## +Программирование на основе MPI. Стандарты MPI. Реализации MPI. Основные понятия (параллельная программа, коммуникаторы, типы данных). Структура MPI-программы. Простейшие операции передачи данных.

**Стандарты MPI**

MPI 1.0 – у1994 год

MPI 1.1

опубликован 12 июня 1995,

первая реализация в 2002 году

MPI 2.0 опубликован 18 июля 1997

MPI 2.1 – начало сентября 2008

MPI 2.2 – 4 сентября 2009

MPI 3.0 – 21 сентября 2012

MPI 3.1 – 4 июня 2015

**Реализации MPI**

Открытые

MPICH - поддержка MPI-3

OpenMPI - поддержка MPI-3.1

MPJ – для Java

Коммерческие

Intel MPI - поддержка MPI-3.1

HP-MPI

MPI/PRO - MPI-2

Microsoft MPI

**Параллельная MPI-программа** – множество одновременно выполняемых процессов. Процессы могут выполняться на разных процессорах, но на одном процессоре может располагаться несколько процессов. Количество процессов и число используемых процессоров задается в момент запуска параллельной программы средствами среды исполнения MPI. Все процессы в программе пронумерованы. Номер процесса называется рангом процесса.

Перед использованием функций MPI нужно подключить библиотеку mpi.h

**#include <mpi.h>**

Начало параллельной области задаётся функцией **MPI\_Init( &argc, &argv);**

В конце должна быть

**MPI\_Finalize();**

Нужно помнить, что у процессов нет доступа к памяти друг друга и всё общение происходит через передачу сообщений.

Для взаимодействия процессов используются коммуникаторы. Для всех процессов программы автоматически создается коммуникатор MPI\_COMM\_WORLD.

#include <mpi.h>

int main( int argc, char \*argv[] )

{

int proc\_num, proc\_rank;

<программный код без использования MPI функций>

MPI\_Init( &argc, &argv);

MPI\_Comm\_size( MPI\_COMM\_WORLD, &proc\_num);

MPI\_Comm\_rank( MPI\_COMM\_WORLD, &proc\_rank);

<программный код с использованием MPI функций>

MPI\_Finalize();

<программный код без использования MPI функций> }

Все функции MPI возвращают код ошибки. Все возвращаемые значения функций MPI приведены в стандарте. Вот основные из них:

MPI\_SUCCESS – успешно выполнена;

MPI\_ERR\_BUFFER – неправильный указатель на буфер;

MPI\_ERR\_COMM – неправильный коммуникатор;

MPI\_ERR\_RANK – неправильный ранг процесса.

**Парные коммуникации**

Все функции передачи сообщений MPI можно разделить на две группы: парные и коллективные. Парные функции используются для передачи сообщений между двумя процессами, а коллективные выполняются одновременно всеми процессами заданного коммуникатора.

В этой лабораторной работе рассмотрим парные коммуникации.

Для отправки сообщения используется функция MPI\_Send:

**int MPI\_Send(void \*buf, int count, MPI\_Datatype type, int dest, int tag, MPI\_Comm comm)**, где

* buf – адрес буфера памяти, в котором располагаются отправляемые данные;
* count – количество элементов данных в сообщении;
* type – тип элементов данных в сообщении;
* dest – ранг процесса, которому отправляется сообщение;
* tag – значение, используемое для идентификации сообщений;
* comm – коммуникатор, в рамках которого выполняется передача данных.

В MPI используются специальные типы данных, такие как

* MPI\_CHAR (char)
* MPI\_INT (int)
* MPI\_FLOAT (float)
* MPI\_DOUBLE (double)
* MPI\_CXX\_BOOL (bool)
* MPI\_CXX\_FLOAT\_COMPLEX (std::complex<float>)

Все доступные типы данных можно посмотреть в нужном стандарте.

Получить отправленное сообщение можно при помощи функции MPI\_Recv: int **MPI\_Recv(void \*buf, int count, MPI\_Datatype type, int source, int tag, MPI\_Comm comm, MPI\_Status \*status)**, где

* buf – адрес буфера памяти, в который принимаются данные;
* count – размер буфера;
* type – тип принимаемых элементов;
* source – ранг процесса, от которого принимается сообщение;
* tag – значение, используемое для идентификации сообщений;
* comm – коммуникатор, в рамках которого выполняется передача данных;
* status – указатель на структуру данных с информацией о результате выполнения операции приема данных.

