режим, в котором нахоОится GPU.
                          // Возможные значения:
    // cudaComputeModeDefault,
    // cudaComputeModeExclusive - только одна нить может
    // звать cudaSetDevice для данного GPU,
    // cudaComputeModeProhibited - ни одна нить не может
    // вызвать cudaSetDevice для данного GPU. )
```

Для обозначения возможностей GPU CUDA использует понятие *Computer Capability,* выражаемое парой целых чисел - *major.minor.* Первое число обозначает глобальную архитектурную версию, второе - небольшие изменения. Так, GPU GeForce 8800 Ultra/GTX/GTS имеют Compute Capability, равную 1.0.

Таблица 3.4. Compute Capability для основных моделей GPU

Модель GPU	Количество мультипроцессоров	Compute Capability
GeForce GTX 295	2x30	1.3
GeForce GTX 285, GTX 280	30	1.3
GeForce GTX 260	24	1.3
GeForce 9800 GX2	2x16	1.1
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 880 GTS 512	16	1.1
GeForce 8800 Ultra, 8800 GTX	16	1.0
GeForce 9800 GT, 8800 GT, GTX 280M, 9800M GTX	14	1.1
GeForce GT 130, 9600 GSO, 8800 GS, 8800M GTX, GTX260M.9800MGT	12	1.1
GeForce 8800 GTS	12	1.0
GeForce 9600 GT, 8800M GTS, 9800M GTS	8	1.1
GeForce 9700M GT	6	1.1
GeForce GT 120, 9500 GT, 8600 GTS, 8600 GT, 9700M GT, 9650M GS, 9600M GT, 9600M GS, 9500M GS, 8700M GT, 8600M GT, 8600M GS	4	1.1
GeForce G100, 8500 GT, 8400 GS, 8400M GT, 9500M GT, 9300M G, 8400M GS, 9400 mGPU, 9300 mGPU, 8300 mGPU, 8200 mGPU, 8100 mGPU	2	1.1
GeForce 9300M GS, 9200M GS, 9100M G, 8400M G	1	1.1
TeslaS1070	4x30	1.3
TeslaC1060	30	1.3
Tesla S870	4x16	1.0
Tesla D870	2x16	1.0
Tesla C870	16	1.0
Quadro Plex 2200 D2	2x30	1.3
QuadroPlex2100 D4	4x4	1.1
Quadro Plex 2100 Model S4	4x16	1.0

Quadro Plex 1000 Model IV	2x16	1.0
Quadro FX 5800	30	1.3
Quadro FX 4800	24	1.3
Quadro FX 4700 X2	2x14	1.1
Quadro FX 3700M	16	1.1
Quadro FX 5600	16	1.0
Quadro FX 3700	14	1.1
Quadro FX 3600M	12	1.1
Quadro FX 4600	12	1.0
Quadro FX 2700M	6	1.1
Quadro FX 1700, FX570, NVS320M, FX 1700M, FX 1600M, FX 770M, FX 570M	4	1.1
Quadro FX 370, NVS 290, NVS 140M, NVS 135M, FX360M	2	1.1
Quadro FX370M, NVS 130M	1	1.1

Ниже приводится исходный текст простой программы, перечисляющей все доступные GPU и их основные возможности.

```
#include <stdio.h>
int main ( int argc,   char  *  argv   []   )
{int deviceCount;
    cudaDeviceProp devProp;
    cudaGetDeviceCount  (  StdeviceCount  )  ;
    printf  (  "Found %d devices\n", deviceCount  );
    for  (  int device = 0; device < deviceCount; device++  )  {
      cudaGetDevicePropeities { fcdevProp, device  );
      printf  (  "Device %d\n", device  )  ;
      printf  (  "Compute capability     : %d.%d\n", devProp.major,
     devProp.minor  );
      printf  (  "Name  :  %s\n", devProp.name  );
      printf  (  "Total Global Memory    : %d\n",
       devProp.totalGlobalMem  );
```

