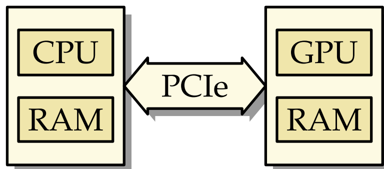
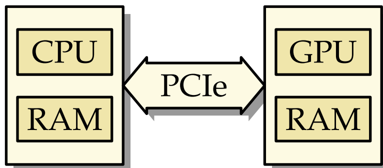
- Schon sehr früh gab es diverse Grafik-Beschleuniger, die der normalen CPU Arbeit abnahmen.
- Die im März 2001 von Nvidia eingeführte GeForce 3 Series führte programmierbares Shading ein.
- Im August 2002 folgte die Radeon R300 von ATI, die die Fähigkeiten der GeForce 3 deutlich erweiterte um mathematische Funktionen und Schleifen.
- Zunehmend werden die GPUs zu GPGPUs (*general purpose GPUs*).
- Zur generellen Nutzung wurden mehrere Sprachen und Schnittstellen entwickelt: OpenCL (Open Computing Language), DirectCompute (von Microsoft), CUDA (Compute Unified Device Architecture, von Nvidia) und OpenACC (*open accelerators*, von Cray, CAPS, Nvidia und PGI). Wir beschäftigen uns hier zunächst mit CUDA, da es recht populär ist und bei uns auch zur Verfügung steht.



- Überwiegend stehen GPUs als PCI-Express-Steckkarten zur Verfügung.
- Sie leben und arbeiten getrennt von der CPU und ihrem Speicher. Eine Kommunikation muss immer über die PCI-Express-Verbindung erfolgen.
- Meistens haben sie völlig eigenständige und spezialisierte Prozessorarchitekturen. Eine prominente Ausnahme ist die Larrabee-Mikroarchitektur von Intel, die sich an der x86-Architektur ausrichtete und der später die Xeon-Phi-Architektur folgte, die keine GPU mehr ist und nur noch dem High-Performance-Computing dient.



- Eine entsprechende Anwendung besteht normalerweise aus zwei Programmen (eines für die CPU und eines für die GPU).
- Das Programm für die GPU muss in die GPU heruntergeladen werden; anschließend sind die Daten über die PCI-Express-Verbindung auszutauschen.
- Die Kommunikation kann (mit Hilfe der zur Verfügung stehenden Bibliothek) sowohl synchron als auch asynchron erfolgen.

- Für die GPUs stehen höhere Programmiersprachen zur Verfügung, typischerweise Varianten, die auf C bzw. C++ basieren.
- Hierbei kommen Spracherweiterungen für GPUs hinzu, und andererseits wird C bzw. C++ nicht umfassend unterstützt. Insbesondere stehen außer wenigen mathematischen Funktionen keine Standard-Bibliotheken zur Verfügung.
- Nach dem Modell von CUDA sind die Programmtexte für die CPU und die GPU in der gleichen Quelle vermischt. Diese werden auseinandersortiert und dann getrennt voneinander übersetzt. Der entstehende GPU-Code wird dann als Byte-Array in den CPU-Code eingefügt. Zur Laufzeit wird dann nur noch der fertig übersetzte GPU-Code zur GPU geladen.
- Beim Modell von OpenCL sind die Programmtexte getrennt, wobei der für die GPU bestimmte Programmtext erst zur Laufzeit übersetzt und mit Hilfe von OpenCL-Bibliotheksfunktionen in die GPU geladen wird.
- Zunächst betrachten wir CUDA...

CUDA ist ein von Nvidia für Linux, MacOS und Windows kostenfrei zur Verfügung gestelltes Paket (jedoch nicht *open source*), das folgende Komponenten umfasst:

- einen Gerätetreiber,
- eine Spracherweiterung von C++ (CUDA C++), die es ermöglicht, in einem Programmtext die Teile für die CPU und die GPU zu vereinen,
- einen Übersetzer *nvcc* (zu finden im Verzeichnis */usr/local/cuda-10.1/bin* auf dem Rechner Livingstone, der CUDA C++ unterstützt,
- eine zugehörige Laufzeitbibliothek (*libcudart.so* in */usr/local/cuda-10.1/lib64* auf Livingstone und
- darauf aufbauende Bibliotheken (einschließlich BLAS und FFT).

