|  |  |
| --- | --- |
| Оригинал | Перевод |
| **NVIDIA CUDA™**  *NVIDIA CUDA C Programming Guide*  **CHAPTER 1. INTRODUCTION**  **1.1 From Graphics Processing to General-Purpose Parallel Computing**  Driven by the insatiable market demand for realtime, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth, as illustrated by Figure 1-1.  ***<Figure 1-1 is omitted>***  The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation – exactly what graphics rendering is about – and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by Figure 1-2.  ***<Figure 1-2 is omitted>***  More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations – the same program is executed on many data elements in parallel – with high arithmetic intensity – the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.  Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D rendering, large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. In fact, many algorithms outside the field of image rendering and processing are accelerated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology. | **NVIDIA CUDA™**  *Руководство по программированию для NVIDIA CUDA на C.*  **ГЛАВА 1. ВВЕДЕНИЕ**  **1.1 От обработки графики к параллельным вычислениям общего назначения**  Под влиянием ненасытного рыночного спроса на трехмерную графику высокого разрешения в реальном времени программируемый графический процессор (ГП) развился в высокопараллельный, многопоточный, многоядерный процессор с огромной вычислительной мощностью и очень высокой пропускной способностью памяти, как показано на рисунке 1-1.  ***<Рисунок 1-1 опущен>***  Причина различий ЦП и ГП в способности обработки чисел с плавающей точкой состоит в том, что ГП ориентирован на выполнение интенсивных, высокопараллельных вычислений – как раз в этом и состоит суть рендеринга графики – и поэтому разработан таким образом, что большее число транзисторов предназначено для обработки данных, а не их кэширования и управления потоком выполнения, что показано на рисунке 1-2.  ***<Рисунок 1-2 опущен>***  Выражаясь точнее, ГП особенно хорошо подходит для решения проблем, которые могут быть представлены как параллельная обработка данных: одна и та же программа выполняется на множестве элементов параллельно с высокой арифметической интенсивностью, т.е. отношением числа арифметических операций к числу операций обращения к памяти. Так как одна и та же программа выполняется для каждого элемента данных, снижается потребность в сложном управлении потоком выполнения, и так как она выполняется на множестве элементов данных и имеет высокую арифметическую интенсивность, задержка доступа к памяти может быть скрыта заменой больших кэшей данных вычислениями.  Параллельная обработка данных отображает элементы данных на параллельные вычислительные нити. Многие приложения, обрабатывающие большие наборы данных, могут использовать параллельную относительно данных модель программирования для ускорения вычислений. При 3D-рендеринге большие наборы пикселей и вершин отображаются на параллельные нити. Схожим образом приложения обработки изображений и медиа, такие как постобработка прошедших рендеринг изображений, кодирование и декодирование видео, масштабирование изображений, стереовидение и распознавание образов, могут отображать блоки изображения на параллельные нити обработки. В сущности, многие алгоритмы, не касающиеся рендеринга и обработки изображений, ускоряются с помощью параллельной обработки данных: от стандартной обработки сигналов или физического моделирования до финансового инжиниринга и вычислительной биологии. |
| **1.2 CUDA™: a General-Purpose Parallel Computing Architecture**  In November 2006, NVIDIA introduced CUDA™, a general purpose parallel computing architecture – with a new parallel programming model and instruction set architecture – that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.  CUDA comes with a software environment that allows developers to use C as a high-level programming language. As illustrated by Figure 1-3, other languages, application programming interfaces, or directives-based approaches are supported, such as FORTRAN, DirectCompute, OpenCL, OpenACC.  ***<Figure 1-3 is omitted>*** | **1.2 CUDA™: архитектура параллельных вычислений общего назначения**  В ноябре 2006 года NVIDIA представила CUDA™, архитектуру параллельных вычислений общего назначения с новыми параллельными моделью программирования и архитектурой набора команд, использующую возможности параллельных вычислений ГП NVIDIA для решения многих сложных вычислительных проблем с большей эффективностью, чем ЦП.  CUDA поставляется со средой программирования, позволяющей разработчикам использовать C как язык программирования высокого уровня. Как показано на рисунке 1-3, также поддерживаются и другие языки, интерфейсы программирования приложений и подходы, основанные на использовании директив (например, FORTRAN, DirectCompute, OpenCL, OpenACC).  ***<Рисунок 1-3 опущен>*** |
| **1.3 A Scalable Programming Model**  The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore’s law. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores.  The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C.  At its core are three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization – that are simply exposed to the programmer as a minimal set of language extensions.  These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.  This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors as illustrated by Figure 1-4, and only the runtime system needs to know the physical multiprocessor count.  This scalable programming model allows the CUDA architecture to span a wide market range by simply scaling the number of multiprocessors and memory partitions: from the high-performance enthusiast GeForce GPUs and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs (see Appendix A for a list of all CUDA-enabled GPUs).  ***<Figure 1-4 is omitted>***  A GPU is built around an array of Streaming Multiprocessors (SMs) (see Chapter 4 for more details). A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors. | **1.3 Модель масштабируемого программирования**  Появленияе многоядерных ЦП и ГП означает, что большинство процессорных чипов сейчас представляют собой параллельные системы. Более того – их параллелизм продолжает расти в соответствии с законом Мура. Проблемой является разработка приложений, прозрачно масштабирующих свой параллелизм для использования растущего числа процессорных ядер, подобно тому как приложения, работающие с 3D-графикой, прозрачно масштабируют свой параллелизм на многоядерные ГП со значительно различающимся числом ядер.  Модель параллельного программирования CUDA разработана для решения этой проблемы с сохранением низких требований к дополнительному обучению программистов, знакомых со стандартными языками программирования, такими как С.  В ее основе лежат три ключевых абстракции – иерархия групп нитей, разделяемые области памяти и барьерная синхронизация – которые доступны программисту как минимальный набор расширений языка.  Эти абстракции предоставляют мелкомодульный параллелизм данных и нитей, вложенный в крупномодульный параллелизм данных и задач. Они помогают программисту в разделении задачи на несколько подзадач, которые могут быть решены независимо-параллельно блоками нитей, и каждой подзадачи – на более мелкие, которые могут быть решены совместно-параллельно всеми нитями в блоке.  Эта декомпозиция сохраняет выразительность языка, позволяя нитям сотрудничать при решении каждой подзадачи, и в то же время дает возможность автоматического масштабирования. Действительно, кадый блок нитей может быть назначен для выполнения на любой из доступных в ГП мультипроцессоров в любом порядке, одновременно или последовательно, так что скомпилированная программа на CUDA может выполняться на любом числе мультипроцессоров, как показано на рисунке 1-4, и только среде выполнения необходимо знать число физических мультипроцессоров.  Эта масштабируемая модель программирования позволяет архитектуре CUDA охватить значительную часть рынка простым изменением числа мультипроцессоров и разделов памяти: от высокопроизводительных GeForce для энтузиастов и профессиональных Quadro и Tesla до разнообразных бюджетных GeForce (см. список всех поддерживающих CUDA ГП в Приложении А).  ***<Рисунок 1-4 опущен>***  ГП построен на базе массива потоковых мультипроцессоров (ПМ) (подробнее см. главу 4). Многопоточная программа разделена на блоки нитей, выполняемых независимо, так что ГП с бо́льшим числом мультипроцессоров автоматически выполнит программу быстрее, чем ГП с меньшим числом мультипроцессоров. |
| **1.4 Document’s Structure**  This document is organized into the following chapters:   * Chapter 1 is a general introduction to CUDA. * Chapter 2 outlines the CUDA programming model. * Chapter 3 describes the programming interface. * Chapter 4 describes the hardware implementation. * Chapter 5 gives some guidance on how to achieve maximum performance. * Appendix A lists all CUDA-enabled devices. * Appendix B is a detailed description of all extensions to the C language. * Appendix C lists the mathematical functions supported in CUDA. * Appendix D lists the C++ features supported in device code. * Appendix E gives more details on texture fetching. * Appendix F gives the technical specifications of various devices, as well as more architectural details. * Appendix G introduces the low-level driver API. | **1.4 Структура документа**  Этот документ состоит из следующих глав:   * Глава 1 описывает CUDA в целом. * Глава 2 в общих чертах описывает модель программирования CUDA. * Глава 3 описывает интерфейс программирования (API). * Глава 4 описывает аппаратные реализации. * Глава 5 дает указания по достижению максимальной производительности. * Приложение А содержит список всех поддерживающих CUDA устройств. * Приложение Б дает подробное описание всех расширений языка C. * Приложение В приводит список всех математических функций, поддерживаемых CUDA. * Приложение Г описывает все возможности языка C++, поддерживаемые кодом устройства. * Приложение Д содержит дополнительную информацию по извлечению текстур. * Приложение Е предоставляет технические спецификации различных устройств, а также некоторые подробности архитектуры. * Приложение Ж дает представление о низкоуровневом API драйверов. |
| **CHAPTER 2. PROGRAMMING MODEL**  This chapter introduces the main concepts behind the CUDA programming model by outlining how they are exposed in C. An extensive description of CUDA C is given in Chapter 3.  Full code for the vector addition example used in this chapter and the next can be found in the *vectorAdd* SDK code sample.  **2.1 Kernels**  CUDA C extends C by allowing the programmer to define C functions, called *kernels*, that, when called, are executed N times in parallel by N different *CUDA threads*, as opposed to only once like regular C functions.  A kernel is defined using the **\_\_global\_\_** declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new **<<<…>>>** *execution configuration* syntax (see Appendix B.18). Each thread that executes the kernel is given a unique *thread ID* that is accessible within the kernel through the built-in **threadIdx** variable.  As an illustration, the following sample code adds two vectors *A* and *B* of size *N* and stores the result into vector *C*:  // Kernel definition  \_\_global\_\_ void VecAdd(float\* A, float\* B, float\* C)  {  int i = threadIdx.x;  C[i] = A[i] + B[i];  }  int main()  {  ...  // Kernel invocation with N threads  VecAdd<<<1, N>>>(A, B, C);  ...  }  Here, each of the *N* threads that execute **VecAdd()** performs one pair-wise addition. | **ГЛАВА 2. МОДЕЛЬ ПРОГРАММИРОВАНИЯ**  Эта глава описывает понятия, составляющие основу модели программирования CUDA, указывая их проявления в языке C. Более подробное описание CUDA C дано в главе 3.  Полный исходный код примера сложения векторов, использованного в этой и следующих главах, может быть найден в примере кода *vectorAdd* комплекта средств разработки.  **2.1 Ядра**  CUDA C расширяет C, позволяя разработчикам определять функции, называемые *ядрами*, которые при вызове выполняются N раз параллельно N различными *нитями CUDA*, в отличии от однократного выполнения обычных функций C.  Ядро определяется использованием спецификатора объявления **\_\_global\_\_** и числа нитей CUDA, которые выполняют данное ядро при его вызове с помощью *конфигурации выполнения* new **<<<…>>>** (см. Приложение Б.18). Каждая нить, выполняющий ядро, обладает уникальным *идентификатором нити*, доступным внутри ядра через встроенную переменную **threadIdx.**  В качестве примера рассмотрим код, складывающий два вектора A и B размера N и сохраняющий результат в векторе C:  // Определение ядра  \_\_global\_\_ void VecAdd(float\* A, float\* B, float\* C)  {  int i = threadIdx.x;  C[i] = A[i] + B[i];  }  int main()  {  ...  // Вызов ядра с N нитями  VecAdd<<<1, N>>>(A, B, C);  ...  }  Здесь каждый из N нитей, выполняющих **VecAdd()** производит одно попарное сложение. |
| **2.2 Thread Hierarchy**  For convenience, **threadIdx** is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional *thread index*, forming a one-dimensional, two-dimensional, or three-dimensional *thread block*. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.  The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size *(Dx, Dy)*, the thread ID of a thread of index *(x, y)* is *(x + y Dx)*; for a three-dimensional block of size *(Dx, Dy, Dz)*, the thread ID of a thread of index *(x, y, z)* is *(x + y Dx + z Dx Dy*).  As an example, the following code adds two matrices *A* and *B* of size *NxN* and stores the result into matrix *C*:  // Kernel definition  \_\_global\_\_ void MatAdd(float A[N][N], float B[N][N],  float C[N][N])  {  int i = threadIdx.x;  int j = threadIdx.y;  C[i][j] = A[i][j] + B[i][j];  }  int main()  {  ...  // Kernel invocation with one block of N \* N \* 1 threads  int numBlocks = 1;  dim3 threadsPerBlock(N, N);  MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);  ...  }  There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads.  However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.  Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional *grid* of thread blocks as illustrated by Figure 2-1. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed.  ***<Figure 2-1 is omitted>***  The number of threads per block and the number of blocks per grid specified in the **<<<…>>>** syntax can be of type **int** or **dim3**. Two-dimensional blocks or grids can be specified as in the example above.  Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in **blockIdx** variable. The dimension of the thread block is accessible within the kernel through the built-in **blockDim** variable.  Extending the previous **MatAdd()** example to handle multiple blocks, the code becomes as follows.  // Kernel definition  \_\_global\_\_ void MatAdd(float A[N][N], float B[N][N],  float C[N][N])  {  int i = blockIdx.x \* blockDim.x + threadIdx.x;  int j = blockIdx.y \* blockDim.y + threadIdx.y;  if (i < N && j < N)  C[i][j] = A[i][j] + B[i][j];  }  int main()  {  ...  // Kernel invocation  dim3 threadsPerBlock(16, 16);  dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);  MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);  ...  }  A thread block size of 16x16 (256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before. For simplicity, this example assumes that the number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension, although that need not be the case.  Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores as illustrated by Figure 1-4, enabling programmers to write code that scales with the number of cores.  Threads within a block can cooperate by sharing data through some *shared memory* and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the **\_\_syncthreads()** intrinsic function; **\_\_syncthreads()** acts as a barrier at which all threads in the block must wait before any is allowed to proceed. Section 3.2.3 gives an example of using shared memory.  For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and **\_\_syncthreads()** is expected to be lightweight. | 2.2 Иерархия нитей  Для удобства **threadIdx** состоит из 3 компонент, так что нити могут быть идентифицированы с использованием одно-, двух- или трехмерного *индекса нити*, формирующего одно-, двух- или трехмерный *блок нитей*. Это позволяет естественным образом запрашивать выполнение вычислений у элементов во всей области определения – векторе, матрице или объеме.  Индекс нити и его идентификатор прямо связаны с друг другом: для одномерного блока они одинаковы; для двухмерного блока размера *(Dx, Dy)* идентификатор нити с индексом *(x, y)* будет равен *(x + y Dx)*; для трехмерного блока размера *(Dx, Dy, Dz)* идентификатор нити с индексом *(x, y, z)* будет равен *(x + y Dx + z Dx Dy*).  Например, следующий код складывает две матрицы *A* и *B* размера *NxN* и сохраняет результат в матрице *C*:  // Определение ядра  \_\_global\_\_ void MatAdd(float A[N][N], float B[N][N],  float C[N][N])  {  int i = threadIdx.x;  int j = threadIdx.y;  C[i][j] = A[i][j] + B[i][j];  }  int main()  {  ...  // Вызов ядра с одним блоком N \* N \* 1 нитей  int numBlocks = 1;  dim3 threadsPerBlock(N, N);  MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);  ...  }  Существует ограничение на число нитей в блоке, так как ожидается размещение всех нитей на одном процессорном ядре и разделение ими ограниченных ресурсов памяти этого ядра. На данный момент блок нитей может содержать до 1024 нитей.  Тем не менее, ядро может быть выполнено несколькими одинаково сформированными блоками нитей, так что общее число нитей будет равно числу нитей на блок, помноженному на число блоков.  Блоки организованы в одно-, двух- или трехмерные *сетки* блоков нитей, как показано на рисунке 2-1. Число блоков нитей обычно определяется размером обрабатываемых данных или числом процессоров в системе, которое оно может значительно превосходить.  ***<Рисунок 2-1 опущен>***  Число нитей в блоке и число блоков в сетке, указанное в синтаксисе **<<<…>>>,** может иметь тип **int** или **dim3**. Двухмерные блоки и сетки могут быть объявлены как в примере выше.  Каждый блок сетки может быть идентифицирован с помощью одно-, двух- или трехмерного индекса, доступного в ядре через встроенную переменную **blockIdx**. Размерность блока нитей доступна в ядре через встроенную переменную **blockDim**.  При расширении предыдущего примера **MatAdd()** для возможности работы с несколькими блоками код примет следующий вид.  // Определение ядра  \_\_global\_\_ void MatAdd(float A[N][N], float B[N][N],  float C[N][N])  {  int i = blockIdx.x \* blockDim.x + threadIdx.x;  int j = blockIdx.y \* blockDim.y + threadIdx.y;  if (i < N && j < N)  C[i][j] = A[i][j] + B[i][j];  }  int main()  {  ...  // Вызов ядра  dim3 threadsPerBlock(16, 16);  dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);  MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);  ...  }  Блок нитей размером 16x16 (256 нитей), хотя и может быть выбран произвольно, часто применятся на практике. Сетка создается с числом блоков, достаточным для выделения одной нити на один элемент матрицы, как и раньше. Для простоты этот пример предполагает, что число нитей в сетке в каждой размерности делится нацело на число нитей в блоке в этой размерности, хотя это и не обязательно.  Блоки нитей требуется выполнять независимо: должно быть возможно их выполнение в любом порядке, параллельно или в последовательности. Требование независимости позволяет распределить блоки нитей в любом порядке на любое число ядер, как показано на рисунке 1-4, позволяя программистам писать код, соизмеримый с числом ядер («подстраивающийся» под него).  Нити внутри блока могут работать совместно, разделяя данные с помощью некоторой разделяемой памяти, синхронизируя выполнение для координации доступа к памяти. Конкретно можно задать точки синхронизации в ядре, вызывая внутреннюю функцию **\_\_syncthreads()**; функция **\_\_syncthreads()** действует как барьер, на котором все нити должны ждать, прежде чем любому будет дано разрешение продолжать выполнение. Раздел 3.2.3 содержит пример использования разделяемой памяти.  Для эффективной совместной работы необходимо иметь в качестве разделяемой памяти память, расположенную близко к процессорному ядру и имеющую малую задержку доступа (например, кэш первого уровня), при этом **\_\_syncthreads()** не должна быть ресурсоемкой. |
| **2.3 Memory Hierarchy**  CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 2-2. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.  There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages (see Sections 5.3.2.1, 5.3.2.4, and 5.3.2.5). Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats (see Section 3.2.10).  The global, constant, and texture memory spaces are persistent across kernel launches by the same application.  <Figure 2-2 is omitted> | 2.3 Иерархия памяти  Нити CUDA во время выполнения могут получать доступ к данным из различных областей памяти, как показано на рисунке 2-2. Каждая нить обладает частной местной памятью. Каждый блок нитей обладает разделяемой памятью, доступной всем нитям в блоке, имеющим то же время жизни, что и блок. Все нити имеют доступ к общей глобальной памяти.  Также существуют две дополнительных области памяти, доступные только для чтения всем нитям: постоянная и текстурная. Глобальная, постоянная и текстурная области памяти оптимизированы для различных способов использования памяти (см. разделы 5.3.2.1, 5.3.2.4 и 5.3.2.5). Текстурная память также предлагает различные режимы адресации и фильтрацию данных для некоторых форматов данных (см. раздел 3.2.10).  Глобальная, постоянная и текстурная области памяти неизменны на протяжении запусков ядра одним приложением.  ***<Рисунок 2-2 опущен>*** |
| 2.4 Heterogeneous Programming  As illustrated by Figure 2-3, the CUDA programming model assumes that the CUDA threads execute on a physically separate *device* that operates as a coprocessor to the *host* running the C program. This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU.  The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as *host memory* and *device memory*, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime (described in Chapter 3). This includes device memory allocation and deallocation as well as data transfer between host and device memory.  <Figure 2-3 is omitted> | **2.4 Гетерогенное программирование**  Как показано на рисунке 2-3, модель программирования CUDA предполагает, что нити CUDA выполняются на физически отдельном устройстве, работающем как сопроцессор *хоста*, выполняющего программу на C. То есть, например, ядро выполняется на ГП, а остальная часть программы на C выполняется на ЦП.  Модель программирования CUDA также предполагает, что и хост, и устройство поддерживают отдельные области памяти в ДППД, называемые соответственно *память хоста* и *память устройства*. Таким образом, программа управляет глобальной, постоянной и текстурной областями памяти, доступными всем ядрам через обращение к среде времени выполнения CUDA (описывается в главе 3). Целями обращения могут быть выделение и освобождение памяти, передача данных между памятью хоста и устройства.  <Рисунок 2-3 опущен> |
| 2.5 Compute Capability  The *compute capability* of a device is defined by a major revision number and a minor revision number.  Devices with the same major revision number are of the same core architecture. The major revision number is 3 for devices based on the *Kepler* architecture, 2 for devices based on the *Fermi* architecture, and 1 for devices based on the *Tesla* architecture.  The minor revision number corresponds to an incremental improvement to the core architecture, possibly including new features.  Appendix A lists of all CUDA-enabled devices along with their compute capability. Appendix F gives the technical specifications of each compute capability. | **2.5 Вычислительная способность**  *Вычислительная способность* устройства определяется старшим и младшим номерами ревизии.  Устройства, имеющие одинаковый первичный номер ревизии, имеют одинаковую архитектуру ядра. Первичный номер ревизии равен 3 для устройств, основанных на архитектуре *Kepler*, architecture, 2 – для устройств, основанных на архитектуре *Fermi*, и 1 для устройств, основанных на архитектуре *Tesla*.  Вторичный номер ревизии указывает на постепенные улучшения архитектуры ядра, возможно, включая новые характеристики.  Приложение А приводит список всех устройств, поддерживающих CUDA, а также их вычислительную способность. Приложение Е дает техническую спецификацию каждой вычислительной способности. |
| **CHAPTER 3. PROGRAMMING INTERFACE**  CUDA C provides a simple path for users familiar with the C programming language to easily write programs for execution by the device.  It consists of a minimal set of extensions to the C language and a runtime library.  The core language extensions have been introduced in Chapter 2. They allow programmers to define a kernel as a C function and use some new syntax to specify the grid and block dimension each time the function is called. A complete description of all extensions can be found in Appendix B. Any source file that contains some of these extensions must be compiled with **nvcc** as outlined in Section 3.1.  The runtime is introduced in Section 3.2. It provides C functions that execute on the host to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices, etc. A complete description of the runtime can be found in the CUDA reference manual.  The runtime is built on top of a lower-level C API, the *CUDA driver API*, which is also accessible by the application. The driver API provides an additional level of control by exposing lower-level concepts such as CUDA contexts –the analogue of host processes for the device– and CUDA modules –the analogue of dynamically loaded libraries for the device. Most applications do not use the driver API as they do not need this additional level of control and when using the runtime, context and module management are implicit, resulting in more concise code. The driver API is introduced in Appendix G and fully described in the reference manual. | **ГЛАВА 3. ИНТЕРФЕЙС ПРОГРАММИРОВАНИЯ**  CUDA C обеспечивает простой способ написания программ, исполняемых устройством, для пользователей, знакомых с языком программирования C.  Он состоит из минимального набора расширений языка C и библиотеки времени выполнения.  Основные расширения языка были представлены в главе 2. Они позволяют программистам определять ядро как функцию C использовать новый синтаксис для указания размерностей сетки и блока при каждом вызове функции. Полное описание всех расширений может быть найдено в приложении Б. Любой файл исходного кода, содержащий некоторые из этих расширений, должен быть скомпилирован с использованием **nvcc**, как указано в разделе 3.1.  Среда времени выполнения представлена в разделе 3.2. Она предоставляет функции C, выполняемые на хосте для выделения и освобождения памяти, передачи данных между памятью хоста и устройства, управления системами с несколькими устройствами и т.д. Полное описание среды времени выполнения может быть найдено в справочном руководстве.  Среда времени выполнения построена поверх низкоуровневого API, написанного на C, называемого *API драйверов CUDA*, также доступного приложению. API драйверов предоставляет дополнительный уровень управления, открывая низкоуровневые концепции, такие как контексты CUDA – аналог процесса хоста для устройства – и модули CUDA – аналог динамически загружаемых библиотек для устройства. Большинство приложений не используют API драйверов, так как не нуждаются в этом дополнительном уровне контроля, и при использовании среды времени выполнения контекст и управление модулями скрыты, что делает код более лаконичным. API драйверов представлен в приложении Ж и полностью описан в справочном руководстве. |