```
    printf ( "Shared memory per block: %d\n",
     devProp.sharedMemPerBlock );
     piintf ( "Registers per block   : %d\n",
    devProp.regsPerBlock );
   printf ( "Warp size    : %d\n", devProp.waipsize );
   printf ( "Max threads per block : %d\n",
    devProp.maxThreadsPerBlock );
   printf ( "Total constant memory  : %d\n",
    devProp.totalConstMem );
   printf ( "Clock Rate   : %d\n", devProp.clockRate );
   printf ( "Texture Alignment     : %u\n",
    devProp.textureAlignment );
   printf ( "Device Overlap       : %d\n",
    devProp.deviceOverlap );
   printf ( "Multiprocessor Count  : %d\n",
    devProp.multiProcessorCount );
   printf ( "Max Threads Dim      ! %d %d %d\n",
    devProp.maxThreadsDim [0],
   devProp.maxThreadsDim [1], devProp.maxThreadsDim [2] );
   printf ( "Max Grid Size        i %d %d %d\n",
    devProp.maxGridSize [0],
   devProp.maxGridSize [1], devProp.maxGridSize [21 ); }
 return 0;
```

Для установки CUDA на компьютер необходимо по адресу http://www.nvidiacom/object/cuda get.html скачать и установить на свой компьютер следующие файлы: CUDA driver, CUDA Toolkit и CUDA SDK.

Перейдите в своем веб-браузере на эту страницу, выберите в комбо-боксе тип и разрядность своей операционной системы, после чего вы получите ссылки для скачивания нескольких последних версий CUDA (рис. 2.5).

Версия CUDA для платформы Linux также содержит CUDA-отладчик (cuda-gdb), устанавливаемый как отдельная компонента.

После установки всех этих компонент можно будет посмотреть готовые примеры, изучить прилагаемую документацию и начать писать свои программы с использованием CUDA

Обратите внимание, что для компиляции программ на CUDA также потребуется установленный компилятор с C/C++. В качестве такого компилятора может выступать компилятор, входящий в состав Microsoft Visual Studio, а также компиляторы mingw и cygwin.

Это связано с тем, что используемый для компиляции программ на CUDAI компилятор nvcc использует внешний компилятор для компиляции частей кои выполняемых на CPU.

3.4 Особенности работы с памятью в CUDA

3.4.1 Типы памяти CUDA

Одним из серьезных отличий между GPU и CPU являются организация памяти и работа с ней. Обычно большую часть CPU занимают кеши различных уровней. Основная часть GPU отведена на вычисления. Как следствие, в отличие от CPU, где есть всего один тип памяти с несколькими уровнями кеширования в самом CPU, GPU обладает более сложной, но в то же время гораздо более документированной структурой памяти.

Чисто физически память GPU можно разделить на DRAM и на память, размещенную непосредственно на GPU (точнее, в потоковых мультипроцессорах). Однако классификация памяти в CUDA не ограничивается ее чисто физическим расположением.

В табл. 3.4 приводятся доступные виды памяти в CUDA и их основные характеристики.

Таблица 3.4. Типы памяти в CUDA

Тип памяти	Расположение	Кеши-руется	Доступ	Уровень доступа	Время жизни
Регистры	Мультипроцессор	Нет	R/w	Per-thread	Нить
Локальная	DRAM	Нет	R/w	Per-thread	Нить
Разделяемая	Мультип	Нет	R/w	Все нити	Блок

	роцессор			блока	
Глобальная	DRAM	Нет	**R/w**	Все нити и CPU	Выделяется CPU
Константная	DRAM	**Да**	**R/o**	Все нити и CPU	Выделяется CPU
Текстурная	DRAM	**Да**	**R/o**	Все нити и CPU	Выделяется CPU

Как видно из приведенной таблицы, часть памяти расположена непосредственно в каждом из потоковых мультипроцессоров (регистры и разделяемая память), часть памяти размещена в DRAM.

Наиболее простым видом памяти является регистровая память (регистры). Каждый потоковый мультипроцессор содержит 8192 или 16 384 32-битовых регистров (для обозначения всех регистров потокового мультипроцессора используется термин register file).

Имеющиеся регистры распределяются между нитями блока на этапе компиляции (и, соответственно, влияют на количество блоков, которые может выполнять один мультипроцессор).

Каждая нить получает в свое монопольное пользование некоторое количество регистров, которые доступны как на чтение, так и на запись (read/write). Нить не имеет доступа к регистрам других нитей, но свои регистры доступны ей на протяжении выполнения данного ядра.

Поскольку регистры расположены непосредственно в потоковом мультипроцессоре, то они обладают максимальной скоростью доступа.

Если имеющихся регистров не хватает, то для размещения локальных данных (переменных) нити используется так называемая локальная память, размещенная в DRAM. Поэтому доступ к локальной памяти характеризуется очень высокой латентностью - от 400 до 600 тактов.

Следующим типом памяти в CUD A является так называемая разделяемая (shared) память. Эта память расположена непосредственно в потоковом мультипроцессоре, но она выделяется на уровне блоков - каждый блок получает в свое распоряжение одно и то же количество разделяемой памяти. Всего каждый мультипроцессор содержит 16 Кбайт

разделяемой памяти, и от того, сколько разделяемой памяти требуется блоку, зависит количество блоков, которое может быть запущено на одном мультипроцессоре.

Разделяемая память обладает очень небольшой латентностью (доступ к ней может быть настолько же быстрым, как и доступ к регистрам), доступна всем нитям блока, как на чтение, так и на запись. Более подробно работа с разделяемой памятью будет рассмотрена в следующей главе.

Глобальная память - это обычная DRAM-память, которая выделяется при помощи специальных функций на CPU. Все нити сетки могут читать и писать в глобальную память.

Поскольку глобальная память расположена вне GPU, то естественно, что она обладает высокой латентностью (от 400 до 600 тактов). Правильное использование глобальной памяти является одним из краеугольных камней оптимизации в CUDA

Константная и текстурная память хоть и расположены в DRAM, тем не менее кешируются, поэтому скорость доступа к ним может быть очень высокой (намного выше скорости доступа к глобальной памяти). Обратите внимание, что оба этих типа памяти доступны сразу всем нитям сетки, но только на чтение, запись в них может осуществлять только CPU при помощи специальных функций.

Общий объем константной памяти ограничен 64 Кбайтами. Текстурная память позволяет получить доступ к возможностям GPU по работе с текстурами и будет подробно рассмотрена в отдельной главе.

3.4.2 Работа с константной памятью

Константная память выделяется непосредственно в коде программы при по мощи спецификатора __constant . Все нити сетки могут читать из нее данные,

и чтение из нее кешируется. CPU имеет доступ к ней как на чтение, так и на запись при помощи следующих функций:

