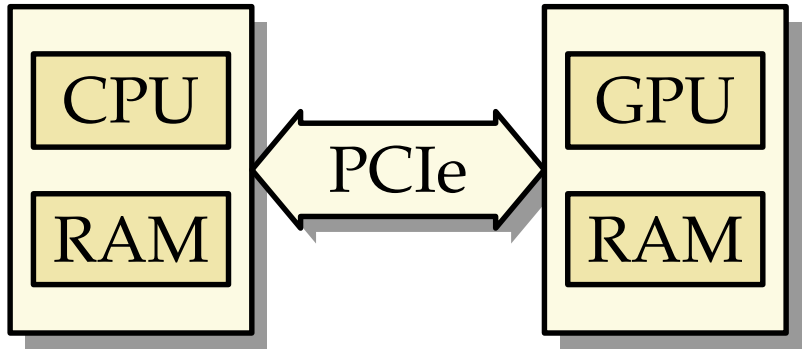
High Performance Computing I
WS 2017/2018

Andreas F. Borchert and Michael C. Lehn
Ulm University

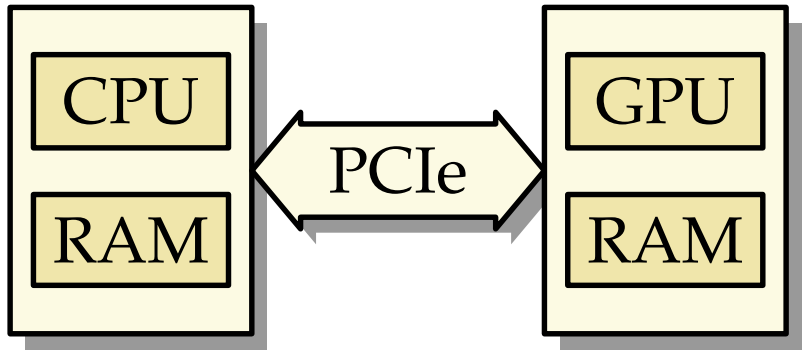
February 6, 2018

- Graphics accelerators were developed quite early to unburden the CPU.
- In March 2001, Nvidia introduced the GeForce 3 Series which supported programmable shading.
- Radeon R300 from ATI followed in August 2002 which extended the capabilities of the GeForce 3 series by adding mathematical functions and loops.
- In the following years GPUs developed step by step into the direction of so-called GPGPUs (*general purpose GPUs*).
- Multiple languages and interfaces were developed to support accelerating devices: OpenCL (Open Computing Language), DirectCompute (by Microsoft), CUDA (Compute Unified Device Architecture by Nvidia) and OpenACC (*open accelerators*, an open standard supported by Cray, CAPS, Nvidia, and PGI.



- Usually, GPUs are available as PCI Express cards.
- They operate separately from the CPU and its memory. Communications is done using the PCI Express bus.
- Most GPUs have very specialized processor architectures on base of SIMD (*single instruction multiple data*). One prominent exception are the Larrabee micro architecture and its successors by Intel which supports the x86 instruction set and which can no longer be considered as a GPU. Their only purpose is high performance computing.

Relation between CPU and GPU



- An application consists usually of two programs, one for the host (i.e. the CPU) and one for the accelerating device (i.e. the GPU).
- The program for the device is uploaded from the host to the device. Afterwards both communicate using the PCIe connection.
- The communication is done using a library which supports synchronous and asynchronous operations.

- High-level languages are supported for GPUs, typically variants that are specialized extensions of C and C++.
- The extensions are required to access all features of the GPU. On the other hand, support for C and C++ is not necessarily complete, in particular most libraries are not available on the GPU with the exception of some selected mathematical functions.
- The CUDA programming model allows program texts for the host and the device to be freely mixed, i.e. the same code can be compiled for both platforms. During compilation, the compiler separates the codes required for each platform and compiles to the corresponding architecture.
- In case of OpenCL programm texts for the host and the device have to be stored in different files.

CUDA is a package free of charge (but not open source) that is available by Nvidia for selected variants of Linux, MacOS, and Windows with following components:

- a device driver that supports the upload of GPU programs and the communication between host and device code,
- a language extension of C++ (CUDA C++) that allows to have a unified source for host and device,
- a compiler named *nvcc* that supports CUDA C++ (on base of C++11 in case of CUDA-8 and C++14 in case of CUDA-9), and
- a runtime library, and
- more libraries (including the support of BLAS and 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);
   // ...
}
```

- All CUDA applications can freely access the CUDA runtime library. Error codes of all CUDA runtime library invocations shall be checked.

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* fills a voluminous struct with the properties of one of the available GPUs.
- There exist manifold Nvidia graphic cards with different micro architectures and configurations.
- CUDA program sources have the suffix ".cu".

