FREE eBook

LEARNING cuda

Free unaffiliated eBook created from
Stack Overflow contributors.

#cuda

Table of Contents

About

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

Chapter 1: Getting started with cuda

Remarks

CUDA is a proprietary NVIDIA parallel computing technology and programming language for their GPUs.

GPUs are highly parallel machines capable of running thousands of lightweight threads in parallel. Each GPU thread is usually slower in execution and their context is smaller. On the other hand, GPU is able to run several thousands of threads in parallel and even more concurrently (precise numbers depend on the actual GPU model). CUDA is a C++ dialect designed specifically for NVIDIA GPU architecture. However, due to the architecture differences, most algorithms cannot be simply copy-pasted from plain C++ - they would run, but would be very slow.

Terminology

- *host* -- refers to normal CPU-based hardware and normal programs that run in that environment
- *device* -- refers to a specific GPU that CUDA programs run in. A single host can support multiple devices.
- *kernel* -- a function that resides on the device that can be invoked from the host code.

Physical Processor Structure

The CUDA-enabled GPU processor has the following physical structure:

- *the chip* - the whole processor of the GPU. Some GPUs have two of them.
- *streamming multiprocessor* (SM) - each chip contains up to ~100 SMs, depending on a model. Each SM operates nearly independently from another, using only global memory to communicate to each other.
- *CUDA core* - a single scalar compute unit of a SM. Their precise number depends on the architecture. Each core can handle a few threads executed concurrently in a quick succession (similar to hyperthreading in CPU).

In addition, each SM features one or more *warp schedulers*. Each scheduler dispatches a single instruction to several CUDA cores. This effectively causes the SM to operate in 32-wide SIMD mode.

CUDA Execution Model

The physical structure of the GPU has direct influence on how kernels are executed on the device, and how one programs them in CUDA. Kernel is invoked with a *call configuration* which specifies how many parallel threads are spawned.

- *the grid* - represents all threads that are spawned upon kernel call. It is specified as a one or two dimentional set of *blocks*
- *the block* - is a semi-independent set of *threads*. Each block is assigned to a single SM. As such, blocks can communicate only through global memory. Blocks are not synchronized in any way. If there are too many blocks, some may execute sequentially after others. On the other hand, if resources permit, more than one block may run on the same SM, but the programmer cannot benefit from that happening (except for the obvious performance boost).
- *the thread* - a scalar sequence of instructions executed by a single CUDA core. Threads are 'lightweight' with minimal context, allowing the hardware to quickly swap them in and out. Because of their number, CUDA threads operate with a few registers assigned to them, and very short stack (preferably none at all!). For that reason, CUDA compiler prefers to inline all function calls to flatten the kernel so that it contains only static jumps and loops. Function ponter calls, and virtual method calls, while supported in most newer devices, usually incur a major performance penality.

Each thread is identified by a block index `blockIdx` and thread index within the block `threadIdx`. These numbers can be checked at any time by any running thread and is the only way of distinguishing one thread from another.

In addition, threads are organized into *warps*, each containing exactly 32 threads. Threads within a single warp execute in a perfect sync, in SIMD fahsion. Threads from different warps, but within the same block can execute in any order, but can be forced to synchronize by the programmer. Threads from different blocks cannot be synchronized or interact directly in any way.

Memory Organisation

In normal CPU programming the memory organization is usually hidden from the programmer. Typical programs act as if there was just RAM. All memory operations, such as managing registers, using L1- L2- L3- caching, swapping to disk, etc. is handled by the compiler, operating system or hardware itself.

This is not the case with CUDA. While newer GPU models partially hide the burden, e.g. through the Unified Memory in CUDA 6, it is still worth understanding the organization for performance reasons. The basic CUDA memory structure is as follows:

- *Host memory* -- the regular RAM. Mostly used by the host code, but newer GPU models may access it as well. When a kernel access the host memory, the GPU must communicate with the motherboard, usually through the PCIe connector and as such it is relatively slow.
- *Device memory / Global memory* -- the main off-chip memory of the GPU, available to all threads.
- *Shared memory* - located in each SM allows for much quicker access than global. Shared memory is private to each block. Threads within a single block can use it for communication.
- *Registers* - fastest, private, unaddressable memory of each thread. In general these cannot be used for communication, but a few intrinsic functions allows to shuffle their contents within a warp.
- *Local memory* - private memory of each thread that *is* addressable. This is used for register spills, and local arrays with variable indexing. Physically, they reside in global memory.

