eBook Gratuit

APPRENEZ cuda

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#cuda

Table des matières

À propos

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

Chapitre 1: Commencer avec cuda

Remarques

CUDA est une technologie de programmation parallèle et un langage de programmation NVIDIA propriétaires pour leurs GPU.

Les GPU sont des machines hautement parallèles capables d'exécuter des milliers de threads légers en parallèle. Chaque thread GPU est généralement plus lent dans l'exécution et son contexte est plus petit. D'autre part, le GPU est capable d'exécuter plusieurs milliers de threads en parallèle et même plus simultanément (les nombres précis dépendent du modèle de GPU réel). CUDA est un dialecte C ++ conçu spécifiquement pour l'architecture GPU NVIDIA. Cependant, en raison des différences d'architecture, la plupart des algorithmes ne peuvent pas être simplement copiés-collés à partir de C ++, ils seraient exécutés mais seraient très lents.

Terminologie

- *host* - fait référence au matériel normal basé sur le processeur et aux programmes normaux exécutés dans cet environnement
- *device* - fait référence à un GPU spécifique exécuté par les programmes CUDA. Un hôte unique peut prendre en charge plusieurs périphériques.
- *kernel* - une fonction qui réside sur le périphérique et qui peut être appelée à partir du code hôte.

Structure du processeur physique

Le processeur GPU compatible CUDA a la structure physique suivante:

- *la puce* - le processeur entier du GPU. Certains GPU en ont deux.
- *streaming multiprocessor* (SM) - chaque puce contient jusqu'à ~ 100 SM, selon le modèle. Chaque SM fonctionne presque indépendamment l'un de l'autre, en utilisant uniquement la mémoire globale pour communiquer entre eux.
- *CUDA core* - une unité de calcul scalaire unique d'un SM. Leur nombre précis dépend de l'architecture. Chaque cœur peut gérer quelques threads exécutés simultanément dans une succession rapide (similaire à l'hyperthreading dans le CPU).

De plus, chaque SM comporte un ou plusieurs *ordonnanceurs de chaîne* . Chaque planificateur envoie une seule instruction à plusieurs cœurs CUDA. Cela amène le SM à fonctionner en mode SIMD à 32 largeurs .

Modèle d'exécution CUDA

La structure physique du GPU a une influence directe sur la manière dont les noyaux sont exécutés sur le périphérique et sur la façon dont ils sont programmés dans CUDA. Le noyau est

appelé avec une *configuration d'appel* qui spécifie le nombre de threads parallèles générés.

- *the grid* - représente tous les threads générés lors de l'appel du noyau. Il est spécifié comme un ensemble de *blocs* de 1 ou 2 *dimensions*
- *le bloc* - est un ensemble semi-indépendant de *threads* . Chaque bloc est affecté à un seul SM. En tant que tels, les blocs ne peuvent communiquer que par la mémoire globale. Les blocs ne sont synchronisés en aucune façon. S'il y a trop de blocs, certains peuvent être exécutés séquentiellement après d'autres. D'un autre côté, si les ressources le permettent, plusieurs blocs peuvent s'exécuter sur le même serveur de stockage, mais le programmeur ne peut pas en bénéficier (sauf pour l'amélioration des performances évidentes).
- *le thread* - une séquence scalaire d'instructions exécutées par un seul cœur CUDA. Les threads sont «légers» avec un contexte minimal, permettant au matériel de les échanger rapidement. En raison de leur nombre, les threads CUDA fonctionnent avec quelques registres qui leur sont assignés et une pile très courte (de préférence aucune du tout!). Pour cette raison, le compilateur CUDA préfère incorporer tous les appels de fonctions pour aplatir le noyau afin qu'il ne contienne que des sauts et des boucles statiques. Les appels de fonction et les appels de méthode virtuels, bien que pris en charge par la plupart des nouveaux périphériques, entraînent généralement une pénalité majeure en termes de performances.

Chaque thread est identifié par un index de bloc `blockIdx` et un index de thread dans le `threadIdx` . Ces nombres peuvent être vérifiés à tout moment par n'importe quel thread en cours d'exécution et constituent le seul moyen de distinguer un thread d'un autre.

De plus, les threads sont organisés en *chaînes* , chacune contenant exactement 32 threads. Les threads au sein d'une même chaîne s'exécutent dans une synchronisation parfaite, en mode SIMD. Les threads provenant de différentes chaînes, mais dans le même bloc, peuvent s'exécuter dans n'importe quel ordre, mais peuvent être forcés de se synchroniser par le programmeur. Les threads provenant de différents blocs ne peuvent pas être synchronisés ou interagir directement de quelque manière que ce soit.

Organisation de la mémoire

Dans la programmation normale du processeur, l’organisation de la mémoire est généralement masquée par le programmeur. Les programmes typiques agissent comme s'il n'y avait que de la RAM. Toutes les opérations de mémoire, telles que la gestion des registres, l'utilisation de la mise en cache L1-L2-L3, la permutation sur disque, etc. sont gérées par le compilateur, le système d'exploitation ou le matériel lui-même.