```
[ul_l_course01@vis02 properties]$ make
nvcc -o properties properties.cu
[ul_l_course01@vis02 properties]$ ./properties
one device present:
name: Quadro K6000
compute capability: 3.5
total global memory: 12799180800
total constant memory: 65536
total shared memory per block: 49152
registers per block: 65536
L2 Cache Size: 1572864
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: 15
kernel exec timeout enabled: yes
device overlap: yes
integrated: no
can map host memory: yes
unified addressing: yes
[ul_l_course01@vis02 properties]$
```

```
[ul_l_course01@vis02 properties]$ make
nvcc -o properties properties.cu
```

- We compile using the *nvcc* utility.
- The code which is required on the GPU is compiled for the corresponding architecture and stored as a byte-array in the host program, ready to be uploaded to the device. All other parts are forwarded to *gcc*.
- Options like "–gpu-architecture compute_30" allow to specify the architecture and enable the support of corresponding features.
- Options like "–code sm_30" ask to generate the code during compilation time, not later at runtime.

simple.cu

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

- Functions (or methods) that are to be run on the device but to be invoked from the host are called *kernel functions*. They are designated by using the CUDA keyword **__global__**.
- In this example, the function *add* computes $\vec{y} \leftarrow \vec{x} + \alpha\vec{y}$.
- All pointers refer to device memory.

simple.cu

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

- Kernel functions can be invoked anywhere in code that is run on the host.
- Between the name of the function and the parameter list a kernel configuration has be given that is enclosed in "<<<...>>>". This is strictly required and the parameters will be explained in the following.
- Elementary data types can be transmitted as usual (in this case N and the value 2.0). Pointers, however, must point to device memory. That means that the data of vectors and matrices has to be transfered first from host to device.

Copying between host and device

simple.cu

```
double a[N]; double b[N];
for (unsigned int i = 0; i < N; ++i) {
   a[i] = i; b[i] = i * i;
}

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

- The CUDA runtime library function *cudaMalloc* allows to allocate device memory.
- Fortunately, all elementary data types are represented in the same way on host and device memory. Hence, we can simply copy data from host to device without needing to convert anything.