| 3.1 Compilationwith NVCC  Kernels can be written using the CUDA instruction set architecture, called *PTX*, which is described in the *PTX* reference manual. It is however usually more effective to use a high-level programming language such as C. In both cases, kernels must be compiled into binary code by **nvcc** to execute on the device.  **nvcc** is a compiler driver that simplifies the process of compiling C or *PTX* code: It provides simple and familiar command line options and executes them by invoking the collection of tools that implement the different compilation stages. This section gives an overview of **nvcc** workflow and command options. A complete description can be found in the **nvcc** user manual. | 3.1 Компиляция с использованием NVCC  Ядра могут быть написаны с использованием архитектуры набора инструкций CUDA, называемой *PTX*, описанной в руководстве по *PTX*. Однако обычно более эффективным является использование языка программирования высокого уровня, например, C. В обоих случаях ядро должно быть скомпилировано в двоичный код с использованием **nvcc** для выполнения на устройстве.  **nvcc** это драйвер компилятора, упрощающий процесс компиляции кода на C или *PTX*: он предоставляет простые и знакомые опции командной строки и выполняет их, обращаясь к коллекции инструментов, реализующих различные стадии компиляции. Этот раздел дает общий обзор процесса выполнения **nvcc** и опций командной строки. Полное описание может быть найдено в руководстве пользователя **nvcc**. |
| 3.1.1 CompilationWorkflow  3.1.1.1 Offline Compilation  Source files compiled with **nvcc** can include a mix of host code (i.e. code that executes on the host) and device code (i.e. code that executes on the device). **nvcc**’s basic workflow consists in separating device code from host code and then:   * compiling the device code into an assembly form (*PTX* code) and/or binary form (*cubin* object), * and modifying the host code by replacing the **<<<…>>>** syntax introduced in Section 2.1 (and described in more details in Section B.18) by the necessary CUDA C runtime function calls to load and launch each compiled kernel from the *PTX* code and/or *cubin* object.   The modified host code is output either as C code that is left to be compiled using another tool or as object code directly by letting **nvcc** invoke the host compiler during the last compilation stage.  Applications can then:   * Either link to the compiled host code, * Or ignore the modifed host code (if any) and use the CUDA driver API (see Appendix G) to load and execute the *PTX* code or *cubin* object. | 3.1.1 Процесс компиляции  3.1.1.1 Offline-компиляция  Файлы исходного кода, скомпилированные **nvcc** могут включать смесь кода хоста (т.е. кода, выполняющегося на хосте) и кода устройства (т.е. кода, выполняющегося на устройстве). Базовый процесс выполнения **nvcc** состоит в разделении кодов хоста и устройства и последующих:   * компиляции кода устройства в виде сборки (код *PTX*) и/или двоичной формы (объект *cubin*), * и изменения кода хоста путем замены синтаксиса **<<<…>>>,** приведенного в разделе 2.1 (и подробно описанного в Б.18) вызовами необходимых функций времени выполнения CUDA C для загрузки и запуска каждого скомпилированного ядра из *PTX* и/или объекта *cubin* .   Измененный код хоста на выходе является либо кодом C, который остается скомпилировать с использованием другого инструмента, либо целевым кодом, если nvcc вызывает компилятор хоста во время последней стадии компиляции.  Приложения затем могут:   * Или создать связи со скомпилированным кодом хоста, * или игнорировать измененный код хоста (при его наличии) и использовать API драйверов CUDA (смотри приложение Ж) для загрузки и исполнения кода *PTX* или объекта *cubin*. |
| 3.1.1.2 Just-in-Time Compilation  Any *PTX* code loaded by an application at runtime is compiled further to binary code by the device driver. This is called *just-in-time compilation*. Just-in-time compilation increases application load time, but allows applications to benefit from latest compiler improvements. It is also the only way for applications to run on devices that did not exist at the time the application was compiled, as detailed in Section 3.1.4.  When the device driver just-in-time compiles some *PTX* code for some application, it automatically caches a copy of the generated binary code in order to avoid repeating the compilation in subsequent invocations of the application. The cache – referred to as *compute cache* – is automatically invalidated when the device driver is upgraded, so that applications can benefit from the improvements in the new just-in-time compiler built into the device driver.  Environment variables are available to control just-in-time compilation:   * Setting CUDA\_CACHE\_DISABLE to 1 disables caching (i.e. no binary code is added to or retrieved from the cache). * CUDA\_CACHE\_MAXSIZE specifies the size of the compute cache in bytes; the default size is 32 MB and the maximum size is 4 GB; binary codes whose size exceeds the cache size are not cached; older binary codes are evicted from the cache to make room for newer binary codes if needed. * CUDA\_CACHE\_PATH specifies the folder where the compute cache files are stored; the default values are: * on Windows, **%APPDATA%\NVIDIA\ComputeCache**, * on MacOS, $HOME/Library/Application\ Support/NVIDIA/ComputeCache, * on Linux, ~/.nv/ComputeCache. * Setting CUDA\_FORCE\_PTX\_JIT to 1 forces the device driver to ignore any binary code embedded in an application (see Section 3.1.4) and to just-in-time compile embedded PTX code instead; if a kernel does not have embedded PTX code, it will fail to load; this environment variable can be used to validate that PTX code is embedded in an application and that its just-in-time compilation works as expected to guarantee application forward compatibility with future architectures. | 3.1.1.2 Компиляция «на лету»  Любой код *PTX,* загруженный приложением во время выполнения далее компилируется в двоичный код драйвером устройства. Это называется *компиляцией «на лету»*. Компиляция «на лету» увеличивает время загрузки приложения, но позволяет приложению получить выгоду от последних улучшений компилятора. Также это единственный способ запустить приложение на устройствах, которые не существовали во время компиляции приложения, как указано в разделе 3.1.4.  Когда драйвер устройства компилирует «на лету» некоторый код *PTX* для некоторого приложения, он автоматически кэширует копию созданного двоичного кода для предотвращения повторной компиляции при последующих вызовах приложения. Кэш, называемый *вычислительный кэш*, автоматически помечается как нерабочий при улучшении драйвера устройства, так что приложения могут получить выгоду от улучшений в новых компиляторах «на лету», встроенных в драйвер устройства.  Следующие переменные среды позволяют управлять компиляцией «на лету»:   * Установка CUDA\_CACHE\_DISABLE в 1 отключает кэширование (т.е. в кэш не записывается и из него не считывается двоичный код). * CUDA\_CACHE\_MAXSIZE указывает размер вычислительного кэша в байтах; размер по умолчанию равен 32 МБ, максимальный – 4 ГБ; двоичный код, размер которого превышает размеры кэша, не кэшируется; более старый код удаляется из кэша для освобождения места для нового, если это необходимо. * CUDA\_CACHE\_PATH указывает директорию, в которой хранится вычислительный кэш; значения по умолчанию таковы: * для Windows, **%APPDATA%\NVIDIA\ComputeCache**, * для MacOS, $HOME/Library/Application\ Support/NVIDIA/ComputeCache, * дляLinux, ~/.nv/ComputeCache. * Установка CUDA\_FORCE\_PTX\_JIT в 1 заставляет драйвер устройства игнорировать встроенный в приложение двоичный код (см. раздел 3.1.4) и вместо этого компилировать «на лету» встроенный код PTX; если ядро не имеет встроенного кода PTX, оно не загрузится; эта переменная среды может быть использована для проверки наличия встроенного в приложение кода PTX и того, что компиляция «на лету» работает как ожидается для обеспечения дальнейшей совместимости приложения с будущими архитектурами. |
| 3.1.2 BinaryCompatibility  Binary code is architecture-specific. A *cubin* object is generated using the compiler option **–code** that specifies the targeted architecture: For example, compiling with **–code=sm\_13** produces binary code for devices of compute capability 1.3. Binary compatibility is guaranteed from one minor revision to the next one, but not from one minor revision to the previous one or across major revisions. In other words, a *cubin* object generated for compute capability X.y is only guaranteed to execute on devices of compute capability X.z where z≥y. | 3.1.2 Двоичная совместимость  Двоичный код зависит от архитектуры. Объект *cubin* создается при использовании опции компилятора **–**code, указывающей целевую архитектуру. Например, компиляция с **–code=sm\_13** производит двоичный код для устройств с вычислительной способностью 1.3. Двоичная совместимость гарантируется при переходе к между вторичными ревизиями вперед, но не назад или между первичными ревизиями. Другими словами, объект *cubin,* созданный для вычислительной мощности X.y гарантированно выполнится только на устройствах с вычислительной способностью X.z, где z≥y. |
| 3.1.3 PTX Compatibility  Some *PTX* instructions are only supported on devices of higher compute capabilities. For example, atomic instructions on global memory are only supported on devices of compute capability 1.1 and above; double-precision instructions are only supported on devices of compute capability 1.3 and above. The **–arch** compiler option specifies the compute capability that is assumed when compiling C to *PTX* code. So, code that contains double-precision arithmetic, for example, must be compiled with “**-arch=sm\_13**” (or higher compute capability), otherwise double-precision arithmetic will get demoted to single-precision arithmetic.  *PTX* code produced for some specific compute capability can always be compiled to binary code of greater or equal compute capability. | 3.1.3 Совместимость с PTX  Некоторые инструкции *PTX* поддерживаются только на устройствах высокой вычислительной способности. Например, атомарные инструкции над глобальной памятью поддерживаются только устройствами с вычислительной способностью 1.1 и выше; инструкции двойной точности поддерживаются только устройствами с вычислительной способностью 1.3 и выше. Опция компилятора **–arch** указывает вычислительную способность, которая предполагается при компиляции из кода C в код *PTX*. Таким образом, код, содержащий, например, арифметику двойной точности, должен быть скомпилирован с указанием “**-arch=sm\_13**” (или более высокой вычислительной способности), иначе арифметика двойной точности будет понижена до арифметики одинарной точности.  Код *PTX*, созданный для определенной вычислительной способности, всегда может быть скомпилирован в двоичный код равной или более высокой вычислительной способности. |
| 3.1.4 Application Compatibility  To execute code on devices of specific compute capability, an application must load binary or *PTX* code that is compatible with this compute capability as described in Sections 3.1.2 and 3.1.3. In particular, to be able to execute code on future architectures with higher compute capability – for which no binary code can be generated yet –, an application must load *PTX* code that will be just-in-time compiled for these devices (see Section 3.1.1.2).  Which *PTX* and binary code gets embedded in a CUDA C application is controlled by the **–arch** and **–code** compiler options or the **–gencode** compiler option as detailed in the **nvcc** user manual. For example,  nvcc x.cu  –gencode arch=compute\_10,code=sm\_10  –gencode arch=compute\_11,code=\’compute\_11,sm\_11\’  embeds binary code compatible with compute capability 1.0 (first **–gencode** option) and *PTX* and binary code compatible with compute capability 1.1 (second **-gencode** option).  Host code is generated to automatically select at runtime the most appropriate code to load and execute, which, in the above example, will be:   * binary code for devices with compute capability 1.0, * binary code for devices with compute capability 1.1, 1.2, 1.3, * binary code obtained by compiling 1.1 *PTX* code for devices with compute capabilities 2.0 and higher.   x.cu can have an optimized code path that uses atomic operations, for example, which are only supported in devices of compute capability 1.1 and higher. The \_\_CUDA\_ARCH\_\_ macro can be used to differentiate various code paths based on compute capability. It is only defined for device code. When compiling with “arch=compute\_11” for example, \_\_CUDA\_ARCH\_\_ is equal to 110.  Applications using the driver API must compile code to separate files and explicitly load and execute the most appropriate file at runtime.  The nvcc user manual lists various shorthands for the –arch, –code, and -gencode compiler options. For example, “-arch=sm\_13” is a shorthand for “-arch=compute\_13 -code=compute\_13,sm\_13” (which is the same as “-gencode -arch=compute\_13,code=\’compute\_13,sm\_13\’”). | 3.1.4 Совместимость приложений  Для выполнения кода на устройствах с определенной вычислительной способностью приложение должно загрузить бинарный код или код *PTX,* совместимый с этой вычислительной способностью, как описано в разделах 3.1.2 и 3.1.3. В частности, для возможности выполнения кода на будущих архитектурах с большей вычислительной способностью – для которых еще нельзя создать двоичный код – приложение должно загружать код *PTX,* который будет скомпилирован «на лету» для этих устройств (см. раздел 3.1.1.2).  То, какой код *PTX* и двоичный код встраивается в приложение на CUDA C, определяется опциями компилятора **–arch и** **–code** или опцией компилятора **–**gencode, как указано в руководстве пользователя **nvcc**. Например,  nvcc x.cu  –gencode arch=compute\_10,code=sm\_10  –gencode arch=compute\_11,code=\’compute\_11,sm\_11\’  встраивает двоичный код, совместимый с вычислительной способностью 1.0 (первая опция **–gencode**) и код *PTX* и двоичный код, совместимые с вычислительной способностью 1.1 (вторая опция **-gencode**).  Код устройства создается таким, чтобы автоматически выбирать во время выполнения для загрузки и исполнения наиболее подходящий код, который для предыдущего примера будет таков:   * двоичный код для устройств с вычислительной способностью 1.0, * двоичный код для устройств с вычислительной способностью 1.1, 1.2, 1.3, * двоичный код, полученный компиляцией кода *PTX* 1.1 для устройств с вычислительной способностью 2.0 или более высокой.   x.cu может иметь оптимизированный код, использующий атомарные операции, которые поддерживаются только в устройствах с вычислительной способностью 1.1 и выше. Макрос \_\_CUDA\_ARCH\_\_ может быть использован для различения кодов, основываясь на вычислительной способности. Он определен только для кода устройства. При компиляции с, например “arch=compute\_11”, \_\_CUDA\_ARCH\_\_ равен 110.  Приложения, использующие API драйверов, должны компилировать код в отдельные файлы и явно загружать и исполнять наиболее подходящий файл во время выполнения.  Руководство пользователя nvcc приводит список различных сокращений для опций компилятора the –arch, –code, и -gencode. Например, “-arch=sm\_13” это сокращение для “-arch=compute\_13 -code=compute\_13,sm\_13” (что эквивалентно “-gencode -arch=compute\_13,-code=\’compute\_13,sm\_13\’”). |
| 3.1.5 C/C++ Compatibility  The front end of the compiler processes CUDA source files according to C++ syntax rules. Full C++ is supported for the host code. However, only a subset of C++ is fully supported for the device code as described in Appendix D. As a consequence of the use of C++ syntax rules, **void** pointers (e.g., returned by **malloc()**) cannot be assigned to non-**void** pointers without a typecast. | 3.1.5 Совместимость с C/C++  Препроцессор компилятора обрабатывает исходный код CUDA в соответствии с синтаксическими правилами C++. C++ поддерживается полностью для кода хоста. Однако лишь подмножество C++ полностью поддерживается для кода устройства, что описано в приложении Г. Как следствие использования синтаксических правил C++, указатель **void** (например, возвращаемый **malloc()**) не может быть назначен не-**void** указателям без приведения типа. |
| 3.1.6 64-Bit Compatibility  The 64-bit version of **nvcc** compiles device code in 64-bit mode (i.e. pointers are 64-bit). Device code compiled in 64-bit mode is only supported with host code compiled in 64-bit mode.  Similarly, the 32-bit version of **nvcc** compiles device code in 32-bit mode and device code compiled in 32-bit mode is only supported with host code compiled in 32-bit mode.  The 32-bit version of **nvcc** can compile device code in 64-bit mode also using the -**m64** compiler option.  The 64-bit version of **nvcc** can compile device code in 32-bit mode also using the -**m32** compiler option. | 3.1.6 Совместимость с 64-битной архитектурой  64-битная версия **nvcc** компилирует код устройства в 64-битном режиме (т.е. указатели являются 64-битными). Код устройства, скомпилированный в 64-битном режиме, поддерживается только вместе с кодом хоста, скомпилированным в 64-битном режиме.  Аналогично 32-битная версия **nvcc** компилирует код устройства в 32-битном режиме, и код устройства, скомпилированный в 32-битном режиме, поддерживается только с кодом хоста, скомпилированным в 32-битном режиме.  32-битная версия **nvcc** может компилировать код устройства в 64-битном режиме, используя опцию компилятора -**m64**.  64-битная версия **nvcc** может компилировать код устройства в 32-битном режиме, используя опцию компилятора -**m32**. |
| 3.2 CUDA C Runtime  The runtime is implemented in the **cudart** dynamic library which is typically included in the application installation package. All its entry points are prefixed with **cuda**.  As mentioned in Section 2.4, the CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. Section 3.2.2 gives an overview of the runtime functions used to manage device memory.  Section 3.2.3 illustrates the use of shared memory, introduced in Section 2.2, to maximize performance.  Section 3.2.4 introduces page-locked host memory that is required to overlap kernel execution with data transfers between host and device memory.  Section 3.2.5 describes the concepts and API used to enable asynchronous concurrent execution at various levels in the system.  Section 3.2.6 shows how the programming model extends to a system with multiple devices attached to the same host.  Section 3.2.8 describe how to properly check the errors generated by the runtime.  Section 3.2.9 mentions the runtime functions used to manage the CUDA C call stack.  Section 3.2.10 presents the texture and surface memory spaces that provide another way to access device memory; they also expose a subset of the GPU texturing hardware.  Section 3.2.11 introduces the various functions the runtime provides to interoperate with the two main graphics APIs, OpenGL and Direct3D. | 3.2 Среда выполнения CUDA C  Среда выполнения CUDA C реализована в динамической библиотеке cudart, которая обычно включается в установочный пакет приложения. Все входные точки имеют префикс **cuda**.  Как упоминается в разделе 2.4, модель программирования CUDA предполагает систему, состоящую из хоста и устройства, каждое со своей отдельной памятью. Раздел 3.2.2 дает общий обзор функций среды выполнения, используемых для управления памятью устройства.  Раздел 3.2.3 демонстрирует использование разделяемой памяти, представленной в разделе 2.2, для максимизации производительности.  Раздел 3.2.4 знакомит со странично-фиксированной памятью хоста, необходимой для перекрывания выполнения ядра передачей данных между памятью хоста и устройства.  Раздел 3.2.5 описывает понятия и API, используемые для обеспечения возможности асинхронного одновременного исполнения на различных уровнях системы.  Раздел 3.2.6 показывает, как модель программирования расширяется для систем с несколькими устройствами, подключенными к одному хосту.  Раздел 3.2.8 описывает, как правильно проверить ошибки, сгенерированные во время выполнения.  Раздел 3.2.9 кратко описывает функции среды выполнения, используемые для управления стеком вызовов CUDA C.  Раздел 3.2.10 представляет области памяти текстур и поверхностей, дающие другой способ доступа к памяти устройства, а также дающие доступ к подмножеству аппаратного обеспечения текстурирования ГП.  Раздел 3.2.11 знакомит с различными функциями, предоставляемыми средой выполнения для взаимодействия с двумя основными графическими API, OpenGL и Direct3D. |
| 3.2.1 Initialization  There is no explicit initialization function for the runtime; it initializes the first time a runtime function is called (more specifically any function other than functions from the device and version management sections of the reference manual). One needs to keep this in mind when timing runtime function calls and when interpreting the error code from the first call into the runtime.  During initialization, the runtime creates a CUDA context for each device in the system (see Section G.1 for more details on CUDA contexts). This context is the *primary context* for this device and it is shared among all the host threads of the application. This all happens under the hood and the runtime does not expose the primary context to the application.  When a host thread calls **cudaDeviceReset()**, this destroys the primary context of the device the host thread currently operates on (i.e. the current device as defined in Section 3.2.6.2). The next runtime function call made by any host thread that has this device as current will create a new primary context for this device. | 3.2.1 Инициализация  Для среды выполнения не существует явной функции инициализации; инициализация происходит во время первого вызова функции времени выполнения (точнее, любой функция, не входящей в множество функций из разделов об управлении устройством и версиями справочного руководства). Необходимо помнить об этом при синхронизации вызовов функций времени выполнения и интерпретации кодов ошибок при первом вызове во время выполнения.  Во время инициализации среда выполнения создает контекст CUDA для каждого устройства в системе (см. раздел Ж.1 для получения дополнительной информации по контекстам CUDA). Этот контекст – *первичный контекст* для этого устройства и разделяется между всеми нитями хоста приложения. Все это происходит скрыто, и среда выполнения не раскрывает основной контекст приложению.  Когда поток хоста вызывает функцию **cudaDeviceReset()**, происходит уничтожение первичного контекста устройства, на котором в данный момент выполняется поток хоста (т.е. текущего устройства, как определяется в разделе 3.2.6.2). Следующий вызов функции времени выполнения любой нитью хоста, для которого это устройство является текущим, создаст новый первичный контекст для этого устройства. |
| 3.2.2 Device Memory  As mentioned in Section 2.4, the CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. Kernels can only operate out of device memory, so the runtime provides functions to allocate, deallocate, and copy device memory, as well as transfer data between host memory and device memory.  Device memory can be allocated either as *linear memory* or as *CUDA arrays*.  CUDA arrays are opaque memory layouts optimized for texture fetching. They are described in Section 3.2.10.  Linear memory exists on the device in a 32-bit address space for devices of compute capability 1.x and 40-bit address space of devices of higher compute capability, so separately allocated entities can reference one another via pointers, for example, in a binary tree.  Linear memory is typically allocated using **cudaMalloc()** and freed using **cudaFree()** and data transfer between host memory and device memory are typically done using **cudaMemcpy()**. In the vector addition code sample of Section 2.1, the vectors need to be copied from host memory to device memory:  // Device code  \_\_global\_\_ void VecAdd(float\* A, float\* B, float\* C, int N)  {  int i = blockDim.x \* blockIdx.x + threadIdx.x;  if (i < N)  C[i] = A[i] + B[i];  }  // Host code  int main()  {  int N = ...;  size\_t size = N \* sizeof(float);  // Allocate input vectors h\_A and h\_B in host memory  float\* h\_A = (float\*)malloc(size);  float\* h\_B = (float\*)malloc(size);  // Initialize input vectors  ...  // Allocate vectors in device memory  float\* d\_A;  cudaMalloc(&d\_A, size);  float\* d\_B;  cudaMalloc(&d\_B, size);  float\* d\_C;  cudaMalloc(&d\_C, size);  // Copy vectors from host memory to device memory  cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice);  cudaMemcpy(d\_B, h\_B, size, cudaMemcpyHostToDevice);  // Invoke kernel  int threadsPerBlock = 256;  int blocksPerGrid =  (N + threadsPerBlock – 1) / threadsPerBlock;  VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d\_A, d\_B, d\_C, N);  // Copy result from device memory to host memory  // h\_C contains the result in host memory  cudaMemcpy(h\_C, d\_C, size, cudaMemcpyDeviceToHost);  // Free device memory  cudaFree(d\_A);  cudaFree(d\_B);  cudaFree(d\_C);  // Free host memory  ...  }  Linear memory can also be allocated through **cudaMallocPitch()** and **cudaMalloc3D()**. These functions are recommended for allocations of 2D or 3D arrays as it makes sure that the allocation is appropriately padded to meet the alignment requirements described in Section 5.3.2.1, therefore ensuring best performance when accessing the row addresses or performing copies between 2D arrays and other regions of device memory (using the **cudaMemcpy2D()** and **cudaMemcpy3D()** functions). The returned pitch (or stride) must be used to access array elements. The following code sample allocates a **width×height** 2D array of floating-point values and shows how to loop over the array elements in device code:  // Host code  int width = 64, height = 64;  float\* devPtr;  size\_t pitch;  cudaMallocPitch(&devPtr, &pitch,  width \* sizeof(float), height);  MyKernel<<<100, 512>>>(devPtr, pitch, width, height);  // Device code  \_\_global\_\_ void MyKernel(float\* devPtr,  size\_t pitch, int width, int height)  {  for (int r = 0; r < height; ++r) {  float\* row = (float\*)((char\*)devPtr + r \* pitch);  for (int c = 0; c < width; ++c) {  float element = row[c];  }  }  }  The following code sample allocates a **width×height×depth** 3D array of floating-point values and shows how to loop over the array elements in device code:  // Host code  int width = 64, height = 64, depth = 64;  cudaExtent extent = make\_cudaExtent(width \* sizeof(float),  height, depth);  cudaPitchedPtr devPitchedPtr;  cudaMalloc3D(&devPitchedPtr, extent);  MyKernel<<<100, 512>>>(devPitchedPtr, width, height, depth);  // Device code  \_\_global\_\_ void MyKernel(cudaPitchedPtr devPitchedPtr,  int width, int height, int depth)  {  char\* devPtr = devPitchedPtr.ptr;  size\_t pitch = devPitchedPtr.pitch;  size\_t slicePitch = pitch \* height;  for (int z = 0; z < depth; ++z) {  char\* slice = devPtr + z \* slicePitch;  for (int y = 0; y < height; ++y) {  float\* row = (float\*)(slice + y \* pitch);  for (int x = 0; x < width; ++x) {  float element = row[x];  }  }  }  }  The reference manual lists all the various functions used to copy memory between linear memory allocated with **cudaMalloc()**, linear memory allocated with **cudaMallocPitch()** or **cudaMalloc3D()**, CUDA arrays, and memory allocated for variables declared in global or constant memory space.  The following code sample illustrates various ways of accessing global variables via the runtime API:  \_\_constant\_\_ float constData[256];  float data[256];  cudaMemcpyToSymbol(constData, data, sizeof(data));  cudaMemcpyFromSymbol(data, constData, sizeof(data));  \_\_device\_\_ float devData;  float value = 3.14f;  cudaMemcpyToSymbol(devData, &value, sizeof(float));  \_\_device\_\_ float\* devPointer;  float\* ptr;  cudaMalloc(&ptr, 256 \* sizeof(float));  cudaMemcpyToSymbol(devPointer, &ptr, sizeof(ptr));  **cudaGetSymbolAddress()** is used to retrieve the address pointing to the memory allocated for a variable declared in global memory space. The size of the allocated memory is obtained through **cudaGetSymbolSize()**. | 3.2.2 Память устройства  Как упоминается в разделе 2.4, модель программирования CUDA предполагает, что система состоит из хоста и устройства, каждое со своей отдельной памятью. Ядра могут работать только из памяти устройства, так что среда выполнения предоставляет функции для выделения, освобождения и копирования памяти устройства, а также передачи данных между памятью хоста и устройства.  Память устройства может быть выделена или как *линейная память*, или как *массивы CUDA*.  Массивы CUDA это непрозрачные схемы памяти, оптимизированные для извлечения текстур. Они описаны в разделе 3.2.10.  Линейная память существует на устройстве в 32-битном адресном пространстве для устройств с вычислительной способностью 1.x и в 40-битном адресном пространстве для устройств с более высокой вычислительной способностью, так что отдельно размещенные в памяти сущности могут указывать друг на друга с помощью указателей, например, в бинарном дереве.  