- *Texture memory, Constant memory* - a part of global memory that is marked as immutable for the kernel. This allows the GPU to use special-purpose caches.
- *L2 cache* -- on-chip, available to all threads. Given the amount of threads, the expected lifetime of each cache line is much lower than on CPU. It is mostly used aid misaligned and partially-random memory access patterns.
- *L1 cache* -- located in the same space as shared memory. Again, the amount is rather small, given the number of threads using it, so do not expect data to stay there for long. L1 caching can be disabled.

Versions

Compute Capability	Architecture	GPU Codename	Release Date
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
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	2015-04-01
6.0	Pascal	GP100	2016-10-01
6.1	Pascal	GP102, GP104, GP106	2016-05-27

The release date marks the release of the first GPU supporting given compute capability. Some dates are approximate, e.g. 3.2 card was released in Q2 2014.

Examples

Prerequisites

To get started programming with CUDA, download and install the CUDA Toolkit and developer driver. The toolkit includes `nvcc`, the NVIDIA CUDA Compiler, and other software necessary to develop CUDA applications. The driver ensures that GPU programs run correctly on CUDA-capable hardware, which you'll also need.

You can confirm that the CUDA Toolkit is correctly installed on your machine by running `nvcc --version` from a command line. For example, on a Linux machine,

```
$ 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
```

outputs the compiler information. If the previous command was not successful, then the CUDA Toolkit is likely not installed, or the path to `nvcc` (`C:\CUDA\bin` on Windows machines, `/usr/local/cuda/bin` on POSIX OSes) is not part of your `PATH` environment variable.

Additionally, you'll also need a host compiler which works with `nvcc` to compile and build CUDA programs. On Windows, this is `cl.exe`, the Microsoft compiler, which ships with Microsoft Visual Studio. On POSIX OSes, other compilers are available, including `gcc` or `g++`. The official CUDA Quick Start Guide can tell you which compiler versions are supported on your particular platform.

To make sure everything is set up correctly, let's compile and run a trivial CUDA program to ensure all the tools work together correctly.

```
__global__ void foo() {}

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

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

  return 0;
}
```

To compile this program, copy it to a file called test.cu and compile it from the command line. For example, on a Linux system, the following should work:

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

If the program succeeds without error, then let's start coding!

Sum two arrays with CUDA

This example illustrates how to create a simple program that will sum two `int` arrays with CUDA.

A CUDA program is heterogenous and consist of parts runs both on CPU and GPU.

The main parts of a program that utilize CUDA are similar to CPU programs and consist of

- Memory allocation for data that will be used on GPU
- Data copying from host memory to GPUs memory
- Invoking kernel function to process data
- Copy result to CPUs memory

To allocate devices memory we use `cudaMalloc` function. To copy data between device and host `cudaMemcpy` function can be used. The last argument of `cudaMemcpy` specifies the direction of copy operation. There are 5 possible types:

- `cudaMemcpyHostToHost` - Host -> Host
- `cudaMemcpyHostToDevice` - Host -> Device
- `cudaMemcpyDeviceToHost` - Device -> Host
- `cudaMemcpyDeviceToDevice` - Device -> Device
- `cudaMemcpyDefault` - Default based unified virtual address space

Next the kernel function is invoked. The information between the triple chevrons is the execution configuration, which dictates how many device threads execute the kernel in parallel. The first number (`2` in example) specifies number of blocks and second (`(size + 1) / 2` in example) - number of threads in a block. Note that in this example we add 1 to the size, so that we request one extra thread rather than having one thread responsible for two elements.

Since kernel invocation is an asynchronous function `cudaDeviceSynchronize` is called to wait until execution is completed. Result arrays is copied to the host memory and all memory allocated on the device is freed with `cudaFree`.

