EBook Gratis

APRENDIZAJE cuda

Free unaffiliated eBook created from
Stack Overflow contributors.

#cuda

Tabla de contenido

Acerca de 1

Capítulo 1: Empezando con cuda 2

Observaciones 2

Terminología 2

Estructura del procesador físico 2

Modelo de Ejecución CUDA 2

Organización de la memoria 3

Versiones 4

Examples 5

Prerrequisitos 5

Suma dos matrices con CUDA 6

Vamos a lanzar un solo hilo CUDA para saludar 8

Compilando y ejecutando los programas de muestra 9

Capítulo 2: Comunicación interbloque 12

Observaciones 12

Examples 12

Guardia de ultima cuadra 12

Cola de trabajo global 13

Capítulo 3: Instalando cuda 15

Observaciones 15

Examples 17

Código CUDA muy simple 17

Capítulo 4: Reducción paralela (por ejemplo, cómo sumar una matriz) 19

Observaciones 19

Examples 20

Reducción en paralelo de bloque único para operador conmutativo 20

Reducción en paralelo de bloque único para operador no conmutativo 21

Reducción paralela multibloque para operador conmutativo 22

Reducción paralela multibloque para operador no conmutativo 24

Reducción paralela de una sola urdimbre para operador conmutativo 25

Reducción paralela de una sola urdimbre para un operador no conmutativo. 27
Reducción paralela de una sola urdimbre usando solo registros 28
Creditos 29

Acerca de

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

Capítulo 1: Empezando con cuda

Observaciones

CUDA es una tecnología de computación paralela de NVIDIA y un lenguaje de programación para sus GPU.

Las GPU son máquinas altamente paralelas capaces de ejecutar miles de subprocesos ligeros en paralelo. Cada subproceso de GPU suele ser más lento en ejecución y su contexto es más pequeño. Por otro lado, GPU puede ejecutar varios miles de subprocesos en paralelo e incluso más concurrentemente (los números precisos dependen del modelo de GPU real). CUDA es un dialecto de C ++ diseñado específicamente para la arquitectura de GPU NVIDIA. Sin embargo, debido a las diferencias de arquitectura, la mayoría de los algoritmos no se pueden copiar y pegar simplemente desde C ++ simple; se ejecutarían, pero serían muy lentos.

Terminología

- *host* : se refiere al hardware normal basado en CPU y a los programas normales que se ejecutan en ese entorno
- *dispositivo* : se refiere a una GPU específica en la que se ejecutan los programas CUDA. Un solo host puede admitir múltiples dispositivos.
- *kernel* : una función que reside en el dispositivo que se puede invocar desde el código del host.

Estructura del procesador físico

El procesador de GPU habilitado para CUDA tiene la siguiente estructura física:

- *El chip* - todo el procesador de la GPU. Algunas GPU tienen dos de ellas.
- *multiprocesador de flujo continuo* (SM): cada chip contiene hasta ~ 100 SM, según el modelo. Cada SM funciona de forma casi independiente, utilizando solo memoria global para comunicarse entre sí.
- *CUDA core* : una única unidad de cálculo escalar de un SM. Su número preciso depende de la arquitectura. Cada núcleo puede manejar unos pocos subprocesos ejecutados simultáneamente en una sucesión rápida (similar a un hipervínculo en la CPU).

Además, cada SM cuenta con uno o más *programadores warp* . Cada programador envía una única instrucción a varios núcleos CUDA. Esto hace que el SM funcione de manera efectiva en el modo SIMD de 32 anchos.

Modelo de Ejecución CUDA

La estructura física de la GPU tiene una influencia directa sobre cómo se ejecutan los núcleos en el dispositivo y cómo uno los programa en CUDA. El kernel se invoca con una *configuración de*

llamada que especifica cuántos subprocesos paralelos se generan.

- *la cuadrícula* : representa todos los subprocesos que se generan a partir de la llamada al kernel. Se especifica como un conjunto de *bloques* de una o dos dimensiones.
- *El bloque* - es un conjunto semi-independiente de *hilos* . Cada bloque se asigna a un solo SM. Como tal, los bloques solo pueden comunicarse a través de la memoria global. Los bloques no están sincronizados de ninguna manera. Si hay demasiados bloques, algunos pueden ejecutarse secuencialmente después de otros. Por otro lado, si los recursos lo permiten, más de un bloque puede ejecutarse en el mismo SM, pero el programador no puede beneficiarse de que eso suceda (excepto por el aumento de rendimiento obvio).
- *el hilo* : una secuencia escalar de instrucciones ejecutadas por un solo núcleo CUDA. Los hilos son 'ligeros' con un contexto mínimo, lo que permite que el hardware los intercambie rápidamente hacia adentro y hacia afuera. Debido a su número, los hilos CUDA operan con unos pocos registros asignados a ellos, y una pila muy corta (¡preferiblemente ninguno!). Por esa razón, el compilador CUDA prefiere alinear todas las llamadas de función para aplanar el kernel de modo que contenga solo saltos y bucles estáticos. Las llamadas de ponter de función y las llamadas de método virtual, aunque son compatibles con la mayoría de los dispositivos más nuevos, generalmente incurren en una mayor penalidad de rendimiento.

Cada subproceso se identifica mediante un índice de bloque `blockIdx` e índice de subproceso dentro del bloque `threadIdx` . Estos números pueden ser verificados en cualquier momento por cualquier hilo en ejecución y es la única forma de distinguir un hilo de otro.

