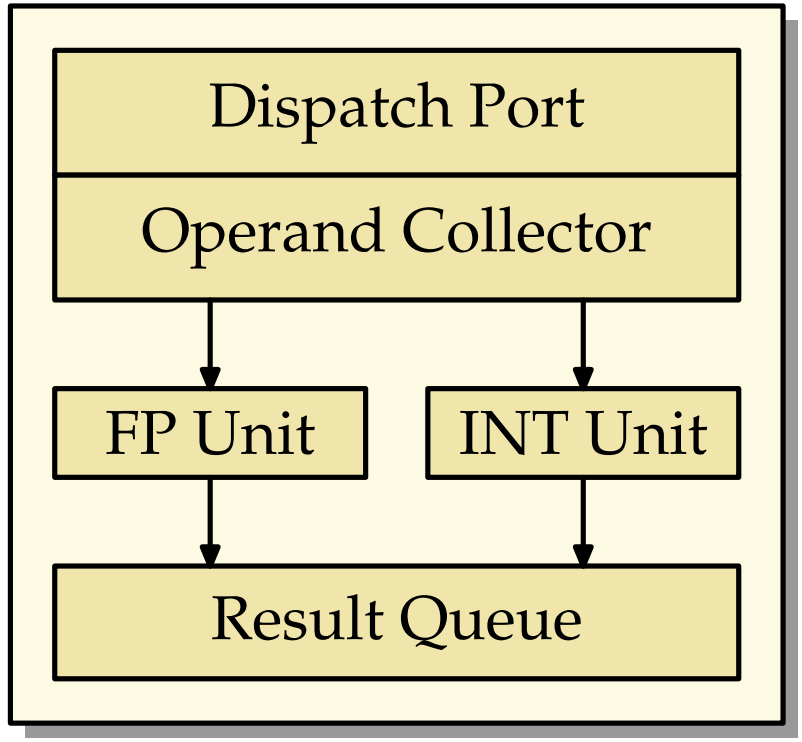
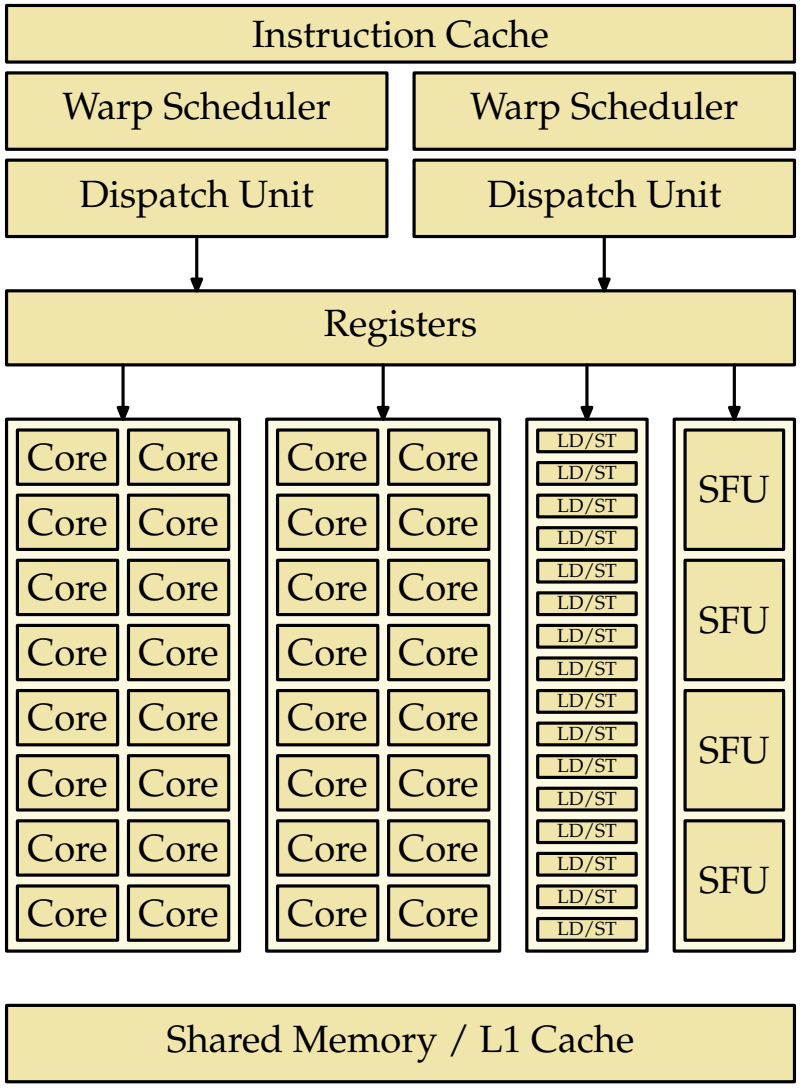
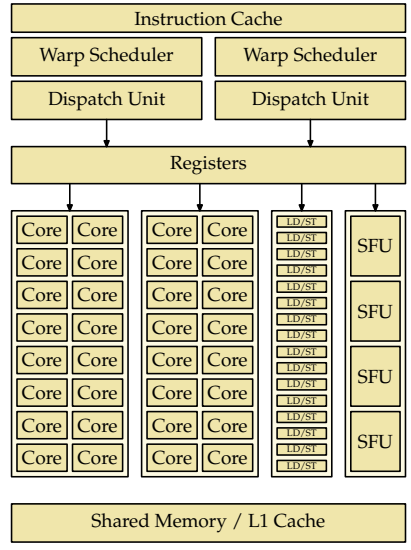
- Schon sehr früh gab es diverse Grafik-Beschleuniger, die der normalen CPU Arbeit abnahmen.
- Die im März 2001 von Nvidia eingeführte GeForce 3 Series führte programmierbares Shading ein.
- Im August 2002 folgte die Radeon R300 von ATI, die die Fähigkeiten der GeForce 3 deutlich erweiterte um mathematische 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) und CUDA (Compute Unified Device Architecture, von Nvidia). Wir beschäftigen uns hier mit CUDA, da es zur Zeit die größte Popularität genießt und bei uns auch zur Verfügung steht.