URL: `https://developer.nvidia.com/cuda-downloads`

properties.cu

```
#include <cstdlib>
#include <iostream>

inline void check_cuda_error(const char* cudaop, const char* source,
       unsigned int line, cudaError_t error) {
   if (error != cudaSuccess) {
      std::cerr << cudaop << " at " << source << ":" << line
         << " failed: " << cudaGetErrorString(error) << std::endl;
      exit(1);
   }
}

#define CHECK_CUDA(opname, ...) \
   check_cuda_error(#opname, __FILE__, __LINE__, opname(__VA_ARGS__))

int main() {
   int device; CHECK_CUDA(cudaGetDevice, &device);
   // ...
}
```

- Bei CUDA-Programmen steht die CUDA-Laufzeit-Bibliothek zur Verfügung. Die Fehler-Codes aller CUDA-Bibliotheksaufrufe sollten jeweils überprüft werden.

properties.cu

```
int device_count; CHECK_CUDA(cudaGetDeviceCount, &device_count);
struct cudaDeviceProp device_prop;
CHECK_CUDA(cudaGetDeviceProperties, &device_prop, device);
if (device_count > 1) {
   std::cout << "device " << device << " selected out of "
      << device_count << " devices:" << std::endl;
} else {
   std::cout << "one device present:" << std::endl;
}
std::cout << "name: " << device_prop.name << std::endl;
std::cout << "compute capability: " << device_prop.major
   << "." << device_prop.minor << std::endl;
// ...
```

- *cudaGetDeviceProperties* füllt eine umfangreiche Datenstruktur mit den Eigenschaften einer der zur Verfügung stehenden GPUs.
- Es gibt eine Vielzahl von Nvidia-Karten mit sehr unterschiedlichen Fähigkeiten und Ausstattungen, so dass bei portablen Anwendungen diese u.U. zuerst überprüft werden müssen.
- Die Namen der CUDA-Programme enden normalerweise in „.cu“.

```
livingstone$ make
nvcc -o properties properties.cu
livingstone$ ./properties
one device present:
name: Quadro P620
compute capability: 6.1
total global memory: 2096103424
total constant memory: 65536
total shared memory per block: 49152
registers per block: 65536
L2 Cache Size: 524288
warp size: 32
mem pitch: 2147483647
max threads per block: 1024
max threads dim: 1024 1024 64
max grid dim: 2147483647 65535 65535
multi processor count: 4
kernel exec timeout enabled: no
device overlap: yes
integrated: no
can map host memory: yes
unified addressing: yes
livingstone$
```

```
livingstone$ make
nvcc -o properties properties.cu
```

- Übersetzt wird mit *nvcc*.
- Hierbei werden die für die GPU vorgesehenen Teile übersetzt und in Byte-Arrays verwandelt, die dann zusammen mit den für die CPU bestimmten Programmteilen dem *gcc* zur Übersetzung gegeben werden.
- Optionen wie „–gpu-architecture compute_60“ ermöglicht die Nutzung wichtiger später hinzu gekommener Erweiterungen.
- Wenn die Option „-code sm_60“ hinzukommt, wird beim Übersetzen auch der Code für die CPU erzeugt. Ansonsten geschieht dies bei neueren CUDA-Versionen erst zur Laufzeit im JIT-Verfahren.

simple.cu

```
__global__ void add(std::size_t len, double alpha, double* x, double* y) {
   for (std::size_t i = 0; i < len; ++i) {
      y[i] += alpha * x[i];
   }
}
```

- Funktionen (oder Methoden), die für die GPU bestimmt sind und von der CPU aus aufrufbar sind, werden Kernel-Funktionen genannt und mit dem CUDA-Schlüsselwort **__global__** gekennzeichnet.
- Die Funktion *add* in diesem Beispiel addiert den Vektor x multipliziert mit *alpha* zu y.
- Alle Zeiger verweisen hier auf GPU-Speicher.