```
cudaError_t cudaMemcpyToSymbol ( const char * symbol, const void
* src, size_t count, size_t offset, enum cudaMemcpyKind kind );
cudaError_t cudaMemcpyFromSymbol { void * dst, const char *
symbol,
size_t count, size_t offset, enum cudaMemcpyKind kind );
cudaError_t cudaMemcpyToSymbolAsync ( const char * symbol,
```

```
const void * src, size_t count, size_t offset, enum
cudaMemcpyKind kind, cudaStream_t stream );
cudaError_t cudaMemcpyFromSymbolAsync ( void * dst, const char *
symbol,
size_t count, size_t offset, enum cudaMemcpyKind kind,
cudaStream t stream ) ;
```

В качестве значения параметра kind выступает одна из следующих констант, задающих направление копирования, - cudaMemcpyHostToHost, cudaMemcpy-HostToDevice, cudaMemcpyDeviceToHost и cudaMemcpyDeviceToDevice. Параметр stream позволяет организовывать несколько потоков команд, если вы не используете эту возможность, то следует использовать поток по умолчанию - 0.

Ниже приводится простейший пример - фрагмент кода, объявляющий массив в константной памяти и инициализирующий этот массив данными из массива из памяти CPU.

```
__constant     float contsData [256]; // константная память GPU
float     hostData  [256]; // Оанные в памяти CPU
// скопировать Оанные из памяти CPU в // константную память GPU
cudaMemcpyToSymbol ( constData, hostData, sizeof ( data ), 0,
cudaMemcpyHostToDevice );
```

Поскольку константная память кешируется, то она является идеальным местом для размещения небольшого объема часто используемых неизменяемых данных, которые должны быть доступны всем нитям сетки сразу.

3.4.3 Работа с глобальной памятью

Основная часть DRAM GPU доступна приложениям именно как глобальная память. Доступный ее объем фактически определяется объемом установленной DRAM-памяти на устройстве. Глобальная память является основным местом для размещения и хранения большого объема данных для обработки ядрами, она сохраняет свои значения между вызовами ядер, что позволяет через нее передавать данные между ядрами.

Глобальная память выделяется и освобождается CPU при помощи следующих вызовов:

cudaError_t cudaMalloc (void ** devPtr, size_t size);

cudaError_t cudaFree (void * devPtr);

cudaError t cudaMallocPitch (void ** devPtr, size t * pitch, size t width, size t height);

Функции cudaMalloc и cudaFree являются стандартными функциями выделения и освобождения глобальной памяти. Поскольку для эффективного доступа к глобальной памяти важным требованием является выравнивание данных в памяти, то для выделения памяти под двухмерные массивы лучше использовать функцию cudaMallocPitch, которая при выделении памяти может увеличить объем памяти под каждую строку, чтобы гарантировать выравнивание всех строк. При этом дополнительный объем памяти в байтах, служащий для выравнивания строки, возвращается через параметр pitch.

Если при помощи функции cudaMallocPitch выделялась память под матрицу из элементов типа T, то для получения адреса элемента, расположенного в строке row и столбце col, используется следующая формула:

```
T * item = (T *) ((char «) baseAddress + row * pitch) + col;
```

Обратите внимание, что хотя функции выделения глобальной памяти и возвращают указатель на память, но это указатель на память GPU, поэтому доступ по данному указателю может осуществлять только код, выполняемый на GPU. Для доступа CPU к этой памяти следует использовать функции копирования памяти:

```
cudaErroi_t cudaMemcpy ( void * dst, const void * src, size_t size, enum cudaMemcpyKind kind );

cudaErroi_t cudaMemcpyAsync ( void * dst, const void * src, size_t size, enum cudaMemcpyKind kind, cudaStream_t stream );

cudaError_t cudaMemcpy2D ( void * dst, size_t dpitch, const void * src, size_t spitch, I size_t width, size_t height, enum cudaMemcpyKind kind );

cudaError_t cudaMemcpy2DAsync ( void * dst, size_t dpitch, const void ♦ src, size_t spitch,  size t width, size t height, enum cudaMemcpyKind kind, cudaStream t strea:' ;
```

В этих функциях параметр kind задает направление копирования и может принимать только одно из следующих значений - cudaMemcpyHostToHost, cuda-MemcpyHostToDevice, cudaMemcpyDeviceTbHost и cudaMemcpyDeviceToDevice.

Обратите внимание, что копирование памяти между CPU и GPU является очень дорогостоящей операцией, и число подобных операций должно быть минимизировано. Это связано с тем, что скорость передачи данных ограничивается скоростью передачи данных по шине PCI Express (сейчас это около 8 Гбайт/сек).

В то же время передача данных в пределах DRAM оказывается намного выше -так, у GPU GeForce GTX 280 она составляет 141 Гбайт/сек.

Скорость передачи данных между CPU и GPU может быть повышена за счет использования так называемой page-locked (или pinned)-памяти. Для выделения такой памяти служат следующий вызов:

```
cudaError t cudaMallocHost ( void ** devPtr, size t size );
```

При асинхронном копировании памяти между CPU и GPU (cudaMemcpyAsync) должна использоваться именно pinned-память. Однако при этом важно учитывать, что такая память является ограниченным ресурсом, и чрезмерное ее использование может отрицательно сказаться на быстродействии всей системы.

Пример: построение таблицы значений функции с заданным шагом

Рассмотрим простейший пример - построение таблицы значений заданной функции с заданным шагом.

Для этого необходимо выделить два массива одинакового размера для хранения результата: один - в памяти CPU, другой - в памяти GPU. После этого запускается ядро, заполняющее массив в глобальной памяти заданными значениями. После завершения ядра необходимо скопировать результаты вычислений из памяти GPU в память CPU и освободить выделенную глобальную память.