Ce n'est pas le cas avec CUDA. Alors que les nouveaux modèles de GPU cachent partiellement le fardeau, par exemple via la mémoire unifiée de CUDA 6, il est toujours utile de comprendre l’organisation pour des raisons de performances. La structure de base de la mémoire CUDA est la suivante:

- *Mémoire hôte* - la RAM normale. Principalement utilisé par le code hôte, mais les nouveaux modèles de GPU peuvent également y accéder. Lorsqu'un noyau accède à la mémoire de l'hôte, le processeur graphique doit communiquer avec la carte mère, généralement via le

connecteur PCIe, ce qui le rend relativement lent.
- *Mémoire de l'appareil / Mémoire globale* - la *mémoire* hors puce principale du GPU, disponible pour tous les threads.
- *Mémoire partagée* - située dans chaque SM permet un accès beaucoup plus rapide que global. La mémoire partagée est privée à chaque bloc. Les threads d'un même bloc peuvent l'utiliser pour la communication.
- *Registers* - Mémoire la plus rapide, privée et non adressable de chaque thread. En général, ils ne peuvent pas être utilisés pour la communication, mais quelques fonctions intrinsèques permettent de mélanger leur contenu dans une chaîne.
- La *mémoire locale* - mémoire privée de chaque fil qui *est* adressable. Ceci est utilisé pour les déversements de registres et les tableaux locaux avec indexation variable. Physiquement, ils résident dans la mémoire globale.
- *Mémoire de texture, mémoire constante* - une partie de la mémoire globale marquée comme immuable pour le noyau. Cela permet au GPU d'utiliser des caches spéciaux.
- *Cache L2* - sur puce, disponible pour tous les threads. Compte tenu de la quantité de threads, la durée de vie attendue de chaque ligne de cache est nettement inférieure à celle du processeur. Il est principalement utilisé des modèles d'accès à la mémoire mal alignés et partiellement aléatoires.
- *Cache L1* - situé dans le même espace que la mémoire partagée. Encore une fois, la quantité est plutôt petite, étant donné le nombre de threads qui l'utilisent, ne vous attendez donc pas à ce que les données y restent longtemps. La mise en cache L1 peut être désactivée.

Versions

Capacité de calcul	Architecture	Nom de code GPU	Date de sortie
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

Capacité de calcul	Architecture	Nom de code GPU	Date de sortie
5.0	Maxwell	GM107, GM108	2014-02-18
5.2	Maxwell	GM200, GM204, GM206	2014-09-18
5.3	Maxwell	GM20B	2015-04-02
6,0	Pascal	GP100	2016-10-01
6.1	Pascal	GP102, GP104, GP106	2016-05-27

La date de publication marque la sortie du premier processeur graphique prenant en charge une capacité de calcul donnée. Certaines dates sont approximatives, par exemple la carte 3.2 a été publiée au deuxième trimestre 2014.

Examples

Conditions préalables

Pour démarrer la programmation avec CUDA, téléchargez et installez CUDA Toolkit et le pilote de développement . La boîte à outils comprend `nvcc` , le compilateur NVIDIA CUDA et d'autres logiciels nécessaires au développement d'applications CUDA. Le pilote garantit que les programmes GPU fonctionnent correctement sur le matériel compatible CUDA , dont vous aurez également besoin.

Vous pouvez confirmer que CUDA Toolkit est correctement installé sur votre machine en exécutant `nvcc --version` partir d'une ligne de commande. Par exemple, sur une machine 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
```

génère les informations du compilateur. Si la commande précédente a `nvcc` , le CUDA Toolkit n'est probablement pas installé ou le chemin d'accès à `nvcc` (`C:\CUDA\bin` sur les machines Windows, `/usr/local/cuda/bin` sur les systèmes d'exploitation POSIX) ne fait pas partie de votre Variable d'environnement `PATH` .

De plus, vous aurez également besoin d'un compilateur hôte qui fonctionne avec `nvcc` pour compiler et construire des programmes CUDA. Sous Windows, il s'agit de `cl.exe` , le compilateur Microsoft, `cl.exe` avec Microsoft Visual Studio. Sur les systèmes d'exploitation POSIX, d'autres compilateurs sont disponibles, y compris `gcc` ou `g++` . Le Guide de démarrage rapide CUDA officiel peut vous indiquer quelles versions du compilateur sont prises en charge sur votre plate-forme particulière.

Pour vous assurer que tout est configuré correctement, compilons et exécutons un programme

CUDA trivial pour nous assurer que tous les outils fonctionnent correctement ensemble.

