

## CUDA C++ Programming Guide

Введение в программирование технологии и интерфейса CUDA.

### Changes from Version 11.1

- Updated [Asynchronous Data Copies](#) using `cuda::memcpy_async` and `cooperative_group::memcpy_async`.
- Updated [Asynchronous Barrier](#) using `cuda::barrier`.
- Added [Compiler Optimization Hint Functions](#).

## 1. Введение

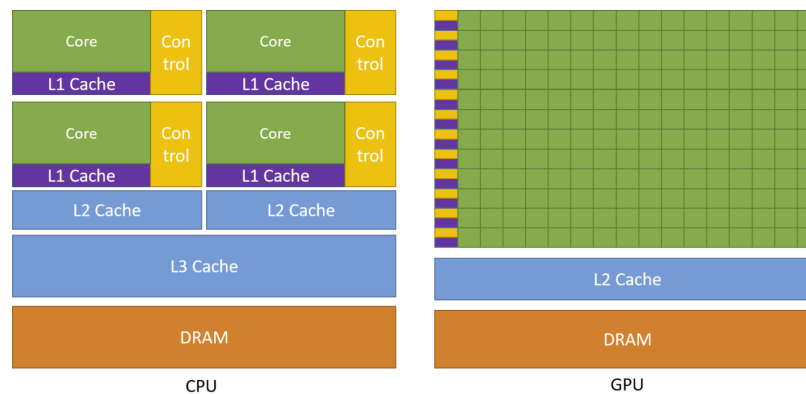
### 1.1. Преимущества использования графических процессоров

Графический процессор (GPU) обеспечивает гораздо более высокую пропускную способность инструкций и пропускную способность памяти, чем ЦП, при аналогичной цене и мощности. Термин GPU происходит из того факта, что при первоначальном создании графического процессора, два десятилетия назад, он проектировался как специализированный процессор для ускорения рендеринга графики. Под влиянием ненасытного рыночного спроса на трехмерную графику высокого разрешения в реальном времени, он превратился в универсальный процессор, используемый для многих других рабочих нагрузок, чем просто рендеринг графики.

Эта разница в возможностях графического процессора и процессора существует потому, что они разработаны с разными целями. В то время как ЦП предназначен для максимально быстрого выполнения последовательности операций, называемых потоками, и может выполнять несколько десятков этих потоков параллельно, графический процессор предназначен для превосходного выполнения тысяч из них параллельно (амортизируя более медленную однопоточную производительность для достижения большей пропускной способности).

Графический процессор специализируется на высокопараллельных вычислениях и поэтому спроектирован таким образом, что больше транзисторов используется для обработки данных, а не для кэширования данных и управления потоком. На схематическом рисунке 1 показан пример распределения ресурсов микросхемы для центрального процессора по сравнению с графическим процессором.

Рис. 1. Графический процессор выделяет больше транзисторов для обработки данных.



Выделение большого количества транзисторов для обработки данных, например вычислений с плавающей запятой, полезно для высокопараллельных вычислений; GPU может скрыть задержки доступа к памяти с помощью вычислений, вместо того, чтобы полагаться на большие кешы данных и сложное управление потоком, чтобы избежать длительных задержек доступа к памяти, оба из которых являются дорогостоящими с точки зрения транзисторов.

Как правило, приложение состоит из параллельных и последовательных частей, поэтому системы разрабатываются с сочетанием графических и центральных процессоров, чтобы максимизировать общую производительность. Приложения с высокой степенью параллелизма могут использовать эту массово-параллельную природу графического процессора для достижения более высокой производительности, чем на центральном процессоре.

## 1.2. CUDA®: универсальная платформа параллельных вычислений и модель программирования

В ноябре 2006 года NVIDIA® представила CUDA®, платформу параллельных вычислений общего назначения и модель программирования, которая использует механизм параллельных вычислений в графических процессорах NVIDIA для решения многих сложных вычислительных задач более эффективным способом, чем на CPU.

CUDA поставляется с программной средой, которая позволяет разработчикам использовать C++ как язык программирования высокого уровня. Как показано на рисунке 2, поддерживаются другие языки, интерфейсы прикладного программирования или подходы на основе директив, такие как FORTRAN, DirectCompute, OpenACC.

Рисунок 2. Вычислительные приложения на GPU. CUDA разработан для поддержки различных языков и интерфейсов прикладного программирования.


# GPU Computing Applications

Libraries and Middleware





cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica
-------------------	---------------------------------------	---------------	---------------	-----------------------------	------------------------	-----------------------

Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)
---	-----	---------	----------------------------	---------------	------------------------------



CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
	<div><div></div><div>Embedded</div></div>	<div><div></div><div>Consumer Desktop/Laptop</div></div>	<div><div></div><div>Professional Workstation</div></div>	<div><div></div><div>Data Center</div></div>

### 1.3. Масштабируемая модель программирования

Появление многоядерных центральных процессоров и многоядерных графических процессоров означает, что чипы процессора теперь являются параллельными системами. Задача состоит в том, чтобы разработать прикладное программное обеспечение, которое прозрачно масштабирует свой параллелизм для увеличения числа ядер процессора, подобно тому, как приложения трехмерной графики прозрачно масштабируют свой параллелизм на многоядерные графические процессоры с сильно различающимся числом ядер.

Модель параллельного программирования CUDA предназначена для решения этой проблемы при обучении программистов, знакомых со стандартными языками программирования, такими как C.

В его основе лежат три ключевые абстракции - иерархия групп потоков, общая память и барьерная синхронизация - которые просто открываются программисту как минимальный набор языковых расширений.

Эти абстракции обеспечивают детальный параллелизм данных и параллелизм потоков, вложенный в крупномасштабный параллелизм данных и параллелизм задач. Они помогают программисту разделить проблему на грубые подзадачи, которые могут быть решены независимо параллельно блоками потоков, и каждую подзадачу на более мелкие части, которые могут быть решены совместно параллельно всеми потоками в блоке.

Эта декомпозиция сохраняет выразительность языка, позволяя потокам взаимодействовать при решении каждой подзадачи, и в то же время обеспечивает автоматическую масштабируемость. Действительно, каждый блок потоков может быть запланирован на любом из доступных мультипроцессоров в графическом процессоре в любом порядке, одновременно или последовательно, так что скомпилированная программа CUDA может выполняться на любом количестве мультипроцессоров, как показано на рисунке 3, и только во время выполнения. системе необходимо знать количество физических мультипроцессоров.

Эта масштабируемая модель программирования позволяет архитектуре графического процессора охватить широкий диапазон рынка путем простого масштабирования количества мультипроцессоров и разделов памяти: от высокопроизводительных графических процессоров GeForce для энтузиастов и профессиональных вычислительных продуктов Quadro и Tesla до различных недорогих массовых графических процессоров GeForce ( список всех графических процессоров с поддержкой CUDA см. в разделе «Графические процессоры с поддержкой CUDA»

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-enabled-gpus>

).

Рисунок 3. Автоматическая масштабируемость



Примечание. Графический процессор построен на основе массива потоковых мультипроцессоров (SM streaming multiprocessor) (подробнее см. В разделе «Аппаратная реализация»). Многопоточная программа разбивается на блоки потоков, которые выполняются независимо друг от друга, поэтому графический процессор с большим количеством мультипроцессоров автоматически выполняет программу за меньшее время, чем графический процессор с меньшим количеством мультипроцессоров.

#### 1.4. Структура документа

Этот документ состоит из следующих глав:

Глава Введение - это общее введение в CUDA.

Глава Programming Model описывает модель программирования CUDA.

Глава Programming Interface описывает программный интерфейс.

Глава «Аппаратная реализация» описывает аппаратную реализацию.

Глава «Рекомендации по производительности» дает некоторые рекомендации о том, как достичь максимальной производительности.

В приложении «Графические процессоры с поддержкой CUDA» перечислены все устройства с поддержкой CUDA.

Приложение «Расширения языка C ++» - это подробное описание всех расширений языка C ++.

Приложение «Кооперативные группы» описывает примитивы синхронизации для различных групп потоков CUDA.

Приложение CUDA Dynamic Parallelism описывает, как запускать и синхронизировать одно ядро с другим.

В приложении «Математические функции» перечислены математические функции, поддерживаемые в CUDA.

В приложении «Поддержка языка C ++» перечислены функции C ++, поддерживаемые в коде устройства.

Приложение "Выборка текстуры" дает более подробную информацию о выборке текстуры.

В приложении «Вычислительные возможности» приведены технические характеристики различных устройств, а также более подробная информация об архитектуре.

Приложение Driver API знакомит с API драйвера низкого уровня.

В Приложении «Переменные среды CUDA» перечислены все переменные среды CUDA. Приложение Unified Memory Programming знакомит с моделью программирования Unified Memory.

## 2. Модель программирования

В этой главе представлены основные концепции, лежащие в основе модели программирования CUDA, и показано, как они представлены в C++. Подробное описание CUDA C++ дано в Интерфейсе программирования.

Полный код примера сложения векторов, использованного в этой и следующей главах, можно найти в примере vectorAdd CUDA.

### 2.1. Ядра (kernels)

CUDA C++ расширяет C++, позволяя программисту определять функции C++, называемые ядрами (kernel), которые при вызове выполняются N раз параллельно N различными потоками CUDA, а не только один раз, как обычные функции C++.

Ядро определяется с помощью спецификатора объявления `__global__`, а количество потоков CUDA, которые выполняют это ядро для данного вызова ядра, указывается с использованием нового синтаксиса конфигурации выполнения `<<< ... >>>` (см. Расширения языка C++). Каждому потоку, выполняющему ядро, дается уникальный идентификатор потока, который доступен в ядре через встроенные переменные.

