Бесплатная электронная книга

УЧУСЬ

cuda

Free unaffiliated eBook created from
Stack Overflow contributors.

#cuda

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: cuda

It is an unofficial and free cuda ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official cuda.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с cuda

замечания

CUDA - это собственная технология параллельных вычислений NVIDIA и язык программирования для своих графических процессоров.

Графические процессоры - это высокопараллельные машины, способные параллельно запускать тысячи легких потоков. Каждый поток графического процессора, как правило, медленнее в исполнении, а их контекст меньше. С другой стороны, GPU может запускать несколько тысяч потоков параллельно и даже больше одновременно (точные числа зависят от фактической модели графического процессора). CUDA - диалект C ++, разработанный специально для архитектуры графического процессора NVIDIA. Однако из-за различий в архитектуре большинство алгоритмов нельзя просто скопировать с простого C ++ - они будут работать, но будут очень медленными.

терминология

- *host* - относится к обычным аппаратным средствам на базе процессоров и обычным программам, которые запускаются в этой среде
- *device* - относится к определенному графическому процессору, в котором запускаются программы CUDA. Один хост может поддерживать несколько устройств.
- *kernel* - функция, которая находится на устройстве, которое может быть вызвано из главного кода.

Структура физического процессора

Процессор с графическим процессором с поддержкой CUDA имеет следующую физическую структуру:

- *чип* - весь процессор GPU. Некоторые графические процессоры имеют два из них.
- *потоковый мультипроцессор* (SM) - каждый чип содержит до ~ 100 SM, в зависимости от модели. Каждый SM работает практически независимо от другого, используя только глобальную память для связи друг с другом.
- *Ядро CUDA* - единая скалярная вычислительная единица SM. Их точное количество зависит от архитектуры. Каждое ядро может обрабатывать несколько потоков, выполняемых одновременно в быстрой последовательности (аналогично гиперпотоку в CPU).

Кроме того, каждый SM имеет один или несколько *планировщиков деформаций* . Каждый планировщик отправляет одну команду в несколько ядер CUDA. Это фактически

заставляет SM работать в 32- разрядном режиме SIMD .

Модель исполнения CUDA

Физическая структура GPU оказывает прямое влияние на то, как ядра выполняются на устройстве, и как один из них реализует их в CUDA. Ядро вызывается с *конфигурацией вызова,* которая определяет количество параллельных потоков.

- *сетка* - представляет все потоки, которые порождаются при вызове ядра. Он задается как один или два различных набора *блоков*
- *блок* - это полунезависимый набор *потоков* . Каждому блоку присваивается один SM. Таким образом, блоки могут связываться только через глобальную память. Блоки никак не синхронизированы. Если слишком много блоков, некоторые могут выполняться последовательно после других. С другой стороны, если позволяют ресурсы, более одного блока может работать на одном и том же SM, но программист не может извлечь выгоду из этого (кроме очевидного повышения производительности).
- *поток* - скалярная последовательность инструкций, выполняемых одним ядром CUDA. Темы «легкие» с минимальным контекстом, позволяя аппаратным средствам быстро менять их. Из-за их количества, потоки CUDA работают с несколькими зарегистрированными регистрами и очень коротким стеком (предпочтительно вообще нет!). По этой причине компилятор CUDA предпочитает встроить все вызовы функций, чтобы сгладить ядро так, чтобы оно содержало только статические прыжки и циклы. Функциональные вызовы ponter и вызовы виртуальных методов, поддерживаемые на большинстве более новых устройств, обычно несут большую эффективность.

Каждый поток идентифицируется блочным индексом `blockIdx` и индексом потока внутри блока `threadIdx` . Эти числа могут быть проверены в любой момент любым бегущим потоком и являются единственным способом отличить один поток от другого.

Кроме того, потоки организованы в *основы* , каждая из которых содержит ровно 32 потока. Нити внутри одной основы выполняются в идеальной синхронизации, в SIMD fahsion. Нити разных разломов, но внутри одного и того же блока могут выполняться в любом порядке, но могут быть принудительно синхронизированы программистом. Нити из разных блоков нельзя синхронизировать или напрямую взаимодействовать.

Организация памяти

В обычном программировании процессора организация памяти обычно скрыта от программиста. Типичные программы действуют так, как будто есть только ОЗУ. Все операции с памятью, такие как управление реестрами, использование L1-L2-L3-кэширования, свопинг на диск и т. Д. Обрабатываются компилятором, операционной

системой или оборудованием.

Это не относится к CUDA. В то время как новые модели графических процессоров частично скрывают нагрузку, например, через Unified Memory в CUDA 6, по-прежнему стоит понимать организацию по соображениям производительности. Основная структура памяти CUDA выглядит следующим образом:

- *Хост-память* - обычная оперативная память. В основном используется хост-код, но новые модели графических процессоров также могут получить к нему доступ. Когда ядро получает доступ к памяти хоста, графический процессор должен взаимодействовать с материнской платой, как правило, через разъем PCIe и, как таковой, относительно медленный.
- *Память устройств / Глобальная память* - основная внепиковая память графического процессора, доступная для всех потоков.
- *Общая память,* расположенная в каждом SM, обеспечивает гораздо более быстрый доступ, чем глобальный. Общая память является частной для каждого блока. Потоки внутри одного блока могут использовать его для связи.
- *Регистры* - самая быстрая, приватная, непривлекательная память каждого потока. В общем, они не могут использоваться для связи, но несколько встроенных функций позволяют перетасовывать их содержимое в пределах основы.
- *Локальная память* - Собственная память каждого потока , который адресация. Это используется для разливов регистров и локальных массивов с переменной индексацией. Физически они находятся в глобальной памяти.
- *Память текстур, Постоянная память* - часть глобальной памяти, которая помечается как неизменная для ядра. Это позволяет графическому процессору использовать специальные кэши.
- *L2 cache*- on-chip, доступный для всех потоков. Учитывая количество потоков, ожидаемое время жизни каждой строки кэша намного ниже, чем на процессоре. В основном используется вспомогательная система с неправильным и частично случайным доступом к памяти.
- *L1* - находится в том же пространстве, что и разделяемая память. Опять же, сумма довольно мала, учитывая количество потоков, использующих ее, поэтому не ожидайте, что данные останутся там надолго. L1 кэширование может быть отключено.

Версии

Способность вычислять	Архитектура	Кодовое имя GPU	Дата выхода
1,0	тесла	G80	2006-11-08
1,1	тесла	G84, G86, G92, G94, G96, G98,	2007-04-17

Способность вычислять	Архитектура	Кодовое имя GPU	Дата выхода
1.2	тесла	GT218, GT216, GT215	2009-04-01
1,3	тесла	GT200, GT200b	2009-04-09
2,0	Ферми	GF100, GF110	2010-03-26
2,1	Ферми	GF104, GF106 GF108, GF114, GF116, GF117, GF119	2010-07-12
3.0	Kepler	GK104, GK106, GK107	2012-03-22
3,2	Kepler	GK20A	2014-04-01
3,5	Kepler	GK110, GK208	2013-02-19
3,7	Kepler	GK210	2014-11-17
5.0	максвелл	GM107, GM108	2014-02-18
5,2	максвелл	GM200, GM204, GM206	2014-09-18
5,3	максвелл	GM20B	2015-04-01
6,0	паскаль	GP100	2016-10-01
6,1	паскаль	GP102, GP104, GP106	2016-05-27

Дата выпуска обозначает выпуск первого графического процессора, поддерживающего данную вычислительную способность. Некоторые даты являются приблизительными, например, 3.2 карта была выпущена во втором квартале 2014 года.