Además, los hilos se organizan en *urdimbres* , cada uno de los cuales contiene exactamente 32 hilos. Los hilos dentro de una sola urdimbre se ejecutan en una sincronización perfecta, en la versión SIMD. Los hilos de diferentes deformaciones, pero dentro del mismo bloque pueden ejecutarse en cualquier orden, pero el programador puede obligarlos a sincronizar. Los hilos de diferentes bloques no se pueden sincronizar o interactuar directamente de ninguna manera.

Organización de la memoria

En la programación normal de la CPU, la organización de la memoria suele estar oculta al programador. Los programas típicos actúan como si solo hubiera RAM. Todas las operaciones de la memoria, como la gestión de registros, el uso del almacenamiento en caché L1-L2-L3, el intercambio en el disco, etc., se realizan mediante el compilador, el sistema operativo o el hardware.

Este no es el caso con CUDA. Si bien los modelos de GPU más recientes ocultan parcialmente la carga, por ejemplo, a través de la Memoria unificada en CUDA 6, todavía vale la pena entender a la organización por razones de rendimiento. La estructura básica de la memoria CUDA es la siguiente:

- *Memoria de host* - la memoria RAM regular. Principalmente utilizado por el código del host, pero los modelos más nuevos de GPU también pueden acceder a él. Cuando un kernel accede a la memoria del host, la GPU debe comunicarse con la placa base, generalmente a través del conector PCIe y, como tal, es relativamente lento.

- *Memoria del dispositivo* / *Memoria global: la memoria* principal fuera de chip de la GPU, disponible para todos los subprocesos.
- *Memoria compartida* : ubicada en cada SM permite un acceso mucho más rápido que el global. La memoria compartida es privada para cada bloque. Los hilos dentro de un solo bloque pueden usarlo para la comunicación.
- *Registros* : la memoria más rápida, privada y no direccionable de cada hilo. En general, no se pueden usar para la comunicación, pero algunas funciones intrínsecas permiten barajar su contenido dentro de una deformación.
- *Memoria local* - *memoria* privada de cada hilo que *es* direccionable. Esto se usa para registros de derrames y arreglos locales con indexación variable. Físicamente, residen en la memoria global.
- *Memoria de textura, Memoria constante* : una parte de la memoria global que está marcada como inmutable para el kernel. Esto permite que la GPU use cachés de propósito especial.
- *L2 cache* - on-chip, disponible para todos los hilos. Dada la cantidad de subprocesos, la vida útil esperada de cada línea de caché es mucho menor que en la CPU. Se utiliza principalmente para ayudar a los patrones de acceso de memoria desalineados y parcialmente aleatorios.
- *Caché L1* : ubicado en el mismo espacio que la memoria compartida. Nuevamente, la cantidad es bastante pequeña, dado el número de subprocesos que la utilizan, por lo que no espere que los datos permanezcan allí por mucho tiempo. El almacenamiento en caché de L1 se puede deshabilitar.

Versiones

Capacidad de cálculo	Arquitectura	Nombre en clave de GPU	Fecha de lanzamiento
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

Capacidad de cálculo	Arquitectura	Nombre en clave de GPU	Fecha de lanzamiento
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

La fecha de lanzamiento marca el lanzamiento de la primera GPU que admite la capacidad de cálculo dada. Algunas fechas son aproximadas, por ejemplo, la tarjeta 3.2 se lanzó en el segundo trimestre de 2014.

Examples

Prerrequisitos

Para comenzar a programar con CUDA, descargue e instale el kit de herramientas CUDA y el controlador del desarrollador . El kit de herramientas incluye `nvcc` , el compilador NVIDIA CUDA y otro software necesario para desarrollar aplicaciones CUDA. El controlador garantiza que los programas de GPU se ejecuten correctamente en un hardware compatible con CUDA , que también necesitará.

Puede confirmar que el kit de herramientas CUDA está instalado correctamente en su máquina ejecutando `nvcc --version` desde una línea de comandos. Por ejemplo, en una máquina 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
```

genera la información del compilador. Si el comando anterior no tuvo éxito, es probable que el kit de herramientas CUDA no esté instalado, o que la ruta a `nvcc` (`C:\CUDA\bin` en máquinas Windows, `/usr/local/cuda/bin` en sistemas operativos POSIX) no sea parte de su Variable de entorno `PATH` .

Además, también necesitará un compilador host que funcione con `nvcc` para compilar y crear programas CUDA. En Windows, este es `cl.exe` , el compilador de Microsoft, que se incluye con Microsoft Visual Studio. En los sistemas operativos POSIX, hay otros compiladores disponibles, incluidos `gcc` o `g++` . La Guía de inicio rápido oficial de CUDA puede decirle qué versiones de compilador son compatibles con su plataforma en particular.

Para asegurarnos de que todo esté configurado correctamente, compilemos y ejecutemos un programa trivial de CUDA para asegurarnos de que todas las herramientas funcionen juntas

correctamente.