```
__global__ void foo() {}

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

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

  return 0;
}
```

Pour compiler ce programme, copiez-le dans un fichier appelé test.cu et compilez-le à partir de la ligne de commande. Par exemple, sur un système Linux, les éléments suivants devraient fonctionner:

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

Si le programme réussit sans erreur, alors commençons à coder!

Sommez deux tableaux avec CUDA

Cet exemple montre comment créer un programme simple qui additionnera deux tableaux `int` à CUDA.

Un programme CUDA est hétérogène et comprend des parties exécutées à la fois sur le processeur et sur le GPU.

Les parties principales d'un programme utilisant CUDA sont similaires aux programmes CPU et se composent de

- Allocation de mémoire pour les données qui seront utilisées sur le GPU
- Copie de données de la mémoire hôte vers la mémoire GPU
- Invoquer la fonction du noyau pour traiter des données
- Copier le résultat dans la mémoire de la CPU

Pour allouer la mémoire des périphériques, nous utilisons la fonction `cudaMalloc` . Pour copier des données entre le périphérique et l'hôte, la fonction `cudaMemcpy` peut être utilisée. Le dernier argument de `cudaMemcpy` spécifie la direction de l'opération de copie. Il y a 5 types possibles:

- `cudaMemcpyHostToHost` - Hôte -> Hôte
- `cudaMemcpyHostToDevice` - Hôte -> Périphérique
- `cudaMemcpyDeviceToHost` - Périphérique -> Hôte
- `cudaMemcpyDeviceToDevice` - Périphérique -> Périphérique
- `cudaMemcpyDefault` - Espace d'adressage virtuel unifié par défaut

Ensuite, la fonction du noyau est appelée. L'information entre les chevrons triples est la

configuration d'exécution, qui dicte combien de threads de périphérique exécutent le noyau en parallèle. Le premier nombre (`2` dans l'exemple) spécifie le nombre de blocs et le second (`(size + 1) / 2` dans l'exemple) - nombre de threads dans un bloc. Notez que dans cet exemple nous ajoutons 1 à la taille, de sorte que nous demandons un thread supplémentaire plutôt que d'avoir un thread responsable de deux éléments.

Comme l'invocation du noyau est une fonction asynchrone, `cudaDeviceSynchronize` est appelée pour attendre que l'exécution soit terminée. Les tableaux de résultats sont copiés dans la mémoire hôte et toute la mémoire allouée sur le périphérique est libérée avec `cudaFree` .

Pour définir la fonction comme noyau, le `__global__` déclaration `__global__` est utilisé. Cette fonction sera appelée par chaque thread. Si nous voulons que chaque thread traite un élément du tableau résultant, nous avons besoin d'un moyen de distinguer et d'identifier chaque thread. CUDA définit les variables `blockDim` , `blockIdx` et `threadIdx` . La variable prédéfinie `blockDim` contient les dimensions de chaque bloc de thread comme spécifié dans le deuxième paramètre de configuration d'exécution pour le lancement du noyau. Les variables prédéfinies `threadIdx` et `blockIdx` contiennent respectivement l'index du thread dans son bloc de thread et le bloc de thread dans la grille. Notez que puisque nous demandons potentiellement un thread de plus que des éléments dans les tableaux, nous devons passer en `size` pour nous assurer de ne pas accéder au-delà de la fin du tableau.

```
#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;
}
```

Lançons un seul thread CUDA pour dire bonjour

Ce programme CUDA simple montre comment écrire une fonction qui s'exécutera sur le GPU (aka "device"). Le CPU, ou "host", crée des threads CUDA en appelant des fonctions spéciales appelées "noyaux". Les programmes CUDA sont des programmes C ++ avec une syntaxe supplémentaire.

Pour voir comment cela fonctionne, placez le code suivant dans un fichier nommé `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;
}
```

(Notez que pour utiliser la fonction `printf` sur le périphérique, vous avez besoin d'un périphérique ayant une capacité de calcul d'au moins 2.0. Voir la présentation des versions pour plus de détails.)

Maintenant, compilons le programme en utilisant le compilateur NVIDIA et exécutons-le:

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

Quelques informations supplémentaires sur l'exemple ci-dessus:

- `nvcc` signifie "NVIDIA CUDA Compiler". Il sépare le code source en composants hôte et périphérique.
- `__global__` est un mot-clé CUDA utilisé dans les déclarations de fonction indiquant que la fonction s'exécute sur le périphérique GPU et est appelée depuis l'hôte.
- Les crochets à trois angles (`<<<` , `>>>`) marquent un appel du code hôte vers le code du périphérique (également appelé «lancement du noyau»). Les nombres entre ces crochets indiquent le nombre de fois à exécuter en parallèle et le nombre de threads.

Compiler et exécuter les exemples de programmes

Le guide d'installation NVIDIA se termine par l'exécution des exemples de programmes pour vérifier votre installation de CUDA Toolkit, mais n'indique pas explicitement comment. Commencez par vérifier toutes les conditions préalables. Vérifiez le répertoire CUDA par défaut pour les exemples de programmes. S'il n'est pas présent, il peut être téléchargé à partir du site Web officiel de CUDA. Accédez au répertoire où les exemples sont présents.

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

Vous devriez voir une sortie similaire à:

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

Assurez-vous que le `Makefile` est présent dans ce répertoire. La commande `make` dans les systèmes UNIX générera tous les exemples de programmes. Vous pouvez également accéder à un sous-répertoire dans lequel un autre `Makefile` est présent et exécuter la commande `make` partir de là pour créer uniquement cet échantillon.

Exécutez les deux exemples de programmes `deviceQuery` - `deviceQuery` et `bandwidthTest` :

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

La sortie sera similaire à celle ci-dessous:

```
./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
```

L'instruction `Result = PASS` à la fin indique que tout fonctionne correctement. Maintenant, exécutez l'autre exemple de `bandwidthTest` programme suggérée d'une manière similaire. La sortie sera similaire à:

```
[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.
```

Encore une fois, l'instruction `Result = PASS` indique que tout a été exécuté correctement. Tous les autres programmes exemples peuvent être exécutés de la même manière.

Lire Commencer avec cuda en ligne: https://riptutorial.com/fr/cuda/topic/1860/commencer-avec-cuda

Chapitre 2: Communication inter-blocs

Remarques

Les blocs dans CUDA fonctionnent de manière semi-indépendante. Il n'y a pas de moyen sûr de les synchroniser tous. Cependant, cela ne signifie pas qu'ils ne peuvent pas interagir les uns avec les autres.

Examples

Garde de dernier bloc

Considérons une grille travaillant sur une tâche, par exemple une réduction parallèle. Au départ, chaque bloc peut faire son travail de manière indépendante, produisant un résultat partiel. À la fin, les résultats partiels doivent être combinés et fusionnés. Un exemple typique est un algorithme de réduction sur un big data.