Examples

Предпосылки

Чтобы начать программирование с CUDA, загрузите и установите CUDA Toolkit и драйвер разработчика . Инструментарий включает в себя `nvcc` , компилятор NVIDIA CUDA и другое программное обеспечение, необходимое для разработки приложений CUDA. Драйвер гарантирует, что программы GPU будут работать правильно на оборудовании с поддержкой CUDA , которое вам также понадобится.

Вы можете подтвердить, что CUDA Toolkit правильно установлен на вашем компьютере, запустив `nvcc --version` из командной строки. Например, на машине Linux,

```
$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2016 NVIDIA Corporation
Built on Tue_Jul_12_18:28:38_CDT_2016
Cuda compilation tools, release 8.0, V8.0.32
```

выводит информацию о компиляторе. Если предыдущая команда не была успешной, то CUDA Toolkit, скорее всего, не установлен или путь к `nvcc` (`C:\CUDA\bin` на машинах Windows, `/usr/local/cuda/bin` в ОС POSIX) не является частью вашего `PATH` .

Кроме того, вам также понадобится компилятор хоста, который работает с `nvcc` для компиляции и сборки программ CUDA. В Windows это `cl.exe` , компилятор Microsoft, который поставляется с Microsoft Visual Studio. В ОС POSIX доступны другие компиляторы, включая `gcc` или `g++` . В официальном кратком руководстве CUDA вы можете узнать, какие версии компилятора поддерживаются на вашей конкретной платформе.

Чтобы убедиться, что все настроено правильно, давайте скомпилируем и запустим тривиальную программу CUDA, чтобы все инструменты работали правильно.

```
__global__ void foo() {}

int main()
{
  foo<<<1,1>>>();

  cudaDeviceSynchronize();
  printf("CUDA error: %s\n", cudaGetErrorString(cudaGetLastError()));

  return 0;
}
```

Чтобы скомпилировать эту программу, скопируйте ее в файл с именем test.cu и скомпилируйте ее из командной строки. Например, в системе Linux должно работать следующее:

```
$ nvcc test.cu -o test
$ ./test
CUDA error: no error
```

Если программа удалась без ошибок, тогда давайте начнем кодирование!

Суммируйте два массива с CUDA

В этом примере показано, как создать простую программу, которая суммирует два массива `int` с CUDA.

Программа CUDA гетерогенна и состоит из частей, работающих как на процессоре, так и на графическом процессоре.

Основные части программы, использующие CUDA, аналогичны программам ЦП и состоят из

- Распределение памяти для данных, которые будут использоваться на графическом процессоре
- Копирование данных из памяти хоста в память графических процессоров
- Вызов функции ядра для обработки данных
- Результат копирования в память ЦП

Чтобы выделить память устройств, мы используем функцию `cudaMalloc` . Для копирования данных между устройством и хостом может использоваться функция `cudaMemcpy` . Последний аргумент `cudaMemcpy` указывает направление операции копирования. Существует 5 возможных типов:

- `cudaMemcpyHostToHost` - Хост -> Хост
- `cudaMemcpyHostToDevice` - Хост -> Устройство
- `cudaMemcpyDeviceToHost` - Устройство -> Хост
- `cudaMemcpyDeviceToDevice` - Устройство -> Устройство
- `cudaMemcpyDefault` - унифицированное виртуальное адресное пространство по умолчанию

Затем вызывается функция ядра. Информация между тройными шевронами - это конфигурация исполнения, которая определяет, сколько потоков устройств выполняет ядро параллельно. Первое число (`2` в примере) указывает количество блоков и второе (`(size + 1) / 2` в примере) - количество потоков в блоке. Обратите внимание, что в этом примере мы добавляем 1 к размеру, так что мы запрашиваем один дополнительный поток, а не один поток, ответственный за два элемента.

Поскольку вызов ядра является асинхронной функцией, `cudaDeviceSynchronize` вызывается для ожидания завершения выполнения. Массивы результатов копируются в память хоста, и вся память, выделенная на устройстве, освобождается `cudaFree` .

Для определения функции используется `__global__` объявления `__global__` . Эта функция будет вызываться каждым потоком. Если мы хотим, чтобы каждый поток обрабатывал элемент результирующего массива, нам нужно средство для выделения и идентификации каждого потока. CUDA определяет переменные `blockDim` , `blockIdx` и `threadIdx` . `blockDim` переменная `blockDim` содержит размеры каждого потока, как указано во втором параметре конфигурации выполнения для запуска ядра. `threadIdx` переменные `threadIdx` и `blockIdx` содержат индекс потока в его `blockIdx` блоке и блок потока в сетке, соответственно. Обратите внимание: поскольку мы потенциально запрашиваем еще один поток, чем элементы в массивах, нам нужно передать `size` чтобы гарантировать, что мы не получаем доступ к концу массива.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

__global__ void addKernel(int* c, const int* a, const int* b, int size) {
```

```
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < size) {
        c[i] = a[i] + b[i];
    }
}

// Helper function for using CUDA to add vectors in parallel.
void addWithCuda(int* c, const int* a, const int* b, int size) {
    int* dev_a = nullptr;
    int* dev_b = nullptr;
    int* dev_c = nullptr;

    // Allocate GPU buffers for three vectors (two input, one output)
    cudaMalloc((void**)&dev_c, size * sizeof(int));
    cudaMalloc((void**)&dev_a, size * sizeof(int));
    cudaMalloc((void**)&dev_b, size * sizeof(int));

    // Copy input vectors from host memory to GPU buffers.
    cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

    // Launch a kernel on the GPU with one thread for each element.
    // 2 is number of computational blocks and (size + 1) / 2 is a number of threads in a
block
    addKernel<<<2, (size + 1) / 2>>>(dev_c, dev_a, dev_b, size);

    // cudaDeviceSynchronize waits for the kernel to finish, and returns
    // any errors encountered during the launch.
    cudaDeviceSynchronize();

    // Copy output vector from GPU buffer to host memory.
    cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);

    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);
}

int main(int argc, char** argv) {
    const int arraySize = 5;
    const int a[arraySize] = {  1,  2,  3,  4,  5 };
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c[arraySize] = { 0 };

    addWithCuda(c, a, b, arraySize);

    printf("{1, 2, 3, 4, 5} + {10, 20, 30, 40, 50} = {%d, %d, %d, %d, %d}\n", c[0], c[1],
c[2], c[3], c[4]);

    cudaDeviceReset();

    return 0;
}
```

Давайте запустим один поток CUDA, чтобы поздороваться

Эта простая программа CUDA демонстрирует, как написать функцию, которая будет выполняться на графическом процессоре (aka «устройство»). ЦП или «хост» создают потоки CUDA, вызывая специальные функции, называемые «ядрами». Программы CUDA -

это программы на C ++ с дополнительным синтаксисом.

Чтобы увидеть, как это работает, поместите следующий код в файл с именем `hello.cu` :