Линейная память обычно выделяется с использованием **cudaMalloc()** и освобождается с использованием **cudaFree()**, передача данных между памятью хоста и устройства обычно выполняется при помощи **cudaMemcpy()**. В примере кода для векторного сложения в разделе 2.1 векторы необходимо копировать из памяти хоста в память устройства:  // Код устройства  \_\_global\_\_ void VecAdd(float\* A, float\* B, float\* C, int N)  {  int i = blockDim.x \* blockIdx.x + threadIdx.x;  if (i < N)  C[i] = A[i] + B[i];  }  // Код хоста  int main()  {  int N = ...;  size\_t size = N \* sizeof(float);  // Выделяем память хоста для входных векторов h\_A и h\_B  float\* h\_A = (float\*)malloc(size);  float\* h\_B = (float\*)malloc(size);  // Инициализируем входные векторы  ...  // Размещаем векторы в памяти устройства  float\* d\_A;  cudaMalloc(&d\_A, size);  float\* d\_B;  cudaMalloc(&d\_B, size);  float\* d\_C;  cudaMalloc(&d\_C, size);  // Копируем векторы из памяти хоста в память устройства  cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice);  cudaMemcpy(d\_B, h\_B, size, cudaMemcpyHostToDevice);  // Вызываем ядро  int threadsPerBlock = 256;  int blocksPerGrid =  (N + threadsPerBlock – 1) / threadsPerBlock;  VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d\_A, d\_B, d\_C, N);  // Копирум результат из памяти устройства в память хоста  // h\_C содержит результат в памяти хоста  cudaMemcpy(h\_C, d\_C, size, cudaMemcpyDeviceToHost);  // Освобождаем память устройства  cudaFree(d\_A);  cudaFree(d\_B);  cudaFree(d\_C);  // Освобождаем память хоста  ...  }  Линейная память также может быть выделена с использованием **cudaMallocPitch()** и **cudaMalloc3D()**. Эти функции рекомендуется использовать для размещения двух- и трехмерных массивов, так как выполняется проверка того, что выделение подходящим образом дополнено для выполнения требований по размещению, описанных в разделе 5.3.2.1, что позволяет обеспечить лучшую производительность при доступе к адресам строк или выполнении копирования между двухмерными массивами и другими областями памяти устройства (с использованием функций **cudaMemcpy2D()** и **cudaMemcpy3D()**). Возвращенный питч (шаг) нужно использовать для доступа к элементам массива. Следующий пример кода размещает двухмерный массив чисел с плавающей точкой размером **width×height** и показывает, как пройти в цикле по элементам массива в коде устройства:  // Код хоста  int width = 64, height = 64;  float\* devPtr;  size\_t pitch;  cudaMallocPitch(&devPtr, &pitch,  width \* sizeof(float), height);  MyKernel<<<100, 512>>>(devPtr, pitch, width, height);  // Код устройства  \_\_global\_\_ void MyKernel(float\* devPtr,  size\_t pitch, int width, int height)  {  for (int r = 0; r < height; ++r) {  float\* row = (float\*)((char\*)devPtr + r \* pitch);  for (int c = 0; c < width; ++c) {  float element = row[c];  }  }  }  Следующий пример кода размещает трехмерный массив чисел с плавающей точкой размера **width×height×depth** и показывает, как пройти в цикле по элементам массива в коде:  // Код хоста  int width = 64, height = 64, depth = 64;  cudaExtent extent = make\_cudaExtent(width \* sizeof(float),  height, depth);  cudaPitchedPtr devPitchedPtr;  cudaMalloc3D(&devPitchedPtr, extent);  MyKernel<<<100, 512>>>(devPitchedPtr, width, height, depth);  // Код устройства  \_\_global\_\_ void MyKernel(cudaPitchedPtr devPitchedPtr,  int width, int height, int depth)  {  char\* devPtr = devPitchedPtr.ptr;  size\_t pitch = devPitchedPtr.pitch;  size\_t slicePitch = pitch \* height;  for (int z = 0; z < depth; ++z) {  char\* slice = devPtr + z \* slicePitch;  for (int y = 0; y < height; ++y) {  float\* row = (float\*)(slice + y \* pitch);  for (int x = 0; x < width; ++x) {  float element = row[x];  }  }  }  }  Справочное руководство приводит список всех функций, используемых для копирования памяти между памятью, выделенной с помощью **cudaMalloc()**, линейной памяти, выделенной с помощью **cudaMallocPitch()** или **cudaMalloc3D()**, массивами CUDA и памятью, выделенной для переменных, объявленных в глобальной или постоянной области памяти.  Следующий пример кода демонстрирует различные способы доступа к глобальным переменным с использованием API времени выполнения:  \_\_constant\_\_ float constData[256];  float data[256];  cudaMemcpyToSymbol(constData, data, sizeof(data));  cudaMemcpyFromSymbol(data, constData, sizeof(data));  \_\_device\_\_ float devData;  float value = 3.14f;  cudaMemcpyToSymbol(devData, &value, sizeof(float));  \_\_device\_\_ float\* devPointer;  float\* ptr;  cudaMalloc(&ptr, 256 \* sizeof(float));  cudaMemcpyToSymbol(devPointer, &ptr, sizeof(ptr));  **cudaGetSymbolAddress()** используется для получения адреса, указывающего на память, выделенную для переменной, объявленной в глобальной области памяти. Размер выделенной памяти получается через **cudaGetSymbolSize()**. |
| 3.2.3 Shared Memory  As detailed in Section B.2 shared memory is allocated using the **\_\_shared\_\_** qualifier.  Shared memory is expected to be much faster than global memory as mentioned in Section 2.2 and detailed in Section 5.3.2.3. Any opportunity to replace global memory accesses by shared memory accesses should therefore be exploited as illustrated by the following matrix multiplication example.  The following code sample is a straightforward implementation of matrix multiplication that does not take advantage of shared memory. Each thread reads one row of *A* and one column of *B* and computes the corresponding element of *C* as illustrated in Figure 3-1. *A* is therefore read *B.width* times from global memory and *B* is read *A.height* times.  // Matrices are stored in row-major order:  // M(row, col) = \*(M.elements + row \* M.width + col)  typedef struct {  int width;  int height;  float\* elements;  } Matrix;  // Thread block size  #define BLOCK\_SIZE 16  // Forward declaration of the matrix multiplication kernel  \_\_global\_\_ void MatMulKernel(const Matrix, const Matrix, Matrix);  // Matrix multiplication - Host code  // Matrix dimensions are assumed to be multiples of BLOCK\_SIZE  void MatMul(const Matrix A, const Matrix B, Matrix C)  {  // Load A and B to device memory  Matrix d\_A;  d\_A.width = A.width; d\_A.height = A.height;  size\_t size = A.width \* A.height \* sizeof(float);  cudaMalloc(&d\_A.elements, size);  cudaMemcpy(d\_A.elements, A.elements, size,  cudaMemcpyHostToDevice);  Matrix d\_B;  d\_B.width = B.width; d\_B.height = B.height;  size = B.width \* B.height \* sizeof(float);  cudaMalloc(&d\_B.elements, size);  cudaMemcpy(d\_B.elements, B.elements, size,  cudaMemcpyHostToDevice);  // Allocate C in device memory  Matrix d\_C;  d\_C.width = C.width; d\_C.height = C.height;  size = C.width \* C.height \* sizeof(float);  cudaMalloc(&d\_C.elements, size);  // Invoke kernel  dim3 dimBlock(BLOCK\_SIZE, BLOCK\_SIZE);  dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);  MatMulKernel<<<dimGrid, dimBlock>>>(d\_A, d\_B, d\_C);  // Read C from device memory  cudaMemcpy(C.elements, Cd.elements, size,  cudaMemcpyDeviceToHost);  // Free device memory  cudaFree(d\_A.elements);  cudaFree(d\_B.elements);  cudaFree(d\_C.elements);  }  // Matrix multiplication kernel called by MatMul()  \_\_global\_\_ void MatMulKernel(Matrix A, Matrix B, Matrix C)  {  // Each thread computes one element of C  // by accumulating results into Cvalue  float Cvalue = 0;  int row = blockIdx.y \* blockDim.y + threadIdx.y;  int col = blockIdx.x \* blockDim.x + threadIdx.x;  for (int e = 0; e < A.width; ++e)  Cvalue += A.elements[row \* A.width + e]  \* B.elements[e \* B.width + col];  C.elements[row \* C.width + col] = Cvalue;  }  <Figure 3-1 is omitted>  The following code sample is an implementation of matrix multiplication that does take advantage of shared memory. In this implementation, each thread block is responsible for computing one square sub-matrix *Csub* of *C* and each thread within the block is responsible for computing one element of *Csub*. As illustrated in Figure 3-2, *Csub* is equal to the product of two rectangular matrices: the sub-matrix of *A* of dimension *(A.width, block\_size)* that has the same row indices as *Csub*, and the sub-matrix of *B* of dimension *(block\_size, A.width)* that has the same column indices as *Csub*. In order to fit into the device’s resources, these two rectangular matrices are divided into as many square matrices of dimension *block\_size* as necessary and *Csub* is computed as the sum of the products of these square matrices. Each of these products is performed by first loading the two corresponding square matrices from global memory to shared memory with one thread loading one element of each matrix, and then by having each thread compute one element of the product. Each thread accumulates the result of each of these products into a register and once done writes the result to global memory.  By blocking the computation this way, we take advantage of fast shared memory and save a lot of global memory bandwidth since *A* is only read *(B.width / block\_size)* times from global memory and *B* is read *(A.height / block\_size)* times.  The *Matrix* type from the previous code sample is augmented with a *stride* field, so that sub-matrices can be efficiently represented with the same type. **\_\_device\_\_** functions (see Section B.1.1) are used to get and set elements and build any sub-matrix from a matrix.  // Matrices are stored in row-major order:  // M(row, col) = \*(M.elements + row \* M.stride + col)  typedef struct {  int width;  int height;  int stride;  float\* elements;  } Matrix;  // Get a matrix element  \_\_device\_\_ float GetElement(const Matrix A, int row, int col)  {  return A.elements[row \* A.stride + col];  }  // Set a matrix element  \_\_device\_\_ void SetElement(Matrix A, int row, int col,  float value)  {  A.elements[row \* A.stride + col] = value;  }  // Get the BLOCK\_SIZExBLOCK\_SIZE sub-matrix Asub of A that is  // located col sub-matrices to the right and row sub-matrices down  // from the upper-left corner of A  \_\_device\_\_ Matrix GetSubMatrix(Matrix A, int row, int col)  {  Matrix Asub;  Asub.width = BLOCK\_SIZE;  Asub.height = BLOCK\_SIZE;  Asub.stride = A.stride;  Asub.elements = &A.elements[A.stride \* BLOCK\_SIZE \* row  + BLOCK\_SIZE \* col];  return Asub;  }  // Thread block size  #define BLOCK\_SIZE 16  // Forward declaration of the matrix multiplication kernel  \_\_global\_\_ void MatMulKernel(const Matrix, const Matrix, Matrix);  // Matrix multiplication - Host code  // Matrix dimensions are assumed to be multiples of BLOCK\_SIZE  void MatMul(const Matrix A, const Matrix B, Matrix C)  {  // Load A and B to device memory  Matrix d\_A;  d\_A.width = d\_A.stride = A.width; d\_A.height = A.height;  size\_t size = A.width \* A.height \* sizeof(float);  cudaMalloc(&d\_A.elements, size);  cudaMemcpy(d\_A.elements, A.elements, size,  cudaMemcpyHostToDevice);  Matrix d\_B;  d\_B.width = d\_B.stride = B.width; d\_B.height = B.height;  size = B.width \* B.height \* sizeof(float);  cudaMalloc(&d\_B.elements, size);  cudaMemcpy(d\_B.elements, B.elements, size,  cudaMemcpyHostToDevice);  // Allocate C in device memory  Matrix d\_C;  d\_C.width = d\_C.stride = C.width; d\_C.height = C.height;  size = C.width \* C.height \* sizeof(float);  cudaMalloc(&d\_C.elements, size);  // Invoke kernel  dim3 dimBlock(BLOCK\_SIZE, BLOCK\_SIZE);  dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);  MatMulKernel<<<dimGrid, dimBlock>>>(d\_A, d\_B, d\_C);  // Read C from device memory  cudaMemcpy(C.elements, d\_C.elements, size,  cudaMemcpyDeviceToHost);  // Free device memory  cudaFree(d\_A.elements);  cudaFree(d\_B.elements);  cudaFree(d\_C.elements);  }  // Matrix multiplication kernel called by MatMul()  \_\_global\_\_ void MatMulKernel(Matrix A, Matrix B, Matrix C)  {  // Block row and column  int blockRow = blockIdx.y;  int blockCol = blockIdx.x;  // Each thread block computes one sub-matrix Csub of C  Matrix Csub = GetSubMatrix(C, blockRow, blockCol);  // Each thread computes one element of Csub  // by accumulating results into Cvalue  float Cvalue = 0;  // Thread row and column within Csub  int row = threadIdx.y;  int col = threadIdx.x;  // Loop over all the sub-matrices of A and B that are  // required to compute Csub  // Multiply each pair of sub-matrices together  // and accumulate the results  for (int m = 0; m < (A.width / BLOCK\_SIZE); ++m) {  // Get sub-matrix Asub of A  Matrix Asub = GetSubMatrix(A, blockRow, m);  // Get sub-matrix Bsub of B  Matrix Bsub = GetSubMatrix(B, m, blockCol);  // Shared memory used to store Asub and Bsub respectively  \_\_shared\_\_ float As[BLOCK\_SIZE][BLOCK\_SIZE];  \_\_shared\_\_ float Bs[BLOCK\_SIZE][BLOCK\_SIZE];  // Load Asub and Bsub from device memory to shared memory  // Each thread loads one element of each sub-matrix  As[row][col] = GetElement(Asub, row, col);  Bs[row][col] = GetElement(Bsub, row, col);  // Synchronize to make sure the sub-matrices are loaded  // before starting the computation  \_\_syncthreads();  // Multiply Asub and Bsub together  for (int e = 0; e < BLOCK\_SIZE; ++e)  Cvalue += As[row][e] \* Bs[e][col];  // Synchronize to make sure that the preceding  // computation is done before loading two new  // sub-matrices of A and B in the next iteration  \_\_syncthreads();  }  // Write Csub to device memory  // Each thread writes one element  SetElement(Csub, row, col, Cvalue);  }  <Figure 3-2 is omitted> | 3.2.3 Разделяемая память  Как подробно описано в разделе Б.2, разделяемая память выделяется с использованием квалификатора **\_\_shared\_\_**.  От разделяемой памяти ожидается намного более быстрая работа, чем от глобальной, что упомянуто в разделе 2.2 и детально рассмотрено в разделе 5.3.2.3. Любая возможность заменить доступ к глобальной памяти доступом к разделяемой должна быть использована, как, например, показано в следующем коде перемножения матриц.  Следующий пример кода это простая реализация перемножения матриц, не использующая преимуществ разделяемой памяти. Каждый поток читает одну строку *A* и один столбец *B* и вычисляет соответствующий элемент *C,* как показано на рисунке 3-1. Поэтому *A* читается из глобальной памяти *B.width* раз, *B* читается *A.height* раз.  // Матрицы хранятся по строкам:  // M(row, col) = \*(M.elements + row \* M.width + col)  typedef struct {  int width;  int height;  float\* elements;  } Matrix;  // Размер блока нитей  #define BLOCK\_SIZE 16  // Предварительное объявление ядра перемножения матриц  \_\_global\_\_ void MatMulKernel(const Matrix, const Matrix, Matrix);  // Перемножение матриц – код хоста  // Размерности матриц предполагаются кратными BLOCK\_SIZE  void MatMul(const Matrix A, const Matrix B, Matrix C)  {  // Загружаем A и B в память устройства  Matrix d\_A;  d\_A.width = A.width; d\_A.height = A.height;  size\_t size = A.width \* A.height \* sizeof(float);  cudaMalloc(&d\_A.elements, size);  cudaMemcpy(d\_A.elements, A.elements, size,  cudaMemcpyHostToDevice);  Matrix d\_B;  d\_B.width = B.width; d\_B.height = B.height;  size = B.width \* B.height \* sizeof(float);  cudaMalloc(&d\_B.elements, size);  cudaMemcpy(d\_B.elements, B.elements, size,  cudaMemcpyHostToDevice);  // Размещаем C в памяти устройства  Matrix d\_C;  d\_C.width = C.width; d\_C.height = C.height;  size = C.width \* C.height \* sizeof(float);  cudaMalloc(&d\_C.elements, size);  // Вызываем ядро  dim3 dimBlock(BLOCK\_SIZE, BLOCK\_SIZE);  dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);  MatMulKernel<<<dimGrid, dimBlock>>>(d\_A, d\_B, d\_C);  // Читаем C из памяти устройства  cudaMemcpy(C.elements, Cd.elements, size,  cudaMemcpyDeviceToHost);  // Освобождаем память устройства  cudaFree(d\_A.elements);  cudaFree(d\_B.elements);  cudaFree(d\_C.elements);  }  // Ядро перемножения матриц, вызываемое MatMul()  \_\_global\_\_ void MatMulKernel(Matrix A, Matrix B, Matrix C)  {  // Каждый поток вычисляет один элемент C,  // накапливая результат в Cvalue  float Cvalue = 0;  int row = blockIdx.y \* blockDim.y + threadIdx.y;  int col = blockIdx.x \* blockDim.x + threadIdx.x;  for (int e = 0; e < A.width; ++e)  Cvalue += A.elements[row \* A.width + e]  \* B.elements[e \* B.width + col];  C.elements[row \* C.width + col] = Cvalue;  }  <Рисунок 3-1 опущен>  Следующий пример кода это реализация перемножения матриц, использующая преимущества разделяемой памяти. В это реализации каждый блок нитей отвечает за вычисление одной квадратной подматрицы *Csub* матрицы C, и каждый поток внутри блока отвечает за вычисление одного элемента *Csub*. Как показано на рисунке 3-2, *Csub* эквивалентна произведению двух прямоугольных матриц: подматрицы матрицы *A* размерности *(A.width, block\_size)*, имеющей те же индексы строк, что и *Csub*, и подматрицы матрицы *B* размерности *(block\_size, A.width),* имеющей те же индексы столбцов, что и *Csub*. Для помещения в памяти устройства эти две прямоугольные матрицы разделяются на необходимое число квадратных матриц размерности *block\_size,* и *Csub* вычисляется как сумма произведений этих квадратных матриц. Каждое из этих произведений вычисляется так: сначала загружаются две соответствующих квадратных матрицы из глобальной памяти в разделяемую, причем один поток загружает один элемент каждой матрицы, а затем каждый поток вычисляет один элемент произведения. Каждый поток накапливает результаты каждого из этих произведений в регистре и по выполнении записывает результат в глобальную память.  Организуя вычисления таким образом, мы используем преимущества быстрой разделяемой памяти и сохраняем ресурс пропускной способности глобальной памяти, так как А читается из глобальной памяти только *(B.width / block\_size)* раз, *B* читается *(A.height / block\_size)* раз.  Тип *Matrix* из предыдущего примера кода дополняется полем *stride*, так что подматрицы могут быть эффективно представлены тем же типом. Функции **\_\_device\_\_** (см. раздел Б.1.1) используются для получения и установки значения элементов и построения подматриц из матрицы.  // Матрицы хранятся по строкам:  // M(row, col) = \*(M.elements + row \* M.stride + col)  typedef struct {  int width;  int height;  int stride;  float\* elements;  } Matrix;  // Получаем значение элемента матрицы  \_\_device\_\_ float GetElement(const Matrix A, int row, int col)  {  return A.elements[row \* A.stride + col];  }  // Устанавливаем значение элемента матрицы  \_\_device\_\_ void SetElement(Matrix A, int row, int col,  float value)  {  A.elements[row \* A.stride + col] = value;  }  // Получаем подматрицу матрицы А Asub размера BLOCK\_SIZExBLOCK\_SIZE,  // расположенную col подматриц правее и row подматриц ниже  // верхнего левого угла A  \_\_device\_\_ Matrix GetSubMatrix(Matrix A, int row, int col)  {  Matrix Asub;  Asub.width = BLOCK\_SIZE;  Asub.height = BLOCK\_SIZE;  Asub.stride = A.stride;  Asub.elements = &A.elements[A.stride \* BLOCK\_SIZE \* row  + BLOCK\_SIZE \* col];  return Asub;  }  // Размер блока нитей  #define BLOCK\_SIZE 16  // Предварительное объявление ядра перемножения матриц  \_\_global\_\_ void MatMulKernel(const Matrix, const Matrix, Matrix);  // Перемножение матриц – код хоста  // Размерности матриц предполагаются кратными BLOCK\_SIZE  void MatMul(const Matrix A, const Matrix B, Matrix C)  {  // Загружаем A и B в память устройства  Matrix d\_A;  d\_A.width = d\_A.stride = A.width; d\_A.height = A.height;  size\_t size = A.width \* A.height \* sizeof(float);  cudaMalloc(&d\_A.elements, size);  cudaMemcpy(d\_A.elements, A.elements, size,  cudaMemcpyHostToDevice);  Matrix d\_B;  d\_B.width = d\_B.stride = B.width; d\_B.height = B.height;  size = B.width \* B.height \* sizeof(float);  cudaMalloc(&d\_B.elements, size);  cudaMemcpy(d\_B.elements, B.elements, size,  cudaMemcpyHostToDevice);  // Размещаем C в памяти устройства  Matrix d\_C;  d\_C.width = d\_C.stride = C.width; d\_C.height = C.height;  size = C.width \* C.height \* sizeof(float);  cudaMalloc(&d\_C.elements, size);  // Вызываем ядро  dim3 dimBlock(BLOCK\_SIZE, BLOCK\_SIZE);  dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);  MatMulKernel<<<dimGrid, dimBlock>>>(d\_A, d\_B, d\_C);  // Читаем C из памяти устройства  cudaMemcpy(C.elements, d\_C.elements, size,  cudaMemcpyDeviceToHost);  // Освобождаем память устройства  cudaFree(d\_A.elements);  cudaFree(d\_B.elements);  cudaFree(d\_C.elements);  }  // Ядро перемножения матриц, вызываемое MatMul()  \_\_global\_\_ void MatMulKernel(Matrix A, Matrix B, Matrix C)  {  // Строка и столбец блока  int blockRow = blockIdx.y;  int blockCol = blockIdx.x;  // Каждый блок нитей вычисляет одну подматрицу Csub матрицы C  Matrix Csub = GetSubMatrix(C, blockRow, blockCol);  // Каждый поток вычисляет один элемент Csub,  // накапливая результат в Cvalue  float Cvalue = 0;  // Строка и столбец нити в Csub  int row = threadIdx.y;  int col = threadIdx.x;  // Цикл по всем подматрицам A и B,  // необходимым для вычисления Csub  // Перемножаем каждую пару  // и накапливаем результат  for (int m = 0; m < (A.width / BLOCK\_SIZE); ++m) {  // Получаем подматрицу Asub матрицы A  Matrix Asub = GetSubMatrix(A, blockRow, m);  // Получаем подматрицу Bsub матрицы B  Matrix Bsub = GetSubMatrix(B, m, blockCol);  // Разделяемая память используется для хранения Asub и Bsub соответственно  \_\_shared\_\_ float As[BLOCK\_SIZE][BLOCK\_SIZE];  \_\_shared\_\_ float Bs[BLOCK\_SIZE][BLOCK\_SIZE];  // Загружаем Asub и Bsub из памяти устройства в разделяемую память  // Каждый поток загружает один элемент каждой подматрицы  As[row][col] = GetElement(Asub, row, col);  Bs[row][col] = GetElement(Bsub, row, col);  // Синхронизируем для обеспечения загрузки подматриц  // перед началом вычислений  \_\_syncthreads();  // Перемножаем Asub и Bsub  for (int e = 0; e < BLOCK\_SIZE; ++e)  Cvalue += As[row][e] \* Bs[e][col];  // Синхронизируем для обеспечения выполнения предыдущего  // вычисления перед загрузкой двух новых  // подматриц A и B в следующей итерации  \_\_syncthreads();  }  // Записываем Csub в память устройства  // Каждый поток пишет один элемент  SetElement(Csub, row, col, Cvalue);  }  <Рисунок 3-2 опущен> |
| 3.2.4 Page-Locked Host Memory  The runtime provides functions to allow the use of *page-locked* (also known as *pinned*) host memory (as opposed to regular pageable host memory allocated by **malloc()**):   * **cudaHostAlloc()** and **cudaFreeHost()** allocate and free page-locked host memory; * **cudaHostRegister()** page-locks a range of memory allocated by **malloc()** (see reference manual for limitations).   Using page-locked host memory has several benefits:   * Copies between page-locked host memory and device memory can be performed concurrently with kernel execution for some devices as mentioned in Section 3.2.5; * On some devices, page-locked host memory can be mapped into the address space of the device, eliminating the need to copy it to or from device memory as detailed in Section 3.2.4.3; * On systems with a front-side bus, bandwidth between host memory and device memory is higher if host memory is allocated as page-locked and even higher if in addition it is allocated as write-combining as described in Section 3.2.4.2.   Page-locked host memory is a scarce resource however, so allocations in page-locked memory will start failing long before allocations in pageable memory. In addition, by reducing the amount of physical memory available to the operating system for paging, consuming too much page-locked memory reduces overall system performance.  The simple zero-copy SDK sample comes with a detailed document on the page-locked memory APIs. | 3.2.4 Странично-фиксированная память хоста  Среда выполнения предоставляет функции для разрешения использования *странично-фиксированной* (также известной как *закрепленная*) памяти хоста (в противоположность обычной памяти со страничной организацией («листаемой»), выделяемой с помощью **malloc()**):   * **cudaHostAlloc()** и **cudaFreeHost()** выделяют и освобождают странично-фиксированную память; * **cudaHostRegister()** странично-блокирует диапазон памяти, выделенной с помощью **malloc()** (для выяснения ограничений см. справочное руководство).   Использование странично-фиксированной памяти хоста имеет несколько преимуществ:   * Копирование между странично-фиксированной памятью хоста и памятью устройства может быть выполнено одновременно с выполнением ядра для некоторых устройств, как указано в разделе 3.2.5; * На некоторых устройствах странично-фиксированная память хоста может быть отображена в адресное пространство устройства, что устраняет необходимость ее копирования в/из устройства (подробнее см. раздел 3.2.4.3); * На системах с управляющей шиной (front-side bus) пропускная способность между памятью хоста и памятью устройства выше, если память хоста выделена как странично-фиксированная, и еще выше, если дополнительно она выделена в *режиме комбинированной записи* (*write-combining*), как описано в разделе 3.2.4.2.   Странично-фиксированная память хоста, тем не менее, является дефицитным ресурсом, так что выделения в странично-фиксированной памяти перестанут выполняться намного раньше, чем выделения в памяти со страничной организацией. К тому же из-за уменьшения объема физической памяти, доступной операционной системе для подкачки (paging), потребление слишком большого количества странично-фиксированной памяти снижает общую производительность системы.  Простой пример без копирования (SDK) поставляется вместе с подробным описанием API странично-фиксированной памяти. |
| 3.2.4.1 Portable Memory  A block of page-locked memory can be used in conjunction with any device in the system (see Section 3.2.6 for more details on multi-device systems), but by default, the benefits of using page-locked memory described above are only available in conjunction with the device that was current when the block was allocated (and with all devices sharing the same unified address space, if any, as described in Section 3.2.7). To make these advantages available to all devices, the block needs to be allocated by passing the flag **cudaHostAllocPortable** to **cudaHostAlloc()** or page-locked by passing the flag **cudaHostRegisterPortable** to **cudaHostRegister()**. | 3.2.4.1 Переносимая память  Блок странично-фиксированной памяти может быть использован вместе с любым устройством в системе (см. раздел 3.2.6 для получения дополнительной информации о системах со многими устройствами), но по умолчанию описанные выше преимущества использования странично-фиксированной памяти доступны только для устройства, которое было текущим в момент выделения блока (и при разделении всеми устройствами унифицированного адресного пространства (при его наличии), как описано в разделе 3.2.7). Для получения возможности использования этих преимуществ всеми устройствами блок должен быть выделен с помощью передачи флага **cudaHostAllocPortable** функции **cudaHostAlloc()** или странично-фиксирован с помощью передачи флага **cudaHostRegisterPortable** функции **cudaHostRegister()**. |
| 3.2.4.2 Write-Combining Memory  By default page-locked host memory is allocated as cacheable. It can optionally be allocated as *write-combining* instead by passing flag **cudaHostAllocWriteCombined** to **cudaHostAlloc()**. Write-combining memory frees up the host’s L1 and L2 cache resources, making more cache available to the rest of the application. In addition, write-combining memory is not snooped during transfers across the PCI Express bus, which can improve transfer performance by up to 40%.  Reading from write-combining memory from the host is prohibitively slow, so write-combining memory should in general be used for memory that the host only writes to. | 3.2.4.2 Память в режиме комбинированной записи  По умолчанию странично-фиксированная память выделена как кэшируемая. Она может быть альтернативно выделена в режиме комбинированной записи передачей флага **cudaHostAllocWriteCombined** функции **cudaHostAlloc()**. Память в режиме комбинированной записи освобождает ресурсы кэша хоста L1 и L2, давая остальной части приложения доступ к большему объему кэша. Кроме того, память в режиме комбинированной записи не отслеживается во время передачи по шине PCI Express, что может увеличить производительность передачи на 40% (максимум).  Чтение из памяти хоста, находящейся в режиме комбинированной записи происходит крайне медленно, так что она должна быть в основном использована как память хоста только для записи. |