```
__global__ void foo() {}

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

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

  return 0;
}
```

Para compilar este programa, cópielo en un archivo llamado test.cu y compílelo desde la línea de comandos. Por ejemplo, en un sistema Linux, lo siguiente debería funcionar:

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

Si el programa tiene éxito sin error, ¡entonces comencemos a codificar!

Suma dos matrices con CUDA.

Este ejemplo ilustra cómo crear un programa simple que sumará dos arrays `int` con CUDA.

Un programa CUDA es heterogéneo y consta de partes que se ejecutan tanto en la CPU como en la GPU.

Las partes principales de un programa que utiliza CUDA son similares a los programas de CPU y consisten en

- Asignación de memoria para los datos que se utilizarán en la GPU
- Copia de datos desde la memoria del host a la memoria de las GPU
- Invocando la función del kernel para procesar datos
- Copie el resultado a la memoria de la CPU

Para asignar memoria a los dispositivos `cudaMalloc` función `cudaMalloc` . Para copiar datos entre el dispositivo y el host se puede utilizar la función `cudaMemcpy` . El último argumento de `cudaMemcpy` especifica la dirección de la operación de copia. Hay 5 tipos posibles:

- `cudaMemcpyHostToHost` - Host -> Host
- `cudaMemcpyHostToDevice` - Host -> Dispositivo
- `cudaMemcpyDeviceToHost` - Dispositivo -> Host
- `cudaMemcpyDeviceToDevice` - Dispositivo -> Dispositivo
- `cudaMemcpyDefault` : espacio de direcciones virtuales unificadas basadas en `cudaMemcpyDefault` predeterminados

A continuación se invoca la función del núcleo. La información entre los chevrones triples es la configuración de ejecución, que determina cuántos subprocesos de dispositivo ejecutan el kernel

en paralelo. El primer número (`2` en el ejemplo) especifica el número de bloques y el segundo (`(size + 1) / 2` en el ejemplo) - número de hilos en un bloque. Tenga en cuenta que en este ejemplo agregamos 1 al tamaño, de modo que solicitamos un hilo adicional en lugar de tener un hilo responsable de dos elementos.

Dado que la invocación del kernel es una función asíncrona, se llama a `cudaDeviceSynchronize` para esperar hasta que se complete la ejecución. Las matrices de resultados se copian en la memoria del host y toda la memoria asignada en el dispositivo se libera con `cudaFree` .

Para definir la función como kernel `__global__` se usa el especificador de declaración. Esta función será invocada por cada hilo. Si queremos que cada subproceso procese un elemento de la matriz resultante, entonces necesitamos un medio para distinguir e identificar cada subproceso. CUDA define las variables `blockDim` , `blockIdx` y `threadIdx` . La variable predefinida `blockDim` contiene las dimensiones de cada bloque de subprocesos como se especifica en el segundo parámetro de configuración de ejecución para el lanzamiento del kernel. Las variables predefinidas `threadIdx` y `blockIdx` contienen el índice del hilo dentro de su bloque de hilo y el bloque del hilo dentro de la cuadrícula, respectivamente. Tenga en cuenta que dado que potencialmente solicitamos un subproceso más que los elementos en las matrices, debemos pasar de `size` para asegurarnos de que no accedamos más allá del final de la matriz.

```
#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;
}
```

Vamos a lanzar un solo hilo CUDA para saludar

Este sencillo programa CUDA demuestra cómo escribir una función que se ejecutará en la GPU (también conocido como "dispositivo"). La CPU, o "host", crea subprocesos CUDA llamando a funciones especiales llamadas "kernels". Los programas CUDA son programas C ++ con sintaxis adicional.

Para ver cómo funciona, coloque el siguiente código en un archivo llamado `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;
}
```

(Tenga en cuenta que para utilizar la función `printf` en el dispositivo, necesita un dispositivo que

tenga una capacidad de cómputo de al menos 2.0. Consulte la descripción general de las versiones para obtener más información).

Ahora compilemos el programa usando el compilador NVIDIA y ejecutémoslo:

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

Alguna información adicional sobre el ejemplo anterior:

- `nvcc` significa "Compilador NVIDIA CUDA". Separa el código fuente en componentes de host y dispositivo.
- `__global__` es una palabra clave CUDA utilizada en las declaraciones de funciones que indica que la función se ejecuta en el dispositivo GPU y se llama desde el host.
- Los corchetes angulares triples (`<<<` , `>>>`) marcan una llamada desde el código del host al código del dispositivo (también llamado "lanzamiento del kernel"). Los números entre estos tres paréntesis indican el número de veces que se ejecutan en paralelo y el número de subprocesos.

Compilando y ejecutando los programas de muestra

La guía de instalación de NVIDIA termina con la ejecución de los programas de ejemplo para verificar su instalación del kit de herramientas de CUDA, pero no indica explícitamente cómo. En primer lugar, compruebe todos los requisitos previos. Compruebe el directorio predeterminado de CUDA para los programas de muestra. Si no está presente, se puede descargar desde el sitio web oficial de CUDA. Navegue hasta el directorio donde están presentes los ejemplos.

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

Debería ver una salida similar a:

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

Asegúrese de que el `Makefile` esté presente en este directorio. El comando `make` en los sistemas basados en UNIX construirá todos los programas de ejemplo. Alternativamente, navegue a un subdirectorio donde esté presente otro `Makefile` y ejecute el comando `make` desde allí para compilar solo esa muestra.

Ejecute los dos programas de ejemplo sugeridos: `deviceQuery` y `bandwidthTest` :

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

La salida será similar a la que se muestra a continuación:

```
./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
```

La instrucción `Result = PASS` al final indica que todo funciona correctamente. Ahora, ejecute la otra prueba de `bandwidthTest` de `bandwidthTest` programa de muestra sugerida de una manera similar. La salida será similar a:

```
[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.
```

Nuevamente, la instrucción `Result = PASS` indica que todo se ejecutó correctamente. Todos los demás programas de ejemplo se pueden ejecutar de una manera similar.