Copying between host and device

simple.cu

```
double a[N]; double b[N];
for (unsigned int i = 0; i < N; ++i) {
   a[i] = i; b[i] = i * i;
}

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

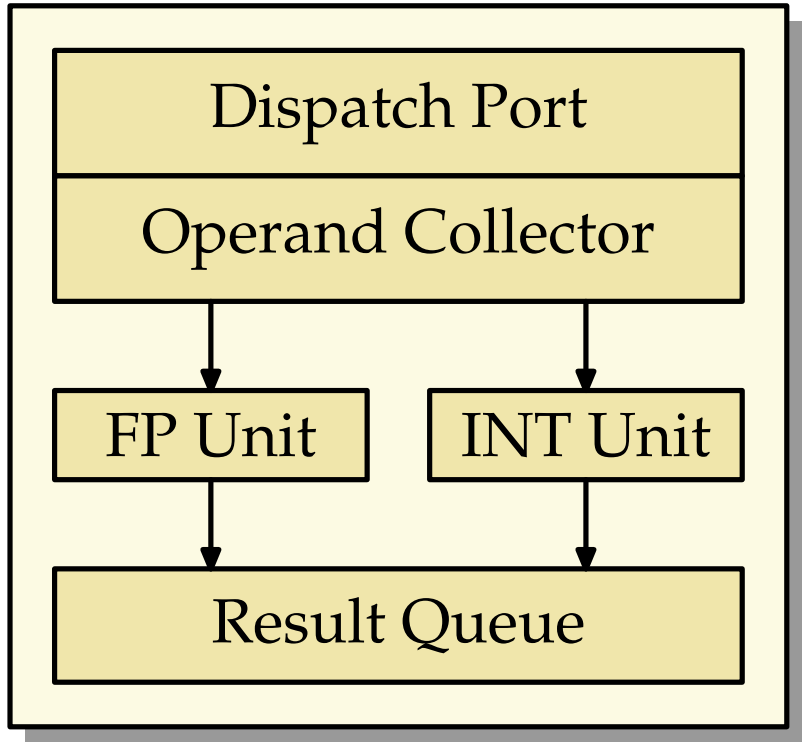
- *cudaMemcpy* supports copying in both directions. The parameters specify first the target address, then the source address, then the number of bytes, and finally the direction.
- In dependence of the direction, the addresses are interpreted to be addresses of host or device memory.

simple.cu

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

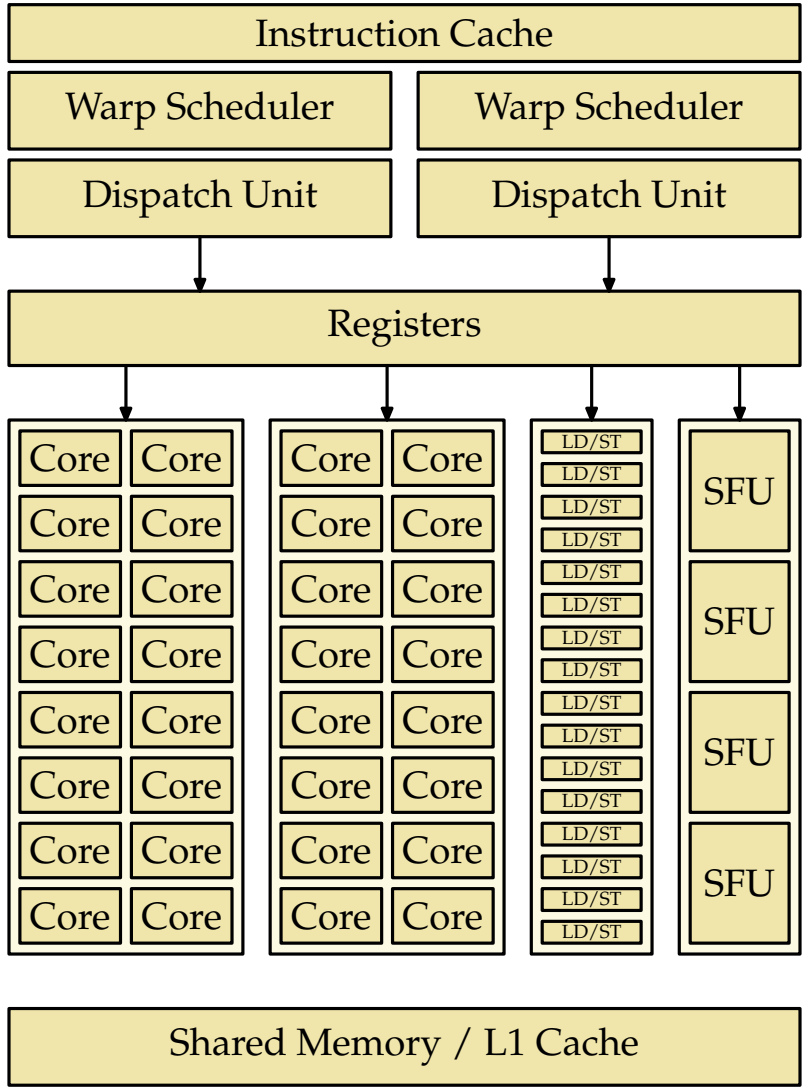
- We transfer the result from device to host after invoking the kernel function.
- Kernel functions run asynchronously, i.e. the host continues computing when the kernel function runs on the device. However, by default requests for the device (like copying or kernel function invocations) are serialized. Thereby, the call of *cudaMemcpy* that copies the result back from device to the host waits implicitly until the kernel function is finished.
- If the kernel function couldn't be started or failed for some reason, the associated error code is delivered with the next serialized CUDA operation.
- Finally, the allocated device memory ought to be released.

- The first example shows the typical order of execution of many applications:
  - Allocate device memory
  - Transfer input data from host to device
  - Invoke kernel function
  - Copy results from device to host
  - Release device memory
- We had no real parallelization in this example as the host waited for the device and the device employed one single core only.

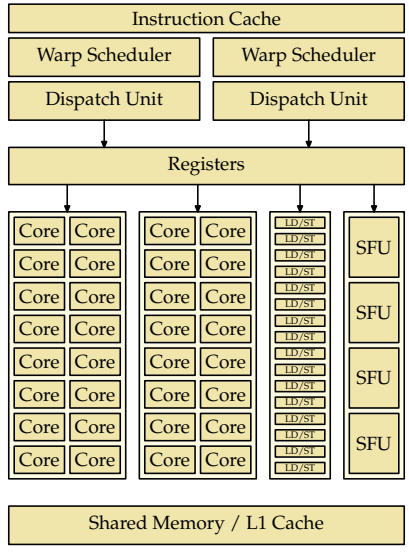


- The stripped-down cores of a GPU consist mainly of two units, one supporting arithmetic operations for floating point numbers, the other for integers.
- GPU cores don't do much beyond that. Instructions and data are delivered (*Dispatch Port*, *Operand Collector*), and the result is passed to the *Result Queue*.

Multiprocessors of a GPU

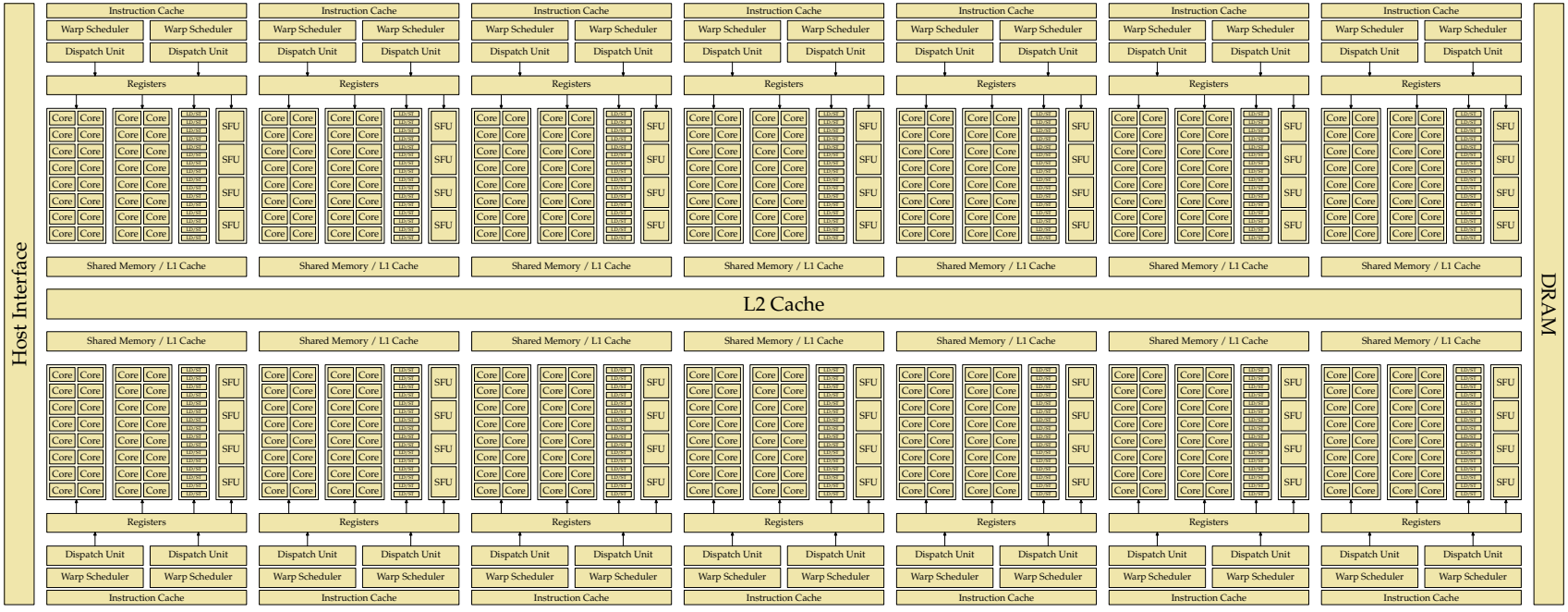


Multiprocessors of a GPU



- In dependence of the actual GPU configuration a fixed number of cores are packed along with additional units to a so-called multiprocessor.
- The multiprocessors provide additional units that support parallel loads and stores (*LD/ST*) and units for special mathematical operations like *sin* or *sqrt* (*SFU*).
- Some advanced GPU cards pack multiple multiprocessors to one cluster.

GPU with multiple multiprocessors



- Each GPU consists of multiple multiprocessors.
- In the design shown we have 14 multiprocessors with 448 cores in total.

- The individual cores do not operate independent from each other.
- Usually, 32 cores are unified to a so-called warp. (Some architectures support also so-called half warps with 16 cores.)
- All cores of a warp execute synchronously the same instruction at the same time but get different data (SIMD architecture: array processor).
- At the beginning of each instruction, the input data are fetched in parallel by *LD/ST* units and stored into registers which can be used by the cores.

- The instruction set of the Nvidia GPUs is proprietary and has not been published yet – just a summary is available.
- The utility *cuobjdump* allows to disassemble code.
- The instructions have a length of 32 or 64 bits, 64-bit instructions must be on 8-byte-boundaries.
- Arithmetic operations can have up to three source operands and a target where the result is to be stored. For example, the instruction *FMAD* computes $d = a * b + c$ in single precision.

Jumps

How are conditional jumps implemented? Remember, all cores of a warp executes always the same instruction. What is to be done if we hit an if statement where different cores would like to continue differently?

- We have two stacks:
- One stack provides masks consisting of 32 bit which tell which core of a warp has to execute the instruction. (Others will simply ignore it.)
- The other stack maintains target addresses (of conditional jumps).
- In case of conditional jumps, every core sets its bit in the mask to signal if the following instructions are to be executed or not. The joined mask is then pushed onto the corresponding stack.
- The target address of the conditional jump is pushed onto the second stack.
- When the target address is reached, the top-most elements of both stacks are removed.

- A block is an abstraction that joins multiple virtual warps. (At least one physical warp is required, we can have more virtual warps as long we have sufficient registers to store the current state of all virtual warps, allowing us to switch the physical warps between different virtual warps).
- A block is always connected with a multiprocessor. The *Warp Scheduler* of a warp decides which virtual warp is to be run at which time on which physical warp.
- All threads that belong to a block have access to shared memory.
- Threads within a block are free to synchronize using barriers and to communicate using the shared memory of the multiprocessor.
- Typically, one block supports up to 32 warps (or 1024 threads).

vector.cu

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

- The **for**-loop is gone. Instead each core executes just one addition.
- Using *threadIdx.x* we learn who we are, i.e. which thread within a block (beginning at 0).
- The kernel function will then be executed for each element of the vector. And this will be parallelized to the extent possible by one multiprocessor.

vector.cu

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

- The configuration of the kernel function has been adapted.
- $<<<1, N>>>$ configures one block with N threads.
- Hence, the individual threads operate with *threadIdx.x* values in the range from 0 to $N - 1$.
- N must not exceed the maximal number of threads per block.

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 is an acronym for *parallel thread execution* and stards for an assembly language of a virtual GPU processor. This is the target language of the *nvcc* for the device code.

- The PTX instruction set is public:
  `http://docs.nvidia.com/cuda/pdf/ptx_isa_6.1.pdf`
- The virtual machine of PTX has been developed to have an intermediate architecture-independent language which can be efficiently translated to a particular architecture.
- PTX can be generated using following command:
  `nvcc -o vector.ptx -ptx vector.cu`

Compilation process: Generation of CUBIN

vector.disas

