Chapter 7
General Purpose GPU Programming

Manycore GPUs (Graphics Processing Units) are available in almost all current hardware platforms, from standard desktops to computer clusters and thus, provide easily accessible and low cost parallel hardware to a broad community. Originally, these processors have been designed for graphics applications, however, today there is an increasing importance for applications from scientific computing and scientific simulations. Especially for data parallel programs there can be a considerable increase of efficiency. This efficiency improvement is mainly caused by the specific hardware design of GPUs, which has been optimized for large data of graphics applications and high throughput of floating-point operations to be executed by a large number of threads. Nowadays, GPUs may comprise hundreds of cores executing these threads. A brief overview of the current architecture design of GPUs is provided in the first Sect. 7.1.

At the beginning, the amount of effort required to use GPUs for general non-graphics applications and simulations was extremely high due to programming environments, such as DirectX or OpenGL, designed for graphics applications. More recent programming environments for GPUs are CUDA and OpenCL. CUDA (Compute Unified Device Architecture) is a more generic parallel programming environment supported by NVIDIA GPUs for new generations since 2007 [34] and can also be emulated on CPUs. OpenCL (Open Computing Language) has been jointly developed by industrial partners, including Apple, Intel, AMD/ATI, and NVIDIA, as a standardized programming model for GPUs. This chapter provides an introduction to the parallel programming with CUDA according to [35, 118, 174] in Sects. 7.2–7.5 followed by a brief introduction to OpenCL written from a CUDA perspective according to [82, 142] in Sect. 7.6.

7.1 The Architecture of GPUs

GPUs have been evolved from graphics accelerator with the main purpose to do graphics applications well. This explains why the architecture of manycore GPUs has been developed quite independently from the architecture of general CPUs.

T. Rauber and G. Rünger, *Parallel Programming*,
DOI: 10.1007/978-3-642-37801-0_7, © Springer-Verlag Berlin Heidelberg 2013

Because of the high potential of data parallelism in graphics applications, the design of GPU architectures has used several specialized processor cores much earlier than this has been done for the architecture of CPUs. Today, a single GPU can comprise several hundreds of compute cores, which is a much higher number of cores than used in current CPU technology.

In addition to graphics processing, GPUs can also be employed for general non-graphics applications. This is useful if the potential of data parallelism of the non-graphics application program is large enough to fully utilize the high number of compute cores in a GPU. Scientific simulations internally processing numerical calculations, such as those given in Chap. 8, usually have this property. Numerous example implementations have shown that those applications can benefit from the many-core CPU architecture resulting in a much better compute performance than on a CPU. The trend to use GPUs for general numerical applications has inspired GPU manufacturers, such as NVIDIA, to develop the programming environment CUDA and OpenCL. These parallel programming environments combine the effort to support non-graphics applications on GPUs on the one hand and to provide a specific programming model which is adequate for the architecture of GPUs.

Both CUDA and OpenCL separate a program into a CPU program (the host program), which includes all I/O operations or operation for user interaction, and a GPU program (the device program), which contains all computations to be executed on the GPU. The simplest case of interaction between host program and device program is implemented by a host program that first copies data into the global memory of the GPU and then calls device functions to initiate the processing on the GPU. These device functions take the parallel architecture of the GPU into account to appropriately map the data parallel computations to compute cores. Important aspects are also the memory organization of the GPU and the specific data transfer between CPU and GPU. These aspects will be discussed later in this chapter. This section is devoted to the basic concepts of the architecture of GPUs and concentrates specifically on NVIDIA-GPUs. More detailed architecture descriptions are given in [36, 37, 94].

A GPU comprises several multi-threaded SIMD processors which are independent MIMD processor cores processing independent sequences of compute instructions each. The actual number of SIMD processors incorporated in a GPU depends on the specific GPU model. For example, the NVIDIA GTX480 GPU belonging to the family of the Fermi-architecture has up to 15 independent SIMD processors. Each of the SIMD processors has several SIMD function units, which can execute the same SIMD instruction on different data. Each SIMD function unit has a separate set of registers, and the data for the SIMD instructions have to be available in these local registers. Specific transfer operations are provided to initiate the data transfer from the memory into the registers. The actual transfer may require several machine cycles because the data may reside in the global memory of the GPU (off-chip) from which they have to be fetched. More recent GPU architectures contain a cache memory hierarchy, so that some data transfer operations can use cached data and are faster than data accesses to non-cached data.

At first glance, it seems to be sufficient to provide one thread of control (SIMD thread) to one SIMD processor, since parallelism results because each SIMD function unit executes the same instruction on different data. However, the data transfer operations will cause waiting times of uncertain length. To hide these waiting times, SIMD processors are able to execute several independent SIMD threads. A SIMD thread scheduler picks a SIMD thread ready for execution and initiates the execution of the next SIMD instruction of this thread. Each of the SIMD threads uses an independent set of registers. This method is a special form of multithreading, see Sect. 2.3.3.

In each execution step, the SIMD thread scheduler can select a different SIMD thread, since the SIMD threads are independent of each other. To support the selection, the SIMD thread scheduler uses a scoreboard which contains for each SIMD thread the current instruction to be executed and the information whether the operands of this instruction reside in registers or not. The maximum number of SIMD threads to be supported depends on the size of the scoreboard. The scoreboard size is an architectural feature and a typical size for the Fermi-architecture from NVIDIA is 32 [94]. The actual number of independent SIMD threads is determined by the application program as described later in this chapter.

The number of function units of a SIMD processor also depends on the specific GPU model. Each of the function units contains an integer unit and a floating-point unit. Figure 7.1 illustrates the internal organization of a single SIMD processor with 16 physical function units (FU). In total, 32768 32-bit registers are provided so that each of the 16 function units owns 2048 physical registers. Two neighboring physical

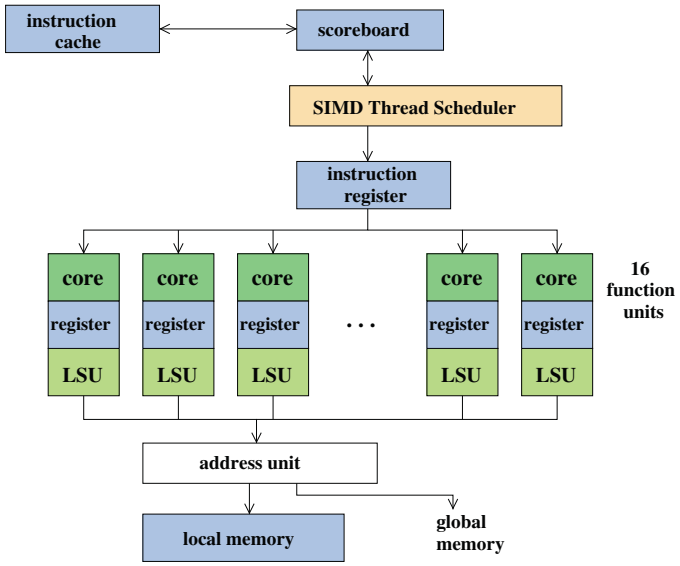

Fig. 7.1 Block diagram of a SIMD processor according to [94]. The SIMD processor has 16 function units (FU), each of which has a separate set of registers and a separate Load-Store-Unit (LSU).

