

 免费电子书

学习

cuda

Free unaffiliated eBook created from
Stack Overflow contributors.

#cuda

.....	1
1: cuda	2
.....	2
.....	2
.....	2
CUDA.....	2
.....	2
.....	2
Examples.....	3
.....	3
CUDA.....	4
CUDA.....	5
.....	6
2:	9
.....	9
Examples.....	9
.....	9
.....	9
3: cuda	11
.....	11
Examples.....	12
CUDA.....	12
4:	14
.....	14
Examples.....	14
.....	14
.....	15
.....	16
.....	18
.....	19
.....	20
.....	

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

1: cuda

CUDAGPUNVIDIA。

GPU。 GPU。 GPUGPU。 CUDANVIDIA GPUC ++。 C ++ - 。

- *host* - CPU
- *device* - CUDAGPU。 。
- *kernel* - 。

CUDAGPU

- - GPU。 GPU。
- SM - ~100SM。 SMSM。
- *CUDA* - SM。 。

SMwarp。 CUDA。 SM32SIMD。

CUDA

GPUCUDA。 。

- *grid* - 。
- - 。
- - CUDA。 “”。

blockIdxthreadIdxthreadIdx 。

warp warp32。 warpSIMD fahsion。 warp。 。

CPU。 RAM。 L1-L2-L3。

CUDA。 GPUCUDA 6。 CUDA

- - RAM。 GPU。 GPUPCie。
- /- GPU。
- - SM。 。
- - 。
- - 。
- - GPU。
- L2 - CPU。 。
- L1 - 。

	GPU	
1.0	G80	2006-11-08
1.1	G84G86G92G94G96G98	2007-04-17
1.2	GT218GT216GT215	2009-04-01
1.3	GT200GT200b	2009-04-09
2.0	GF100GF110	2010-03-26
2.1	GF104GF106 GF108GF114GF116GF117GF119	2010-07-12
3.0	GK104GK106GK107	2012-03-22
3.2	GK20A	201441
3.5	GK110GK208	2013219
3.7	GK210	
5	GM107GM108	2014218
5.2	GM200GM204GM206	2014-09-18
5.3	GM20B	2015-04-01
6	GP100	2016101
6.1	GP102GP104GP106	2016527

GPU。 20143.2。

Examples

CUDA [CUDA Toolkit](#)。 `nvcc` NVIDIA CUDACUDA。 GPU [CUDA](#)。

`nvcc --version` [CUDA Toolkit](#)。 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
```

。 [CUDA Toolkit](#) `nvcc` Windows `C:\CUDA\bin /usr/local/cuda/bin` POSIX `/usr/local/cuda/bin` PATH。

`nvcc` [CUDA](#)。 Windows Microsoft Visual Studio Microsoft `cl.exe`。 POSIX OS `gcc++`。 [CUDA](#)。

CUDA

```
__global__ void foo() {}

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

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

    return 0;
}
```

test.cu Linux

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

CUDA

CUDA_{int}

CUDACPUGPU

CUDACPU

- GPU
- GPU
-
- CPU

cudaMalloc° cudaMemcpy° cudaMemcpy° 5

- cudaMemcpyHostToHost - - >
- cudaMemcpyHostToDevice - - >
- cudaMemcpyDeviceToHost - - >
- cudaMemcpyDeviceToDevice - - >
- cudaMemcpyDefault -

◦ $V = 2 \cdot (\text{size} + 1) / 2 - 1$

cudaDeviceSynchronize° cudaDeviceSynchronize° cudaFree°

__global__ ° ° ° **CUDA** blockDim blockIdx threadIdx ° blockDim° threadIdx blockIdx° size°

```
#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;
}

```

CUDA

CUDA GPU。 CPU。 CUDA。 CUDAC ++。

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;
}
```

printf2.0。 ◦

NVIDIA

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

- nvcc“NVIDIA CUDA”。 ◦
- __global__CUDAGPU。 ◦
- <<< >>>“””。 ◦

NVIDIACUDA Toolkit。 ◦ CUDA。 CUDA。 ◦

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

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

Makefile。 UNIXmake。 Makefilemake。 ◦

- deviceQuerybandwidthTest

```
$ cd 1_Uutilities/deviceQuery/
$ ./deviceQuery
```

```
./deviceQuery Starting...
```

```

CUDA Device Query (Runtime API) version (CUDA 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