```
#include <stdio.h>

// __global__ functions, or "kernels", execute on the device
__global__ void hello_kernel(void)
{
  printf("Hello, world from the device!\n");
}

int main(void)
{
  // greet from the host
  printf("Hello, world from the host!\n");

  // launch a kernel with a single thread to greet from the device
  hello_kernel<<<1,1>>>();

  // wait for the device to finish so that we see the message
  cudaDeviceSynchronize();

  return 0;
}
```

(Обратите внимание, что для использования `printf` на устройстве вам требуется устройство с вычислительной способностью не менее 2.0. Подробнее см. В обзоре версий .)

Теперь давайте скомпилируем программу с помощью компилятора NVIDIA и запустим ее:

```
$ nvcc hello.cu -o hello
$ ./hello
Hello, world from the host!
Hello, world from the device!
```

Некоторая дополнительная информация о вышеупомянутом примере:

- `nvcc` означает «NVIDIA CUDA Compiler». Он отделяет исходный код от компонентов хоста и устройства.
- `__global__` - это ключевое слово CUDA, используемое в объявлениях функций, указывающее, что функция выполняется на устройстве GPU и вызывается из хоста.
- Тройные угловые скобки (`<<<` , `>>>`) отмечают вызов из кода хоста на код устройства (также называемый «запуск ядра»). Числа в этих тройных скобках указывают количество раз для выполнения параллельно и количество потоков.

Компиляция и запуск пробных программ

Руководство по установке NVIDIA заканчивается запуском выборочных программ, чтобы проверить установку CUDA Toolkit, но не указывается явно. Сначала проверьте все предварительные условия. Проверьте каталог CUDA по умолчанию для выборочных

программ. Если его нет, его можно загрузить с официального сайта CUDA. Перейдите в каталог, в котором присутствуют примеры.

```
$ cd /path/to/samples/
$ ls
```

Вы должны увидеть результат, похожий на:

```
0_Simple     2_Graphics  4_Finance      6_Advanced       bin     EULA.txt
1_Utilities  3_Imaging   5_Simulations  7_CUDALibraries  common  Makefile
```

Убедитесь, что `Makefile` присутствует в этом каталоге. Команда `make` в UNIX-системах будет создавать все примеры программ. Кроме того, перейдите в подкаталог, в котором присутствует другой `Makefile` и запустите команду `make` оттуда, чтобы создать только этот образец.

Запустите две предложенные примеры программ - `deviceQuery` и `bandwidthTest` :

```
$ cd 1_Utilities/deviceQuery/
$ ./deviceQuery
```

Выход будет аналогичен показанному ниже:

```
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 950M"
  CUDA Driver Version / Runtime Version          7.5 / 7.5
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 4096 MBytes (4294836224 bytes)
  ( 5) Multiprocessors, (128) CUDA Cores/MP:     640 CUDA Cores
  GPU Max Clock rate:                            1124 MHz (1.12 GHz)
  Memory Clock rate:                             900 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096,
4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
```

```
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 7.5, CUDA Runtime Version = 7.5,
NumDevs = 1, Device0 = GeForce GTX 950M
Result = PASS
```

Заявление `Result = PASS` в конце указывает, что все работает правильно. Теперь запустите другую предложенную тестовую программу `bandwidthTest` аналогичным образом. Выход будет похож на:

```
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: GeForce GTX 950M
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)    Bandwidth(MB/s)
   33554432            10604.5

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)    Bandwidth(MB/s)
   33554432            10202.0

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)    Bandwidth(MB/s)
   33554432            23389.7

Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU
Boost is enabled.
```

Опять же, оператор `Result = PASS` указывает, что все выполнено правильно. Все остальные примеры программ можно запускать аналогичным образом.

Прочитайте Начало работы с cuda онлайн: https://riptutorial.com/ru/cuda/topic/1860/начало-работы-с-cuda

глава 2: Межблочная связь

замечания

Блоки в CUDA работают полуавтоматически. Безопасного способа их синхронизации не существует. Однако это не означает, что они никак не могут взаимодействовать друг с другом.

Examples

Защитник последнего блока

Рассмотрим сетку, работающую над некоторой задачей, например параллельное сокращение. Первоначально каждый блок может выполнять свою работу независимо, производя некоторый частичный результат. В конце концов, однако, частичные результаты необходимо объединить и объединить. Типичным примером является алгоритм сокращения больших данных.

Типичный подход заключается в том, чтобы вызывать два ядра, один для частичного вычисления, а другой для слияния. Однако, если слияние может быть выполнено эффективно одним блоком, требуется только один вызов ядра. Это достигается с помощью `lastBlock` устройства `lastBlock` определенного как:

2,0

```
__device__ bool lastBlock(int* counter) {
  __threadfence(); //ensure that partial result is visible by all blocks
  int last = 0;
  if (threadIdx.x == 0)
    last = atomicAdd(counter, 1);
  return __syncthreads_or(last == gridDim.x-1);
}
```

1,1

```
__device__ bool lastBlock(int* counter) {
  __shared__ int last;
  __threadfence(); //ensure that partial result is visible by all blocks
  if (threadIdx.x == 0) {
    last = atomicAdd(counter, 1);
  }
  __syncthreads();
  return last == gridDim.x-1;
}
```

С таким защитником последний блок гарантированно будет видеть все результаты, полученные всеми другими блоками, и может выполнять слияние.

```
__device__ void computePartial(T* out) { ... }
__device__ void merge(T* partialResults, T* out) { ... }

__global__ void kernel(int* counter, T* partialResults, T* finalResult) {
    computePartial(&partialResults[blockIdx.x]);
    if (lastBlock(counter)) {
      //this is executed by all threads of the last block only
      merge(partialResults,finalResult);
    }
}
```

Предположения:

- Счетчик должен быть глобальным указателем памяти, инициализированным до 0 *до* вызова ядра.
- Функция `lastBlock` вызывается равномерно всеми потоками во всех блоках
- Ядро вызывается в одномерной сетке (для простоты примера)
- `T` называет любой тип, который вам нравится, но этот пример не предназначен для шаблона в смысле C ++

Глобальная рабочая очередь

Рассмотрим множество рабочих элементов. Время, необходимое для завершения каждого рабочего элемента, сильно различается. Чтобы сбалансировать распределение работы между блоками, для каждого блока может быть разумным выборку следующего элемента только тогда, когда предыдущий закончен. Это контрастирует с априорным назначением элементов блокам.