registers can be used as one 64-bit register. The 2048 registers of one function unit are distributed among the SIMD threads available in the current program execution. In case of 32 SIMD threads, each of these threads can use 64 separate registers of each function unit for the storage of its data. The registers are dynamically assigned to the SIMD threads when they are created. A local memory (on-chip) is available for each SIMD processor, providing fast access for the SIMD threads running on this SIMD processor. The global memory (off-chip) is shared by all SIMD processors and can be accessed also by the CPU. Access to the global memory is much slower than access to the local memories of the SIMD processors.

The real design of a GPU of the Fermi-architecture is more complex than illustrated in Fig. 7.1 and is as follows: Each SIMD processor employs two SIMD thread schedulers (and not only one) with corresponding dispatch units for launching the SIMD instructions. Correspondingly, there are two sets of function units with 16 function units each. In addition, there are 16 transfer units for the transfer of data between registers and the memory system. Four special function units (SFU) are dedicated to the execution of special functions, such as sinus, cosinus, square root, or reciprocal value. Figure 7.2 shows the block diagram of a SIMD processor of the Fermi-architecture according to [36, 94].

Supplementary to the technique to hide the latency of memory accesses by switching between different SIMD threads, the Fermi-architecture has a cache memory hierarchy. Each SIMD processor owns an L1 instruction cache and an L1 data cache. Together, the L1 data cache and the local memory have a total size of 64 KB, which can be divided into a 16 KB L1 cache and a 48 KB local memory or a 48 KB L1 cache and a 16 KB local memory. A 768 KB L2 cache is shared by all SIMD processors of the GPU.

Depending on the specific version, a GPU chip of the Fermi-architecture contains 7, 11, 14, or 15 SIMD processors. The threads are scheduled in a two-layered way: (1) The thread block scheduler assigns blocks of threads to the SIMD processors, according to the data layout. (2) The SIMD thread scheduler of a single SIMD processor selects a SIMD thread for execution.

An NVIDIA Fermi GTX480 with 15 SIMD processors has more than 3 billion transistors and a power consumption of 150 Watt. In contrast to earlier generations of GPUs, the GPUs of the Fermi architecture are able to execute floating-point operations with double precision quite fast, requiring only about twice the execution time of a floating-point operation with single precision. In contrast, for earlier GPUs there was a time difference of about a factor of 10 between the execution time of floating-point operations of single precision and double precision. The peak performance of the NVIDIA Fermi GTX480 GPU is 515 GFLOPS (Giga Floating Point Operations per Second) for operations with double precision, which is ten times faster than standard desktop processors. A detailed performance comparison between CPUs and GPUs is given in [94].

The successor GPU architecture of the Fermi-architecture is the NVIDIA Kepler GPU architecture, see [37]. An essential design goal of the Kepler architecture was to improve the energy balance, i.e., the performance per Watt. The first GPU of this architecture is the NVIDIA GeForce GTX680 comprising only eight SIMD

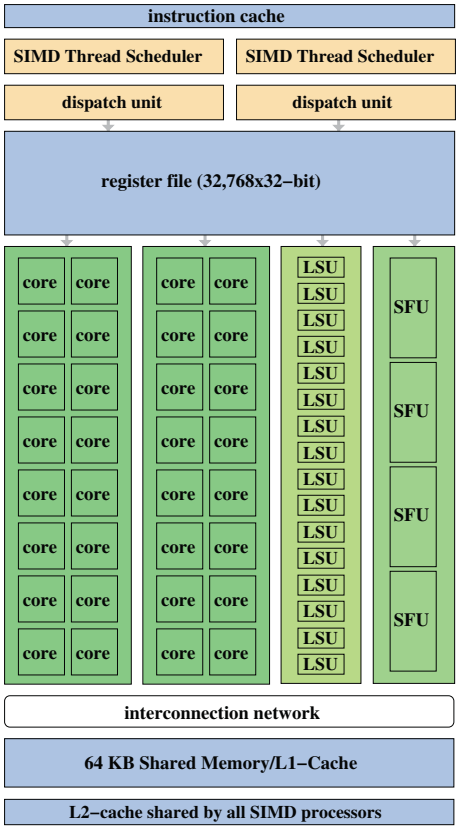

Fig. 7.2 Block diagram of an SIMD processor of the NVIDIA Fermi architecture according to [94].

processors, also called Streaming Multiprocessors, abbreviated as SMX. Two of these SMX are combined to a cluster, the Graphics Processing Cluster (GPC), so that a GeForce GTX680 consists of four GPCs, see Fig. 7.4. Additionally, each SIMD processor has a so-called Raster Engine which consists of several layers and is responsible for a fast processing of graphics data. The actual computations are executed on the SMX.

Each of the SIMD processors of the GTX680 GPU consists of 192 SIMD cores (function units) resulting in 1536 SIMD cores, 32 transfer units (load/store units, LSU), and 32 special function units (SFU). The number of 32-bit registers for all SIMD cores is 65536. For each SIMD processor, the assignment of SIMD threads to function units is done by four SIMD thread schedulers. Eight dispatch units are available to initiate the execution of SIMD instructions, see Fig. 7.3. The design of the L1 data cache is the same as for the Fermi-architecture: For each SIMD processor, 64 KB memory is available, which can be used as L1 cache and local memory (16 or 48 KB). Additionally, a 512 KB L2-Cache is shared by all SIMD processors of the GPU. Table 7.1 summarizes important characteristics of several generations of NVIDIA GPU architectures. From this table, an evolution toward a higher number