```
// ядро, осуществляющее заполнение массива _global__ void

tableKernal ( float * devPtr, float step ) {
// получить глобальный адрес нити
int index = blockldx.x * blockDim.x + threadldx.x;
// вычислить значение аргумента float x    = step * index;
```

```
// используем «быстрые» версии функций
   devPtr [index] -    sinf(    sqrtf ( x ) );
}
    void buildTable ( float * res, int n, float step ) {
    float * devPtr;
    // убедимся, что n кратно 256 assert ( n % 256 == 0 );
    // выделяем глобальную память под таблицу cudaMalloc (
    SdevPtr, n * sizeof (float) );
    // запускаем ядро для вычисления значений
    tableKernel<<<dim3(n/256),dim3(256)>>>(devPtr, step ) ;
    // копируем результат из глобальной памяти в память CPU
    cudaMemcpy ( res, devPtr, n * sizeof(float),
    cudaMemcpyDeviceToHost );
    // освобождаем выделенную глобальную память cudaFree  (
    devPtr ); }
```

Пример: транспонирование матрицы

В качестве следующего примера рассмотрим задачу транспонирования квадратной матрицы A размера NxN, далее мы будем считать, что N кратно 16.

Поскольку сама матрица A двухмерна, то будет удобно использовать двухмерную сетку и двухмерные блоки. В качестве размера блока выберем 16x16, это позволит запустить до трех блоков на одном мультипроцессоре. Тогда для транспонирования матрицы можно использовать следующее ядро:

```
    __global__ void transposel ( float * inData, float * outData,
int n )
    {
    unsigned int xIndex  = blockDim.x *  blockIdx.x +
threadIdx.x;
    unsigned int yIndex  = blockDim.y *  blockIdx.y +
threadIdx.y;
    unsigned int inIndex = xIndex + n *  yIndex;
    unsigned int outIndex = yIndex + n * xIndex;
```

```
    outData [outIndex] = inData [inIndex]; }
```

Пример: перемножение двух матриц

Несколько более сложным примером (к которому мы также еще вернемся в следующей главе) будет задача перемножения двух квадратным матриц A и B (также будем считать, что они обе имеют размер NxN, где N кратно 16). Произведение C двух матриц A и B задается при помощи следующей формулы:

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \cdot b_{k,j} .$$

Как и в предыдущем примере, будем использовать двухмерные блоки 16x16 и двухмерную сетку. Ниже приводится простейший пример «перемножения в лоб».

```
    __global__ void matMult ( float  * a, float * b, int n, 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;        //  здесь накапливается результат
    //              смещение для a [i][0]
int  ia  = n * BLOCK_SIZE * by + n * ty;
                      // смещение Оля b [0] [j]
    int  ib = BLOCK_SIZE * bx + tx;
    // перемножаем и суммируем for ( int k = 0; k Sit; n; k++ )
    sum += a [ia + k] * b [ib + k*n] ;
    // сохраняем результат в глобальной памяти // смещение Оля
записываемого элемента int ic = n * BLOCK_SIZE * by * BLOCK_SIZE
* bx;
    c [ic + n * ty + tx] = sum;
    }
```

Как легко можно убедиться, в данном примере для нахождения одного элемента произведения двух матриц нам нужно прочесть 2xN значений из глобальной памяти и выполнить 2xN арифметических операций. В данном случае основным фактором,

лимитирующим быстродействие данной программы, является чтение из глобальной памяти (а не вычисления), подобные случаи называются memory bound.

3.4.5 Оптимизация работы с глобальной памятью

Поскольку глобальная память обладает столь высокой латентностью, то крайне важными являются понимание способов доступа к ней и соответствующая оптимизация доступа.

Обращение к глобальной памяти происходит через чтение/запись 32/64/128-битовых слов. Крайне важным является то, что адрес, по которому происходит доступ, должен быть выровнен по размеру слова, то есть кратен размеру слова в байтах.

Так, если происходит чтение 32-битого слова по адресу 0, то потребуется одно обращение к памяти. Если же чтение будет происходить с адреса 1, то потребуются два обращения к памяти, каждое из которых будет выровнено (первое читает по адресу 0, а второе - по адресу 4).

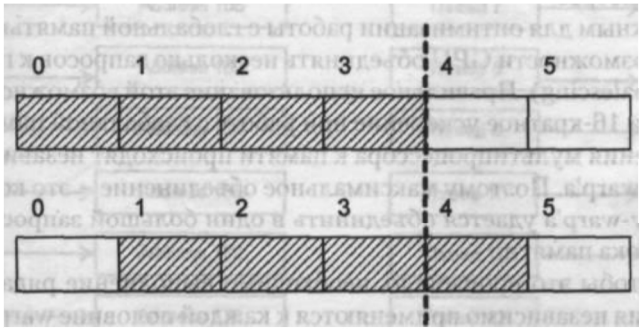

Рис. 3.1. Пример выровненного (сверху) и невыровненного (внизу) 4-байтового блока

Хотя все функции, выделяющие глобальную память, выделяют ее выровненной по 256 байтам, тем не менее проблемы с выравниванием могут возникать и в этом случае. Пусть у нас есть массив из следующих структур, выделенный в глобальной памяти:

```
struct vec3 {
float x,  y,   z; };
```

Хотя каждый элемент массива (длиной в 12 байт) полностью помещается в 16 байтах, но даже если адрес первого элемента массива и выровнен по 16 байтам, то адрес

следующего элемента уже не будет выровнен по 16 байтам, и его чтение потребует двух обращений.

Поэтому если элементы этого массива читаются нитями, то каждое второе обращение к массиву будет осуществляться в два обращения, то есть будет заметный проигрыш в скорости.

Проще всего это исправить - обеспечить выравнивание элементов массива, которое можно достигнуть следующим способом (или добавить один фиктивный элемент):

```
struct    align    (16) vec3
{
float x, y, z;
};
```

Теперь все элементы массива будут находиться на адресах, кратных 16, что обеспечит чтение одного элемента за раз. Таким образом, хотя мы и увеличили объем выделяемой и читаемой памяти, но работа с этой памятью будет происходить заметно быстрее.

Крайне важным для оптимизации работы с глобальной памятью является использование возможности GPU объединять несколько запросов к глобальной памяти в один (coalescing). Правильное использование этой возможности позволяет получить почти 16-кратное ускорение при работе с глобальной памятью.

Все обращения мультипроцессора к памяти происходят независимо для каждой половины warp'a. Поэтому максимальное объединение - это когда все запросы одного полу-warp'a удается объединить в один большой запрос на чтение непрерывного блока памяти.

Для того чтобы это произошло, необходимо выполнение ряда условий, при этом эти условия независимо применяются к каждой половине *warp'*а. Сами усло вия зависят от используемого GPU, а точнее от его compute capability.

Чтобы GPU с compute capability 1.0 или 1.1 произвел объединение запросов нитей половины warp'a, необходимо, чтобы были выполнены следующие условия:

□ все нити обращаются к 32-битовым словам, давая в результате один 64-байтовый блок, или все нити обращаются к 64-битовым словам, давая в результате один 128-байтовый блок;

□ получившийся блок выровнен по своему размеру, то есть адрес получаю щегося 64-байтового блока кратен 64, а адрес получающегося 128-байтового блока кратен 128;

□ все 16 слов, к которым обращаются нити, лежат в пределах этого блока;

□ нити обращаются к словам последовательно - k-ая нить должна обращаться к к-му слову (при этом допускается что отдельные нити пропустят обращение к соответствующим словам).

Если нити полу-warp'а не удовлетворяют какому-либо из данных условий, то каждое обращение к памяти происходит как отдельная транзакция. На следующих рисунках приводятся типичные паттерны обращения, дающие объединения и не дающие объединения.

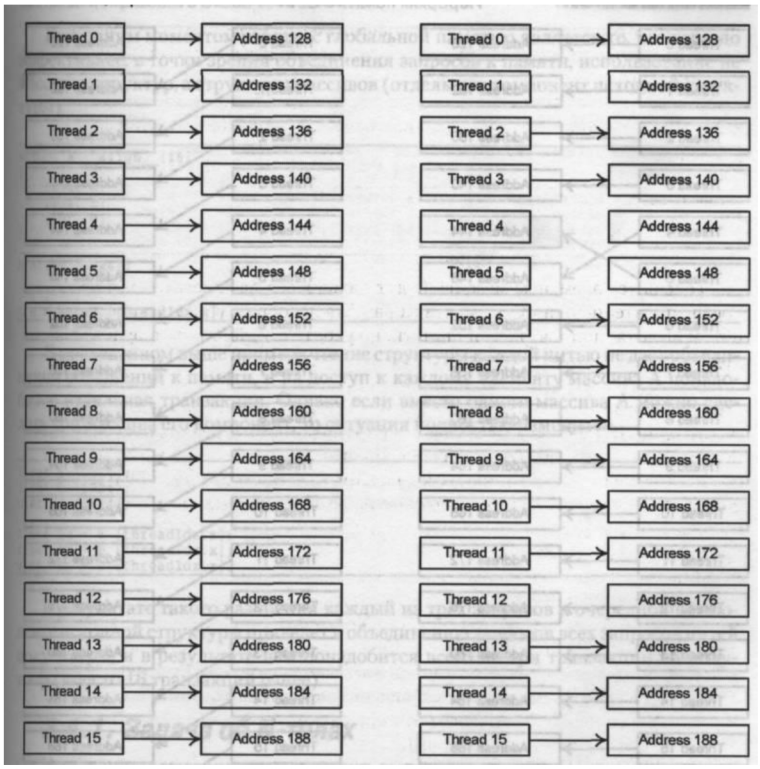

Рис. 3.2. Паттерны обращения к памяти, дающие объединение для GPU с Compute Capability 1.0 и 1.1

Для GPU с Compute Capability 1.2 и выше объединение запросов в один будет происходить, если слова, к которым идет обращение нитей, лежат в одном сегменте размера 32 байта (если все нити обращаются к 8-битовым словам), 64 байта (если все нити обращаются к 16-битовым словам) и 128 байт (если все нити обра щаются к 32-битовым или 64-битовым словам). Получающийся сегмент (блок) должен быть выровнен по 32/64/128 байтам.

Обратите внимание, что в этом случае порядок, в котором нити обращаются к словам, не играет никакой роли, и ситуация на рис. 3.3 слева приведет к объединению всех

запросов в одну транзакцию (для случая слева произойдет объединение запросов в две транзакции).

Если идет обращение к n соответствующим сегментам, то происходит группировка запросов в n транзакций (только для GPU с Compute Capability 1.2 и выше).

Еще одним моментом работы с глобальной памятью является то, что гораздо эффективнее, с точки зрения объединения запросов к памяти, использование не массивов структур, а структуры массивов (отдельных компонент исходной структуры).