Lea Empezando con cuda en línea: https://riptutorial.com/es/cuda/topic/1860/empezando-con-cuda

Capítulo 2: Comunicación interbloque

Observaciones

Los bloques en CUDA operan semi-independientemente. No hay forma segura de sincronizarlos todos. Sin embargo, esto no significa que no puedan interactuar entre sí de ninguna manera.

Examples

Guardia de ultima cuadra

Considere una cuadrícula trabajando en alguna tarea, por ejemplo, una reducción paralela. Inicialmente, cada bloque puede hacer su trabajo de manera independiente, produciendo un resultado parcial. Sin embargo, al final, los resultados parciales deben combinarse y fusionarse. Un ejemplo típico es un algoritmo de reducción en un big data.

Un enfoque típico es invocar dos núcleos, uno para el cálculo parcial y el otro para la fusión. Sin embargo, si la fusión se puede realizar de manera eficiente mediante un solo bloque, solo se requiere una llamada al kernel. Esto se logra mediante una protección de `lastBlock` definida como:

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;
}
```

Con tal protección, se garantiza que el último bloque vea todos los resultados producidos por todos los demás bloques y puede realizar la fusión.

```
__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);
    }
}
```

Suposiciones

- El contador debe ser un puntero de memoria global, inicializado a 0 *antes de* invocar el kernel.
- La función `lastBlock` es invocada uniformemente por todos los hilos en todos los bloques
- El kernel se invoca en una cuadrícula unidimensional (para simplificar el ejemplo)
- `T` nombra cualquier tipo que te guste, pero el ejemplo no pretende ser una plantilla en el sentido de C ++

Cola de trabajo global

Considere una serie de elementos de trabajo. El tiempo necesario para completar cada elemento de trabajo varía mucho. Para equilibrar la distribución de trabajo entre bloques, puede ser prudente que cada bloque obtenga el siguiente elemento solo cuando el anterior esté completo. Esto contrasta con la asignación a priori de elementos a bloques.