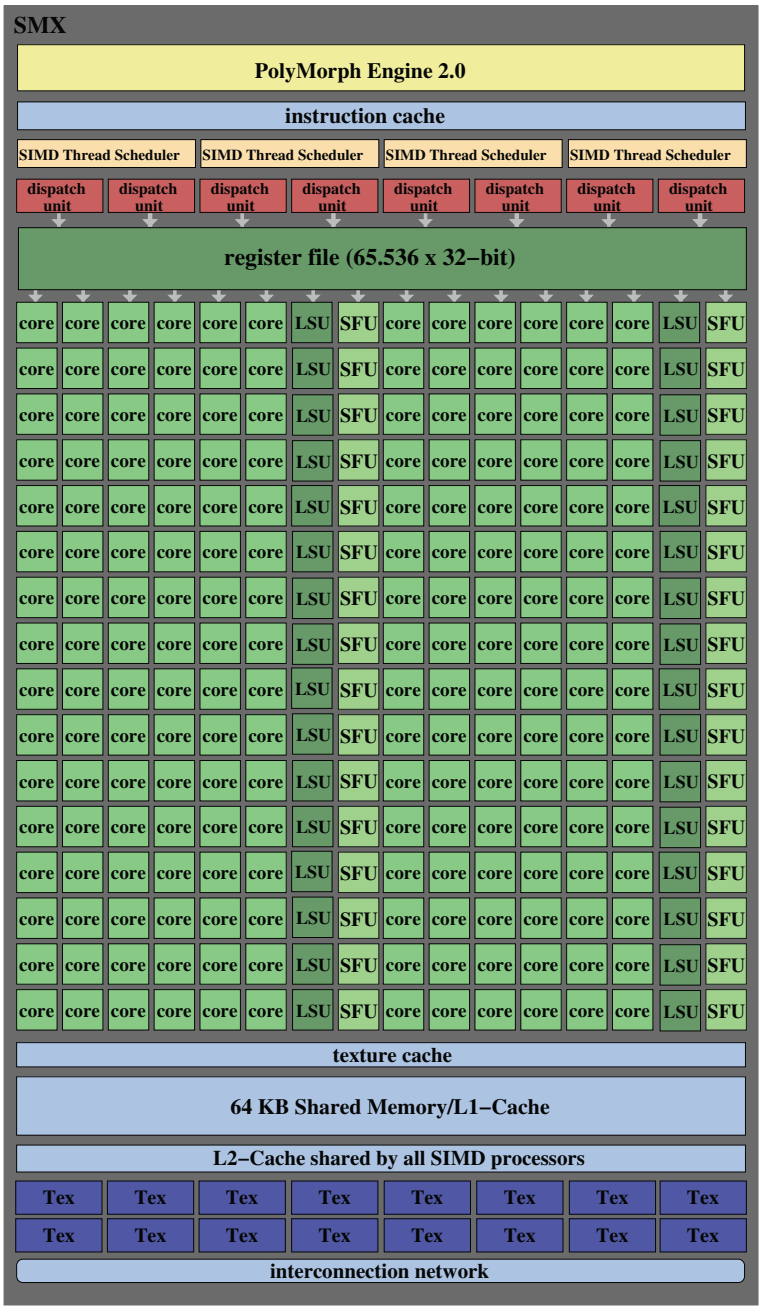

Fig. 7.3 Block diagram of an SIMD processors of the Kepler architecture.

Table 7.1 Summary of important characteristics of several NVIDIA GPUs of different architectures: GTX280 (Tesla-architecture), GTX480 (Fermi-architecture), and GTX680 (Kepler architecture), see also [37] and [94]. The GFLOPS values, denoted as GF in the table, are given for floating point operations of single precision and are the maximum reachable values. The consumption in Watt is the thermal design power (TDP) of the manufacturers, which is actually the value for the cooling system, but can also serve as indication for the maximum power consumption.

GPU	GTX 285 (Tesla)	GTX 480 (Fermi)	GTX 580 (Fermi)	GTX 680 (Kepler)
Transistors	$1,4 \cdot 10^9$	$3,2 \cdot 10^9$	$3,0 \cdot 10^9$	$3,54 \cdot 10^9$
SIMD processors	30	15	16	8
SIMD cores per SIMD processor	8	32	32	192
total number of SIMD cores	240	480	512	1536
L2-Cache	/	768 KB	768 KB	512 KB
Performance	1063 GF	1344 GF	1581 GF	3090 GF
Bandwidth memory	159 GB/sec	177 GB/sec	192 GB/sec	192 GB/sec
clock rate memory	2484 MHz	3696 MHz	4008 MHz	6008 MHz
power consumption	204 W	250 W	244 W	195 W

of SIMD cores and an increase of the computational power, given in GFLOPS, can be observed.

The use of the parallel units of GPU architectures can result in an efficient execution if the program is coded so that the SIMD processors can be fully employed. Especially, the application program has to provide an appropriate number of SIMD threads so that the SIMD thread scheduler of each SIMD processor has enough threads available for switching between them to hide the latency of memory accesses.

For efficiency purposes, a good organization of the memory layout of the application data is also important. This can be explicitly planned by the application programmer using appropriate commands in CUDA or OpenCL and can be supported by specific parallel programming techniques, as described in the following sections of this chapter.

7.2 Introduction to CUDA Programming

The coarse structure of a CUDA program is built up of phases which are either executed on a host or on one of the devices. The host is a traditional central processing unit (CPU) and the program phases for the host are C programs which can be compiled by a standard C compiler. The devices are massively parallel processors with a large number of execution units, such as GPUs, for processing massively data parallel program phases. The device code is written in C and CUDA-specific extensions for specifying data parallel executions. A data parallel function to be executed on the device is called a kernel function or simply kernel. The parallelism of a CUDA program resides in the kernel functions, which typically generate a large number

