Paralleles Rechnen

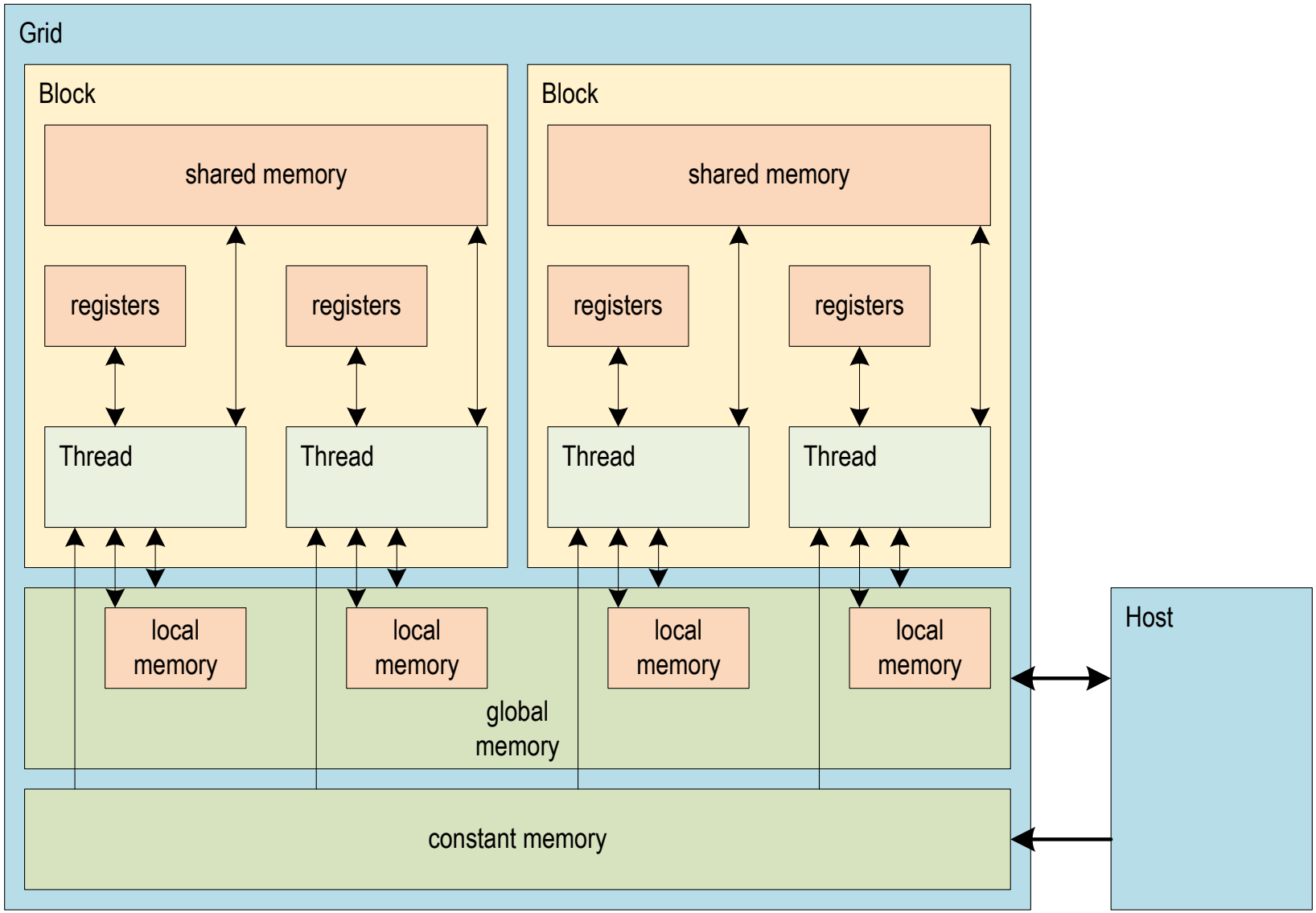
GPGPU mit NVIDIA CUDA – Speicherarten

Peter Kulczycki
peter.kulczycki<AT>fh-hagenberg.at

Department of Medical Informatics and Bioinformatics
University of Applied Sciences Upper Austria
Softwarepark 11, 4232 Hagenberg, Austria

Version 1·02·29452 – 14. Oktober 2015

Speicher / Übersicht



Die Speicherhierarchie auf der GPU

Speicher / Variable Type Qualifiers

Diese Attribute für Variablen bestimmen, in welchem Speicherbereich eine Variable auf dem Device abgelegt wird [NVIDIA, 2015, CUDA C Programming Guide, Kapitel B.2 und E.2.2]:

Attribut	**Memory**	**Lifet.**	**Scope**
`__constant__`	*constant*	App.	alle Threads in einem Grid, Host (`cudaMemcpyToSymbol`)
`__device__`	*global*	App.	alle Threads in einem Grid, Host (`cudaMemcpy`)
`__managed__`	*global*	App.	wie `__device__`, direkter Zugriff via Zeiger vom Host
`__shared__`	*shared*	Block	alle Threads in einem Block

Anmerkung: Wird ein C-Array mit `extern __shared__` deklariert, dann bestimmt der dritte Parameter in der *execution configuration* die Größe dieses C-Arrays. Siehe dazu Folie 3-20.

Speicher / Variablendefinitionen

Variablen werden, abhängig von ihrer genauen Definition, in unterschiedlichen Speicherregionen abgelegt:

Definition	**Memory**	**Speed**	**Lifet.**	**Scope**
`T v;`	*register*	*on-chip*	Thread	Thread
`T v [10];`	*local*	*uncached off-chip*	Thread	Thread
`__shared__ T v;`	*shared*	*on-chip*	Block	Block
`__device__ T v;`	*global*	*uncached off-chip*	App.	Grid
`__constant__ T v;`	*constant*	*cached off-chip*	App.	Grid

Anmerkung: Skalare Variablen ohne Attribute in ihren Definitionen werden im *register memory* abgelegt. C-Arrays ohne Attribute werden im (*thread-*)*local memory* abgelegt.

Speicher / Variablendefinitionen

Variablen (bzw. die damit assoziierten Speicherbereiche) können in den folgenden Kontexten definiert und verwendet werden:

1. Vom Host aus kann Speicher im *global memory* allokiert (und dann natürlich auch verwendet) werden. Zum Beispiel mit den API-Funktionen `cudaMalloc`, `cudaMallocHost`, `cudaMallocPitch`, `cudaHostGetDevicePointer`, `cudaMemcpy`, `cudaMemcpy2D`, `cudaMemset` und `cudaFree`.
2. Globale Variablen, die mit den Attributen `__constant__` und `__device__` versehen sind, können vom Host aus verwendet werden. Benötigt wird dazu die API-Funktion `cudaGetSymbolAddress`.
3. Im Kernel können lokale Skalare und lokale C-Arrays definiert werden (ohne Attribute bzw. mit dem Attribut `__shared__`). Diese Variablen sind vom Host aus nicht verwendbar.
4. Die Größe des *shared memory* kann vom Host aus definiert werden. Siehe dazu Folie 3-20.

Speicher / Local Memory

```
1  __device__ void kernel () {
2     int s;       // a scalar
3     int v [10]; // a c-array
4  }
```

1. Die lokale Variable `s` wird in einem Register abgelegt.
2. Die lokale Variable `v` wird im *local memory* abgelegt (ein Register ist nicht indizierbar).
3. Stehen nicht genügend Register zur Verfügung, dann wird auf den *local memory* ausgewichen.
4. Die Größe eines C-Arrays muss ein Literal sein (so ist z. B. die Definition `int v [n];` nicht erlaubt).
5. Der *local memory* ist eigentlich kein eigener Speicher, er ist ein Bereich im *global memory*.

Speicher / Shared Memory

1. *Shared memory* ist (fast) so schnell wie *register memory*.
2. Alle Threads in einem Block können untereinander über den *shared memory* kommunizieren. (Threads in unterschiedlichen Blöcken können nur über den *global memory* kommunizieren.)
3. Auf manchen Plattformen (z. B. Fermi) sind *shared memory* und *L1 cache* durch dieselbe Hardware dargestellt.
4. Die Größe des *shared memory* kann dynamisch festgelegt werden. Siehe dazu Folie 3-20.

Anmerkung: Die Reihenfolge, in der die Threads eines Blocks auf ihren *shared memory* zugreifen, ist undefiniert:

```
1  __device__ void kernel () {
2      __shared__ int i;
3      i = threadIdx.x; // race condition, use e.g. __syncthreads
4  }
```

Speicher / Shared Memory

Werden mehrere C-Arrays mit `extern __shared__` deklariert, so liegen sie alle „übereinander“ an derselben Adresse im *shared memory*:

```
1  extern __shared__ int a []; // e.g. m elements
2  extern __shared__ int b []; // e.g. n elements; but a == b
```