```
/*0000*/         MOV R1, c[0x1][0x100];
/*0008*/         NOP;
/*0010*/         MOV32I R3, 0x8;
/*0018*/         S2R R0, SR_TID.X;
/*0020*/         IMAD.U32.U32 R8.CC, R0, R3, c[0x0][0x28];
/*0028*/         MOV R10, c[0x0][0x20];
/*0030*/         IMAD.U32.U32.HI.X R9, R0, R3, c[0x0][0x2c];
/*0038*/         MOV R11, c[0x0][0x24];
/*0040*/         IMAD.U32.U32 R2.CC, R0, R3, c[0x0][0x30];
/*0048*/         LD.E.64 R6, [R8];
/*0050*/         IMAD.U32.U32.HI.X R3, R0, R3, c[0x0][0x34];
/*0058*/         LD.E.64 R4, [R2];
/*0060*/         DFMA R4, R6, R10, R4;
/*0068*/         ST.E.64 [R2], R4;
/*0070*/         EXIT ;
```

- *ptxas* translates PTX for a particular architecture into machine code (CUBIN):
  `ptxas -output-file vector.cubin -gpu-name sm_30 vector.ptx`

- Nvidia provides *cuobjdump* which allows to disassemble generated CUBIN code:
  `cuobjdump -dump-sass vector.cubin >vector.disas`
- The instruction summary of the Kepler architecture is to be considered for the GPU on our JUSTUS node.

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;
}
```