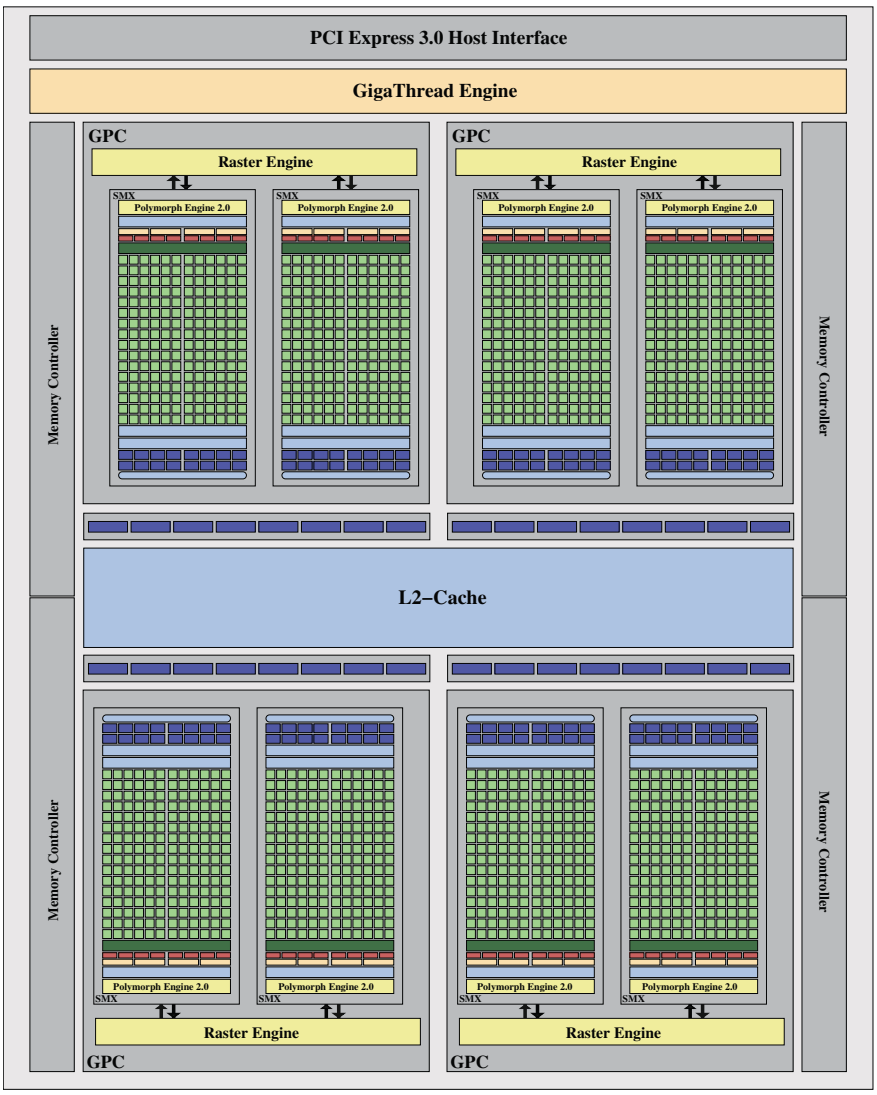

Fig. 7.4 Block diagram of a GeForce GTX 680 with Kepler architecture.

of CUDA threads. CUDA threads are light weight threads and require only a few cycles for generating and scheduling threads, in contrast to CPU threads, which need thousands of cycles for those tasks.

A CUDA program given in a file `*.cu` consisting of a host program and kernel functions is compiled by the NVIDIA-C-compiler (nvcc), which separates both program parts. The kernel functions are translated into PTX (Parallel Thread Execution) assembler code. PTX is the NVIDIA assembler language which, similar to

the X86 assembler language, provides a set of machine commands that ensure the compatibility of different generations of NVIDIA GPUs. A detailed description of PTX is given in [94]. Calls of kernel functions in the host program are translated into CUDA runtime system calls for starting the corresponding function on the GPU. The host program is then translated by a standard C compiler.

The execution of a CUDA program starts by executing the host program which calls kernel functions to be processed in parallel on the GPU. The call of a kernel function initiates the generation of a set of CUDA threads, called a grid of threads. A grid is terminated as soon as all threads of the grid have finished their part of the kernel function. After invoking a kernel function, the CPU continues to process the host program which might lead to the call of other kernel functions. CUDA extends the C function declaration syntax to distinguish host and kernel functions. The CUDA-specific keyword `__global_` indicates that the function is a kernel that can be called from a host function to be executed on a GPU. The keyword `__device_` indicates that the function is a kernel to be called from another kernel or device function. Recursive function calls or indirect function calls through pointers are not allowed in these device functions. A host function is declared using the keyword `__host_` and is simply a traditional C function that is executed on the host and can only be called by another host function. All functions without any keyword are host functions by default.

In order to execute a kernel function on a device, the data have to reside in the device memory. Thus, a CUDA program typically contains data transfer operations from the host to the GPU memory and also from the GPU to the CPU memory to transfer data results back to the host. These data transfer operations are explicitly coded in CUDA programs using CUDA-specific functions.

Before performing the data transfer from the host to the GPU memory, an appropriate amount of memory has to be allocated. This is done by the function

```
cudaMalloc(void**, size_t)
```

which is called by the host program to allocate memory in the global memory of the GPU. The function `cudaMalloc()` has two parameters: The first parameter is a pointer to the memory to be allocated and the second parameter specifies the size in Bytes of the memory allocated. The function

```
cudaFree(void*)
```

is called to free the storage space of the data objects given in the parameter after the computation is done. A data transfer from host to GPU is requested by calling the copy function

```
cudaMemcpy(void*, const void*, size_t, enum cudaMemcpyKind)
```

with four parameters:

- a pointer to the destination of the transfer operation,
- a pointer to the source of the transfer operation,
- the number of bytes to be copied, and
- a predefined symbolic constant specifying the type of the memory operation to be used in the transfer operations, e.g., from host to device.

Types provided for the data transfer are `cudaMemcpyHostToDevice` for the transfer from host to device and `cudaMemcpyDeviceToHost` for the transfer from device to host. Copy operations from host to host or from device to device are also possible. However, a copy operation between two different GPUs is not possible.

After having initiated the data transfer into the global memory of the GPU, the host program can invoke a kernel function on the GPU working on those data. The invocation of a kernel function contains an execution configuration which specifies the grid organization of the threads to be generated for the execution of that kernel function. Threads in a grid are organized as a two-level hierarchy. At the first level, each grid consists of several thread blocks all of which have the same number of threads. The second level is the organization of the threads within each thread block, which is identical for all thread blocks of the same grid.

Blocks of a grid have a two-dimensional structure, in which each thread block has a unique two-dimensional coordinate given in the CUDA-specific keywords `blockIdx.x` and `blockIdx.y`. This holds for CUDA Versions 2 and older; since CUDA Version 3, a three-dimensional block structure is supported. The threads within a thread block are, in turn, organized in a three-dimensional structure and the unique three-dimensional coordinate of each thread is given in the CUDA-specific variables `threadIdx.x, threadIdx.y`, and `threadIdx.z`. The maximum number of threads in a thread block is 512 threads (up to Version 2) and 1024 threads (since Version 3). According to this two-level hierarchy, each thread of a grid is uniquely identified by the block coordinates of the thread block it belongs to and its thread coordinates within this block. These coordinate values can be used in kernel functions to distinguish the parallel threads resulting in a data parallel execution.