Une approche typique consiste à invoquer deux noyaux, l'un pour le calcul partiel et l'autre pour la fusion. Cependant, si la fusion peut être effectuée efficacement par un seul bloc, un seul appel au noyau est requis. Ceci est réalisé par un garde de dernier `lastBlock` défini comme:

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;
}
```

Avec un tel garde, le dernier bloc est garanti pour voir tous les résultats produits par tous les autres blocs et peut effectuer la fusion.

```
__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);
    }
}
```

Hypothèses:

- Le compteur doit être un pointeur de mémoire global initialisé à 0 *avant que* le noyau soit appelé.
- La fonction `lastBlock` est invoquée uniformément par tous les threads de tous les blocs
- Le noyau est appelé dans une grille de dimension unique (pour simplifier l'exemple)
- `T` nomme n'importe quel type que vous aimez, mais l'exemple n'est pas destiné à être un modèle au sens C ++

File d'attente de travail globale

Considérez un tableau d'éléments de travail. Le temps nécessaire à l'exécution de chaque tâche varie grandement. Afin d'équilibrer la répartition du travail entre les blocs, il peut être prudent pour chaque bloc de récupérer l'élément suivant uniquement lorsque le précédent est terminé. Ceci contraste avec l'attribution a priori d'éléments aux blocs.