- We need to spread our task across multiple blocks if the number of elements exceeds the maximal number of threads per block.

bigvector.cu

```
   /* execute kernel function on GPU */
   unsigned int max_threads = max_threads_per_block();
   if (N <= max_threads) {
      add<<<1, N>>>(2.0, cuda_a, cuda_b);
   } else {
      add<<<(N+max_threads-1)/max_threads,
         max_threads>>>(2.0, cuda_a, cuda_b);
   }
```

bigvector.cu

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

- As *threadIdx.x* just identifies the thread within a block, it no longer works as a global index within the range of 0 to $N-1$.
- *blockIdx.x* gives the index of the current block and *blockDim.x* the number of threads per block.
- Hence, we will have idling threads if *max_threads* is not a divisor of N. An **if**-statement is necessary to avoid out-of-bound accesses of the arrays.

In the most general form, a kernel configuration accepts four parameters where the last two are optional:
$<<<$ *Dg*, *Db*, *Ns*, *S* $>>>$

- *Dg* specifies the dimensions of the grid (in one, two, or three dimensions).
- *Db* specifies the dimensions of a block (in one, two, or three dimensions).
- *Ns* specifies the size of the shared memory per block (by default 0 which causes the compiler to allocate shared memory automatically).
- *S* allows to associate the kernel with a stream (by default none).
- Both, *Dg* and *Db* are of type *dim3* whose constructor accepts up to three whole numbers.
- The limits of the current device are to be taken into consideration. If the parameters exceed the limits, the kernel function will not start.

