Kostenloses eBook

LERNEN

cuda

Free unaffiliated eBook created from
Stack Overflow contributors.

#cuda

Inhaltsverzeichnis

Über

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

Kapitel 1: Erste Schritte mit cuda

Bemerkungen

CUDA ist eine proprietäre NVIDIA Parallel Computing-Technologie und Programmiersprache für ihre GPUs.

GPUs sind hochparallele Maschinen, mit denen Tausende leichter Threads parallel ausgeführt werden können. Jeder GPU-Thread wird normalerweise langsamer ausgeführt und sein Kontext ist kleiner. Auf der anderen Seite kann GPU mehrere Tausend Threads parallel und sogar parallel ausführen (genaue Zahlen hängen vom tatsächlichen GPU-Modell ab). CUDA ist ein C ++ - Dialekt, der speziell für die NVIDIA-GPU-Architektur entwickelt wurde. Aufgrund der Architekturunterschiede können die meisten Algorithmen jedoch nicht einfach aus C ++ kopiert werden - sie würden ausgeführt, wären aber sehr langsam.

Terminologie

- *Host* - bezieht sich auf normale CPU-basierte Hardware und normale Programme, die in dieser Umgebung ausgeführt werden
- *Gerät* - bezieht sich auf eine bestimmte GPU, in der CUDA-Programme ausgeführt werden. Ein einzelner Host kann mehrere Geräte unterstützen.
- *Kernel* - Eine Funktion, die sich auf dem Gerät befindet und vom Hostcode aus aufgerufen werden kann.

Physikalische Prozessorstruktur

Der CUDA-fähige GPU-Prozessor hat die folgende physische Struktur:

- *der Chip* - der gesamte Prozessor der GPU. Einige GPUs haben zwei davon.
- *Streamming-Multiprozessor* (SM) - Jeder Chip enthält je nach Modell bis zu ~ 100 SMs. Jedes SM arbeitet nahezu unabhängig voneinander und verwendet nur globalen Speicher, um miteinander zu kommunizieren.
- *CUDA-Kern* - eine einzelne Skalar-Recheneinheit eines SM. Ihre genaue Anzahl hängt von der Architektur ab. Jeder Kern kann einige Threads verarbeiten, die gleichzeitig ausgeführt werden (ähnlich wie Hyperthreading in CPU).

Darüber hinaus verfügt jedes SM über einen oder mehrere *Warp-Scheduler* . Jeder Scheduler sendet eine einzelne Anweisung an mehrere CUDA-Kerne. Dies bewirkt effektiv, dass der SM im 32-breiten SIMD- Modus arbeitet.

CUDA-Ausführungsmodell

Die physische Struktur der GPU hat direkten Einfluss darauf, wie Kernel auf dem Gerät ausgeführt werden und wie sie in CUDA programmiert werden. Der Kernel wird mit einer *Aufrufkonfiguration*

aufgerufen, die angibt, wie viele parallele Threads erzeugt werden.

- *Das Raster* - stellt alle Threads dar, die beim Kernel-Aufruf erzeugt werden. Es wird als ein oder zwei dimensionale *Blöcke angegeben*
- *Der Block* - ist ein semi-unabhängiger Satz von *Threads* . Jeder Block ist einem einzelnen SM zugeordnet. Daher können Blöcke nur über den globalen Speicher kommunizieren. Blöcke werden in keiner Weise synchronisiert. Wenn zu viele Blöcke vorhanden sind, werden einige nacheinander ausgeführt. Auf der anderen Seite können, wenn die Ressourcen dies zulassen, mehr als ein Block auf demselben SM ausgeführt werden, aber der Programmierer kann davon nicht profitieren (abgesehen von der offensichtlichen Leistungssteigerung).
- *der Thread* - eine skalare Folge von Anweisungen, die von einem einzelnen CUDA-Kern ausgeführt werden. Threads sind "leicht" mit minimalem Kontext, sodass die Hardware sie schnell ein- und auswechseln kann. Aufgrund ihrer Anzahl arbeiten CUDA-Threads mit einigen zugewiesenen Registern und sehr kurzen Stapeln (vorzugsweise gar nicht!). Aus diesem Grund zieht es der CUDA-Compiler vor, alle Funktionsaufrufe einzubinden, um den Kernel so zu glätten, dass er nur statische Sprünge und Schleifen enthält. Funktions-Ponter-Aufrufe und Aufrufe von virtuellen Methoden werden zwar von den meisten neueren Geräten unterstützt, sind jedoch in der Regel mit einer erheblichen Leistungseinschränkung verbunden.

Jeder Thread wird durch einen Blockindex `blockIdx` und einen `blockIdx` innerhalb des Blockes `threadIdx` . Diese Nummern können jederzeit von jedem laufenden Thread überprüft werden und sind die einzige Möglichkeit, einen Thread von einem anderen zu unterscheiden.

Darüber hinaus sind Threads in *Warps* organisiert, die jeweils genau 32 Threads enthalten. Threads innerhalb eines einzelnen Warp werden in einer SIMD-Funktion in perfekter Synchronisation ausgeführt. Threads aus verschiedenen Warps, aber innerhalb desselben Blocks, können in beliebiger Reihenfolge ausgeführt werden, können aber vom Programmierer zur Synchronisation gezwungen werden. Threads aus verschiedenen Blöcken können nicht synchronisiert werden oder in irgendeiner Weise direkt interagieren.

Speicherorganisation

Bei der normalen CPU-Programmierung ist die Speicherorganisation normalerweise vor dem Programmierer verborgen. Typische Programme verhalten sich so, als wäre nur RAM vorhanden. Alle Speicheroperationen, wie z. B. das Verwalten von Registern, das L1-L2-L3-Caching, das Wechseln auf die Festplatte usw., werden vom Compiler, dem Betriebssystem oder der Hardware selbst ausgeführt.