В качестве иллюстрации следующий пример кода, используя встроенную переменную `threadIdx`, добавляет два вектора A и B размера N и сохраняет результат в векторе 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);
    ...
}
```

Здесь каждый из N потоков, выполняющих `VecAdd()`, выполняет одно попарное сложение.

В очень широком смысле:

**API (программный интерфейс приложения, интерфейс прикладного программирования)** (англ. *application programming interface, API*) описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой. Обычно входит в описание какого-либо интернет-протокола (например, RFC<sup>[2]</sup>), программного каркаса (фреймворка)<sup>[3]</sup> или стандарта вызовов функций операционной системы<sup>[4]</sup>. Часто реализуется отдельной программной библиотекой или сервисом операционной системы. Используется программистами при написании всевозможных приложений.

- `cuda.h` определяет функции и типы общедоступного хоста для CUDA API.
- `cuda_runtime_api.h` определяет функции и типы общедоступного хоста для среды выполнения CUDA API
- `cuda_runtime.h` определяет все, что делает `cuda_runtime_api.h`, а также встроенные определения типов и наложения функций для расширений языка CUDA и встроенных функций устройства.

Если бы вы писали код хоста для компиляции с помощью компилятора хоста, который включает в себя вызовы API, вы бы включили либо `cuda.h`, либо `cuda_runtime_api.h`. Если вам нужны другие встроенные функции языка CUDA, такие как типы, и вы используете среду выполнения API и компилируетесь с хост-компилятором, вы бы включили `cuda_runtime.h`. Если вы пишете код, который будет скомпилирован с помощью nvcc, все это не имеет значения, потому что nvcc заботится о включении всех необходимых заголовков автоматически без вмешательства программиста.

## 2.2. Иерархия потоков

Для удобства `threadIdx` представляет собой трехкомпонентный вектор, так что потоки могут быть идентифицированы с использованием одномерного, двухмерного или трехмерного индекса потока, образуя одномерный, двухмерный или трехмерный блок потоков, называемый блоком потоков. Это обеспечивает естественный способ вызова вычислений для элементов в домене, таком как вектор, матрица или объем.

Индекс потока и его идентификатор связаны друг с другом напрямую: для одномерного блока они одинаковы; для двумерного блока размера (Dx, Dy) идентификатор потока для потока с индексом (x, y) равен (x + y Dx); для трехмерного блока размера (Dx, Dy, Dz) идентификатор потока для потока с индексом (x, y, z) равен (x + y Dx + z Dx Dy).

В качестве примера следующий код добавляет две матрицы A и B размером NxN и сохраняет результат в матрицу 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);
    ...
}

```

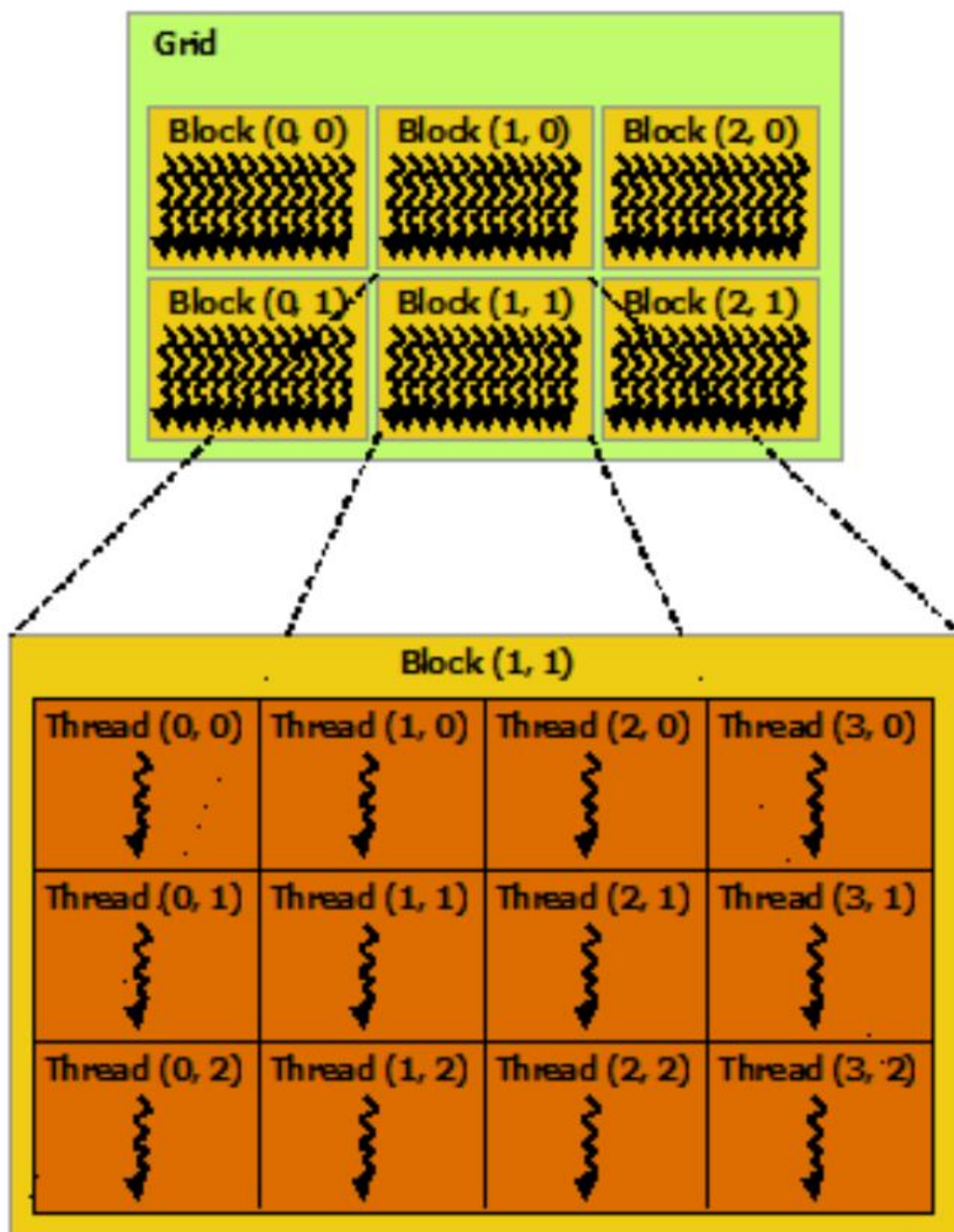
Существует ограничение на количество потоков на блок, поскольку ожидается, что все потоки блока будут находиться в одном ядре процессора и должны совместно использовать ограниченные ресурсы памяти этого ядра. На текущих графических процессорах блок потока может содержать до 1024 потоков.

Однако ядро может выполняться несколькими блоками потоков одинаковой формы, так что общее количество потоков равно количеству потоков в блоке, умноженному на количество блоков.

Блоки организованы в одномерную, двумерную или трехмерную сетку блоков потоков, как показано на рисунке 4. Количество блоков потоков в сетке обычно определяется размером обрабатываемых данных, который обычно превышает количество процессоров в системе.

Рисунок 4. Сетка блоков





Количество потоков на блок и количество блоков на сетку, указанные в синтаксисе <<< ... >>>, могут иметь тип `int` или `dim3`. Двумерные блоки или сетки могут быть указаны, как в приведенном выше примере.

Каждый блок в сетке может быть идентифицирован с помощью одномерного, двумерного или трехмерного уникального индекса, доступного в ядре через встроенную переменную `blockIdx`. Размер блока потока доступен в ядре через встроенную переменную `blockDim`.

Расширяя предыдущий пример MatAdd () для обработки нескольких блоков, код становится следующим.

```
// 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);
    ...
}
```

Размер блока потока 16x16 (256 потоков), хотя и произвольный в данном случае, является обычным выбором. Сетка создается с таким количеством блоков, чтобы, как и раньше, иметь один поток на элемент матрицы. Для простоты в этом примере предполагается, что количество потоков на сетку в каждом измерении без остатка делится на количество потоков на блок в этом измерении, хотя это не обязательно.

Блоки потоков должны выполняться независимо: они должны выполняться в любом порядке, параллельно или последовательно. Это требование независимости позволяет планировать блоки потоков в любом порядке для любого количества ядер, как показано на рисунке 3, позволяя программистам писать код, масштабируемый с увеличением количества ядер.

Потоки внутри блока могут взаимодействовать, обмениваясь данными через некоторую общую память и синхронизируя их выполнение для координации доступа к памяти. Точнее, можно указать точки синхронизации в ядре, вызвав внутреннюю функцию \_\_syncthreads ();

\_\_syncthreads () действует как барьер, на котором все потоки в блоке должны ждать, прежде чем любой из них будет разрешен для продолжения. Общая память дает пример использования общей памяти. В дополнение к \_\_syncthreads () API кооперативных групп предоставляет богатый набор примитивов синхронизации потоков.

Для эффективного взаимодействия ожидается, что совместно используемая память будет памятью с малой задержкой рядом с каждым ядром процессора (во многом как кеш L1), а \_\_syncthreads () будет облегченным.