The kernel and device functions can access the following variables that allow to identify the own thread and to retrieve the dimensions of the blocks and the grid:

threadIdx.x	x index within the own block
threadIdx.y	y index within the own block
threadIdx.z	z index within the own block
blockDim.x	first dimension of the block
blockDim.y	second dimension of the block
blockDim.z	third dimension of the block
blockIdx.x	x index of the own block within the grid
blockIdx.y	y index of the own block within the grid
blockIdx.z	z index of the own block within the grid
gridDim.x	first dimension of the grid
gridDim.y	second dimension of the grid
gridDim.z	third dimension of the grid

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 int N = blockDim.x * gridDim.x;
   const unsigned int 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));
}
```

- The kernel function applies Simpson's rule for one subinterval. Both, the blocks and the grid have one dimension only. Hence, $blockDim.x * gridDim.x$ gives the total number of subintervals.
- Functions that can be invoked on the device (but not as kernel function) can be designated using the keyword **__device__**.

Simpson's rule on the GPU

simpson.cu

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

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

- Aggregation has to be done manually – there are no aggregation operators.

Block-wise aggregation

simpson2.cu

```
#define THREADS_PER_BLOCK 256 /* 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 int N = blockDim.x * gridDim.x;
   const unsigned int 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 */
   unsigned int me = threadIdx.x;
   __shared__ double sums_per_block[THREADS_PER_BLOCK];
   sums_per_block[me] = sum;

   // ...
}
```

- Synchronization and shared memory are supported within a block.
- This opens the option of a block-wise aggregation.

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;
```

- Variables designated with **__shared__** are shared among all threads of a block.
- Access of shared memory is much faster than access of global memory.
- Usually the capacity is very limited. The GPU on our JUSTUS node provides 48 KiB per block.

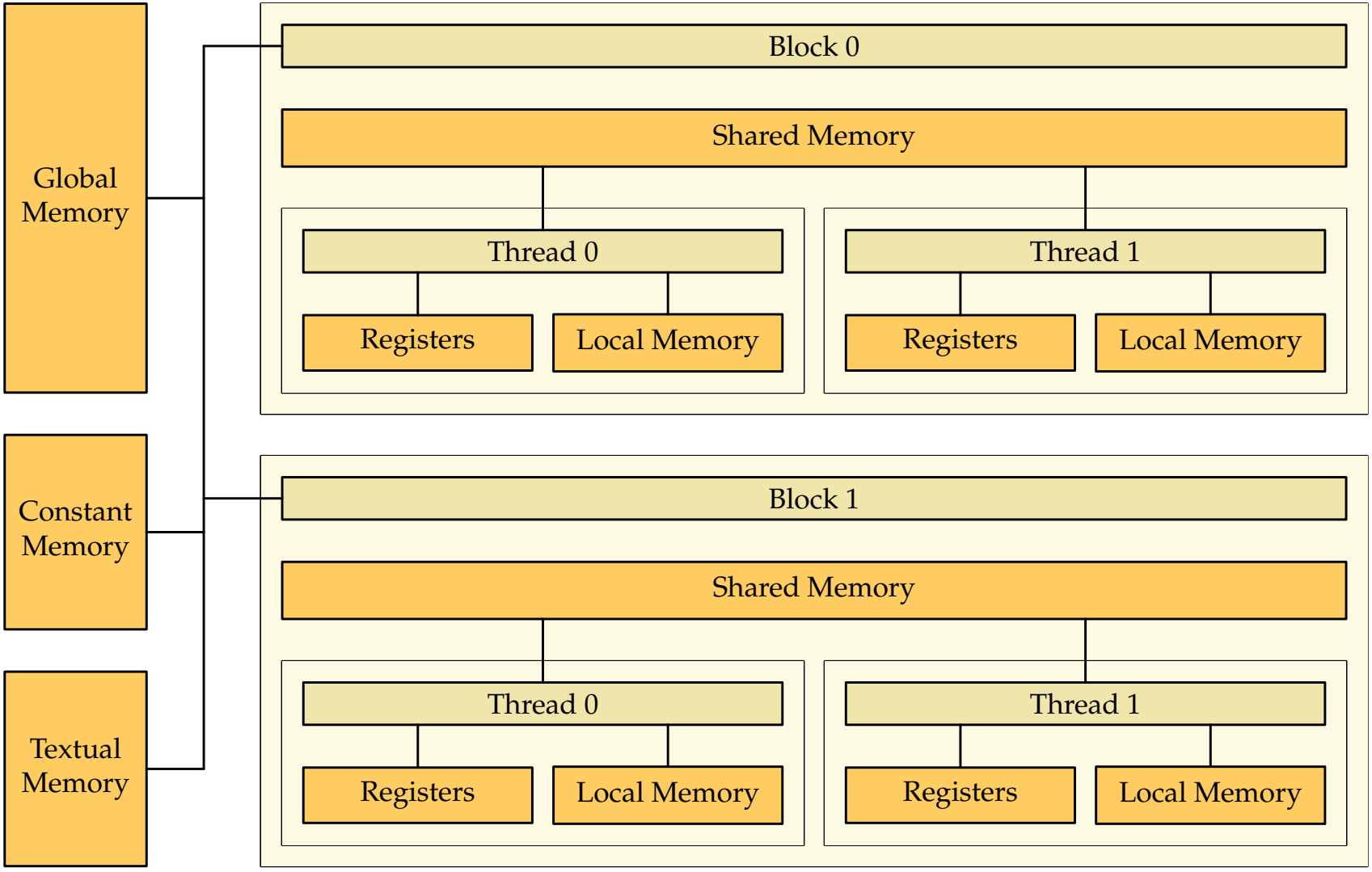
Block-wise aggregation

simpson2.cu

```
/* aggregate sums within a block */
int 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];
}
```