```
struct A     __ align__ (16) {
float a;
float b;
uint    c; };
A  array   [1024] ;
. . .
A a = array  [ threadldx. x] ;
```

В приведенном выше примере чтение структуры каждой нитью не даст объединения обращений к памяти, и на доступ к каждому элементу массива A понадобится отдельная транзакция. Однако если вместо одного массива A можно сделать три массива его компонент, то ситуация полностью изменится.

```
float a [1024] ; float b [1024] ; uint    c   [102 4] ;
. . .
float fa = a [threadldx.x]; float fb = b [threadldx.x]; uint  uc = c  [threadldx.x];
```

В результате такого разбиения каждый из трех запросов к очередной компоненте исходной структуры приведет к объединению запросов всех запросов нитей полу-warp'а, и в результате нам понадобится всего по три транзакции на полу-warp (вместо 16 транзакций ранее).

3.4.6 Работа с разделяемой памятью

Разделяемая память, размещенная непосредственно в самом мультипроцессоре и доступная на чтение и запись всем нитям блока, является одним из важнейших отличий CUDA от традиционного (то есть основанного на использовании графических API)

GPGPU. Правильное использование разделяемой памяти играет огромную роль в написании эффективных программ для GPU.

Для современных GPU каждый потоковый мультипроцессор содержит 16 Кбайт разделяемой памяти. Она поровну делится между всеми блоками сетки, исполняемыми на мультипроцессоре.

Кроме того, разделяемая память используется также для передачи параметров при запуске ядра на выполнение, поэтому желательно избегать передачи большого объема входных параметров, непосредственно передаваемых ядру в конструкции вызова ядра. При необходимости для передачи большого объема параметров можно воспользоваться кешируемой константной памятью.

Существуют два способа управления выделением разделяемой памяти. Са мый простой способ заключается в явном задании размеров массивов, выделяе мых в разделяемой памяти (то есть описанных с использованием спецификатора shared).

```
__global__       void incKernal   (  float * a  )
{        ~
// Явно задали выведение 256*4 байтов на блок.
__shared__     float but   [256];
// Запись значения из глобальной памяти в разделяемую, buf
[threadIdx.x]   = a  [blockIdx.x * blockDim.x +  threadIdx.x];


.   .   .   .
}
```

При этом способе задания компилятор сам произведет выделение необходимого количества разделяемой памяти для каждого блока при запуске ядра.

Кроме того, можно также при запуске ядра задать дополнительный объем разделяемой памяти (в байтах), который необходимо выделить каждому блоку при запуске ядра. Для доступа к такой памяти используется описание массива без явно заданного размера. При запуске ядра начало такого массива будет соответствовать началу дополнительно выделенной разделяемой памяти.

```
__global__ void kernal ( float * a )
```

```
    __shared   float buf []; // Размер явно не указан.
    // Запись значения из глобальной памяти в разделяемую,
buf [threadIdx.x] - a [blockIdx.x * blockDim.x * thieadIdx.x];
 .....
    }
    // Запустить ядро и забать выделяемый (под buf) // объем
разделяемой памяти в байтах. kernel<<<dim3(n/256), dim3(256),
k*sizeof(float)>>> ( a ) ;
```

В приведенном выше примере кода каждому блоку будет дополнительно выделено `k*sizeof(float)` байт разделяемой памяти, которая будет доступна через массив buf. Обратите внимание, что можно задать несколько массивов в разделяемой памяти без явного задания их размера, но тогда в момент выполнения ядра они все будут расположены в начале выделенной блоку дополнительной разделяемой памяти, то есть их начала просто совпадут. В этом случае на программиста ложится ответственность за явное разделение памяти между такими массивами.