- 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

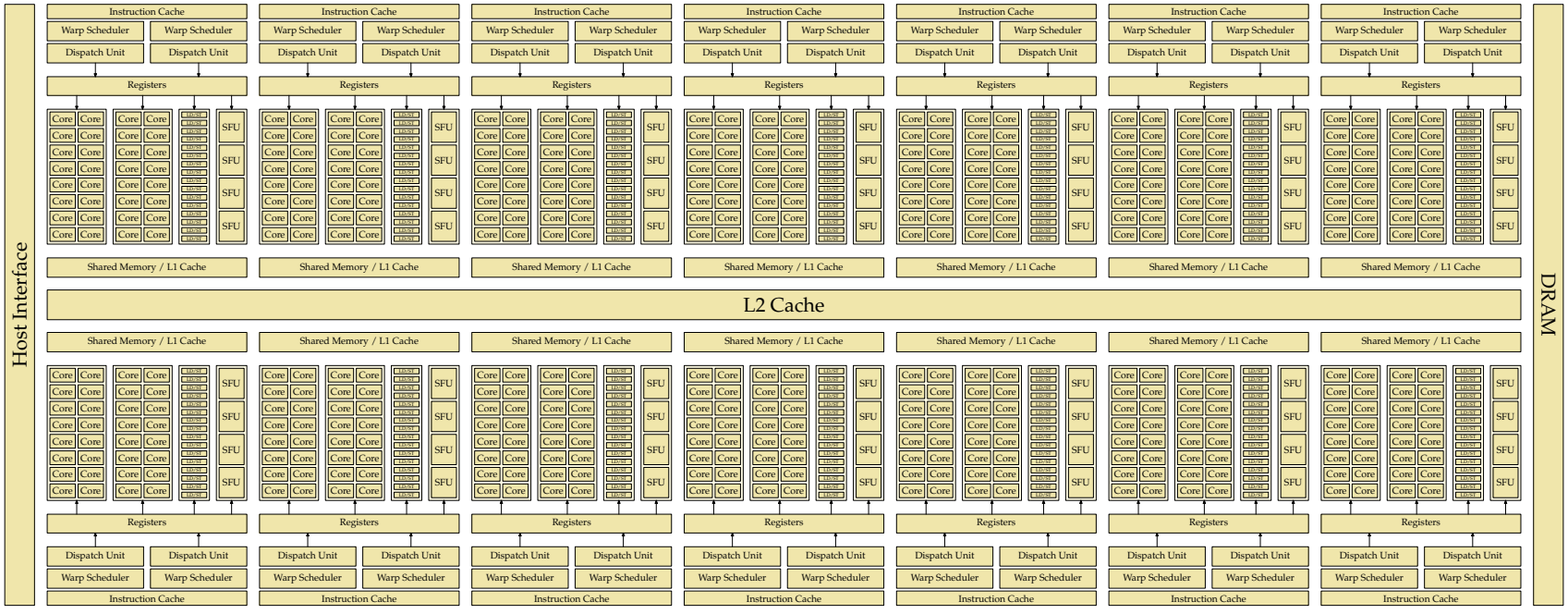




- Je nach Ausführung der GPU werden zahlreiche Kerne und weitere Komponenten zu einem Multiprozessor zusammengefasst.
- Auf Olympia hat ein Multiprozessor 32, auf Hochwanner 48 Kerne.
- Hinzu kommen Einheiten zum parallelisierten Laden und Speichern (*LD/ST*) und Einheiten für spezielle Operationen wie beispielsweise *sin* oder *sqrt* (*SFU*).

- 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.

GPU mit mehreren Multiprozessoren



- Je nach Ausführung der GPU werden mehrere Multiprozessoren zusammengefasst.
- Olympia bietet 14 Multiprozessoren mit insgesamt 448 Kernen an. Bei Hochwanner sind es nur 2 Multiprozessoren mit insgesamt 96 Kernen.

- 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 und hat Zugriff auf gemeinsamen Speicher.
- Threads eines Blockes können sich untereinander synchronisieren und über den gemeinsamen Speicher kommunizieren.
- Ein Block kann (bei uns auf Olympia und Hochwanner) bis zu 32 Warps bzw. 1024 Threads umfassen.

- Drei Speicherbereiche stehen zur Verfügung:
  - Lokaler Speicher: Steht nur einem Thread zur Verfügung.
  - Gemeinsamer Speicher: Steht den Threads eines Blocks gemeinsam zur Verfügung. (48 KiB pro Block auf Olympia und Hochwanner.)
  - Globaler Speicher: Steht allen Threads einer GPU zur Verfügung. (1 GiB auf Hochwanner, ca. 1,2 GiB auf Olympia.)