В случае, если процесс должен получить сообщения от нескольких других, можно указать вместо ранга процесса-отправителя константу MPI\_ANY\_SOURCE. Константа MPI\_ANY\_TAG может заменить тег принимаемого сообщения в случае,

если процессу получателю неважно, с каким тегом отправлено сообщение.

Узнать неизвестные параметры можно через переменную status

* status.MPI\_SOURCE – ранг процесса-отправителя;
* status.MPI\_TAG – тег принятого сообщения.

Количество переданных данных можно узнать при помощи функции MPI\_Get\_count:

**int MPI\_Get\_count(MPI\_Status \* status, MPI\_Datatype type, int \* count)**

int proc\_rank, proc\_num; const int buf\_size = 20; char buf[buf\_size];

MPI\_Status st;

MPI\_Init(&argc,&argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &proc\_rank); MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num); if (proc\_rank == 0) {

for (int i = 1; i < proc\_num; i++) {

MPI\_Recv(buf, buf\_size, MPI\_CHAR, MPI\_ANY\_SOURCE, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &st); printf("%s\n", buf);

}

}

else {

sprintf(buf, "Hello from %d", proc\_rank);

MPI\_Send(buf, buf\_size, MPI\_CHAR, 0, 0,

MPI\_COMM\_WORLD);

}

MPI\_Finalize();

**Одновременное выполнение передачи и приема сообщений**

Функции MPI\_Send и MPI\_Recv являются блокирующими, то есть вызов не возвращает программе управление до тех пор, пока данные не будут скопированы в указанное место (полностью переданы) if(rank == 0) {

MPI\_Send(… 1 …); MPI\_Recv(… 1 …);

} else {

MPI\_Send(… 0 …); MPI\_Recv(… 0 …);

}

Для того, чтобы избежать взаимоблокировки процессов при множественных пересылках сообщений между процессами, можно использовать функции для совмещения передачи и приема сообщений:

int MPI\_Sendrecv(void \*sbuf, int scount, MPI\_Datatype stype, int dest, int stag,

void \*rbuf, int rcount, MPI\_Datatype rtype, int source, int rtag,

MPI\_Comm comm, MPI\_Status \*status);

int MPI\_Sendrecv\_replace(

void \*buf, int count, MPI\_Datatype type,

int dest, int stag, int source, int rtag,

MPI\_Comm comm, MPI\_Status \*status);

**Способы коммуникации**

Способ коммуникации – метод, по которому система обрабатывает сообщения. Способ коммуникации определяется при отправке сообщения.

Существуют 4 способа коммуникации:

* стандартный (standard) – MPI\_Send
* синхронный (synchronous) – MPI\_Ssend
* буферизованный (buffered) – MPI\_Bsend
* по готовности (ready) – MPI\_Rsend

Функция получения не определяет способ коммуникации (всегда MPI\_Recv).

Существуют два источника задержки при передаче сообщения

системная накладка – вызывается копированием данных сообщения из буфера сообщения отправителя и копированием данных из сети в буфер сообщения получателя;

синхронизационная накладка – время, потраченное на ожидание другой задачи.

Если операция приёма не начала выполнение, то поведение не определено

(выдаст ошибку).

На стороне отправителя выделяется дополнительный буфер,

в котором отправляемые данные хранятся до момента их получения

Для использования буферизованного режима передачи должен быть создан и передан MPI буфер памяти для буферизации сообщений int MPI\_Buffer\_attach( void\* buffer, int size )

где

buffer – буфер памяти для буферизации сообщений; ✓ size – размер буфера в байтах.