Dies ist bei CUDA nicht der Fall. Während neuere GPU-Modelle die Belastung teilweise überdecken, z. B. durch das Unified Memory in CUDA 6, ist es dennoch aus Gründen der Leistung sinnvoll, die Organisation zu verstehen. Die grundlegende CUDA-Speicherstruktur sieht wie folgt aus:

- *Hostspeicher* - das reguläre RAM. Wird hauptsächlich vom Hostcode verwendet, aber auch neuere GPU-Modelle können darauf zugreifen. Wenn ein Kernel auf den Hostspeicher

zugreift, muss die GPU normalerweise über den PCIe-Connector mit der Hauptplatine kommunizieren und ist daher relativ langsam.
- *Gerätespeicher / Globaler Speicher* - der Hauptspeicher der GPU, der allen Threads zur Verfügung steht.
- *Shared Memory* - in jedem SM befindet sich ein viel schnellerer Zugriff als global. Der gemeinsam genutzte Speicher ist für jeden Block privat. Threads innerhalb eines einzelnen Blocks können es für die Kommunikation verwenden.
- *Register* - der schnellste, private, nicht adressierbare Speicher jedes Threads. Im Allgemeinen können diese nicht für die Kommunikation verwendet werden, aber einige intrinsische Funktionen ermöglichen das Mischen ihres Inhalts innerhalb eines Warp.
- Die *lokale Speicher* - private Speicher jeden Thread, die adressierbar *ist.* Dies wird für Registerüberläufe und lokale Arrays mit variabler Indizierung verwendet. Physisch befinden sie sich im globalen Speicher.
- *Texture memory, Constant memory* - ein Teil des globalen Speichers, der für den Kernel als unveränderlich markiert ist. Dies ermöglicht der GPU die Verwendung spezieller Caches.
- *L2-Cache* - auf dem Chip, für alle Threads verfügbar. In Anbetracht der Anzahl der Threads ist die erwartete Lebensdauer jeder Cachezeile viel niedriger als bei der CPU. Es wird meistens verwendet, um falsch ausgerichtete und teilweise zufällige Speicherzugriffsmuster zu unterstützen.
- *L1-Cache* - befindet sich im selben Speicherbereich wie der gemeinsam genutzte Speicher. Auch hier ist die Anzahl angesichts der Anzahl der verwendeten Threads eher gering. Erwarten Sie also nicht, dass die Daten dort lange bleiben. L1-Caching kann deaktiviert werden.

Versionen

Rechenleistung	Die Architektur	GPU-Codename	Veröffentlichungsdatum
1,0	Tesla	G80	2006-11-08
1.1	Tesla	G84, G86, G92, G94, G96, G98,	2007-04-17
1.2	Tesla	GT218, GT216, GT215	2009-04-01
1.3	Tesla	GT200, GT200b	2009-04-09
2,0	Fermi	GF100, GF110	2010-03-26
2.1	Fermi	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

Rechenleistung	Die Architektur	GPU-Codename	Veröffentlichungsdatum
3,5	Kepler	GK110, GK208	2013-02-19
3.7	Kepler	GK210	2014-11-17
5,0	Maxwell	GM107, GM108	2014-02-18
5.2	Maxwell	GM200, GM204, GM206	2014-09-18
5.3	Maxwell	GM20B	01.04.2015
6,0	Pascal	GP100	2016-10-01
6.1	Pascal	GP102, GP104, GP106	2016-05-27

Das Veröffentlichungsdatum kennzeichnet die Veröffentlichung der ersten GPU, die die angegebenen Berechnungsfunktionen unterstützt. Einige Daten sind ungefähr, z. B. wurde im 2. Quartal 2014 eine 3,2-Karte veröffentlicht.

Examples

Voraussetzungen

Laden Sie den CUDA Toolkit und den Entwicklertreiber herunter, um mit der Programmierung mit CUDA zu beginnen. Das Toolkit enthält `nvcc` , den NVIDIA CUDA Compiler und andere Software, die zur Entwicklung von CUDA-Anwendungen erforderlich ist. Der Treiber stellt sicher, dass GPU-Programme auf CUDA-fähiger Hardware ordnungsgemäß ausgeführt werden, was auch erforderlich ist.

Sie können bestätigen, dass das CUDA Toolkit auf Ihrem Computer ordnungsgemäß installiert ist, indem Sie `nvcc --version` über eine Befehlszeile `nvcc --version` . Zum Beispiel auf einem Linux-Rechner

```
$ 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
```

gibt die Compiler-Informationen aus. Wenn der vorherige Befehl nicht erfolgreich war, ist das CUDA Toolkit wahrscheinlich nicht installiert, oder der Pfad zu `nvcc` (`C:\CUDA\bin` auf Windows-Computern, `/usr/local/cuda/bin` unter POSIX-Betriebssystemen) gehört nicht zu Ihrem Umgebungsvariable `PATH`

Außerdem benötigen Sie einen Host-Compiler, der mit `nvcc` , um CUDA-Programme zu kompilieren und zu erstellen. Unter Windows ist dies `cl.exe` , der Microsoft-Compiler, der mit Microsoft Visual Studio `cl.exe` wird. Unter POSIX-Betriebssystemen sind andere Compiler

verfügbar, einschließlich `gcc` oder `g++` . Im offiziellen CUDA Quick Start Guide erfahren Sie, welche Compilerversionen auf Ihrer jeweiligen Plattform unterstützt werden.

Um sicherzustellen, dass alles korrekt eingerichtet ist, lassen Sie uns ein triviales CUDA-Programm kompilieren und ausführen, um sicherzustellen, dass alle Tools ordnungsgemäß zusammenarbeiten.

```
__global__ void foo() {}

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

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

  return 0;
}
```

Um dieses Programm zu kompilieren, kopieren Sie es in eine Datei namens test.cu und kompilieren Sie es über die Befehlszeile. Auf einem Linux-System sollte beispielsweise Folgendes funktionieren:

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

Wenn das Programm ohne Fehler erfolgreich ist, beginnen wir mit der Codierung!

Addieren Sie zwei Arrays mit CUDA

Dieses Beispiel zeigt, wie Sie ein einfaches Programm erstellen, das zwei `int` Arrays mit CUDA summiert.

Ein CUDA-Programm ist heterogen und besteht aus Teilen, die auf CPU und GPU laufen.