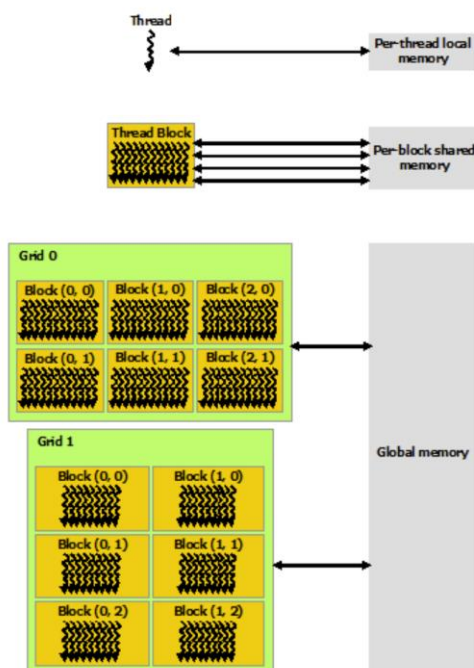
## 2.3. Иерархия памяти

Потоки CUDA могут получать доступ к данным из нескольких пространств памяти во время своего выполнения, как показано на рисунке 5. Каждый поток имеет частную локальную память. Каждый блок потока имеет общую память, видимую для всех потоков блока, и имеет то же время жизни, что и блок. Все потоки имеют доступ к одной и той же глобальной памяти.

Также есть два дополнительных пространства памяти только для чтения, доступных для всех потоков: пространство констант и памяти текстуры. Пространства глобальной, константной и текстурной памяти оптимизированы для различного использования памяти (см. Доступ к памяти устройства). Память текстур также предлагает различные режимы адресации, а также фильтрацию данных для некоторых конкретных форматов данных (см. Память текстуры и поверхности).

Пространства глобальной, константной и текстурной памяти сохраняются при запуске ядра одним и тем же приложением.

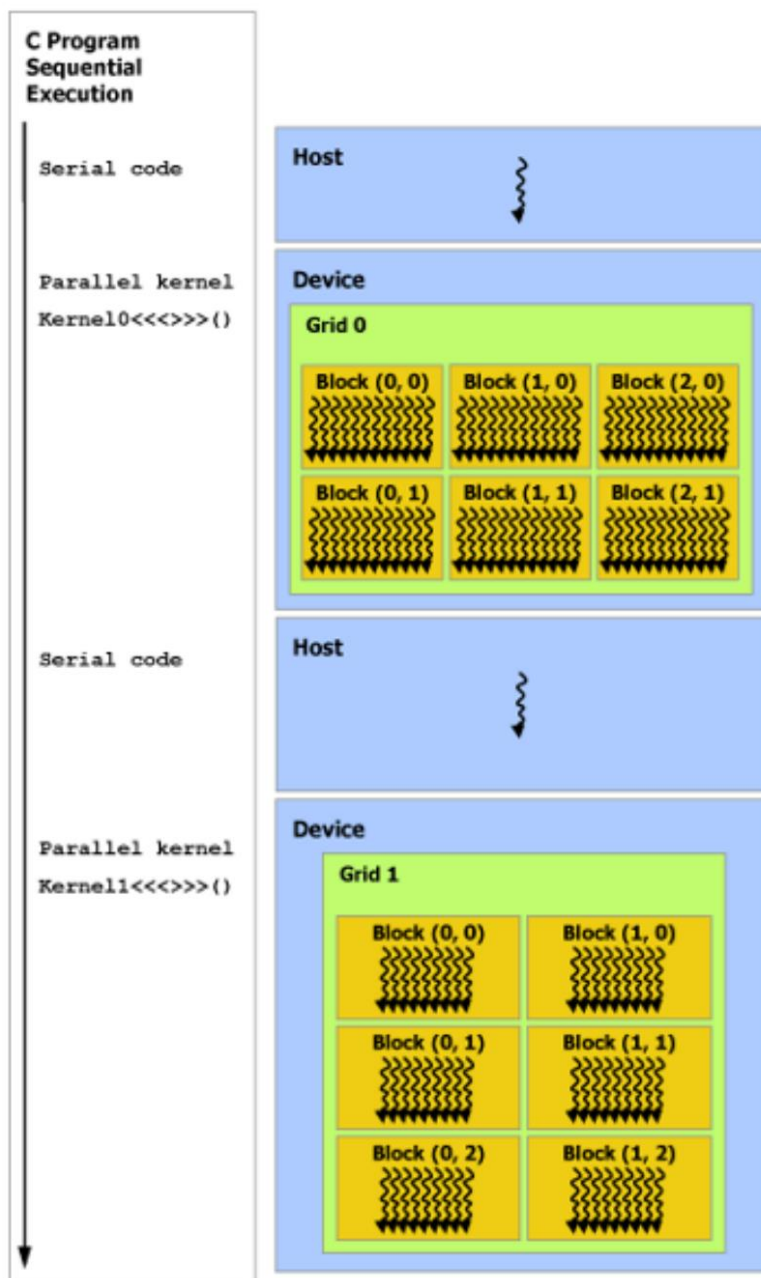
Рисунок 5. Иерархия памяти



## 2.4. Гетерогенное программирование

Как показано на рисунке 6, модель программирования CUDA предполагает, что потоки CUDA выполняются на физически отдельном устройстве, которое работает как сопроцессор для хоста, на котором выполняется программа C++. Это имеет место, например, когда ядра выполняются на графическом процессоре, а остальная часть программы C++ выполняется на процессоре.

Рисунок 6. Гетерогенное программирование



Модель программирования CUDA также предполагает, что и хост, и устройство поддерживают свои собственные отдельные области памяти в DRAM, называемые памятью хоста и памятью устройства соответственно. Следовательно, программа управляет пространством глобальной, константной и текстурной памяти, видимой ядрам посредством вызовов среды выполнения CUDA (описанной в Интерфейсе программирования). Это включает в себя выделение и освобождение памяти устройства, а также передачу данных между хостом и памятью устройства.

Unified Memory предоставляет управляемую память для объединения пространств памяти хоста и устройства. Управляемая память доступна для всех процессоров и графических процессоров в системе как единый согласованный образ памяти с общим адресным пространством. Эта возможность обеспечивает превышение лимита памяти устройства и может значительно упростить задачу переноса приложений, устраняя необходимость в явном зеркальном отображении данных на хосте и устройстве. См. Раздел Программирование унифицированной памяти для ознакомления с единой памятью.

Примечание. Последовательный код выполняется на хосте, а параллельный код выполняется на устройстве.

## 2.5. Вычислительные возможности

Вычислительные возможности устройства представлены номером версии, также иногда называемым «версией SM». Этот номер версии определяет функции, поддерживаемые аппаратным обеспечением графического процессора, и используется приложениями во время выполнения, чтобы определить, какие аппаратные функции и / или инструкции доступны на текущем графическом процессоре.

Вычислительные возможности включают основной номер редакции X и младший номер редакции Y и обозначаются X.Y.

Устройства с одинаковым номером основной версии имеют одинаковую архитектуру ядра. Основной номер ревизии - 8 для устройств на базе архитектуры NVIDIA Ampere GPU, 7 для устройств на базе архитектуры Volta, 6 для устройств на базе архитектуры Pascal, 5 для устройств на базе архитектуры Maxwell, 3 для устройств на базе Kepler архитектура, 2 для устройств на основе архитектуры Fermi и 1 для устройств на основе архитектуры Tesla.

Младший номер редакции соответствует постепенному улучшению основной архитектуры, возможно, включая новые функции.

Turing - это архитектура для устройств с вычислительными возможностями 7.5, которая представляет собой инкрементное обновление, основанное на архитектуре Volta.

К графическим процессорам перечисляют все устройства с поддержкой CUDA вместе с их вычислительными возможностями. Compute Capabilities дает технические характеристики каждой вычислительной возможности.

Примечание. Версию вычислительных возможностей конкретного графического процессора не следует путать с версией CUDA (например, CUDA 7.5, CUDA 8, CUDA 9), которая является версией программной платформы CUDA. Платформа CUDA используется разработчиками приложений для создания приложений, которые работают на многих поколениях архитектур графических

процессоров, включая будущие архитектуры графических процессоров, которые еще предстоит изобрести. В то время как новые версии платформы CUDA часто добавляют встроенную поддержку новой архитектуры GPU, поддерживая версию этой архитектуры с вычислительными возможностями, новые версии платформы CUDA обычно также включают программные функции, которые не зависят от поколения оборудования.

```
[knefedev@jupyter ~]$ nvidia-smi
Sun Feb 28 14:34:08 2021

+-----+
| NVIDIA-SMI 418.40.04      Driver Version: 418.40.04      CUDA Version: 10.1      |
+-----+-----+
| GPU   Name                Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan   Temp   Perf    Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
+-----+-----+
|    0   Tesla P100-SXM2...    On       | 00000002:01:00:0 Off  |            0         |
| N/A   35C    P0      39W / 300W | 15553MiB / 16280MiB |    0%      Default   |
+-----+-----+
|    1   Tesla P100-SXM2...    On       | 00000006:01:00:0 Off  |            0         |
| N/A   35C    P0      41W / 300W | 15553MiB / 16280MiB |    0%      Default   |
+-----+-----+

+-----+
| Processes:                                                       GPU Memory |
|  GPU       PID    Type   Process name                                             Usage   |
+-----+-----+
|    0      115634    C     ...nyh/.conda/envs/powerai-home/bin/python 15543MiB |
|    1      115634    C     ...nyh/.conda/envs/powerai-home/bin/python 15543MiB |
+-----+-----+
```

Архитектуры Tesla и Fermi больше не поддерживаются, начиная с CUDA 7.0 и CUDA 9.0 соответственно.

### 3. Интерфейс программирования.

CUDA C ++ предоставляет пользователям, знакомым с языком программирования C ++, простой способ легко писать программы для выполнения устройством.

Он состоит из минимального набора расширений языка C ++ и библиотеки времени выполнения.

Расширения базового языка были представлены в модели программирования. Они позволяют программистам определять ядро как функцию C ++ и использовать новый синтаксис для указания размера сетки и блока при каждом вызове функции. Полное описание всех расширений можно найти в Расширениях языка C ++. Любой исходный файл, содержащий некоторые из этих расширений, должен быть скомпилирован с помощью nvcc, как описано в разделе «Компиляция с помощью NVCC».