После завершения работы с буфером он должен быть отключен от MPI при помощи функции int MPI\_Buffer\_detach( void\* buffer, int size )

Пример.

int BUFSIZE = sizeof(int) + MPI\_BSEND\_OVERHEAD;

char \*buf = new char[BUFSIZE] {0};

int buf\_size; //размер буфера

int proc\_rank; //ранг процесса

int num; //число

MPI\_Status st;

MPI\_Init(&argc, &argv); //начало области

MPI MPI\_Comm\_rank(MPI\_COMM\_WORLD, &proc\_rank); //ранг

if (proc\_rank == 0) { //0 процесс

cin >> num;

MPI\_Buffer\_attach(buf, BUFSIZE); //присоединяем буфер

MPI\_Bsend(&num, 1, MPI\_INT, 1, 100, MPI\_COMM\_WORLD);

MPI\_Buffer\_detach(buf, &buf\_size); //отсоединяем

}

if (proc\_rank == 1) { //1 процесс

MPI\_Recv(&num, 1, MPI\_INT, 0, 100, MPI\_COMM\_WORLD, &st);

cout << "Process 1 received number " << num << " from process " << st.MPI\_SOURCE << endl;

}

MPI\_Finalize();

Решение о том, будет ли исходящее сообщение буферизовано, принимает MPI. Посылка может зависеть от условий приёма и размера передаваемого сообщения

**Неблокирующие обмены**

У всех парных операций пересылки есть неблокирующие аналоги, имеющие префикс I (Immediate). Неблокирующие (асинхронные) операции лишь

инициируют процесс передачи или приёма сообщения, управление возвращается сразу. При выполнении данных операций отсутствует ожидание завершения копирования данных в промежуточный буфер или из него. Это позволяет экономить время на передачу данных, но возлагает ответственность на программиста за корректную работу с буфером.

Коммуникационные операции разделяются на две стадии

* инициирование операции;
* проверка завершения операции.

Использование неблокирующих операций повышает безопасность программы с точки зрения возникновения тупиковых ситуаций, а также может увеличить скорость работы программы за счёт совмещения выполнения вычислительных и коммуникационных операций.

Замечание. Многое зависит от реализации, не всегда асинхронные операции поддерживаются аппаратурой и системным окружением.

int MPI\_Isend( void\* buf, int count, MPI\_Datatype type, int dest, int tag, MPI\_Comm comm, MPI\_Request\* request );

Дополнительный параметр request типа MPI\_Request используется для идентификации конкретной неблокирующей операции. Возврат из функции происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере.

MPI\_Irecv( void\* buf, int count, MPI\_Datatype type, int dest, int tag, MPI\_Comm comm, MPI\_Request\* request );

Возврат из процедуры происходит сразу после инициализации процесса приёма без ожидания получения всего сообщения и его записи в буфер.

Операция неблокирующей посылки имеет три дополнительных варианта

MPI\_Issend;

MPI\_Ibsend;

MPI\_Irsend.

Определить момент времени, когда можно повторно использовать буфер, можно при помощи функций MPI\_Wait и MPI\_Test.

int MPI\_Wait(MPI\_Request \*request, MPI\_Status \*status) – блокирует процесс до тех пор, пока асинхронная операция с параметром request не будет завершена

request – дескриптор операции;

status – результат выполнения операции обмена.

После выполнения идентификатор неблокирующей операции request устанавливается в значение MPI\_REQUEST\_NULL.

Другие варианты операции ожидания:

int MPI\_Waitany(int count, MPI\_Request requests[], int \*index, MPI\_Status \*status)

int MPI\_Waitall(int count, MPI\_Request requests[], MPI\_Status statuses[])

int MPI\_Waitsome(int incount, MPI\_Request requests[], int \*outcount, int indices[], MPI\_Status statuses[])

int MPI\_Test(MPI\_Request \*request, int \*flag, MPI\_Status \*status) – проверяет, завершена ли асинхронная операция, ассоциированная с индентификатором request. Не является блокирующей. Инициализирует переменную flag (true/false). Если flag = true, то операция завершена, иначе – продолжает выполняться. Если операция завершена, то будет проинициализирована переменная status. После выполнения идентификатор неблокирующей операции request устанавливается в значение MPI\_REQUEST\_NULL.

Другие варианты операции:

MPI\_Testany;

MPI\_Testall;

MPI\_Testsome.

## +Программирование на основе MPI. Коллективные операции передачи данных.

Как уже отмечалось ранее, под коллективными операциями в MPI понимаются операции над данными, в которых принимают участие все процессы используемого коммуникатора.