Die Hauptteile eines Programms, die CUDA verwenden, sind ähnlich wie CPU-Programme und bestehen aus

- Speicherzuordnung für Daten, die auf der GPU verwendet werden
- Daten werden vom Hostspeicher in den GPU-Speicher kopiert
- Aufrufen der Kernel-Funktion zum Verarbeiten von Daten
- Ergebnis in den CPU-Speicher kopieren

Um Speicherplatz für Geräte zuzuweisen, verwenden wir die `cudaMalloc` Funktion. Zum Kopieren von Daten zwischen Gerät und Host `cudaMemcpy` Funktion `cudaMemcpy` verwendet werden. Das letzte Argument von `cudaMemcpy` gibt die Richtung des Kopiervorgangs an. Es gibt 5 mögliche Typen:

- `cudaMemcpyHostToHost` - Host -> Host
- `cudaMemcpyHostToDevice` - Host -> Gerät
- `cudaMemcpyDeviceToHost` - Gerät -> Host

- `cudaMemcpyDeviceToDevice` - Gerät -> Gerät
- `cudaMemcpyDefault` - Standardbasierter, einheitlicher virtueller Adressraum

Als nächstes wird die Kernel-Funktion aufgerufen. Die Information zwischen den dreifachen Chevrons ist die Ausführungskonfiguration, die bestimmt, wie viele Gerätethreads den Kernel parallel ausführen. Die erste Anzahl (`2` im Beispiel) gibt die Anzahl der Blöcke und die zweite (`(size + 1) / 2` im Beispiel) die Anzahl der Threads in einem Block an. Beachten Sie, dass wir in diesem Beispiel die Größe um 1 erhöhen, sodass wir einen zusätzlichen Thread anfordern, anstatt einen Thread für zwei Elemente verantwortlich zu machen.

Da der `cudaDeviceSynchronize` eine asynchrone Funktion ist, wird `cudaDeviceSynchronize` aufgerufen, um zu warten, bis die Ausführung abgeschlossen ist. Ergebnis-Arrays werden in den `cudaFree` kopiert und der gesamte auf dem Gerät zugewiesene Speicher wird mit `cudaFree` .

Um die Funktion als Kernel zu definieren, wird der `__global__` Deklarationsbezeichner verwendet. Diese Funktion wird von jedem Thread aufgerufen. Wenn jeder Thread ein Element des resultierenden Arrays verarbeiten soll, brauchen wir ein Mittel, um jeden Thread zu unterscheiden und zu identifizieren. CUDA definiert die Variablen `blockDim` , `blockIdx` und `threadIdx` . Die vordefinierte Variable `blockDim` enthält die Dimensionen jedes Thread-Blocks, wie im zweiten Konfigurationsparameter für den Kernel-Start angegeben. Die vordefinierten Variablen `threadIdx` und `blockIdx` enthalten den Index des Threads in seinem `threadIdx` bzw. den `blockIdx` innerhalb des Gitters. Da wir möglicherweise einen Thread mehr anfordern als Elemente in den Arrays, müssen wir die `size` um sicherzustellen, dass wir nicht über das Ende des Arrays hinaus zugreifen.

```
#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;
}
```

Lassen Sie uns einen einzelnen CUDA-Thread starten, um Hallo zu sagen

Dieses einfache CUDA-Programm zeigt, wie eine Funktion geschrieben wird, die auf der GPU (auch als "Gerät" bezeichnet) ausgeführt wird. Die CPU oder "Host" erstellt CUDA-Threads durch Aufrufen spezieller Funktionen, die als "Kernels" bezeichnet werden. CUDA-Programme sind C ++ - Programme mit zusätzlicher Syntax.

Um zu sehen, wie es funktioniert, `hello.cu` den folgenden Code in eine Datei namens `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;
}
```

(Beachten Sie, dass Sie zur Verwendung der `printf` Funktion auf dem Gerät ein Gerät mit einer Rechenkapazität von mindestens 2,0 benötigen. Weitere Informationen finden Sie in der Versionsübersicht .)

Lassen Sie uns nun das Programm mit dem NVIDIA-Compiler kompilieren und ausführen:

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

Einige zusätzliche Informationen zum obigen Beispiel:

- `nvcc` steht für "NVIDIA CUDA Compiler". Es trennt den Quellcode in Host- und Gerätekomponenten.
- `__global__` ist ein CUDA-Schlüsselwort, das in Funktionsdeklarationen verwendet wird, um `__global__` , dass die Funktion auf dem GPU-Gerät ausgeführt wird und vom Host aufgerufen wird.
- Dreifache spitze Klammern (`<<<` , `>>>`) kennzeichnen einen Aufruf vom Hostcode zum Gerätecode (auch als "Kernel-Start" bezeichnet). Die Zahlen in diesen dreifachen Klammern geben die Anzahl der parallel auszuführenden Zeiten und die Anzahl der Threads an.

Kompilieren und Ausführen der Beispielprogramme

Das NVIDIA-Installationshandbuch endet mit der Ausführung der Beispielprogramme, um die Installation des CUDA Toolkit zu überprüfen, gibt jedoch nicht explizit die Vorgehensweise an. Überprüfen Sie zunächst alle Voraussetzungen. Überprüfen Sie das Standard-CUDA-Verzeichnis für die Beispielprogramme. Wenn es nicht vorhanden ist, kann es von der offiziellen CUDA-Website heruntergeladen werden. Navigieren Sie zu dem Verzeichnis, in dem die Beispiele vorhanden sind.

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

Sie sollten eine Ausgabe ähnlich der folgenden sehen:

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

Stellen Sie sicher, dass das `Makefile` in diesem Verzeichnis vorhanden ist. Mit dem Befehl `make` in UNIX-basierten Systemen werden alle Beispielprogramme erstellt. Navigieren Sie alternativ zu einem Unterverzeichnis, in dem sich ein anderes `Makefile` befindet, und führen Sie den Befehl `make` von dort aus aus, um nur dieses Beispiel zu erstellen.

Führen Sie die zwei empfohlenen Beispielprogramme aus - `deviceQuery` und `bandwidthTest` :

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

Die Ausgabe ähnelt der unten gezeigten:

```
./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
```

Die Anweisung `Result = PASS` am Ende zeigt an, dass alles korrekt funktioniert. Führen Sie nun das andere vorgeschlagene Beispielprogramm `bandwidthTest` auf ähnliche Weise aus. Die Ausgabe wird ähnlich sein:

```
[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.
```

Die Anweisung `Result = PASS` zeigt erneut an, dass alles ordnungsgemäß ausgeführt wurde. Alle anderen Beispielprogramme können auf ähnliche Weise ausgeführt werden.

Erste Schritte mit cuda online lesen: https://riptutorial.com/de/cuda/topic/1860/erste-schritte-mit-cuda

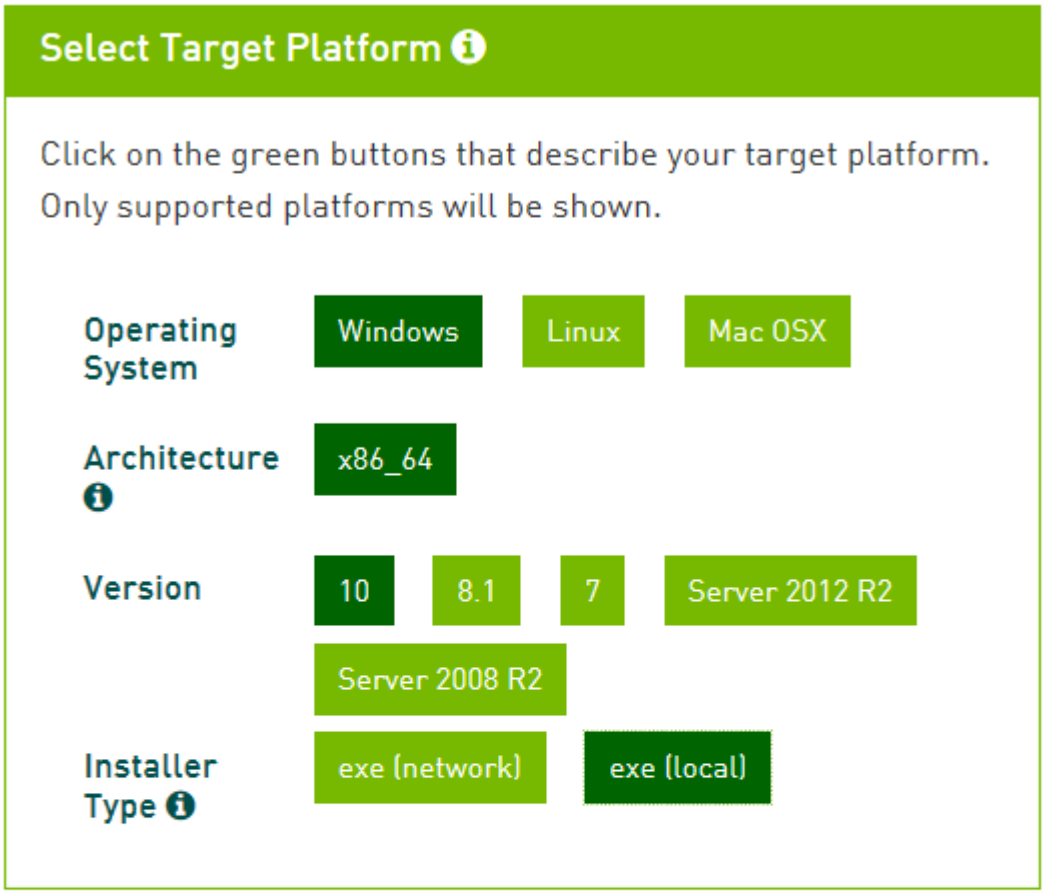
Kapitel 2: Cuda installieren

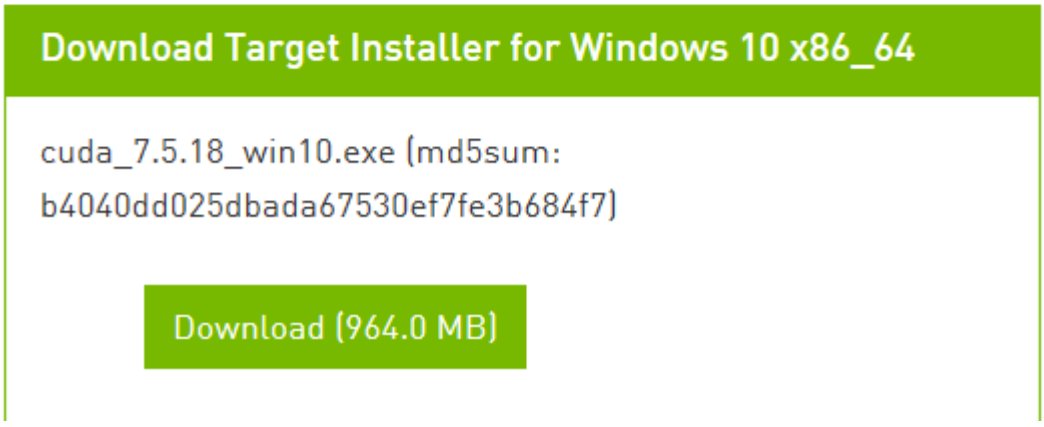
Bemerkungen

Um das CUDA-Toolkit unter Windows zu installieren, müssen Sie zunächst eine geeignete Version von Visual Studio installieren. Wenn Sie CUDA 7.0 oder 7.5 installieren, sollte Visual Studio 2013 installiert sein. Visual Studio 2015 wird für CUDA 8.0 und höher unterstützt.

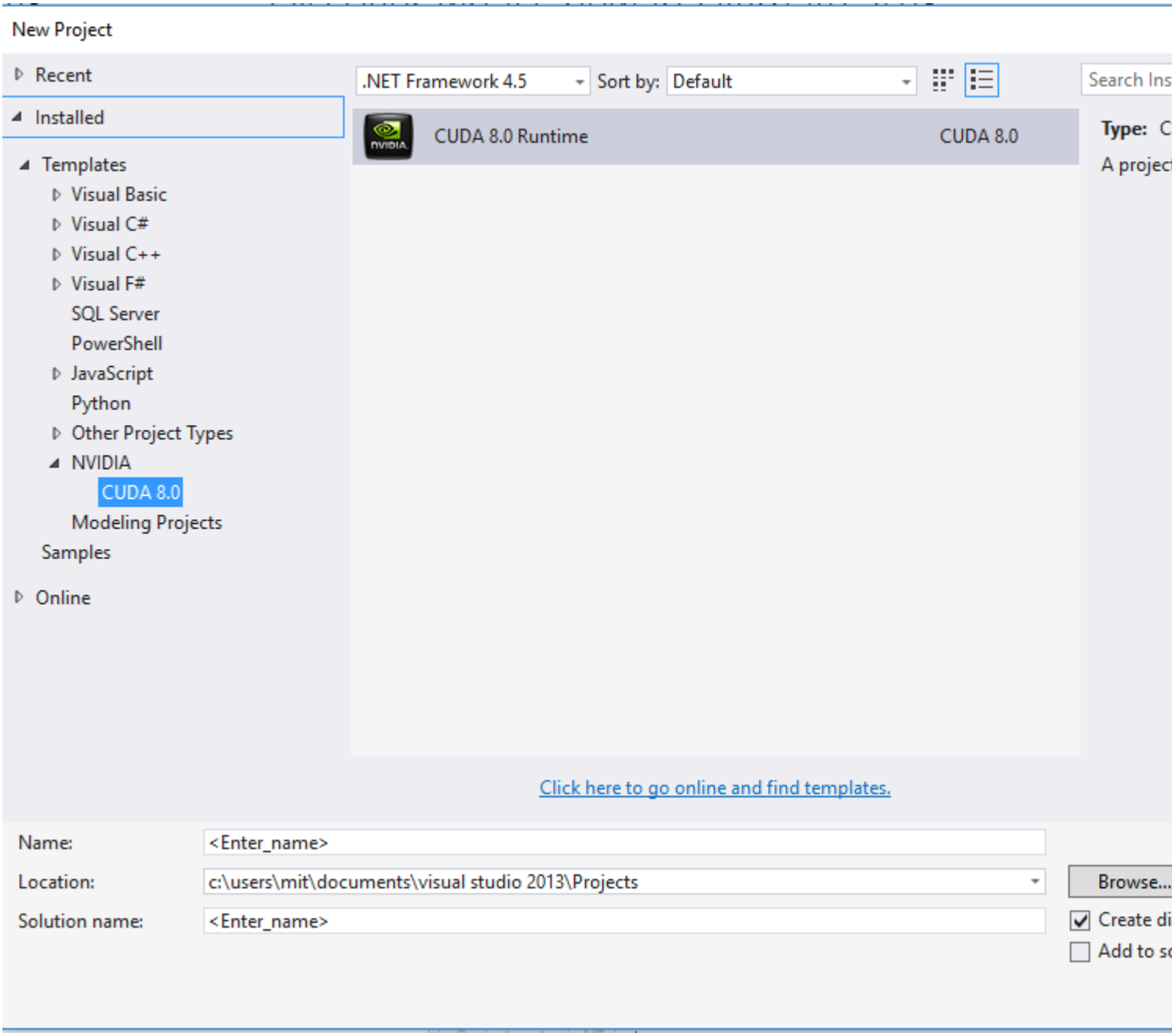
Wenn Sie eine geeignete Version von VS auf Ihrem System installiert haben, ist es an der Zeit, CUDA Toolkit herunterzuladen und zu installieren. Folgen Sie diesem Link, um die gewünschte Version des CUDA-Toolkits zu finden: CUDA-Toolkit-Archiv

Auf der Download-Seite sollten Sie die Version der Fenster auf dem Zielcomputer und den Installer-Typ (lokal auswählen) auswählen.





Nachdem Sie die exe-Datei heruntergeladen haben, müssen Sie sie extrahieren und `setup.exe` . Wenn die Installation abgeschlossen ist, öffnen Sie ein neues Projekt und wählen Sie in den Vorlagen NVIDIA> CUDAX.X aus.



Denken Sie daran, dass die Erweiterung der CUDA-Quelldateien `.cu` . Sie können sowohl Host- als auch Gerätecodes in dieselbe Quelle schreiben.

Examples

Sehr einfacher CUDA-Code