```
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
  }
}
```

Hypothèses:

- L'objet WorkQueue, ainsi que le tableau gItem résident dans la mémoire globale
- Aucun nouvel élément de travail n'est ajouté à l'objet WorkQueue dans le noyau qui en extrait
- Le `WorkItem` est une petite représentation de l'affectation de travail, par exemple un pointeur vers un autre objet.
- `WorkItem::none()` membre statique `WorkItem::none()` crée un objet `WorkItem` qui ne représente aucun travail

- `WorkQueue::fetch()` doit être appelé uniformément par tous les threads du bloc
- Il n'y a pas 2 invocations de `WorkQueue::fetch()` sans un autre `__syncthreads()` entre. Sinon une condition de course apparaîtra!

L'exemple n'inclut pas comment initialiser le `WorkQueue` ou le remplir. Cela est fait par un autre noyau ou code CPU et devrait être assez simple.

Lire Communication inter-blocs en ligne: https://riptutorial.com/fr/cuda/topic/4978/communication-inter-blocs

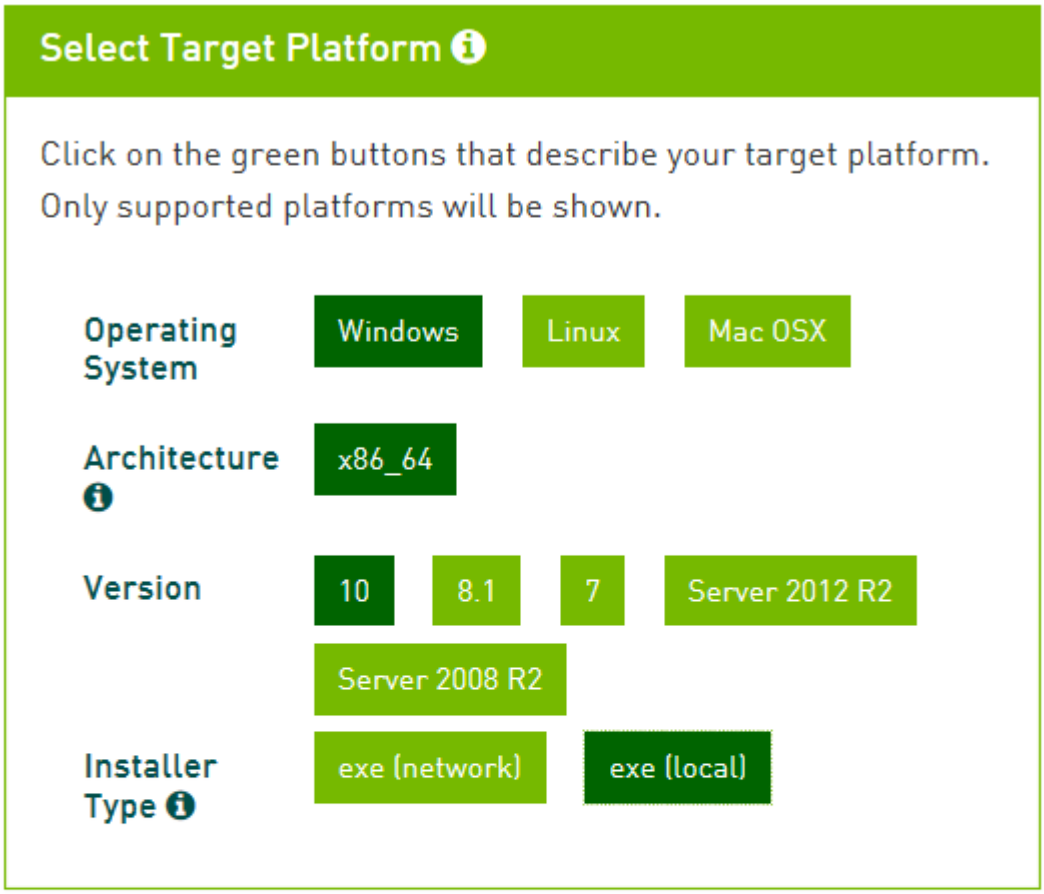
Chapitre 3: Installation de cuda

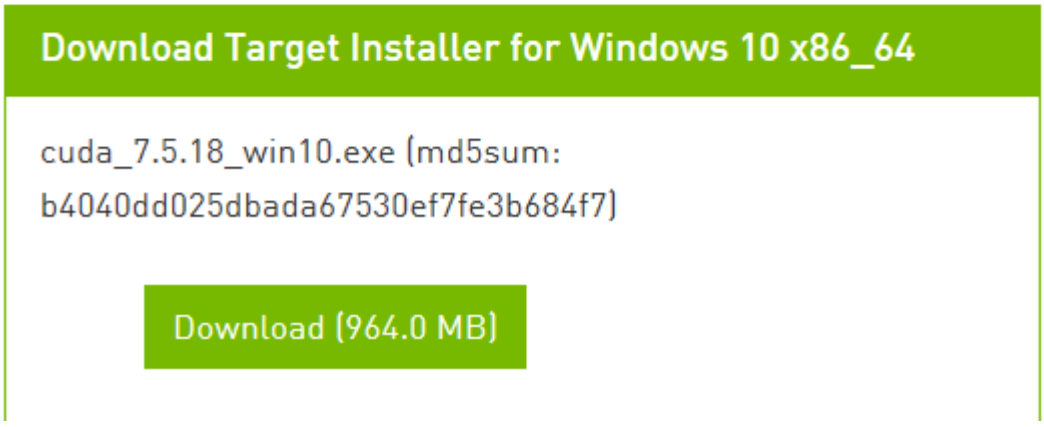
Remarques

Pour installer le toolkit CUDA sur Windows, vous devez d'abord installer une version correcte de Visual Studio. Visual Studio 2013 doit être installé si vous souhaitez installer CUDA 7.0 ou 7.5. Visual Studio 2015 est pris en charge pour CUDA 8.0 et les versions ultérieures.

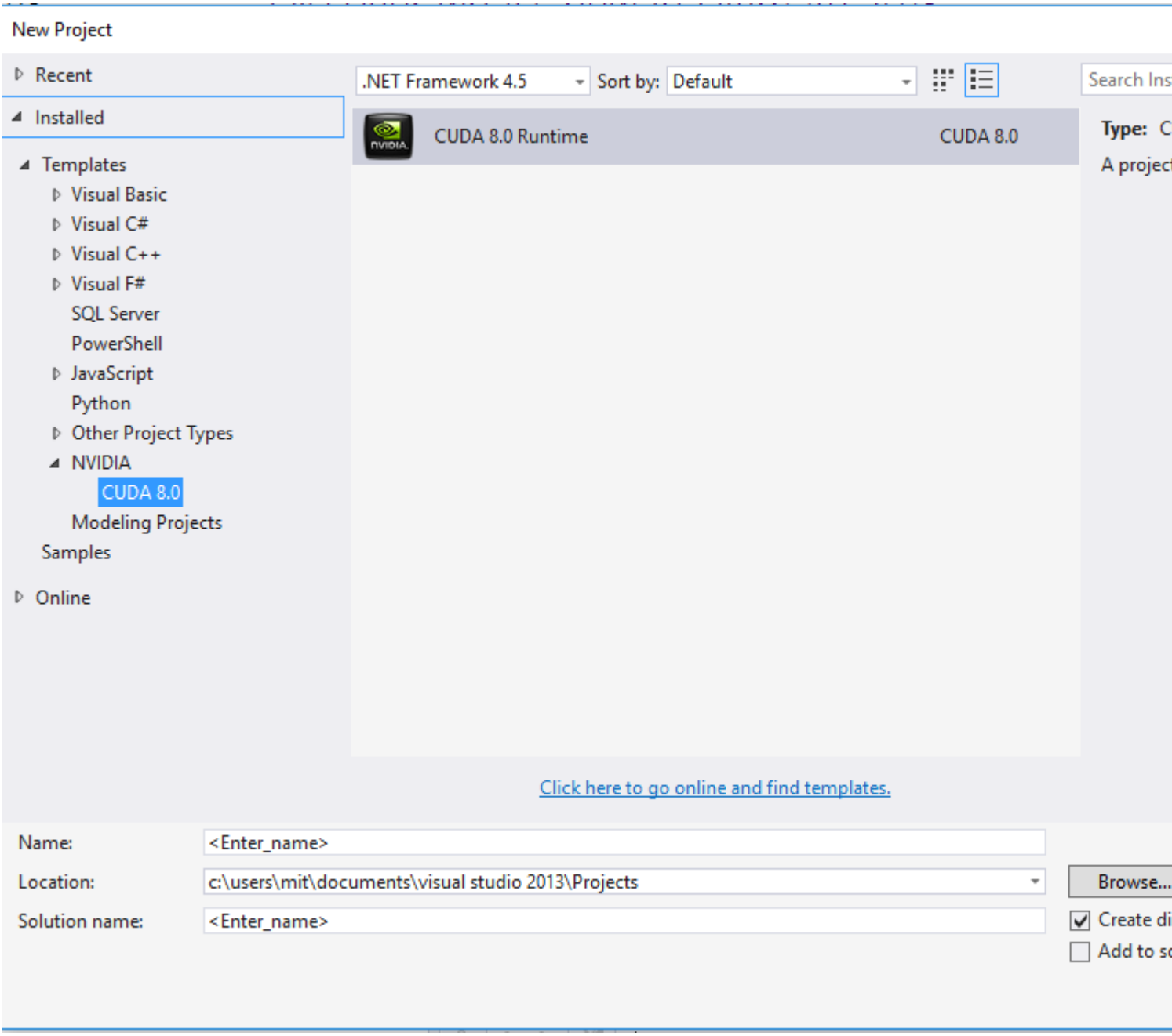
Lorsque vous avez une version appropriée de VS sur votre système, il est temps de télécharger et d'installer la boîte à outils CUDA. Suivez ce lien pour trouver la version de la boîte à outils CUDA que vous recherchez: archive de la boîte à outils CUDA

Dans la page de téléchargement, vous devez choisir la version de Windows sur la machine cible et le type d’installateur (choisissez local).





Après avoir téléchargé le fichier exe, vous devez l'extraire et exécuter `setup.exe` . Une fois l'installation terminée, ouvrez un nouveau projet et choisissez NVIDIA> CUDAX.X dans les modèles.



N'oubliez pas que l'extension des fichiers source CUDA est `.cu` . Vous pouvez écrire les codes hôte et périphérique sur une même source.

Examples

Code CUDA très 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;
}
```

