Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное  
учреждение высшего образования   
«Ивановский государственный энергетический университет  
имени В.И. Ленина»

Факультет информатики и вычислительной техники

Кафедра Высокопроизводительных вычислительных систем

**Отчет по курсовой работе**

**по технологиям параллельного программирования**

Системы параллельного программирования

Выполнил:

Прохоров А.П.

Проверила:

Чернышева Л.П.

**Содержание**

[Технологии параллельного программирования PVM 3](#_Toc8387179)

[Технологии параллельного программирования DVM 6](#_Toc8387180)

[Технологии параллельного программирования MPC 9](#_Toc8387181)

[Технологии параллельного программирования LINDA 10](#_Toc8387182)

[Технологии параллельного программирования OpenCL 11](#_Toc8387183)

[Технологии параллельного программирования OpenACC 13](#_Toc8387184)

# Технологии параллельного программирования PVM

PVM (параллельная виртуальная машина) - это побочный продукт продвижения гетерогенных сетевых исследовательских проектов, распространяемый авторами и институтами, в которых они работают. PVM — плод совместного сотрудничества Окриджской национальной лаборатории, Университета штата Теннесси и Университета Эмори. Работа над проектом началась в Окриджской национальной лаборатории летом 1989 года, и в том же году была выпущена PVM 1.0.

PVM представляет собой набор программных средств и библиотек, которые эмулируют общецелевые, гибкие гетерогенные вычислительные структуры для параллелизма во взаимосвязанных компьютерах с различными архитектурами.

Главной же целью системы PVM является обеспечение возможности совместного использования группы компьютеров совместно для взаимосвязанных или параллельных вычислений.

Система PVM состоит из двух частей. Первая часть - это ``демон'' под названием pvmd3 - часто сокращается как pvmd - который помещается на все компьютеры, создающие виртуальную машину. (Примером программы-демона может быть почтовая программа, которая выполняется в фоновом режиме и обрабатывает всю входящую и исходящую электронную почту компьютера). Разработан pvmd3 таким образом, чтобы любой пользователь с достоверным логином мог инсталлировать его на машину. Когда пользователь желает запустить приложение PVM, он прежде всего создает виртуальную машину.

Вторая часть системы - это библиотека подпрограмм интерфейса PVM. Она содержит функционально полный набор примитивов, которые необходимы для взаимодействия между задачами приложения. Эта библиотека содержит вызываемые пользователем подпрограммы для обмена сообщениями, порождения процессов, координирования задач и модификации виртуальной машины.

PVM поддерживает программирование на языках Fortran, C и C++ путём предоставления специальных библиотек.

**Проектирования программ на PVM:**

* алгоритм предварительно приводится к параллельному виду и записывается в виде нескольких ветвей,
* ветви могут между собой обмениваться данными в виде сообщений,
* ветви могут пересекаться и в пересекающихся частях синхронизировать друг с другом свое выполнение,
* все ветви алгоритма сводятся в единую программу (приложение),
* в ОЗУ программа загружается и выполняется в количестве экземпляров, равном количеству ветвей в алгоритме. Эти экземпляры называются задачами (как и в Юниксе),
* задачи объединяются в группу. Каждая группа в PVM объединяет все задачи, работающие над одним алгоритмом, и только их. Возможностей для взаимодействия соседям по группе PVM предоставляет намного больше, чем просто двум задачам,
* каждая задача в зависимости от своего порядкового номера в группе приступает к выполнению той или иной ветви.

Существуют следующие типы межзадачного обмена информацией:

* **блокируемый обмен**, при котором функция "Послать сообщение" возвращает значение (т.е. завершает работу) только после того как получена положительная или отрицательная квитанция от получателя сообщения. Такой алгоритм передачи с ожиданием уведомления о доставке предпочтителен в тех случаях, когда длинное сообщение передается несколькими порциями, а также при обмене командами, последовательность выполнения которых во времени должна быть строго фиксированной.
* **неблокируемый обмен** сообщений уменьшаются простои процессоров, вызванные ожиданием реакции "собеседника". Особенно большой эффект это дает на приемной стороне при неизвестном времени прихода сообщения. Можно организовать работу приемного процессора так, чтобы он в ожидании сообщения выполнял текущую работу, лишь время от времени опрашивая приемный буфер.

**Управление задачами**

1) int tid = pvm\_mytid ( void );

возвращает идентификатор задачи tid >= 0.