```
class WorkQueue {
private:
  WorkItem* gItems;
  size_t totalSize;
  size_t current;
public:
  __device__ WorkItem& fetch() {
    __shared__ WorkItem item;
    if (threadIdx.x == 0) {
      size_t itemIdx = atomicAdd(current,1);
      if (itemIdx<totalSize)
        item = gItems[itemIdx];
      else
        item = WorkItem::none();
    }
    __syncthreads();
    return item; //returning reference to smem - ok
  }
}
```

Предположения:

- Объект WorkQueue, а также массив gItem находятся в глобальной памяти
- Никакие новые рабочие элементы не добавляются в объект WorkQueue в ядре,

которое извлекает из него
- `WorkItem` - небольшое представление задания, например указатель на другой объект
- Функция `WorkItem::none()` static member создает объект `WorkItem` который не представляет никакой работы вообще
- `WorkQueue::fetch()` должен быть вызван равномерно всеми потоками в блоке
- Нет никаких 2 вызовов `WorkQueue::fetch()` без другого `__syncthreads()` между ними. В противном случае появится состояние гонки!

В примере не указано, как инициализировать `WorkQueue` или заполнить его. Это делается другим кодом ядра или ЦП и должно быть довольно простым.

Прочитайте Межблочная связь онлайн: https://riptutorial.com/ru/cuda/topic/4978/межблочная-связь

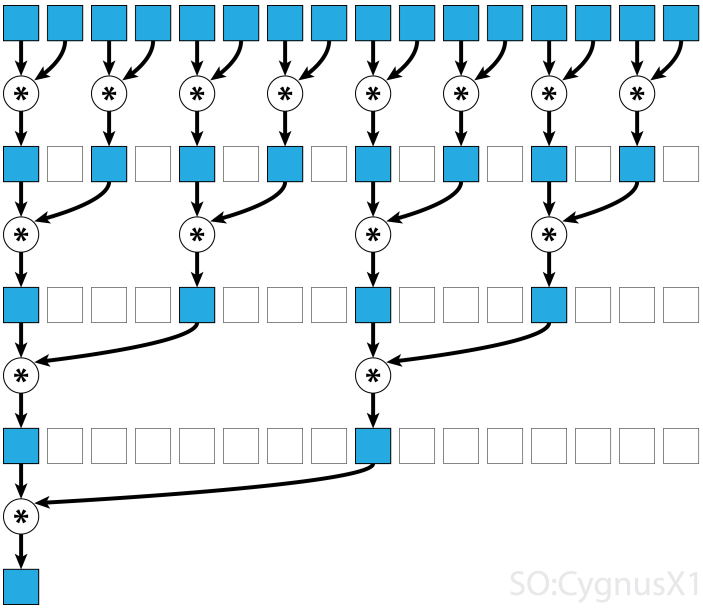
глава 3: Параллельное сокращение (например, как суммировать массив)

замечания

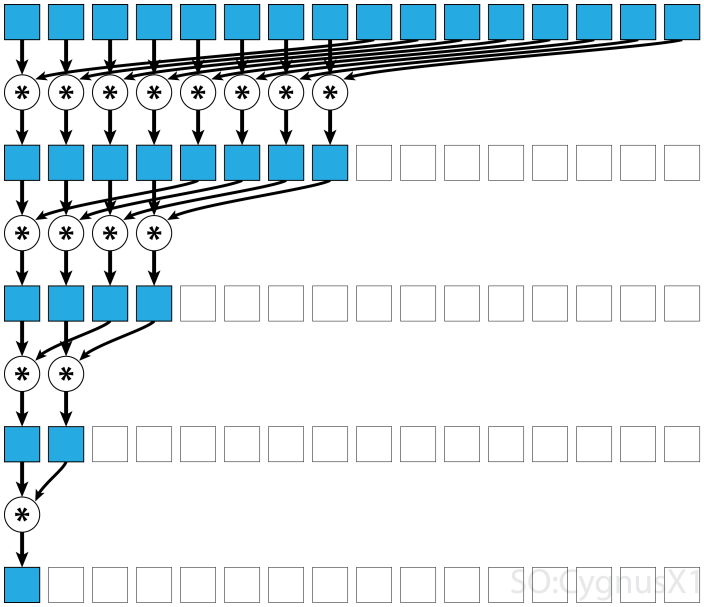
Алгоритм параллельной регрессии обычно относится к алгоритму, который объединяет массив элементов, создавая единственный результат. Типичными проблемами, которые относятся к этой категории, являются:

- суммирование всех элементов в массиве
- нахождение максимума в массиве

В общем случае параллельное редукция может быть применено для любого двоичного ассоциативного оператора , т. Е. `(A*B)*C = A*(B*C)` . С таким оператором * алгоритм параллельной редукции повторно группирует аргументы массива в парах. Каждая пара вычисляется параллельно с другими, уменьшая общий размер массива за один шаг. Процесс повторяется до тех пор, пока не будет существовать только один элемент.



Если оператор коммутативен (т. Е. `A*B = B*A`) в дополнение к ассоциативному, алгоритм может парировать в другой схеме. С теоретической точки зрения это не имеет никакого значения, но на практике это дает лучшую схему доступа к памяти:



Не все ассоциативные операторы коммутативны - например, возьмем матричное умножение.

Examples

Одноблочная параллельная редукция для коммутативного оператора

Простейшим подходом к параллельному сокращению CUDA является назначение отдельного блока для выполнения задачи:

```
static const int arraySize = 10000;
static const int blockSize = 1024;

__global__ void sumCommSingleBlock(const int *a, int *out) {
    int idx = threadIdx.x;
    int sum = 0;
    for (int i = idx; i < arraySize; i += blockSize)
        sum += a[i];
    __shared__ int r[blockSize];
    r[idx] = sum;
    __syncthreads();
    for (int size = blockSize/2; size>0; size/=2) { //uniform
        if (idx<size)
            r[idx] += r[idx+size];
        __syncthreads();
    }
    if (idx == 0)
        *out = r[0];
}

...

sumCommSingleBlock<<<1, blockSize>>>(dev_a, dev_out);
```

Это наиболее возможно, когда размер данных не очень большой (около нескольких тысяч элементов). Обычно это происходит, когда сокращение является частью некоторой более крупной программы CUDA. Если вход соответствует `blockSize` с самого начала, первый цикл

`for` может быть полностью удален.

Обратите внимание, что на первом этапе, когда есть больше элементов, чем потоки, мы добавляем вещи полностью независимо. Только когда проблема сводится к `blockSize` , фактические триггеры параллельной редукции. Тот же код может быть применен к любому другому коммутативному ассоциативному оператору, такому как умножение, минимум, максимум и т. Д.

Обратите внимание, что алгоритм можно сделать быстрее, например, используя параллельное сокращение уровня деформации.

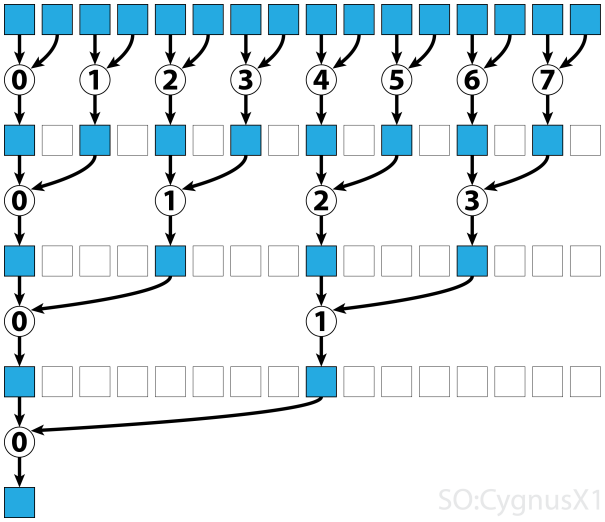
Одноблочная параллельная редукция для некоммутативного оператора

Выполнение параллельной редукции для некоммутативного оператора немного более активно, по сравнению с коммутативной версией. В этом примере мы по-прежнему используем дополнение для целых чисел для простоты. Его можно было бы заменить, например, матричным умножением, которое действительно является некоммутативным. Заметим, что при этом 0 следует заменить нейтральным элементом умножения - то есть единичной матрицей.