Выделение основных видов коллективных операций было выполнено в разделе 3. Часть из коллективных операций уже была рассмотрена в п. 4.2.3 – это операции передачи от одного процесса всем процессам коммуникатора( широковещательная рассылка ) и операции обработки данных, полученных

на одном процессе от всех процессов ( редукция данных).

Рассмотрим далее оставшиеся базовые коллективные операции передачи данных.

Обобщенная операция передачи данных от одного процесса всем процессам (распределениеданных) отличается от широковещательной рассылки тем, что процесс передает процессам различающиеся данные (см. рис. 4.4).Выполнение данной операции может быть обеспечено при помощи функции:

int MPI\_Scatter(void \*sbuf,int scount,MPI\_Datatype stype, void \*rbuf,int rcount,MPI\_Datatype rtype, int root, MPI\_Comm comm),

где - sbuf, scount, stype - параметры передаваемого сообщения (scount определяет количество элементов, передаваемых на каждый процесс),

- rbuf, rcount, rtype - параметры сообщения, принимаемого процессах, - root – ранг процесса, выполняющего рассылку данных,

- comm - коммуникатор, в рамках которого выполняется передача данных.

Операция обобщенной передачи данных от всех процессов одному процессу ( сбор данных) является обратной к процедуре распределения данных (см. рис. 4.5).

Для выполнения этой операции в MPI предназначена функция:

int MPI\_Gather (void \*sbuf,int scount,MPI\_Datatype stype, void \*rbuf,int rcount,MPI\_Datatype rtype, int root, MPI\_Comm comm),

где

-sbuf, scount, stype-параметры передаваемого сообщения,

-rbuf, rcount, rtype-параметры принимаемого сообщения,

-root– ранг процесса, выполняющего сбор данных,

-comm- коммуникатор, в рамках которого выполняется передача данных.

В рассмотренной программе суммирования числовых значений имеющаяся процедура сбора и последующего суммирования данных является примером часто выполняемой коллективной операции передачи данных от всех процессов одному процессу.

В этой операции над собираемыми значениями осуществляется та или иная обработка данных ( для подчеркивания последнего момента данная операция еще именуется операцией редукции данных ). Как и

ранее, реализация операции редукции при

помощи обычных парных операций передачи данных является неэффективной и достаточно трудоемкой. Для наилучшего выполнения действий, связанных с редукцией данных, в MPI предусмотрена функция:

int MPI\_Reduce (void \*sendbuf, void \*recvbuf,int count,MPI\_Datatype type, MPI\_Op op,int root,MPI\_Comm comm),

где

-sendbuf- буфер памяти с отправляемым сообщением,

- recvbuf –буфер памяти для результирующего сообщения ( только для процесса с рангом root),

-count - количество элементов в сообщениях ,

-type – тип элементов сообщений ,

-op - операция, которая должна быть выполнена над данными,

-root - ранг процесса, на котором должен быть получен результат,

-comm - коммуникатор, в рамках которого выполняется операция.

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как стол

Автоматически созданное описание

## +Неграфические вычисления на графических процессорах: предпосылки, история возникновения, основные технологии. Архитектура графических процессоров. Архитектура NVIDIA CUDA. Иерархия памяти в CUDA.

Устройства для превращения персональных компьютеров в маленькие суперкомпьютеры известны довольно давно. Ещё в 80-х годах прошлого века на рынке предлагались так называемые транспьютеры, которые вставлялись в распространенные тогда слоты расширения ISA. Первое время их производительность в соответствующих задачах впечатляла, но затем рост быстродействия универсальных процессоров ускорился, они усилили свои позиции в параллельных вычислениях, и смысла в транспьютерах не осталось. Хотя подобные устройства существуют и сейчас — это разнообразные специализированные ускорители. Но зачастую сфера их применения узка и особого распространения такие ускорители не получили.

Но в последнее время эстафета параллельных вычислений перешла к массовому рынку, так или иначе связанному с трёхмерными играми. Универсальные устройства с многоядерными процессорами для параллельных векторных вычислений, используемых в 3D-графике, достигают высокой пиковой производительности, которая универсальным процессорам не под силу. Конечно, максимальная скорость достигается лишь в ряде удобных задач и имеет некоторые ограничения, но такие устройства уже начали довольно широко применять в сферах, для которых они изначально и не предназначались. Отличным примером такого параллельного процессора является процессор Cell, разработанный альянсом Sony-Toshiba-IBM и применяемый в игровой приставке Sony PlayStation 3, а также и все современные видеокарты от лидеров рынка — компаний Nvidia и AMD.