2) int numt = pvm\_spawn (char \*task, /\*имя исполняемого файла\*/

char \*\*argv,/\*арг-ты командной строки\*/

int flag, /\*опции запуска\*/

char \*where,/\*указывает место запуска\*/

int ntask, /\*число запускаемых копий\*/

int \*tids /\*массив значений tid для

запущенных задач\*/ );

- запускает в PVM ntask копий исполняемого файла с именем "task" с одинаковыми аргументами командной строки в массиве argv и возвращает число запущенных задач numt а также последовательность идентификаторов для запущенных задач. Причем, если numt < ntask, то в последних ntask - numt элементах массива tids записаны отрицательные коды ошибок, объясняющие срыв запуска задачи.

Замечание 1: исполняемый файл для функции pvm\_spawn() должен находиться в строго определенном каталоге. Задавать имя каталога в параметре "task" недопустимо (или, по крайней мере, нежелательно).

Замечание 2: исполняемый файл ищется (и запускается) не только на том компьютере, на котором работает вызвавшая pvm\_spawn() задача, но в зависимости от параметров flag и where, на любом входящем в состав PVM.

Значением параметра flag задается набор опций для запускаемых задач. Каждой опции соответствует целое неотрицательное число - вес опции, и значение flag равно сумме весов выбранных опций.

PvmTaskDefault ( 0 ) - Pvm выбирает по умолчанию, где запустить задачи;

PvmTaskHost ( 1 ) - задачи запускаются на машине, тип которой указан в параметре where;

PvmTaskArch ( 2 ) - задачи запускаются на комплексе, архитектура которого указана в параметре where;

PvmTaskDebug ( 4 ) - задачи стартуют под отладчиком;

PvmTaskTrace ( 8 ) - при выполнении генерируются результаты трассировки;

PvmMppFront ( 16 ) - задачи стартуют на MPP системе;

PvmHostCompl ( 32 ) - дополнение к информации в параметре where.

**Многомашинность**

1) Программное обеспечение PVM должно быть развернуто на всех машинах

2) Пользователь должен на каждой машине создать свой личный конфигурационный файл PVM

В этом файле указываются:

* имена остальных компьютеров, привлекаемых в состав PVM,
* их производительность, выраженная в неких относительных единицах,
* пароли для входа на эти машины,
* и прочая информация, зачастую не обязательная

Чтобы приложение могло быть запущено на каком-то компьютере, его надо сначала скомпилировать и поместить в каталог $HOME/pvm3/bin/$PVM\_ARCH !

Тогда pvm\_spawn() сможет выбирать наиболее подходящее место выполнения для запускаемой задачи, не ограничиваясь текущей физической машиной.

Стадию переноса можно автоматизировать, воспользовавшись утилитами rdist и rsh.

# Технологии параллельного программирования DVM

DVM-система предназначена для создания переносимых и эффективных вычислительных приложений на языках C-DVM и Fortran-DVM для параллельных компьютеров с различной архитектурой.

Аббревиатура DVM соответствует двум понятиям: Distributed Virtual Memory и Distributed Virtual Machine. Первое отражает наличие единого адресного пространства. Второе отражает использование виртуальных машин для двухступенчатой схемы отображения данных и вычислений на реальную параллельную машину.

В систему DVM также входят библиотека поддержки LIB-DVM, DVM-отладчик, предсказатель выполнения DVM-программ, анализатор производительности DVM-программ.

Система разработана в [Институте прикладной математики им. М.В.Келдыша РАН](https://parallel.ru/russia/organizations.html#ipm).

В DVM-системе программисту предоставляются следующие возможности спецификации параллельного выполнения программы:

* распределение элементов массива между процессорами;
* определение циклов, витки которых могут выполняться параллельно (витки циклов распределяются между процессорами в соответствии с распределением данных);
* спецификация параллельно выполняющихся секций программы (параллельных задач) и отображение их на процессоры;
* организация эффективного доступа к удаленным (расположенным на других процессорах) данным;
* организация эффективного выполнения редукционных операций — глобальных операций с расположенными на различных процессорах данными (таких, как их суммирование или нахождение их максимального или минимального значений);
* спецификация регионов – специальных конструкций языка, состоящих из последовательных

**Состав DVM-системы**

DVM-система состоит из следующих компонент:

* Компилятор Fortran-DVMH
* Компилятор C-DVMH
* Библиотека LIB-DVMH
* Средства функциональной отладки DVMH-программ
* Средства отладки эффективности DVMH-программ