simple.cu

```
/* execute kernel function on GPU */
add<<<1, 1>>>(N, 2.0, device_a, device_b);
```

- Eine Kernel-Funktion kann direkt von dem auf der CPU ausgeführten Programmtext aufgerufen werden.
- Zwischen dem Funktionsnamen und den Parametern wird in „<<<...>>>“ die Kernel-Konfiguration angegeben. Diese ist zwingend notwendig. Die einzelnen Konfigurationsparameter werden später erläutert.
- Elementare Datentypen können direkt übergeben werden (hier N und der Wert 2.0), Zeiger müssen aber bereits auf GPU-Speicher zeigen, d.h. dass ggf. Daten zuvor vom CPU-Speicher zum GPU-Speicher zu transferieren sind.

simple.cu

```
double a[N]; double b[N];
for (std::size_t i = 0; i < N; ++i) {
   a[i] = i; b[i] = i * i;
}

/* transfer vectors to GPU memory */
double* device_a;
CHECK_CUDA(cudaMalloc, (void**)&device_a, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, device_a, a, N * sizeof(double),
   cudaMemcpyHostToDevice);
double* device_b;
CHECK_CUDA(cudaMalloc, (void**)&device_b, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, device_b, b, N * sizeof(double),
   cudaMemcpyHostToDevice);
```

- Mit der CUDA-Bibliotheksfunktion *cudaMalloc* kann Speicher bei der GPU belegt werden.
- Es kann davon ausgegangen werden, dass alle Datentypen auf der CPU und der GPU in gleicher Weise repräsentiert werden. Anders als bei MPI kann die Übertragung ohne Konvertierungen erfolgen.

simple.cu

```
double a[N]; double b[N];
for (std::size_t i = 0; i < N; ++i) {
   a[i] = i; b[i] = i * i;
}

/* transfer vectors to GPU memory */
double* device_a;
CHECK_CUDA(cudaMalloc, (void**)&device_a, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, device_a, a, N * sizeof(double),
   cudaMemcpyHostToDevice);
double* device_b;
CHECK_CUDA(cudaMalloc, (void**)&device_b, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, device_b, b, N * sizeof(double),
   cudaMemcpyHostToDevice);
```

- Mit *cudaMemcpy* kann von CPU- in GPU-Speicher oder umgekehrt kopiert werden. Zuerst kommt (wie bei *memcpy*) das Ziel, dann die Quelle, dann die Zahl der Bytes. Zuletzt wird die Richtung angegeben.
- Ohne den letzten Parameter weiß *cudaMemcpy* nicht, um was für Zeiger es sich handelt.

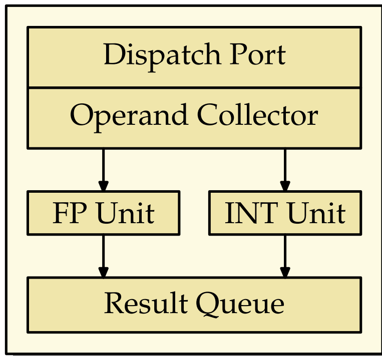
simple.cu

```
/* transfer result vector from GPU device to host memory */
CHECK_CUDA(cudaMemcpy, b, device_b, N * sizeof(double),
   cudaMemcpyDeviceToHost);
/* free space allocated at GPU memory */
CHECK_CUDA(cudaFree, device_a); CHECK_CUDA(cudaFree, device_b);
```

- Nach dem Aufruf der Kernel-Funktion kann das Ergebnis zurückkopiert werden.
- Kernel-Funktionen laufen asynchron, d.h. die weitere Ausführung des Programms auf der CPU läuft parallel zu der Verarbeitung auf der GPU. Sobald die Ergebnisse jedoch zurückkopiert werden, findet implizit eine Synchronisierung statt, d.h. es wird auf die Beendigung der Kernel-Funktion gewartet.
- Wenn der Kernel nicht gestartet werden konnte oder es zu Problemen während der Ausführung kam, wird dies über die Fehler-Codes bei der folgenden synchronisierenden CUDA-Operation mitgeteilt.
- Anschließend sollte der GPU-Speicher freigegeben werden, wenn er nicht mehr benötigt wird.