Lire Installation de cuda en ligne: https://riptutorial.com/fr/cuda/topic/10949/installation-de-cuda

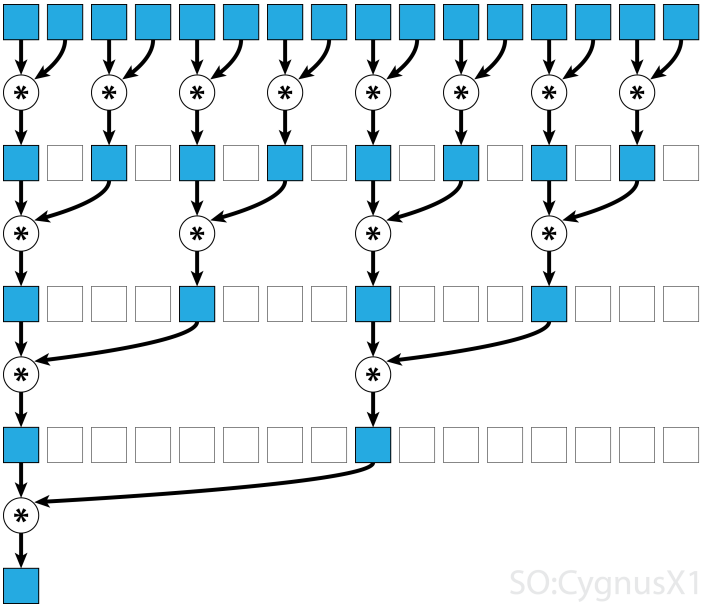
Chapitre 4: Réduction parallèle (par exemple comment additionner un tableau)

Remarques

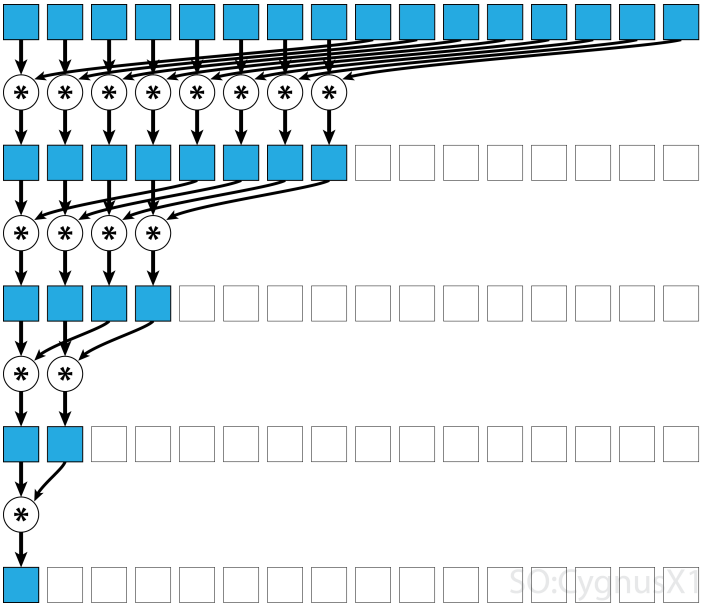
L'algorithme de réduction parallèle fait généralement référence à un algorithme qui combine un ensemble d'éléments, produisant un seul résultat. Les problèmes typiques qui entrent dans cette catégorie sont les suivants:

- résumer tous les éléments d'un tableau
- trouver un maximum dans un tableau

En général, la réduction parallèle peut être appliquée à tout opérateur associatif binaire, à savoir `(A*B)*C = A*(B*C)` . Avec un tel opérateur *, l'algorithme de réduction parallèle regroupe à répétition les arguments du tableau par paires. Chaque paire est calculée en parallèle avec d'autres, réduisant de moitié la taille globale de la matrice en une seule étape. Le processus est répété jusqu'à ce qu'un seul élément existe.



Si l'opérateur est commutatif (c.-à-d. `A*B = B*A`) en plus d'être associatif, l'algorithme peut se coupler selon un modèle différent. Du point de vue théorique, cela ne fait aucune différence, mais en pratique, cela donne un meilleur modèle d'accès à la mémoire:



Tous les opérateurs associatifs ne sont pas commutatifs - prenons par exemple la multiplication matricielle.

Examples

Réduction parallèle monobloc pour opérateur commutatif

L'approche la plus simple pour la réduction parallèle dans CUDA consiste à affecter un seul bloc pour effectuer la tâche:

```
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);
```

Cela est plus facile lorsque la taille des données n'est pas très grande (autour de quelques milliers d'éléments). Cela se produit généralement lorsque la réduction fait partie d'un programme CUDA plus important. Si l'entrée correspond à `blockSize` dès le début, la première `for` la boucle peut être

complètement enlevée.