```
    __global__ void kernal ( float * a, int k )
    {
    ___ shared  float bufl [];  // Размер явно не указан.
    ___ shared  float buf2 [];  // Размер явно не указан,
    // считаем, что он перебан как k.
    bufl IthreadIdx.x]    = a [blockldx.x * blockDim.x *
threadIdx.x]; buf2 [к + threadldx.x] = a [blockldx.x * blockDim.x
+ threadIdx.x + k];
    }
```

Очень часто разделяемая память используется для хранения постоянно используемых значений - вместо того чтобы постоянно извлекать из медленной глобальной памяти, гораздо удобнее их один раз прочесть в разделяемую память и дальше брать оттуда.

Пример: перемножение матриц

Рассмотрим использование разделяемой памяти на примере перемножения двух квадратных матриц A и B из прошлой главы. Как и ранее, будем использовать двухмерные блоки размера 16x16 и будем считать, что размер матриц N кратен 16. Каждый блок будет вычислять одну 16x16 подматрицу C' искомого произведения.

Как видно по рис. 3.4, для вычисления подматрицы C' произведения AxB нам приходится постоянно обращаться к двух полосам (подматрицам) исходных матриц A' и B'.

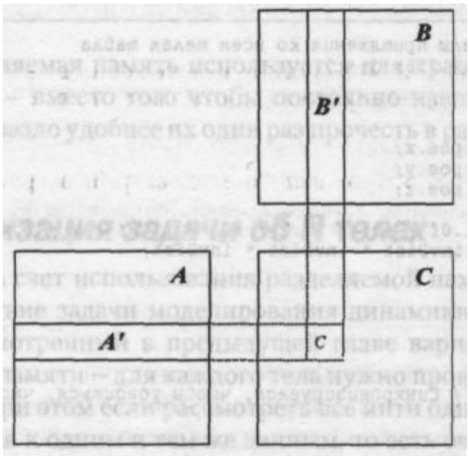

Рис. 3.4. Для вычисления элементов C' нужны только элементы из A' и B'

Обе эти полосы имеют размер $Nx16$, и их элементы многократно используются в расчетах.

Идеальным вариантом было разместить копии этих полос в разделяемой памяти, однако для реальных задач это неприемлемо из-за небольшого объема имеющейся разделяемой памяти (так, если N равно 1024, то одна полоса будет занимать в памяти 1024 x 16 x 4 = 64 Кб).

Однако если каждую из этих полос мы разобьем на квадратные подматрицы 16x16, то становится видно, что результирующая матрица C' просто является суммой попарных произведений подматриц из этих двух полос:

$C' = A'_1 \text{ x } B'_1 + A'_2 \text{ x } B'_2 + \ldots + A'_{N/16} \text{ x } B'_{N/16}$

За счет этого можно выполнить вычисление подматрицы C' всего за $N/16$ шагов. На каждом таком шаге в разделяемую память загружаются одна 16x16 подматрица A и одна подматрица B (при этом нам потребуется 16x16x4x2 = 2 Кбайта разделяемой памяти на блок), при этом каждая нить блока загружает ровно по одному элементу из каждой из этих подматриц, то есть каждая нить делает всего два обращения к глобальной памяти на один шаг.

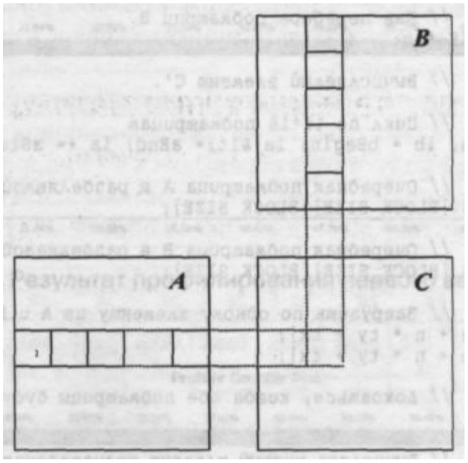

Рис.3.5 Разложение требуемой подматрицы в сумму произведений матриц 16x16

После этого считается произведение подматриц, загруженных в разделяемую память, и суммируется нужный элемент произведения, и идет переход к следующей паре подматриц.

Обратите внимание, что поскольку каждая нить загружает только по одному элементу из каждой из подматриц, а используем все эти элементы, то после загрузки необходимо поставить синхронизацию, чтобы убедиться, что обе подматрицы загружены полностью (а не только 32 элемента, загруженных данным *warp'*ом*).* Точно так же синхронизацию необходимо поставить и после вычисления произведения загруженных подматриц до загрузки следующей пары (чтобы убедиться, что текущие подматрицы уже не нужны никакой нити).

Кроме того, в данном варианте у нас все обращения к глобальной памяти будут *coalesced.*

```
    #define BLOCK_SIZE 16 // Размер блока.
    __global__  void matMult ( float * a, float * b, int n, float
* c ) {
    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = thieadIdx.y;
    // Индекс начала первой подматриины A, обрабатываемой блоком,
int aBegin = n * BLOCK_SIZE * by; int aEnd  = aBegin + n - 1;
    // Шаг перебора подматрицы A. int aStep = BLOCK_SIZE;
    // Индекс первой подматриины B обрабатываемой блоком, int
bBegin = BLOCK_SIZE * bx;
    // Шаг перебора подматрицы B. int bStep = BLOCK_SIZE * n;
```