- Ab CUDA-Level 2.0 liegen alle drei Speicherbereiche im gleichen Adressraum.
- Neuere Implementierungen erlauben es, Teile des Hauptspeichers in den GPU-Adressraum abzubilden. Das hat sowohl Vor- als auch Nachteile. Die Entwicklung geht in Richtung einer besseren Integration von CPUs und GPUs mit dem Ziel, den Speicher gemeinsam effizienter und kooperativer zu nutzen.

- Wenn ein Warp auf den globalen Speicher zugreift, dann wird die höchste Effizienz erreicht, wenn die einzelnen Speicherzugriffe konsekutiv erfolgen.
- Wenn die Zugriffe nicht konsekutiv sind, erhöht sich die Zahl der zu ladenden Cache-Lines und die Bandbreite wird entsprechend reduziert, wodurch die Ausführung sich verzögert.
- Bei dem gemeinsamen und lokalen Speicher spielt das keine Rolle.
- Wenn bei einer Matrix die zu ladenden Vektoren ungünstig liegen (etwa Spaltenvektoren bei einer *row major*-Ordnung), dann kann es sich lohnen, den zu bearbeitenden Teil konsekutiv zu laden und im gemeinsamen Speicher abzulegen. (Siehe später folgendes Beispiel *mmu.cu*).

- Jeder Warp hat entweder den Zustand *active* oder *waiting*.
- Wenn ein Warp auf eine Barrier-Instruktion stößt, wechselt der Zustand von *active* auf *waiting*.
- Warps, die das Ausführungsende erreichen, verbleiben im Zustand *inactive*.
- Wenn ein Block keine aktiven Warps mehr hat und einige davon wegen einer Barrier-Instruktion warten, dann wechseln diese Warps von *waiting* auf *active*.