Среда выполнения представлена в CUDA Runtime. Он предоставляет функции C и C ++, которые выполняются на хосте для выделения и освобождения памяти устройства, передачи данных между памятью хоста и памятью устройства, управления системами с несколькими устройствами и т. Д. Полное описание среды выполнения можно найти в справочном руководстве CUDA.



Среда выполнения построена на основе C API нижнего уровня, API драйвера CUDA, который также доступен для приложения. API драйвера обеспечивает дополнительный уровень контроля, раскрывая концепции нижнего уровня, такие как контексты CUDA - аналог хост-процессов для устройства - и модули CUDA - аналог динамически загружаемых библиотек для устройства. Большинство приложений не используют API драйвера, поскольку им не нужен этот дополнительный уровень контроля, а при использовании среды выполнения управление контекстом и модулями неявно, что приводит к более сжатому коду. Поскольку среда выполнения совместима с API драйвера, большинство приложений, которым требуются некоторые функции API драйвера, могут по умолчанию использовать API среды выполнения и использовать API драйвера только там, где это необходимо. API-интерфейс драйвера представлен в API-интерфейсе драйвера и полностью описан в справочном руководстве.

### 3.1. Компиляция с NVCC

Ядра могут быть написаны с использованием архитектуры набора команд CUDA, называемой PTX, которая описана в справочном руководстве по PTX. Однако обычно более эффективно использовать язык программирования высокого уровня, такой как C ++. В обоих случаях ядра должны быть скомпилированы в двоичный код с помощью nvcc для выполнения на устройстве.

nvcc - это драйвер компилятора, который упрощает процесс компиляции кода C ++ или PTX: он предоставляет простые и знакомые параметры командной строки и выполняет их, вызывая набор инструментов, реализующих различные этапы компиляции. В этом разделе представлен обзор рабочего процесса nvcc и параметров команд. Полное описание можно найти в руководстве пользователя nvcc.

#### 3.1.1. Рабочий процесс компиляции

##### 3.1.1.1. Автономная компиляция

Исходные файлы, скомпилированные с помощью nvcc, могут включать сочетание кода хоста (т. е. кода, который выполняется на хосте) и кода устройства (т.е. кода, который выполняется на устройстве). Основной рабочий процесс nvcc состоит в отделении кода устройства от кода хоста, а затем:

компиляция кода устройства в форму сборки (код PTX) и / или двоичную форму (объект cubin), и изменение кода хоста путем замены синтаксиса <<< ... >>>, представленного в ядрах (и более подробно описанного в разделе "Конфигурация выполнения"), необходимыми вызовами функций времени выполнения CUDA для загрузки и запуска каждого скомпилированного ядра из кода PTX и / или кубинового объекта.

Измененный код хоста выводится либо как код C ++, который нужно скомпилировать с помощью другого инструмента, либо как объектный код напрямую, позволяя nvcc вызывать компилятор хоста на последнем этапе компиляции.

После этого приложения могут:

либо ссылаться на скомпилированный код хоста (это наиболее частый случай),

либо игнорировать измененный код хоста (если есть) и используйте API-интерфейс драйвера CUDA (см. API-интерфейс драйвера) для загрузки и выполнения кода PTX или объекта cubin.

#### 3.1.1.2. Своевременная компиляция

Любой код PTX, загруженный приложением во время выполнения, дополнительно компилируется в двоичный код драйвером устройства. Это называется своевременной компиляцией.

Своевременная компиляция увеличивает время загрузки приложения, но позволяет приложению извлекать выгоду из любых новых улучшений компилятора, поступающих с каждым новым драйвером устройства. Это также единственный способ запуска приложений на устройствах, которые не существовали на момент компиляции приложения, как подробно описано в разделе «Совместимость приложений».

Когда драйвер устройства точно в срок компилирует некоторый код PTX для некоторого приложения, он автоматически кэширует копию сгенерированного двоичного кода, чтобы избежать повторения компиляции при последующих вызовах приложения. Кэш, называемый кешем вычислений, автоматически становится недействительным при обновлении драйвера устройства, так что приложения могут извлечь выгоду из улучшений нового своевременного компилятора, встроенного в драйвер устройства.

Переменные среды доступны для управления своевременной компиляцией, как описано в разделе «Переменные среды CUDA».

В качестве альтернативы использованию nvcc для компиляции кода устройства CUDA C ++, NVRTC может использоваться для компиляции кода устройства CUDA C ++ в PTX во время выполнения. NVRTC - это библиотека компиляции времени выполнения для CUDA C ++; дополнительную информацию можно найти в руководстве пользователя NVRTC.

#### 3.1.2. Двоичная совместимость

Двоичный код зависит от архитектуры. Объект cubin создается с использованием опции компилятора -code, которая определяет целевую архитектуру: например, компиляция с -code = sm\_35 создает двоичный код для устройств с вычислительными возможностями 3.5. Двоичная совместимость гарантируется от одной незначительной ревизии к следующей, но не от одной младшей ревизии к предыдущей или между основными ревизиями. Другими словами, объект cubin, созданный для вычислительных возможностей X.y, будет выполняться только на устройствах с вычислительными возможностями X.z, где  $z \geq y$ .

Примечание. Двоичная совместимость поддерживается только для настольных компьютеров. Он не поддерживается для Tegra. Также не поддерживается двоичная совместимость между настольным компьютером и Tegra.

#### 3.1.3. Совместимость PTX



Некоторые инструкции PTX поддерживаются только на устройствах с более высокими вычислительными возможностями. Например, функции Warp Shuffle поддерживаются только на устройствах с вычислительной мощностью 3.0 и выше. Параметр компилятора `-arch` указывает вычислительные возможности, которые предполагается использовать при компиляции C++ в код PTX. Таким образом, код, содержащий, например, перемешивание деформации, должен быть скомпилирован с параметром `-arch = compute_30` (или выше).

Код PTX, созданный для определенных вычислительных возможностей, всегда можно скомпилировать в двоичный код с большей или равной вычислительной мощностью. Обратите внимание, что двоичный файл, скомпилированный из более ранней версии PTX, может не использовать некоторые аппаратные функции. Например, двоичные целевые устройства с вычислительными возможностями 7.0 (Volta), скомпилированные из PTX, сгенерированного для вычислительных возможностей 6.0 (Pascal), не будут использовать инструкции Tensor Core, поскольку они не были доступны на Pascal. В результате конечный двоичный файл может работать хуже, чем это было бы возможно, если бы двоичный файл был сгенерирован с использованием последней версии PTX.

#### 3.1.4. Совместимость приложений

Чтобы выполнить код на устройствах с определенными вычислительными возможностями, приложение должно загрузить двоичный код или код PTX, который совместим с этой вычислительной возможностью, как описано в разделах «Двоичная совместимость и совместимость с PTX». В частности, чтобы иметь возможность выполнять код на будущих архитектурах с более высокими вычислительными возможностями (для которых еще нельзя сгенерировать двоичный код), приложение должно загрузить код PTX, который будет своевременно скомпилирован для этих устройств (см. Just- своевременная компиляция).

Какой PTX и двоичный код внедряются в приложение CUDA C++, определяется параметрами компилятора `-arch` и `-code` или параметром компилятора `-gencode`, как подробно описано в руководстве пользователя `nvcc`. Например,

```
nvcc x.cu

-gencode arch=compute_50,code=sm_50

-gencode arch=compute_60,code=sm_60

-gencode arch=compute_70,code=\"compute_70,sm_70\"
```

включает двоичный код, совместимый с вычислительными возможностями 5.0 и 6.0 (первый и второй варианты `-gencode`), а также PTX и двоичный код, совместимый с вычислительными возможностями 7.0 (третий вариант `-gencode`).

Код хоста генерируется для автоматического выбора во время выполнения наиболее подходящего кода для загрузки и выполнения, который в приведенном выше примере будет:

Двоичный код 5.0 для устройств с вычислительными возможностями 5.0 и 5.2,

6.0 двоичный код для устройств с вычислительными возможностями 6.0 и 6.1,

7.0 двоичный код для устройств с вычислительными возможностями 7.0 и 7.5,

Код PTX, который компилируется в двоичный код во время выполнения для устройств с вычислительными возможностями 8.0 и 8.6.

x.cu может иметь оптимизированный путь кода, который, например, использует операции перемешивания деформации, которые поддерживаются только в устройствах с вычислительной мощностью 3.0 и выше. Макрос `__CUDA_ARCH__` может использоваться для различения различных путей кода на основе вычислительных возможностей. Он определен только для кода устройства. Например, при компиляции с `-arch = compute_35` `__CUDA_ARCH__` равно 350.

Приложения, использующие API драйвера, должны компилировать код для разделения файлов и явно загружать и выполнять наиболее подходящий файл во время выполнения.

Архитектура Volta вводит независимое планирование потоков, которое изменяет способ планирования потоков на GPU. Для кода, основанного на определенном поведении планирования SIMT в предыдущих архитектурах, независимое планирование потоков может изменить набор участвующих потоков, что приведет к неверным результатам. Чтобы облегчить миграцию при реализации корректирующих действий, подробно описанных в Независимом планировании потоков, разработчики Volta могут отказаться от планирования потоков Pascal с помощью комбинации параметров компилятора `-arch = compute_60 -code = sm_70`.

В руководстве пользователя nvcc перечислены различные сокращения для параметров компилятора `-arch`, `-code` и `-gencode`. Например, `-arch = sm_70` является сокращением для `-arch = compute_70 -code = compute_70, sm_70` (что совпадает с `-gencode arch = compute_70, code = 'compute_70, sm_70'`).