| 3.2.4.3 Mapped Memory  On devices of compute capability greater than 1.0, a block of page-locked host memory can also be mapped into the address space of the device by passing flag **cudaHostAllocMapped** to **cudaHostAlloc()** or by passing flag **cudaHostRegisterMapped** to **cudaHostRegister()**. Such a block has therefore in general two addresses: one in host memory that is returned by **cudaHostAlloc()** or **malloc()**, and one in device memory that can be retrieved using **cudaHostGetDevicePointer()** and then used to access the block from within a kernel. The only exception is for pointers allocated with **cudaHostAlloc()** and when a unified address space is used for the host and the device as mentioned in Section 3.2.7.  Accessing host memory directly from within a kernel has several advantages:   * There is no need to allocate a block in device memory and copy data between this block and the block in host memory; data transfers are implicitly performed as needed by the kernel; * There is no need to use streams (see Section 3.2.5.4) to overlap data transfers with kernel execution; the kernel-originated data transfers automatically overlap with kernel execution.   Since mapped page-locked memory is shared between host and device however, the application must synchronize memory accesses using streams or events (see Section 3.2.5) to avoid any potential read-after-write, write-after-read, or write-after-write hazards.  To be able to retrieve the device pointer to any mapped page-locked memory, page-locked memory mapping must be enabled by calling cudaSetDeviceFlags() with the cudaDeviceMapHost flag before any other CUDA calls is performed. Otherwise, cudaHostGetDevicePointer() will return an error.  cudaHostGetDevicePointer() also returns an error if the device does not support mapped page-locked host memory. Applications may query this capability by checking the canMapHostMemory device property (see Section 3.2.6.1), which is equal to 1 for devices that support mapped page-locked host memory.  Note that atomic functions (Section B.11) operating on mapped page-locked memory are not atomic from the point of view of the host or other devices. | 3.2.4.3 Отображенная память  На устройствах вычислительной способности выше 1.0, блок странично-фиксированной памяти может также быть отображен в адресное пространство устройства передачей флага **cudaHostAllocMapped** функции **cudaHostAlloc()** или передачей флага **cudaHostRegisterMapped** функции **cudaHostRegister()**. Такой блок вследствие этого вообще имеет два адреса: один в памяти хоста, возвращаемый **cudaHostAlloc()** или **malloc()**, и один в памяти устройства, который может быть получен с помощью **cudaHostGetDevicePointer()** и затем использован для доступа к блоку из ядра. Единственное исключение – указатели, размещенные с помощью **cudaHostAlloc()** и при использовании унифицированного адресного пространства для хоста и устройства, как описано в разделе 3.2.7.  Доступ к памяти хоста напрямую из ядра имеет несколько преимуществ:   * Не нужно выделять блок в памяти устройства и копировать данные между этим блоком и блоком в памяти хоста; передача данных происходит скрыто при ее необходимости ядру; * Не нужно использовать потоки (см. раздел 3.2.5.4) для перекрытия передачи данных исполнением ядра; инициированная ядром передача данных автоматически перекрывается исполнением ядра.   Так как отображенная странично-фиксированная память в любом случае разделяется между хостом и устройством, приложение должно синхронизировать доступы к памяти, используя потоки или события (см. раздел 3.2.5) для предотвращения любых возможных случаев «грязного чтения» после записи, «грязной записи» после чтения или «грязной записи».  Для обеспечения возможности получения указателя устройства на любую странично-фиксированную память, отображение странично-фиксированной памяти должно быть включено вызовом **cudaSetDeviceFlags()** с флагом **cudaDeviceMapHost** перед любым другим вызовом CUDA. Иначе **cudaHostGetDevicePointer()** возвратит ошибку.  **cudaHostGetDevicePointer()** также возвращает ошибку, если устройство не поддерживает отображенную странично-фиксированную память хоста. Приложения могут запрашивать наличие такой возможности проверкой свойства устройства **canMapHostMemory** (см. раздел 3.2.6.1), которое равно 1 для устройств, поддерживающих странично-фиксированную память хоста.  Заметьте, что атомарные функции (см. раздел Б.11), оперирующие на отображенной странично-фиксированной памяти, не атомарны с точки зрения хоста или других устройств. |
| 3.2.5 Asynchronous Concurrent Execution  3.2.5.1 Concurrent Execution between Host and Device  In order to facilitate concurrent execution between host and device, some function calls are asynchronous: Control is returned to the host thread before the device has completed the requested task. These are:   * Kernel launches; * Memory copies between two addresses to the same device memory; * Memory copies from host to device of a memory block of 64 KB or less; * Memory copies performed by functions that are suffixed with **Async**; * Memory set function calls.   Programmers can globally disable asynchronous kernel launches for all CUDA applications running on a system by setting the **CUDA\_LAUNCH\_BLOCKING** environment variable to 1. This feature is provided for debugging purposes only and should never be used as a way to make production software run reliably.  When an application is run via cuda-gdb, the Visual Profiler, or the Parallel Nsight CUDA Debugger, all launches are synchronous. | 3.2.5 Асинхронное одновременное исполнение  3.2.5.1 Одновременное исполнение между хостом и устройством.  Для организации одновременного выполнения между хостом и устройством некоторые вызовы функций асинхронны: управление возвращается нити хоста перед завершением устройством запрошенного задания. Это:   * Запуски ядра; * Копирование памяти между двумя адресами в память одного и того же устройства; * Копирование памяти из хоста в устройство с блоком памяти 64 КБ или меньше; * Копирование памяти, выполненное с использованием функций, имеющих суффикс **Async**; * Вызовы функций установки памяти.   Программисты могут глобально отключить асинхронные запуски ядра для всех приложений CUDA, выполняющихся в системе, установкой переменной среды **CUDA\_LAUNCH\_BLOCKING** в 1. Эта возможность предоставлена только для целей отладки и не рекомендуется к использованию как средство обеспечения надежности ПО.  При запуске приложения через cuda-gdb, Visual Profiler или Parallel Nsight CUDA Debugger все вызовы синхронны. |
| 3.2.5.2 Overlap of Data Transfer and Kernel Execution  Some devices of compute capability 1.1 and higher can perform copies between page-locked host memory and device memory concurrently with kernel execution. Applications may query this capability by checking the **asyncEngineCount** device property (see Section 3.2.6.1), which is greater than zero for devices that support it. For devices of compute capability 1.x, this capability is only supported for memory copies that do not involve CUDA arrays or 2D arrays allocated through **cudaMallocPitch()** (see Section 3.2.2). | 3.2.5.2 Перекрытие передач данных и исполнения ядра  Некоторые устройства вычислительной способности 1.1 и выше могут выполнять копирование между странично-фиксированной памятью хоста и памятью устройства одновременно с исполнением ядра. Приложения могут запрашивать наличие такой возможности проверкой свойства устройства **asyncEngineCount** (см. раздел 3.2.6.1), которое больше нуля для устройств, поддерживающих её. Для устройств вычислительной способности 1.x эта возможность поддерживается только для копирования памяти, не включающей массивы CUDA или 2D-массивы, выделенные с использованием **cudaMallocPitch()** (см. раздел 3.2.2). |
| 3.2.5.3 Concurrent Kernel Execution  Some devices of compute capability 2.x and higher can execute multiple kernels concurrently. Applications may query this capability by checking the **concurrentKernels** device property (see Section 3.2.6.1), which is equal to 1 for devices that support it.  The maximum number of kernel launches that a device can execute concurrently is sixteen.  A kernel from one CUDA context cannot execute concurrently with a kernel from another CUDA context.  Kernels that use many textures or a large amount of local memory are less likely to execute concurrently with other kernels. | 3.2.5.3 Одновременное исполнение ядер  Некоторые устройства вычислительной способности 2.x и выше могут одновременно исполнять несколько ядер. Приложения могут запрашивать наличие такой возможности проверкой свойства устройства **concurrentKernels** (см. раздел 3.2.6.1), которое равно 1 для устройств, поддерживающих её.  Максимальное число одновременно исполняемых запусков ядер для одного устройства – 16.  Ядро из одного контекста CUDA не может исполняться одновременно с ядром из другого контекста CUDA.  Для ядер, использующих множество текстур или большой объём локальной памяти, одновременное исполнение с другими ядрами наименее вероятно. |
| 3.2.5.4 Concurrent Data Transfers  Some devices of compute capability 2.x and higher can perform a copy from page-locked host memory to device memory concurrently with a copy from device memory to page-locked host memory.  Applications may query this capability by checking the **asyncEngineCount** device property (see Section 3.2.6.1), which is equal to 2 for devices that support it. | 3.2.5.4 Одновременные передачи данных  Некоторые устройства вычислительной способности 2.x и выше могут выполнять копирование из странично-фиксированной памяти хоста в память устройства одновременно с копированием из памяти устройства в странично-фиксированную память хоста.  Приложения могут запрашивать наличие такой возможности проверкой свойства устройства **asyncEngineCount** (см. раздел 3.2.6.1), которое равно 2 для устройств, которые её поддерживают. |
| 3.2.5.5 Streams  Applications manage concurrency through *streams*. A stream is a sequence of commands (possibly issued by different host threads) that execute in order. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently; this behavior is not guaranteed and should therefore not be relied upon for correctness (e.g. inter-kernel communication is undefined). | 3.2.5.5 Потоки  Приложения управляют одновременностью через *потоки*. Поток это последовательность команд (возможно, запущенные различными нитями хоста), которые исполняются по порядку. Разные потоки, с другой стороны, могут исполнять свои команды не по порядку друг относительно друга или одновременно; это поведение не гарантировано, на его корректность не следует полагаться (например, внутриядерные коммуникации неопределённы). |
| 3.2.5.5.1 Creation and Destruction  A stream is defined by creating a stream object and specifying it as the stream parameter to a sequence of kernel launches and host  device memory copies. The following code sample creates two streams and allocates an array **hostPtr** of **float** in page-locked memory.  cudaStream\_t stream[2];  for (int i = 0; i < 2; ++i)  cudaStreamCreate(&stream[i]);  float\* hostPtr;  cudaMallocHost(&hostPtr, 2 \* size);  Each of these streams is defined by the following code sample as a sequence of one memory copy from host to device, one kernel launch, and one memory copy from device to host:  for (int i = 0; i < 2; ++i) {  cudaMemcpyAsync(inputDevPtr + i \* size, hostPtr + i \* size,  size, cudaMemcpyHostToDevice, stream[i]);  MyKernel<<<100, 512, 0, stream[i]>>>  (outputDevPtr + i \* size, inputDevPtr + i \* size, size);  cudaMemcpyAsync(hostPtr + i \* size, outputDevPtr + i \* size,  size, cudaMemcpyDeviceToHost, stream[i]);  }  Each stream copies its portion of input array **hostPtr** to array **inputDevPtr** in device memory, processes **inputDevPtr** on the device by calling **MyKernel()**, and copies the result **outputDevPtr** back to the same portion of **hostPtr**. Section 3.2.5.5.5 describes how the streams overlap in this example depending on the capability of the device. Note that **hostPtr** must point to page-locked host memory for any overlap to occur.  Streams are released by calling **cudaStreamDestroy()**.  for (int i = 0; i < 2; ++i)  cudaStreamDestroy(stream[i]);  cudaStreamDestroy() waits for all preceding commands in the given stream to complete before destroying the stream and returning control to the host thread. | 3.2.5.5.1 Создание и разрушение  Поток определяется созданием объекта потока и указанием его как параметра потока для последовательности запусков ядер и копирований памяти хост <–> устройство. Следующий пример кода создает два потока и выделяет массив **hostPtr** типа **float** в странично-фиксированной памяти.  cudaStream\_t stream[2];  for (int i = 0; i < 2; ++i)  cudaStreamCreate(&stream[i]);  float\* hostPtr;  cudaMallocHost(&hostPtr, 2 \* size);  Каждый из этих потоков определен следующим примером кода как последовательность одного копирования из памяти хоста в память устройства, одним запуском ядра и одним копированием и памяти устройства в память хоста:  for (int i = 0; i < 2; ++i) {  cudaMemcpyAsync(inputDevPtr + i \* size, hostPtr + i \* size,  size, cudaMemcpyHostToDevice, stream[i]);  MyKernel<<<100, 512, 0, stream[i]>>>  (outputDevPtr + i \* size, inputDevPtr + i \* size, size);  cudaMemcpyAsync(hostPtr + i \* size, outputDevPtr + i \* size,  size, cudaMemcpyDeviceToHost, stream[i]);  }  Каждый поток копирует свою часть входного массива **hostPtr** в массив **inputDevPtr** в памяти устройства, обрабатывает **inputDevPtr** на устройстве, вызывая **MyKernel()**, и копирует результат **outputDevPtr** обратно в ту же часть **hostPtr**. Раздел 3.2.5.5.5 описывает, как потоки перекрываются в этом примере в зависимости от возможностей устройства. Заметьте, что **hostPtr** должен указывать на странично-фиксированную память хоста для создания возможности возникновения перекрывания.  Потоки освобождаются вызовом **cudaStreamDestroy()**.  for (int i = 0; i < 2; ++i)  cudaStreamDestroy(stream[i]);  **cudaStreamDestroy()** ждет выполнения всех предшествующих команд в данном потоке перед разрушением потока и возвращением управления нити хоста. |
| 3.2.5.5.2 Default Stream  Kernel launches and host  device memory copies that do not specify any stream parameter, or equivalently that set the stream parameter to zero, are issued to the default stream. They are therefore executed in order. | 3.2.5.5.2 Поток по умолчанию  Запуски ядра и копирование памяти хост <–> устройство, для которых не указан параметр потока или, что равнозначно, для которых он установлен в ноль, запускаются в поток по умолчанию. Поэтому они исполняются по порядку. |
| 3.2.5.5.3 Explicit Synchronization  There are various ways to explicitly synchronize streams with each other.  **cudaDeviceSynchronize()** waits until all preceding commands in all streams of all host threads have completed.  **cudaStreamSynchronize()** takes a stream as a parameter and waits until all preceding commands in the given stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device.  **cudaStreamWaitEvent()** takes a stream and an event as parameters (see Section 3.2.5.6 for a description of events) and makes all the commands added to the given stream after the call to **cudaStreamWaitEvent()** delay their execution until the given event has completed. The stream can be 0, in which case all the commands added to any stream after the call to **cudaStreamWaitEvent()** wait on the event.  **cudaStreamQuery()** provides applications with a way to know if all preceding commands in a stream have completed.  To avoid unnecessary slowdowns, all these synchronization functions are usually best used for timing purposes or to isolate a launch or memory copy that is failing. | 3.2.5.5.3 Явная синхронизация  Существуют различные способы явной синхронизации потоков.  **cudaDeviceSynchronize()** ждет выполнения всех предшествующих команд во всех потоках всех нитей хоста.  **cudaStreamSynchronize()** принимает поток как параметр и ждет выполнения всех предшествующих команд данного потока. Может быть использована для синхронизации хоста с определенным потоком, позволяя другим потокам продолжать исполнение на устройстве.  **cudaStreamWaitEvent()** принимает поток и событие как параметры (описание событий см. в разделе 3.2.5.6) и заставляет все команды, добавленные в данный поток после вызова **cudaStreamWaitEvent()** отложить своё исполнение до завершения данного события. Поток может иметь значение 0, в этом случае все команды, добавленные в любой поток после вызова **cudaStreamWaitEvent()** ждут завершения события.  **cudaStreamQuery()** предоставляет приложениям способ узнать, выполнились ли все предшествующие команды в потоке.  Для предотвращения излишних замедлений все эти функции синхронизации обычно используются для целей согласования во времени или изоляции неудавшегося запуска копирования памяти. |
| 3.2.5.5.4 Implicit Synchronization  Two commands from different streams cannot run concurrently if either one of the following operations is issued in-between them by the host thread:   * a page-locked host memory allocation, * a device memory allocation, * a device memory set, * a memory copy between two addresses to the same device memory, * any CUDA command to the default stream, * a switch between the L1/shared memory configurations described in Section F.4.1.   For devices that support concurrent kernel execution, any operation that requires a dependency check to see if a streamed kernel launch is complete:   * Can start executing only when all thread blocks of all prior kernel launches from any stream in the CUDA context have started executing; * Blocks all later kernel launches from any stream in the CUDA context until the kernel launch being checked is complete.   Operations that require a dependency check include any other commands within the same stream as the launch being checked and any call to cudaStreamQuery() on that stream. Therefore, applications should follow these guidelines to improve their potential for concurrent kernel execution:   * All independent operations should be issued before dependent operations, * Synchronization of any kind should be delayed as long as possible. | 3.2.5.5.4 Неявная синхронизация  Две команды из разных потоков не могут исполняться одновременно, если любая из следующих операций запущена между ними нитью хоста:   * выделение странично-фиксированной памяти хоста, * выделение памяти устройства, * установка (set) памяти устройства, * копирование памяти между двумя адресами в памяти одного устройства, * любая команда CUDA в потоке по умолчанию, * переключение между конфигурациями памяти L1/разделённая (описаны в разделе Е.4.1).   Для устройств, поддерживающих одновременное исполнение ядер, любая операция, для которой необходима проверка зависимости с целью выяснения завершения запуска ядра в потоке:   * Может начать исполнение только когда все блоки нитей всех предшествующих запусков ядра из любого потока в контексте CUDA начали исполнение; * Блокирует все последующие запуски ядра из любого потока в контексте CUDA до завершения проверки запуска ядра.   В число операций, для которых необходима проверка зависимости, также входят любые другие команды внутри потока, в котором происходит проверка запуска и любой вызов CudaStreamQuery(). Поэтому приложения должны следовать следующим рекомендациям для увеличения возможности одновременного исполнения ядра:   * Все независимые операции должны быть запущены перед зависимыми, * Синхронизация любого вида должна быть отложена на максимально возможный срок. |
| 3.2.5.5.5 Overlapping Behavior  The amount of execution overlap between two streams depends on the order in which the commands are issued to each stream and whether or not the device supports overlap of data transfer and kernel execution (Section 3.2.5.2), concurrent kernel execution (Section 3.2.5.3), and/or concurrent data transfers (Section 3.2.5.4).  For example, on devices that do not support concurrent data transfers, the two streams of the code sample of Section 3.2.5.5.1 do not overlap at all because the memory copy from host to device is issued to stream 1 after the memory copy from device to host is issued to stream 0, so it can only start once the memory copy from device to host issued to stream 0 has completed. If the code is rewritten the following way (and assuming the device supports overlap of data transfer and kernel execution)  for (int i = 0; i < 2; ++i)  cudaMemcpyAsync(inputDevPtr + i \* size, hostPtr + i \* size,  size, cudaMemcpyHostToDevice, stream[i]);  for (int i = 0; i < 2; ++i)  MyKernel<<<100, 512, 0, stream[i]>>>  (outputDevPtr + i \* size, inputDevPtr + i \* size, size);  for (int i = 0; i < 2; ++i)  cudaMemcpyAsync(hostPtr + i \* size, outputDevPtr + i \* size,  size, cudaMemcpyDeviceToHost, stream[i]);  then the memory copy from host to device issued to stream 1 overlaps with the kernel launch issued to stream 0.  On devices that do support concurrent data transfers, the two streams of the code sample of Section 3.2.5.5.1 do overlap: The memory copy from host to device issued to stream 1 overlaps with the memory copy from device to host issued to stream 0 and even with the kernel launch issued to stream 0 (assuming the device supports overlap of data transfer and kernel execution). However, the kernel executions cannot possibly overlap because the second kernel launch is issued to stream 1 after the memory copy from device to host is issued to stream 0, so it is blocked until the first kernel launch issued to stream 0 is complete as per Section 3.2.5.5.4. If the code is rewritten as above, the kernel executions overlap (assuming the device supports concurrent kernel execution) since the second kernel launch is issued to stream 1 before the memory copy from device to host is issued to stream 0. In that case however, the memory copy from device to host issued to stream 0 only overlaps with the last thread blocks of the kernel launch issued to stream 1 as per Section 3.2.5.5.4, which can represent only a small portion of the total execution time of the kernel. | 3.2.5.5.5 Перекрывающее поведение  Объем перекрытия исполнения между двумя потоками зависит от порядка, в котором команды выпущены в каждый поток и поддержки устройством перекрытия передач данных и исполнения ядер (раздел 3.2.5.2), одновременного исполнения ядер (раздел 3.2.5.3), и/или одновременных передач данных (раздел 3.2.5.4).  Например, на устройствах, не поддерживающих одновременные передачи данных, два потока примера кода из раздела 3.2.5.5.1 совсем не перекрываются, так как копирование памяти из хоста в устройство запущено в поток 1 после того, как копирование памяти из устройства в хост запущено в поток 0, так что оно может начаться только после завершения копирования памяти из устройства в хост. Если переписать код следующим образом (предполагая, что устройство поддерживает перекрывание передачи данных и исполнения ядра)  for (int i = 0; i < 2; ++i)  cudaMemcpyAsync(inputDevPtr + i \* size, hostPtr + i \* size,  size, cudaMemcpyHostToDevice, stream[i]);  for (int i = 0; i < 2; ++i)  MyKernel<<<100, 512, 0, stream[i]>>>  (outputDevPtr + i \* size, inputDevPtr + i \* size, size);  for (int i = 0; i < 2; ++i)  cudaMemcpyAsync(hostPtr + i \* size, outputDevPtr + i \* size,  size, cudaMemcpyDeviceToHost, stream[i]);  то копирование памяти из хоста в устройство, запущенное в поток 1 будет перекрываться с запуском ядра в потоке 0.  На устройствах, поддерживающих одновременные передачи данных, два потока примера кода из раздела 3.2.5.5.1 перекрываются: копирование памяти из хоста в устройство, запущенное в поток 1 перекрывается с копированием памяти из устройства в хост и даже с запуском ядра, запущенными в потоке 0 (предполагая, что устройство поддерживает перекрывание передач данных и исполнения ядра). Тем не менее, исполнения ядра не могут перекрываться, так как запуск второго ядра запущен в потоке 1 после того, как копирование памяти из устройства в хост было запущено в потоке 0, так что он блокирован до завершения первого запуска ядра, запущенного в поток 0 (согласно разделу 3.2.5.5.4). Если переписать код, как показано выше, исполнения ядра перекрываются (предполагая, что устройство поддерживает одновременное исполнение ядра), так как второй запуск ядра запущен в поток 1 перед копированием памяти из устройства в хост, запущенным в поток 0. Однако в этом случае копирование памяти из устройства в хост, запущенное в поток 0, перекрывается только с последними блоками нитей запуска ядра, запущенного в поток 1 (согласно разделу 3.2.5.5.4), что является лишь малой частью полного времени исполнения ядра. |
| 3.2.5.6 Events  The runtime also provides a way to closely monitor the device’s progress, as well as perform accurate timing, by letting the application asynchronously record *events* at any point in the program and query when these events are completed. An event has completed when all tasks – or optionally, all commands in a given stream – preceding the event have completed. Events in stream zero are completed after all preceding task and commands in all streams are completed.  3.2.5.6.1 Creation and Destruction  The following code sample creates two events:  cudaEvent\_t start, stop;  cudaEventCreate(&start);  cudaEventCreate(&stop);  They are destroyed this way:  cudaEventDestroy(start);  cudaEventDestroy(stop);  3.2.5.6.2 Elapsed Time  The events created in Section 3.2.5.6.1 can be used to time the code sample of Section 3.2.5.5.1 the following way:  cudaEventRecord(start, 0);  for (int i = 0; i < 2; ++i) {  cudaMemcpyAsync(inputDev + i \* size, inputHost + i \* size,  size, cudaMemcpyHostToDevice, stream[i]);  MyKernel<<<100, 512, 0, stream[i]>>>  (outputDev + i \* size, inputDev + i \* size, size);  cudaMemcpyAsync(outputHost + i \* size, outputDev + i \* size,  size, cudaMemcpyDeviceToHost, stream[i]);  }  cudaEventRecord(stop, 0);  cudaEventSynchronize(stop);  float elapsedTime;  cudaEventElapsedTime(&elapsedTime, start, stop);  3.2.5.7 Synchronous Calls  When a synchronous function is called, control is not returned to the host thread before the device has completed the requested task. Whether the host thread will then yield, block, or spin can be specified by calling **cudaSetDeviceFlags()**with some specific flags (see reference manual for details) before any other CUDA calls is performed by the host thread. | 3.2.5.6 События  Среда выполнения также предоставляет способ тщательного наблюдения прогресса устройства, а также выполнения точного распределения во времени, позволяя приложению асинхронно регистрировать *события* в любой точке программы и запрашивать момент выполнения этих событий. Событие выполняется, когда все задания – или, опционально, все команды в данном потоке – предшествующие событию, выполнились. События в потоке ноль выполняются после выполнения всех предшествующих задач и команд во всех потоках.  3.2.5.6.1 Создание и разрушение  Следующий пример кода создает два события:  cudaEvent\_t start, stop;  cudaEventCreate(&start);  cudaEventCreate(&stop);  Они разрушаются следующим образом:  cudaEventDestroy(start);  cudaEventDestroy(stop);  3.2.5.6.2 Прошедшее время  События, созданные в разделе 3.2.5.6.1, могут быть использованы для измерения фиксации продолжительности исполнения примера кода из раздела 3.2.5.5.1 следующим образом:  cudaEventRecord(start, 0);  for (int i = 0; i < 2; ++i) {  cudaMemcpyAsync(inputDev + i \* size, inputHost + i \* size,  size, cudaMemcpyHostToDevice, stream[i]);  MyKernel<<<100, 512, 0, stream[i]>>>  (outputDev + i \* size, inputDev + i \* size, size);  cudaMemcpyAsync(outputHost + i \* size, outputDev + i \* size,  size, cudaMemcpyDeviceToHost, stream[i]);  }  cudaEventRecord(stop, 0);  cudaEventSynchronize(stop);  float elapsedTime;  cudaEventElapsedTime(&elapsedTime, start, stop);  3.2.5.7 Синхронные вызовы  При вызове синхронной функции управление не возвращается нити хоста до выполнения устройством поставленной задачи. Дальнейшие действия нити хоста – выдачу результатов, блокирование или продление – можно задать вызовом **cudaSetDeviceFlags()** с определенными флагами (подробности см. в справочном руководстве) перед выполнением нитью хоста любого другого вызова CUDA. |