Beide C-Arrays sind in einem einzelnen C-Array hintereinander abzulegen und dann vom Programmierer durch geeignete Indizierung zu trennen:

```
1  extern __shared__ int a_and_b []; // m + n elements
2
3  // ...
4
5  a_and_b[0 + i] // i.e., a[i]
6  a_and_b[m + j] // i.e., b[j]
```

Speicher / Shared Memory

Die allgemeine Strategie im Zusammenspiel von *shared memory* und *global memory*:

1. Die zu verarbeitenden Daten werden vom *global memory* in den *shared memory* kopiert.
2. Die Daten werden im *shared memory* verarbeitet.
3. Die Daten werden vom *shared memory* wieder in den *global memory* zurückkopiert.

```
1  __global__ void kernel (RGB_4_t const * const p_tbl) {
2     __shared__ RGB_4_t tbl [SIZE];
3
4     if (threadIdx.x < SIZE) {                  // each thread
5        tbl[threadIdx.x] = p_tbl[threadIdx.x]; // copies one
6     }                                          // table entry
7
8     __syncthreads ();
9
10    // ...
11 }
```

Speicher / Global Memory

```
1   __constant__ int v0 [32]; // in constant memory
2                int v1 [32]; // on host
3
4   cudaMemcpyToSymbol   (v0, v1, 32 * sizeof (int));
5   cudaMemcpyFromSymbol (v1, v0, 32 * sizeof (int));
6
7   // ------------------------------------------------
8
9   __device__ int v2;       // in global memory
10              int v3 = 42; // on host
11
12  cudaMemcpyToSymbol (v2, &v3, sizeof (int));
13
14  // ------------------------------------------------
15
16  __device__ int * p0; // in global memory
17              int * p1; // on host
18
19  cudaMalloc (&p1, 32 * sizeof (int));
20  cudaMemcpyToSymbol (p0, &p1, sizeof (int *));
```

Siehe dazu [NVIDIA, 2015, CUDA C Programming Guide, Kapitel 3.2.2] (*symbol* bezeichnet eine Variable im *global* oder im *constant memory*).

Speicher / Pinned-, Mapped-, WC-Memory

Page-locked host memory (auch *pinned memory*) ist Speicher auf dem Host, der vom Betriebssystem nicht ausgelagert wird (*virtual memory*).

1. Das Kopieren von Daten zwischen Host und Device kann parallel zum Ablauf eines Kernels erfolgen. [NVIDIA, 2015, CUDA C Programming Guide, Kapitel 3.2.5]
2. *Page-locked host memory* kann in den Adressraum eines Devices eingeblendet werden. Dadurch kann aufwändiges Kopieren zwischen Host und Device vermieden werden. [NVIDIA, 2015, CUDA C Programming Guide, Kapitel 3.2.4.3]
3. Auf Systemen mit einem *Front-Side Bus* (CPU - PCIe - GPU) bietet *page-locked host memory* die größte Bandbreite.

Anmerkung: *Page-locked host memory* sollte trotz seiner Vorteile nicht zu häufig verwendet werden. Reduziert er doch den verfügbaren Hauptspeicher am Host, die Gesamtrechenleistung des Systems kann darunter leiden.

Speicher / Pinned-, Mapped-, WC-Memory

Die API-Funktionen `cudaMallocHost` and `cudaFreeHost` allokieren *page-locked host memory* bzw. geben ihn wieder frei. Mit der API-Funktion `cudaHostRegister` kann ein auf dem Host mit `malloc` allokierter Speicherbereich gesperrt werden (*page-locked*).

Mögliche Werte für den Parameter `flags` der API-Funktion `cudaMallocHost` sind (`cudaHostAllocXXX`):

`Default`	Keine Wirkung (die API-Funktionen `cudaHostAlloc` und `cudaMallocHost` sind synonym).
`Mapped`	Blendet den Speicher in den CUDA-Adressraum ein (*zero copy*, *mapped*, `canMapHostMemory`, `cudaSetDeviceFlags`, `cudaDeviceMapHost`, `cudaHostGetDevicePointer`).
`Portable`	Alle CUDA-Kontexte (nicht nur der aktuelle) betrachten den Speicher als gesperrt.
`WriteCombined`	Allokiert den Speicher als *write-combining memory*.

Speicher / Pinned-, Mapped-, WC-Memory

```
1  cudaDeviceProp props;
2  cudaGetDeviceProperties (&props, 0); // device #0
3
4  if (props.canMapHostMemory) {
5     cudaSetDeviceFlags (cudaDeviceMapHost);
6
7     // allocate page-locked (pinned) host memory
8     cudaMallocHost (/*...*/);
9     cudaMalloc (/*...*/);
10
11    // allocate zero-copy (mapped) host memory
12    cudaMallocHost (/*...*/);
13    cudaHostGetDevicePointer (/*...*/);
14
15    // ...
16
17    cudaFree (/*...*/); // omit when zero-copy
18    cudaFreeHost (/*...*/);
19 }
```

Anmerkung: *Page-locked host memory* wird *by default* als „cachebar“ allokiert.