Эти компоненты выполняют следующие функции:

* Компилятор Fortran-DVMH превращает параллельную программу в программу на языке Фортран, использующую стандартные технологии программирования OpenMP и CUDA и расширенную вызовами функций Lib-DVMH.
* Компилятор C-DVMH превращает параллельную программу в программу на языке Си, использующую стандартные технологии программирования OpenMP и CUDA и расширенную вызовами функций Lib-DVMH.
* Библиотека Lib-DVMH — это система поддержки выполнения DVMH-программ (написанных на языках Fortran-DVMH или C-DVMH). Функции Lib-DVMH используют стандартную коммуникационную систему MPI и технологию CUDA.
* Средства функциональной отладки DVMH-программ обеспечивают выполнение программы на рабочей станции в специальном режиме проверки DVMH-указаний, а также выполнение программы на параллельном компьютере в специальном режиме, когда промежуточные результаты выполнения сравниваются с результатами эталонной трассировки (например, результатами последовательного выполнения). Кроме того, для отладки программ, использующих в качестве ускорителей графические процессоры, имеется режим сравнения результатов выполнения региона на ЦПУ и ГПУ.
* Анализатор производительности определяет характеристики эффективности выполнения параллельной программы. Он с заданной степенью детализации предоставляет пользователю информацию о производительности всей DVMH-программы и различных ее фрагментов.

**Использование DVM на кластере**

Необходимо создать поддиректорию, в которой будет находится DVM-программа, и сделать ее текущей. Затем по команде "**dvminit**" надо получить номера установленных версий DVM-системы (сейчас это 315 и 315n) и инициализировать одну из них командой "**dvminit номер\_версии**" (рекомендуется использовать последнюю).

**Трансляция и запуск программ**

Программы на языках C-DVM и Fortran-DVM имеют расширения .cdv и .fdv, соответственно. Оттранслировать DVM-программу можно с помощью команды "**dvm c program.cdv**" или "**dvm f program.fdv**". Если нет ошибок, генерируется исполняемая программа (program), работающая в параллельном режиме поверх MPI.

Запустить программу на выполнение можно командой "**dvm run M N program**". Здесь MxN - размерность решетки процессоров (программа запускается на M\*N процессорах).

**Запуск DVMH-программы**

Попробуем скомпилировать и запустить полученную DVMH-программу. Для этого нам понадобится вычислительная машина, на которой установлена DVM-система

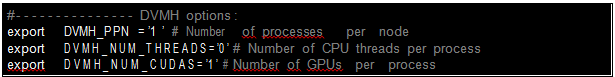
Директорию, где находится наша программа jac2d.cdv, следует скопировать dvm скрипт из директории с установленной DVM-системой.



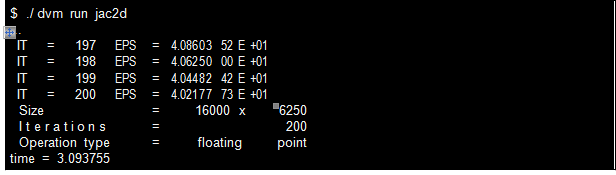
Теперь можно скомпилировать нашу программу.



В текущей директории появился исполняемый файл jac2d, который можно запустить с помощью скрипта dvm, но предварительно надо настроить параметры запуска, отредактировав dvm скрипт.



Задав такие параметры, мы сообщаем, что хотим выполнять регионы только на одном гра-фическом ускорителе. Всё, можем запускать:



# Технологии параллельного программирования MPC

Язык параллельного программирования mpC был разработан в конце 90х в Институте системного программирования РАН. В языке mpC, который является расширением языка С, вводится понятие вычислительного пространства, которое определяется как некоторое доступное для управления множество виртуальных процессоров. Основным понятием mpC является понятие сетевого объекта или просто сети. Сеть является областью вычислительного пространства, которая может быть использована для вычисления выражений и выполнения операторов.

Программист на mpC не может управлять тем, сколько процессов составляют программу и на каких компьютерах эти процессы выполняются. Это делается внешними по отношению к языку средствами. Исходный код на mpC управляет лишь тем, какие именно вычисления выполняются каждым из процессов, составляющих программу.

Основное внимание в языке *mpC* уделяется средствам, позволяющим максимально облегчить разработку как можно более эффективных программ для решения задач на обыкновенных сетях компьютеров.