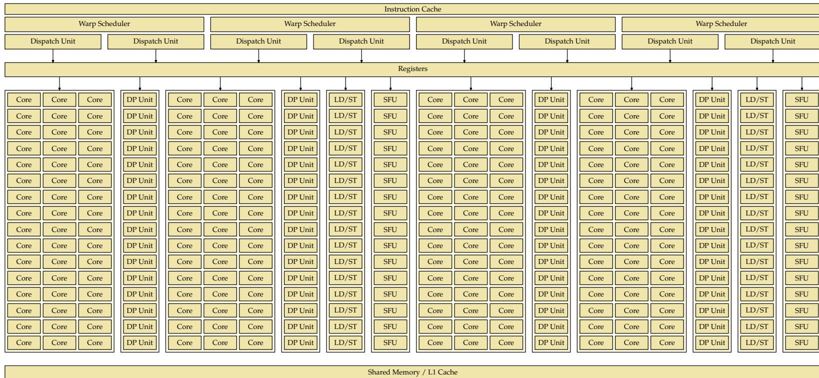
Ein erstes Programm für die GPU

- Das erste Beispiel zeigt den typischen Ablauf vieler Anwendungen:
  - Speicher auf der GPU belegen
  - Daten zur GPU transferieren
  - Kernel-Funktion aufrufen
  - Ergebnisse zurücktransferieren
  - GPU-Speicher freigeben
- Eine echte Parallelisierung hat das Beispiel nicht gebracht, da die CPU auf die GPU wartete und auf der GPU nur ein einziger GPU-Kern aktiv war...



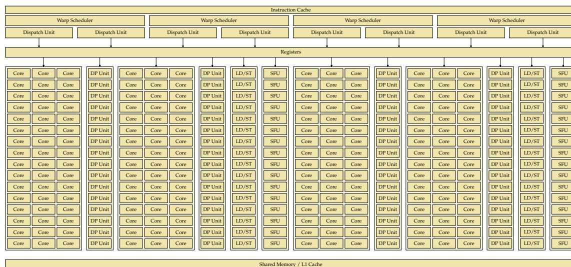
- Die elementaren Recheneinheiten einer GPU bestehen im wesentlichen nur aus zwei Komponenten, jeweils eine für arithmetische Operationen für Gleitkommazahlen und eine für ganze Zahlen.
- Mehr Teile hat ein GPU-Kern nicht. Die Instruktion und die Daten werden angeliefert (*Dispatch Port* und *Operand Collector*), und das Resultat wird bei der *Result Queue* abgeliefert.

Multiprozessor einer GPU



- Dieses Diagramm zeigt einen einzelnen Multiprozessor der Kepler-Mikroarchitektur mit 192 CUDA-Kernen, die mit einfacher Genauigkeit arbeiten („Core“) und 64 Kerne, die mit doppelter Genauigkeit operieren („DP Unit“).

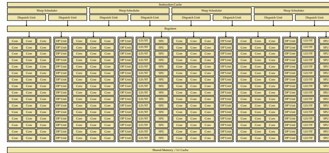
Multiprozessor einer GPU



- Auf der Livingstone haben wir die Pascal-Mikroarchitektur, wobei jeder Multiprozessor 128 CUDA-Kerne besitzt.
- Hinzu kommen Einheiten zum parallelisierten Laden und Speichern (*LD/ST*) und Einheiten für spezielle Operationen wie beispielsweise *sin* oder *sqrt* (*SFU*).
- (Abgebildet ist hier die Kepler-Architektur.)

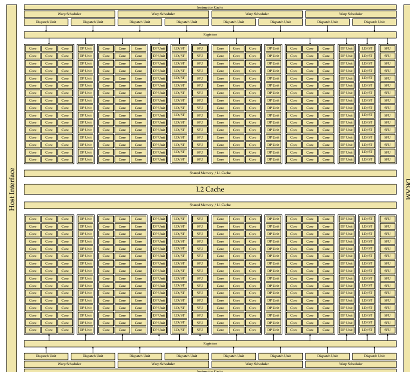
- Die Kerne operieren nicht unabhängig voneinander.
- Im Normalfall werden 32 Kerne zu einem Warp zusammengefasst. (Unterstützt werden auch halbe Warps mit 16 Kernen.)
- Alle Kerne eines Warps führen synchron die gleiche Instruktion aus – auf unterschiedlichen Daten (SIMD-Architektur: Array-Prozessor).
- Dabei werden jeweils zunächst die zu ladenden Daten parallel organisiert (durch die *LD/ST*-Einheiten) und dann über die Register den einzelnen Kernen zur Verfügung gestellt.