### 3.1.5. Совместимость с C ++

Передняя часть компилятора обрабатывает исходные файлы CUDA в соответствии с правилами синтаксиса C ++. Полный C ++ поддерживается для кода хоста. Однако только подмножество C ++ полностью поддерживается для кода устройства, как описано в разделе «Поддержка языка C ++».

### 3.1.6. 64-битная совместимость

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 с кодом хоста, скомпилированным в 64-битном режиме.

Точно так же 32-битная версия nvcc компилирует код устройства в 32-битном режиме, а код устройства, скомпилированный в 32-битном режиме, поддерживается только с кодом хоста, скомпилированным в 32-битном режиме.

32-разрядная версия nvcc может компилировать код устройства в 64-разрядном режиме, также используя параметр компилятора -m64.

64-разрядная версия nvcc может компилировать код устройства в 32-разрядном режиме, также используя параметр компилятора -m32.

## 3.2. CUDA Runtime

Среда выполнения реализована в библиотеке cudart, которая связана с приложением либо статически через cudart.lib или libcudart.a, либо динамически через cudart.dll или libcudart.so. Приложения, которым для динамической компоновки требуется cudart.dll и / или cudart.so, обычно включают их как часть установочного пакета приложения. Безопасно передавать адрес символов времени выполнения CUDA только между компонентами, которые ссылаются на один и тот же экземпляр среды выполнения CUDA.

Все его точки входа имеют префикс cuda.

Как упоминалось в разделе «Гетерогенное программирование», модель программирования CUDA предполагает систему, состоящую из хоста и устройства, каждое из которых имеет свою отдельную память. Память устройства дает обзор функций времени выполнения, используемых для управления памятью устройства.

Общая память иллюстрирует использование общей памяти, представленной в иерархии потоков, для максимальной производительности.

Память хоста с блокировкой страниц представляет собой память хоста с блокировкой страниц, которая требуется для перекрытия выполнения ядра с передачей данных между хостом и памятью устройства.

Асинхронное параллельное выполнение описывает концепции и API, используемые для обеспечения асинхронного одновременного выполнения на различных уровнях системы.

Система с несколькими устройствами показывает, как модель программирования распространяется на систему с несколькими устройствами, подключенными к одному хосту.

Проверка ошибок описывает, как правильно проверять ошибки, генерируемые средой выполнения.

В стеке вызовов упоминаются функции времени выполнения, используемые для управления стеком вызовов CUDA C ++.

Текстурная и поверхностная память представляет собой текстурные и поверхностные области памяти, которые обеспечивают другой способ доступа к памяти устройства; они также предоставляют подмножество аппаратного обеспечения текстурирования графического процессора.

Взаимодействие графики представляет различные функции, которые среда выполнения предоставляет для взаимодействия с двумя основными графическими API-интерфейсами, OpenGL и Direct3D.

### 3.2.1. Инициализация

Для среды выполнения нет явной функции инициализации; он инициализируется при первом вызове функции времени выполнения (точнее, любой функции, кроме функций из разделов обработки ошибок и управления версиями справочного руководства). Это необходимо иметь в виду при синхронизации вызовов функций времени выполнения и при интерпретации кода ошибки от первого вызова до среды выполнения.

Среда выполнения создает контекст CUDA для каждого устройства в системе (дополнительные сведения о контекстах CUDA см. В разделе «Контекст»). Этот контекст является основным контекстом для этого устройства и инициализируется первой функцией времени выполнения, которая требует активного контекста на этом устройстве. Он распределяется между всеми хост-потокками приложения. Как часть этого создания контекста, код устройства при необходимости компилируется точно в срок (см. Компиляция точно в срок) и загружается в память устройства. Все это происходит прозрачно. При необходимости, например для взаимодействия API драйвера к первичному контексту устройства можно получить доступ из API драйвера, как описано в разделе «Взаимодействие между API среды выполнения и API драйвера».

Когда хост-поток вызывает `cudaDeviceReset()`, это разрушает первичный контекст устройства, на котором в настоящее время работает хост-поток (то есть текущее устройство, как определено в разделе «Выбор устройства»). Следующий вызов функции времени выполнения, сделанный любым потоком хоста, который имеет это устройство как текущее, создаст новый первичный контекст для этого устройства.

Примечание. Интерфейсы CUDA используют глобальное состояние, которое инициализируется при запуске программы хоста и уничтожается во время завершения программы хоста. Среда выполнения CUDA и драйвер не могут определить, является ли это состояние недопустимым,

поэтому использование любого из этих интерфейсов (неявно или явно) во время запуска или завершения программы после основного) приведет к неопределенному поведению.

### 3.2.2. Память устройства

Как упоминалось в разделе «Гетерогенное программирование», модель программирования CUDA предполагает систему, состоящую из хоста и устройства, каждое из которых имеет свою отдельную память. Ядра работают вне памяти устройства, поэтому среда выполнения предоставляет функции для выделения, освобождения и копирования памяти устройства, а также передачи данных между памятью хоста и памятью устройства.

Память устройства может быть выделена как линейная память или как массивы CUDA.

Массивы CUDA - это непрозрачные макеты памяти, оптимизированные для выборки текстур. Они описаны в разделе «Память текстуры и поверхности».

Линейная память распределяется в едином унифицированном адресном пространстве, что означает, что отдельно выделенные объекты могут ссылаться друг на друга через указатели, например, в двоичном дереве или связанном списке. Размер адресного пространства зависит от хост-системы (ЦП) и вычислительных возможностей используемого графического процессора:

Table 1. Linear Memory Address Space

	x86_64 (AMD64)	POWER (ppc64le)	ARM64
up to compute capability 5.3 (Maxwell)	40bit	40bit	40bit
compute capability 6.0 (Pascal) or newer	up to 47bit	up to 49bit	up to 48bit

Примечание. На устройствах с вычислительной мощностью 5.3 (Maxwell) и ранее драйвер CUDA создает незафиксированное 40-битное резервирование виртуального адреса, чтобы гарантировать, что выделенная память (указатели) попадает в поддерживаемый диапазон. Это резервирование отображается как зарезервированная виртуальная память, но не занимает никакой физической памяти, пока программа не выделит память.

Линейная память обычно выделяется с помощью `cudaMalloc()` и освобождается с помощью `cudaFree()`, а передача данных между памятью хоста и памятью устройства обычно выполняется с помощью `cudaMemcpy()`. В образце кода сложения векторов ядер, векторы необходимо скопировать из памяти хоста в память устройства:

```
// 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
...
}

```

Линейную память также можно выделить с помощью `cudaMallocPitch()` и `cudaMalloc3D()`. Эти функции рекомендуются для выделения 2D- или 3D-массивов, так как это обеспечивает надлежащее заполнение выделения для удовлетворения требований выравнивания, описанных в разделе Доступ к памяти устройства, что обеспечивает лучшую производительность при доступе к адресам строк или выполнении копий между 2D-массивами и другими регионами. памяти устройства (с помощью функций `cudaMemcpy2D()` и `cudaMemcpy3D()`). Возвращенный шаг (или шаг) должен использоваться для доступа к элементам массива. В следующем примере кода выделяется двумерный массив значений с плавающей запятой шириной  $x$  высотой  $y$  и показано, как перебирать элементы массива в коде устройства:

```

// 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];
        }
    }
}
```

В следующем примере кода выделяется трехмерный массив значений с плавающей запятой шириной x высотой x глубиной и показано, как перебирать элементы массива в коде устройства:

```
// 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];
            }
        }
    }
}
```

Примечание. Чтобы не выделять слишком много памяти и, таким образом, не влиять на производительность всей системы, запросите параметры выделения у пользователя в зависимости от размера проблемы. Если выделение не удастся, вы можете вернуться к другим более медленным типам памяти (`cudaMallocHost()`, `cudaHostRegister()` и т. Д.) Или вернуть сообщение об ошибке, сообщающее пользователю, сколько памяти было необходимо, но было отказано. Если ваше приложение не может запросить параметры распределения по какой-либо причине, мы рекомендуем использовать `cudaMallocManaged()` для платформ, которые его поддерживают.

В справочном руководстве перечислены все различные функции, используемые для копирования памяти между линейной памятью, выделенной с помощью `cudaMalloc()`, линейной памятью, выделенной с помощью `cudaMallocPitch()` или `cudaMalloc3D()`, массивами CUDA и памятью, выделенной для переменных, объявленных в глобальном или постоянном пространстве памяти.

В следующем примере кода показаны различные способы доступа к глобальным переменным через API (**API** (программный интерфейс приложения, интерфейс прикладного

программирования) (англ. application programming interface, **API** [эй-пи-ай]) — описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой.) среды 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 глобальном пространстве памяти. Размер выделенной памяти получается с помощью `cudaGetSymbolSize ()`.

### 3.2.3. Управление доступом L2 к памяти устройства

Когда ядро CUDA повторно обращается к области данных в глобальной памяти, такие обращения к данным можно рассматривать как постоянные. С другой стороны, если доступ к данным осуществляется только один раз, такие обращения к данным можно рассматривать как потоковые.

Начиная с CUDA 11.0, устройства с вычислительной мощностью 8.0 и выше имеют возможность влиять на сохраняемость данных в кэше L2, потенциально обеспечивая более высокую пропускную способность и меньшую задержку доступа к глобальной памяти.

#### 3.2.3.1. Кэш L2, выделенный для постоянных доступов

Часть кэша L2 можно выделить для использования для сохранения доступа к данным в глобальной памяти. При постоянном доступе приоритет отдается использованию этой отложенной части кэша L2, тогда как при обычном или потоковом доступе к глобальной памяти эта часть L2 может использоваться только тогда, когда она не используется постоянными доступами.