Notez que dans la première étape, quand il y a plus d'éléments que de threads, nous ajoutons les choses complètement indépendamment. Ce n'est que lorsque le problème est réduit à `blockSize` la réduction parallèle réelle se déclenche. Le même code peut être appliqué à tout autre opérateur associatif commutatif, tel que multiplication, minimum, maximum, etc.

Notez que l'algorithme peut être réalisé plus rapidement, par exemple en utilisant une réduction parallèle au niveau de la chaîne.

Réduction parallèle monobloc pour opérateur non commutatif

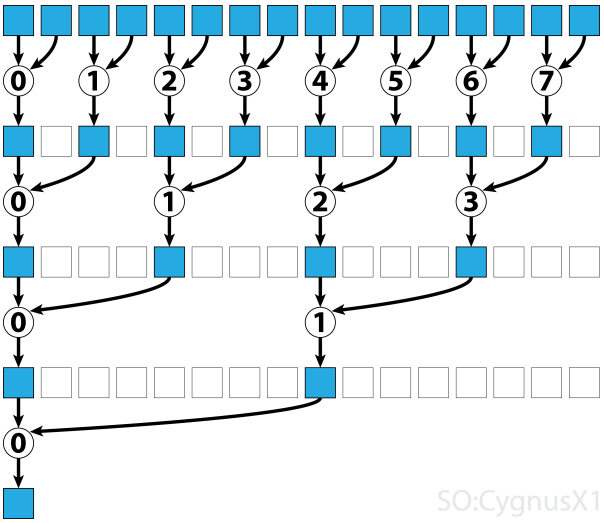
Faire de la réduction parallèle pour un opérateur non-commutatif est un peu plus complexe que la version commutative. Dans l'exemple, nous utilisons encore un ajout sur les entiers pour la simplicité. Il pourrait être remplacé, par exemple, par la multiplication matricielle qui est vraiment non-commutative. Notez que ce faisant, 0 doit être remplacé par un élément neutre de la multiplication, c'est-à-dire une matrice d'identité.

```
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);
```

Dans la première, la boucle while s'exécute tant qu'il y a plus d'éléments d'entrée que de threads. Dans chaque itération, une seule réduction est effectuée et le résultat est compressé dans la

première moitié du tableau `shArr` . La seconde moitié est ensuite remplie de nouvelles données.

Une fois toutes les données chargées depuis `gArr` , la deuxième boucle est `gArr` . Maintenant, nous ne compressons plus le résultat (ce qui coûte un `__syncthreads()` supplémentaire). A chaque étape, le thread n accède au `2*n` ème élément actif et l'ajoute avec `2*n+1` -ème élément:



Il existe de nombreuses manières d'optimiser cet exemple simple, par exemple en réduisant le niveau de distorsion et en supprimant les conflits de banque de mémoire partagée.

Réduction parallèle multiblocs pour opérateur commutatif

Une approche multi-blocs pour la réduction parallèle de CUDA pose un défi supplémentaire, comparé à l'approche à un seul bloc, car les blocs sont limités dans la communication. L'idée est de laisser chaque bloc calculer une partie du tableau d'entrée, puis de disposer d'un dernier bloc pour fusionner tous les résultats partiels. Pour ce faire, il est possible de lancer deux noyaux, créant implicitement un point de synchronisation à l'échelle de la grille.

```
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;
}
```

On souhaite idéalement lancer suffisamment de blocs pour saturer tous les multiprocesseurs du processeur graphique à pleine capacité. Dépasser ce nombre - en particulier, lancer autant de threads qu'il y a d'éléments dans le tableau - est contre-productif. Cela n'augmente plus la puissance de calcul brute, mais empêche d'utiliser la première boucle très efficace.

Il est également possible d’obtenir le même résultat en utilisant un seul noyau, à l’aide de la protection du dernier bloc :

```
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;
}
```

Notez que le noyau peut être rendu plus rapide, par exemple en utilisant une réduction parallèle au niveau de la chaîne.

Réduction parallèle multibloc pour un opérateur non commutatif

L'approche à plusieurs blocs pour la réduction parallèle est très similaire à l'approche à un seul bloc. Le tableau d'entrée global doit être divisé en sections, chacune étant réduite d'un seul bloc. Lorsqu'un résultat partiel de chaque bloc est obtenu, un bloc final les réduit pour obtenir le résultat final.

- `sumNoncommSingleBlock` est expliqué plus en détail dans l'exemple de réduction de bloc unique.
- `lastBlock` n'accepte que le dernier bloc atteint. Si vous voulez éviter cela, vous pouvez diviser le noyau en deux appels distincts.

```
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);
}
```

On souhaite idéalement lancer suffisamment de blocs pour saturer tous les multiprocesseurs du processeur graphique à pleine capacité. Dépasser ce nombre - en particulier, lancer autant de threads qu'il y a d'éléments dans le tableau - est contre-productif. Cela n'augmente plus la puissance de calcul brute, mais empêche d'utiliser la première boucle très efficace.