Multiprozessoren einer GPU



- Jeder Warp-Scheduler hat die Kontrolle über einen Warp mit 32 Threads.
- Die Kepler-Architektur bietet einen Warp-Scheduler mit zwei Dispatch-Units an, wobei jeder von diesen in der Lage ist, eine Instruktion für eine Gruppe von CUDA-Kernen oder speziellen Einheiten weiterzugeben in Abhängigkeit von der aktuellen Instruktion.
- Entsprechend können zwei Instruktionen bei einem Thread parallel ausgeführt werden.
- Instruktionen mit doppelter Genauigkeit müssen zweifach aussgeführt werden, je einmal für einen halben Warp mit 16 Threads, da jedem Warp-Scheduler nur 16 Kerne mit doppelter Genauigkeit zur Verfügung stehen.
- Jeder Kepler-Multiprozessor ist mit vier Warp-Schedulern ausgestattet.

GPU mit mehreren Multiprozessoren



- Je nach Ausführung der GPU werden mehrere Multiprozessoren zusammengefasst.
- Livingstone bietet 4 Multiprozessoren mit insgesamt 512 CUDA-Kernen an. Auf meinem iMac sind es zwei Multiprozessoren mit insgesamt 384 CUDA-Kernen (abgebildet).

- Der Instruktionssatz der Nvidia-GPUs ist proprietär und bis heute wurde abgesehen von einer kleinen Übersicht von Nvidia kein umfassendes öffentliches Handbuch dazu herausgegeben.
- Mit Hilfe des Werkzeugs *cuobjdump* lässt sich der Code disassemblieren und ansehen.
- Die Instruktionen haben entweder einen Umfang von 32 oder 64 Bits. 64-Bit-Instruktionen sind auf 64-Bit-Kanten.
- Arithmetische Instruktionen haben bis zu drei Operanden und ein Ziel, bei dem das Ergebnis abgelegt wird. Beispiel ist etwa eine Instruktion, die in einfacher Genauigkeit $d = a * b + c$ berechnet (FMAD).

Wie können bedingte Sprünge umgesetzt werden, wenn ein Warp auf eine if-Anweisung stößt und die einzelnen Threads des Warps unterschiedlich weitermachen wollen? (Zur Erinnerung: Alle Threads eines Warps führen immer die gleiche Instruktion aus.)

- Es stehen zwei Stacks zur Verfügung:
- Ein Stack mit Masken, bestehend aus 32 Bits, die festlegen, welche der 32 Threads die aktuellen Instruktionen ausführen.
- Ferner gibt es noch einen Stack mit Zieladressen.
- Bei einer bedingten Verzweigung legt jeder der Threads in der Maske fest, ob die folgenden Instruktionen ihn betreffen oder nicht. Diese Maske wird auf den Stack der Masken befördert.
- Die Zieladresse des Sprungs wird auf den Stack der Zieladressen befördert.
- Wenn die Zieladresse erreicht wird, wird auf beiden Stacks das oberste Element jeweils entfernt.