```
    float sum = 0.0f;     // Вычисляемый элемент C'.
    // Цикл по 16*16 подматрицам
    for ( int ia = aBegin, ib = bBegin; ia &lt; = aEnd; ia +=
aStep, ib += bStep ) {
    // Очереоная подматрииа A в разделяемой памяти.
 __ shared     float as [BLOCK_SIZE][BLOCK_SIZE];
    // Очереоная подматрииа B в разделяемой памяти.
 __ shared     float bs [BLOCK_SIZE][BLOCK_SIZE];
    // Загрузить по одному элементу из A и B в разделяемую
память. as [ty][tx] = a [ia + n * ty + tx]; bs [ty][tx] = b [ib +
n * ty + tx];
    // Дождаться, когда обе подматрицы будут полностью загружены.
 __ syncthreads() ;
    // Вычисляем нужный элемент произведения загруженных
подматриц. for ( int k = 0; k &lt; BLOCK_SIZE; k++ ) sum += as
[ty][k] * bs [k][tx];
    // Дождаться, пока все остальные нити блока закончат
вычислять // свои элементы. syncthreads(); }
    // Записать результат. int ic - n * BLOCK_SIZE * by +
BLOCK_SIZE * bx;
    c [ic + n * ty + tx] = sum; }
```

Таким образом, при вычислении произведения матриц нам на каждый элемент произведения C нужно выполнить всего 2xN/16 чтений из глобальной памяти, в отличие от предыдущего варианта без использования глобальной памяти, где нам требовалось на каждый элемент 2xN чтений. Количество арифметических операций не изменилось и осталось равным 2xN - 1. В табл. 31.6 приведено сравнение быстродействия этой версии и «перемножения в лоб» из предыдущей главы.

Таблица 3.6. Сравнение быстродействия двух вариантов перемножения матриц

Способ перемножения матриц	Затраченное время в миллисекундах
«В лоб» без использования разделяемой памят	2484.06
С использованием разделяемой памяти	133.47

Как видно из приведенной таблицы, быстродействие выросло больше чем на порядок.

В первом случае основное время было затрачено на чтение из глобальной памяти, причем почти все оно было *uncoalesced,* а на вычисления было затрачено менее одной десятой от времени чтения из глобальной памяти.

Во втором случае свыше 80% времени было затрачено на вычисления, доступ к глобальной памяти занял менее 13%, и весь доступ был *coalesced.*

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Барский А.Б. Параллельные процессы в вычислительных системах. Планирование и организация. – М.: Радио и связь, 1990. – 256 с.
2. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002 г.
3. Гергель В.П. Теория и практика параллельных вычислений: учебное пособие/ В.П. Гергель.- М.: Интернет-Университет Информационных технологий; БИНОМ. Лаборатория знаний, 2007. – 423 с.
4. Гладких Б.А. Информатика от абака до Интернета. Введение в специальность: Учебное пособие. – Томск: Изд-во НТЛ, 2005. – 484 с.
5. Головашкин Д.Л. Методы параллельных вычислений. Часть I. Учебное пособие. Изд. СГАУ, Самара. 2002, 92 с.
6. Головашкин Д.Л., Головашкина С.П. Методы параллельных вычислений. Часть II. Учебное пособие. Изд. СГАУ, Самара. 2003. 103 с.
7. Кравчук В.В. и др. Введение в программирование для параллельных ЭВМ и кластеров: Учебн. пособие/ Авторы-сост.: Кравчук В.В., Попов С.Б., Привалов А.Ю., Фурсов В.А., Шустов В.А.; Самар. научный центр РАН, Самар. гос. аэрокосм. ун-т. Самара. 2000. 87 с.
8. Т. Тоффоли, Н. Марголус. Машины клеточных автоматов, М.: Мир,1991 г.
9. Фурсов В.А., Шустов В.А., Скуратов С.А. Технология итерационного планирования распределения ресурсов гетерогенного кластера. Труды Всероссийской научной конференции "Высокопроизводительные вычисления и их приложения".- Труды Всероссийской научной конференции г. Черноголовка , 30 октября - 2 ноября 2000 г. с.43-46.
10. Bruce P. Lester, The Art of Parallel Programming. Prentice Hall, Englewood Cliffs, New Jersey, 1993, 376 p.
11. Computational Science: Ensuring America`s Competitiveness. President`s Information Tehnology Advisory Committee. May 27, 2005.

12. Баканов В.М., Осипов Д.В., Введение в практику разработки параллельных программ в стандарте MPI, М.:МГАПИ, 2005, 63с.
13. Основы работы с технологией OpenMP http://www.microsoft.com/Rus/Msdn/Magazine/2005/10/OpenMP.mspx
14. Тестирование CUDA приложений http://www.ixbt.com/video3/cuda-2.shtml
15. Боресков А.В., Харламов А.А. Основы работы с технологией CUDA, ДМК-Пресс, 2010, 232с.

Учебное издание

Никоноров Артем Владимирович
Фурсов Владимир Алексеевич

РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ РЕШЕНИЯ ЗАДАЧ ВЫСОКОЙ ВЫЧИСЛИТЕЛЬНОЙ СЛОЖНОСТИ В СРЕДАХ MPI, OPEN MP И CUDA

Учебное пособие

Компьютерная верстка _____

Подписано в печать 2010 г. Формат 60×84 1/16.
Бумага офсетная. Печать офсетная.
Печ. л. . Тираж ___ экз. Заказ . ИП /2009

Самарский государственный
аэрокосмический университет.
443086 Самара, Московское шоссе, 34.

Изд-во Самарского государственного
аэрокосмического университета.
443086 Самара, Московское шоссе, 34.