Библиотечная узловая функция MPC\_Printf языка mpC гарантирует вывод приветствия на терминал пользователя от каждого компьютера, участвующего в процессе параллельной программы.

*#include <mpc.h>*

*int [\*]main()*

*{*

*MPC\_Printf(" Hello, world! \n");*

*}*

**Функции**

Библиотечная узловая функция MPC\_Printf языка mpC гарантирует вывод на терминал пользователя от каждого выполняющего её процесса параллельной программы.

int [\*]main() {

MPC\_Printf("Hello, world.\n");

}

Функция int [\*]MPC\_Global\_barrier(void) синхронизирует работу всех процессов параллельной программы.

Фунция int [net SimpleNet(n) w] MPC\_Barrier( void ) позволяет синхронизировать работу виртуальных процессоров любой сети.

# Технологии параллельного программирования LINDA

Идея построения системы Linda исключительно проста, а потому красива и очень привлекательна. Параллельная программа есть множество параллельных процессов, и каждый процесс работает согласно обычной последовательной программе. Все процессы имеют доступ к общей памяти, единицей хранения в которой является кортеж. Отсюда происходит и специальное название для общей памяти - пространство кортежей. Каждый кортеж это упорядоченная последовательность значений. Например,

( "Hello", 42, 3.14 ),

( "P", 5, FALSE, 97, 1024, 2),

( "worker", 5 ) .

Все процессы работают с пространством кортежей по принципу: поместить кортеж, забрать, скопировать. В отличие от традиционной памяти, процесс может забрать кортеж из пространства кортежей, после чего данный кортеж станет недоступным остальным процессам. В отличие от традиционной памяти, если в пространство кортежей положить два кортежа с одним и тем же именем, то не произойдет привычного для нас "обновления" значения переменной - в пространстве кортежей окажется два кортежа с одним и тем же именем. В отличие от традиционной памяти, изменить кортеж непосредственно в пространстве нельзя. Для изменения значений элементов кортежа, его нужно сначала оттуда изъять, затем процесс, изъявший кортеж, может изменить значения его элементов и вновь добавить измененный кортеж в память. В отличие от других систем программирования, процессы в системе Linda никогда не взаимодействуют друг с другом явно, и все общение всегда идет через пространство кортежей.

Первый элемент кортежа всегда является символьной строкой и выступает в роли имени кортежа. Так первый кортеж предыдущего примера состоит из имени ("Hello"), элемента целого типа (42) и вещественного числа (3.14). Во втором кортеже кроме имени "P" есть элемент целого типа (5), элемент логического типа (FALSE) и три целых числа. Последний кортеж состоит из двух элементов: имени ("worker") и целого числа (5). Количество элементов в кортеже может быть любым.

**Функции Linda**

1. Функция OUT помещает кортеж в пространство кортежей.
2. Функция IN ищет подходящий кортеж в пространстве кортежей, присваивает значения его элементов элементам своего параметра-кортежа и удаляет найденный кортеж из пространства кортежей.
3. Функция READ отличается от функции in лишь тем, что выбранный кортеж не удаляется из пространства кортежей. Все остальное точно так же, как и у функции in. Этой функцией удобно пользоваться в том случае, когда значения переменных менять не нужно, но к ним необходим параллельный доступ из нескольких процессов.
4. Функция EVAL похожа на функцию out. Разница заключается лишь в том, что дополнительным элементом кортежа у eval является функция пользователя. Для вычисления значения этой функции система Linda порождает параллельный процесс, на основе работы которого она формирует кортеж и помещает его в пространство кортежей.

**Синхронизация потоков**

Не имея в системе Linda никаких явных средств для синхронизации процессов, совсем не сложно их смоделировать самому. Предположим, что в некоторой точке нужно выполнить барьерную синхронизацию N процессов. Какой-то один процесс, например, стартовый, заранее помещает в пространство кортеж ("ForBarrier", N). Подходя к точке синхронизации, каждый процесс выполняет следующий фрагмент, который и будет выполнять функции барьера:

in( "ForBarrier", formal Bar);

Bar = Bar - 1;

if( Bar != 0 ) {

out( "ForBarrier", Bar);

read( "Barrier" );

} else

out( "Barrier" );

Если кортеж с именем "ForBarrier" есть в пространстве, то процесс его изымает, в противном случае блокируется до его появления. Анализируя второй элемент данного кортежа, процесс выполняет одно из двух действий. Если есть процессы, которые еще не дошли до данной точки, то он возвращает кортеж в пространство с уменьшенным на единицу вторым элементом и встает на ожидание кортежа "Barrier". В противном случае он сам помещает кортеж "Barrier" в пространство, который для всех является сигналом к продолжению работы.