```
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
  }
}
```

Suposiciones

- El objeto WorkQueue, así como la matriz gItem, residen en la memoria global
- No se agregan nuevos elementos de trabajo al objeto WorkQueue en el kernel que se está recuperando
- `WorkItem` es una pequeña representación de la asignación de trabajo, por ejemplo, un puntero a otro objeto
- `WorkItem::none()` función miembro estática `WorkItem::none()` crea un objeto `WorkItem` que no representa ningún trabajo
- `WorkQueue::fetch()` debe llamarse de manera uniforme por todos los subprocesos del bloque

- No hay 2 invocaciones de `WorkQueue::fetch()` sin otras `__syncthreads()` entre ellas. De lo contrario aparecerá una condición de carrera!

El ejemplo no incluye cómo inicializar el `WorkQueue` o rellenarlo. Lo hace otro kernel o código de CPU y debería ser bastante sencillo.

Lea Comunicación interbloque en línea: https://riptutorial.com/es/cuda/topic/4978/comunicacion-interbloque

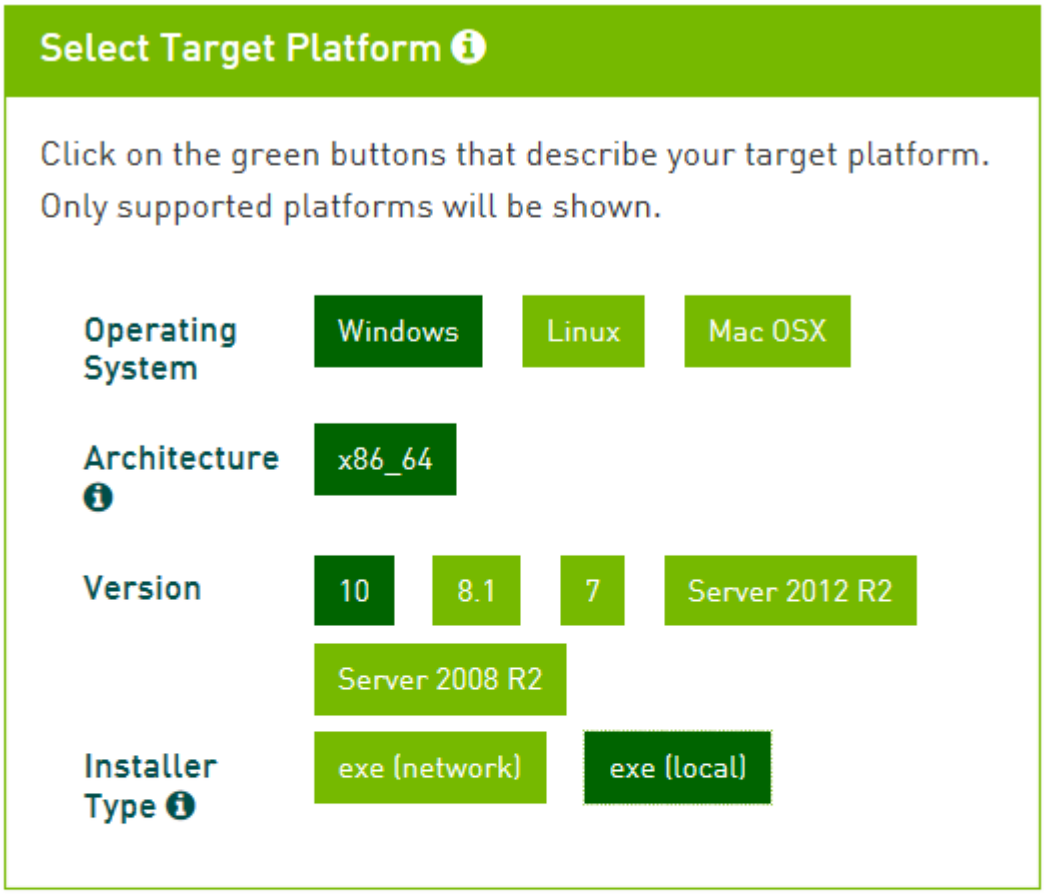
Capítulo 3: Instalando cuda

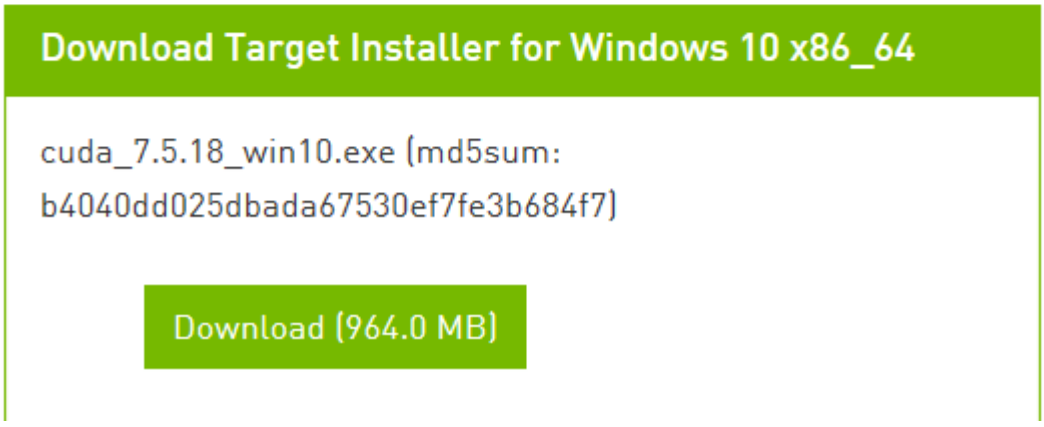
Observaciones

Para instalar el kit de herramientas CUDA en Windows, primero debe instalar una versión adecuada de Visual Studio. Visual Studio 2013 debe instalarse si va a instalar CUDA 7.0 o 7.5. Visual Studio 2015 es compatible con CUDA 8.0 y más allá.

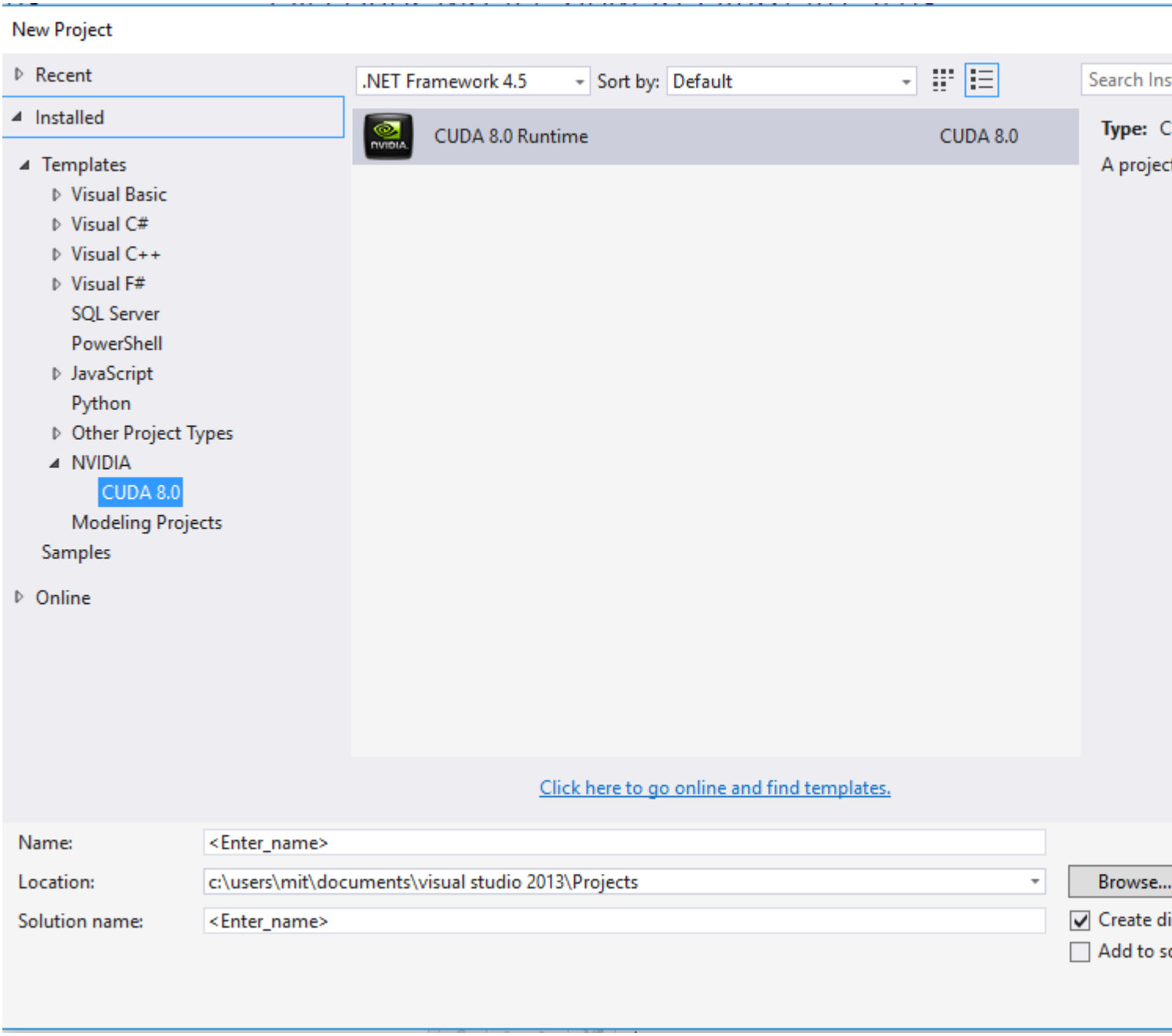
Cuando tenga una versión correcta de VS en su sistema, es hora de descargar e instalar el kit de herramientas CUDA. Siga este enlace para encontrar la versión del kit de herramientas CUDA que está buscando: archivo del kit de herramientas CUDA

En la página de descarga, debe elegir la versión de Windows en la máquina de destino y el tipo de instalador (elija local).





Después de descargar el archivo exe, debe extraerlo y ejecutar `setup.exe` . Cuando finalice la instalación, abra un nuevo proyecto y elija NVIDIA> CUDAX.X en las plantillas.



Recuerde que la extensión de los archivos fuente CUDA es `.cu` . Puede escribir tanto códigos de dispositivo como de host en una misma fuente.

Examples

Código CUDA muy simple.