```
static const int arraySize = 1000000;
static const int blockSize = 1024;

__global__ void sumNoncommSingleBlock(const int *gArr, int *out) {
    int thIdx = threadIdx.x;
    __shared__ int shArr[blockSize*2];
    __shared__ int offset;
    shArr[thIdx] = thIdx<arraySize ? gArr[thIdx] : 0;
    if (thIdx == 0)
        offset = blockSize;
    __syncthreads();
    while (offset < arraySize) { //uniform
        shArr[thIdx + blockSize] = thIdx+offset<arraySize ? gArr[thIdx+offset] : 0;
        __syncthreads();
        if (thIdx == 0)
            offset += blockSize;
        int sum = shArr[2*thIdx] + shArr[2*thIdx+1];
        __syncthreads();
        shArr[thIdx] = sum;
    }
    __syncthreads();
    for (int stride = 1; stride<blockSize; stride*=2) { //uniform
        int arrIdx = thIdx*stride*2;
        if (arrIdx+stride<blockSize)
            shArr[arrIdx] += shArr[arrIdx+stride];
        __syncthreads();
    }
    if (thIdx == 0)
        *out = shArr[0];
}

...

sumNoncommSingleBlock<<<1, blockSize>>>(dev_a, dev_out);
```

В первом цикле while выполняется столько, сколько есть элементов ввода, кроме потоков. На каждой итерации выполняется одно сокращение, и результат сжимается в первую половину массива `shArr` . Вторая половина заполняется новыми данными.

Когда все данные загружаются из `gArr` , выполняется второй цикл. Теперь мы больше не `__syncthreads()` результат (который `__syncthreads()` дополнительных `__syncthreads()`). На каждом шаге поток n получает `2*n` активный элемент и добавляет его с помощью `2*n+1` элемента:



Существует множество способов дальнейшей оптимизации этого простого примера, например, путем сокращения уровня деформации и устранения конфликтов в банках с разделяемой памятью.

Многоблочное параллельное сокращение для коммутативного оператора

Многоблочный подход к параллельному сокращению CUDA создает дополнительную проблему по сравнению с одноблочным подходом, поскольку блоки ограничены в общении. Идея состоит в том, чтобы позволить каждому блоку вычислить часть входного массива, а затем иметь один заключительный блок для объединения всех частичных результатов. Для этого можно запустить два ядра, неявно создав точку синхронизации по всей сетке.

```
static const int wholeArraySize = 100000000;
static const int blockSize = 1024;
static const int gridSize = 24; //this number is hardware-dependent; usually #SM*2 is a good
number.

__global__ void sumCommMultiBlock(const int *gArr, int arraySize, int *gOut) {
    int thIdx = threadIdx.x;
    int gthIdx = thIdx + blockIdx.x*blockSize;
    const int gridSize = blockSize*gridDim.x;
    int sum = 0;
    for (int i = gthIdx; i < arraySize; i += gridSize)
        sum += gArr[i];
    __shared__ int shArr[blockSize];
    shArr[thIdx] = sum;
    __syncthreads();
```

```
    for (int size = blockSize/2; size>0; size/=2) { //uniform
        if (thIdx<size)
            shArr[thIdx] += shArr[thIdx+size];
        __syncthreads();
    }
    if (thIdx == 0)
        gOut[blockIdx.x] = shArr[0];
}

__host__ int sumArray(int* arr) {
    int* dev_arr;
    cudaMalloc((void**)&dev_arr, wholeArraySize * sizeof(int));
    cudaMemcpy(dev_arr, arr, wholeArraySize * sizeof(int), cudaMemcpyHostToDevice);

    int out;
    int* dev_out;
    cudaMalloc((void**)&dev_out, sizeof(int)*gridSize);

    sumCommMultiBlock<<<gridSize, blockSize>>>(dev_arr, wholeArraySize, dev_out);
    //dev_out now holds the partial result
    sumCommMultiBlock<<<1, blockSize>>>(dev_out, gridSize, dev_out);
    //dev_out[0] now holds the final result
    cudaDeviceSynchronize();

    cudaMemcpy(&out, dev_out, sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(dev_arr);
    cudaFree(dev_out);
    return out;
}
```

В идеале желательно запустить достаточно блоков, чтобы насытить все многопроцессоры на GPU при полной занятости. Превышение этого числа - в частности, запуск как можно большего количества потоков, чем есть элементы в массиве, - является контрпродуктивным. Это не увеличивает необработанную вычислительную мощность, но предотвращает использование очень эффективного первого цикла.

Также можно получить тот же результат, используя одно ядро, с помощью защиты последнего блока :

```
static const int wholeArraySize = 100000000;
static const int blockSize = 1024;
static const int gridSize = 24;

__device__ bool lastBlock(int* counter) {
    __threadfence(); //ensure that partial result is visible by all blocks
    int last = 0;
    if (threadIdx.x == 0)
        last = atomicAdd(counter, 1);
    return __syncthreads_or(last == gridDim.x-1);
}

__global__ void sumCommMultiBlock(const int *gArr, int arraySize, int *gOut, int*
lastBlockCounter) {
    int thIdx = threadIdx.x;
    int gthIdx = thIdx + blockIdx.x*blockSize;
    const int gridSize = blockSize*gridDim.x;
    int sum = 0;
```

```
    for (int i = gthIdx; i < arraySize; i += gridSize)
        sum += gArr[i];
    __shared__ int shArr[blockSize];
    shArr[thIdx] = sum;
    __syncthreads();
    for (int size = blockSize/2; size>0; size/=2) { //uniform
        if (thIdx<size)
            shArr[thIdx] += shArr[thIdx+size];
        __syncthreads();
    }
    if (thIdx == 0)
        gOut[blockIdx.x] = shArr[0];
    if (lastBlock(lastBlockCounter)) {
        shArr[thIdx] = thIdx<gridSize ? gOut[thIdx] : 0;
        __syncthreads();
        for (int size = blockSize/2; size>0; size/=2) { //uniform
            if (thIdx<size)
                shArr[thIdx] += shArr[thIdx+size];
            __syncthreads();
        }
        if (thIdx == 0)
            gOut[0] = shArr[0];
    }
}

__host__ int sumArray(int* arr) {
    int* dev_arr;
    cudaMalloc((void**)&dev_arr, wholeArraySize * sizeof(int));
    cudaMemcpy(dev_arr, arr, wholeArraySize * sizeof(int), cudaMemcpyHostToDevice);

    int out;
    int* dev_out;
    cudaMalloc((void**)&dev_out, sizeof(int)*gridSize);

    int* dev_lastBlockCounter;
    cudaMalloc((void**)&dev_lastBlockCounter, sizeof(int));
    cudaMemset(dev_lastBlockCounter, 0, sizeof(int));

    sumCommMultiBlock<<<gridSize, blockSize>>>(dev_arr, wholeArraySize, dev_out,
dev_lastBlockCounter);
    cudaDeviceSynchronize();

    cudaMemcpy(&out, dev_out, sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(dev_arr);
    cudaFree(dev_out);
    return out;
}
```

Обратите внимание, что ядро можно сделать быстрее, например, используя параллельное сокращение уровня деформации.

Многоблочное параллельное сокращение для некоммутативного оператора

Многоблочный подход к параллельному сокращению очень похож на одноблочный подход. Глобальный массив ввода должен быть разделен на разделы, каждый из которых уменьшается на один блок. Когда получается частичный результат из каждого блока, один

конечный блок уменьшает их, чтобы получить конечный результат.

- `sumNoncommSingleBlock` объясняется более подробно в примере восстановления с одним блоком.
- `lastBlock` принимает только последний его блок. Если вы хотите этого избежать, вы можете разделить ядро на два отдельных вызова.