Размер кэша L2, зарезервированный для постоянных доступов, может быть скорректирован в следующих пределах:

```
cudaGetDeviceProperties(&prop, device_id);
```



```
size_t size = min(int(prop.l2CacheSize * 0.75), prop.persistingL2CacheMaxSize);
cudaDeviceSetLimit(cudaLimitPersistingL2CacheSize, size); /* set-aside 3/4 of L2
cache for persisting accesses or the max allowed*/
```

Когда графический процессор настроен в режиме многоэкземплярного графического процессора (MIG), функция резервирования кэша второго уровня отключена.

При использовании многопроцессорной службы (MPS) размер зарезервированного кэша L2 не может быть изменен `cudaDeviceSetLimit`. Вместо этого отложенный размер можно указать только при запуске сервера MPS через переменную среды `CUDA_DEVICE_DEFAULT_PERSISTING_L2_CACHE_PERCENTAGE_LIMIT`.

### *3.2.3.2. Политика L2 для постоянных доступов*

Окно политики доступа определяет непрерывную область глобальной памяти и свойство постоянства в кэше L2 для доступа в этой области.

В приведенном ниже примере кода показано, как установить постоянное окно доступа L2 с помощью потока CUDA.

#### Пример CUDA Stream

```
cudaStreamAttrValue stream_attribute; //
Stream level attributes data structure

stream_attribute.accessPolicyWindow.base_ptr = reinterpret_cast<void*>(ptr); //
Global Memory data pointer

stream_attribute.accessPolicyWindow.num_bytes = num_bytes; //
Number of bytes for persistence access.

//
(Must be less than cudaDeviceProp::accessPolicyMaxWindowSize)

stream_attribute.accessPolicyWindow.hitRatio = 0.6; //
Hint for cache hit ratio

stream_attribute.accessPolicyWindow.hitProp = cudaAccessPropertyPersisting; //
Type of access property on cache hit

stream_attribute.accessPolicyWindow.missProp = cudaAccessPropertyStreaming; //
Type of access property on cache miss.

//Set the attributes to a CUDA stream of type cudaStream_t
```

```
cudaStreamSetAttribute(stream, cudaStreamAttributeAccessPolicyWindow,
&stream_attribute);
```

Когда ядро впоследствии выполняется в потоке CUDA, обращения к памяти в пределах глобальной памяти [ptr..ptr + num\_bytes) с большей вероятностью сохранятся в кэше L2, чем обращения к другим областям глобальной памяти.

Сохраняемость L2 также может быть установлена для узла ядра CUDA Graph, как показано в примере ниже:

Пример CUDA GraphKernelNode

```
cudaKernelNodeAttrValue node_attribute; //
Kernel level attributes data structure

node_attribute.accessPolicyWindow.base_ptr = reinterpret_cast<void*>(ptr); //
Global Memory data pointer

node_attribute.accessPolicyWindow.num_bytes = num_bytes; //
Number of bytes for persistence access.

//
(Must be less than cudaDeviceProp::accessPolicyMaxWindowSize)

node_attribute.accessPolicyWindow.hitRatio = 0.6; //
Hint for cache hit ratio

node_attribute.accessPolicyWindow.hitProp = cudaAccessPropertyPersisting; //
Type of access property on cache hit

node_attribute.accessPolicyWindow.missProp = cudaAccessPropertyStreaming; //
Type of access property on cache miss.

//Set the attributes to a CUDA Graph Kernel node of type cudaGraphNode_t

cudaGraphKernelNodeSetAttribute(node, cudaKernelNodeAttributeAccessPolicyWindow,
&node_attribute);
```

Параметр hitRatio можно использовать для указания доли обращений, получающих свойство hitProp. В обоих приведенных выше примерах 60% обращений к памяти в области глобальной памяти [ptr..ptr + num\_bytes) имеют свойство сохранения, а 40% обращений к памяти имеют свойство потоковой передачи. Какой конкретный доступ к памяти классифицируется как постоянный (hitProp), является случайным с вероятностью приблизительно hitRatio; распределение вероятностей зависит от архитектуры оборудования и объема памяти.

Например, если размер отложенного кэша L2 составляет 16 КБ, а значение num\_bytes в accessPolicyWindow равно 32 КБ:

При коэффициенте попадания 0,5 аппаратное обеспечение будет случайным образом выбирать 16 Кбайт из 32 Кбайт окна, которое будет обозначено как сохраняющееся и кэшируемое в зарезервированной области кэша L2.

При коэффициенте попадания 1.0 оборудование будет пытаться кэшировать все окно размером 32 КБ в выделенной области кэша L2. Поскольку зарезервированная область меньше окна, строки кэша будут исключены, чтобы сохранить последние использованные 16 Кбайт из 32 Кбайт данных в зарезервированной части кэша L2.

Таким образом, HitRatio можно использовать, чтобы избежать перегрузки строк кэша и в целом уменьшить объем данных, перемещаемых в кэш L2 и из него.

Значение hitRatio ниже 1,0 можно использовать для ручного управления объемом данных, которые разные accessPolicyWindows из параллельных потоков CUDA могут кэшировать в L2. Например, пусть размер отложенного кэша L2 составляет 16 КБ; два параллельных ядра в двух разных потоках CUDA, каждый с 16 КБ accessPolicyWindow и оба со значением hitRatio 1.0, могут вытеснять строки кэша друг друга при конкуренции за общий ресурс L2. Однако, если оба accessPolicyWindows имеют значение HitRatio 0,5, они с меньшей вероятностью вытеснят свои собственные или постоянные строки кэша друг друга.

#### *3.2.3.3. Свойства доступа L2*

Для разных обращений к данным глобальной памяти определены три типа свойств доступа:

cudaAccessPropertyStreaming: доступы к памяти, которые происходят со свойством потоковой передачи, с меньшей вероятностью будут сохраняться в кэше L2, поскольку эти обращения предпочтительно исключены.

cudaAccessPropertyPersisting: доступы к памяти, которые происходят с сохраняющимся свойством, с большей вероятностью сохраняются в кэше L2, поскольку эти обращения предпочтительно сохраняются в отложенной части кэша L2.

cudaAccessPropertyNormal: это свойство доступа принудительно сбрасывает ранее примененное свойство постоянного доступа к нормальному состоянию. Доступ к памяти с сохранением свойства из предыдущих ядер CUDA может сохраняться в кэше L2 еще долгое время после их предполагаемого использования. Это постоянство после использования уменьшает объем кэша L2, доступного для последующих ядер, которые не используют свойство сохранения. Сброс окна свойств доступа с помощью свойства cudaAccessPropertyNormal удаляет постоянный (преимущественное сохранение) статус предыдущего доступа, как если бы предыдущий доступ был без свойства доступа.

#### *3.2.3.4. Пример сохранения L2*

В следующем примере показано, как выделить кэш L2 для постоянного доступа, использовать отложенный кеш L2 в ядрах CUDA через поток CUDA, а затем сбросить кеш L2.

```
cudaStream_t stream;
```

```

cudaStreamCreate(&stream);
// Create CUDA stream

cudaDeviceProp prop;
// CUDA device properties variable

cudaGetDeviceProperties( &prop, device_id);
// Query GPU properties

size_t size = min( int(prop.l2CacheSize * 0.75) , prop.persistingL2CacheMaxSize
);

cudaDeviceSetLimit( cudaLimitPersistingL2CacheSize, size);
// set-aside 3/4 of L2 cache for persisting accesses or the max allowed

size_t window_size = min(prop.accessPolicyMaxWindowSize, num_bytes);
// Select minimum of user defined num_bytes and max window size.

cudaStreamAttrValue stream_attribute;
// Stream level attributes data structure

stream_attribute.accessPolicyWindow.base_ptr = reinterpret_cast<void*>(data1);
// Global Memory data pointer

stream_attribute.accessPolicyWindow.num_bytes = window_size;
// Number of bytes for persistence access

stream_attribute.accessPolicyWindow.hitRatio = 0.6;
// Hint for cache hit ratio

stream_attribute.accessPolicyWindow.hitProp = cudaAccessPropertyPersisting;
// Persistence Property

stream_attribute.accessPolicyWindow.missProp = cudaAccessPropertyStreaming;
// Type of access property on cache miss

cudaStreamSetAttribute(stream, cudaStreamAttributeAccessPolicyWindow,
&stream_attribute); // Set the attributes to a CUDA Stream

for(int i = 0; i < 10; i++) {

    cuda_kernelA<<<grid_size,block_size,0,stream>>>(data1);
// This data1 is used by a kernel multiple times

}
// [data1 + num_bytes) benefits from L2 persistence

cuda_kernelB<<<grid_size,block_size,0,stream>>>(data1);
// A different kernel in the same stream can also benefit

```

```
// from the persistence of data1

stream_attribute.accessPolicyWindow.num_bytes = 0;
// Setting the window size to 0 disable it

cudaStreamSetAttribute(stream, cudaStreamAttributeAccessPolicyWindow,
&stream_attribute); // Overwrite the access policy attribute to a CUDA Stream

cudaCtxResetPersistingL2Cache();
// Remove any persistent lines in L2

cuda_kernelC<<<grid_size,block_size,0,stream>>>(data2);
// data2 can now benefit from full L2 in normal mode
```

#### 3.2.3.5. Сбросить доступ L2 к нормальному

Постоянная строка кэша L2 из предыдущего ядра CUDA может сохраняться в L2 еще долгое время после того, как она была использована. Следовательно, возврат к нормальному состоянию для кэша L2 важен для потоковой передачи или обычного доступа к памяти, чтобы использовать кэш L2 с нормальным приоритетом. Есть три способа вернуть постоянный доступ к нормальному состоянию.

Сбросьте предыдущую сохраняемую область памяти с помощью свойства доступа `cudaAccessPropertyNormal`.

Сбросьте все сохраняющиеся строки кэша L2 в нормальное состояние, вызвав `cudaCtxResetPersistingL2Cache()`.