```
#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;
}
```

Cuda installieren online lesen: https://riptutorial.com/de/cuda/topic/10949/cuda-installieren

Kapitel 3: Kommunikation zwischen Blöcken

Bemerkungen

Blöcke in CUDA arbeiten halb unabhängig. Es gibt keine sichere Methode, um alle zu synchronisieren. Dies bedeutet jedoch nicht, dass sie nicht in irgendeiner Weise miteinander interagieren können.

Examples

Last-Block-Wächter

Stellen Sie sich ein Netz vor, das an einer Aufgabe arbeitet, z. B. eine parallele Reduzierung. Anfangs kann jeder Block seine Arbeit unabhängig machen, was zu einem Teilergebnis führt. Am Ende müssen jedoch die Teilergebnisse kombiniert und zusammengeführt werden. Ein typisches Beispiel ist ein Reduktionsalgorithmus für Big Data.

Ein typischer Ansatz besteht darin, zwei Kernel aufzurufen, einen für die Teilberechnung und den anderen für das Zusammenführen. Wenn das Zusammenführen jedoch durch einen einzelnen Block effizient durchgeführt werden kann, ist nur ein Kernelaufruf erforderlich. Dies wird durch einen `lastBlock` Guard erreicht, der wie `lastBlock` definiert ist:

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;
}
```

Mit einem solchen Schutz werden garantiert, dass der letzte Block alle Ergebnisse aller anderen Blöcke anzeigt und das Zusammenführen ausführen kann.

```
__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);
    }
}
```

Annahmen:

- Der Zähler muss ein globaler Speicherzeiger sein, der *vor* dem Aufrufen des Kernels auf 0 initialisiert wird.
- Die `lastBlock` Funktion wird von allen Threads in allen Blöcken einheitlich aufgerufen
- Der Kernel wird in einem eindimensionalen Raster aufgerufen (zur Vereinfachung des Beispiels)
- `T` benennt jeden beliebigen Typ, aber das Beispiel soll keine Vorlage im Sinne von C ++ sein

Globale Arbeitswarteschlange

Betrachten Sie eine Reihe von Arbeitselementen. Die Zeit, die ein Workitem zum Abschluss benötigt, ist sehr unterschiedlich. Um die Arbeitsverteilung zwischen den Blöcken auszugleichen, kann es sinnvoll sein, für jeden Block den nächsten Artikel erst dann zu holen, wenn der vorherige abgeschlossen ist. Dies steht im Gegensatz zu dem Zuordnen von Elementen zu Blöcken von vornherein.

```
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
  }
}
```