- Der Instruktionssatz ist proprietär und bis heute wurde von Nvidia kein öffentliches Handbuch dazu herausgegeben.
- Dank Wladimir J. van der Laan ist dieser jedoch weitgehend dekodiert und es gibt sogar einen an der Université de Perpignan entwickelten Simulator.
- 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.

- Zunächst wurden nur ganzzahlige Datentypen (32 Bit) und Gleitkommazahlen (**float**) unterstützt.
- Erst ab Level 1.3 kam die Unterstützung von **double** hinzu. Die GeForce GTX 470 auf Olympia unterstützt Level 2.0, die Quadro 600 auf Hochwanner unterstützt Level 2.1. (Beim Übersetzen mit *nvcc* sollte immer die Option „–gpu-architecture compute_20“ angegeben werden.)
- Ferner werden Zeiger unterstützt.
- Zugriffe sind (auch per Zeiger ab Level 2.0) möglich auf den gemeinsamen Speicher der GPU (*global memory*), auf den gemeinsamen Speicher eines Blocks (*shared memory*) und auf lokalen Speicher.

- Anwendungen, die GPUs ausnutzen möchten, werden zunächst auf der CPU gestartet.
- Eine Anwendung fällt somit in einen Teil, der auf der regulären CPU läuft und einen Teil, der von der GPU verarbeitet wird.
- Beide Teile haben völlig verschiedene Architekturen und Instruktionssätze.
- Die GPU ist der Anwendung nicht direkt zugänglich, sondern nur über einen speziellen Geräte-Treiber (unter Linux */dev/nvidia0* und */dev/nvidiactl*), der das Laden von Programmen, die Konfiguration eines Programmlaufs und den Austausch von Daten ermöglicht.

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 bzw. C++ (CUDA C bzw. 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/bin*), der CUDA C bzw. CUDA C++ unterstützt,
- eine zugehörige Laufzeitbibliothek (*libcudart.so* in */usr/local/cuda/lib*) und
- darauf aufbauende Bibliotheken (einschließlich BLAS und FFT).

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

vecadd.cu

```
__global__ void VecAdd(float* a, float* b, float* c) {
   int i = threadIdx.x;
   c[i] = a[i] + b[i];
}
```

- Der Übersetzer *nvcc* muss nach der statischen und semantischen Analyse vor der Code-Generierung eine Aufteilung entsprechend der Zielarchitektur durchführen, je nachdem ob der Programmtext für die GPU oder die reguläre CPU bestimmt ist.
- *VecAdd* ist ein Beispiel für eine Funktion, die für die GPU bestimmt ist. In CUDA wird eine Funktion, die von der CPU aufrufbar ist, jedoch auf der GPU ausgeführt wird, mit dem Schlüsselwort **__global__** gekennzeichnet. (Solche Funktionen werden *kernel* genannt.)
- *threadIdx.x* liefert hier die Thread-Nummer (mehr dazu später).

vecadd.ptx

```
        cvt.s32.u16     %r1, %tid.x;
        cvt.u64.s32     %rd1, %r1;
        mul.lo.u64      %rd2, %rd1, 4;
        ld.param.u64    %rd3, [__cudaparm__Z6VecAddPfS_S__a];
        add.u64         %rd4, %rd3, %rd2;
        ld.global.f32   %f1, [%rd4+0];
        ld.param.u64    %rd5, [__cudaparm__Z6VecAddPfS_S__b];
        add.u64         %rd6, %rd5, %rd2;
        ld.global.f32   %f2, [%rd6+0];
        add.f32         %f3, %f1, %f2;
        ld.param.u64    %rd7, [__cudaparm__Z6VecAddPfS_S__c];
        add.u64         %rd8, %rd7, %rd2;
        st.global.f32   [%rd8+0], %f3;
```

- 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.
- Die PTX-Instruktionssatz ist öffentlich:

  `http://docs.nvidia.com/cuda/pdf/ptx_isa_3.1.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.