To define function as kernel `__global__` declaration specifier is used. This function will be invoked by each thread. If we want each thread to process an element of the resultant array, then we need a means of distinguishing and identifying each thread. CUDA defines the variables `blockDim`, `blockIdx`, and `threadIdx`. The predefined variable `blockDim` contains the dimensions of each thread block as specified in the second execution configuration parameter for the kernel launch. The predefined variables `threadIdx` and `blockIdx` contain the index of the thread within its thread block and the thread block within the grid, respectively. Note that since we potentially request one more thread than elements in the arrays, we need to pass in `size` to ensure we don't access past the end of the array.

```
#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;
}
```

Let's launch a single CUDA thread to say hello

This simple CUDA program demonstrates how to write a function that will execute on the GPU (aka "device"). The CPU, or "host", creates CUDA threads by calling special functions called "kernels". CUDA programs are C++ programs with additional syntax.

To see how it works, put the following code in a file named `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;
}
```

(Note that in order to use the `printf` function on the device, you need a device that has a compute capability of at least 2.0. See the versions overview for details.)

Now let's compile the program using the NVIDIA compiler and run it:

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

Some additional information about the above example:

- `nvcc` stands for "NVIDIA CUDA Compiler". It separates source code into host and device components.
- `__global__` is a CUDA keyword used in function declarations indicating that the function runs on the GPU device and is called from the host.
- Triple angle brackets (`<<<,>>>`) mark a call from host code to device code (also called "kernel launch"). The numbers within these triple brackets indicate the number of times to execute in parallel and the number of threads.

Compiling and Running the Sample Programs

The NVIDIA installation guide ends with running the sample programs to verify your installation of the CUDA Toolkit, but doesn't explicitly state how. First check all the prerequisites. Check the default CUDA directory for the sample programs. If it is not present, it can be downloaded from the official CUDA website. Navigate to the directory where the examples are present.

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

You should see an output similar to:

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

Ensure that the `Makefile` is present in this directory. The `make` command in UNIX based systems will build all the sample programs. Alternatively, navigate to a subdirectory where another `Makefile` is present and run the `make` command from there to build only that sample.

Run the two suggested sample programs - `deviceQuery` and `bandwidthTest`:

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

The output will be similar to the one shown below:

```
./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
```

The statement `Result = PASS` at the end indicates that everything is working correctly. Now, run the other suggested sample program `bandwidthTest` in a similar fashion. The output will be similar to:

```
[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.
```

Again, the `Result = PASS` statement indicates that everything was executed properly. All other sample programs can be run in a similar fashion.

Read Getting started with cuda online: https://riptutorial.com/cuda/topic/1860/getting-started-with-cuda

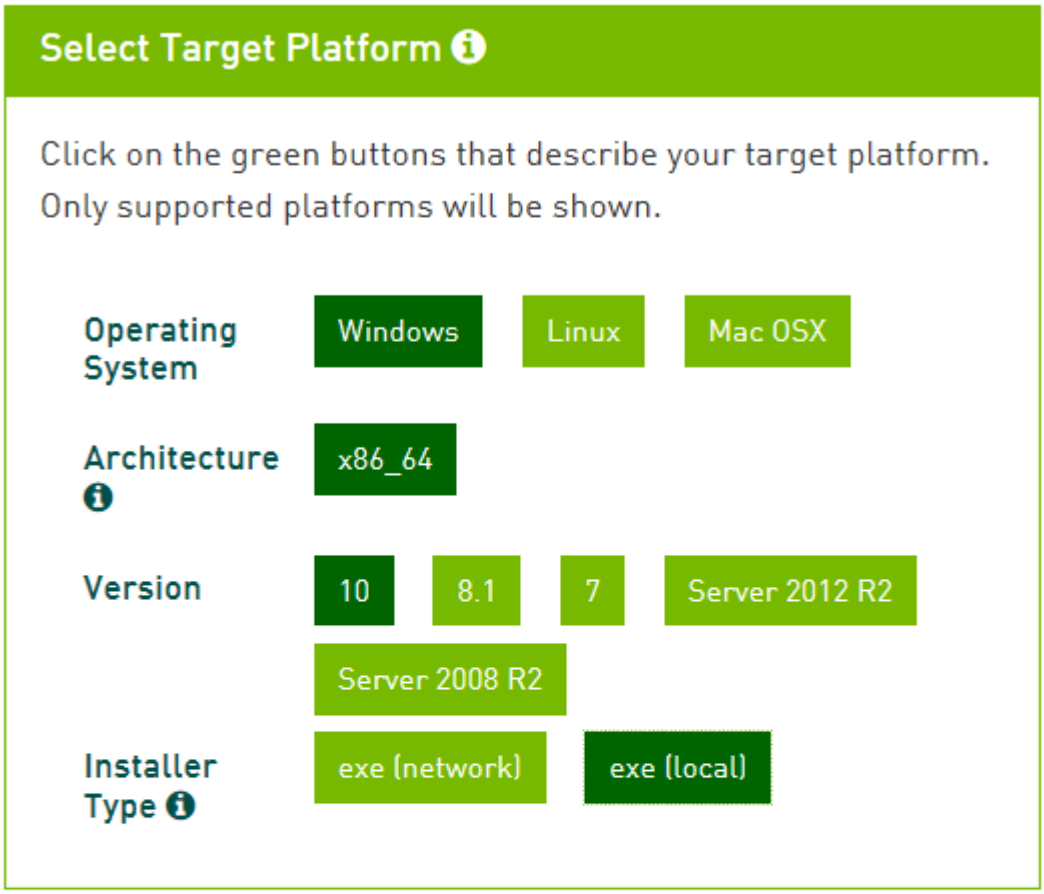
Chapter 2: Installing cuda

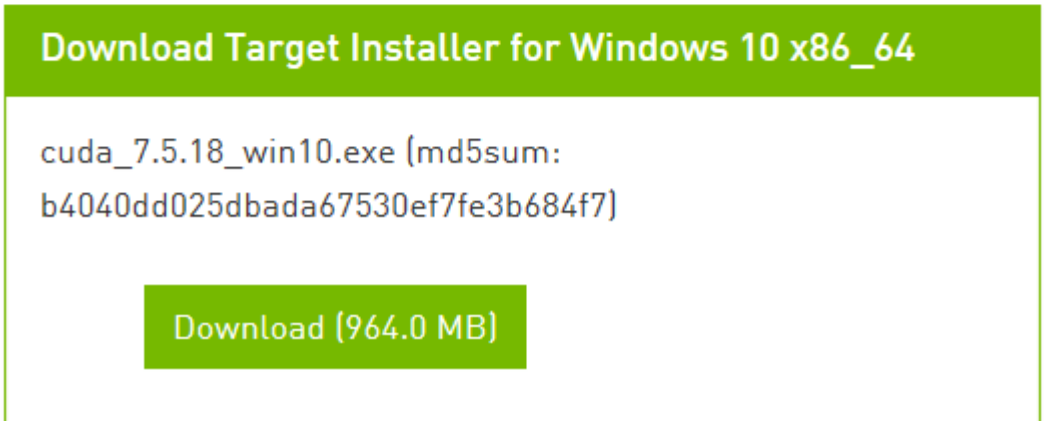
Remarks

To install CUDA toolkit on Windows, fist you need to install a proper version of Visual Studio. Visual studio 2013 should be installed if you're going to install CUDA 7.0 or 7.5. Visual Studio 2015 is supported for CUDA 8.0 and beyond.

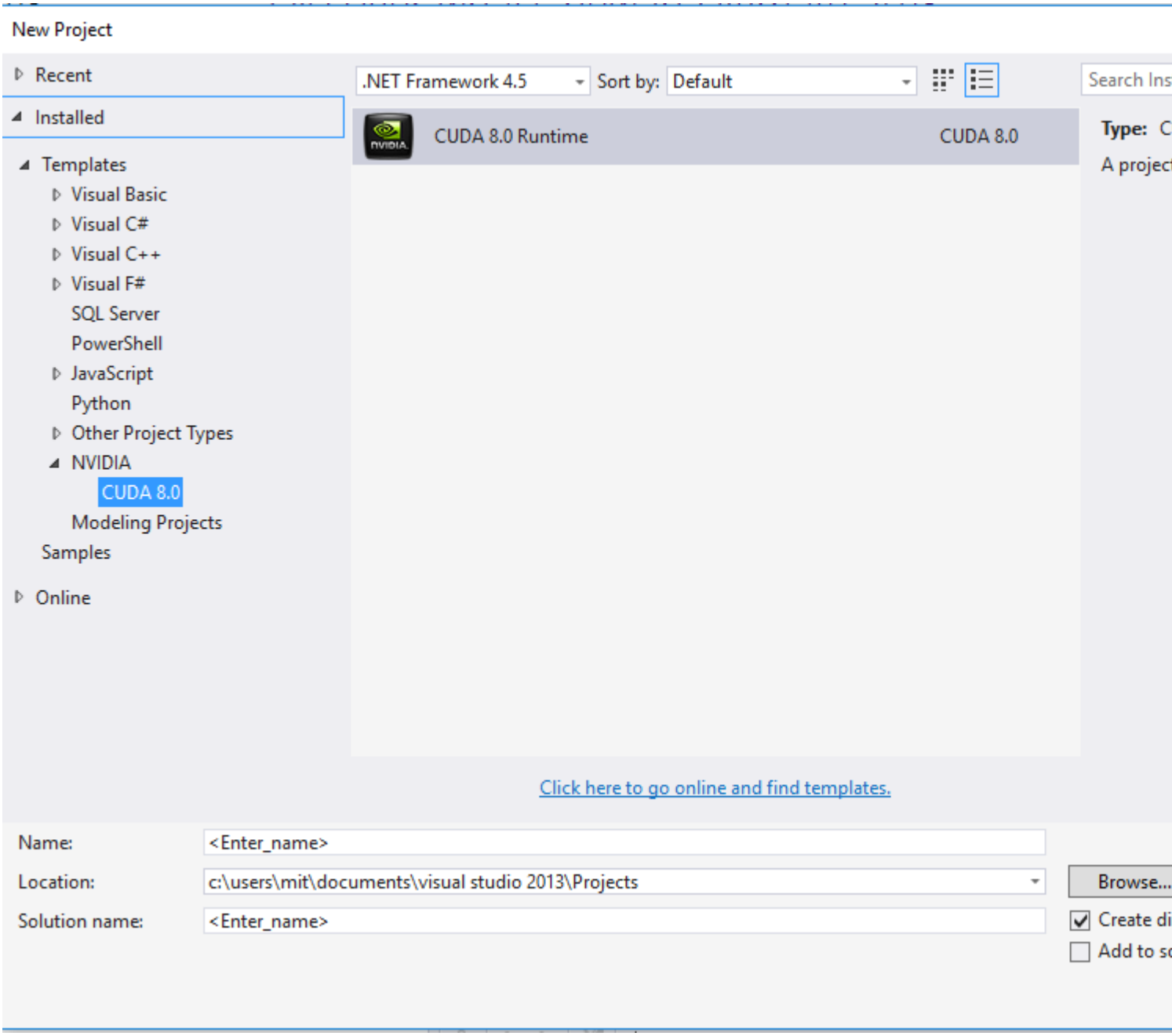
When you've a proper version of VS on your system, it's time to download and install CUDA toolkit. Follow this link to find a the version of CUDA toolkit you're looking for: CUDA toolkit archive

In the download page you should choose the version of windows on target machine, and installer type (choose local).





After downloading exe file, you shall extract it and run `setup.exe`. When installation is complete open a new project and choose NVIDIA>CUDAX.X from templates.



Remember that CUDA source files extension is `.cu`. You can write both host and device codes on a same source.

Examples

Very simple 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;
}
```