- Ein Block ist eine Abstraktion, die mehrere Warps zusammenfasst.
- Bei CUDA-Programmen werden Blöcke konfiguriert, die dann durch den jeweiligen Warp-Scheduler auf einzelne Warps aufgeteilt werden, die sukzessive zur Ausführung kommen.
- Ein Block läuft immer nur auf einem Multiprozessor bietet einen gemeinsamen Speicherbereich für alle zugehörigen Threads an.
- Threads eines Blockes können sich untereinander synchronisieren und über den gemeinsamen Speicher kommunizieren.
- Ein Block kann (bei uns auf Livingstone) bis zu 32 Warps bzw. 1024 Threads umfassen.

vector.cu

```
__global__ void add(double alpha, double* x, double* y) {
   std::size_t tid = threadIdx.x;
   y[tid] += alpha * x[tid];
}
```

- Die **for**-Schleife ist entfallen. Stattdessen führt die Kernel-Funktion jetzt nur noch eine einzige Addition durch.
- Mit *threadIdx.x* erfahren wir hier, der wievielte Thread (beginnend ab 0) unseres Blocks wir sind.
- Die Kernel-Funktion wird dann für jedes Element aufgerufen, wobei dies im Rahmen der Möglichkeiten parallelisiert wird.

vector.cu

```
/* execute kernel function on GPU */
add<<<1, N>>>(2.0, device_a, device_b);
```

- Beim Aufruf der Kernel-Funktion wurde jetzt die Konfiguration verändert.
- $<<<1, N>>>$ bedeutet jetzt, dass ein Block mit N Threads gestartet wird.
- Entsprechend starten die einzelnen Threads mit Werten von 0 bis $N-1$ für *threadIdx.x*.
- Da die Zahl der Threads pro Block beschränkt ist, kann N hier nicht beliebig groß werden.

vector.ptx

```
        ld.param.f64    %fd1, [_Z3adddPdS__param_0];
        ld.param.u64    %rd1, [_Z3adddPdS__param_1];
        ld.param.u64    %rd2, [_Z3adddPdS__param_2];
        cvta.to.global.u64      %rd3, %rd2;
        cvta.to.global.u64      %rd4, %rd1;
        mov.u32         %r1, %tid.x;
        mul.wide.u32    %rd5, %r1, 8;
        add.s64         %rd6, %rd4, %rd5;
        ld.global.f64   %fd2, [%rd6];
        add.s64         %rd7, %rd3, %rd5;
        ld.global.f64   %fd3, [%rd7];
        fma.rn.f64      %fd4, %fd2, %fd1, %fd3;
        st.global.f64   [%rd7], %fd4;
        ret;
```

- PTX steht für *Parallel Thread Execution* und ist eine Assembler-Sprache für einen virtuellen GPU-Prozessor. Dies ist die erste Zielsprache des Übersetzers für den für die GPU bestimmten Teil.

- PTX steht für *Parallel Thread Execution* und ist eine Assembler-Sprache für einen virtuellen GPU-Prozessor. Dies ist die erste Zielsprache des Übersetzers für den für die GPU bestimmten Teil.
- Der PTX-Instruktionssatz ist öffentlich: `http://docs.nvidia.com/cuda/pdf/ptx_isa_6.1.pdf`
- PTX wurde entwickelt, um eine portable vom der jeweiligen Grafikkarte unabhängige virtuelle Maschine zu haben, die ohne größeren Aufwand effizient für die jeweiligen GPUs weiter übersetzt werden kann.
- PTX lässt sich mit Hilfe des folgenden Kommandos erzeugen:

```
nvcc -o vector.ptx --ptx --gpu-architecture compute_60
vector.cu
```

vector.disas

```
/*0008*/                      MOV R1, c[0x0][0x20] ;
/*0010*/                      S2R R0, SR_TID.X ;
/*0018*/                      SHL R4, R0.reuse, 0x3 ;
/*0028*/                      SHR.U32 R0, R0, 0x1d ;
/*0030*/                      IADD R6.CC, R4, c[0x0][0x148] ;
/*0038*/                      IADD.X R7, R0, c[0x0][0x14c] ;
/*0048*/           {          IADD R4.CC, R4, c[0x0][0x150] ;
/*0050*/                      LDG.E.64 R2, [R6]        }
/*0058*/                      IADD.X R5, R0, c[0x0][0x154] ;
/*0068*/                      LDG.E.64 R8, [R4] ;
/*0070*/                      DFMA R2, R2, c[0x0][0x140], R8 ;
/*0078*/                      STG.E.64 [R4], R2 ;
/*0088*/                      EXIT ;
```