The size of the thread grid and the thread blocks generated for the execution of a specific kernel invocation are defined in the execution configuration mentioned before. For the specification of an execution configuration, two `struct` variables of type `dim3` are declared; `dim3` is an integer vector-type built up from the vector-type `unit3` and initiated with the value 1, if not specified otherwise. These `struct` parameters describe the two- or three-dimensional organization of the thread grid and the thread blocks and are included in the kernel call syntax surrounded by `<<<` and `>>>` as illustrated in the following example:

```
// execution configuration
dim3 gsize(gx, gy);
dim3 bsize(bx, by, bz);
// call to a kernel function
KernelFct <<< gsize, bsize>>>(...);
```

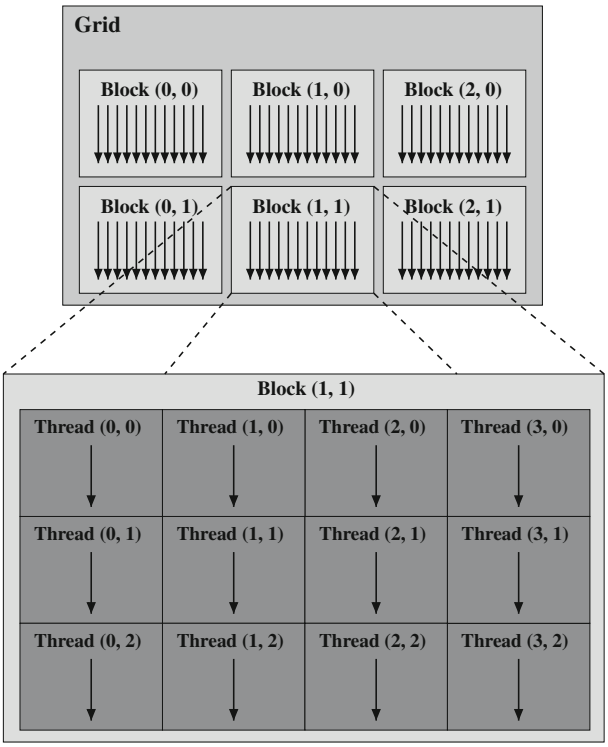

Fig. 7.5 Illustration of a CUDA execution configuration for grid dimension `dim3 gsize(3, 2)` and block dimension `dim3 bsize(4, 3)`.

The two-dimensional grid structure for the execution of kernel `KernelFct` is given in `gsize` and has size `gx` × `gy`. The three-dimensional structure of the block is given in `bsize` and has size `bx` × `by` × `bz`. The sizes of the current grid and its blocks are stored in the CUDA-specific variables `gridDim` and `blockDim`. More precisely, the sizes of the grid are given by `gridDim.x` and `gridDim.y` and would have the values `gx` and `gy`, respectively, in the example given above. Analogously, the size of the thread blocks are stored in `blockDim.x`, `blockDim.y`, and `blockDim.z` and would contain the values `bx`, `by`, and `bz`, respectively, in the example.

Figure 7.5 illustrates a specific execution configuration for a grid of size `dim3 gsize(3,2)` and thread blocks of size `dim3 bsize(4,3)`. The third dimension is not given any size and, thus, the size of this dimension in 1. The invocation of a kernel function `KernelFct` with this execution configuration results in the assignment `gridDim.x=3`, `gridDim.y=2`, `blockDim.x=4`, `blockDim.y=3`, and `blockDim.z=1`. Typical grids contain many more blocks and the values for `gridDim.x` and `gridDim.y` can be between 0 and 65535. Correspondingly, also the possible value for the block identifier `blockIdx.x` and `blockIDx.y` are

limited and can be between 0 and `gridDim.x−1` and 0 and `gridDim.y−1`, respectively. It has to be noticed that the pair of identifiers `(blockIdx.x, blockIdx.y)` denotes the column in the first component and the row in the second, which is the reverse order than in the specification of field elements.

The example execution configuration in Fig. 7.5 generates 3*2 = 6 thread blocks with 4*3 = 12 threads each, resulting in a total of 72 threads. If only a one-dimensional grid with one-dimensional thread blocks is needed for a kernel function call, the size of the grid and the blocks can be used directly in the execution configuration. For example, the call

```
KernelFct <<<8, 16>>>(...);
```

is identical to the call

```
dim3 gsize(8, 1);
dim3 bsize(16, 1);
KernelFct <<<gsize, bsize>>>(...);
```

As mentioned before, the invocation of a kernel function initiates the generation of a grid of threads according to the given execution configuration, and the sizes of the grid and the thread blocks as well as the identifiers are stored in the variables `gridDim, blockDim, blockIdx`, and `threadIdx`. The threads generated can use these variables to execute the kernel function in a SIMD programming model, which is illustrated by the following example vector addition in Fig. 7.6. This CUDA program also illustrates the cooperation between host program and kernel function as well as the use of the execution configuration.

Example: The CUDA program in Fig. 7.6 implements the addition of two integer vectors `a` and `b` of length `N` and stores the result in integer vector `c` of length `N`. The host program starts with the declaration of the arrays `a, b, c` of length `N` on the CPU and then allocates memory of size `N * sizeof(int)` for corresponding vectors `ad, bd, cd` in the global memory of the GPU, using the CUDA function `cudaMalloc()`. Next, the input vectors `a` and `b` are copied into the vectors `ad` and `bd` in the GPU memory using the CUDA function `cudaMemcpy()` with constant `cudaMemcpyHostToDevice`. The kernel function `vecadd()` is called using the configuration `<<< 10, 16 >>>`, i.e., a grid with 10 blocks having 16 threads each is generated. The result vector `c` is copied back to the CPU memory using `cudaMempcy` with constant `cudaMempcyDeciceToHost`. The function `write_out(c)` is meant to output the result. Finally, the host programs free the data structures `ad`, `bd`, and `cd` in the GPU memory by calls to `cudaFree()`. The kernel function `vecadd` has three parameters of type pointer to `int`. Each of the threads generated in the thread grid of size 10×16 executes the function `vecadd()` in the SIMD programming model, see Sect. 3.3.2. The actual executions of the threads differ because of different thread identifiers. For each thread, a program specific thread identifier `tid` is declared and has the value `threadIdx.x + blockId.x * blockDim.x`. This results in a single one-dimensional array of program specific thread identifiers.

The computations to be executed are the additions of two values `a[i]` and `b[i], i` $\in \{0, \ldots, N-1\}$. In each step of the `for` loop, the threads process

```
#include <stdio.h>
#define N 1600

__global__ void vecadd (int *a, int *b, int *c) {
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   int i;
   for (i=tid; i<N; i += blockDim.x * gridDim.x)
      c[i] = a[i] + b[i];
}

int main( void ) {
   int a[N], b[N], c[N];
   int *ad, *bd, *cd;

   cudaMalloc ((void **) &ad, N * sizeof(int));
   cudaMalloc ((void **) &bd, N * sizeof(int));
   cudaMalloc ((void **) &cd, N * sizeof(int));
   read in (a); read_in (b);
   cudaMemcpy (ad, a, N * sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy (bd, b, N * sizeof(int), cudaMemcpyHostToDevice);
   vecadd<<<10,16>>> (ad, bd, cd);
   cudaMemcpy (c, cd, N * sizeof(int), cudaMemcpyDeviceToHost);
   write_out (c);
   cudaFree (ad);
   cudaFree (bd);
   cudaFree (cd);
}
```

Fig. 7.6 A CUDA program for the addition of two vectors `a` and `b`.

consecutive elements of the vectors `a` and `b` in parallel, starting with the element `a[i]` and `b[i]` with `i=tid`. If the length `N` of the vectors is larger than the number of threads, i.e., `blockDim.x *gridDim.x = 10 *16`, each thread executes `N/(blockDim.x * gridDim.x)` additions, assuming `N` is divisible by `blockDim.x * gridDim.x`. The assignment of threads to additions implements a cyclic data distribution for the one-dimensional arrays `a`, `b`, and `c`, see Sect. 3.5.