```
#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;
}
```

Lea Instalando cuda en línea: https://riptutorial.com/es/cuda/topic/10949/instalando-cuda

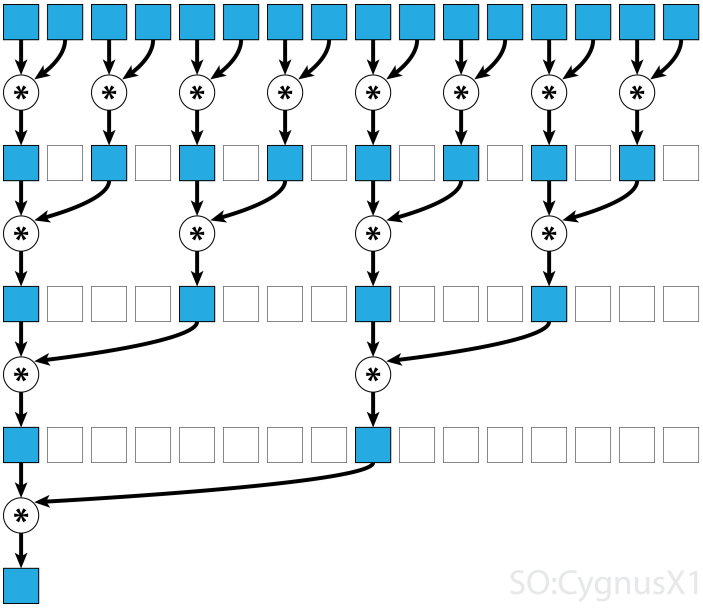
Capítulo 4: Reducción paralela (por ejemplo, cómo sumar una matriz)

Observaciones

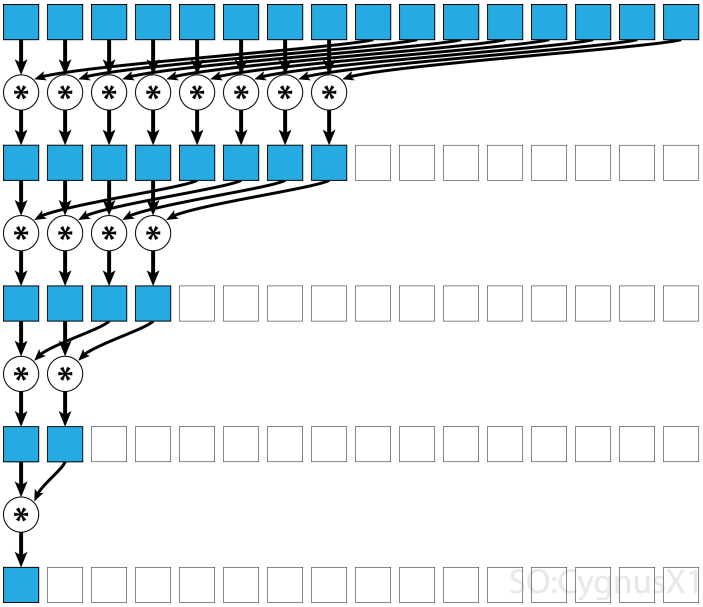
El algoritmo de reducción paralela generalmente se refiere a un algoritmo que combina una matriz de elementos, produciendo un único resultado. Los problemas típicos que entran en esta categoría son:

- resumiendo todos los elementos en una matriz
- encontrar un máximo en una matriz

En general, la reducción paralela se puede aplicar a cualquier operador asociativo binario, es decir `(A*B)*C = A*(B*C)` . Con tal operador *, el algoritmo de reducción paralelo agrupa repetidamente los argumentos de la matriz en pares. Cada par se calcula en paralelo con otros, reduciendo a la mitad el tamaño del arreglo en un solo paso. El proceso se repite hasta que solo existe un elemento.



Si el operador es conmutativo (es decir, `A*B = B*A`) además de ser asociativo, el algoritmo puede emparejarse en un patrón diferente. Desde el punto de vista teórico, no hace ninguna diferencia, pero en la práctica proporciona un mejor patrón de acceso a la memoria:



No todos los operadores asociativos son conmutativos, como la multiplicación de matrices, por ejemplo.

Examples

Reducción en paralelo de bloque único para operador conmutativo

El enfoque más simple para la reducción paralela en CUDA es asignar un solo bloque para realizar la tarea:

```
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);
```

Esto es más factible cuando el tamaño de los datos no es muy grande (alrededor de unos pocos elementos). Esto suele ocurrir cuando la reducción es parte de un programa CUDA más grande. Si la entrada coincide con `blockSize` desde el principio, el primer bucle `for` se puede eliminar por

completo.

Tenga en cuenta que en el primer paso, cuando hay más elementos que hilos, agregamos cosas de forma completamente independiente. Solo cuando el problema se reduce a `blockSize` , se activa la reducción paralela real. El mismo código se puede aplicar a cualquier otro operador asociativo, como conmutación, mínimo, máximo, etc.

Tenga en cuenta que el algoritmo se puede hacer más rápido, por ejemplo, mediante el uso de una reducción paralela de nivel de deformación.

Reducción en paralelo de bloque único para operador no conmutativo

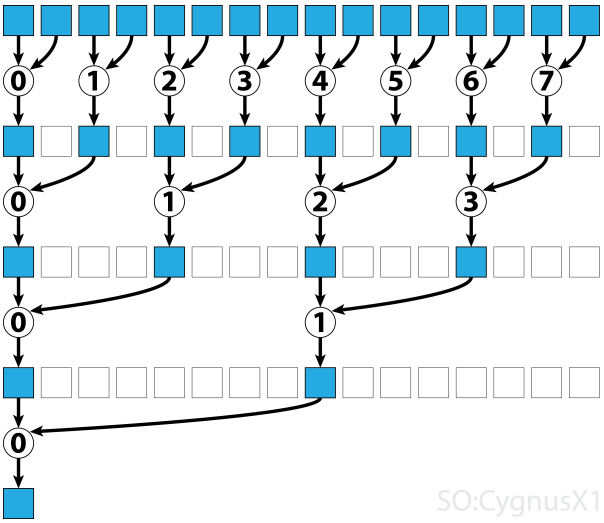
Hacer una reducción paralela para un operador no conmutativo es un poco más complicado, en comparación con la versión conmutativa. En el ejemplo, todavía utilizamos una suma sobre enteros para simplificar. Podría reemplazarse, por ejemplo, con la multiplicación de matrices que realmente no es conmutativa. Tenga en cuenta que, al hacerlo, 0 debe reemplazarse por un elemento neutral de la multiplicación, es decir, una matriz de identidad.

```
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);
```

En el primer ciclo, el bucle se ejecuta siempre que haya más elementos de entrada que hilos. En cada iteración, se realiza una reducción única y el resultado se comprime en la primera mitad de

la matriz `shArr` . La segunda mitad se llena con nuevos datos.

Una vez que todos los datos se cargan desde `gArr` , se ejecuta el segundo bucle. Ahora, ya no comprimimos el resultado (lo que cuesta `__syncthreads()` adicionales). En cada paso, el subproceso n accede al elemento activo `2*n` -th y lo suma con el elemento `2*n+1` -th:



Hay muchas formas de optimizar aún más este simple ejemplo, por ejemplo, a través de la reducción del nivel de distorsión y eliminando los conflictos del banco de memoria compartida.

Reducción paralela multibloque para operador conmutativo.

El enfoque de bloques múltiples para la reducción paralela en CUDA plantea un desafío adicional, en comparación con el enfoque de un solo bloque, porque los bloques están limitados en la comunicación. La idea es dejar que cada bloque calcule una parte de la matriz de entrada y luego tener un bloque final para combinar todos los resultados parciales. Para hacer eso, uno puede lanzar dos núcleos, creando implícitamente un punto de sincronización en toda la red.

```
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;
}
```

Lo ideal sería lanzar bloques suficientes para saturar todos los multiprocesadores en la GPU en plena ocupación. Superar este número, en particular, lanzar tantos subprocesos como elementos en la matriz, es contraproducente. Al hacerlo, ya no aumenta la potencia de cálculo sin procesar, pero evita el uso del primer bucle muy eficiente.

También es posible obtener el mismo resultado utilizando un solo kernel, con la ayuda de la última guardia de bloque :

```
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;
}
```

Tenga en cuenta que el kernel se puede hacer más rápido, por ejemplo, utilizando una reducción paralela de nivel de deformación.

Reducción paralela multibloque para operador no conmutativo

El enfoque de bloques múltiples para la reducción paralela es muy similar al enfoque de un solo bloque. La matriz de entrada global debe dividirse en secciones, cada una reducida en un solo bloque. Cuando se obtiene un resultado parcial de cada bloque, un bloque final los reduce para obtener el resultado final.

- `sumNoncommSingleBlock` se explica con más detalle en el ejemplo de reducción de un solo bloque.
- `lastBlock` acepta solo el último bloque que llega a él. Si quiere evitar esto, puede dividir el kernel en dos llamadas separadas.

```
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);
}
```

Lo ideal sería lanzar bloques suficientes para saturar todos los multiprocesadores en la GPU en plena ocupación. Superar este número, en particular, lanzar tantos subprocesos como elementos en la matriz, es contraproducente. Al hacerlo, ya no aumenta la potencia de cálculo sin procesar, pero evita el uso del primer bucle muy eficiente.

Reducción paralela de una sola urdimbre para operador conmutativo

A veces, la reducción debe realizarse en una escala muy pequeña, como parte de un núcleo

CUDA más grande. Supongamos, por ejemplo, que los datos de entrada tienen exactamente 32 elementos: el número de subprocesos en una deformación. En tal escenario, se puede asignar una sola urdimbre para realizar la reducción. Dado que la deformación se ejecuta en una sincronización perfecta, muchas instrucciones `__syncthreads()` se pueden eliminar, en comparación con una reducción a nivel de bloque.