- Mit Hilfe von *ptxas* kann der PTX-Text für eine konkrete GPU in Maschinen-Code übersetzt werden (CUBIN):
  `ptxas --output-file vector.cubin --gpu-name sm_60 vector.ptx`

- In neueren Versionen stellt Nvidia mit *cuobjdump* ein Werkzeug zur Verfügung, der den CUBIN-Code wieder disassembliert:
  ```
  cuobjdump --dump-sass vector.cubin >vector.disas
  ```
- Bei uns auf Livingstone ist in der zugehörigen Dokumentation die Übersicht zu Pascal relevant.

bigvector.cu

```
unsigned int max_threads_per_block() {
   int device; CHECK_CUDA(cudaGetDevice, &device);
   struct cudaDeviceProp device_prop;
   CHECK_CUDA(cudaGetDeviceProperties, &device_prop, device);
   return device_prop.maxThreadsPerBlock;
}
```

- Wenn die Zahl der Elemente größer ist als die maximale Zahl der Threads pro Block, wird es notwendig, die Aufgabe auf mehrere Blöcke aufzusplitten.

bigvector.cu

```
/* execute kernel function on GPU */
size_t max_threads = hpc::cuda::get_max_threads_per_block();
if (N <= max_threads) {
   add<<<1, N>>>(2.0, device_a, device_b);
} else {
   add<<<(N+max_threads-1)/max_threads, max_threads>>>(2.0,
      device_a, device_b);
}
```

bigvector.cu

```
__global__ void add(double alpha, double* x, double* y) {
   std::size_t tid = threadIdx.x + blockIdx.x * blockDim.x;
   if (tid < N) {
      y[tid] += alpha * x[tid];
   }
}
```

- *threadIdx.x* gibt jetzt nur noch den Thread-Index innerhalb eines Blocks an und ist somit alleine nicht mehr geeignet, die eindeutige Thread-ID im Bereich von 0 bis $N - 1$ zu bestimmen.
- *blockIdx.x* liefert jetzt zusätzlich noch den Block-Index und *blockDim.x* die Zahl der Threads pro Block.
- Wenn N nicht durch *max_threads* teilbar ist, dann erhalten wir Threads, die nichts zu tun haben. Um einen Zugriff außerhalb der Array-Grenzen zu vermeiden, benötigen wir eine entsprechende **if**-Anweisung.

Konfiguration eines Kernels

In der allgemeinen Form akzeptiert die Konfiguration vier Parameter. Davon sind die beiden letzten optional:
<<< *Dg*, *Db*, *Ns*, *S* >>>

- *Dg* legt die Dimensionierung des Grids fest (ein- oder zwei- oder dreidimensional).
- *Db* legt die Dimensionierung eines Blocks fest (ein-, zwei- oder dreidimensional).
- *Ns* legt den Umfang des gemeinsamen Speicherbereichs per Block fest (per Voreinstellung 0 bzw. vom Übersetzer bestimmt).
- *S* erlaubt die Verknüpfung mit einem Stream (per Voreinstellung keine).
- *Dg* und *Db* sind beide vom Typ *dim3*, der mit eins bis drei ganzen Zahlen initialisiert werden kann.
- Vorgegebene Beschränkungen sind bei der Dimensionierung zu berücksichtigen. Sonst kann der Kernel nicht gestartet werden.

In den auf der GPU laufenden Funktionen stehen spezielle Variablen zur Verfügung, die die Identifizierung bzw. Einordnung des eigenen Threads ermöglichen:

Variable	Bedeutung
threadIdx.x	x-Koordinate innerhalb des Blocks
threadIdx.y	y-Koordinate innerhalb des Blocks
threadIdx.z	z-Koordinate innerhalb des Blocks
blockDim.x	Dimensionierung des Blocks für x
blockDim.y	Dimensionierung des Blocks für y
blockDim.z	Dimensionierung des Blocks für z
blockIdx.x	x-Koordinate innerhalb des Gitters
blockIdx.y	y-Koordinate innerhalb des Gitters
blockIdx.z	z-Koordinate innerhalb des Gitters
gridDim.x	Dimensionierung des Gitters für x
gridDim.y	Dimensionierung des Gitters für y
gridDim.z	Dimensionierung des Gitters für z