```

Result = PASS◦ bandwidthTest ◦

```

[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.

Result = PASS° °

cuda <https://riptutorial.com/zh-CN/cuda/topic/1860/cuda>

2:

CUDA . . .

Examples

. . . .

. . lastBlock guard

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;
}

```

.

```

__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);
    }
}

```

- 0.
- lastBlock
-
- **TC ++**

. . . .

```

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
    }
}

```

- WorkQueueItem
- WorkQueue
- WorkItem
- WorkItem::none()WorkItem
- WorkQueue::fetch()
- WorkQueue::fetch()² WorkQueue::fetch()__syncthreads() ◦

WorkQueue◦ CPU◦

<https://riptutorial.com/zh-CN/cuda/topic/4978/-->

3: cuda

WindowsCUDAVisual Studio。 CUDA 7.07.5Visual Studio 2013。 CUDA 8.0Visual Studio 2015。

VSCUDA。 CUDA [CUDA](#)

Windows。

Select Target Platform ⓘ

Click on the green buttons that describe your target platform.
Only supported platforms will be shown.

Operating System	Windows	Linux	Mac OSX
Architecture ⓘ	x86_64		
Version	10	8.1	7
	Server 2012 R2		
	Server 2008 R2		
Installer Type ⓘ	exe (network)	exe (local)	

Download Target Installer for Windows 10 x86_64

cuda_7.5.18_win10.exe (md5sum:
b4040dd025dbada67530ef7fe3b684f7)

[Download \(964.0 MB\)](#)

exe_{setup.exe}。 NVIDIA> CUDAX.X。

New Project

Recent

Installed

Templates

- Visual Basic
- Visual C#
- Visual C++
- Visual F#
- SQL Server
- PowerShell
- JavaScript
- Python
- Other Project Types
- NVIDIA
 - CUDA 8.0**
 - Modeling Projects
 - Samples

Online

.NET Framework 4.5 Sort by: Default

CUDA 8.0 Runtime CUDA 8.0

Search Ins... Type: C A project

[Click here to go online and find templates.](#)

Name: <Enter_name>

Location: c:\users\mit\documents\visual studio 2013\Projects Browse...

Solution name: <Enter_name> Create di Add to s

CUDA.cu ◦ ◦

Examples

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 >> >();

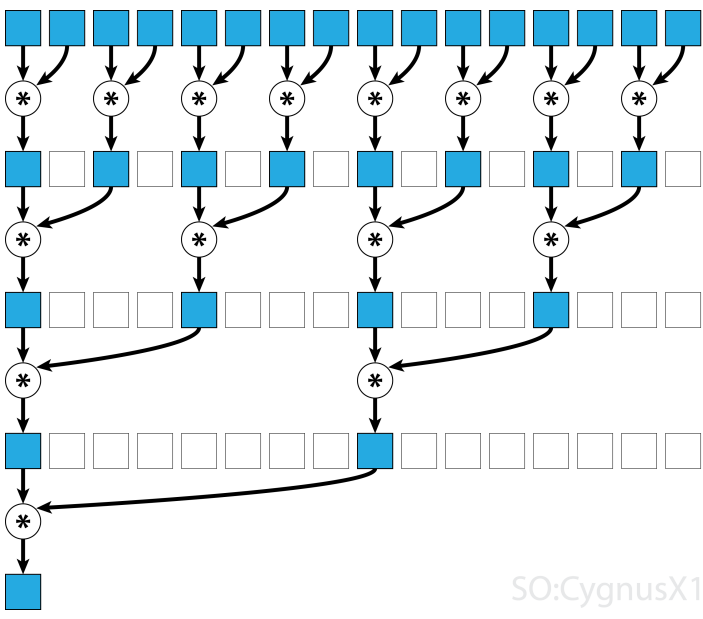
    return 0;
}
```

cuda <https://riptutorial.com/zh-CN/cuda/topic/10949/cuda>

4:

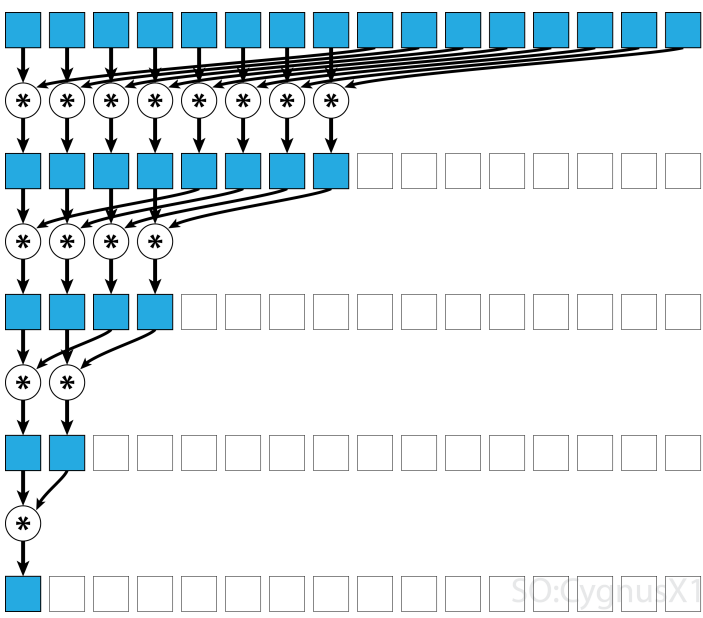
-
-
-

$$(A*B)*C = A*(B*C) \circ * \circ \circ$$



SO: CygnusX1

$$A*B = B*A \circ$$



SO: CygnusX1

- ◦

Examples

CUDA