vecadd.cubin.dis

```
000000: a0000001 04000780 cvt.rn.u32.u16 $r0, $r0.lo
000008: 30020009 c4100780 shl.u32 $r2, $r0, 0x00000002
000010: 2102e800          add.half.b32 $r0, s[0x0010], $r2
000014: 2102ec0c          add.half.b32 $r3, s[0x0018], $r2
000018: d00e0005 80c00780 mov.u32 $r1, g[$r0]
000020: d00e0601 80c00780 mov.u32 $r0, g[$r3]
000028: b0000204          add.half.rn.f32 $r1, $r1, $r0
00002c: 2102f000          add.half.b32 $r0, s[0x0020], $r2
000030: d00e0005 a0c00781 mov.end.u32 g[$r0], $r1
```

- Mit *ptxas* (wird normalerweise von *nvcc* implizit aufgerufen) lässt sich PTX-Assembler für eine vorgegebene GPU übersetzen.
- Der Instruktionssatz der GPU-Architektur ist bei Nvidia proprietär und wurde bislang nicht veröffentlicht.
- Obiger Text wurde mit Hilfe des von Wladimir J. van der Laan entwickelten Disassemblers erzeugt:
  `http://wiki.github.com/laanwj/decuda/`
- *$r0* ist ein Register, *s*[0*x0010*] adressiert den gemeinsamen Speicher des Blocks (*shared memory*) und *g*[*$ro*] eine mit einem Register indizierte Zelle im globalen Speicher der GPU (*global memory*).

Im Normalfall wird von *nvcc* ein C- bzw. C++-Programmtext erzeugt, der

- aus dem regulären für die CPU bestimmten Programmtext besteht,
- die für die GPU erzeugten CUBIN-Code als Daten-Arrays enthält und
- mit zahlreichen Aufrufen der Laufzeitbibliothek ergänzt wurde, die den Datenaustauch vornehmen, den CUBIN-Code in die GPU laden und zur Ausführung bringen (dies erfolgt mit Hilfe des entsprechenden Gerätetreibers).

Dieser C- bzw. C++-Code kann dann ganz regulär weiter übersetzt werden. Somit liefert *nvcc* im Normalfall am Ende ein fertiges, alleinstehendes ausführbares Programm, das alle benötigten Teile enthält.

vecadd.cu

```
float* cuda_a; cudaMalloc((void**)&cuda_a, N * sizeof(float));
float* cuda_b; cudaMalloc((void**)&cuda_b, N * sizeof(float));
float* cuda_c; cudaMalloc((void**)&cuda_c, N * sizeof(float));
cudaMemcpy(cuda_a, a, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(cuda_b, b, N * sizeof(float), cudaMemcpyHostToDevice);
VecAdd<<<1, N>>>(cuda_a, cuda_b, cuda_c);
cudaMemcpy(c, cuda_c, N * sizeof(float), cudaMemcpyDeviceToHost);
cudaFree(cuda_a); cudaFree(cuda_b); cudaFree(cuda_c);
```

- Die GPU und die CPU haben getrennten Speicher.
- Mit der Funktion *cudaMalloc* kann auf der GPU-Speicher belegt werden (*global memory*). Dieser wird nicht initialisiert.
- Der globale GPU-Speicher bleibt über den gesamten Programmlauf hinweg persistent, d.h. auch über mehrere Kernel-Aufrufe hinweg.
- Mit *cudaMemcpy* können Daten von der CPU in die GPU oder zurück kopiert werden.
- Mit *cudaFree* kann der GPU-Speicher wieder freigegeben werden.

vecadd.cu

```
VecAdd<<<1, N>>>(cuda_a, cuda_b, cuda_c);
```

- Jeder Aufruf eines Kernels ist mit einer Konfiguration der GPU verbunden, die in <<< ... >>> gefasst wird.
- Im einfachsten Falle enthält die Konfiguration zwei Parameter: Die Zahl der Blöcke (hier 1) und die Zahl der Threads pro Block (hier N).
- Die Zahl der Threads pro Block darf das von der jeweiligen GPU gesetzte Limit nicht überschreiten. Dies ist bei uns 1024.
- Alle Kernel-Parameter sollten entweder elementar sein (also etwa **int** oder **double**), Zeiger auf den globalen Speicher der GPU oder kleine Strukturen, die daraus bestehen. (Neuere CUDA-Versionen akzeptieren auch C++-Klassen.)