It should be noticed that the thread generation in CUDA is implicit in contrast to an explicit thread generation in Pthreads for example. The thread generation in CUDA is achieved by just providing an execution configuration. □

7.3 Synchronization and Shared Memory

The threads generated for the parallel execution of a kernel function can be synchronized using the barrier synchronization operation

```
__syncthreads();
```

A barrier synchronization involves all threads of the same thread block. Threads of different blocks of the same grid of threads, however, cannot be synchronized by a barrier synchronization. The barrier synchronization of one block of threads has the effect that the threads wait at the synchronization call until all other threads of the block also reach this program point. It is important that the function `__synthreads()` is actually called by all threads of the block, since the execution can only proceed if all threads of the block synchronize at this function call. This constraint requires a careful coding of `if-then-else` constructs in a program with SIMD threads such that all threads are able to reach the synchronization point.

If a call to `__synthreads()` would reside in the `then` part of an `if-then-else` construct, then only those threads executing the `then` part would reach the synchronization point and would wait for other threads of the block. However, if some of the threads do not execute the `then` part, the waiting threads can never continue their execution, since some of the threads of the block can never reach the synchronization point. Even if an `if-then-else` construct is used and both parts, the `then` part and the `else` part, contain a call to the function `__synthreads()` these two calls define different synchronization points, each of which is reached only by a subset of the threads of the blocks so that the execution cannot be continued.

The design of an execution configuration, especially the subdivision of a grid into thread blocks is essential for the potential to synchronize, as it has been explained above. Thus, we have already introduced two reasons why different execution configurations lead to different executions for the same kernel: These are the increase of the number of threads in a grid for a potential SIMD parallelism and the definition of a potential synchronization structure. Another important property to be exploited by the thread block structure is the memory organization of the CUDA programming model. This is introduced next.

The CUDA programming model provides the CPU memory and the GPU memory. The GPU memory is organized in a hierarchy of different memory types, as illustrated in Fig. 7.7. The global memory and the constant memory can be accessed by the CPU for reading or writing, as already introduced for the function `cudaMemcpy()`. The global memory can also be written and read by the GPU. The constant memory supports read-only access by the GPU, which has a short latency. The registers and the shared memory are on-chip memories and have short access times. Registers are assigned to individual threads and a thread can only access its own registers. The data stored in registers are declared as private variables.

The shared memory is assigned to an entire block of threads and the shared data within the shared memory can be accessed by all threads of the block. The threads of the same thread block can exchange data through the shared variables. The entire shared memory is broken up into fragments so that each block has its own shared memory. Figure 7.7 shows two thread blocks (0, 0) and (1, 0) and two shared memories, one for each block.

The storage of data in the different memories of the CUDA memory organization is declared explicitly in a CUDA program by a CUDA-specific declaration system. All scalar variables declared in a kernel or a device function are private variables to be stored in registers if possible. There is a copy of such a private variable for

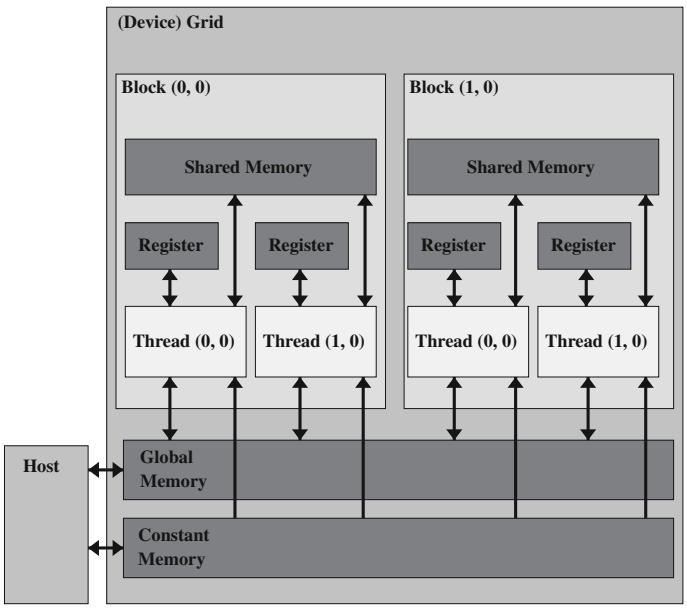

Fig. 7.7 Memory organization of the CUDA programming model containing the global, the constant, and the shared memories assigned to blocks, and registers assigned to single threads.

each thread. The lifetime of a private variable is within a kernel invocation and there is no way to access it after the kernel function has been finished and its threads are terminated. Private array variables are also declared within a kernel or a device function and a separate copy of the variable in stored for each of the threads in the global memory.

A variable declaration preceded by the keyword `__shared_` declares a shared variable, for which several versions are created, one for each block. All threads of a block can access the variable copy of their block. The lifetime of shared variables is also restricted to the execution of the kernel function. Since the access to shared variables is very fast, these variables should be used for data that are accessed frequently in the computation.

The keyword `__constant_` is used for the declaration of variables in the constant memory. These variables are declared outside any function and their lifetime is the entire execution time of the program. The size of the constant memory is currently limited to 65536 Bytes. Variables in the global memory are declared using the keyword `__global_`. Global variables can be accessed by all threads during the entire execution time of the program. The access to global variables cannot be synchronized in CUDA, i.e., there are no further synchronization mechanisms, such as locking methods. The only way to synchronize threads is the barrier synchronization introduced above. There are, however, several atomic operations for integer variables available and also an atomic addition for floating-point variables that allow

a synchronized access. For the atomic operations, the variables can reside in the global memory or, since CUDA Version 2, in the shared memory. The next example implementing the multiplication of two vectors, i.e., a scalar product, illustrates the usage of the shared memory and the block-oriented synchronization.