```

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);

```

thousands. CUDA. blockSize for.

- blockSize blockSize ◦
-
- ◦ ◦ 0 - ◦

```

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];
}

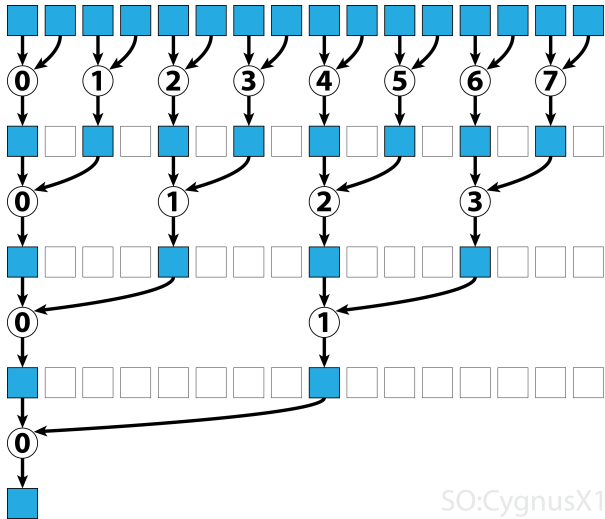
...

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

```

while shArr .

```
gArr __syncthreads() . n2*n2*n+1
```



SO:GygnusX1

warp .

CUDA . . .

```

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;
}

```

GPU。 - - 。 。

last-block guard

```

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 == blockDim.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;
}

```

warp◦

◦ ◦ ◦

- `sumNoncommSingleBlock◦`
- `lastBlock◦ ◦`

```

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);
}

```

GPU。 - - 。 。

CUDA。 32 - warp。 。 warp__syncthreads()。

```

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。 warp。 sumCommSingleWarpwarpshArrshArrwarp。

shArrvolatile。 shArr[idx]shArr。 。 volatileconstconst。

shArr[1..31]

```

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];
}

```

if ◦ ◦ **warpSIMD** ◦ if ◦ shArr[32..47] 0 if ◦

```

__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];
    }
}

```

&r[idx & ~(warpSize-1)]r + warpIdx*32 ◦ r32warp ◦

CUDA ◦ 32 - warp ◦ ◦ warp__syncthreads() ◦

```

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 ◦ **warp** ◦ sumCommSingleWarpwarpshArrshArrwarp ◦

shArrvolatile ◦ shArr[idx]shArr ◦ ◦ **volatileconstconst** ◦

shArr[1..31]shArr[32..47]

```

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];
}
```

if ◦ shArr ◦ warpSIMD ◦

◦ CUDAwarp ◦ KeplerCC> = 3.0warp-shuffle ◦

warp ◦ 32

```
__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);
}
```

◦

<https://riptutorial.com/zh-CN/cuda/topic/6566/-->

S. No		Contributors
1	cuda	Community , CygnusX1 , Dev-iL , harrism , havogt , infinite.potential , Jared Hoberock , Ken Y-N , Marco13 , NikolayKondratyev , Tejus Prasad
2		CygnusX1 , tera
3	cuda	Mo Sani
4		CygnusX1