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 zweidimensional). (Bei neueren Versionen auch 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).
- *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 Beschänkungen sind bei der Dimensionierung zu berücksichtigen. Sonst kann der Kernel nicht gestartet werden.

simpson.cu

```
// numerical integration according to the Simpson rule
__global__ void simpson(Real a, Real b, Real* sums) {
   const int N = get_nofthreads();
   const int i = get_id();
   Real xleft = a + (b - a) / N * i;
   Real xright = xleft + (b - a) / N;
   Real xmid = (xleft + xright) / 2;
   sums[i] = (xright - xleft) / 6 * (f(xleft) + 4 * f(xmid) + f(xright));
}
```

- Bei der Vielzahl möglicher Threads kann häufig auf den Einsatz von Schleifen verzichtet werden.
- Aggregierende Funktionen existieren nicht. Deswegen ist es sinnvoll, die Einzelresultate im globalen Speicher abzulegen.
- *Real* wurde hier per **typedef** definiert (normalerweise **double**, bei älteren Karten muss ggf. **float** verwendet werden).

simpson.cu

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

- Nur mit **___device___** gekennzeichnete Funktionen dürfen auf der Seite der GPU aufgerufen werden.
- Es stehen auch diverse mathematischen Funktionen zur Verfügung.

In den auf der GPU laufenden Funktionen stehen spezielle Variablen zur Verfügung, die die Identifizierung bzw. Einordnung des eigenen Threads ermöglichen im bis zu drei-dimensional strukturierten Block und dem maximal zweidimensionalen Gitter von Blocks:

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
gridDim.x	Dimensionierung des Gitters für x
gridDim.y	Dimensionierung des Gitters für y

Identifizierung des eigenen Threads

simpson.cu

```
/* return unique id within a block */
__device__ int get_threadid() {
   return threadIdx.z * blockDim.x * blockDim.y +
  threadIdx.y * blockDim.x +
  threadIdx.x;
}

/* return block id a thread is associated to */
__device__ int get_blockid() {
   return blockIdx.x + blockIdx.y * gridDim.x;
}

/* return number of threads per block */
__device__ int get_nofthreads_per_block() {
   return blockDim.x * blockDim.y * blockDim.z;
}

/* return number of blocks */
__device__ int get_nofblocks() {
   return gridDim.x * gridDim.y;
}

/* return total number of threads */
__device__ int get_nofthreads() {
   return get_nofthreads_per_block() * get_nofblocks();
}

/* return id which is unique throughout all threads */
__device__ int get_id() {
   return get_blockid() * blockDim.x * blockDim.y * blockDim.z +
          get_threadid();
}
```

simpson.cu

```
int blocksize = max_threads_per_block();
if (blocksize > N) {
   blocksize = N;
} else {
   if (N % blocksize != 0) {
      cerr << cmdname << ": please select a multiple of "
         << blocksize << endl;
      exit(1);
   }
}
int nof_blocks = N / blocksize;
dim3 blockdim(blocksize, 1, 1);
dim3 griddim(nof_blocks, 1);
// ...
simpson<<<griddim, blockdim>>>(a, b, cuda_sums);
```

- Bei dem Simpson-Verfahren ist es sinnvoll, sowohl das Gitter als auch jeden Block eindimensional zu strukturieren.

simpson.cu

```
Real sums[N];
Real* cuda_sums; cudaMalloc((void**)&cuda_sums, N * sizeof(Real));
simpson<<<griddim, blockdim>>>(a, b, cuda_sums);
cudaMemcpy(sums, cuda_sums, N * sizeof(Real), cudaMemcpyDeviceToHost);
cudaFree(cuda_sums);
double sum = 0;
for (int i = 0; i < N; ++i) {
   sum += sums[i];
}
```

- Hier wird das Feld mit Summen zu Beginn im globalen Speicher der GPU belegt, dann mit der *simpson*-Funktion gefüllt, danach zum Speicher der CPU kopiert und schließlich aufsummiert.

jacobi.cu

```
typedef Real Matrix[BLOCK_SIZE+2][BLOCK_SIZE+2];

__global__ void jacobi(Matrix A, int nofiterations) {
   int i = threadIdx.x + 1;
   int j = threadIdx.y + 1;
   for (int it = 0; it < nofiterations; ++it) {
      Real Aij = 0.25 * (A[i-1][j] + A[i][j-1] + A[i][j+1] + A[i+1][j]);
      __syncthreads();
      A[i][j] = Aij;
      __syncthreads();
   }
}
```