Example: Figure 7.8 shows a CUDA program for the multiplication of two vectors `a` and `b` of length `N`. Such a multiplication of two vectors is also called a scalar product, see Sect. 3.7. The host program for the scalar product first allocates memory for the vectors `a` and `b` in the CPU memory as well as memory for the corresponding vectors `ad` and `bd` in the global memory of the GPU. Additionally, memory space is allocated for the vector `part_c` of length `n_blocks` on the GPU. The length `n_blocks` denotes the number of thread blocks in the execution configuration of the kernel function `scal_prod()` and the vector `part_cd` is intended to store intermediate results of those thread blocks. After reading in the vectors `a` and `b`, their content is copied to the vectors `ad` and `bd` on the GPU using the function `cudaMemcpy()`. This is analogous to the program shown earlier in Fig. 7.6.

The scalar product is computed by the kernel function `scal_prod()` using an execution configuration with a one-dimensional block organization in the grid and a one-dimensional organization of the threads within each block. It is assumed that the number `threadsPerBlock` of threads in a block is a power of two and the length `N` of the input vectors is divisible by this number of threads per block. The array `part_prod` is a shared variable and is declared in the kernel function so that a private copy is created for each block.

The scalar product is calculated in three computation phases. The first two of these phases are executed by the kernel function `scal_prod()` on the GPU and the third phase is executed on the CPU providing the final result. The first phase on the GPU starts with the calculation of intermediate results of the scalar product. Each thread computes the scalar product of a part of the vectors `ad` and `bd` according to a data cyclic distribution of these vectors and stores the intermediate results at position `thread_index` in array `part_prod`. This calculation is performed in a `for`-loop whose index starts with the specific thread identifier `tid` and has a step size `blockDim.x * gridDim.x` resulting in the cyclic distribution. For the specific case in Fig. 7.8, each thread computes only one element, since the number of threads is identical to the length of the input array, i.e., `N = 32*1024`. The first phase is completed by a synchronization with `__syncthreads()`.

The second phase on the GPU adds up the intermediate results in each block. This is done in a `while`-loop, in which the number of values to be added is halved in each iteration step: In the first iteration step, half of the threads in a block add two intermediate results each and store the result back into the array `part_prod` in the position of the first operand. In the second iteration step, a quarter of the threads of a block add two values of `part_prod` and store the result back. After $\log_2$(`blockDim.x`) steps of the `while`-loop, a single thread adds the last two values resulting in the final result of the scalar product of the thread block and stores it into `part_prod[0]`. The threads to be involved actively in an iteration step of the `while`-loop are chosen according to their thread identifiers `thread_index` in the block. For a correct

```
#include <stdio.h>
const int N = 32 * 1024;
const int threadsPerBlock = 256;
const int n_blocks = N / threadsPerBlock;

__global__ void scal_prod (float *a, float *b, float *c) {
   __shared__ float part_prod[threadsPerBlock];
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   int i,size, thread_index = threadIdx.x;
   float part_res = 0;

   for(i=tid; i<N; i += blockDim.x * gridDim.x)
      part_res += a[i] * b[i];
   part-prod[thread_index] = part_res;
   __syncthreads();
   size = blockDim.x/2;
   while (size != 0) {
      if(thread_index < size)
        part-prod[thread_index] += part-prod[thread_index + size];
      __syncthreads();
      size = size/2;
   }
   if (thread index == 0)
      c[blockIdx.x] = part-prod[0];
}

int main (void) {
   float *a, *b, c, *part_c;
   float *ad, *bd, *part_cd;
   a = (float *) malloc (N*sizeof(float));
   b = (float *) malloc (N*sizeof(float));
   part_c = (float*) malloc( n_blocks*sizeof(float));
   cudaMalloc ((void **) &ad, N*sizeof(float));
   cudaMalloc ((void **) &bd, N*sizeof(float));
   cudaMalloc ((void **) &part_cd, n_blocks*sizeof(float));
   read_in (a); read_in (b);
   cudaMemcpy (ad, a, N*sizeof(float), cudaMemcpyHostToDevice);
   cudaMemcpy (bd, b, N*sizeof(float), cudaMemcpyHostToDevice);
   scal_prod<<<n_blocks,threadsPerBlock>>>( ad, bd, part_cd);
   cudaMemcpy (part_c, part_cd,
               n_blocks*sizeof(float), cudaMemcpyDeviceToHost);
   c = 0;
   for (int i=0; i<n_blocks; i++) c += part_cd[i];
   write_out (c);
   cudaFree (ad);
   cudaFree (bd);
   cudaFree (part_cd);
}
```

Fig. 7.8 CUDA program for the multiplication of two vectors `a` and `b`.

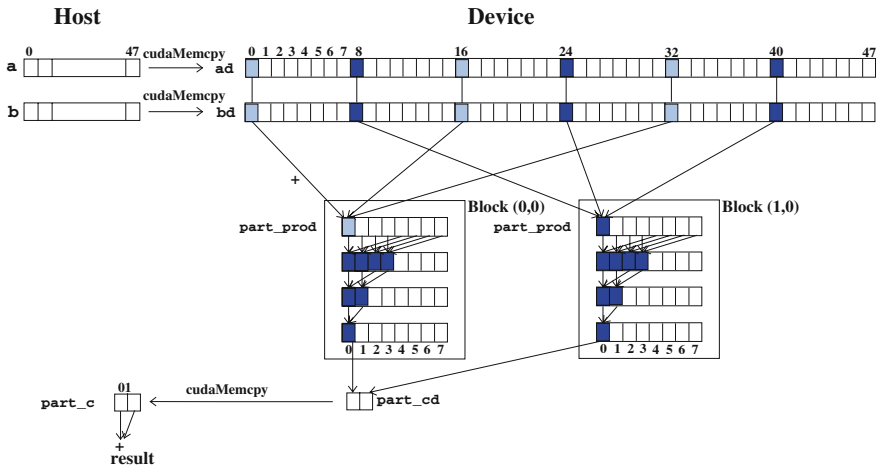

Fig. 7.9 Illustration of the computations and data structures of the CUDA program implementing a scalar product in Fig. 7.8 with a vector size `N=3*16` and an execution configuration `<<< 2, 8 >>>`, i.e., `2*8=16` threads, in the call of kernel function `scal_prod`. The figure shows the computations of the host on the left side and the computations of the GPU on the right side. The vectors `a` and `b` of length 48 are processed by 16 threads organized in the two blocks (0, 0) and (1, 0). Each of the threads computes its unique thread identification number `tid` as a number between 0 and 15. As an example it is illustrated that the thread with `tid=0` computes the partial scalar product of the vector elements 0, 16, and 32 and stores the result in the global memory of the GPU and that the thread with `tid=8` processes vector elements 8, 24, and 40. After the computations of the partial scalar products by all threads, the resulting values are accumulated in each of the blocks separately in $log_2\ 8 = 3$ steps. The result is stored in array `part_cd`, which has one component for each block. After copying this array back into the CPU memory, the CPU computes the final result of the scalar product by accumulating the two values in array `part_c`.

computation, each iteration step is completed by a synchronization of the threads in the block.