```
static const int wholeArraySize = 100000000;
static const int blockSize = 1024;
static const int gridSize = 24; //this number is hardware-dependent; usually #SM*2 is a good
number.

__device__ bool lastBlock(int* counter) {
    __threadfence(); //ensure that partial result is visible by all blocks
    int last = 0;
    if (threadIdx.x == 0)
        last = atomicAdd(counter, 1);
    return __syncthreads_or(last == gridDim.x-1);
}

__device__ void sumNoncommSingleBlock(const int* gArr, int arraySize, int* out) {
    int thIdx = threadIdx.x;
    __shared__ int shArr[blockSize*2];
    __shared__ int offset;
    shArr[thIdx] = thIdx<arraySize ? gArr[thIdx] : 0;
    if (thIdx == 0)
        offset = blockSize;
    __syncthreads();
    while (offset < arraySize) { //uniform
        shArr[thIdx + blockSize] = thIdx+offset<arraySize ? gArr[thIdx+offset] : 0;
        __syncthreads();
        if (thIdx == 0)
            offset += blockSize;
        int sum = shArr[2*thIdx] + shArr[2*thIdx+1];
        __syncthreads();
        shArr[thIdx] = sum;
    }
    __syncthreads();
    for (int stride = 1; stride<blockSize; stride*=2) { //uniform
        int arrIdx = thIdx*stride*2;
        if (arrIdx+stride<blockSize)
            shArr[arrIdx] += shArr[arrIdx+stride];
        __syncthreads();
    }
    if (thIdx == 0)
        *out = shArr[0];
}

__global__ void sumNoncommMultiBlock(const int* gArr, int* out, int* lastBlockCounter) {
    int arraySizePerBlock = wholeArraySize/gridSize;
    const int* gArrForBlock = gArr+blockIdx.x*arraySizePerBlock;
    int arraySize = arraySizePerBlock;
    if (blockIdx.x == gridSize-1)
        arraySize = wholeArraySize - blockIdx.x*arraySizePerBlock;
    sumNoncommSingleBlock(gArrForBlock, arraySize, &out[blockIdx.x]);
    if (lastBlock(lastBlockCounter))
        sumNoncommSingleBlock(out, gridSize, out);
}
```

В идеале желательно запустить достаточно блоков, чтобы насытить все многопроцессоры на GPU при полной занятости. Превышение этого числа - в частности, запуск как можно большего количества потоков, чем есть элементы в массиве, - является контрпродуктивным. Это не увеличивает необработанную вычислительную мощность, но предотвращает использование очень эффективного первого цикла.

Однопараметрическое параллельное сокращение для коммутативного оператора

Иногда сокращение должно выполняться в очень малом масштабе, как часть большего ядра CUDA. Предположим, например, что входные данные имеют ровно 32 элемента - количество потоков в основе. В таком сценарии для сокращения можно назначить одну деформацию. Учитывая, что warp выполняется в идеальной синхронизации, многие команды `__syncthreads()` могут быть удалены - по сравнению с уменьшением уровня блока.

```
static const int warpSize = 32;

__device__ int sumCommSingleWarp(volatile int* shArr) {
    int idx = threadIdx.x % warpSize; //the lane index in the warp
    if (idx<16) shArr[idx] += shArr[idx+16];
    if (idx<8) shArr[idx] += shArr[idx+8];
    if (idx<4) shArr[idx] += shArr[idx+4];
    if (idx<2) shArr[idx] += shArr[idx+2];
    if (idx==0) shArr[idx] += shArr[idx+1];
    return shArr[0];
}
```

`shArr` предпочтительно представляет собой массив в общей памяти. Значение должно быть одинаковым для всех потоков в warp. Если `sumCommSingleWarp` вызывается несколькими перекосами, `shArr` должен быть разным между искажениями (одинаковыми в каждом warp).

Аргумент `shArr` помечен как `volatile` чтобы гарантировать, что операции над массивом фактически выполняются там, где это указано. В противном случае повторное назначение `shArr[idx]` может быть оптимизировано как присвоение регистру, при этом только конечный атрибут является фактическим хранилищем для `shArr` . Когда это происходит, немедленные назначения не видны другим потокам, что приводит к неправильным результатам. Обратите внимание, что вы можете передать нормальный энергонезависимый массив в качестве аргумента volatile, так же, как когда вы передаете не const в качестве параметра const.

Если после сокращения ничего не волнует содержимое `shArr[1..31]` , можно еще более упростить код:

```
static const int warpSize = 32;

__device__ int sumCommSingleWarp(volatile int* shArr) {
    int idx = threadIdx.x % warpSize; //the lane index in the warp
    if (idx<16) {
```

```
      shArr[idx] += shArr[idx+16];
      shArr[idx] += shArr[idx+8];
      shArr[idx] += shArr[idx+4];
      shArr[idx] += shArr[idx+2];
      shArr[idx] += shArr[idx+1];
    }
    return shArr[0];
}
```

В этой установке мы удалили многие условия `if` . Дополнительные потоки выполняют некоторые ненужные дополнения, но мы больше не заботимся о содержимом, которое они создают. Поскольку перекосы выполняются в режиме SIMD, мы фактически не экономим время, когда эти потоки ничего не делают. С другой стороны, оценка условий занимает относительно большое количество времени, так как тело этих операторов `if` настолько мало. Исходный оператор `if` может быть удален, если `shArr[32..47]` дополняется 0.

Снижение уровня деформации может быть использовано для ускорения снижения уровня блока:

```
__global__ void sumCommSingleBlockWithWarps(const int *a, int *out) {
    int idx = threadIdx.x;
    int sum = 0;
    for (int i = idx; i < arraySize; i += blockSize)
        sum += a[i];
    __shared__ int r[blockSize];
    r[idx] = sum;
    sumCommSingleWarp(&r[idx & ~(warpSize-1)]);
    __syncthreads();
    if (idx<warpSize) { //first warp only
        r[idx] = idx*warpSize<blockSize ? r[idx*warpSize] : 0;
        sumCommSingleWarp(r);
        if (idx == 0)
            *out = r[0];
    }
}
```

Аргумент `&r[idx & ~(warpSize-1)]` в основном равен `r + warpIdx*32` . Это эффективно разделяет массив `r` на куски из 32 элементов, и каждый кусок присваивается отдельной структуре.

Однопаральное параллельное сокращение для некоммутативного оператора

Иногда сокращение должно выполняться в очень малом масштабе, как часть большего ядра CUDA. Предположим, например, что входные данные имеют ровно 32 элемента - количество потоков в основе. В таком сценарии для сокращения можно назначить одну деформацию. Учитывая, что warp выполняется в идеальной синхронизации, многие команды `__syncthreads()` могут быть удалены - по сравнению с уменьшением уровня блока.