| 3.2.6 Multi-Device System  3.2.6.1 Device Enumeration  A host system can have multiple devices. The following code sample shows how to enumerate these devices, query their properties, and determine the number of CUDA-enabled devices.  int deviceCount;  cudaGetDeviceCount(&deviceCount);  int device;  for (device = 0; device < deviceCount; ++device) {  cudaDeviceProp deviceProp;  cudaGetDeviceProperties(&deviceProp, device);  printf("Device %d has compute capability %d.%d.\n",  device, deviceProp.major, deviceProp.minor);  }  3.2.6.2 Device Selection  A host thread can set the device it operates on at any time by calling **cudaSetDevice()**. Device memory allocations and kernel launches are made on the currently set device; streams and events are created in association with the currently set device. If no call to **cudaSetDevice()** is made, the current device is device 0.  The following code sample illustrates how setting the current device affects memory allocation and kernel execution.  size\_t size = 1024 \* sizeof(float);  cudaSetDevice(0); // Set device 0 as current  float\* p0;  cudaMalloc(&p0, size); // Allocate memory on device 0  MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0  cudaSetDevice(1); // Set device 1 as current  float\* p1;  cudaMalloc(&p1, size); // Allocate memory on device 1  MyKernel<<<1000, 128>>>(p1); // Launch kernel on device 1  3.2.6.3 Stream and Event Behavior  A kernel launch or memory copy will fail if it is issued to a stream that is not associated to the current device as illustrated in the following code sample.  cudaSetDevice(0); // Set device 0 as current  cudaStream\_t s0;  cudaStreamCreate(&s0); // Create stream s0 on device 0  MyKernel<<<100, 64, 0, s0>>>(); // Launch kernel on device 0 in s0  cudaSetDevice(1); // Set device 1 as current  cudaStream\_t s1;  cudaStreamCreate(&s1); // Create stream s1 on device 1  MyKernel<<<100, 64, 0, s1>>>(); // Launch kernel on device 1 in s1  // This kernel launch will fail:  MyKernel<<<100, 64, 0, s0>>>(); // Launch kernel on device 1 in s0  **cudaEventRecord()** will fail if the input event and input stream are associated to different devices.  **cudaEventElapsedTime()** will fail if the two input events are associated to different devices.  **cudaEventSynchronize()** and **cudaEventQuery()** will succeed even if the input event is associated to a device that is different from the current device.  **cudaStreamWaitEvent()** will succeed even if the input stream and input event are associated to different devices. **cudaStreamWaitEvent()** can therefore be used to synchronize multiple devices with each other.  Each device has its own default stream (see Section 3.2.5.5.2), so commands issued to the default stream of a device may execute out of order or concurrently with respect to commands issued to the default stream of any other device.  3.2.6.4 Peer-to-Peer Memory Access  When the application is run as a 64-bit process on Windows Vista/7 in TCC mode (see Section 3.6), on Windows XP, or on Linux, devices of compute capability 2.0 and higher from the Tesla series may address each other’s memory (i.e. a kernel executing on one device can dereference a pointer to the memory of the other device). This peer-to-peer memory access feature is supported between two devices if **cudaDeviceCanAccessPeer()** returns true for these two devices.  Peer-to-peer memory access must be enabled between two devices by calling **cudaDeviceEnablePeerAccess()** as illustrated in the following code sample.  A unified address space is used for both devices (see Section 3.2.7), so the same pointer can be used to address memory from both devices as shown in the code sample below.  cudaSetDevice(0); // Set device 0 as current  float\* p0;  size\_t size = 1024 \* sizeof(float);  cudaMalloc(&p0, size); // Allocate memory on device 0  MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0  cudaSetDevice(1); // Set device 1 as current  cudaDeviceEnablePeerAccess(0, 0); // Enable peer-to-peer access  // with device 0  // Launch kernel on device 1  // This kernel launch can access memory on device 0 at address p0  MyKernel<<<1000, 128>>>(p0);  3.2.6.5 Peer-to-Peer Memory Copy  Memory copies can be performed between the memories of two different devices.  When a unified address space is used for both devices (see Section 3.2.7), this is done using the regular memory copy functions mentioned in Section 3.2.2.  Otherwise, this is done using **cudaMemcpyPeer()**, **cudaMemcpyPeerAsync()**, **cudaMemcpy3DPeer()**, or **cudaMemcpy3DPeerAsync()** as illustrated in the following code sample.  cudaSetDevice(0); // Set device 0 as current  float\* p0;  size\_t size = 1024 \* sizeof(float);  cudaMalloc(&p0, size); // Allocate memory on device 0  cudaSetDevice(1); // Set device 1 as current  float\* p1;  cudaMalloc(&p1, size); // Allocate memory on device 1  cudaSetDevice(0); // Set device 0 as current  MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0  cudaSetDevice(1); // Set device 1 as current  cudaMemcpyPeer(p1, 1, p0, 0, size); // Copy p0 to p1  MyKernel<<<1000, 128>>>(p1); // Launch kernel on device 1  A copy between the memories of two different devices   * does not start until all commands previously issued to either device have completed and * runs to completion before any asynchronous commands (see Section 3.2.5) issued after the copy to either device can start.   Note that if peer-to-peer access is enabled between two devices via **cudaDeviceEnablePeerAccess()** as described in Section 3.2.6.4, peer-to-peer memory copy between these two devices no longer needs to be staged through the host and is therefore faster. | 3.2.6 Системы с несколькими устройствами  3.2.6.1 Перечисление устройств  Система может иметь несколько устройств. Следующий пример кода показывает, как перечислить эти устройства, запросить их свойства и определить число устройств, поддерживающих CUDA.  int deviceCount;  cudaGetDeviceCount(&deviceCount);  int device;  for (device = 0; device < deviceCount; ++device) {  cudaDeviceProp deviceProp;  cudaGetDeviceProperties(&deviceProp, device);  printf("Device %d has compute capability %d.%d.\n",  device, deviceProp.major, deviceProp.minor);  }  3.2.6.2 Выбор устройства  Нить хоста может указать рабочее (*текущее выбранное*) устройство в любой момент вызовом **cudaSetDevice()**. Выделение памяти и вызовы ядра устройства производятся на текущем устройстве; потоки и события создаются в связи с текущем выбранным устройством. Если вызов **cudaSetDevice()** не производится, текущим выбранным устройством является устройство 0.  Следующий пример кода показывает, как выбор текущего устройства влияет на выделение памяти и исполнение ядра.  size\_t size = 1024 \* sizeof(float);  cudaSetDevice(0); // Установить устройство 0 как текущее  float\* p0;  cudaMalloc(&p0, size); // Выделить память на устройстве 0  MyKernel<<<1000, 128>>>(p0); // Запустить ядро на устройстве 0  cudaSetDevice(1); // Установить устройство 1 как текущее  float\* p1;  cudaMalloc(&p1, size); // Выделить память на устройстве 1  MyKernel<<<1000, 128>>>(p1); // Запустить ядро на устройстве 1  3.2.6.3 Поведение потоков и событий  Запуск ядра или копирование памяти не удастся, если оно запущено в поток, не связанный с текущим устройством, как показывает следующий пример кода.  cudaSetDevice(0); // Установить устройство 0 как текущее  cudaStream\_t s0;  cudaStreamCreate(&s0); // Создать поток s0 на устройстве 0  MyKernel<<<100, 64, 0, s0>>>(); // Запустить ядро на устройстве 0 в s0  cudaSetDevice(1); // Установить устройство 1 как текущее  cudaStream\_t s1;  cudaStreamCreate(&s1); // Создать поток s1 на устройстве 1  MyKernel<<<100, 64, 0, s1>>>(); // Запустить ядро на устройстве 1 в s1  // Этот запуск ядра не удастся:  MyKernel<<<100, 64, 0, s0>>>(); // Запустить ядро на устройстве 1 в s0  **cudaEventRecord()** не выполнится, если входное событие и выходной поток связаны с разными устройствами.  **cudaEventElapsedTime()** не выполнится, если два входных события связаны с разными устройствами.  **cudaEventSynchronize()** и **cudaEventQuery()** выполнятся только если входное событие связано с устройством, не являющимся текущим.  **cudaStreamWaitEvent()** выполнится только если входной поток и входное событие связаны с разными устройствами. **cudaStreamWaitEvent()** может, таким образом, быть использована для синхронизации разных нескольких устройств друг с другом.  Каждое устройство имеет свой собственный поток по умолчанию (см. раздел 3.2.5.5.2), поэтому команды, выпущенные в поток устройства по умолчанию, могут выполняться не по порядку или одновременно по отношению к командам, выпущенным в поток по умолчанию любого другого устройства.    3.2.6.4 Равноправный доступ к памяти  Когда приложение выполняется как 64-битный процесс на Windows Vista/7 в режиме TCC (см. раздел 3.6), на Windows XP или Linux, устройства вычислительной способности 2.0 и выше серии Tesla могут обращаться к памяти друг друга (т.е. ядро, исполняемое на одном устройстве, может разыменовывать указатель на память другого устройства). Возможность равноправного доступа к памяти поддерживается между двумя устройствами, если вызов **cudaDeviceCanAccessPeer()** возвращает «true» для этих двух устройств.  Равноправный доступ к памяти должен быть задействован между двумя устройствами вызовом **cudaDeviceEnablePeerAccess()**, как показано в следующем примере кода.  Для обоих устройств используется унифицированное адресное пространство (см. раздел 3.2.7), так что один и тот же указатель может быть использован для обращения к памяти обоими устройствами, как показано в следующем примере кода.  cudaSetDevice(0); // Установить устройство 0 как текущее  float\* p0;  size\_t size = 1024 \* sizeof(float);  cudaMalloc(&p0, size); // Выделить память на устройстве 0  MyKernel<<<1000, 128>>>(p0); // Запустить ядро на устройстве 0  cudaSetDevice(1); // Установить устройство 1 как текущее  cudaDeviceEnablePeerAccess(0, 0); // Задействовать равноправный доступ к памяти с устройством 0  // Запустить ядро на устройстве 1  // Это ядро может получать доступ к памяти устройства 0 по адресу p0  MyKernel<<<1000, 128>>>(p0);  3.2.6.5 Равноправное копирование памяти  Копирование памяти может быть выполнено между памятями двух разных устройств.  Когда для обоих устройств используется унифицированное адресное пространство (см. раздел 3.2.7), оно выполняется с использованием стандартных функций копирования памяти, упомянутых в разделе 3.2.2.  Иначе оно выполняется с использованием функций **cudaMemcpyPeer()**, **cudaMemcpyPeerAsync()**, **cudaMemcpy3DPeer()** и **cudaMemcpy3DPeerAsync()**, как показано в следующем примере кода.  cudaSetDevice(0); // Установить устройство 0 как текущее  float\* p0;  size\_t size = 1024 \* sizeof(float);  cudaMalloc(&p0, size); // Выделить память на устройстве 0  cudaSetDevice(1); // Установить устройство 1 как текущее  float\* p1;  cudaMalloc(&p1, size); // Выделить память на устройстве 1  cudaSetDevice(0); // Установить устройство 0 как текущее  MyKernel<<<1000, 128>>>(p0); // Запустить ядро на устройстве 0  cudaSetDevice(1); // Установить устройство 1 как текущее  cudaMemcpyPeer(p1, 1, p0, 0, size); // Копировать p0 в p1  MyKernel<<<1000, 128>>>(p1); // Запустить ядро на устройстве 1  Копирование между памятями двух разных устройств   * не начинается, пока все команды, ранее запущенные на любом из устройств не завершатся и * выполняется до завершения, так что любая асинхронная команда (см. раздел 3.2.5), запущенная после копирования на любом из устройств, должна ждать этого момента до начала выполнения.   Отметьте, что если равноправный доступ задействован между двумя устройствами через **cudaDeviceEnablePeerAccess()**, как описано в разделе 3.2.6.4, равноправное копирование между этими двумя устройствами не нужно осуществлять через хост, и поэтому оно быстрее. |
| 3.2.7 Unified Virtual Address Space  For 64-bit applications on Windows Vista/7 in TCC mode (see Section 3.6), on Windows XP, or on Linux, a single address space is used for the host and all the devices of compute capability 2.0 and higher. This address space is used for all allocations made in host memory via **cudaHostAlloc()**and in any of the device memories via **cudaMalloc\*()**. Which memory a pointer points to – host memory or any of the device memories – can be determined from the value of the pointer using **cudaPointerGetAttributes()**. As a consequence:   * When copying from or to the memory of one of the devices for which the unified address space is used, the **cudaMemcpyKind** parameter of **cudaMemcpy\*()** becomes useless and can be set to **cudaMemcpyDefault**; * Allocations via **cudaHostAlloc()** are automatically portable (see Section 3.2.4.1) across all the devices for which the unified address space is used, and pointers returned by **cudaHostAlloc()** can be used directly from within kernels running on these devices (i.e. there is no need to obtain a device pointer via **cudaHostGetDevicePointer()** as described in Section 3.2.4.3).   Applications may query if the unified address space is used for a particular device by checking that the **unifiedAddressing** device property (see Section 3.2.6.1) is equal to 1. | 3.2.7 Унифицированное виртуальное адресное пространство  Для 64-битных приложений на Windows Vista/7 в режиме TCC (см. раздел 3.6), на Windows XP или Linux, одно адресное пространство используется для хоста и всех устройств вычислительной способности 2.0 и выше. Это адресное пространство используется для всех выделений памяти хоста через **cudaHostAlloc()** и любого из устройств через **cudaMalloc\*()**. То, на какую память указывает указатель – хоста или любого из устройств – можно узнать по значению указателя с помощью **cudaPointerGetAttributes()**. Как следствие:   * При копировании из или в память устройства, для которого используется унифицированное адресное пространство, параметр **cudaMemcpyKind** функции **cudaMemcpy\*()** становится ненужным и может быть установлен в **cudaMemcpyDefault**; * Память, выделенная через **cudaHostAlloc()**,автоматически является переносимой (см. раздел 3.2.4.1) для всех устройств, для которых используется унифицированное адресное пространство, и указатели, возвращенные **cudaHostAlloc()**, могут быть использованы напрямую из любого ядра, исполняемого на этих устройствах (т.е. нет необходимости получать указатель на устройство через **cudaHostGetDevicePointer()**, как описано в разделе 3.2.4.3).   Приложения могут запрашивать, используется ли унифицированное адресное пространство для определенного устройства, проверяя равенство значения свойства устройства **unifiedAddressing** (см. раздел 3.2.6.1) единице. |
| 3.2.8 Error Checking  All runtime functions return an error code, but for an asynchronous function (see Section 3.2.5), this error code cannot possibly report any of the asynchronous errors that could occur on the device since the function returns before the device has completed the task; the error code only reports errors that occur on the host prior to executing the task, typically related to parameter validation; if an asynchronous error occurs, it will be reported by some subsequent unrelated runtime function call.  The only way to check for asynchronous errors just after some asynchronous function call is therefore to synchronize just after the call by calling **cudaDeviceSynchronize()** (or by using any other synchronization mechanisms described in Section 3.2.5) and checking the error code returned by **cudaDeviceSynchronize()**.  The runtime maintains an error variable for each host thread that is initialized to **cudaSuccess** and is overwritten by the error code every time an error occurs (be it a parameter validation error or an asynchronous error). **cudaPeekAtLastError()** returns this variable. **cudaGetLastError()** returns this variable and resets it to **cudaSuccess**.  Kernel launches do not return any error code, so **cudaPeekAtLastError()** or **cudaGetLastError()** must be called just after the kernel launch to retrieve any pre-launch errors. To ensure that any error returned by **cudaPeekAtLastError()** or **cudaGetLastError()** does not originate from calls prior to the kernel launch, one has to make sure that the runtime error variable is set to **cudaSuccess** just before the kernel launch, for example, by calling **cudaGetLastError()** just before the kernel launch. Kernel launches are asynchronous, so to check for asynchronous errors, the application must synchronize in-between the kernel launch and the call to **cudaPeekAtLastError()** or **cudaGetLastError()**.  Note that **cudaErrorNotReady** that may be returned by **cudaStreamQuery()** and **cudaEventQuery()** is not considered an error and is therefore not reported by **cudaPeekAtLastError()** or **cudaGetLastError()**. | 3.2.8 Проверка ошибок  Все функции среды выполнения возвращают код ошибки, но для асинхронных функций (см. раздел 3.2.5) этот код не может сообщить о каких-либо асинхронных ошибках, могущих произойти на устройстве, так как функция возвращает его до завершения устройством задания; код ошибки сообщает лишь об ошибках, которые возникают на хосте до выполнения задания, обычно они относятся к допустимости параметров; если возникает асинхронная ошибка, она будет сообщена через некоторую позднее вызванную функцию, не относящуюся к вызвавшей ошибку.  Единственный способ проверить возникновение асинхронных ошибок сразу после вызова некоторой асинхронной функции – синхронизироваться с ней сразу после вызова с помощью **cudaDeviceSynchronize()** (или используя любой другой механизм синхронизации из описанных в разделе 3.2.5) и проверить код ошибки, возвращенный **cudaDeviceSynchronize()**.  Среда выполнения поддерживает переменную ошибки равной **cudaSuccess** для каждой инициализированной нити и переписывает его кодом ошибки при каждом ее возникновении (будь это проверка допустимости параметра или асинхронная ошибка). **cudaPeekAtLastError()** возвращает эту переменную. **cudaGetLastError()** возвращает эту переменную и сбрасывает ее в значение **cudaSuccess**.  Запуски ядра не возвращают какой-либо код ошибки, так что **cudaPeekAtLastError()** или **cudaGetLastError()** должны быть вызваны сразу после запуска ядра для получения ошибок, произошедших перед запуском. Чтобы убедиться, что ошибка, возвращенная **cudaPeekAtLastError()** или **cudaGetLastError()**, не происходит из вызовов, предшествующих запуску ядра, необходимо проверять, что переменная ошибки среды выполнения установлена в **cudaSuccess** как раз перед запуском ядра, например, вызовом **cudaGetLastError()**. Запуски ядра асинхронны, так что для проверки асинхронных ошибок приложение должно синхронизировать запуск ядра и вызов функции **cudaPeekAtLastError()** или **cudaGetLastError()**.  Отметьте, что **cudaErrorNotReady**, возвращаемая **cudaStreamQuery()** и **cudaEventQuery()**, не рассматривается как ошибка и поэтому не отображается функциями  **cudaPeekAtLastError()** и **cudaGetLastError()**. |
| 3.2.9 Call Stack  On devices of compute capability 2.x and higher, the size of the call stack can be queried using **cudaDeviceGetLimit()** and set using **cudaDeviceSetLimit()**.  When the call stack overflows, the kernel call fails with a stack overflow error if the application is run via a CUDA debugger (cuda-gdb, Parallel Nsight) or an unspecified launch error, otherwise. | 3.2.9 Стек вызовов  На устройствах вычислительной способности 2.x и выше размер стека вызовов может быть получен с помощью **cudaDeviceGetLimit()** и установлен с помощью **cudaDeviceSetLimit()**.  Когда стек вызов переполняется, попытка вызова ядра приводит к выводу ошибки переполнения стека, если приложение запущено через отладчик CUDA (cuda-gdb, Parallel Nsight), иначе – неуказанной ошибки запуска. |
| 3.2.10 Texture and Surface Memory  CUDA supports a subset of the texturing hardware that the GPU uses for graphics to access texture and surface memory. Reading data from texture or surface memory instead of global memory can have several performance benefits as described in Section 5.3.2.5. | 3.2.10 Текстурная и поверхностная память  CUDA поддерживает подмножество аппаратного обеспечения текстурирования, которое ГП использует для обработки графики, для получения доступа к текстурной и поверхностной памяти. Чтение данных из текстурной или поверхностной памяти вместо глобальной может приводить к некоторому выигрышу в производительности, что описано в разделе 5.3.2.5. |
| 3.2.10.1 Texture Memory  Texture memory is read from kernels using the device functions described in Section B.8. The process of reading a texture is called a *texture fetch*. The first parameter of a texture fetch specifies an object called a *texture reference*.  A texture reference defines which part of texture memory is fetched. As detailed in Section 3.2.10.1.3, it must be bound through runtime functions to some region of memory, called a *texture*, before it can be used by a kernel. Several distinct texture references might be bound to the same texture or to textures that overlap in memory.  A texture reference has several attributes. One of them is its dimensionality that specifies whether the texture is addressed as a one-dimensional array using one *texture coordinate*, a two-dimensional array using two texture coordinates, or a three-dimensional array using three texture coordinates. Elements of the array are called *texels*, short for “texture elements.” The type of a texel is restricted to the basic integer and single-precision floating-point types and any of the 1-, 2-, and 4-component vector types defined in Section B.3.1  Other attributes define the input and output data types of the texture fetch, as well as how the input coordinates are interpreted and what processing should be done.  A texture can be any region of linear memory or a CUDA array (described in Section 3.2.10.2.3).  Table F-2 lists the maximum texture width, height, and depth depending on the compute capability of the device.  Textures can also be layered as described in Section 3.2.10.1.5. | 3.2.10.1 Текстурная память  Текстурная память читается из ядер с помощью функций устройства, описанных в разделе Б.8. Процесс чтения текстуры называется извлечением текстуры (*texture fetch*). Первый параметр извлечения текстуры указывает на объект, называемый *ссылкой на текстуру*.  Ссылка на текстуру указывает, какая часть текстурной памяти извлекается. Как описано в разделе 3.2.10.1.3, перед использованием в ядре она должна быть привязана к некоторому региону памяти, называемому *текстурой*, при помощи функций среды выполнения. Несколько разных ссылок могут быть привязаны к одной текстуре или текстурам, перекрывающимся в памяти.  Ссылка на текстуру имеет несколько атрибутов. Один из них – размерность, указывающая, адресуется ли текстура как одномерный массив с использованием одной *координаты текстуры*, как двухмерный массив с использованием двух координат текстуры или как трехмерный массив с использованием трех координат текстуры. Элементы массива называются текселями (*texels)*, сокращенно от «texture elements» («элементы текстуры»). Возможный тип текселя ограничен следующими: базовое целое, плавающие типы одинарной точности и любой из типов 1-, 2-, и 4-компонентных векторов, определенных в разделе Б.3.1  Другие атрибуты определяют типы входных и выходных данных извлечения текстуры, а также входные координаты и необходимые операции обработки.  Текстурой может быть любой регион линейной памяти или массива CUDA (описан в разделе 3.2.10.2.3).  Таблица Е-2 приводит список максимальных значений ширины, высоты и глубины текстуры в зависимости от вычислительной способности устройства.  Текстуры также могут быть расслоены, как описано в разделе 3.2.10.1.5. |
| 3.2.10.1.1 Texture Reference Declaration  Some of the attributes of a texture reference are immutable and must be known at compile time; they are specified when declaring the texture reference. A texture reference is declared at file scope as a variable of type texture:  texture<DataType, Type, ReadMode> texRef;  where:   * **DataType** specifies the type of data that is returned when fetching the texture; **Type** is restricted to the basic integer and single-precision floating-point types and any of the 1-, 2-, and 4-component vector types defined in Section B.3.1; * **Type** specifies the type of the texture reference and is equal to **cudaTextureType1D**, **cudaTextureType2D**, or **cudaTextureType3D**, for a one-dimensional, two-dimensional, or three-dimensional texture, respectively, or **cudaTextureType1DLayered** or **cudaTextureType2DLayered** for a one-dimensional or two-dimensional layered texture respectively; **Type** is an optional argument which defaults to **cudaTextureType1D**; * **ReadMode** is equal to **cudaReadModeNormalizedFloat** or **cudaReadModeElementType**; if it is **cudaReadModeNormalizedFloat** and **Type** is a 16-bit or 8-bit integer type, the value is actually returned as floating-point type and the full range of the integer type is mapped to [0.0, 1.0] for unsigned integer type and [-1.0, 1.0] for signed integer type; for example, an unsigned 8-bit texture element with the value 0xff reads as 1; if it is **cudaReadModeElementType**, no conversion is performed; **ReadMode** is an optional argument which defaults to **cudaReadModeElementType**.   A texture reference can only be declared as a static global variable and cannot be passed as an argument to a function. | 3.2.10.1.1 Определение ссылки на текстуру  Некоторые атрибуты ссылки на текстуру должны быть обязательно определены, так как должны быть известны во время компиляции; они указываются при объявлении ссылки на текстуру. Ссылка на текстуру объявляется в области видимости файла как переменная типа текстуры:  texture<DataType, Type, ReadMode> texRef;  где:   * **DataType** указывает тип данных, возвращаемый при получении текстуры; **Type** ограничен следующими: базовое целое, плавающие типы одинарной точности и любой из типов 1-, 2-, и 4-компонентных векторов, определенных в разделе Б.3.1; * **Type** указывает тип ссылки на текстуру и равен **cudaTextureType1D**, **cudaTextureType2D** или **cudaTextureType3D** для одно-, двух- или трехмерного массива, соответственно, или **cudaTextureType1DLayered** или **cudaTextureType2DLayered** для одно- или двухмерной слоеной текстуры, соответственно; **Type** необязателен для указания, по умолчанию равен **cudaTextureType1D**; * **ReadMode** равен **cudaReadModeNormalizedFloat** или **cudaReadModeElementType**; если он равен **cudaReadModeNormalizedFloat** и **Type** является 16- или 8-битным целым, значение фактически возвращается как плавающее, и весь диапазон целого типа отображается на [0.0, 1.0] для беззнакового целого и [-1.0, 1.0] для целого со знаком; например, беззнаковый 8-битный элемент текстуры со значением 0xff читается как 1; если он равен **cudaReadModeElementType**, преобразование не производится; **ReadMode** необязателен для указания, по умолчанию равен **cudaReadModeElementType**.   Ссылка на текстуру может быть объявлена только как статическая глобальная переменная и не может быть передана как аргумент в функцию. |