# Технологии параллельного программирования OpenCL

OpenCL (Open Computing Language ) это спецификация, описывающая технологию параллельного программирования, которая в первую очередь ориентирована на GPGPU. Изначально она была разработана компанией Apple, в последствии для развития спецификаций OpenCL был образована группа разработчиков Khronos Compute , в неё вошли Apple, nVidia, AMD, IBM, Intel, ARM, Motorola и др. Первая версия стандарта была опубликована в конце 2008 года.

В отличии от nVidia CUDA, AMD Stream и т.п., в OpenCL изначально закладывалась мультиплатформенность, т.е. OpenCL программа должна без изменений в коде работать на GPU разных типов (разных производителей). Такая программа без изменений должна работать даже на CPU без GPU, хотя в этом случае она может выполняться существенно медленнее чем на GPU.

**Схема работы с аппаратурой**

OpenCL-программа работает с т.н. **платформами (platform)**. Платформа это программный пакет, который поставляется соответствующим разработчиком аппаратных средств. Например "AMD Accelerated Parallel Processing" или "Intel OpenCL". При этом несколько платформ могут работать одновременно на одной машине.

Каждая платформа включает в себя **ICD (Installable Client Driver)**-- программный интерфейс OpenCL для работы с устройствами, которые эта платформа поддерживает.

В среде Linux список ссылок на ICD, присутствующих в системе, обычно хранится в каталоге */etc/OpenCL/vendors/*, а библиотека *libOpenCL.so* выполняет роль **диспетчера (ICD loader)**, т.е. она направляет вызовы OpenCL функций на устройства, через соответствующие ICD.

В среде Linux список ссылок на ICD, присутствующих в системе, обычно хранится в каталоге */etc/OpenCL/vendors/*, а библиотека *libOpenCL.so* выполняет роль **диспетчера (ICD loader)**, т.е. она направляет вызовы OpenCL функций на устройства, через соответствующие ICD.

**Структура программы**

В OpenCL (аналогично CUDA), программа разделяется на две части: первая часть - управляющая, вторая - вычислительная. В роли управляющего устройства (**host**) выступает центральный процессор (CPU), вычислительное устройство (**device**)

Код, который должен выполняться на device, оформляется специальным способом. Поскольку device могут быть от разных производителей и разных типов, то скомпилировать один бинарник для всех не получится. Решается эта проблема следующим образом:

Текст **ядра (kernel)**, т.е. части программы выполняемой на device, включается в основную часть программы (выполняемой на host) в "чистом" виде т.е. в виде текстовой строки. Этот исходник компилируется средствами OpenCL непосредственно в процессе работы программы (runtime) для выбранного в данный момент вычислительного устройства, это происходит каждый раз при запуске OpenCL-программы.

**Модели памяти и синхронизация процессов**

Модель памяти OpenCL имеет несколько типов. Рассмотрим их подробней : global, local, private.

* Память global - основная память уcтройства, самая большая по размеру (512MB для FX1700) и самая медленная, она является общей для всех тредов.
* Память local - общая память для одной группы тредов (shared в терминах CUDA), этот тип быстрее global но существенно меньше по размеру (16KB для FX1700)
* Память private - память треда, быстрая но маленькая (8KB для FX1700)

# Технологии параллельного программирования OpenACC

В ноябре 2011 года был анонсирован стандарт OpenACC совместное детище суперкомпьютерных гигантов CRAY, CAPS и PGI и лидера рынка графических процессоров NVIDIA. Сам стандарт призван значительно упростить работу программиста и создать высокоуровневую прослойку над уже известными CUDA и OpenCL.

Вот, как прост и неприхотлив в использовании OpenACC, он очень сильно напоминает [OpenMP](https://ru.bmstu.wiki/OpenMP_(Open_Multi-Processing)). Скоро можно будет легко распараллеливать свои задачи на огромных гетерогенных кластерах, почти не имея представления об их архитектуре. К плюсам можно отнести также высокую степень абстракции и кроссплатформенность – сразу после выхода новых архитектур необязательно переписывать весь код, большую часть компилятор сделает за нас.