- Beim Jacobi-Verfahren bietet sich eine zweidimensionale Organisation eines Blocks an.
- Es ist hierbei darauf zu achten, dass das Quadrat von *BLOCK_SIZE* noch kleiner als 512 ist. Hierfür bieten sich 16 (Zweier-Potenz) und 22 (maximaler Wert) an.

jacobi.cu

```
for (int it = 0; it < nofiterations; ++it) {
   Real Aij = 0.25 * (A[i-1][j] + A[i][j-1] + A[i][j+1] + A[i+1][j]);
   __syncthreads();
   A[i][j] = Aij;
   __syncthreads();
}
```

- Beim Jacobi-Verfahren werden bekanntlich die letzten Werte der Nachbarn eingeholt.
- Dies muss synchronisiert erfolgen. Solange alles in einen Warp passen würde, wäre das kein Problem, aber die Warps können unterschiedlich schnell vorankommen.
- Mit einem Aufruf von *__syncthreads()* wird eine Synchronisierung aller Threads eines Blocks erzwungen. D.h. erst wenn alle den Aufruf erreicht haben, geht es für alle weiter.

- Matrix-Matrix-Multiplikationen sind hochgradig parallelisierbar.
- Bei der Berechung von $C = A * B$ kann beispielsweise die Berechnung von $c_{i,j}$ an einen einzelnen Thread delegiert werden.
- Da größere Matrizen nicht mehr in einen Block (mit bei uns maximal 1024 Threads) passen, ist es sinnvoll, die gesamte Matrix in Blocks zu zerlegen.
- Dazu bieten sich 16×16 Blöcke mit 256 Threads an.
- O.B.d.A. betrachten wir nur quadratische $N \times N$ Matrizen mit $16 \mid N$.

mmm.cu

```
int main(int argc, char** argv) {
   cmdname = *argv++; --argc;
   if (argc != 2) usage();
   Matrix A; if (!read_matrix(*argv++, A)) usage(); --argc;
   Matrix B; if (!read_matrix(*argv++, B)) usage(); --argc;
   cout << "A = " << endl << A << endl;
   cout << "B = " << endl << B << endl;
   if (A.N != B.N) {
      cerr << cmdname << ": sizes of the matrices do not match" << endl;
      exit(1);
   }
   if (A.N % BLOCK_SIZE) {
      cerr << cmdname << ": size of matrices is not a multiply of "
         << BLOCK_SIZE << endl;
      exit(1);
   }

   A.copy_to_gpu();
   B.copy_to_gpu();
   Matrix C; C.resize(A.N); C.allocate_cuda_data();
   dim3 block(BLOCK_SIZE, BLOCK_SIZE);
   dim3 grid(A.N / BLOCK_SIZE, A.N / BLOCK_SIZE);

   mmm<<<grid, block>>>(A.cuda_data, B.cuda_data, C.cuda_data);

   C.copy_from_gpu();
   cout << "C = " << endl << setprecision(14) << C << endl;
}
```

- Es ist sinnvoll, eine Klasse für Matrizen zu verwenden, die die Daten sowohl auf der CPU als auch auf der GPU je nach Bedarf hält.
- Diese Klasse kann dann auch das Kopieren der Daten unterstützen.
- Generell ist es sinnvoll, Kopieraktionen soweit wie möglich zu vermeiden, indem etwa Zwischenresultate nicht unnötig von der GPU zur CPU kopiert werden.
- Eine Klasse hat auch den Vorteil, dass das die Freigabe der Datenflächen automatisiert wird.

mmm.cu

```
struct Matrix {
   unsigned int N;
   Real* data;
   bool cuda_allocated;
   Real* cuda_data;

   Matrix() :
      N(0), data(0), cuda_allocated(false), cuda_data(0) {
   }
   ~Matrix() {
      if (data) delete data;
      if (cuda_allocated) release_cuda_data();
   }

   // ...
};
```

- *N* ist die Größe der Matrix, *data* der Zeiger in den Adressraum der CPU, *cuda_data* der Zeiger in den Adressraum der GPU.