Annahmen:

- Das WorkQueue-Objekt sowie das gItem-Array befinden sich im globalen Speicher
- Dem WorkQueue-Objekt im Kernel, der es abruft, werden keine neuen Arbeitselemente hinzugefügt
- Das `WorkItem` ist eine kleine Darstellung der Arbeitszuordnung, z. B. ein Zeiger auf ein anderes Objekt

- `WorkItem::none()` statische Member-Funktion `WorkItem::none()` erstellt ein `WorkItem` Objekt, das überhaupt keine Arbeit darstellt
- `WorkQueue::fetch()` muss von allen Threads im Block einheitlich aufgerufen werden
- Es gibt keine zwei `WorkQueue::fetch()` von `WorkQueue::fetch()` ohne einen anderen `__syncthreads()` dazwischen. Andernfalls wird eine Rennbedingung angezeigt!

Das Beispiel beinhaltet nicht, wie `WorkQueue` initialisiert oder `WorkQueue` wird. Es wird von einem anderen Kernel- oder CPU-Code ausgeführt und sollte ziemlich geradlinig sein.

Kommunikation zwischen Blöcken online lesen:
https://riptutorial.com/de/cuda/topic/4978/kommunikation-zwischen-blocken

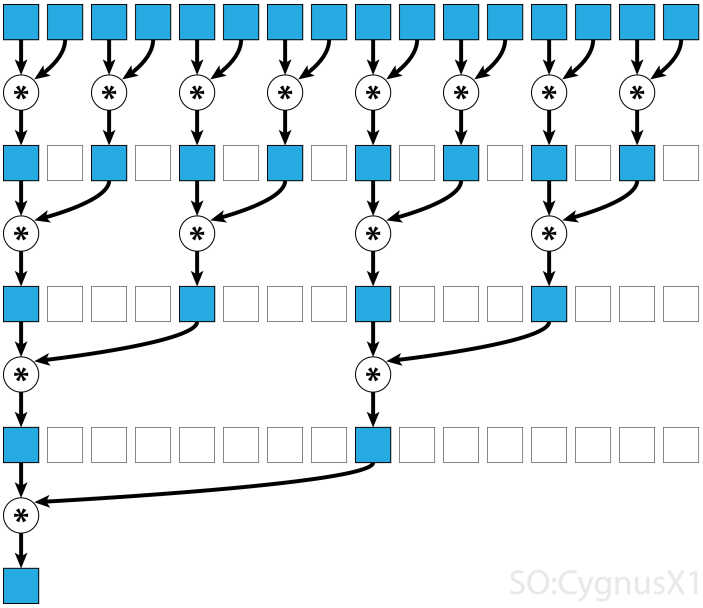
Kapitel 4: Parallele Reduktion (z. B. wie ein Array summiert wird)

Bemerkungen

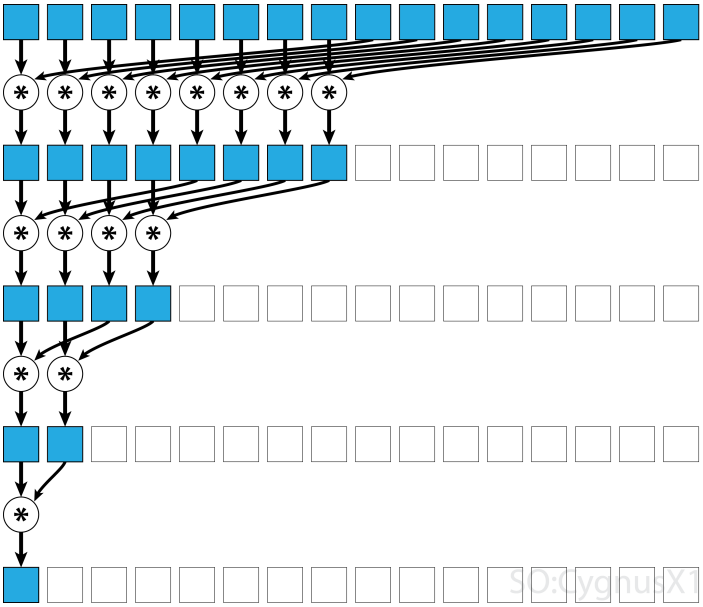
Paralleler Reduktionsalgorithmus bezieht sich typischerweise auf einen Algorithmus, der ein Array von Elementen kombiniert, um ein einzelnes Ergebnis zu erzeugen. Typische Probleme, die in diese Kategorie fallen, sind:

- Aufsummieren aller Elemente in einem Array
- ein Maximum in einem Array finden

Im Allgemeinen kann die parallele Reduktion für jeden binären assoziativen Operator angewendet werden , dh `(A*B)*C = A*(B*C)` . Mit einem solchen Operator * gruppiert der Parallelreduktionsalgorithmus die Array-Argumente wiederholt in Paaren. Jedes Paar wird parallel zu anderen berechnet, wobei die Gesamtgröße des Arrays in einem Schritt halbiert wird. Der Vorgang wird wiederholt, bis nur ein einziges Element vorhanden ist.



Wenn der Operator kommutativ ist (dh `A*B = B*A`) und nicht assoziativ ist, kann der Algorithmus in einem anderen Muster paaren. Vom theoretischen Standpunkt aus macht es keinen Unterschied, aber in der Praxis ergibt sich ein besseres Speicherzugriffsmuster:



Nicht alle assoziativen Operatoren sind kommutativ - nehmen Sie zum Beispiel die Matrixmultiplikation.

Examples

Parallele Reduktion um einen Block für kommutative Operatoren

Der einfachste Ansatz für die parallele Reduzierung von CUDA besteht darin, einen einzelnen Block zur Ausführung der Aufgabe zuzuweisen:

```
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);
```

Dies ist am besten machbar, wenn die Datengröße nicht sehr groß ist (etwa einige tausend Elemente). Dies geschieht normalerweise, wenn die Reduktion Teil eines größeren CUDA-Programms ist. Wenn die Eingabe von Anfang an mit `blockSize` , kann die erste `for` Schleife

vollständig entfernt werden.