```
static const int warpSize = 32;
```

```
__device__ int sumNoncommSingleWarp(volatile int* shArr) {
    int idx = threadIdx.x % warpSize; //the lane index in the warp
    if (idx%2 == 0) shArr[idx] += shArr[idx+1];
    if (idx%4 == 0) shArr[idx] += shArr[idx+2];
    if (idx%8 == 0) shArr[idx] += shArr[idx+4];
    if (idx%16 == 0) shArr[idx] += shArr[idx+8];
    if (idx == 0) shArr[idx] += shArr[idx+16];
    return shArr[0];
}
```

`shArr` предпочтительно представляет собой массив в общей памяти. Значение должно быть одинаковым для всех потоков в warp. Если `sumCommSingleWarp` вызывается несколькими перекосами, `shArr` должен быть разным между искажениями (одинаковыми в каждом warp).

Аргумент `shArr` помечен как `volatile` чтобы гарантировать, что операции над массивом фактически выполняются там, где это указано. В противном случае повторное назначение `shArr[idx]` может быть оптимизировано как присвоение регистру, при этом только конечный атрибут является фактическим хранилищем для `shArr` . Когда это происходит, немедленные назначения не видны другим потокам, что приводит к неправильным результатам. Обратите внимание, что вы можете передать нормальный энергонезависимый массив в качестве аргумента volatile, так же, как когда вы передаете не const в качестве параметра const.

Если не нужно окончательное содержание `shArr[1..31]` и может `shArr[32..47]` с нулями, можно упростить приведенный выше код:

```
static const int warpSize = 32;

__device__ int sumNoncommSingleWarpPadded(volatile int* shArr) {
    //shArr[32..47] == 0
    int idx = threadIdx.x % warpSize; //the lane index in the warp
    shArr[idx] += shArr[idx+1];
    shArr[idx] += shArr[idx+2];
    shArr[idx] += shArr[idx+4];
    shArr[idx] += shArr[idx+8];
    shArr[idx] += shArr[idx+16];
    return shArr[0];
}
```

В этой установке мы удалили все `if` таковые условия, которые составляют около половины инструкции. Дополнительные потоки выполняют некоторые ненужные дополнения, сохраняя результат в ячейках `shArr` которые в конечном итоге не влияют на конечный результат. Поскольку перекосы выполняются в режиме SIMD, мы фактически не экономим время, когда эти потоки ничего не делают.

Параллельное сокращение одиночной варпы с использованием только регистров

Как правило, сокращение выполняется для глобального или общего массива. Однако, когда

редукция выполняется в очень небольшом масштабе, как часть большего ядра CUDA, ее можно выполнить с помощью единственного деформирования. Когда это происходит, на Keppler или более высоких архитектурах (CC> = 3.0), можно использовать функции warp-shuffle, чтобы вообще не использовать общую память.

Предположим, например, что каждый поток в warp содержит одно значение входных данных. Все потоки вместе имеют 32 элемента, которые нам нужно подвести (или выполнить другую ассоциативную операцию)