Speicher / Write-Combining Memory

1. Wenn *page-locked host memory* als *write-combining memory* allokiert wird, dann werden die entsprechenden L1- und L2-Cache-Ressourcen freigegeben. Andere Caches können dadurch größer werden.
2. Auch entfällt dadurch der Aufwand für Cache-Kohärenzprotokolle (*snooping*). Die Bandbreite kann um bis zu 40 % steigen.
3. Der Host kann auf *write-combining memory* schreibend zwar sehr schnell, lesend aber nur sehr langsam zugreifen. Daher wird dieser Speicher verwendet, um Daten vom Host zum Device zu übertragen, aber nicht umgekehrt.

Ablauf / Threads und Warps

Software-Hierarchie:	Grid - Block - Thread
Hardware-Hierarchie:	Processor - SM - Core

1. Threads werden zu Blöcken gruppiert.
2. Alle Threads eines Blocks werden in einem Streaming-Multiprocessor (SM) ausgeführt.
3. Jeder SM hat einen *shared memory*, mit dessen Hilfe die Threads eines Blocks kommunizieren können.
4. Alle Threads eines Blocks müssen fertig abgelaufen sein, bevor ein anderer Block „geschedult“ werden kann.
5. Hardware-Schedulers ordnen die Threads eines Blocks einem SM zu.
6. Hat ein SM mehr Ressourcen, so kann der Scheduler mehrere Blöcke einem SM zuordnen.
7. Es wird keine Ausführungsreihenfolge unter den Threads definiert.

Ablauf / Threads und Warps

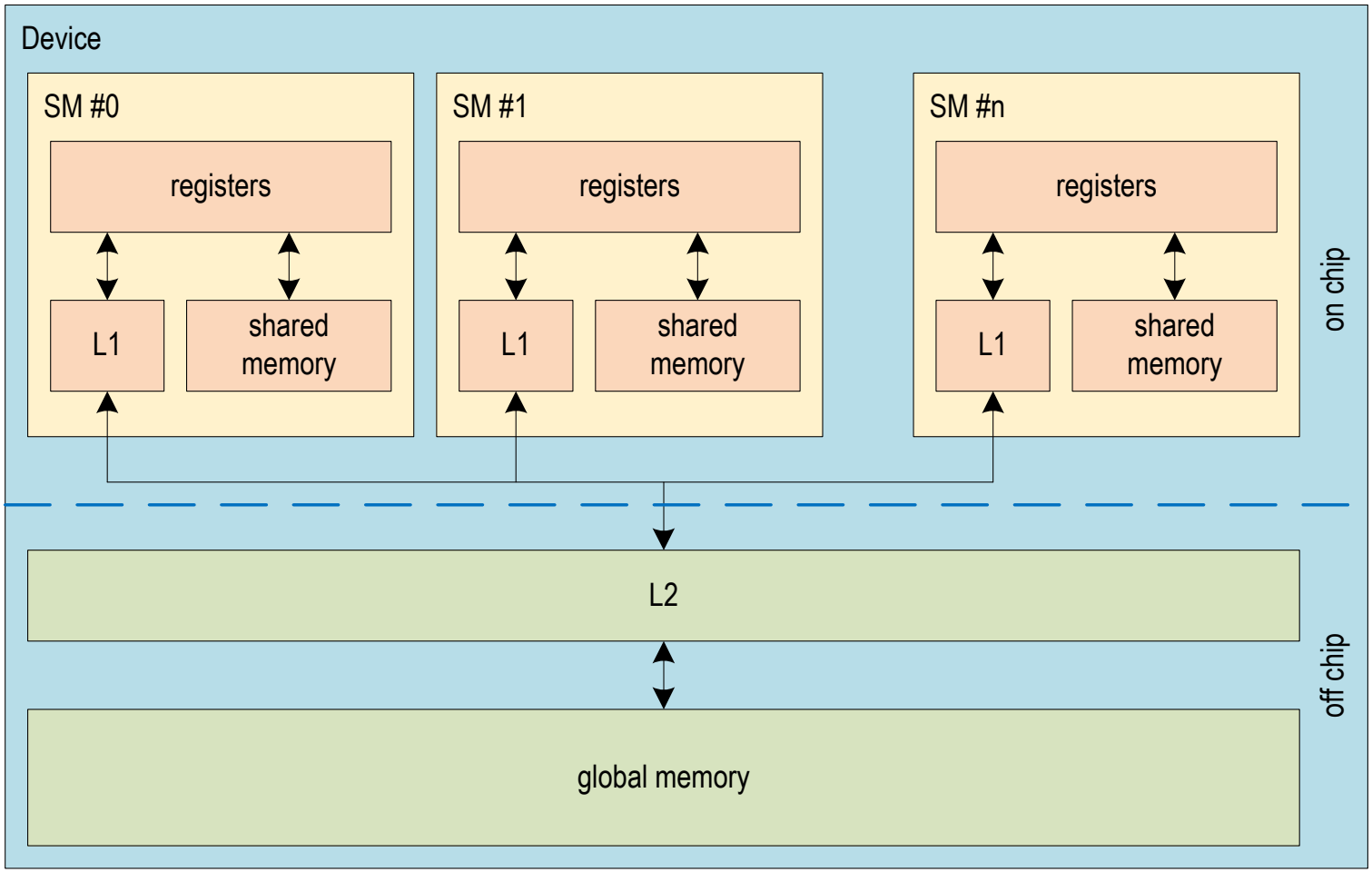
1. Ein SM startet Threads immer in Gruppen zu 32. Diese 32 Threads werden Warp genannt.
2. Warps werden durch einen Warp-Scheduler verwaltet.
3. Pro SM läuft immer nur ein Warp.
4. Der *execution context* eines Warps (IP, Registers etc.) wird von der Hardware verwaltet.
5. Der *context-switch* zwischen zwei Warps eines SMs verursacht keine Kosten.
6. Die Threads eines Warps sollten immer dieselbe Anweisung ausführen (SIMD). Führt der Programmpfad (`if`, `while`) einzelne Threads eines Warps in unterschiedliche Richtungen (*control flow divergence*), so werden die verschiedenen Ausführungspfade serialisiert.
7. Hängt der Programmpfad von der *thread id* ab, so sollte die Verzweigungsbedingung nur von `threadIdx / 32` abhängen.