Read Installing cuda online: https://riptutorial.com/cuda/topic/10949/installing-cuda

Chapter 3: Inter-block communication

Remarks

Blocks in CUDA operate semi-independently. There is no safe way to synchronize them all. However, it does not mean that they cannot interact with each other in any way.

Examples

Last-block guard

Consider a grid working on some task, e.g. a parallel reduction. Initially, each block can do its work independently, producing some partial result. At the end however, the partial results need to be combined and merged together. A typical example is a reduction algorithm on a big data.

A typical approach is to invoke two kernels, one for the partial computation and the other for merging. However, if the merging can be done efficiently by a single block, only one kernel call is required. This is achieved by a `lastBlock` guard defined as:

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;
}
```

With such a guard the last block is guaranteed to see all the results produced by all other blocks and can perform the merging.

```
__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);
    }
}
```

Assumptions:

- The counter must be a global memory pointer, initialized to 0 *before* the kernel is invoked.
- The `lastBlock` function is invoked uniformly by all threads in all blocks
- The kernel is invoked in one-dimentional grid (for simplicity of the example)
- `T` names any type you like, but the example is not intended to be a template in C++ sense

Global work queue

Consider an array of work items. The time needed for an each work item to complete varies greatly. In order to balance the work distribution between blocks it may be prudent for each block to fetch the next item only when previous one is complete. This is in contrast to a-priori assigning items to blocks.