Компания Nvidia выпустила платформу CUDA — C-подобный язык программирования со своим компилятором и библиотеками для вычислений на GPU. Конечно же, написание оптимального кода для видеочипов совсем не такое простое и эта задача нуждается в длительной ручной работе, но CUDA как раз и раскрывает все возможности и даёт программисту больший контроль над аппаратными возможностями GPU. Важно, что поддержка Nvidia CUDA есть у чипов G8x, G9x и GT2xx, применяемых в видеокартах Geforce серий 8, 9 и 200, которые очень широко распространены. В настоящее время выпущена финальная версия CUDA 2.0, в которой появились некоторые новые возможности, например, поддержка расчётов с двойной точностью. CUDA доступна на 32-битных и 64-битных операционных системах Linux, Windows и MacOS X.

• Вычисления на GPU развивались и развиваются очень быстро.

• Два основных производителя видеочипов, Nvidia и AMD, разработали и анонсировали соответствующие платформы под названием CUDA (Compute Unified Device Architecture) и CTM (Close To Metal или AMD Stream Computing), соответственно.

• В отличие от предыдущих моделей программирования GPU, эти были выполнены с учетом прямого доступа к аппаратным возможностям видеокарт.

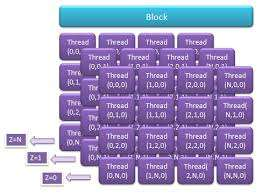
• Платформы не совместимы между собой: CUDA — это расширение языка программирования C, а CTM — виртуальная машина, исполняющая ассемблерный код.



• Внутренняя модель nVidia GPU – ключевой момент в понимании GPGPU с использованием CUDA.

• Верхний уровень ядра GPU состоит из блоков, которые группируются в сетку или грид (grid) размерностью N1 \* N2 \* N3.

• Любой блок в свою очередь состоит из нитей (threads), которые являются непосредственными исполнителями вычислений. Нити в блоке сформированы в виде трехмерного массива, размерность которого так же можно узнать с помощью функции cudaGetDeviceProperties, за это отвечает поле maxThreadsDim.



• Устройство (device) — GPU. Выполняет роль «подчиненного» — делает только то, что ему говорит CPU.

• Хост (host) — CPU. Выполняет управляющую роль — запускает задачи на устройстве, выделяет память на устройстве, перемещает память на/с устройства. И да, использование CUDA предполагает, что как устройство так и хост имеют свою отдельную память.

• Ядро (kernel) — задача, запускаемая хостом на устройстве.

• При использовании технологии CUDA пишется код на своем любимом языке программирования, после чего компилятор CUDA сгенерирует код отдельно для хоста и отдельно для устройства.

• Небольшая оговорка: код для устройства должен быть написан только на языке C с некоторыми 'CUDA-расширениями'.

**Список поддерживаемых языков:**

1. С, С++

2. Common Lisp – cl-cuda

3. Fortran – FORTRAN CUDA, PGI CUDA Fortran Compiler

4. F# – Alea.CUDA

5. Haskell – Data.Array.Accelerate

6. IDL – GPULib

7. Java – jCUDA, JCuda, JCublas, JCufft, CUDA4J

**Основные этапы CUDA-программы**

1. Хост выделяет нужное количество памяти на устройстве.

2. Хост копирует данные из своей памяти в память устройства.

3. Хост стартует выполнение определенных ядер на устройстве.

4. Устройство выполняет ядра.

5. Хост копирует результаты из памяти устройства в свою память.

**Эффективность**

• Естественно, для наибольшей эффективности использования GPU нужно чтобы соотношение времени, потраченного на работу ядер, к времени, потраченному на выделение памяти и перемещение данных, было как можно больше.

**Ядра**

• Важный принцип — ядра пишутся как (практически) обычные последовательные программы. То есть создания и запуска потоков в коде самих ядер нет.

• Вместо этого, для организации параллельных вычислений GPU запустит большое количество копий одного и того же ядра в разных потоках. А точнее, программист определяет сам, сколько потоков запустить.