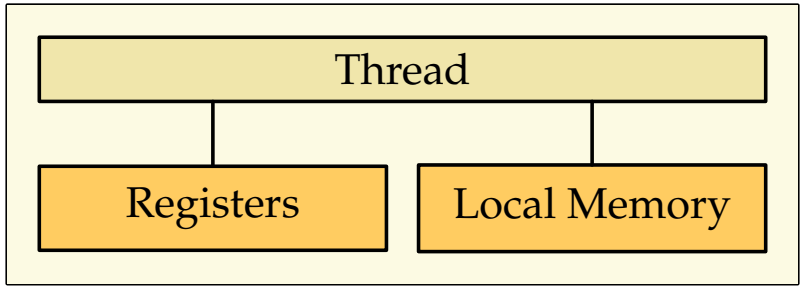
- Individual threads of a block are grouped to warps that are run synchronously (SIMD architecture). However, threads of the same block that belong to different warps are not necessarily executed synchronously.
- *__syncthreads* is a barrier function that suspends the thread (along with its warp) until all threads of the same block reached the barrier by invoking this function.

Memory areas of a GPU device



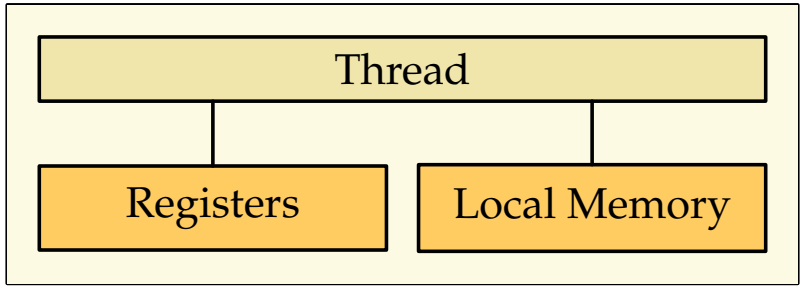
```
hochwanner$ make
nvcc -o simpson --gpu-architecture compute_20 -code sm_20 --ptxas-options=-v s
ptxas info    : 304 bytes gmem, 40 bytes cmem[14]
ptxas info    : Compiling entry function '_Z7simpsonddPd' for 'sm_20'
ptxas info    : Function properties for _Z7simpsonddPd
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 25 registers, 56 bytes cmem[0], 12 bytes cmem[16]
hochwanner$
```

- *ptxas* provides a summary of the memory usage of a kernel function if the verbose flag was switched on.
- *gmem* refers to global memory, *cmem* to constant memory.
- Local memory is necessary for stack frames and for saving away registers (*spill stores*). Both are counted statically.