В конце концов, нетронутые линии автоматически возвращаются в нормальное состояние. Настоятельно не рекомендуется использовать автоматический сброс из-за неопределенного времени, необходимого для выполнения автоматического сброса.

#### 3.2.3.6. Управление использованием отложенного кэша L2

Несколько ядер CUDA, выполняющихся одновременно в разных потоках CUDA, могут иметь разные окна политики доступа, назначенные их потокам. Однако часть отложенного кэша L2 используется всеми этими параллельными ядрами CUDA. В результате чистое использование этой отложенной части кэша является суммой индивидуального использования всех параллельных ядер. Преимущества обозначения доступов к памяти как постоянных уменьшаются по мере того, как объем постоянных доступов превышает выделенную емкость кэша L2.

Чтобы управлять использованием выделенной части кэша L2, приложение должно учитывать следующее:

Размер отложенного кэша L2.

Ядра CUDA, которые могут выполняться одновременно.

Окно политики доступа для всех ядер CUDA, которые могут выполняться одновременно.

Когда и как требуется сброс L2, чтобы разрешить нормальный или потоковый доступ для использования ранее отложенного кэша L2 с равным приоритетом.

#### *3.2.3.7. Свойства кэша L2 запроса*

Свойства, связанные с кешем L2, являются частью структуры `cudaDeviceProp` и могут быть запрошены с помощью API времени выполнения CUDA `cudaGetDeviceProperties`

Свойства устройства CUDA включают:

`l2CacheSize`: объем доступного кэша L2 на графическом процессоре.

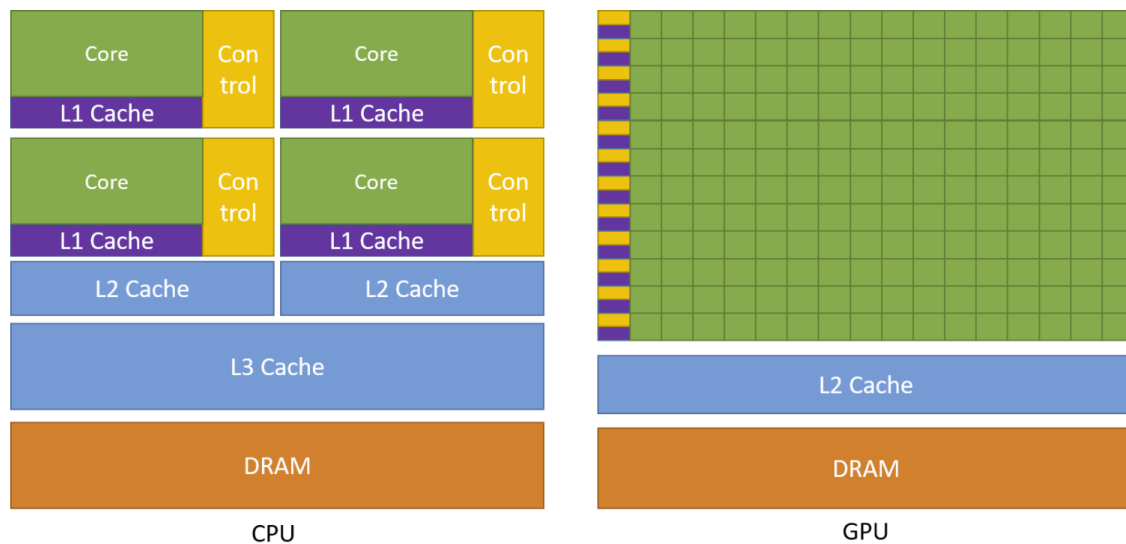
`persistingL2CacheMaxSize`: максимальный объем кэша L2, который можно выделить для постоянных обращений к памяти.

`accessPolicyMaxWindowSize`: максимальный размер окна политики доступа.

#### *3.2.3.8. Управление размером зарезервированного кэша L2 для постоянного доступа к памяти*

Размер отложенного кэша L2 для постоянных обращений к памяти запрашивается с помощью API среды выполнения CUDA `cudaDeviceGetLimit` и устанавливается с помощью API среды выполнения CUDA `cudaDeviceSetLimit` в качестве `cudaLimit`. Максимальное значение для установки этого ограничения - `cudaDeviceProp :: persistingL2CacheMaxSize`.

```
enum cudaLimit {  
    /* other fields not shown */  
    cudaLimitPersistingL2CacheSize  
};
```



### 3.2.4. Общая память

Как подробно описано в Спецификаторах пространства переменной памяти, общая память выделяется с помощью спецификатора пространства памяти `__shared__`.

Ожидается, что общая память будет намного быстрее, чем глобальная память, как указано в разделе «Иерархия потоков» и подробно описано в разделе «Общая память». Его можно использовать в качестве оперативной памяти (или программного управляемого кеша), чтобы минимизировать доступ к глобальной памяти из блока CUDA, как показано в следующем примере умножения матриц.

Следующий пример кода представляет собой прямую реализацию умножения матриц, в которой не используется общая память. Каждый поток считывает одну строку A и один столбец B и вычисляет соответствующий элемент C, как показано на рисунке 7. Таким образом, A считывается B. раз по ширине из глобальной памяти, а B считывается A по высоте.

```
// 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);

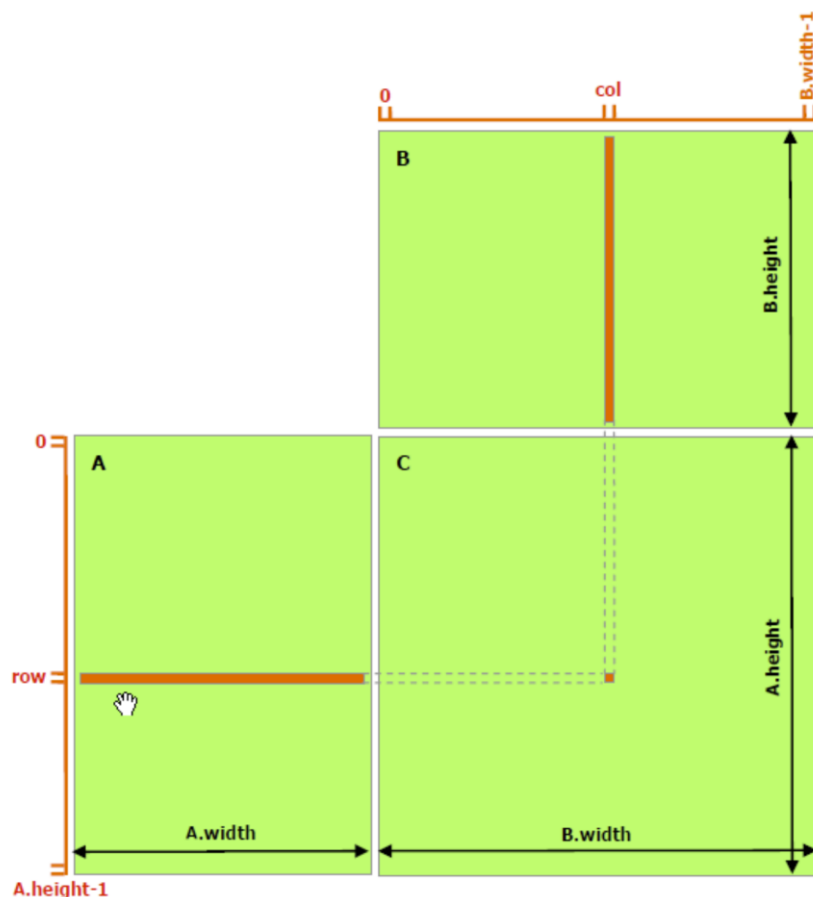
    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size,
               cudaMemcpyDeviceToHost);

    // 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)
{
    // 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;
}

```

Рисунок 7. Умножение матриц без общей памяти



Следующий пример кода представляет собой реализацию умножения матриц, в которой используется общая память. В этой реализации каждый блок потока отвечает за вычисление одной квадратной подматрицы  $C_{sub}$  из  $C$ , а каждый поток внутри блока отвечает за вычисление одного элемента  $C_{sub}$ . Как показано на рисунке 8,  $C_{sub}$  равно произведению двух прямоугольных матриц: подматрицы  $A$  размерности  $(A.width, block\_size)$ , которая имеет те же индексы строк, что и  $C_{sub}$ , и подматрицы  $B$  размерности  $(block\_size, A.width)$ , который имеет те же индексы столбца, что и  $C_{sub}$ . Чтобы соответствовать ресурсам устройства, эти две прямоугольные матрицы делятся на столько квадратных матриц размерности  $block\_size$ , сколько необходимо, и  $C_{sub}$  вычисляется как сумма произведений этих квадратных матриц. Каждый из этих продуктов выполняется, сначала загружая две соответствующие квадратные матрицы из глобальной памяти в общую память, причем один поток загружает один элемент каждой матрицы, а затем каждый поток вычисляет один элемент продукта. Каждый поток накапливает результат каждого из этих продуктов в регистре и после этого записывает результат в глобальную память.

Блокируя вычисления таким образом, мы пользуемся преимуществами быстрой разделяемой памяти и сохраняем большую пропускную способность глобальной памяти, поскольку  $A$  только считывается  $(B.width / block\_size)$  раз из глобальной памяти, а  $B$  читается  $(A.height / block\_size)$  раз.