• Чем больше потоков запускается (при условии что все они будут выполнять полезную работу) — тем лучше.

Код для ядер отличается от обычного последовательного кода в таких моментах:

1. Внутри ядер имеется возможность узнать «идентификатор» или, проще говоря, позицию потока, который сейчас выполняется — используя эту позицию мы добиваемся того, что одно и то же ядро будет работать с разными данными в зависимости от потока, в котором оно запущено.

2. В некоторых случаях в коде ядра необходимо использовать различные способы синхронизации.

**Преимущества CUDA**

• интерфейс программирования приложений CUDA основан на стандартном языке программирования Си с расширениями, что упрощает процесс изучения и внедрения архитектуры CUDA;

• CUDA обеспечивает доступ к разделяемой между потоками памяти размером в 16 Кб на мультипроцессор, которая может быть использована для организации кэша с широкой полосой пропускания, по сравнению с текстурными выборками;

• более эффективная передача данных между системной и видеопамятью

• отсутствие необходимости в графических API с избыточностью и накладными расходами;

• линейная адресация памяти, возможность записи по произвольным адресам;

• аппаратная поддержка целочисленных и битовых операций.

**Основные ограничения CUDA:**

• отсутствие поддержки рекурсии для выполняемых функций;

• минимальная ширина блока в 32 потока;

• закрытая архитектура CUDA, принадлежащая Nvidia.

**Иерархия памяти**

Изображение выглядит как стол

Автоматически созданное описание

1. Регистровая память (register) является самой быстрой из всех видов. Определить количество регистров доступных GPU можно с помощью уже хорошо известной функции cudaGetDeviceProperties. Рассчитать количество регистров, доступных одной нити GPU, также не составляет труда, для этого необходимо разделить общее число регистров на произведение количества нитей в блоке и количества блоков в гриде. Все регистры GPU 32 разрядные. В CUDA нет явных способов использования регистровой памяти, всю работу по размещению данных в регистрах берет на себя компилятор.
2. Локальная память (local memory) может быть использована компилятором при большом количестве локальных переменных в какой-либо функции. По скоростным характеристикам локальная память значительно медленнее, чем регистровая. В документации от nVidia рекомендуется использовать локальную память только в самых необходимых случаях. Явных средств, позволяющих блокировать использование локальной памяти, не предусмотрено, поэтому при падении производительности стоит тщательно проанализировать код и исключить лишние локальные переменные.
3. Глобальная память (global memory) – самый медленный тип памяти, из доступных GPU. Глобальные переменные можно выделить с помощью спецификатора \_\_global\_\_, а также динамически, с помощью функций из семейства cudMallocXXX. Глобальная память в основном служит для хранения больших объемов данных, поступивших на device с host’а, данное перемещение осуществляется с использованием функций cudaMemcpyXXX. В алгоритмах, требующих высокой производительности, количество операций с глобальной памятью необходимо свести к минимуму.
4. Разделяемая память (shared memory) относится к быстрому типу памяти. Разделяемую память рекомендуется использовать для минимизации обращение к глобальной памяти, а также для хранения локальных переменных функций. Адресация разделяемой памяти между нитями потока одинакова в пределах одного блока, что может быть использовано для обмена данными между потоками в пределах одного блока. Для размещения данных в разделяемой памяти используется спецификатор \_\_shared\_\_.
5. Константная память (constant memory) является достаточно быстрой из доступных GPU. Отличительной особенностью константной памяти является возможность записи данных с хоста, но при этом в пределах GPU возможно лишь чтение из этой памяти, что и обуславливает ее название. Для размещения данных в константной памяти предусмотрен спецификатор \_\_constant\_\_. Если необходимо использовать массив в константной памяти, то его размер необходимо указать заранее, так как динамическое выделение в отличие от глобальной памяти в константной не поддерживается. Для записи с хоста в константную память используется функция cudaMemcpyToSymbol, и для копирования с device на хост cudaMemcpyFromSymbol, как видно этот подход несколько отличается от подхода при работе с глобальной памятью.
6. Текстурная память (texture memory), как и следует из названия, предназначена главным образом для работы с текстурами. Текстурная память имеет специфические особенности в адресации, чтении и записи данных. Более подробно о текстурной памяти я расскажу при рассмотрении вопросов обработки изображений на GPU.