mmm.cu

```
bool copy_to_gpu() {
   if (!cuda_allocated) {
      if (!allocate_cuda_data()) return false;
   }
   return cudaMemcpy(cuda_data, data, N * N * sizeof(Real),
            cudaMemcpyHostToDevice) == cudaSuccess;
}

bool copy_from_gpu() {
   assert(cuda_allocated);
   return cudaMemcpy(data, cuda_data, N * N * sizeof(Real),
            cudaMemcpyDeviceToHost) == cudaSuccess;
}
```

- *copy_to_gpu* und *copy_from_gpu* kopieren die Matrix zur GPU und zurück.

mmm.cu

```
bool allocate_cuda_data() {
   if (cuda_allocated) return true;
   Real* cudap;
   if (cudaMalloc((void**)&cudap, N * N * sizeof(Real)) !=
         cudaSuccess) {
      return false;
   }
   cuda_data = cudap;
   cuda_allocated = true;
   return true;
}

void release_cuda_data() {
   if (cuda_data) {
      cudaFree(cuda_data);
      cuda_data = 0;
   }
}
```

- Mit *allocate_cuda_data* wird die Matrix im Adressraum der GPU belegt, mit *release_cuda_data* wieder freigegeben.

mmm.cu

```
bool resize(unsigned int N_) {
   if (N == N_) return true;
   Real* rp = new Real[N_ * N_];
   if (!rp) return false;
   if (data) delete data;
   release_cuda_data();
   data = rp; N = N_;
   return true;
}

Real& operator()(unsigned int i, unsigned int j) {
   return data[i*N + j];
}

const Real& operator()(unsigned int i, unsigned int j) const {
   return data[i*N + j];
}
```

- Mit *resize* wird die Größe festgelegt bzw. verändert. Die beiden ()-Operatoren dienen dem indizierten Zugriff (auf der Seite der CPU).

mmm-ab.cu

```
#define ELEMENT(m,i,j) ((m)[(i) * stride + (j)])

__global__ void mmm(Real* a, Real* b, Real* c) {
   unsigned int stride = gridDim.y * BLOCK_SIZE;
   unsigned int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
   unsigned int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;

   Real sum = 0;
   for (int k = 0; k < BLOCK_SIZE * gridDim.y; ++k) {
      sum += ELEMENT(a, row, k) * ELEMENT(b, k, col);
   }
   ELEMENT(c, row, col) = sum;
}
```

- Dies ist die triviale Implementierung, bei der jeder Thread $c_{row,col}$ direkt berechnet.
- Der Zugriff auf *a* ist hier ineffizient, da ein Warp hier nicht auf konsekutiv im Speicher liegende Werte zugreift.

mmm.cu

```
__shared__ Real ablock[BLOCK_SIZE][BLOCK_SIZE];
```

- Wenn kein konsekutiver Zugriff erfolgt, kann es sich lohnen, dies über Datenstruktur abzuwickeln, die allen Threads eines Blocks gemeinsam ist.
- Die Idee ist, dass dieses Array gemeinsam von allen Threads eines Blocks konsekutiv gefüllt wird.
- Der Zugriff auf das gemeinsame Array ist recht effizient und muss nicht mehr konsekutiv sein.
- Die Matrix-Matrix-Multiplikation muss dann aber blockweise organisiert werden.

Blockweise Multiplikation

mmm.cu

```
#define ELEMENT(m,i,j) ((m)[(i) * stride + (j)])

__global__ void mmm(Real* a, Real* b, Real* c) {
   __shared__ Real ablock[BLOCK_SIZE][BLOCK_SIZE];
   unsigned int stride = gridDim.y * BLOCK_SIZE;
   unsigned int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
   unsigned int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;

   Real sum = 0;
   for (int round = 0; round < gridDim.y; ++round) {
      ablock[threadIdx.y][threadIdx.x] =
         ELEMENT(a, row, round*BLOCK_SIZE + threadIdx.x);
      __syncthreads();

      #pragma unroll
      for (int k = 0; k < BLOCK_SIZE; ++k) {
         sum += ablock[threadIdx.y][k] *
            ELEMENT(b, round*BLOCK_SIZE + k, col);
      }
      __syncthreads();
   }
   ELEMENT(c, row, col) = sum;
}
```