Тип `Matrix` из предыдущего примера кода дополнен полем шага, так что субматрицы могут быть эффективно представлены одним и тем же типом. Функции `__device__` используются для получения и установки элементов и построения любой подматрицы из матрицы.

```

// 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

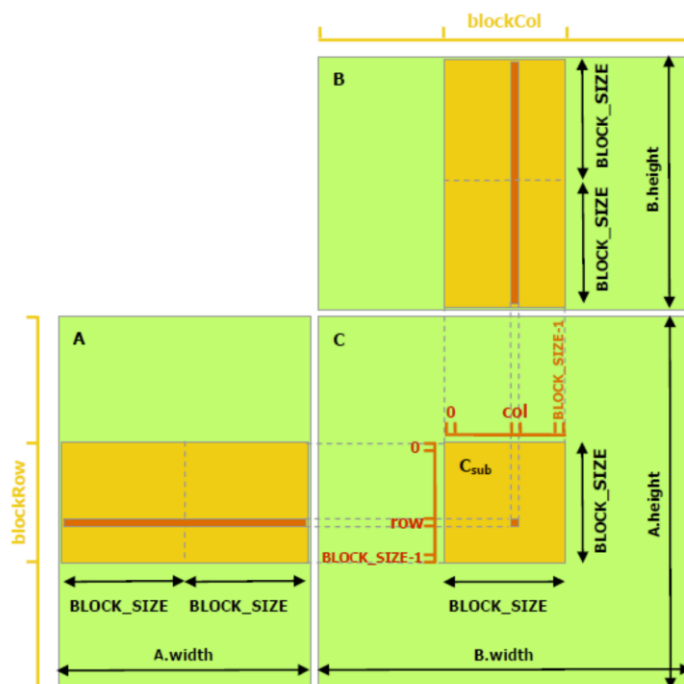
    for (int e = 0; e < BLOCK_SIZE; ++e)
        Cvalue += As[row][e] * Bs[e][col];

    // 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);
}

```

Рисунок 8. Умножение матриц с общей памятью



### 3.2.5. Память хоста с блокировкой страниц

Среда выполнения предоставляет функции, позволяющие использовать заблокированную на странице (также известную как закрепленная) память хоста (в отличие от обычной страничной памяти хоста, выделенной с помощью `malloc ()`):

`cudaHostAlloc ()` и `cudaFreeHost ()` выделяют и освобождают память хоста с блокировкой страниц;

Страница `cudaHostRegister ()` блокирует диапазон памяти, выделенной `malloc ()` (ограничения см. в справочном руководстве).

Использование памяти хоста с блокировкой страниц имеет несколько преимуществ:

Копии между памятью хоста с блокировкой страниц и памятью устройства могут выполняться одновременно с выполнением ядра для некоторых устройств, как упомянуто в разделе «Асинхронное параллельное выполнение».

На некоторых устройствах память хоста с блокировкой страниц может быть отображена в адресное пространство устройства, что устраняет необходимость копировать ее в память устройства или из нее, как подробно описано в разделе «Отображаемая память».

В системах с внешней шиной полоса пропускания между памятью хоста и памятью устройства выше, если память хоста выделяется как с блокировкой страниц, и даже выше, если она дополнительно выделяется как с объединением записи, как описано в разделе «Память с объединением записи».

Однако память хоста с блокировкой страниц является дефицитным ресурсом, поэтому выделения в памяти с блокировкой страниц начнут давать сбой задолго до выделения в страничной памяти. Кроме того, из-за уменьшения объема физической памяти, доступной операционной системе для подкачки, потребление слишком большого объема памяти с блокировкой страниц снижает общую производительность системы.

Примечание. Память хоста с блокировкой страниц не кэшируется на некогерентных устройствах ввода-вывода Tegra. Кроме того, `cudaHostRegister ()` не поддерживается на устройствах Tegra без когерентного ввода-вывода.

Простой пример CUDA с нулевым копированием поставляется с подробным документом по API-интерфейсам памяти с блокировкой страниц.

< Previous | [Next](#) >

cuRAND ([PDF](#)) - v11.2.1 ([older](#)) - Last updated February 9, 2021 - [Send Feedback](#)

## cuRAND

The API reference guide for cuRAND, the CUDA random number generation library.



## Table of Contents

- [Introduction](#)
- [1. Compatibility and Versioning](#)
- [2. Host API Overview](#)
  - [2.1. Generator Types](#)
  - [2.2. Generator Options](#)
    - [2.2.1. Seed](#)
    - [2.2.2. Offset](#)
    - [2.2.3. Order](#)
  - [2.3. Return Values](#)
  - [2.4. Generation Functions](#)
  - [2.5. Host API Example](#)
  - [2.6. Static Library support](#)
  - [2.7. Performance Notes](#)
- [3. Device API Overview](#)
  - [3.1. Pseudorandom Sequences](#)
    - [3.1.1. Bit Generation with XORWOW and MRG32k3a generators](#)
    - [3.1.2. Bit Generation with the MTGP32 generator](#)
    - [3.1.3. Bit Generation with Philox 4x32 10 generator](#)
    - [3.1.4. Distributions](#)
  - [3.2. Quasirandom Sequences](#)
  - [3.3. Skip-Ahead](#)
  - [3.4. Device API for discrete distributions](#)
  - [3.5. Performance Notes](#)
  - [3.6. Device API Examples](#)
  - [3.7. Thrust and cuRAND Example](#)
  - [3.8. Poisson API Example](#)
- [4. Testing](#)
- [5. Modules](#)
  - [5.1. Host API](#)
  - [5.2. Device API](#)
- [A. Bibliography](#)
- [B. Acknowledgements](#)

### Вступление

Библиотека cuRAND предоставляет средства, ориентированные на простое и эффективное генерирование высококачественных псевдослучайных и квазислучайных чисел. Псевдослучайная последовательность чисел удовлетворяет большинству статистических свойств истинно случайной последовательности, но генерируется детерминированным алгоритмом. Квазислучайная последовательность  $n$ -мерных точек генерируется детерминированным алгоритмом, предназначенным для равномерного заполнения  $n$ -мерного пространства.

cuRAND состоит из двух частей: библиотеки на стороне хоста (CPU) и файла заголовка устройства (GPU). Библиотека на стороне хоста рассматривается как любая другая библиотека ЦП: пользователи включают файл заголовка, `/include/curand.h`, чтобы получить объявления функций, а затем связываются с библиотекой. Случайные числа могут быть сгенерированы на устройстве или на центральном процессоре. Для генерации устройства вызовы библиотеки происходят на хосте,

но фактическая работа по генерации случайных чисел происходит на устройстве. Полученные случайные числа сохраняются в глобальной памяти устройства. Затем пользователи могут вызывать свои собственные ядра, чтобы использовать случайные числа, или они могут копировать случайные числа обратно на хост для дальнейшей обработки. Для генерации ЦП хоста вся работа выполняется на хосте, а случайные числа хранятся в памяти хоста.

Вторая часть cuRAND - это файл заголовка устройства `/include/curand_kernel.h`. Этот файл определяет функции устройства для настройки состояний генератора случайных чисел и генерации последовательностей случайных чисел. Код пользователя может включать этот файл заголовка, а написанные пользователем ядра могут затем вызывать функции устройства, определенные в файле заголовка. Это позволяет генерировать случайные числа и немедленно использовать их ядрами пользователя, не требуя записи случайных чисел и последующего чтения из глобальной памяти.

## 1. Совместимость и управление версиями

Хост-интерфейс cuRAND предназначен для обеспечения обратной совместимости на уровне исходного кода с будущими выпусками (если иное не указано в примечаниях к выпуску конкретного будущего выпуска). Другими словами, если программа использует cuRAND, она должна продолжать компилироваться и правильно работать с новыми версиями cuRAND без изменения исходного кода.

Обратная совместимость cuRAND на двоичном уровне не гарантируется. Использование разных версий заголовочного файла `curand.h` и общей библиотеки не поддерживается. Использование разных версий cuRAND и среды выполнения CUDA не поддерживается.

API устройства в большинстве случаев должен быть обратно совместимым на уровне исходного кода для общедоступных функций.

<http://ray-tracing.ru/articles199.html>

## What is the maximum CUDA Stack frame size per Kerenl.

Accelerated ComputingCUDACUDA Programming and Performance

[Home](#)

[Nov 2013](#)

**2 / 2**

Nov 2013

[Nov 2013](#)



[rohit89](#)

[Nov '13](#)

I am working on a Project which uses local arrays of size almost : 135200 Bytes,inside the cuda kernel ;

The problem comes when the application size is increased, which increases the local array size to 320,000 Bytes leading to kernel Launch Failures; Similarly, it happens for other increases in the application sizes.

i have tried not using local arrays, but that increases the computation so much that the speed is almost halved.

I have tried to increase the Stack Size Limit by setting the variable `cudaLimitStackSize` using the function : `cudaDeviceSetLimit`, but of no avail.

i have found some comments regarding the matter on : <http://stackoverflow.com/questions/7810740/where-does-cuda-allocate-the-stack-frame-for-kernels> 25

But the issue is not getting solved. So any thoughts ?.

Reply

- *created*

 Nov '13

- 

*last reply*

 Nov '13

- 1

*reply*

- 7.2k

*views*

- 2

*users*

- 2

*links*



[njuffa](#) Top Contributor

Nov '13

The compiler reports stack frame usage on a per-thread basis. The maximum stack frame size per thread for a given GPU is determined by (a) a hard architecture limit on the amount of local memory per thread (b) the amount of available GPU memory.

The architectural limit on the amount of local memory per thread is documented in the programming guide section G.1, table 12.

<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications> 156

Available stack frame size per thread can then be approximated by

stack frame size available per thread =

min (amount of local memory per thread as documented in section G.1 table 12,  
available GPU memory / number of SMs / maximum resident threads per SM)

The reason this is approximate is because there are various levels of allocation granularity that, best I know, are not documented and may vary from GPU to GPU. I do not know anything about your use case, but in general massive local memory usage would suggest to me that one might want to re-think the mapping of work to the GPU.