```
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
  }
}
```

Assumptions:

- The WorkQueue object, as well as gItem array reside in global memory
- No new work items are added to the WorkQueue object in the kernel that is fetching from it
- The `WorkItem` is a small representation of the work assignment, e.g. a pointer to another object
- `WorkItem::none()` static member function creates a `WorkItem` object that represents no work at all
- `WorkQueue::fetch()` must be called uniformly by all threads in the block
- There are no 2 invocations of `WorkQueue::fetch()` without another `__syncthreads()` in between. Otherwise a race condition will appear!

The example does not include how the initialize the `WorkQueue` or populate it. It is done by another

kernel or CPU code and should be pretty straight-forward.

Read Inter-block communication online: https://riptutorial.com/cuda/topic/4978/inter-block-communication

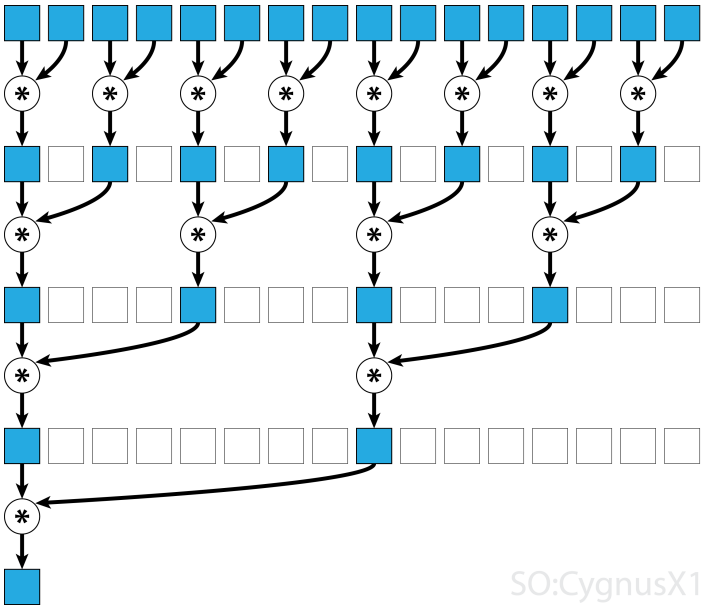
Chapter 4: Parallel reduction (e.g. how to sum an array)

Remarks

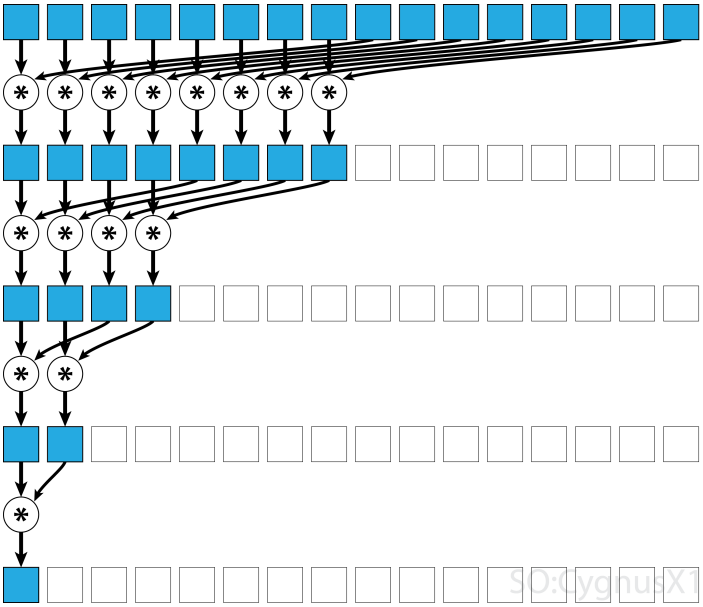
Parallel reduction algorithm typically refers to an algorithm which combines an array of elements, producing a single result. Typical problems that fall into this category are:

- summing up all elements in an array
- finding a maximum in an array

In general, the parallel reduction can be applied for any binary associative operator, i.e. `(A*B)*C = A*(B*C)`. With such operator *, the parallel reduction algorithm repetedely groups the array arguments in pairs. Each pair is computed in parallel with others, halving the overall array size in one step. The process is repeated until only a single element exists.



If the operator is commutative (i.e. `A*B = B*A`) in addition to being associative, the algorithm can pair in a different pattern. From theoretical standpoint it makes no difference, but in practice it gives a better memory access pattern:



Not all associative operators are commutative - take matrix multiplication for example.

Examples

Single-block parallel reduction for commutative operator

The simplest approach to parallel reduction in CUDA is to assign a single block to perform the task:

```
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);
```

This is most feasable when the data size is not very large (around a few thousants elements). This usually happens when the reduction is a part of some bigger CUDA program. If the input matches `blockSize` from the very beginning, the first `for` loop can be completely removed.

Note that in first step, when there are more elements than threads, we add things up completely

independently. Only when the problem is reduced to `blockSize`, the actual parallel reduction triggers. The same code can be applied to any other commutative, associative operator, such as multiplication, minimum, maximum, etc.

Note that the algorithm can be made faster, for example by using a warp-level parallel reduction.

Single-block parallel reduction for non-commutative operator

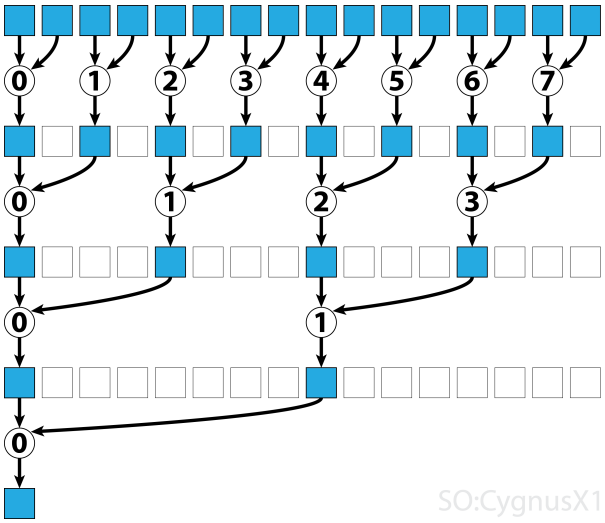
Doing parallel reduction for a non-commutative operator is a bit more involved, compared to commutative version. In the example we still use a addition over integers for the simplicity sake. It could be replaced, for example, with matrix multiplication which really is non-commutative. Note, when doing so, 0 should be replaced by a neutral element of the multiplication - i.e. an identity matrix.

```
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 the first while loop executes as long as there are more input elements than threads. In each iteration, a single reduction is performed and the result is compressed into the first half of the `shArr` array. The second half is then filled with new data.

Once all data is loaded from `gArr`, the second loop executes. Now, we no longer compress the result (which costs an extra `__syncthreads()`). In each step the thread n access the `2*n`-th active

element and adds it up with `2*n+1`-th element:



There are many ways to further optimize this simple example, e.g. through warp-level reduction and by removing shared memory bank conflicts.

Multi-block parallel reduction for commutative operator

Multi-block approach to parallel reduction in CUDA poses an additional challenge, compared to single-block approach, because blocks are limited in communication. The idea is to let each block compute a part of the input array, and then have one final block to merge all the partial results. To do that one can launch two kernels, implicitly creating a grid-wide synchronization point.

```
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;
}
```

One ideally wants to launch enough blocks to saturate all multiprocessors on the GPU at full occupancy. Exceeding this number - in particular, launching as many threads as there are elements in the array - is counter-productive. Doing so it does not increase the raw compute power anymore, but prevents using the very efficient first loop.

It is also possible to obtain the same result using a single kernel, with the help of 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 == 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;
}
```

Note that the kernel can be made faster, for example by using a warp-level parallel reduction.

Multi-block parallel reduction for noncommutative operator

Multi-block approach to parallel reduction is very similar to the single-block approach. The global input array must be split into sections, each reduced by a single block. When a partial result from each block is obtained, one final block reduces these to obtain the final result.

- `sumNoncommSingleBlock` is explained more in detail in the Single-block reduction example.
- `lastBlock` accepts only the last block reaching it. If you want to avoid this, you can split the kernel into two separate calls.