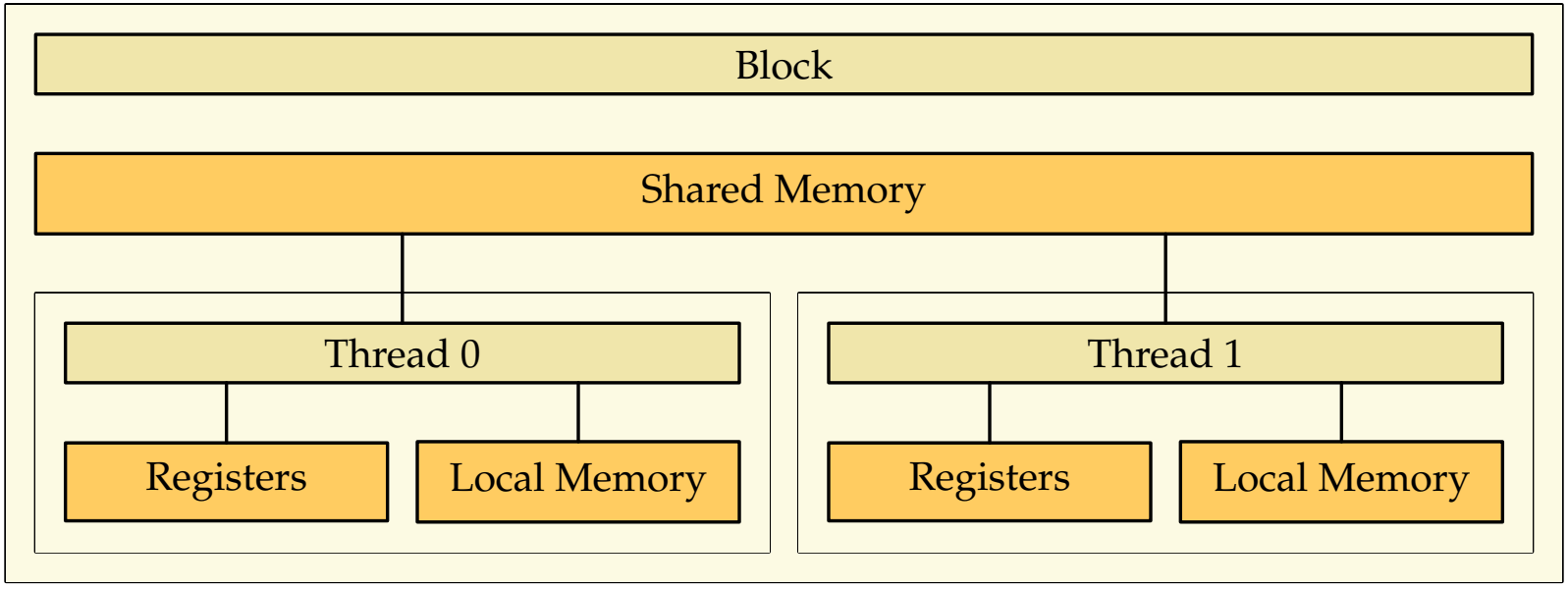


- Registers are available for whole integers and floating point.
- Local variables in a thread are stored into registers to the extent possible.
- In case of large number of needed registers per thread the actual limit of permitted threads per block is possibly decreased.
- The GPU on our JUSTUS node provides 65536 registers per block. If we configure 1024 threads per block, we will have up to 64 registers for each thread.

Local memory



- Local memory is allocated in the global memory area.
- However, the architecture supports cache-optimized instructions for loading and storing from and to local memory. Optimization is possible as cache coherence is not required.
- Nonetheless, local memory is much slower than shared memory which is served from the L1 cache.
- Local memory is used for the stack, when we are running out of registers, and for local arrays where indices are computed at runtime.



- Next to the registers, shared memory provides the fastest access as it resides in the L1 cache.
- The capacity is very limited. The GPU on our JUSTUS node provides 48 KiB per block.

- Shared memory is cyclically divided into banks. The first word of shared memory (32 or 64 bits in dependence of the configuration, see cudaDeviceSetSharedMemConfig) belongs to the first bank, the second word to the second bank etc. The GPU on our JUSTUS node has 32 banks. Hence the 33rd word refers to the first bank.
- Shared memory accesses of the threads of a warp are fully parallelized, provided each thread of a warp accesses a different bank. However, there is special support for broadcasts and multicasts, i.e. one word can be delivered to multiple threads at the same time.

- Global memory is accessible to all threads and, using *cudaMemcpy*, also by the host.
- There is no paging, i.e. we have no more virtual memory than physically available.
- The GPU on our JUSTUS node provides 11.92 GiB global memory.
- Global memory on the Kepler architecture is accessed through L2 which is also responsible for cache coherence.
- Global variables can be declared using the keyword **__global__**.

Access of global memory

If following conditions are met, access of global memory is fast independently of the architectural variant:

- 32-bit words or larger entities are accessed.
- The access addresses are consecutively increasing for the individual threads of a warp.
- The first word shall be aligned:

Word size	Alignment
32 bits	64 bytes
64 bits	128 bytes
128 bits	256 bytes

Access of global memory

In case of the Kepler architecture (on our JUSTUS node) global memory is by default accessed through L2.

- The cache lines for L1 and L2 have a size of 128 bytes.
- One cache line can be transfered within one transaction from L2 into registers. There is no requirement that the loads are done consecutively by the threads of a warp.
- Loads of global memory into L2 are split into individual memory requests where one memory request can load one cache line.
- Example: If each threads accesses a **double** variable out of a properly aligned array we need at least two memory requests.

- Constant memory is used for the parameters of a kernel function. Variables can be declared to reside in constant memory by using the **__constant__** specifier.
- The GPU on our JUSTUS node provides 64 KiB constant memory.
- Accesses of global memory are optimized as they do not require cache coherence.
- Write accesses are not permitted by the device. They must be set by the host before the kernel function is invoked.

tracer.cu

```
__constant__ char sphere_storage[sizeof(Sphere)*SPHERES];
```

- Variables in constant memory are declared using the **__constant__** specifier.
- Data types with non-default constructors or destructors are not supported as these declarations just allocate memory.
- Using *cudaMemcpyToSymbol* it is possible to copy data by the host from host memory to a variable in constant memory.

tracer.cu

```
  Sphere host_spheres[SPHERES];
  // fill host_spheres...
  // copy spheres to constant memory
  CHECK_CUDA(cudaMemcpyToSymbol, sphere_storage,
     host_spheres, sizeof(host_spheres));
```