Плюсов и правда много, но не может же быть все так хорошо. Давайте обратимся к минусам: самое первое, что бросается в глаза – все компиляторы с поддержкой OpenACC стоят денег. Может для научных лабораторий лицензия и не такое уж дорогое удовольствие, но студенты вряд-ли соберутся потратиться на это. Второй минус – производительность: ни один компилятор не сможет оптимизировать код лучше, чем это можно сделать вручную, или с использованием библиотек от [NVIDIA](https://ru.bmstu.wiki/Nvidia_Corporation).

**Директивы**

Основным режимом использования OpenACC являются директивы, точно также как и в [OpenMP](https://ru.wikipedia.org/wiki/OpenMP) 3.x или более раннем [OpenHMPP](https://ru.wikipedia.org/w/index.php?title=OpenHMPP&action=edit&redlink=1),[[18]](https://ru.wikipedia.org/wiki/OpenACC#cite_note-18)[[19]](https://ru.wikipedia.org/wiki/OpenACC#cite_note-19). Библиотека поддержки предоставляет несколько вспомогательных функций, описанных в заголовочных файлах "openacc.h" для C/C++ и "openacc\_lib.h" для Fortran;[[20]](https://ru.wikipedia.org/wiki/OpenACC#cite_note-20).

# pragma acc parallel

# pragma acc kernels

Директива parallel указывает на необходимость распараллеливания. Компилятор, проводя анализ кода, определяет необходимость исполнения различных его частей на GPU, или на хосте.

Директива kernels – аналог parallel, указывает на то, что для каждого нового цикла необходимо создать отдельную \_\_device\_\_ функцию.

Основная директива для определения и копирования данных:

# pragma acc data

Директива, определяющая тип параллелизма в регионах parallel и kernels:

# pragma acc loop

Директива loop предшествует оператору цикла и используется для спецификации его свойств. Современные компиляторы не требуют её явного указания.

Несмотря на всю мощь компилятора, иногда нужно подсказывать, какие данные необходимо передать с хоста на устройство и обратно, а поскольку зачастую копирование выполняется дольше расчетов, нужно заранее продумать, где и как оптимизировать доступ к данным. Все условия передачи данных требуют входные данные, выглядящие следующим образом: a[start:length], где a – массив, или указатель на него, start – номер стартового элемента для копирования, а length –длина региона данных, копируемого на GPU, или с него; start и length указываются в элементах массива (для Fortran есть существенное отличие – вместо length указывается end – конечный элемент). Эти условия можно использовать только с директивами kernels, parallel и data region. Ниже представлены те из них, которые используются наиболее часто:

copy – говорит компилятору скопировать данные на устройство перед выполнением ядра и назад после его завершения.

copyin — указывает, что данные на GPU используются только для чтения, и нет необходимости копировать их обратно на хост.

copyout — данные появятся только в результате выполнения ядра на GPU и никак не зависят от предыдущих значений по этому адресу, их нужно скопировать на хост после выполнения кернела.

create – выделяет в памяти устройства место для данных, не требующих какого-либо копирования, например массив для хранения промежуточных результатов.

present - подсказывает компилятору, что эти данные уже были переданы на устройство ранее. Вызывает ошибку, если данных на GPU нет.

**Функции библиотеки**

Некоторые стандартные функции библиотек, реализующих OpenACC: acc\_get\_num\_devices(), acc\_set\_device\_type(), acc\_get\_device\_type(), acc\_set\_device\_num(), acc\_get\_device\_num(), acc\_async\_test(), acc\_async\_test\_all(), acc\_async\_wait(), acc\_async\_wait\_all(), acc\_init(), acc\_shutdown(), acc\_on\_device(), acc\_malloc(), acc\_free().

**Пример**

Рассмотрим на простом примере как можно ускорить перемножение матриц:

1 #include <openacc.h>

2 #include <stdio.h>

3 #include <stdlib.h>

4 void main() {

5 int n = 100;

6 float a[n][n];

7 float b[n][n];

8 float c[n][n];

9 float elements [n];

10 **for**(int i = 0; i < n; i++)

11 **for** (int j=0; j<n; j++){

12 a[i][j] = i+j;

13 b[i][j] = 100 + 2 \* i;

14 }

15 #pragma acc kernels loop independent

16 **for**(int i = 0; i < n; i++)

17 **for** (int j=0; j < n; j++){

18 **for** (int k=0; k<n; k++)

19 c[i][j]=+a[i][k]\*b[k][j];

20 }

21 free(a); free(b); free(c);

22 } *// main*