```
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);
}
```

One ideally wants to launch enough blocks to saturate all multiprocessors on the GPU at full occupancy. Exceeding this number - in particular, launching as many threads as there are elements in the array - is counter-productive. Doing so it does not increase the raw compute power anymore, but prevents using the very efficient first loop.

Single-warp parallel reduction for commutative operator

Sometimes the reduction has to be performed on a very small scale, as a part of a bigger CUDA kernel. Suppose for example, that the input data has exactly 32 elements - the number of threads in a warp. In such scenario a single warp can be assigned to perform the reduction. Given that warp executes in a perfect sync, many `__syncthreads()` instructions can be removed - when compared to a block-level reduction.

```
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` is preferably an array in shared memory. The value should be the same for all threads in the warp. If `sumCommSingleWarp` is called by multiple warps, `shArr` should be different between warps (same within each warp).

The argument `shArr` is marked as `volatile` to ensure that operations on the array are actually performed where indicated. Otherwise, the repetetive assignment to `shArr[idx]` may be optimized as an assignment to a register, with only final assigment being an actual store to `shArr`. When that happens, the immediate assignments are not visible to other threads, yielding incorrect results. Note, that you can pass a normal non-volatile array as an argument of volatile one, same as when you pass non-const as a const parameter.

If one does not care about the contents of `shArr[1..31]` after the reduction, one can simplify the code even further:

```
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 this setup we removed many `if` conditions. The extra threads perform some unnecessary additions, but we no longer care about the contents they produce. Since warps execute in SIMD mode we do not actually save on time by having those threads doing nothing. On the other hand, evaluating the conditions does take relatively big amount of time, since the body of these `if` statements are so small. The initial `if` statement can be removed as well if `shArr[32..47]` are padded with 0.

The warp-level reduction can be used to boost the block-level reduction as well:

```
__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];
    }
}
```

The argument `&r[idx & ~(warpSize-1)]` is basically `r + warpIdx*32`. This effectively splits the `r` array into chunks of 32 elements, and each chunk is assigned to separate warp.

Single-warp parallel reduction for noncommutative operator

Sometimes the reduction has to be performed on a very small scale, as a part of a bigger CUDA kernel. Suppose for example, that the input data has exactly 32 elements - the number of threads in a warp. In such scenario a single warp can be assigned to perform the reduction. Given that warp executes in a perfect sync, many `__syncthreads()` instructions can be removed - when compared to a block-level reduction.

```
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` is preferably an array in shared memory. The value should be the same for all threads in the warp. If `sumCommSingleWarp` is called by multiple warps, `shArr` should be different between warps (same within each warp).

The argument `shArr` is marked as `volatile` to ensure that operations on the array are actually performed where indicated. Otherwise, the repetetive assignment to `shArr[idx]` may be optimized as an assignment to a register, with only final assigment being an actual store to `shArr`. When that happens, the immediate assignments are not visible to other threads, yielding incorrect results. Note, that you can pass a normal non-volatile array as an argument of volatile one, same as when you pass non-const as a const parameter.

If one does not care about the final contents of `shArr[1..31]` and can pad `shArr[32..47]` with zeros, one can simplify the above code:

```
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 this setup we removed all `if` conditions, which constitute about the half of the instructions. The extra threads perform some unnecessary additions, storing the result into cells of `shArr` that ultimately have no impact on the final result. Since warps execute in SIMD mode we do not actually save on time by having those threads doing nothing.

Single-warp parallel reduction using registers only

Typically, reduction is performed on global or shared array. However, when the reduction is performed on a very small scale, as a part of a bigger CUDA kernel, it can be performed with a single warp. When that happens, on Keppler or higher architectures (CC>=3.0), it is possible to use warp-shuffle functions to avoid using shared memory at all.

Suppose for example, that each thread in a warp holds a single input data value. All threads together have 32 elements, that we need to sum up (or perform other associative operation)

```
__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);
}
```

This version works for both commutative and non-commutative operators.

Read Parallel reduction (e.g. how to sum an array) online:
https://riptutorial.com/cuda/topic/6566/parallel-reduction--e-g--how-to-sum-an-array-

Credits

S. No	Chapters	Contributors
1	Getting started with cuda	Community, CygnusX1, Dev-iL, harrism, havogt, infinite.potential, Jared Hoberock, Ken Y-N, Marco13, NikolayKondratyev, Tejus Prasad
2	Installing cuda	Mo Sani
3	Inter-block communication	CygnusX1, tera
4	Parallel reduction (e.g. how to sum an array)	CygnusX1