Ablauf / Vereinigter Zugriff auf Global Memory

1. Greifen die Threads eines Warps schreibend oder lesend auf den *global memory* zu, dann verbindet das Device diese Zugriffe, falls möglich, zu einem einzigen.
2. Auf wieviele Speicherzugriffe ein Device die 32 Zugriffe der Threads eines Warps reduzieren kann hängt davon ab, wie viele *cache lines* von diesen 32 Zugriffen betroffen sind.
3. Eine *cache line* des L1-Caches hat 128 Bytes. Eine *cache line* des L2-Caches hat 32 Bytes.

Siehe dazu [NVIDIA, 2015, CUDA C Programming Guide, Kapitel 9.2.1].

Ablauf / Vereinigter Zugriff auf Global Memory



Der Zugriff auf die Speicher der GPU

Ablauf / Function Type Qualifiers

Diese Attribute für Funktionen bestimmen, wo eine Funktion läuft und wer sie aufrufen kann [NVIDIA, 2015, CUDA C Programming Guide, Kapitel B.1]:

Attribut	**läuft auf**	**Aufruf von**
`__device__`	Device	Device
`__global__`	Device	Host, Device (ab *compute capability* 3.x)
`__host__`	Host	Host

Anmerkung: Mit `__global__` versehene Funktionen laufen asynchron und müssen `void` retournieren. Eine Funktion ohne Attribut erhält implizit das Attribut `__host__`. Die Attribute `__device__` und `__host__` können miteinander verwendet werden. Andere Kombinationen sind nicht zulässig.

Ablauf / Execution Configuration

Wird eine mit dem Attribut `__global__` versehene Funktion aufgerufen, so muss dabei die *execution configuration* angegeben werden [NVIDIA, 2015, CUDA C Programming Guide, Kapitel B.19]:

```
kernel <<<grid, block, shared, stream>>> (/* ... */);
```

Name	Typ	Default	Bedeutung
`grid`	`dim3`		Anzahl der Blöcke pro Grid
`block`	`dim3`		Anzahl der Threads pro Block
`shared`	`size_t`	`0`	Größe des pro Block dynamisch allokierten *shared memory*
`stream`	`cudaStream_t`	`0`	der mit dem Kernelaufruf assoziierte *command stream*

Literatur


NVIDIA.
CUDA Documents, 2015.
Verfügbar im Internet unter *http://docs.nvidia.com/cuda*.