Beachten Sie, dass wir im ersten Schritt, wenn es mehr Elemente als Threads gibt, die Dinge vollständig unabhängig voneinander addieren. Nur wenn das Problem auf `blockSize` reduziert `blockSize` , wird die tatsächliche parallele Reduktion `blockSize` . Derselbe Code kann auf jeden anderen kommutativen, assoziativen Operator angewendet werden, wie Multiplikation, Minimum, Maximum usw.

Beachten Sie, dass der Algorithmus schneller gemacht werden kann, z. B. durch eine parallele Reduktion auf Warp-Ebene.

Parallele Reduktion um einen Block für nicht kommutativen Operator

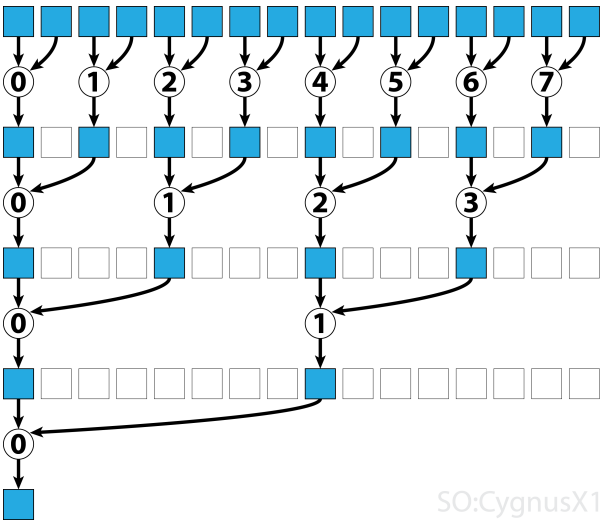
Die parallele Reduzierung für einen nicht kommutativen Operator ist im Vergleich zur kommutativen Version etwas komplizierter. Im Beispiel verwenden wir der Einfachheit halber immer noch eine Addition über Ganzzahlen. Sie könnte beispielsweise durch eine Matrixmultiplikation ersetzt werden, die eigentlich nicht kommutativ ist. Beachten Sie dabei, dass 0 durch ein neutrales Element der Multiplikation ersetzt werden sollte, dh eine Identitätsmatrix.

```
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);
```

In der ersten while-Schleife wird ausgeführt, solange mehr Eingabeelemente als Threads

vorhanden sind. Bei jeder Iteration wird eine einzelne Reduktion durchgeführt und das Ergebnis in die erste Hälfte des `shArr` Arrays `shArr` . Die zweite Hälfte wird dann mit neuen Daten gefüllt.

Sobald alle Daten von `gArr` geladen `gArr` , wird die zweite Schleife ausgeführt. Jetzt komprimieren wir das Ergebnis nicht mehr (was `__syncthreads()` zusätzlich `__syncthreads()`). In jedem Schritt greift der Thread n auf das `2*n` te aktive Element zu und summiert es mit `2*n+1` ten Element auf:



Es gibt viele Möglichkeiten, dieses einfache Beispiel weiter zu optimieren, z. B. durch Reduzierung des Warp-Pegels und durch das Entfernen von Konflikten in der gemeinsamen Speicherbank.

Parallele Multi-Block-Reduktion für kommutativen Operator

Ein Ansatz mit mehreren Blöcken zur parallelen Reduktion von CUDA stellt im Vergleich zum Ansatz mit einem Block eine zusätzliche Herausforderung dar, da Blöcke in der Kommunikation begrenzt sind. Die Idee ist, dass jeder Block einen Teil des Eingabearrays berechnen lässt und dann einen letzten Block hat, um alle Teilergebnisse zusammenzuführen. Dazu kann man zwei Kernel starten und implizit einen gitterweiten Synchronisationspunkt erstellen.

```
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;
}
```

Idealerweise möchte man genügend Blöcke starten, um alle Multiprozessoren der GPU bei voller Belegung zu sättigen. Das Überschreiten dieser Anzahl, insbesondere das Starten von so vielen Threads, wie Elemente im Array vorhanden sind, ist kontraproduktiv. Dadurch wird die reine Rechenleistung nicht mehr erhöht, sondern die Verwendung der sehr effizienten ersten Schleife verhindert.

Es ist auch möglich, mit Hilfe des Last-Block-Guard das gleiche Ergebnis mit einem einzelnen Kernel zu erzielen:

```
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;
}
```

Beachten Sie, dass der Kernel schneller gemacht werden kann, beispielsweise durch eine parallele Reduktion auf Warp-Ebene.

Parallele Multi-Block-Reduktion für nichtkommutativen Operator

Der Multi-Block-Ansatz für die parallele Reduktion ist dem Single-Block-Ansatz sehr ähnlich. Das globale Eingabearray muss in Abschnitte unterteilt werden, die jeweils um einen einzelnen Block reduziert sind. Wenn ein Teilergebnis von jedem Block erhalten wird, reduziert ein letzter Block diese, um das Endergebnis zu erhalten.

- `sumNoncommSingleBlock` wird im Beispiel zur Reduzierung von Einzelblöcken ausführlicher erklärt.
- `lastBlock` akzeptiert nur den letzten Block, der ihn erreicht. Wenn Sie dies vermeiden möchten, können Sie den Kernel in zwei separate Aufrufe aufteilen.

```
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);
}
```

Idealerweise möchte man genügend Blöcke starten, um alle Multiprozessoren der GPU bei voller Belegung zu sättigen. Das Überschreiten dieser Anzahl, insbesondere das Starten von so vielen Threads, wie Elemente im Array vorhanden sind, ist kontraproduktiv. Dadurch wird die reine Rechenleistung nicht mehr erhöht, sondern die Verwendung der sehr effizienten ersten Schleife verhindert.