Réduction parallèle à une seule chaîne pour un opérateur commutatif

Parfois, la réduction doit être effectuée à très petite échelle, dans le cadre d'un noyau CUDA plus important. Supposons, par exemple, que les données d’entrée contiennent exactement 32

éléments - le nombre de threads dans une chaîne. Dans un tel scénario, une seule chaîne peut être affectée à la réduction. Étant donné que warp s'exécute dans une synchronisation parfaite, de nombreuses instructions `__syncthreads()` peuvent être supprimées - par rapport à une réduction au niveau du bloc.

```
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` est de préférence un tableau en mémoire partagée. La valeur doit être la même pour tous les threads du Warp. Si `sumCommSingleWarp` est appelé par plusieurs `shArr` , `shArr` devrait être différent entre les warps (même dans chaque warp).

L'argument `shArr` est marqué comme `volatile` pour garantir que les opérations sur le tableau sont réellement effectuées là où elles sont indiquées. Sinon, l'affectation répétitive à `shArr[idx]` peut être optimisée en tant qu'affectation à un registre, seule l'assignation finale étant un magasin réel pour `shArr` . Lorsque cela se produit, les affectations immédiates ne sont pas visibles pour les autres threads, ce qui produit des résultats incorrects. Notez que vous pouvez passer un tableau non-volatile normal en tant qu'argument de volatile, comme lorsque vous transmettez un paramètre non-const comme paramètre const.

Si l'on ne se soucie pas du contenu de `shArr[1..31]` après la réduction, on peut simplifier davantage le code:

```
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];
}
```

Dans cette configuration , nous avons supprimé beaucoup `if` les conditions. Les threads supplémentaires effectuent des ajouts inutiles, mais nous ne nous soucions plus du contenu qu'ils produisent. Comme les warps s'exécutent en mode SIMD, nous ne réalisons pas vraiment de gain de temps en ne faisant rien. D'un autre côté, l'évaluation des conditions prend beaucoup de temps, puisque le corps de celles `if` ci est si petit. L'instruction `if` initiale peut également être supprimée si `shArr[32..47]` est `shArr[32..47]` par 0.

La réduction au niveau de la chaîne peut également être utilisée pour augmenter la réduction au niveau du bloc:

```
__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];
    }
}
```

L'argument `&r[idx & ~(warpSize-1)]` est essentiellement `r + warpIdx*32` . Cela divise effectivement le tableau `r` en morceaux de 32 éléments, et chaque segment est assigné à séparer la chaîne.

Réduction parallèle à une seule chaîne pour opérateur non commutatif

Parfois, la réduction doit être effectuée à très petite échelle, dans le cadre d'un noyau CUDA plus important. Supposons, par exemple, que les données d'entrée contiennent exactement 32 éléments - le nombre de threads dans une chaîne. Dans un tel scénario, une seule chaîne peut être affectée à la réduction. Étant donné que warp s'exécute dans une synchronisation parfaite, de nombreuses instructions `__syncthreads()` peuvent être supprimées - par rapport à une réduction au niveau du bloc.

```
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` est de préférence un tableau en mémoire partagée. La valeur doit être la même pour tous les threads du Warp. Si `sumCommSingleWarp` est appelé par plusieurs `shArr` , `shArr` devrait être différent entre les warps (même dans chaque warp).

L'argument `shArr` est marqué comme `volatile` pour garantir que les opérations sur le tableau sont réellement effectuées là où elles sont indiquées. Sinon, l'affectation répétitive à `shArr[idx]` peut être optimisée en tant qu'affectation à un registre, seule l'assignation finale étant un magasin réel pour `shArr` . Lorsque cela se produit, les affectations immédiates ne sont pas visibles pour les autres threads, ce qui produit des résultats incorrects. Notez que vous pouvez passer un tableau

non-volatile normal en tant qu'argument de volatile, comme lorsque vous transmettez un paramètre non-const comme paramètre const.

Si l'on ne se soucie pas du contenu final de `shArr[1..31]` et que l'on peut `shArr[32..47]` avec des zéros, on peut simplifier le code ci-dessus:

```
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];
}
```

Dans cette configuration , nous avons supprimé toutes `if` les conditions, qui constituent environ la moitié des instructions. Les threads supplémentaires effectuent des ajouts inutiles, stockant le résultat dans des cellules de `shArr` qui n'ont finalement aucun impact sur le résultat final. Comme les warps s'exécutent en mode SIMD, nous ne réalisons pas vraiment de gain de temps en ne faisant rien.

Réduction parallèle à une seule chaîne à l'aide de registres uniquement

En règle générale, la réduction est effectuée sur un tableau global ou partagé. Cependant, lorsque la réduction est effectuée à très petite échelle, dans le cadre d'un noyau CUDA plus important, elle peut être effectuée avec une seule chaîne. Lorsque cela se produit, sur les architectures Keppler ou supérieures (CC> = 3.0), il est possible d'utiliser des fonctions Warp-Shuffle pour éviter d'utiliser la mémoire partagée.

Supposons, par exemple, que chaque thread dans une chaîne contient une seule valeur de données d'entrée. Tous les threads ensemble ont 32 éléments, que nous devons résumer (ou effectuer d'autres opérations associatives)

```
__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);
}
```

Cette version fonctionne à la fois pour les opérateurs commutatifs et non commutatifs.

Lire Réduction parallèle (par exemple comment additionner un tableau) en ligne: https://riptutorial.com/fr/cuda/topic/6566/reduction-parallele--par-exemple-comment-additionner-un-tableau-

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec cuda	Community, CygnusX1, Dev-iL, harrism, havogt, infinite.potential, Jared Hoberock, Ken Y-N, Marco13, NikolayKondratyev, Tejus Prasad
2	Communication inter-blocs	CygnusX1, tera
3	Installation de cuda	Mo Sani
4	Réduction parallèle (par exemple comment additionner un tableau)	CygnusX1