simpson.cu

```
// to be integrated function
__device__ double f(double x) {
   return 4 / (1 + x*x);
}

// numerical integration according to the Simpson rule
// for f over the i-th subinterval of [a,b]
__global__ void simpson(double a, double b, double* sums) {
   const std::size_t N = blockDim.x * gridDim.x;
   const std::size_t i = threadIdx.x + blockIdx.x * blockDim.x;
   double xleft = a + (b - a) / N * i;
   double xright = xleft + (b - a) / N;
   double xmid = (xleft + xright) / 2;
   sums[i] = (xright - xleft) / 6 * (f(xleft) + 4 * f(xmid) + f(xright));
}
```

- Die Kernel-Funktion berechnet hier jeweils nur ein Teilintervall und ermittelt mit Hilfe von *blockDim.x* ∗ *gridDim.x*, wieviel Teilintervalle es gibt.
- Funktionen, die auf der GPU nur von anderen GPU-Funktionen aufzurufen sind, werden mit dem Schlüsselwort **__device__** gekennzeichnet.

simpson.cu

```
double* device_sums;
CHECK_CUDA(cudaMalloc, (void**)&device_sums, N * sizeof(double));
simpson<<<nof_blocks, blocksize>>>(a, b, device_sums);

double sums[N];
CHECK_CUDA(cudaMemcpy, sums, device_sums,
   N * sizeof(double), cudaMemcpyDeviceToHost);
CHECK_CUDA(cudaFree, device_sums);
double sum = 0;
for (std::size_t i = 0; i < N; ++i) {
   sum += sums[i];
}
```

- Es gibt keine vorgegebenen Aggregierungs-Operatoren, so dass diese „von Hand“ durchgeführt werden müssen.

simpson2.cu

```
constexpr std::size_t THREADS_PER_BLOCK = 1024; // must be a power of 2

// numerical integration according to the Simpson rule
// for f over the i-th subinterval of [a,b]
__global__ void simpson(double a, double b, double* sums) {
   /* compute approximative sum for our sub-interval */
   const std::size_t N = blockDim.x * gridDim.x;
   const std::size_t i = threadIdx.x + blockIdx.x * blockDim.x;
   // ...
   double sum = (xright - xleft) / 6 * (f(xleft) +
      4 * f(xmid) + f(xright));

   /* store it into the per-block shared array of sums */
   std::size_t me = threadIdx.x;
   __shared__ double sums_per_block[THREADS_PER_BLOCK];
   sums_per_block[me] = sum;

   // ...
}
```

- Innerhalb eines Blocks ist eine Synchronisierung und die Nutzung eines gemeinsamen Speicherbereiches möglich.
- Das eröffnet die Möglichkeit der blockweisen Aggregierung.

simpson2.cu

```
/* store it into the per-block shared array of sums */
unsigned int me = threadIdx.x;
__shared__ double sums_per_block[THREADS_PER_BLOCK];
sums_per_block[me] = sum;
```

- Mit **__shared__** können Variablen gekennzeichnet werden, die allen Threads eines Blocks gemeinsam zur Verfügung stehen.
- Die Zugriffe auf diese Bereiche sind schneller als auf den globalen GPU-Speicher.
- Allerdings ist die Kapazität begrenzt. Auf Livingstone stehen nur 48 KiB zur Verfügung.

simpson2.cu

```
/* aggregate sums within a block */
std::size_t index = blockDim.x / 2;
while (index) {
   __syncthreads();
   if (me < index) {
      sums_per_block[me] += sums_per_block[me + index];
   }
   index /= 2;
}
/* publish result */
if (me == 0) {
   sums[blockIdx.x] = sums_per_block[0];
}
```

- Zwar werden die jeweils einzelnen Gruppen eines Blocks zu Warps zusammengefasst, die entsprechend der SIMD-Architektur synchron laufen, aber das umfasst nicht den gesamten Block.
- Wenn alle Threads eines globalen Blocks synchronisiert werden sollen, geht dies mit der Funktion *__syncthreads*, die den aufrufenden Thread solange blockiert, bis alle Threads des Blocks diese Funktion aufrufen.