Single-Warp-Parallelreduktion für kommutative Operatoren

Manchmal muss die Reduktion in sehr kleinem Umfang als Teil eines größeren CUDA-Kernels durchgeführt werden. Angenommen, die Eingabedaten haben genau 32 Elemente - die Anzahl der Threads in einem Warp. In einem solchen Szenario kann ein einzelner Warp zugewiesen werden, um die Reduzierung durchzuführen. Da Warp in einer perfekten Synchronisation ausgeführt wird, können viele `__syncthreads()` Anweisungen entfernt werden - verglichen mit einer Reduktion auf Blockebene.

```
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` ist vorzugsweise ein Array im Shared Memory. Der Wert sollte für alle Threads im Warp derselbe sein. Wenn `sumCommSingleWarp` von mehreren Warps `shArr` sollte `shArr` zwischen Warps unterschiedlich sein (in jedem Warp gleich).

Das Argument `shArr` wird als `volatile` markiert, um sicherzustellen, dass Operationen an dem Array tatsächlich ausgeführt werden, `shArr` angegeben. Andernfalls kann die wiederholte Zuordnung zu `shArr[idx]` als Zuordnung zu einem Register optimiert werden, wobei nur die endgültige Zuordnung ein tatsächlicher Speicher für `shArr` . In diesem Fall sind die unmittelbaren Zuweisungen für andere Threads nicht sichtbar, was zu falschen Ergebnissen führt. Beachten Sie, dass Sie ein normales nichtflüchtiges Array als Argument eines flüchtigen Arrays übergeben können, genauso wie wenn Sie non-const als const-Parameter übergeben.

Wenn sich der Inhalt von `shArr[1..31]` nach der Reduktion nicht interessiert, kann der Code noch weiter vereinfacht werden:

```
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];
}
```

In diesem Setup haben wir viele `if` Bedingungen entfernt. Die zusätzlichen Threads führen einige unnötige Ergänzungen aus, aber wir kümmern uns nicht mehr um die Inhalte, die sie produzieren.

Da Warps im SIMD-Modus ausgeführt werden, sparen wir eigentlich keine Zeit, da diese Threads nichts unternehmen. Auf der anderen Seite dauert die Bewertung der Bedingungen relativ viel Zeit, da der Rumpf dieser `if` Anweisungen so klein ist. Die anfängliche `if` Anweisung kann auch entfernt werden, wenn `shArr[32..47]` mit 0 aufgefüllt wird.

Die Reduzierung des Warp-Pegels kann auch verwendet werden, um die Reduktion auf Blockebene zu beschleunigen:

```
__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];
    }
}
```

Das Argument `&r[idx & ~(warpSize-1)]` lautet grundsätzlich `r + warpIdx*32` . Dadurch wird das `r` Array effektiv in Blöcke mit 32 Elementen aufgeteilt, und jeder Block wird einem separaten Warp zugewiesen.

Single-Warp-Parallelreduktion für nichtkommutative Operatoren

Manchmal muss die Reduktion in sehr kleinem Umfang als Teil eines größeren CUDA-Kernels durchgeführt werden. Angenommen, die Eingabedaten haben genau 32 Elemente - die Anzahl der Threads in einem Warp. In einem solchen Szenario kann ein einzelner Warp zugewiesen werden, um die Reduzierung durchzuführen. Da Warp in einer perfekten Synchronisation ausgeführt wird, können viele `__syncthreads()` Anweisungen entfernt werden - verglichen mit einer Reduktion auf Blockebene.

```
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` ist vorzugsweise ein Array im Shared Memory. Der Wert sollte für alle Threads im Warp derselbe sein. Wenn `sumCommSingleWarp` von mehreren Warps `shArr` sollte `shArr` zwischen Warps unterschiedlich sein (in jedem Warp gleich).

Das Argument `shArr` wird als `volatile` markiert, um sicherzustellen, dass Operationen an dem Array tatsächlich ausgeführt werden, `shArr` angegeben. Andernfalls kann die wiederholte Zuordnung zu `shArr[idx]` als Zuordnung zu einem Register optimiert werden, wobei nur die endgültige Zuordnung ein tatsächlicher Speicher für `shArr` . In diesem Fall sind die unmittelbaren Zuweisungen für andere Threads nicht sichtbar, was zu falschen Ergebnissen führt. Beachten Sie, dass Sie ein normales nichtflüchtiges Array als Argument eines flüchtigen Arrays übergeben können, genauso wie wenn Sie non-const als const-Parameter übergeben.

Wenn man sich nicht für den endgültigen Inhalt von `shArr[1..31]` und `shArr[32..47]` mit Nullen `shArr[32..47]` kann, kann man den obigen Code vereinfachen:

```
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];
}
```

In diesem Setup haben wir alle `if` Bedingungen entfernt, die etwa die Hälfte der Anweisungen ausmachen. Die zusätzlichen Threads führen einige unnötige Hinzufügungen durch und speichern das Ergebnis in Zellen von `shArr` , die letztendlich keinen Einfluss auf das Endergebnis haben. Da Warps im SIMD-Modus ausgeführt werden, sparen wir eigentlich keine Zeit, da diese Threads nichts unternehmen.

Single-Warp-Parallel-Reduktion nur mit Registern

Normalerweise wird die Reduktion für globale oder gemeinsam genutzte Arrays durchgeführt. Wenn die Reduktion jedoch in sehr kleinem Umfang als Teil eines größeren CUDA-Kernels durchgeführt wird, kann sie mit einem einzelnen Warp durchgeführt werden. In diesem Fall ist es bei Keppler- oder höheren Architekturen (CC> = 3.0) möglich, Warp-Shuffle-Funktionen zu verwenden, um den gemeinsamen Speicher zu vermeiden.

Angenommen, jeder Thread in einem Warp enthält einen einzelnen Eingabedatenwert. Alle Threads zusammen haben 32 Elemente, die wir zusammenfassen müssen (oder andere assoziative Operationen ausführen).

```
__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);
}
```

Diese Version funktioniert sowohl für kommutative als auch für nicht kommutative Operatoren.

Parallele Reduktion (z. B. wie ein Array summiert wird) online lesen:
https://riptutorial.com/de/cuda/topic/6566/parallele-reduktion--z--b--wie-ein-array-summiert-wird-

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit cuda	Community, CygnusX1, Dev-iL, harrism, havogt, infinite.potential, Jared Hoberock, Ken Y-N, Marco13, NikolayKondratyev, Tejus Prasad
2	Cuda installieren	Mo Sani
3	Kommunikation zwischen Blöcken	CygnusX1, tera
4	Parallele Reduktion (z. B. wie ein Array summiert wird)	CygnusX1