| 3.2.10.1.2 Runtime Texture Reference Attributes  The other attributes of a texture reference are mutable and can be changed at runtime through the host runtime. They specify whether texture coordinates are normalized or not, the addressing mode, and texture filtering, as detailed below.  By default, textures are referenced (by the functions of Section B.8) using floating-point coordinates in the range [0, N-1] where N is the size of the texture in the dimension corresponding to the coordinate. For example, a texture that is 6432 in size will be referenced with coordinates in the range [0, 63] and [0, 31] for the x and y dimensions, respectively. Normalized texture coordinates cause the coordinates to be specified in the range [0.0, 1.0-1/N] instead of [0, N-1], so the same 6432 texture would be addressed by normalized coordinates in the range [0, 1-1/N] in both the x and y dimensions. Normalized texture coordinates are a natural fit to some applications’ requirements, if it is preferable for the texture coordinates to be independent of the texture size.  It is valid to call the device functions of Section B.8 with coordinates that are be out of range. The addressing mode defines what happens if that case. The default addressing mode is to clamp the coordinates to the valid range: [0, N) for non-normalized coordinates and [0.0, 1.0) for normalized coordinates. If the border mode is specified instead, texture fetches with out-of-range texture coordinates return zero. For normalized coordinates, the warp mode and the mirror mode are also available. When using the wrap mode, each coordinate *x* is converted to *frac(x)=x-floor(x*. where *floor(x)* is the largest integer not greater than *x*. When using the mirror mode, each coordinate *x* is converted to *frac(x)* if *floor(x)* is even and *1-frac(x)* if *floor(x)* is odd.  Linear texture filtering may be done only for textures that are configured to return floating-point data. It performs low-precision interpolation between neighboring texels. When enabled, the texels surrounding a texture fetch location are read and the return value of the texture fetch is interpolated based on where the texture coordinates fell between the texels. Simple linear interpolation is performed for one-dimensional textures, bilinear interpolation for two-dimensional textures, and trilinear interpolation for three-dimensional textures.  Appendix E gives more details on texture fetching. | 3.2.10.1.2 Атрибуты времени выполнения для ссылки на текстуру  Другие атрибуты ссылки на текстуру необязательны для указания и могут быть изменены во время выполнения через среду выполнения хоста. Они указывают нормализованность координат текстуры, режим адресации и фильтрование текстур, как описано ниже.  По умолчанию ссылки на текстуры (с помощью функций раздела Б.8) являются плавающими координатами в диапазоне [0, N-1], где N – это размер текстуры в измерении, соответствующем координате. Например, текстура размера 64x32 будет иметь ссылку с координатами в диапазонах [0, 63] и [0, 31] для измерений x и y, соответственно. Нормализованные координаты текстуры указываются в диапазоне [0.0, 1.0-1/N] вместо [0, N-1], так что та же текстура размера 64x32 будет иметь координаты в диапазоне [0, 1-1/N] для обоих измерений x и y. Нормализованные координаты текстуры естественны для требований некоторых приложений, если требуется, чтобы координаты текстуры были независимы от ее размера.  Допускается вызывать функции устройства из раздела Б.8 с координатами, выходящими за пределы массива. Режим адресации определяет, что при этом происходит. Режим адресации по умолчанию предполагает ввод координат в допустимый диапазон: [0, N) для ненормализованных координат и [0.0, 1.0) для нормализованных. Если же указан граничный режим, текстура извлекается с координатами, не входящими в границы массива, возвращающими ноль. Для нормализованных координат также доступны режимы деформации и зеркального отражения. В режиме деформации каждая координата *x* преобразуется в *frac(x)=x-floor(x)*, где *floor(x)* это наибольшее целое, не превышающее *x*. При использовании режима зеркального отражения каждая координата *x* преобразуется в  *frac(x)*, если *floor(x)* четное, и в *1-frac(x)*, если *floor(x)* нечетное.  Линейная фильтрация текстур может быть проведена для текстур, настроенных на возвращение плавающих данных. Она производит интерполяцию низкой точности между соседними текселями. При её задействовании тексели, окружающие извлекаемую текстуру, читаются, и возвращенное значение приближается на основе положения координат текстуры между текселями. Простая линейная интерполяция производится для одномерных текстур, билинейная интерполяция – для двухмерных, и трилинейная – для трехмерных.  Приложение Д содержит дополнительную информацию о получении текстур. |
| 3.2.10.1.3 Texture Binding  As explained in the reference manual, the runtime API has a *low-level* C-style interface and a *high-level* C++-style interface. The **texture** type is defined in the high-level API as a structure publicly derived from the **textureReference** type defined in the low-level API as such:  struct textureReference {  int normalized;  enum cudaTextureFilterMode filterMode;  enum cudaTextureAddressMode addressMode[3];  struct cudaChannelFormatDesc channelDesc;  }   * **normalized** specifies whether texture coordinates are normalized or not, as described in Section 3.2.10.1.2; * **filterMode** specifies the filtering mode, that is how the value returned when fetching the texture is computed based on the input texture coordinates; **filterMode** is equal to **cudaFilterModePoint** or **cudaFilterModeLinear**; if it is **cudaFilterModePoint**, the returned value is the texel whose texture coordinates are the closest to the input texture coordinates; if it is **cudaFilterModeLinear**, the returned value is the linear interpolation of the two (for a one-dimensional texture), four (for a two-dimensional texture), or eight (for a three-dimensional texture) texels whose texture coordinates are the closest to the input texture coordinates; **cudaFilterModeLinear** is only valid for returned values of floating-point type; * **addressMode** specifies the addressing mode, as described in Section 3.2.10.1.2; **addressMode** is an array of size three whose first, second, and third elements specify the addressing mode for the first, second, and third texture coordinates, respectively; the addressing mode are **cudaAddressModeBorder**, **cudaAddressModeClamp**, **cudaAddressModeWrap**, and **cudaAddressModeMirror; cudaAddressModeWrap** and **cudaAddressModeMirror** are only supported for normalized texture coordinates; * **channelDesc** describes the format of the value that is returned when fetching the texture; **channelDesc** is of the following type:   struct cudaChannelFormatDesc {  int x, y, z, w;  enum cudaChannelFormatKind f;  };  where **x**, **y**, **z**, and **w** are equal to the number of bits of each component of the returned value and **f** is:  **cudaChannelFormatKindSigned** if these components are of signed integer type,  **cudaChannelFormatKindUnsigned** if they are of unsigned integer type,  **cudaChannelFormatKindFloat** if they are of floating point type.  **normalized**, **addressMode**, and **filterMode** may be directly modified in host code.  Before a kernel can use a texture reference to read from texture memory, the texture reference must be bound to a texture using **cudaBindTexture()** or **cudaBindTexture2D()** for linear memory, or **cudaBindTextureToArray()** for CUDA arrays. **cudaUnbindTexture()** is used to unbind a texture reference. It is recommended to allocate two-dimensional textures in linear memory using **cudaMallocPitch()** and use the pitch returned by **cudaMallocPitch()** as input parameter to **cudaBindTexture2D()**.  The following code samples bind a texture reference to linear memory pointed to by **devPtr**:   * Using the low-level API:   texture<float, cudaTextureType2D,  cudaReadModeElementType> texRef;  textureReference\* texRefPtr;  cudaGetTextureReference(&texRefPtr, "texRef");  cudaChannelFormatDesc channelDesc =  cudaCreateChannelDesc<float>();  size\_t offset;  cudaBindTexture2D(&offset, texRefPtr, devPtr, &channelDesc,  width, height, pitch);   * Using the high-level API:   texture<float, cudaTextureType2D,  cudaReadModeElementType> texRef;  cudaChannelFormatDesc channelDesc =  cudaCreateChannelDesc<float>();  size\_t offset;  cudaBindTexture2D(&offset, texRef, devPtr, channelDesc,  width, height, pitch);  The following code samples bind a texture reference to a CUDA array **cuArray**:   * Using the low-level API:   texture<float, cudaTextureType2D,  cudaReadModeElementType> texRef;  textureReference\* texRefPtr;  cudaGetTextureReference(&texRefPtr, "texRef");  cudaChannelFormatDesc channelDesc;  cudaGetChannelDesc(&channelDesc, cuArray);  cudaBindTextureToArray(texRef, cuArray, &channelDesc);   * Using the high-level API:   texture<float, cudaTextureType2D,  cudaReadModeElementType> texRef;  cudaBindTextureToArray(texRef, cuArray);  The format specified when binding a texture to a texture reference must match the parameters specified when declaring the texture reference; otherwise, the results of texture fetches are undefined.  The following code sample applies some simple transformation kernel to a texture.  // 2D float texture  texture<float, cudaTextureType2D, cudaReadModeElementType> texRef;  // Simple transformation kernel  \_\_global\_\_ void transformKernel(float\* output,  int width, int height, float theta)  {  // Calculate normalized texture coordinates  unsigned int x = blockIdx.x \* blockDim.x + threadIdx.x;  unsigned int y = blockIdx.y \* blockDim.y + threadIdx.y;  float u = x / (float)width;  float v = y / (float)height;  // Transform coordinates  u -= 0.5f;  v -= 0.5f;  float tu = u \* cosf(theta) – v \* sinf(theta) + 0.5f;  float tv = v \* cosf(theta) + u \* sinf(theta) + 0.5f;  // Read from texture and write to global memory  output[y \* width + x] = tex2D(texRef, tu, tv);  }  // Host code  int main()  {  // Allocate CUDA array in device memory  cudaChannelFormatDesc channelDesc =  cudaCreateChannelDesc(32, 0, 0, 0,  cudaArray\* cuArray;  cudaMallocArray(&cuArray, &channelDesc, width, height);  // Copy to device memory some data located at address h\_data  // in host memory  cudaMemcpyToArray(cuArray, 0, 0, h\_data, size,  cudaMemcpyHostToDevice);  // Set texture parameters  texRef.addressMode[0] = cudaAddressModeWrap;  texRef.addressMode[1] = cudaAddressModeWrap;  texRef.filterMode = cudaFilterModeLinear;  texRef.normalized = true;  // Bind the array to the texture reference  cudaBindTextureToArray(texRef, cuArray, channelDesc);  // Allocate result of transformation in device memory  float\* output;  cudaMalloc(&output, width \* height \* sizeof(float));  // Invoke kernel  dim3 dimBlock(16, 16);  dim3 dimGrid((width + dimBlock.x – 1) / dimBlock.x,  (height + dimBlock.y – 1) / dimBlock.y);  transformKernel<<<dimGrid, dimBlock>>>(output, width, height,  angle);  // Free device memory  cudaFreeArray(cuArray);  cudaFree(output);  return 0;  } | 3.2.10.1.3 Связывание текстур  Как объясняется в справочном руководстве, API среды выполнения имеет *низкоуровневый* интерфейс в стиле C и высокоуровневый интерфейс в стилеC++. Тип **texture** определяется в высокоуровневом API как структура, открыто полученная из типа **textureReference**, определенного в низкоуровневом API так:  struct textureReference {  int normalized;  enum cudaTextureFilterMode filterMode;  enum cudaTextureAddressMode addressMode[3];  struct cudaChannelFormatDesc channelDesc;  }   * **normalized** указывает, нормализованы ли координаты текстуры, что описано в разделе 3.2.10.1.2; * **filterMode** указывает режим фильтрации, т.е. способ возвращения значения, когда извлечение текстуры происходит на основании входных координат текстуры; **filterMode** равно **cudaFilterModePoint** или **cudaFilterModeLinear**; если она равняется **cudaFilterModePoint**, возвращаемое значение – это тексель, координаты текстуры которого наиболее близки к входным координатам текстуры; если она равняется **cudaFilterModeLinear**, возвращаемое значение – это линейная интерполяция двух (для одномерной текстуры), четырех (для двухмерной текстуры) или восьми (для трехмерной текстуры) текселей, текстурные координаты которых наиболее близки к входным координатам; **cudaFilterModeLinear** действительна только для возвращаемых значений плавающего типа; * **addressMode** указывает режим адресации, как описано в разделе 3.2.10.1.2; **addressMode –** это массив из трех элементов, каждый из которых указывает соответственно режимы адресации для первой, второй и третьей координат текстуры; режимы адресации следующие: **cudaAddressModeBorder**, **cudaAddressModeClamp**, **cudaAddressModeWrap** и **cudaAddressModeMirror; cudaAddressModeWrap** и **cudaAddressModeMirror** поддерживаются только для нормализованных координат текстуры; * **channelDesc** описывает формат величины, возвращаемой при извлечении текстуры; **channelDesc** имеет следующий вид:   struct cudaChannelFormatDesc {  int x, y, z, w;  enum cudaChannelFormatKind f;  };  где **x**, **y**, **z** и **w** равны числу бит каждого компонента возвращаемой величины и **f** равна:  **cudaChannelFormatKindSigned**, если эти компоненты имеют тип целого со знаком,  **cudaChannelFormatKindUnsigned**, если они имеют тип беззнакового целого,  **cudaChannelFormatKindFloat**, если они имеют плавающий тип.  **normalized**, **addressMode** и **filterMode** могут быть непосредственно изменены в коде хоста.  Прежде чем ядро сможет использовать ссылку на текстуру для чтения из текстурной памяти, ссылка на текстуру должна быть привязана к текстуре с использованием **cudaBindTexture()** или **cudaBindTexture2D()**для линейной памяти или **cudaBindTextureToArray()** для массивов CUDA. **cudaUnbindTexture()** используется для удаления привязки ссылки к текстуре. Рекомендуется выделять память для двухмерных текстур в линейной памяти с использованием **cudaMallocPitch()** и использовать шаг, возвращенный **cudaMallocPitch()**, как входной параметр для **cudaBindTexture2D()**.  Следующие примеры кода привязывают ссылку на текстуру к линейной памяти с использованием указателя **devPtr**:   * При помощи низкоуровневого API:   texture<float, cudaTextureType2D,  cudaReadModeElementType> texRef;  textureReference\* texRefPtr;  cudaGetTextureReference(&texRefPtr, "texRef");  cudaChannelFormatDesc channelDesc =  cudaCreateChannelDesc<float>();  size\_t offset;  cudaBindTexture2D(&offset, texRefPtr, devPtr, &channelDesc,  width, height, pitch);   * При помощи высокоуровневого API:   texture<float, cudaTextureType2D,  cudaReadModeElementType> texRef;  cudaChannelFormatDesc channelDesc =  cudaCreateChannelDesc<float>();  size\_t offset;  cudaBindTexture2D(&offset, texRef, devPtr, channelDesc,  width, height, pitch);  Следующие примеры кода привязывают ссылку на текстуру к массиву CUDA **cuArray**:   * При помощи низкоуровневого API:   texture<float, cudaTextureType2D,  cudaReadModeElementType> texRef;  textureReference\* texRefPtr;  cudaGetTextureReference(&texRefPtr, "texRef");  cudaChannelFormatDesc channelDesc;  cudaGetChannelDesc(&channelDesc, cuArray);  cudaBindTextureToArray(texRef, cuArray, &channelDesc);   * При помощи высокоуровневого API:   texture<float, cudaTextureType2D,  cudaReadModeElementType> texRef;  cudaBindTextureToArray(texRef, cuArray);  Формат, указанный при привязке ссылки на текстуру к текстуре, должен совпадать с параметрами, указанными при объявлении ссылки на текстуру; иначе результаты извлечения текстуры будут неопределенными.  Следующий пример кода применяет некоторые простые ядра преобразования к текстуре.  // Двумерная плавающая текстура  texture<float, cudaTextureType2D, cudaReadModeElementType> texRef;  // Простое ядро преобразования  \_\_global\_\_ void transformKernel(float\* output,  int width, int height, float theta)  {  // Вычисляем нормализованные координаты текстуры  unsigned int x = blockIdx.x \* blockDim.x + threadIdx.x;  unsigned int y = blockIdx.y \* blockDim.y + threadIdx.y;  float u = x / (float)width;  float v = y / (float)height;  // Преобразуем координаты  u -= 0.5f;  v -= 0.5f;  float tu = u \* cosf(theta) – v \* sinf(theta) + 0.5f;  float tv = v \* cosf(theta) + u \* sinf(theta) + 0.5f;  // Читаем из текстуры и пишем в глобальную память  output[y \* width + x] = tex2D(texRef, tu, tv);  }  // Код хоста  int main()  {  // Выделяем память устройства под массив CUDA  cudaChannelFormatDesc channelDesc =  cudaCreateChannelDesc(32, 0, 0, 0,  cudaArray\* cuArray;  cudaMallocArray(&cuArray, &channelDesc, width, height);  // Копируем некоторую часть данных, размещенных по адресу h\_data, в память устройства  cudaMemcpyToArray(cuArray, 0, 0, h\_data, size,  cudaMemcpyHostToDevice);  // Устанавливаем параметры текстуры  texRef.addressMode[0] = cudaAddressModeWrap;  texRef.addressMode[1] = cudaAddressModeWrap;  texRef.filterMode = cudaFilterModeLinear;  texRef.normalized = true;  // Привязываем ссылку на текстуру к массиву cudaBindTextureToArray(texRef, cuArray, channelDesc);  // Размещаем результат преобразования в памяти  float\* output;  cudaMalloc(&output, width \* height \* sizeof(float));  // Вызываем ядро  dim3 dimBlock(16, 16);  dim3 dimGrid((width + dimBlock.x – 1) / dimBlock.x,  (height + dimBlock.y – 1) / dimBlock.y);  transformKernel<<<dimGrid, dimBlock>>>(output, width, height,  angle);  // Освобождаем память устройства  cudaFreeArray(cuArray);  cudaFree(output);  return 0;  } |
| 3.2.10.1.4 16-Bit Floating-Point Textures  The 16-bit floating-point or *half* format supported by CUDA arrays is the same as the IEEE 754-2008 binary2 format.  CUDA C does not support a matching data type, but provides intrinsic functions to convert to and from the 32-bit floating-point format via the **unsigned short** type: **\_\_float2half\_rn(float)** and **\_\_half2float(unsigned short)**. These functions are only supported in device code. Equivalent functions for the host code can be found in the OpenEXR library, for example.  16-bit floating-point components are promoted to 32 bit float during texture fetching before any filtering is performed.  A channel description for the 16-bit floating-point format can be created by calling one of the **cudaCreateChannelDescHalf\*()** functions. | 3.2.10.1.4 16-битные плавающие текстуры  16-битный плавающий или *половинный* (*half*)формат, поддерживаемый массивами CUDA, аналогичен формату binary2 в IEEE 754-2008.  CUDA C не поддерживает аналогичный формат данных, но предоставляет встроенные функции преобразования в и из 32-битного плавающего формата с помощью типа **unsigned short**: **\_\_float2half\_rn(float)** и **\_\_half2float(unsigned short)**. Эти функции поддерживаются только кодом устройства. Эквивалентные функции для кода хоста могут быть найдены, например, в библиотеке OpenEXR.  16-битные плавающие компоненты повышаются до 32-битных плавающих во время извлечения текстуры перед выполнением любой фильтрации.  Описание канала для 16-битного плавающего формата может быть создано вызовом одной из функций **cudaCreateChannelDescHalf\*()**. |
| 3.2.10.1.5 Layered Textures  A one-dimensional or two-dimensional *layered texture* (also known as *texture array* in Direct3D and *array texture* in OpenGL) is a texture made up of a sequence of layers, all of which are regular textures of same dimensionality, size, and data type.  A one-dimensional layered texture is addressed using an integer index and a floating-point texture coordinate; the index denotes a layer within the sequence and the coordinate addresses a texel within that layer. A two-dimensional layered texture is addressed using an integer index and two floating-point texture coordinates; the index denotes a layer within the sequence and the coordinates address a texel within that layer.  A layered texture can only be bound to a CUDA array created by calling **cudaMalloc3DArray()** with the **cudaArrayLayered** flag (and a height of zero for one-dimensional layered texture).  Layered textures are fetched using the device functions described in Sections B.8.5 and B.8.6. Texture filtering (see Appendix E) is done only within a layer, not across layers.  Layered textures are only supported on devices of compute capability 2.0 and higher. | 3.2.10.1.5 Слоеные текстуры  Одно- или двухмерная *слоеная текстура* (также известная как *массив текстур* в Direct3D и *массивная текстура* в OpenGL) это текстура, состоящая из последовательности слоев, каждый из которых является обычной текстурой, причем все они имеют одну размерность, размер и тип данных.  Одномерная слоеная текстура адресуется с использованием целого индекса и плавающей координаты текстуры; индекс обозначает слой последовательности, а координата – тексель внутри этого слоя. Двухмерная слоеная текстура адресуется с использованием целого индекса и двух плавающих координат текстуры; индекс обозначает слой последовательности, а координаты указывают на тексель внутри этого слоя.  Слоеная текстура может быть привязана только к массиву CUDA, созданному вызовом **cudaMalloc3DArray()** с флагом **cudaArrayLayered** (и нулевой высотой для одномерной слоеной текстуры).  Слоеные текстуры извлекаются с использованием функций устройства, описанных в разделах Б.8.5 и Б.8.6. Фильтрация текстуры (см. приложение Д) производится только внутри одного, а не нескольких слоев.  Слоеные текстуры поддерживаются только устройствами вычислительной способности 2.0 и выше. |
| 3.2.10.1.6 Cubemap Textures  A *cubemap* texture is a special type of two-dimensional layered texture that has six layers representing the faces of a cube:   * The width of a layer is equal to its height. * The cubemap is addressed using three texture coordinates *x*, *y*, and *z* that are interpreted as a direction vector emanating from the center of the cube and pointing to one face of the cube and a texel within the layer corresponding to that face. More specifically, the face is selected by the coordinate with largest magnitude *m* and the corresponding layer is addressed using coordinates *(s/m+1)/2* and *(t/m+1)/2* where s and *t* are defined in Table 3-1.   *Table 3-1. Cubemap Fetch*    A layered texture can only be bound to a CUDA array created by calling **cudaMalloc3DArray()** with the **cudaArrayCubemap** flag.  Cubemap textures are fetched using the device function described in Sections B.8.7.  Cubemap textures are only supported on devices of compute capability 2.0 and higher. | 3.2.10.1.6 Текстуры куб-отображения  Текстура куб-отображенияэто особый тип двухмерной слоеной текстуры, имеющий шесть слоев, представляющих грани куба.   * Ширина слоя равна его высоте. * Куб-отображение адресуется с использованием трех координат текстуры *x*, *y*, и *z*, интерпретируемых как направляющий вектор, исходящий из центра куба и указывающий на одну из граней куба и тексель внутри слоя, соответствующего этой грани. Точнее, грань выбирается координатой с наибольшей величиной *m*, а соответствующий слой указывается с помощью координат *(s/m+1)/2* и *(t/m+1)/*2, где s и *t* такие, как указано в таблице 3-1.   *Таблица 3-1. Извлечение куб-отображения*    Слоеная текстура может быть привязана только к массиву CUDA, созданному вызовом **cudaMalloc3DArray()** с флагом **cudaArrayCubemap**.  Структуры куб-отображения извлекаются с помощью функций устройства, описанных в разделе Б.8.7.  Структуры куб-отображения поддерживаются только устройствами вычислительной способности 2.0 и выше. |
| 3.2.10.1.7 Cubemap Layered Textures  A *cubemap layered* texture is a layered texture whose layers are cubemaps of same dimension.  A cubemap layered texture is addressed using an integer index and three floating-point texture coordinates; the index denotes a cubemap within the sequence and the coordinates address a texel within that cubemap.  A layered texture can only be bound to a CUDA array created by calling **cudaMalloc3DArray()** with the **cudaArrayLayered** and **cudaArrayCubemap** flags.  Cubemap layered textures are fetched using the device function described in Sections B.8.8. Texture filtering (see Appendix E) is done only within a layer, not across layers.  Cubemap layered textures are only supported on devices of compute capability 2.0 and higher. | 3.2.10.1.7 Слоеные текстуры куб-отображения  *Слоеная текстура куб-отображения* это слоеная текстура, слои которой представляют собой куб-отображения одинаковой размерности.  Структура куб-отображения адресуется с помощью целого индекса и трех плавающих координат текстуры; индекс обозначает куб-отображение внутри последовательности, а координаты указывают тексель внутри этого куб-отображения.  Слоеная текстура может быть привязана только к массиву CUDA, созданному вызовом **cudaMalloc3DArray()** с флагами **cudaArrayLayered** и **cudaArrayCubemap**.  Слоеные текстуры куб-отображения извлекаются с использованием функций устройства, описанных в разделе Б.8.8. Фильтрация текстуры (см. приложение Д) производится только внутри одного, а не нескольких слоев.  Слоеные текстуры куб-отображения поддерживаются только устройствами вычислительной способности 2.0 и выше. |
| 3.2.10.1.8 Texture Gather  Texture *gather* is a special texture fetch that is available for two-dimensional textures only. It is performed by the **tex2Dgather()** function, which has the same parameters as **tex2D()**, plus an additional **comp** parameter equal to 0, 1, 2, or 3 (see Section B.8.9). It returns four 32-bit numbers that correspond to the value of the component **comp** of each of the four texels that would have been used for bilinear filtering during a regular texture fetch. For example, if these texels are of values (253, 20, 31, 255), (250, 25, 29, 254), (249, 16, 37, 253), (251, 22, 30, 250), and **comp** is 2, **tex2Dgather()** returns (31, 29, 37, 30).  Texture gather is only supported for CUDA arrays created with the **cudaArrayTextureGather** flag and of width and height less than the maximum specified in Table F-2 for texture gather, which is smaller than for regular texture fetch.  Texture gather is only supported on devices of compute capability 2.0 and higher. | 3.2.10.1.8 Сбор текстур  *Сбор* текстурэто особый способ извлечения текстур, доступный только для двухмерных текстур. Он выполняется функцией **tex2Dgather()**, имеющей те же параметры, что и **tex2D()**, а также дополнительный параметр **comp**, равный 0, 1, 2 или 3 (см. раздел Б.8.9). Она возвращает четыре 32-битных числа, соответствующих величине компонента **comp** каждого из четырех текселей, которые использовались бы для билинейной фильтрации во время обычного извлечения текстур. Например, если эти тексели равны (253, 20, 31, 255), (250, 25, 29, 254), (249, 16, 37, 253), (251, 22, 30, 250), а **comp** равно 2, **tex2Dgather()** возвратит (31, 29, 37, 30).  Сбор текстур поддерживается только для массивов CUDA, созданным с флагом **cudaArrayTextureGather** ширины и высоты меньше максимальных, указанных в таблице Е-2, которые, в свою очередь, меньше используемых при обычном извлечении текстур.  Сбор текстур поддерживается только устройствами вычислительной способности 2.0 и выше. |