```
__device__ int sumSingleWarpReg(int value) {
    value += __shfl_down(value, 1);
    value += __shfl_down(value, 2);
    value += __shfl_down(value, 4);
    value += __shfl_down(value, 8);
    value += __shfl_down(value, 16);
    return __shfl(value,0);
}
```

Эта версия работает как для коммутативных, так и для некоммутативных операторов.

Прочитайте Параллельное сокращение (например, как суммировать массив) онлайн: https://riptutorial.com/ru/cuda/topic/6566/параллельное-сокращение--например--как-суммировать-массив-

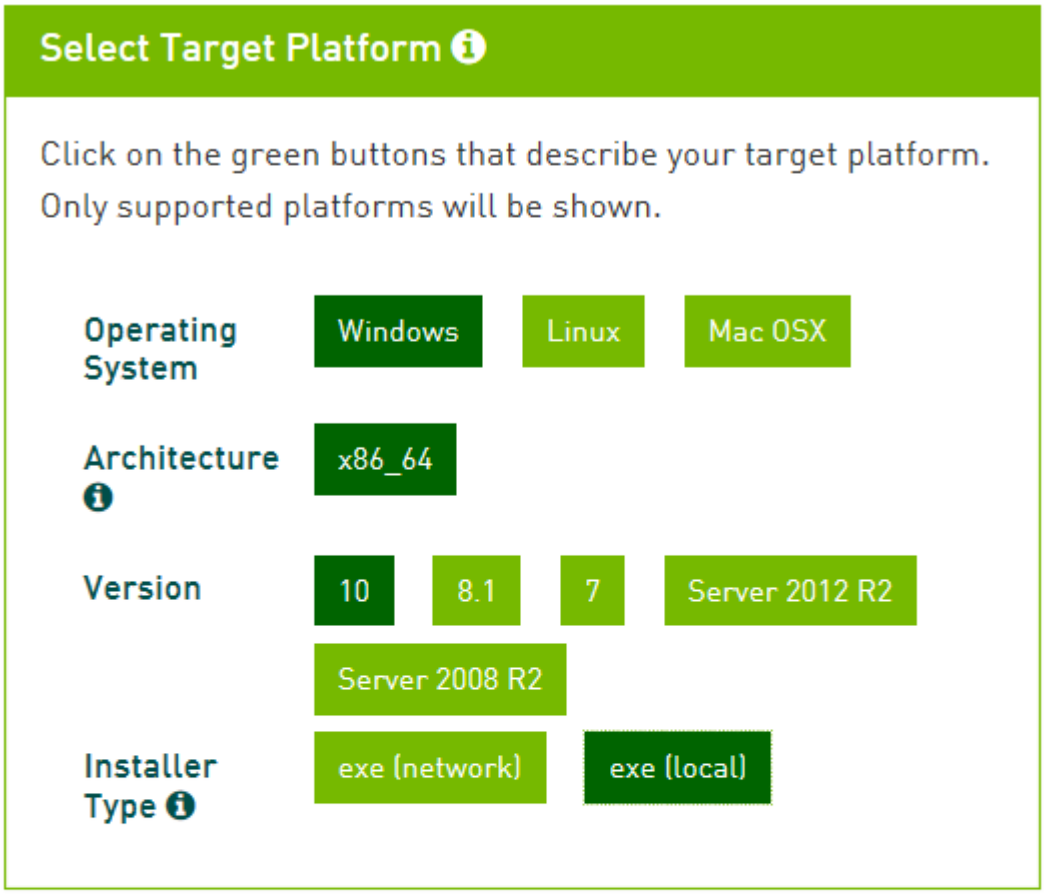
глава 4: Установка cuda

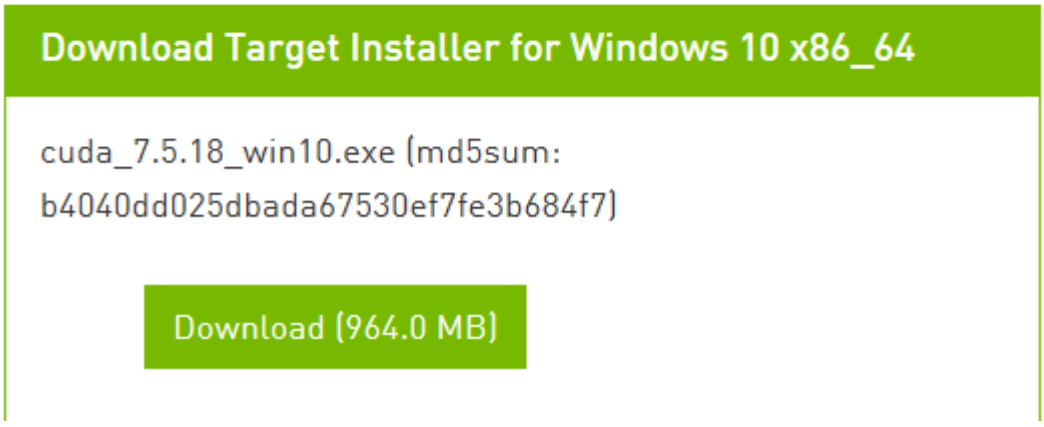
замечания

Чтобы установить CUDA toolkit в Windows, кулак вам нужно установить правильную версию Visual Studio. Visual Studio 2013 должен быть установлен, если вы собираетесь установить CUDA 7.0 или 7.5. Visual Studio 2015 поддерживается для CUDA 8.0 и выше.

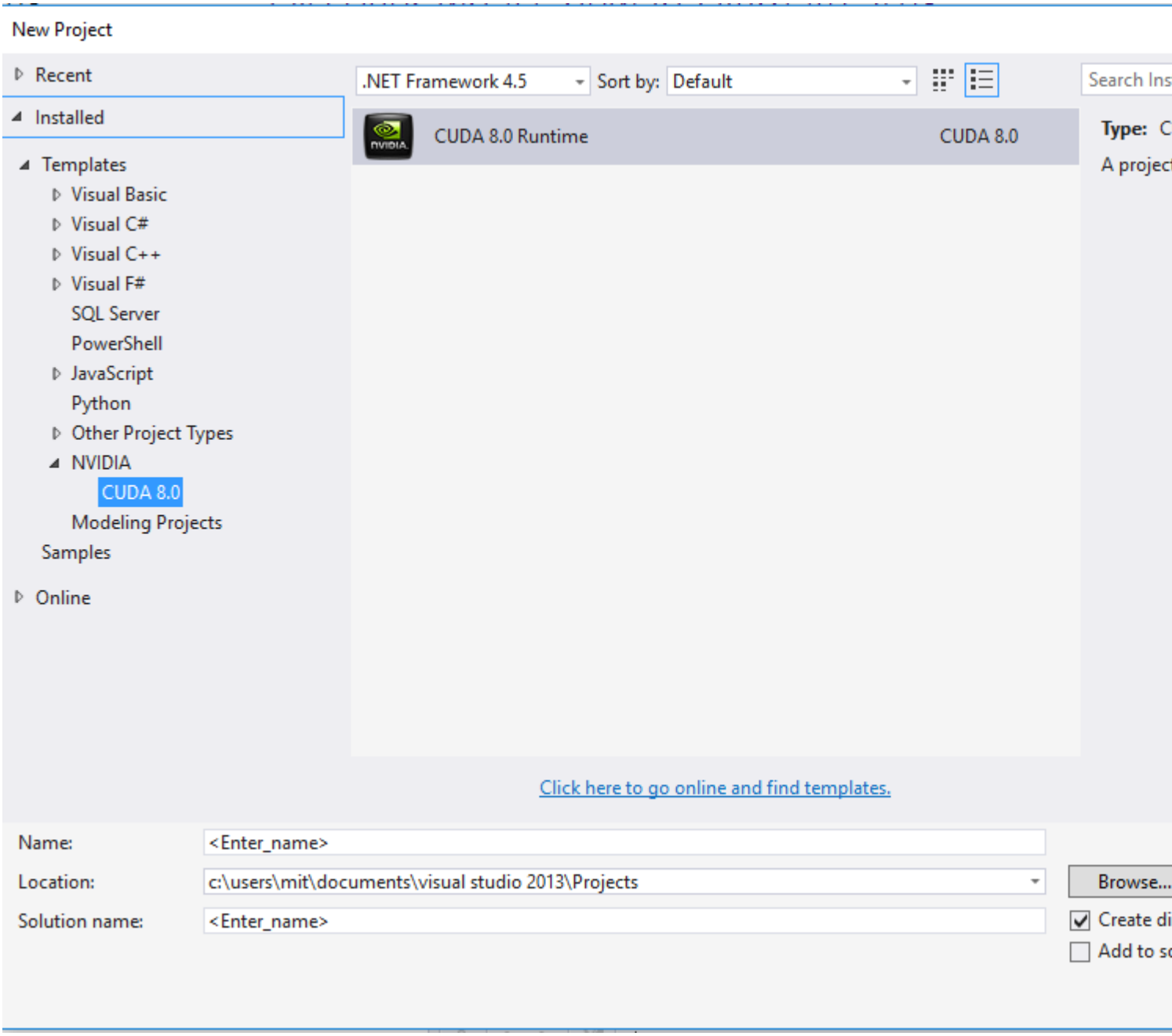
Когда у вас есть правильная версия VS в вашей системе, пришло время загрузить и установить CUDA toolkit. Перейдите по этой ссылке, чтобы найти версию инструментария CUDA, которую вы ищете: CUDA toolkit archive

На странице загрузки вы должны выбрать версию окна на целевой машине и тип установщика (выберите локальный).





После загрузки exe-файла вы должны извлечь его и запустить `setup.exe` . По завершении установки откройте новый проект и выберите NVIDIA> CUDAX.X из шаблонов.



Помните, что расширение исходных файлов CUDA - `.cu` . Вы можете написать код хоста и устройства в одном источнике.

Examples

Очень простой код CUDA

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "cuda.h"
#include <device_functions.h>
#include <cuda_runtime_api.h>

#include<stdio.h>
#include <cmath>
#include<stdlib.h>
```

```
#include<iostream>
#include <iomanip>



using namespace std;
typedef unsigned int uint;


const uint N = 1e6;

__device__ uint Val2[N];

__global__ void set0()
{
    uint index = __mul24(blockIdx.x, blockDim.x) + threadIdx.x;
    if (index < N)
    {
        Val2[index] = 0;
    }

}


int main()
{
    int numThreads = 512;
    uint numBlocks = (uint)ceil(N / (double)numThreads);
    set0 << < numBlocks, numThreads >> >();

    return 0;
}
```

Прочитайте Установка cuda онлайн: https://riptutorial.com/ru/cuda/topic/10949/установка-cuda

кредиты

S. No	Главы	Contributors
1	Начало работы с cuda	Community, CygnusX1, Dev-iL, harrism, havogt, infinite.potential, Jared Hoberock, Ken Y-N, Marco13, NikolayKondratyev, Tejus Prasad
2	Межблочная связь	CygnusX1, tera
3	Параллельное сокращение (например, как суммировать массив)	CygnusX1
4	Установка cuda	Mo Sani