```
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` es preferiblemente una matriz en memoria compartida. El valor debe ser el mismo para todos los subprocesos en la deformación. Si `sumCommSingleWarp` es llamado por múltiples hilos de urdimbre, `shArr` debe ser diferente entre urdimbres (las mismas dentro de cada urdimbre).

El argumento `shArr` se marca como `volatile` para garantizar que las operaciones en la matriz se realicen donde se indica. De lo contrario, la asignación `shArr[idx]` a `shArr[idx]` se puede optimizar como una asignación a un registro, y solo la asignación final es una tienda real para `shArr` . Cuando eso sucede, las asignaciones inmediatas no son visibles para otros subprocesos, lo que produce resultados incorrectos. Tenga en cuenta que puede pasar una matriz normal no volátil como un argumento de volatile, igual que cuando pasa no constante como un parámetro const.

Si a uno no le importa el contenido de `shArr[1..31]` después de la reducción, puede simplificar el código aún más:

```
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];
}
```

En esta configuración eliminamos muchas `if` condiciones. Los hilos adicionales realizan algunas adiciones innecesarias, pero ya no nos preocupamos por los contenidos que producen. Dado que las deformaciones se ejecutan en modo SIMD, en realidad no ahorramos a tiempo porque esos subprocesos no hacen nada. Por otro lado, la evaluación de las condiciones lleva relativamente mucho tiempo, ya que el cuerpo de estas `if` declaraciones son tan pequeñas. La sentencia `if` inicial también se puede eliminar si `shArr[32..47]` se rellena con 0.

La reducción de nivel de deformación se puede utilizar para aumentar la reducción de nivel de bloque también:

```
__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];
    }
}
```

El argumento `&r[idx & ~(warpSize-1)]` es básicamente `r + warpIdx*32` . Esto divide efectivamente la matriz `r` en trozos de 32 elementos, y cada trozo se asigna a warp separado.

Reducción paralela de una sola urdimbre para un operador no conmutativo.

A veces, la reducción debe realizarse en una escala muy pequeña, como parte de un núcleo CUDA más grande. Supongamos, por ejemplo, que los datos de entrada tienen exactamente 32 elementos: el número de subprocesos en una deformación. En tal escenario, se puede asignar una sola urdimbre para realizar la reducción. Dado que la deformación se ejecuta en una sincronización perfecta, muchas instrucciones `__syncthreads()` se pueden eliminar, en comparación con una reducción a nivel de bloque.

```
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` es preferiblemente una matriz en memoria compartida. El valor debe ser el mismo para todos los subprocesos en la deformación. Si `sumCommSingleWarp` es llamado por múltiples hilos de urdimbre, `shArr` debe ser diferente entre urdimbres (las mismas dentro de cada urdimbre).

El argumento `shArr` se marca como `volatile` para garantizar que las operaciones en la matriz se realicen donde se indica. De lo contrario, la asignación `shArr[idx]` a `shArr[idx]` se puede optimizar como una asignación a un registro, y solo la asignación final es una tienda real para `shArr` . Cuando eso sucede, las asignaciones inmediatas no son visibles para otros subprocesos, lo que produce resultados incorrectos. Tenga en cuenta que puede pasar una matriz normal no volátil

como un argumento de volatile, igual que cuando pasa no constante como un parámetro const.

Si a uno no le importa el contenido final de `shArr[1..31]` y puede rellenar `shArr[32..47]` con ceros, puede simplificar el código anterior:

```
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];
}
```

En esta configuración hemos eliminado todo `if` las condiciones, que constituyen aproximadamente la mitad de las instrucciones. Los subprocesos adicionales realizan algunas adiciones innecesarias, almacenando el resultado en celdas de `shArr` que, en última instancia, no tienen impacto en el resultado final. Dado que las deformaciones se ejecutan en modo SIMD, en realidad no ahorramos a tiempo porque esos subprocesos no hacen nada.

Reducción paralela de una sola urdimbre usando solo registros

Normalmente, la reducción se realiza en una matriz global o compartida. Sin embargo, cuando la reducción se realiza en una escala muy pequeña, como parte de un núcleo CUDA más grande, se puede realizar con una sola deformación. Cuando eso sucede, en las arquitecturas Keppler o superiores (CC> = 3.0), es posible usar las funciones de warp-shuffle para evitar el uso de la memoria compartida.

Supongamos, por ejemplo, que cada subproceso en una deformación contiene un solo valor de datos de entrada. Todos los hilos juntos tienen 32 elementos, que necesitamos resumir (o realizar otra operación asociativa)

```
__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);
}
```

Esta versión funciona para operadores conmutativos y no conmutativos.

Lea Reducción paralela (por ejemplo, cómo sumar una matriz) en línea: https://riptutorial.com/es/cuda/topic/6566/reduccion-paralela--por-ejemplo--como-sumar-una-matriz-

Creditos

S. No	Capítulos	Contributors
1	Empezando con cuda	Community, CygnusX1, Dev-iL, harrism, havogt, infinite.potential, Jared Hoberock, Ken Y-N, Marco13, NikolayKondratyev, Tejus Prasad
2	Comunicación interbloque	CygnusX1, tera
3	Instalando cuda	Mo Sani
4	Reducción paralela (por ejemplo, cómo sumar una matriz)	CygnusX1