| 3.2.10.2 Surface Memory  For devices of compute capability 2.0 and higher, a CUDA array (described in Section 3.2.10.2.3), created with the **cudaArraySurfaceLoadStore** flag, can be read and written via a *surface reference* using the functions described in Section B.9.  Table F-2 lists the maximum surface width, height, and depth depending on the compute capability of the device.  3.2.10.2.1 Surface Reference Declaration  A surface reference is declared at file scope as a variable of type **surface**:  surface<void, Type> surfRef;  where **Type** specifies the type of the surface reference and is equal to **cudaSurfaceType1D**, **cudaSurfaceType2D**, **cudaSurfaceType3D**, **cudaSurfaceTypeCubemap**, **cudaSurfaceType1DLayered**, **cudaSurfaceType2DLayered**, or **cudaSurfaceTypeCubemapLayered**; **Type** is an optional argument which defaults to **cudaSurfaceType1D**.  A surface reference can only be declared as a static global variable and cannot be passed as an argument to a function.  3.2.10.2.2 Surface Binding  Before a kernel can use a surface reference to access a CUDA array, the surface reference must be bound to the CUDA array using **cudaBindSurfaceToArray()**.  The following code samples bind a surface reference to a CUDA array **cuArray**:   * Using the low-level API:   surface<void, cudaSurfaceType2D> surfRef;  surfaceReference\* surfRefPtr;  cudaGetSurfaceReference(&surfRefPtr, "surfRef");  cudaChannelFormatDesc channelDesc;  cudaGetChannelDesc(&channelDesc, cuArray);  cudaBindSurfaceToArray(surfRef, cuArray, &channelDesc);   * Using the high-level API:   surface<void, cudaSurfaceType2D> surfRef;  cudaBindSurfaceToArray(surfRef, cuArray);  A CUDA array must be read and written using surface functions of matching dimensionality and type and via a surface reference of matching dimensionality; otherwise, the results of reading and writing the CUDA array are undefined.  Unlike texture memory, surface memory uses byte addressing. This means that the x-coordinate used to access a texture element via texture functions needs to be multiplied by the byte size of the element to access the same element via a surface function. For example, the element at texture coordinate x of a one-dimensional floating-point CUDA array bound to a texture reference **texRef** and a surface reference **surfRef** is read using **tex1d(texRef, x)** via **texRef**, but **surf1Dread(surfRef, 4\*x)** via **surfRef**. Similarly, the element at texture coordinate x and y of a two-dimensional floating-point CUDA array bound to a texture reference **texRef** and a surface reference **surfRef** is accessed using **tex2d(texRef, x, y)** via **texRef**, but **surf2Dread(surfRef, 4\*x, y)** via **surfRef** (the byte offset of the y-coordinate is internally calculated from the underlying line pitch of the CUDA array).  The following code sample applies some simple transformation kernel to a texture.  // 2D surfaces  surface<void, 2> inputSurfRef;  surface<void, 2> outputSurfRef;  // Simple copy kernel  \_\_global\_\_ void copyKernel(int width, int height)  {  // Calculate surface coordinates  unsigned int x = blockIdx.x \* blockDim.x + threadIdx.x;  unsigned int y = blockIdx.y \* blockDim.y + threadIdx.y;  if (x < width && y < height) {  uchar4 data;  // Read from input surface  surf2Dread(&data, inputSurfRef, x \* 4, y);  // Write to output surface  surf2Dwrite(data, outputSurfRef, x \* 4, y);  }  }  // Host code  int main()  {  // Allocate CUDA arrays in device memory  cudaChannelFormatDesc channelDesc =  cudaCreateChannelDesc(8, 8, 8, 8,  cudaChannelFormatKindUnsigned);  cudaArray\* cuInputArray;  cudaMallocArray(&cuInputArray, &channelDesc, width, height,  cudaArraySurfaceLoadStore);  cudaArray\* cuOutputArray;  cudaMallocArray(&cuOutputArray, &channelDesc, width, height,  cudaArraySurfaceLoadStore);  // Copy to device memory some data located at address h\_data  // in host memory  cudaMemcpyToArray(cuInputArray, 0, 0, h\_data, size,  cudaMemcpyHostToDevice);  // Bind the arrays to the surface references  cudaBindSurfaceToArray(inputSurfRef, cuInputArray);  cudaBindSurfaceToArray(outputSurfRef, cuOutputArray);  // Invoke kernel  dim3 dimBlock(16, 16);  dim3 dimGrid((width + dimBlock.x – 1) / dimBlock.x,  (height + dimBlock.y – 1) / dimBlock.y);  copyKernel<<<dimGrid, dimBlock>>>(width, height);  // Free device memory  cudaFreeArray(cuInputArray);  cudaFreeArray(cuOutputArray);  return 0;  }  3.2.10.2.3 Cubemap Surfaces  Cubemap surfaces are accessed using **surfCubemapread()** and **surfCubemapwrite()** (Sections B.9.11 and B.9.12) as a two-dimensional layered surface, i.e. using an integer index denoting a face and two floating-point texture coordinates addressing a texel within the layer corresponding to this face. Faces are ordered as indicated in Table 3-1.  3.2.10.2.4 Cubemap Layered Surfaces  Cubemap layered surfaces are accessed using **surfCubemapLayeredread()** and **surfCubemapLayeredwrite()** (Sections B.9.13 and B.9.14) as a two-dimensional layered surface, i.e. using an integer index denoting a face of one of the cubemaps and two floating-point texture coordinates addressing a texel within the layer corresponding to this face. Faces are ordered as indicated in Table 3-1, so index ((2 \* 6) + 3), for example, accesses the fourth face of the third cubemap. | 3.2.10.2 Поверхностная память  Для устройств вычислительной способности 2.0 и выше массив CUDA (описанный в разделе 3.2.10.2.3), созданный с флагом **cudaArraySurfaceLoadStore**, может быть прочитан и записан через *поверхностную ссылку* с использованием функций, описанных в Б.9.  Таблица Е-2 приводит список максимальных значений ширины, высоты и глубины поверхности в зависимости от вычислительной способности устройства.  3.2.10.2.1 Объявление ссылки на поверхность  Ссылка на поверхность объявляется в области видимости файла как переменная типа **surface**:  surface<void, Type> surfRef;  где **Type** указывает тип ссылки на поверхность и равен **cudaSurfaceType1D**, **cudaSurfaceType2D**, **cudaSurfaceType3D**, **cudaSurfaceTypeCubemap**, **cudaSurfaceType1DLayered**, **cudaSurfaceType2DLayered** или **cudaSurfaceTypeCubemapLayered**; **Type** – необязательный аргумент, по умолчанию устанавливаемый в **cudaSurfaceType1D**.  Ссылка на поверхность может быть объявлена только как статическая глобальная переменная и не может быть передана в качестве аргумента в функцию.  3.2.10.2.2 Привязка поверхностей  Прежде чем ядро сможет использовать ссылку на поверхность для доступа к массиву CUDA, ссылка на поверхность должна быть привязана к массиву CUDA с использованием **cudaBindSurfaceToArray()**.  Следующие примеры кода привязывают ссылку на поверхность к массиву CUDA **cuArray**:   * При помощи низкоуровневого API:   surface<void, cudaSurfaceType2D> surfRef;  surfaceReference\* surfRefPtr;  cudaGetSurfaceReference(&surfRefPtr, "surfRef");  cudaChannelFormatDesc channelDesc;  cudaGetChannelDesc(&channelDesc, cuArray);  cudaBindSurfaceToArray(surfRef, cuArray, &channelDesc);   * При помощи высокоуровневого API:   surface<void, cudaSurfaceType2D> surfRef;  cudaBindSurfaceToArray(surfRef, cuArray);  Массив CUDA должен быть прочитан и записан с использованием функций поверхности соответствующей размерности и типа через ссылку на поверхность соответствующей размерности; иначе результаты чтения и записи массива CUDA будут неопределенными.  В отличие от текстурной памяти, поверхностная использует байтовую адресацию. Это означает, что координата x, использованная для доступа к элементу текстуры через функции текстуры, должна быть помножена на байтовый размер элемента для доступа к тому же элементу через функцию поверхности. Например, обращение к элементу с координатой x одномерного плавающего массива CUDA со связанной ссылкой на текстуру **texRef** и ссылкой на поверхность **surfRef** производится с использованием **tex1d(texRef, x)** через **texRef**, но с использованием **surf1Dread(surfRef, 4\*x)** через **surfRef**. Аналогично, обращение к элементу текстуры с координатами x и y двумерного плавающего массива CUDA со связанной ссылкой на текстуру **texRef** и ссылкой на поверхность **surfRef** производится с помощью **tex2d(texRef, x, y)** через **texRef**, но с использованием **surf2Dread(surfRef, 4\*x, y)** через **surfRef** (байтовый отступ координаты y внутренне вычисляется с помощью лежащего в основе шага массива CUDA).  Следующий пример кода применяет некоторые простые ядра преобразования к текстуре.  // Двумерные поверхности  surface<void, 2> inputSurfRef;  surface<void, 2> outputSurfRef;  // Простое ядро копирования  \_\_global\_\_ void copyKernel(int width, int height)  {  // Вычисляем координаты поверхности  unsigned int x = blockIdx.x \* blockDim.x + threadIdx.x;  unsigned int y = blockIdx.y \* blockDim.y + threadIdx.y;  if (x < width && y < height) {  uchar4 data;  // Читаем из входной поверхности  surf2Dread(&data, inputSurfRef, x \* 4, y);  // Пишем в выходную поверхность  surf2Dwrite(data, outputSurfRef, x \* 4, y);  }  }  // Код хоста  int main()  {  // Разместить массивы CUDA в памяти устройства  cudaChannelFormatDesc channelDesc =  cudaCreateChannelDesc(8, 8, 8, 8,  cudaChannelFormatKindUnsigned);  cudaArray\* cuInputArray;  cudaMallocArray(&cuInputArray, &channelDesc, width, height,  cudaArraySurfaceLoadStore);  cudaArray\* cuOutputArray;  cudaMallocArray(&cuOutputArray, &channelDesc, width, height,  cudaArraySurfaceLoadStore);  // Копируем некоторые данные, расположенные по адресу h\_data в памяти  // хоста, в память устройства  cudaMemcpyToArray(cuInputArray, 0, 0, h\_data, size,  cudaMemcpyHostToDevice);  // Привязываем ссылки на поверхность к массивам  cudaBindSurfaceToArray(inputSurfRef, cuInputArray);  cudaBindSurfaceToArray(outputSurfRef, cuOutputArray);  // Вызываем ядро  dim3 dimBlock(16, 16);  dim3 dimGrid((width + dimBlock.x – 1) / dimBlock.x,  (height + dimBlock.y – 1) / dimBlock.y);  copyKernel<<<dimGrid, dimBlock>>>(width, height);  // Освобождаем память устройства  cudaFreeArray(cuInputArray);  cudaFreeArray(cuOutputArray);  return 0;  }  3.2.10.2.3 Поверхности куб-отображения  Доступ к поверхностям куб-отображения осуществляется через **surfCubemapread()** и **surfCubemapwrite()** (разделы Б.9.11 и Б.9.12) как к двумерным слоеным поверхностям, т.е. с помощью целочисленного индекса, обозначающего грань, и двух плавающих координат текстуры, указывающих на тексель внутри слоя, соответствующего этой грани. Грани упорядочены как показано в таблице 3-1.  3.2.10.2.4 Слоеные поверхности куб-отображения  Доступ к слоеным поверхностям куб-отображения осуществляется через **surfCubemapLayeredread()** и **surfCubemapLayeredwrite()** (разделы Б.9.13 и Б.9.14) как к двухмерным слоеным поверхностям, т.е. с помощью целочисленного индекса, обозначающего грань одного из куб-отображений, и двух плавающих координат текстуры, указывающих на тексель внутри слоя, соответствующего этой грани. Грани упорядочены как показано в таблице 3-1, так что, например, индекс ((2 \* 6) + 3), запршивает четвертую грань третьего куб-отображения. |
| 3.2.10.3 CUDA Arrays  CUDA arrays are opaque memory layouts optimized for texture fetching. They are one-dimensional, two-dimensional, or three-dimensional and composed of elements, each of which has 1, 2 or 4 components that may be signed or unsigned 8-, 16- or 32-bit integers, 16-bit floats, or 32-bit floats. CUDA arrays are only readable by kernels through texture fetching and may only be bound to texture references with the same number of packed components. | 3.2.10.3 Массивы CUDA  Массивы CUDA это непрозрачные схемы памяти, оптимизированные для извлечения текстур. Они одномерны, двухмерны или трехмерны и состоят из элементов, каждый из которых имеет 1, 2 или 4 компонента, которые могут быть со знаком и беззнаковыми 8-, 16- или 32-битными целыми, 16- или 32-битными плавающими. Массивы CUDA доступны для чтения только ядрам через извлечение текстур и могут быть привязаны только к ссылкам на текстуры с тем же числом упакованных компонентов. |
| 3.2.10.4 Read/Write Coherency  The texture and surface memory is cached (see Section 5.3.2.5) and within the same kernel call, the cache is not kept coherent with respect to global memory writes and surface memory writes, so any texture fetch or surface read to an address that has been written to via a global write or a surface write in the same kernel call returns undefined data. In other words, a thread can safely read some texture or surface memory location only if this memory location has been updated by a previous kernel call or memory copy, but not if it has been previously updated by the same thread or another thread from the same kernel call. | 3.2.10.4 Согласованность чтения/записи  Текстурная и поверхностная память кэшируются (см. раздел 5.3.2.5) и внутри одного вызова ядра кэш не является согласованным относительно записи в глобальную и поверхностную память, так что любое получение текстуры или чтение поверхности в адрес, в который запись была произведена через глобальную или поверхностную запись в одном вызове ядра, возвращает неопределённые данные. Другими словами, нить может безопасно читать некоторую часть текстурной или поверхностной памяти только если эта часть памяти была обновлена предыдущим вызовом ядра или копированием памяти, но не в случае, если была ранее обновлена той же нитью или другой нитью из того же вызова ядра. |
| 3.2.11 Graphics Interoperability  Some resources from OpenGL and Direct3D may be mapped into the address space of CUDA, either to enable CUDA to read data written by OpenGL or Direct3D, or to enable CUDA to write data for consumption by OpenGL or Direct3D.  A resource must be registered to CUDA before it can be mapped using the functions mentioned in Sections 3.2.11.1 and 3.2.11.2. These functions return a pointer to a CUDA graphics resource of type **struct cudaGraphicsResource**. Registering a resource is potentially high-overhead and therefore typically called only once per resource. A CUDA graphics resource is unregistered using **cudaGraphicsUnregisterResource()**.  Once a resource is registered to CUDA, it can be mapped and unmapped as many times as necessary using **cudaGraphicsMapResources()** and **cudaGraphicsUnmapResources()**. **cudaGraphicsResourceSetMapFlags()** can be called to specify usage hints (write-only, read-only) that the CUDA driver can use to optimize resource management.  A mapped resource can be read from or written to by kernels using the device memory address returned by **cudaGraphicsResourceGetMappedPointer()** for buffers and **cudaGraphicsSubResourceGetMappedArray()** for CUDA arrays.  Accessing a resource through OpenGL or Direct3D while it is mapped to CUDA produces undefined results.  Sections 3.2.11.1 and 3.2.11.2 give specifics for each graphics API and some code samples.  Section 3.2.11.3 gives specifics for when the system is in SLI mode. | 3.2.11 Функциональная совместимость графики  Некоторые ресурсы OpenGL и Direct3D могут быть отображены в адресное пространство CUDA, либо для задействования возможности CUDA читать данные, записанные в OpenGL или Direct3D, либо для задействования возможности CUDA записывать данные для нужд OpenGL или Direct3D.  Ресурс должен быть зарегистрирован в CUDA прежде чем он может быть отображен с использованием функций, упомянутых в разделах 3.2.11.1 и 3.2.11.2. Эти функции возвращают указатель на графический ресурс CUDA типа **struct cudaGraphicsResource**. Регистрация ресурса потенциально накладна и поэтому обычно вызывается лишь единожды на ресурс. Регистрация графического ресурса CUDA может быть отменена с использованием **cudaGraphicsUnregisterResource()**.  После регистрации ресурса в CUDA, отображение и его отмена могут быть произведены любое количество раз при помощи **cudaGraphicsMapResources()** и **cudaGraphicsUnmapResources()**. **cudaGraphicsResourceSetMapFlags()** может быть вызвана для указания возможностей использования (только для записи, только для чтения), которые драйвер CUDA может использовать для оптимизации управления ресурсами.  Отображенный ресурс можно использовать для записи и чтения ядрами с использованием адреса памяти устройства, возвращаемого **cudaGraphicsResourceGetMappedPointer()** для буферов и **cudaGraphicsSubResourceGetMappedArray()** для массивов CUDA.  Доступ к ресурсу через OpenGL или Direct3D, пока он отображен в CUDA, приводит к неопределенным результатам.  Разделы 3.2.11.1 и 3.2.11.2 дают указания для каждого графического API и некоторые примеры кода.  Раздел 3.2.11.3 дает указания по системам в режиме SLI. |
| 3.2.11.1 OpenGL Interoperability  Interoperability with OpenGL requires that the CUDA device be specified by **cudaGLSetGLDevice()** before any other runtime calls. Note that **cudaSetDevice()** and **cudaGLSetGLDevice()** are mutually exclusive.  The OpenGL resources that may be mapped into the address space of CUDA are OpenGL buffer, texture, and renderbuffer objects.  A buffer object is registered using **cudaGraphicsGLRegisterBuffer()**. In CUDA, it appears as a device pointer and can therefore be read and written by kernels or via **cudaMemcpy()** calls.  A texture or renderbuffer object is registered using **cudaGraphicsGLRegisterImage()**. In CUDA, it appears as a CUDA array. Kernels can read from the array by binding it to a texture or surface reference. They can also write to it via the surface write functions if the resource has been registered with the **cudaGraphicsRegisterFlagsSurfaceLoadStore** flag. The array can also be read and written via **cudaMemcpy2D()** calls. **cudaGraphicsGLRegisterImage()** supports all texture formats with 1, 2, or 4 components and an internal type of float (e.g. **GL\_RGBA\_FLOAT32**), normalized integer (e.g. **GL\_RGBA8**, **GL\_INTENSITY16**), and unnormalized integer (e.g. **GL\_RGBA8UI**) (please note that since unnormalized integer formats require OpenGL 3.0, they can only be written by shaders, not the fixed function pipeline).  The OpenGL context whose resources are being shared has to be current to the host thread making any OpenGL interoperability API calls.  The following code sample uses a kernel to dynamically modify a 2D **width** x **height** grid of vertices stored in a vertex buffer object:  GLuint positionsVBO;  struct cudaGraphicsResource\* positionsVBO\_CUDA;  int main()  {  // Initialize OpenGL and GLUT for device 0  // and make the OpenGL context current  ...  glutDisplayFunc(display);  // Explicitly set device 0  cudaGLSetGLDevice(0);  // Create buffer object and register it with CUDA  glGenBuffers(1, positionsVBO);  glBindBuffer(GL\_ARRAY\_BUFFER, &positionsVBO);  unsigned int size = width \* height \* 4 \* sizeof(float);  glBufferData(GL\_ARRAY\_BUFFER, size, 0, GL\_DYNAMIC\_DRAW);  glBindBuffer(GL\_ARRAY\_BUFFER, 0);  cudaGraphicsGLRegisterBuffer(&positionsVBO\_CUDA,  positionsVBO,  cudaGraphicsMapFlagsWriteDiscard);  // Launch rendering loop  glutMainLoop();  ...  }  void display()  {  // Map buffer object for writing from CUDA  float4\* positions;  cudaGraphicsMapResources(1, &positionsVBO\_CUDA, 0);  size\_t num\_bytes;  cudaGraphicsResourceGetMappedPointer((void\*\*)&positions,  &num\_bytes,  positionsVBO\_CUDA));  // Execute kernel  dim3 dimBlock(16, 16, 1);  dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);  createVertices<<<dimGrid, dimBlock>>>(positions, time,  width, height);  // Unmap buffer object  cudaGraphicsUnmapResources(1, &positionsVBO\_CUDA, 0);  // Render from buffer object  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  glBindBuffer(GL\_ARRAY\_BUFFER, positionsVBO);  glVertexPointer(4, GL\_FLOAT, 0, 0);  glEnableClientState(GL\_VERTEX\_ARRAY);  glDrawArrays(GL\_POINTS, 0, width \* height);  glDisableClientState(GL\_VERTEX\_ARRAY);  // Swap buffers  glutSwapBuffers();  glutPostRedisplay();  }  void deleteVBO()  {  cudaGraphicsUnregisterResource(positionsVBO\_CUDA);  glDeleteBuffers(1, &positionsVBO);  }  \_\_global\_\_ void createVertices(float4\* positions, float time,  unsigned int width, unsigned int height)  {  unsigned int x = blockIdx.x \* blockDim.x + threadIdx.x;  unsigned int y = blockIdx.y \* blockDim.y + threadIdx.y;  // Calculate uv coordinates  float u = x / (float)width;  float v = y / (float)height;  u = u \* 2.0f - 1.0f;  v = v \* 2.0f - 1.0f;  // calculate simple sine wave pattern  float freq = 4.0f;  float w = sinf(u \* freq + time)  \* cosf(v \* freq + time) \* 0.5f;  // Write positions  positions[y \* width + x] = make\_float4(u, w, v, 1.0f);  }  On Windows and for Quadro GPUs, **cudaWGLGetDevice()** can be used to retrieve the CUDA device associated to the handle returned by **wglEnumGpusNV()**. Quadro GPUs offer higher performance OpenGL interoperability than GeForce and Tesla GPUs in a multi-GPU configuration where OpenGL rendering is performed on the Quadro GPU and CUDA computations are performed on other GPUs in the system. | 3.2.11.1 Функциональная совместимость с OpenGL  Функциональная совместимость с OpenGL требует, чтобы устройство CUDA было определено функцией **cudaGLSetGLDevice()** до любых других обращений к среде выполнения. Отметьте, что **cudaSetDevice()** и **cudaGLSetGLDevice()** взаимно исключают друг друга.  Ресурсы OpenGL, которые могут быть отображены в адресное пространство CUDA, это буфер, текстура и буфер-рендер.  Буфер регистрируется с помощью **cudaGraphicsGLRegisterBuffer()**. В CUDA он отображается как указатель на устройство и поэтому может быть использован для чтения и записи устройствами или через **cudaMemcpy()**.  Текстура и буфер-рендер регистрируются с помощью **cudaGraphicsGLRegisterImage()**. В CUDA он отображается как массив CUDA. Ядра могут читать массив при его связывании с ссылкой на текстуру или поверхность. Они также могут писать в него через функции записи поверхности, если ресурс был зарегистрирован с флагом **cudaGraphicsRegisterFlagsSurfaceLoadStore**. Массив также может быть использован для чтения и записи через вызовы **cudaMemcpy2D()**. **cudaGraphicsGLRegisterImage()** поддерживает все форматы текстур с 1, 2 или 4 компонентами и плавающим (например, **GL\_RGBA\_FLOAT32**), нормализованным целым (например, **GL\_RGBA8**, **GL\_INTENSITY16**) и ненормализованным целым (например, **GL\_RGBA8UI**) типами интервала (пожалуйста, отметьте, что так как для использования форматов ненормализованного целого необходим OpenGL 3.0, они могут быть записаны только с помощью шейдеров, но не с помощью фиксированного конвейера функций).  Контекст OpenGL, ресурсы которого разделяются, должен быть текущим по отношению к нити хоста, совершающей вызовы API функциональной совместимости.  Следующий пример кода использует ядро для динамической модификации двумерной сетки вершин размера **width** x **height**, сохраненной в буфере вершин:  GLuint positionsVBO;  struct cudaGraphicsResource\* positionsVBO\_CUDA;  int main()  {  // Инициализируем OpenGL и GLUT для устройства 0  // и делаем контекст OpenGL текущим  ...  glutDisplayFunc(display);  // Явно устанавливаем устройство 0  cudaGLSetGLDevice(0);  // Создаем буфер и регистрируем его в CUDA  glGenBuffers(1, positionsVBO);  glBindBuffer(GL\_ARRAY\_BUFFER, &positionsVBO);  unsigned int size = width \* height \* 4 \* sizeof(float);  glBufferData(GL\_ARRAY\_BUFFER, size, 0, GL\_DYNAMIC\_DRAW);  glBindBuffer(GL\_ARRAY\_BUFFER, 0);  cudaGraphicsGLRegisterBuffer(&positionsVBO\_CUDA,  positionsVBO,  cudaGraphicsMapFlagsWriteDiscard);  // Запускаем цикл рендеринга  glutMainLoop();  ...  }  void display()  {  // Отображаем буфер для записи из CUDA  float4\* positions;  cudaGraphicsMapResources(1, &positionsVBO\_CUDA, 0);  size\_t num\_bytes;  cudaGraphicsResourceGetMappedPointer((void\*\*)&positions,  &num\_bytes,  positionsVBO\_CUDA));  // Исполняем ядро  dim3 dimBlock(16, 16, 1);  dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);  createVertices<<<dimGrid, dimBlock>>>(positions, time,  width, height);  // Отменяем отображение буфера  cudaGraphicsUnmapResources(1, &positionsVBO\_CUDA, 0);  // Отображаем с помощью объекта буфера  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  glBindBuffer(GL\_ARRAY\_BUFFER, positionsVBO);  glVertexPointer(4, GL\_FLOAT, 0, 0);  glEnableClientState(GL\_VERTEX\_ARRAY);  glDrawArrays(GL\_POINTS, 0, width \* height);  glDisableClientState(GL\_VERTEX\_ARRAY);  // Меняем буферы местами  glutSwapBuffers();  glutPostRedisplay();  }  void deleteVBO()  {  cudaGraphicsUnregisterResource(positionsVBO\_CUDA);  glDeleteBuffers(1, &positionsVBO);  }  \_\_global\_\_ void createVertices(float4\* positions, float time,  unsigned int width, unsigned int height)  {  unsigned int x = blockIdx.x \* blockDim.x + threadIdx.x;  unsigned int y = blockIdx.y \* blockDim.y + threadIdx.y;  // Вычисляем координаты uv  float u = x / (float)width;  float v = y / (float)height;  u = u \* 2.0f - 1.0f;  v = v \* 2.0f - 1.0f;  // Вычисляем модель простого синусоидального колебания  float freq = 4.0f;  float w = sinf(u \* freq + time)  \* cosf(v \* freq + time) \* 0.5f;  // Записываем позиции  positions[y \* width + x] = make\_float4(u, w, v, 1.0f);  }  На Windows и для ГП серии Quadro **cudaWGLGetDevice()** может быть использована для получения устройства CUDA, связанного с дескриптором, возвращаемым функцией **wglEnumGpusNV()**. ГП серии Quadro предлагают более высокую производительность при работе в режиме совместимости с OpenGL при многопроцессорной (с большим количеством ГП) конфигурации, где рендеринг OpenGL выполняется на ГП Quadro, и вычисления CUDA выполняются на других ГП системы. |
| 3.2.11.2 Direct3D Interoperability  Direct3D interoperability is supported for Direct3D 9, Direct3D 10, and Direct3D 11.  A CUDA context may interoperate with only one Direct3D device at a time and the CUDA context and Direct3D device must be created on the same GPU. In addition the following considerations must be taken when creating the device: Direct3D 9 devices must be created with **DeviceType** set to **D3DDEVTYPE\_HAL** and **BehaviorFlags** with the **D3DCREATE\_HARDWARE\_VERTEXPROCESSING** flag. Direct3D 10 and Direct3D 11 devices must be created with **DriverType** set to **D3D\_DRIVER\_TYPE\_HARDWARE**.  Interoperability with Direct3D requires that the Direct3D device be specified by **cudaD3D9SetDirect3DDevice()**, **cudaD3D10SetDirect3DDevice()** and **cudaD3D11SetDirect3DDevice()**, before any other runtime calls. **cudaD3D9GetDevice()**, **cudaD3D10GetDevice()**, and **cudaD3D11GetDevice()** can be used to retrieve the CUDA device associated to some adapter.  A set of calls is also available to allow the creation of CUDA contexts with interoperability with Direct3D devices that use NVIDIA SLI in AFR (Alternate Frame Rendering) mode: **cudaD3D[9|10|11]GetDevices()**. A call to **cudaD3D[9|10|11]GetDevices()** can be used to obtain a list of CUDA device handles that can be passed as the (optional) last parameter to **cudaD3D[9|10|11]SetDirect3DDevice()**.  The application has the choice to either create multiple CPU threads, each using a different CUDA context, or a single CPU thread using multiple CUDA context. If using separate CPU threads for each GPU each of the CUDA contexts would be created by the CUDA runtime by calling in a separate CPU thread **cudaD3D[9|10|11]SetDirect3DDevice()** using one of the CUDA device handles returned by **cudaD3D[9|10|11]GetDevices()**.  If using a single CPU thread the CUDA contexts would have to be created using the CUDA driver API context creation functions for interoperability with Direct3D devices that use NVIDIA SLI (**cuD3D[9|10|11]CtxCreateOnDevice()**). The application relies on the interoperability between CUDA driver and runtime APIs (Section G.4), which allows it to call **cuCtxPushCurrent()** and **cuCtxPopCurrent()** to change the CUDA context active at a given time.  The Direct3D resources that may be mapped into the address space of CUDA are Direct3D buffers, textures, and surfaces. These resources are registered using **cudaGraphicsD3D9RegisterResource()**,**cudaGraphicsD3D10RegisterResource()**, and **cudaGraphicsD3D11RegisterResource()**.  The following code sample uses a kernel to dynamically modify a 2D **width** x **height** grid of vertices stored in a vertex buffer object.  **Direct3D 9 Version:**  IDirect3D9\* D3D;  IDirect3DDevice9\* device;  struct CUSTOMVERTEX {  FLOAT x, y, z;  DWORD color;  };  IDirect3DVertexBuffer9\* positionsVB;  struct cudaGraphicsResource\* positionsVB\_CUDA;  int main()  {  // Initialize Direct3D  D3D = Direct3DCreate9(D3D\_SDK\_VERSION);  // Get a CUDA-enabled adapter  unsigned int adapter = 0;  for (; adapter < g\_pD3D->GetAdapterCount(); adapter++) {  D3DADAPTER\_IDENTIFIER9 adapterId;  g\_pD3D->GetAdapterIdentifier(adapter, 0, &adapterId);  int dev;  if (cudaD3D9GetDevice(&dev, adapterId.DeviceName)  == cudaSuccess)  break;  }  // Create device  ...  D3D->CreateDevice(adapter, D3DDEVTYPE\_HAL, hWnd,  D3DCREATE\_HARDWARE\_VERTEXPROCESSING,  &params, &device);  // Register device with CUDA  cudaD3D9SetDirect3DDevice(device);  // Create vertex buffer and register it with CUDA  unsigned int size = width \* height \* sizeof(CUSTOMVERTEX);  device->CreateVertexBuffer(size, 0, D3DFVF\_CUSTOMVERTEX,  D3DPOOL\_DEFAULT, &positionsVB, 0);  cudaGraphicsD3D9RegisterResource(&positionsVB\_CUDA,  positionsVB,  cudaGraphicsRegisterFlagsNone);  cudaGraphicsResourceSetMapFlags(positionsVB\_CUDA,  cudaGraphicsMapFlagsWriteDiscard);  // Launch rendering loop  while (...) {  ...  Render();  ...  }  void Render()  {  // Map vertex buffer for writing from CUDA  float4\* positions;  cudaGraphicsMapResources(1, &positionsVB\_CUDA, 0);  size\_t num\_bytes;  cudaGraphicsResourceGetMappedPointer((void\*\*)&positions,  &num\_bytes,  positionsVB\_CUDA));  // Execute kernel  dim3 dimBlock(16, 16, 1);  dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);  createVertices<<<dimGrid, dimBlock>>>(positions, time,  width, height);  // Unmap vertex buffer  cudaGraphicsUnmapResources(1, &positionsVB\_CUDA, 0);  // Draw and present  ...  }  void releaseVB()  {  cudaGraphicsUnregisterResource(positionsVB\_CUDA);  positionsVB->Release();  }  \_\_global\_\_ void createVertices(float4\* positions, float time,  unsigned int width, unsigned int height)  {  unsigned int x = blockIdx.x \* blockDim.x + threadIdx.x;  unsigned int y = blockIdx.y \* blockDim.y + threadIdx.y;  // Calculate uv coordinates  float u = x / (float)width;  float v = y / (float)height;  u = u \* 2.0f - 1.0f;  v = v \* 2.0f - 1.0f;  // Calculate simple sine wave pattern  float freq = 4.0f;  float w = sinf(u \* freq + time)  \* cosf(v \* freq + time) \* 0.5f;  // Write positions  positions[y \* width + x] =  make\_float4(u, w, v, \_\_int\_as\_float(0xff00ff00));  }  **Direct3D 10 Version:**  ID3D10Device\* device;  struct CUSTOMVERTEX {  FLOAT x, y, z;  DWORD color;  };  ID3D10Buffer\* positionsVB;  struct cudaGraphicsResource\* positionsVB\_CUDA;  int main()  {  // Get a CUDA-enabled adapter  IDXGIFactory\* factory;  CreateDXGIFactory(\_\_uuidof(IDXGIFactory), (void\*\*)&factory);  IDXGIAdapter\* adapter = 0;  for (unsigned int i = 0; !adapter; ++i) {  if (FAILED(factory->EnumAdapters(i, &adapter))  break;  int dev;  if (cudaD3D10GetDevice(&dev, adapter) == cudaSuccess)  break;  adapter->Release();  }  factory->Release();  // Create swap chain and device  ...  D3D10CreateDeviceAndSwapChain(adapter,  D3D10\_DRIVER\_TYPE\_HARDWARE, 0,  D3D10\_CREATE\_DEVICE\_DEBUG,  D3D10\_SDK\_VERSION,  &swapChainDesc, &swapChain,  &device);  adapter->Release();  // Register device with CUDA  cudaD3D10SetDirect3DDevice(device);  // Create vertex buffer and register it with CUDA  unsigned int size = width \* height \* sizeof(CUSTOMVERTEX);  D3D10\_BUFFER\_DESC bufferDesc;  bufferDesc.Usage = D3D10\_USAGE\_DEFAULT;  bufferDesc.ByteWidth = size;  bufferDesc.BindFlags = D3D10\_BIND\_VERTEX\_BUFFER;  bufferDesc.CPUAccessFlags = 0;  bufferDesc.MiscFlags = 0;  device->CreateBuffer(&bufferDesc, 0, &positionsVB);  cudaGraphicsD3D10RegisterResource(&positionsVB\_CUDA,  positionsVB,  cudaGraphicsRegisterFlagsNone);  cudaGraphicsResourceSetMapFlags(positionsVB\_CUDA,  cudaGraphicsMapFlagsWriteDiscard);  // Launch rendering loop  while (...) {  ...  Render();  ...  }  ...  }  void Render()  {  // Map vertex buffer for writing from CUDA  float4\* positions;  cudaGraphicsMapResources(1, &positionsVB\_CUDA, 0);  size\_t num\_bytes;  cudaGraphicsResourceGetMappedPointer((void\*\*)&positions,  &num\_bytes,  positionsVB\_CUDA));  // Execute kernel  dim3 dimBlock(16, 16, 1);  dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);  createVertices<<<dimGrid, dimBlock>>>(positions, time,  width, height);  // Unmap vertex buffer  cudaGraphicsUnmapResources(1, &positionsVB\_CUDA, 0);  // Draw and present  ...  }  void releaseVB()  {  cudaGraphicsUnregisterResource(positionsVB\_CUDA);  positionsVB->Release();  }  \_\_global\_\_ void createVertices(float4\* positions, float time,  unsigned int width, unsigned int height)  {  unsigned int x = blockIdx.x \* blockDim.x + threadIdx.x;  unsigned int y = blockIdx.y \* blockDim.y + threadIdx.y;  // Calculate uv coordinates  float u = x / (float)width;  float v = y / (float)height;  u = u \* 2.0f - 1.0f;  v = v \* 2.0f - 1.0f;  // Calculate simple sine wave pattern  float freq = 4.0f;  float w = sinf(u \* freq + time)  \* cosf(v \* freq + time) \* 0.5f;  // Write positions  positions[y \* width + x] =  make\_float4(u, w, v, \_\_int\_as\_float(0xff00ff00));  }  **Direct3D 11 Version:**  ID3D11Device\* device;  struct CUSTOMVERTEX {  FLOAT x, y, z;  DWORD color;  };  ID3D11Buffer\* positionsVB;  struct cudaGraphicsResource\* positionsVB\_CUDA;  int main()  {  // Get a CUDA-enabled adapter  IDXGIFactory\* factory;  CreateDXGIFactory(\_\_uuidof(IDXGIFactory), (void\*\*)&factory);  IDXGIAdapter\* adapter = 0;  for (unsigned int i = 0; !adapter; ++i) {  if (FAILED(factory->EnumAdapters(i, &adapter))  break;  int dev;  if (cudaD3D11GetDevice(&dev, adapter) == cudaSuccess)  break;  adapter->Release();  }  factory->Release();  // Create swap chain and device  ...  sFnPtr\_D3D11CreateDeviceAndSwapChain(adapter,  D3D11\_DRIVER\_TYPE\_HARDWARE,  0,  D3D11\_CREATE\_DEVICE\_DEBUG,  featureLevels, 3,  D3D11\_SDK\_VERSION,  &swapChainDesc, &swapChain,  &device,  &featureLevel,  &deviceContext);  adapter->Release();  // Register device with CUDA  cudaD3D11SetDirect3DDevice(device);  // Create vertex buffer and register it with CUDA  unsigned int size = width \* height \* sizeof(CUSTOMVERTEX);  D3D11\_BUFFER\_DESC bufferDesc;  bufferDesc.Usage = D3D11\_USAGE\_DEFAULT;  bufferDesc.ByteWidth = size;  bufferDesc.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER;  bufferDesc.CPUAccessFlags = 0;  bufferDesc.MiscFlags = 0;  device->CreateBuffer(&bufferDesc, 0, &positionsVB);  cudaGraphicsD3D11RegisterResource(&positionsVB\_CUDA,  positionsVB,  cudaGraphicsRegisterFlagsNone);  cudaGraphicsResourceSetMapFlags(positionsVB\_CUDA,  cudaGraphicsMapFlagsWriteDiscard);  // Launch rendering loop  while (...) {  ...  Render();  ...  }  ...  }  void Render()  {  // Map vertex buffer for writing from CUDA  float4\* positions;  cudaGraphicsMapResources(1, &positionsVB\_CUDA, 0);  size\_t num\_bytes;  cudaGraphicsResourceGetMappedPointer((void\*\*)&positions,  &num\_bytes,  positionsVB\_CUDA));  // Execute kernel  dim3 dimBlock(16, 16, 1);  dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);  createVertices<<<dimGrid, dimBlock>>>(positions, time,  width, height);  // Unmap vertex buffer  cudaGraphicsUnmapResources(1, &positionsVB\_CUDA, 0);  // Draw and present  ...  }  void releaseVB()  {  cudaGraphicsUnregisterResource(positionsVB\_CUDA);  positionsVB->Release();  }  \_\_global\_\_ void createVertices(float4\* positions, float time,  unsigned int width, unsigned int height)  {  unsigned int x = blockIdx.x \* blockDim.x + threadIdx.x;  unsigned int y = blockIdx.y \* blockDim.y + threadIdx.y;  // Calculate uv coordinates  float u = x / (float)width;  float v = y / (float)height;  u = u \* 2.0f - 1.0f;  v = v \* 2.0f - 1.0f;  // Calculate simple sine wave pattern  float freq = 4.0f;  float w = sinf(u \* freq + time)  \* cosf(v \* freq + time) \* 0.5f;  // Write positions  positions[y \* width + x] =  make\_float4(u, w, v, \_\_int\_as\_float(0xff00ff00));  } | 3.2.11.2 Функциональная совместимость с Direct3D  Функциональная совместимость с Direct3D поддерживается для Direct3D 9, Direct3D 10 и Direct3D 11.  Контекст CUDA может взаимодействовать только с одним устройством Direct3D в один момент времени, причем и контекст CUDA, и устройство Direct3D должны быть созданы на одном GPU. Кроме того, следующие соображения должны быть приняты во внимание при создании устройства: устройства Direct3D 9 должны быть созданы с **DeviceType**, установленным в **D3DDEVTYPE\_HAL** и **BehaviorFlags** с флагом **D3DCREATE\_HARDWARE\_VERTEXPROCESSING**. Устройства Direct3D 10 и Direct3D 11 должны быть созданы с **DriverType**, установленным в **D3D\_DRIVER\_TYPE\_HARDWARE**.  Функциональная совместимость с Direct3D требует, чтобы устройство Direct3D было определено с **cudaD3D9SetDirect3DDevice()**, **cudaD3D10SetDirect3DDevice()** и **cudaD3D11SetDirect3DDevice()** до любого другого вызова среды выпонения. **cudaD3D9GetDevice()**, **cudaD3D10GetDevice()** и **cudaD3D11GetDevice()** могут быть использованы для получения устройства CUDA, ассоциированного с некоторым адаптером.  Набор вызовов также доступен для разрешения создания контекстов CUDA, функционально совместимых с Direct3D, использующими NVIDIA SLI в режиме AFR (Alternate Frame Rendering, «поочередный рендеринг кадров»): **cudaD3D[9|10|11]GetDevices()**. Вызов **cudaD3D[9|10|11]GetDevices()** может быть использован для получения списка дескрипторов устройств CUDA, которые можно передавать как (необязательный) последний параметр в **cudaD3D[9|10|11]SetDirect3DDevice()**.  У приложения есть выбор – либо создать множественные нити на ЦПУ, каждый со своим контекстом CUDA, либо одну нить ЦПУ с многими контекстами CUDA. При использовании отдельных нитей ЦПУ для каждого GPU, каждый контекст CUDA будет создан средой выполнения CUDA через вызов в отдельной нити ЦПУ функции **cudaD3D[9|10|11]SetDirect3DDevice()** с одним из дескрипторов устройства CUDA, возвращенных by **cudaD3D[9|10|11]GetDevices()**.  При использовании одной нити ЦПУ контексты должны быть созданы с помощью функций создания контекста API драйвера CUDA для функциональной совместимости с устройствами Direct3D, использующими NVIDIA SLI (**cuD3D[9|10|11]CtxCreateOnDevice()**). Приложение полагается на функциональную совместимость между API драйвера CUDA и API среды выполнения (раздел Ж.4), которая позволяет ему вызывать **cuCtxPushCurrent()** и **cuCtxPopCurrent()** для изменения активного в данный момент контекста CUDA.  Ресурсы Direct3D, которые могут быть отображены в адресное пространство CUDA, это буферы, текстуры и поверхности Direct3D. Эти ресурсы регистрируются с помощью **cudaGraphicsD3D9RegisterResource()**,**cudaGraphicsD3D10RegisterResource()** и **cudaGraphicsD3D11RegisterResource()**.  Следующий пример кода использует ядро для динамической модификации двухмерной сетки вершин размера **width** x **height**, сохраненной в буфере вершин.  **Версия Direct3D 9:**  IDirect3D9\* D3D;  IDirect3DDevice9\* device;  struct CUSTOMVERTEX {  FLOAT x, y, z;  DWORD color;  };  IDirect3DVertexBuffer9\* positionsVB;  struct cudaGraphicsResource\* positionsVB\_CUDA;  int main()  {  // Инициализируем Direct3D  D3D = Direct3DCreate9(D3D\_SDK\_VERSION);  // Получаем поддерживающий CUDA адаптер  unsigned int adapter = 0;  for (; adapter < g\_pD3D->GetAdapterCount(); adapter++) {  D3DADAPTER\_IDENTIFIER9 adapterId;  g\_pD3D->GetAdapterIdentifier(adapter, 0, &adapterId);  int dev;  if (cudaD3D9GetDevice(&dev, adapterId.DeviceName)  == cudaSuccess)  break;  }  // Создаем устройство  ...  D3D->CreateDevice(adapter, D3DDEVTYPE\_HAL, hWnd,  D3DCREATE\_HARDWARE\_VERTEXPROCESSING,  &params, &device);  // Регистрируем устройство в CUDA  cudaD3D9SetDirect3DDevice(device);  // Создаем буфер вершин и регистрируем его в CUDA  unsigned int size = width \* height \* sizeof(CUSTOMVERTEX);  device->CreateVertexBuffer(size, 0, D3DFVF\_CUSTOMVERTEX,  D3DPOOL\_DEFAULT, &positionsVB, 0);  cudaGraphicsD3D9RegisterResource(&positionsVB\_CUDA,  positionsVB,  cudaGraphicsRegisterFlagsNone);  cudaGraphicsResourceSetMapFlags(positionsVB\_CUDA,  cudaGraphicsMapFlagsWriteDiscard);  // Запускаем цикл рендеринга  while (...) {  ...  Render();  ...  }  void Render()  {  // Отображаем буфер вершин для записи из CUDA  float4\* positions;  cudaGraphicsMapResources(1, &positionsVB\_CUDA, 0);  size\_t num\_bytes;  cudaGraphicsResourceGetMappedPointer((void\*\*)&positions,  &num\_bytes,  positionsVB\_CUDA));  // Исполняем ядро  dim3 dimBlock(16, 16, 1);  dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);  createVertices<<<dimGrid, dimBlock>>>(positions, time,  width, height);  // Отменяем отображение буфера вершин  cudaGraphicsUnmapResources(1, &positionsVB\_CUDA, 0);  // Рисуем и выводим  ...  }  void releaseVB()  {  cudaGraphicsUnregisterResource(positionsVB\_CUDA);  positionsVB->Release();  }  \_\_global\_\_ void createVertices(float4\* positions, float time,  unsigned int width, unsigned int height)  {  unsigned int x = blockIdx.x \* blockDim.x + threadIdx.x;  unsigned int y = blockIdx.y \* blockDim.y + threadIdx.y;  // Вычисляем координаты uv  float u = x / (float)width;  float v = y / (float)height;  u = u \* 2.0f - 1.0f;  v = v \* 2.0f - 1.0f;  // Вычисляем модель простого синусоидального колебания  float freq = 4.0f;  float w = sinf(u \* freq + time)  \* cosf(v \* freq + time) \* 0.5f;  // Записываем позиции  positions[y \* width + x] =  make\_float4(u, w, v, \_\_int\_as\_float(0xff00ff00));  }  **Версия Direct3D 10:**  ID3D10Device\* device;  struct CUSTOMVERTEX {  FLOAT x, y, z;  DWORD color;  };  ID3D10Buffer\* positionsVB;  struct cudaGraphicsResource\* positionsVB\_CUDA;  int main()  {  // Получаем поддерживающий CUDA адаптер  IDXGIFactory\* factory;  CreateDXGIFactory(\_\_uuidof(IDXGIFactory), (void\*\*)&factory);  IDXGIAdapter\* adapter = 0;  for (unsigned int i = 0; !adapter; ++i) {  if (FAILED(factory->EnumAdapters(i, &adapter))  break;  int dev;  if (cudaD3D10GetDevice(&dev, adapter) == cudaSuccess)  break;  adapter->Release();  }  factory->Release();  // Создаем цепочку обменов и устройство  ...  D3D10CreateDeviceAndSwapChain(adapter,  D3D10\_DRIVER\_TYPE\_HARDWARE, 0,  D3D10\_CREATE\_DEVICE\_DEBUG,  D3D10\_SDK\_VERSION,  &swapChainDesc, &swapChain,  &device);  adapter->Release();  // Регистрируем устройство в CUDA  cudaD3D10SetDirect3DDevice(device);  // Создаем буфер вершин и регистрируем его в CUDA  unsigned int size = width \* height \* sizeof(CUSTOMVERTEX);  D3D10\_BUFFER\_DESC bufferDesc;  bufferDesc.Usage = D3D10\_USAGE\_DEFAULT;  bufferDesc.ByteWidth = size;  bufferDesc.BindFlags = D3D10\_BIND\_VERTEX\_BUFFER;  bufferDesc.CPUAccessFlags = 0;  bufferDesc.MiscFlags = 0;  device->CreateBuffer(&bufferDesc, 0, &positionsVB);  cudaGraphicsD3D10RegisterResource(&positionsVB\_CUDA,  positionsVB,  cudaGraphicsRegisterFlagsNone);  cudaGraphicsResourceSetMapFlags(positionsVB\_CUDA,  cudaGraphicsMapFlagsWriteDiscard);  // Запускаем цикл рендеринга  while (...) {  ...  Render();  ...  }  ...  }  void Render()  {  // Отображаем буфер вершин для записи из CUDA  float4\* positions;  cudaGraphicsMapResources(1, &positionsVB\_CUDA, 0);  size\_t num\_bytes;  cudaGraphicsResourceGetMappedPointer((void\*\*)&positions,  &num\_bytes,  positionsVB\_CUDA));  // Исполняем ядро  dim3 dimBlock(16, 16, 1);  dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);  createVertices<<<dimGrid, dimBlock>>>(positions, time,  width, height);  // Отменяем отображение буфера вершин  cudaGraphicsUnmapResources(1, &positionsVB\_CUDA, 0);  // Рисуем и выводим  ...  }  void releaseVB()  {  cudaGraphicsUnregisterResource(positionsVB\_CUDA);  positionsVB->Release();  }  \_\_global\_\_ void createVertices(float4\* positions, float time,  unsigned int width, unsigned int height)  {  unsigned int x = blockIdx.x \* blockDim.x + threadIdx.x;  unsigned int y = blockIdx.y \* blockDim.y + threadIdx.y;  // Вычисляем координаты uv  float u = x / (float)width;  float v = y / (float)height;  u = u \* 2.0f - 1.0f;  v = v \* 2.0f - 1.0f;  // Вычисляем модель простого синусоидального колебания  float freq = 4.0f;  float w = sinf(u \* freq + time)  \* cosf(v \* freq + time) \* 0.5f;  // Записываем позиции  positions[y \* width + x] =  make\_float4(u, w, v, \_\_int\_as\_float(0xff00ff00));  }  **Версия Direct3D 11:**  ID3D11Device\* device;  struct CUSTOMVERTEX {  FLOAT x, y, z;  DWORD color;  };  ID3D11Buffer\* positionsVB;  struct cudaGraphicsResource\* positionsVB\_CUDA;  int main()  {  // Получаем поддерживающий CUDA адаптер  IDXGIFactory\* factory;  CreateDXGIFactory(\_\_uuidof(IDXGIFactory), (void\*\*)&factory);  IDXGIAdapter\* adapter = 0;  for (unsigned int i = 0; !adapter; ++i) {  if (FAILED(factory->EnumAdapters(i, &adapter))  break;  int dev;  if (cudaD3D11GetDevice(&dev, adapter) == cudaSuccess)  break;  adapter->Release();  }  factory->Release();  // Создаем цепочку обменов и устройство  ...  sFnPtr\_D3D11CreateDeviceAndSwapChain(adapter,  D3D11\_DRIVER\_TYPE\_HARDWARE,  0,  D3D11\_CREATE\_DEVICE\_DEBUG,  featureLevels, 3,  D3D11\_SDK\_VERSION,  &swapChainDesc, &swapChain,  &device,  &featureLevel,  &deviceContext);  adapter->Release();  // Регистрируем устройство в CUDA  cudaD3D11SetDirect3DDevice(device);  // Создаем буфер вершин и регистрируем его в CUDA  unsigned int size = width \* height \* sizeof(CUSTOMVERTEX);  D3D11\_BUFFER\_DESC bufferDesc;  bufferDesc.Usage = D3D11\_USAGE\_DEFAULT;  bufferDesc.ByteWidth = size;  bufferDesc.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER;  bufferDesc.CPUAccessFlags = 0;  bufferDesc.MiscFlags = 0;  device->CreateBuffer(&bufferDesc, 0, &positionsVB);  cudaGraphicsD3D11RegisterResource(&positionsVB\_CUDA,  positionsVB,  cudaGraphicsRegisterFlagsNone);  cudaGraphicsResourceSetMapFlags(positionsVB\_CUDA,  cudaGraphicsMapFlagsWriteDiscard);  // Запускаем цикл рендеринга  while (...) {  ...  Render();  ...  }  ...  }  void Render()  {  // Отображаем буфер вершин для записи из CUDA  float4\* positions;  cudaGraphicsMapResources(1, &positionsVB\_CUDA, 0);  size\_t num\_bytes;  cudaGraphicsResourceGetMappedPointer((void\*\*)&positions,  &num\_bytes,  positionsVB\_CUDA));  // Исполняем ядро  dim3 dimBlock(16, 16, 1);  dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);  createVertices<<<dimGrid, dimBlock>>>(positions, time,  width, height);  // Отменяем отображение буфера вершин cudaGraphicsUnmapResources(1, &positionsVB\_CUDA, 0);  // Рисуем и выводим  ...  }  void releaseVB()  {  cudaGraphicsUnregisterResource(positionsVB\_CUDA);  positionsVB->Release();  }  \_\_global\_\_ void createVertices(float4\* positions, float time,  unsigned int width, unsigned int height)  {  unsigned int x = blockIdx.x \* blockDim.x + threadIdx.x;  unsigned int y = blockIdx.y \* blockDim.y + threadIdx.y;  // Вычисляем координаты uv  float u = x / (float)width;  float v = y / (float)height;  u = u \* 2.0f - 1.0f;  v = v \* 2.0f - 1.0f;  // Вычисляем модель простого синусоидального колебания  float freq = 4.0f;  float w = sinf(u \* freq + time)  \* cosf(v \* freq + time) \* 0.5f;  // Записываем позиции  positions[y \* width + x] =  make\_float4(u, w, v, \_\_int\_as\_float(0xff00ff00));  } |
| 3.2.11.3 SLI Interoperability  In a system with multiple GPUs, all CUDA-enabled GPUs are accessible via the CUDA driver and runtime as separate devices. There are however special considerations as described below when the system is in SLI mode.  First, an allocation in one CUDA device on one GPU will consume memory on other GPUs that are part of the SLI configuration of the Direct3D or OpenGL device. Because of this, allocations may fail earlier than otherwise expected.  Second, applications have to create multiple CUDA contexts, one for each GPU in the SLI configuration and deal with the fact that a different GPU is used for rendering by the Direct3D or OpenGL device at every frame. The application can use the **cudaD3D[9|10|11]GetDevices()** for Direct3D and **cudaGLGetDevices()** for OpenGL set of calls to identify the CUDA device handle(s) for the device(s) that are performing the rendering in the current and next frame. Given this information the application will typically map Direct3D or OpenGL resources to the CUDA context corresponding to the CUDA device returned by **cudaD3D[9|10|11]GetDevices()** or **cudaGLGetDevices()** when the **deviceList** parameter is set to **CU\_D3D10\_DEVICE\_LIST\_CURRENT\_FRAME** or **cudaGLDeviceListCurrentFrame***.*  See Sections 3.2.11.2 and 3.2.11.1 for details on how the CUDA runtime interoperate with Direct3D and OpenGL, respectively. | 3.2.11.3 Функциональная совместимость с SLI  В системе с несколькими ГПУ все поддерживающие CUDA ГПУ доступны через драйвер CUDA и среду выполнения как отдельные устройства. Тем не менее, требует особого рассмотрения случай, когда система находится в режиме SLI.  Во-первых, выделение памяти в одном устройстве CUDA на одном ГПУ будет потреблять память на других ГПУ, являющихся частью SLI-конфигурации устройства Direct3D или OpenGL. Из-за этого выделения могут начать не выполняться раньше, чем ожидается в противном случае (то есть при работе не в режиме SLI).  Во-вторых, приложения должны создавать несколько контекстов CUDA, по одному для каждого ГПУ в SLI-конфигурации, и иметь дело с тем фактом, что разные ГПУ используются для рендеринга каждого кадра устройствами Direct3D и OpenGL. Приложение может использовать **cudaD3D[9|10|11]GetDevices()** для набора вызовов Direct3D и **cudaGLGetDevices() –** для OpenGL, чтобы определить дескриптор(ы) устройства CUDA для устройств(а), которые выполняют рендеринг в текущем и следующем кадре. Учитывая данную информацию, приложению следовало бы отображать ресурсы Direct3D или OpenGL в контекст CUDA, соответствующий устройству CUDA, возвращенному **cudaD3D[9|10|11]GetDevices()** или **cudaGLGetDevices()**, когда параметр **deviceList** установлен в **CU\_D3D10\_DEVICE\_LIST\_CURRENT\_FRAME** или **cudaGLDeviceListCurrentFrame***.*  Для получения более подробной информации о том, как среда выполнения CUDA взаимодействует с Direct3D и OpenGL, смотрите разделы 3.2.11.2 и 3.2.11.1 соответственно. |
| 3.3 Versioning and Compatibility  There are two version numbers that developers should care about when developing a CUDA application: The compute capability that describes the general specifications and features of the compute device (see Section 2.5) and the version of the CUDA driver API that describes the features supported by the driver API and runtime.  The version of the driver API is defined in the driver header file as **CUDA\_VERSION**. It allows developers to check whether their application requires a newer device driver than the one currently installed. This is important, because the driver API is *backward compatible*, meaning that applications, plug-ins, and libraries (including the C runtime) compiled against a particular version of the driver API will continue to work on subsequent device driver releases as illustrated in Figure 3-3. The driver API is not *forward compatible*, which means that applications, plug-ins, and libraries (including the C runtime) compiled against a particular version of the driver API will not work on previous versions of the device driver.  It is important to note that mixing and matching versions is not supported; specifically:  All applications, plug-ins, and libraries on a system must use the same version of the CUDA driver API, since only one version of the CUDA device driver can be installed on a system.  All plug-ins and libraries used by an application must use the same version of the runtime.  All plug-ins and libraries used by an application must use the same version of any libraries that use the runtime (such as CUFFT, CUBLAS, …).  ***<Figure 3-3 is omitted>*** | 3.3 Управление версиями и совместимость  При разработке приложений разработчики должны помнить о двух номерах версии: вычислительной способности, описывающей общие спецификации и возможности вычислительного устройства (см. раздел 2.5), и версии API драйвера CUDA, описывающей возможности, поддерживаемые API драйвера и средой выполнения.  Версия API драйвера определена в заголовочном файле как **CUDA\_VERSION**. Она позволяет разработчикам проверять, необходим ли их приложению более новый, чем установленный, драйвер устройства. Это важно, так как API драйвера *обратно совместимо*, т.е. приложения, плагины и библиотеки (в том числе среду выполнения C), скомпилированные с определенной версией API драйвера, будут продолжать работать на последующих выпусках драйвера устройства, как показано на рисунке 3-3. API драйвера не является *прямо совместимым*, т.е. приложения, плагины и библиотеки (в том числе среду выполнения C), скомпилированные с определенной версией API драйвера, не будут работать на предыдущих версиях драйвера устройства.  Важно отметить, что комбинированные версии не поддерживаются, конкретно:  Все приложения, плагины и библиотеки системы должны использовать одну версию API драйвера CUDA, так как в системе может быть установлена только одна версия драйвера устройства CUDA  Все плагины и библиотеки, используемые приложением, должны использовать одну версию среды выполнения.  Все плагины и библиотеки, используемые приложением, должны использовать одну версию любых библиотек, использующих среду выполнения (таких как CUFFT, CUBLAS, …).  ***<Figure 3-3 is omitted>*** |
| 3.4 Compute Modes  On Tesla solutions running Windows Server 2008 and later or Linux, one can set any device in a system in one of the three following modes using NVIDIA’s System Management Interface (*nvidia-smi*), which is a tool distributed as part of the driver:   * *Default* compute mode: Multiple host threads can use the device (by calling **cudaSetDevice()** on this device, when using the runtime API, or by making current a context associated to the device, when using the driver API) at the same time. * *Exclusive-process* compute mode: Only one CUDA context may be created on the device across all processes in the system and that context may be current to as many threads as desired within the process that created that context. * *Exclusive-process-and-thread* compute mode: Only one CUDA context may be created on the device across all processes in the system and that context may only be current to one thread at a time. * *Prohibited* compute mode: No CUDA context can be created on the device.   This means, in particular, that a host thread using the runtime API without explicitly calling **cudaSetDevice()** might be associated with a device other than device 0 if device 0 turns out to be in the exclusive-process mode and used by another process, or in the exclusive-process-and-thread mode and used by another thread, or in prohibited mode. **cudaSetValidDevices()** can be used to set a device from a prioritized list of devices.  Applications may query the compute mode of a device by checking the computeMode device property (see Section 3.2.6.1). | 3.4 Вычислительные режимы  На решениях Tesla, использующих Windows Server 2008 и более поздние или Linux, можно установить любое устройство в системе в один из следующих трех режимов, используя NVIDIA’s System Management Interface (*nvidia-smi*, «интерфейс управления системой»), инструмент, распространяемый как часть драйвера:   * Вычислительный режим *по умолчанию:* несколько нитей хоста могут использовать устройство (вызовом **cudaSetDevice()** на этом устройстве при использовании API среды выполнения; или установке текущим контекста, ассоциированного с устройством, при использовании API драйвера) в одно и то же время. * Вычислительный режим *эксклюзивного процесса*: только один контекст CUDA может быть создан на устройстве для всех процессов в системе, и этот контекст может быть текущим для стольких нитей, сколько необходимо для процесса, создавшего этот контекст. * Вычислительный режим *эксклюзивного процесса и нити*: только один контекст CUDA может быть создан на устройстве для всех процессов в системе, и этот контекст может быть текущим только для одной нити в один момент времени. * *Запрещающий* вычислительный режим: создание контекста CUDA на устройстве запрещено.   Это, в частности, означает, что нить хоста, использующая API времени выполнения без явного вызова **cudaSetDevice()**, может быть ассоциирована с устройством, отличным от устройства 0, если устройство 0 оказывается находящимся или в режиме эксклюзивного процесса и используется другим процессом, или в режиме эксклюзивного процесса и нити и используется другой нитью, или в запрещающем режиме. **cudaSetValidDevices()** может быть использована для установки устройства из списка устройств с приоритетами.  Приложения могут запрашивать вычислительный режим устройства, проверяя свойство устройства **computeMode** (см. раздел 3.2.6.1). |
| 3.5 Mode Switches  GPUs that have a display output dedicate some DRAM memory to the so-called *primary surface*, which is used to refresh the display device whose output is viewed by the user. When users initiate a *mode switch* of the display by changing the resolution or bit depth of the display (using NVIDIA control panel or the Display control panel on Windows), the amount of memory needed for the primary surface changes. For example, if the user changes the display resolution from 1280x1024x32-bit to 1600x1200x32-bit, the system must dedicate 7.68 MB to the primary surface rather than 5.24 MB. (Full-screen graphics applications running with anti-aliasing enabled may require much more display memory for the primary surface.) On Windows, other events that may initiate display mode switches include launching a full-screen DirectX application, hitting Alt+Tab to task switch away from a full-screen DirectX application, or hitting Ctrl+Alt+Del to lock the computer.  If a mode switch increases the amount of memory needed for the primary surface, the system may have to cannibalize memory allocations dedicated to CUDA applications. Therefore, a mode switch results in any call to the CUDA runtime to fail and return an invalid context error. | 3.5 Переключение режимов  ГПУ, имеющие вывод на дисплей, выделяют часть памяти DRAM на поддержку так называемой *первичной поверхности*, используемой для обновления устройства отображения, чей вывод виден пользователю. Когда пользователи инициируют *переключение режима* дисплея изменением разрешения или битовой глубины дисплея (с помощью панели управления NVIDIA или панели управления дисплеем в Windows), количество памяти, необходимое для первичной поверхности, изменяется. Например, если пользователь меняет разрешение дисплея с 1280x1024x32 бита на 1600x1200x32 бита, система должна выделить 7.68 МБ первичной поверхности вместо 5.24 МБ. (Полноэкранные графические приложения, выполняемые в режиме anti-aliasing, могут требовать намного больше памяти для первичной поверхности) На Windows другими событиями, могущими инициировать смену режима дисплея, являются запуск полноэкранного приложения DirectX, нажатие Alt+Tab для смены задачи с полноэкранного приложения DirectX или нажатие Ctrl+Alt+Del для блокирования компьютера.  Если смена режима увеличивает количество памяти, необходимое первичной поверхности, система может отнять выделенную память приложений CUDA. Следовательно, смена режима приводит к провалу любых попыток совершения вызовов среды выполнения CUDA и возврату ошибки недействительности контекста. |
| 3.6 Tesla Compute Cluster Mode for Windows  Using NVIDIA’s System Management Interface (*nvidia-smi*), the Windows device driver can be put in TCC (Tesla Compute Cluster) mode for devices of the Tesla and Quadro Series of compute capability 2.0 and higher.  This mode has the following primary benefits:   * It makes it possible to use these GPUs in cluster nodes with non‐NVIDIA integrated graphics; * It makes these GPUs available via Remote Desktop, both directly and via cluster management systems that rely on Remote Desktop; * It makes these GPUs available to applications running as a Windows service (i.e. in Session 0).   However, the TCC mode removes support for any graphics functionality. | 3.6 Режим вычислительного кластера Tesla для Windows  Используя NVIDIA’s System Management Interface (*nvidia-smi*), можно установить драйвер устройства в Windows в режим TCC (Tesla Compute Cluster, «вычислительный кластер Tesla»), что работает для устройств серий Tesla и Quadro вычислительной способности 2.0 и выше.  Этот режим имеет следующие основные преимущества:   * Становится возможным использование этих ГПУ в узлах кластера с графическими адаптерами, не являющимися продукцией NVIDIA; * Эти ГПУ становятся доступны через Remote Desktop (удаленный рабочий стол): и напрямую, и через системы управления кластером, полагающиеся на Remote Desktop; * Эти ГПУ становятся доступны приложениям, выполняющимся как служба Windows (т.е. в сессии 0).   Однако стоит отметить, что режим TCC отключает поддержку какой-либо графической функциональности. |