At the end of the second phase, there is a final result in the shared variable `part_prod[0]`, i.e., there is one final value for each block. These values are copied back into the CPU memory where they are stored in array `part_c` of length `n_blocks`. These values in `part_c` are accumulated on the CPU and provide the final result of the scalar product in the scalar variable `c`. Figure 7.8 shows a CUDA program implementing the three computation phases described. Figure 7.9 illustrates the computation phases for example data structures of specific sizes. □

7.4 CUDA Thread Scheduling

In typical kernel function executions, the number of threads generated for that kernel exceeds the number of compute units, i.e., the number of streaming multiprocessor or SIMD function units, respectively, see Sect. 7.1. Therefore, there has to be a

methodology to assign threads to compute units. This is implemented by the CUDA thread scheduling. The CUDA thread scheduling exploits that different thread blocks of a grid are independent of each other and can be executed in any execution order. The scheduling of thread blocks can assign as many executable thread blocks to SIMD processors as possible for the specific hardware.

Basically, the threads within the same thread block are also independent of each other if there is no barrier synchronization. However, the CUDA system does not schedule individual threads but uses an additional concept to divide blocks into smaller sets of threads, called warps, and schedules these warps. For current GPU architectures, the size of warps is 32. Threads of a warp are assigned and processed together and the scheduling manages and determines the execution order. The CUDA thread scheduling is part of the CUDA architecture and programming model and cannot be influenced directly by the application programmer.

The division of thread blocks into warps is based on the thread index `threadIdx`. For a one-dimensional block, 32 threads with consecutive values of `threadIdx.x` are combined to form one warp. If the number of threads is not divisible by 32, then the last warp is filled with virtual threads. A two-dimensional block is first linearized in a rowwise manner and then consecutive threads are combined for a warp. Analogously, a three-dimensional block is linearized and subdivided into warps. Thus, depending on the organization of a grid into blocks the same set of threads can result in different divisions into warps and a different scheduling.

Warps are executed in the CUDA computation model SIMT (single instruction, multiple threads). In this model, the hardware executes the same instruction for all threads of a warp and only then proceeds to the next instruction. The SIMT computational model is suited to exploit the hardware efficiently, see Sect. 7.1, especially if all threads of the warp have the same flow of control. If threads of the same warp have different control flow paths in an `if-then-else` construct, since some of the threads execute the `then` part and the other threads execute the `else` part, then the threads have different control flow paths, which have to be executed one after another in the SIMT model. This leads to longer execution times and should be avoided by a more suitable definition of threads. As an example, we consider the reduction operation accumulating the elements of an array in parallel. Such a reduction operation has been used for the second phase in the scalar product in Fig. 7.9.

In the loop implementing the reduction operation, the number of threads executing an iteration is reduced in every iteration step so that only half of the threads continue the computation in the next iteration. The other threads do not execute any operation. This leads to the situation that the threads have different control flow paths, which are either an addition operation or no operation. If the reduction operation is implemented such that each thread adds neighboring array elements, then each warp contains threads with two different control flow paths. In contrast, if the addition of array elements is organized as shown in the reduction operation of Phase 2 of the scalar product, see the illustration in Fig. 7.8, then all threads with smaller `threadIdx.x`-values perform an addition while threads with larger `threadIdx.x`-values do not perform any operation. For large numbers of threads this means that in the earlier iterations of the reduction the warps containing 32 threads each will contain

either only threads performing an addition or only threads performing no operation. When the number of threads executing an addition decreases towards the end of the reduction and is finally smaller than 32, then this final warp has threads with different control flow paths.

7.5 Efficient Memory Access and Tiling Technique

The execution of a kernel function usually requires to access a lot of data from the global memory, caused by the data parallel execution model of the threads. Since these global memory accesses are expensive, data should be copied into the shared memory or the registers, which have a much faster access time. To support copy operations from the global memory into the shared memory or registers, CUDA provides a technique that combines neighboring data and copies them together, resulting in fast copy operations. This technique is called memory coalescing. The coalescing technique exploits the fact that at any point in time threads of a warp execute the same instruction. If this instruction is a load operation, then the hardware can detect whether the parallel load operations address neighboring memory locations in the global memory. The hardware then combines the load operations of neighboring memory locations to only one memory access, which is much faster than several single memory accesses.

To exploit the coalescing technique for efficient memory access, the application programmer should organize the data in the CUDA program in such a way that threads with neighboring thread identifiers to be combined in the same warp access neighboring elements of the arrays used in the program. This means that n threads $T_0, T_1, \ldots, T_{n-1}$ should access consecutive memory locations $M, M+1, \ldots, M+n-1$, where M denotes the first address of an array.

For the access to a two-dimensional array, threads with consecutive thread identifiers should access neighboring element in the rows. The row-wise storage of two-dimensional arrays then enables the hardware to combine the data access to one copy operation. In contrast, if threads in the same warp would access neighboring elements in the columns of an array, then a coalescing of the data accesses would not be possible, since neighboring column elements are stored in different places in the memory. More precisely, for a two-dimensional array with m columns, the elements of the same column are stored in memory locations with a distance of m, which avoids coalescing.

The data layout of CUDA programs should be designed such that coalescing is possible so that efficient programs can result. A well-known programming technique, which can be used for this purpose, is the tiling technique. In the tiling technique, a two-dimensional array is decomposed into smaller two-dimensional arrays of the same size, which are called tiles. The algorithm using the array has then to be modified such that the program execution fits to the tile structure. This means especially that nested loops that access the two-dimensional data structure are modified so that smaller data sets are processed. The tiling techniques are illustrated in the following

```
#include <stdio.h>
typedef float * Matrix;
const int N = 32 * 32;
__global__ void MatMulKernel(const Matrix A,const Matrix B,Matrix C){
   float Cval = 0;
   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;
   for (int e = 0; e < N; ++e)
     Cval += A[row * N + e] * B[e * N + col];
   C[row * N + col] = Cval;
}
void MatMul (const Matrix A, const Matrix B, Matrix C){
   int size = N * N * sizeof(float);
   Matrix Ad,Bd,Cd;
   cudaMalloc (&Ad, size); cudaMalloc(&Bd, size);
   cudaMemcpy (Ad, A, size, cudaMemcpyHostToDevice);
   cudaMemcpy (Bd, B, size, cudaMemcpyHostToDevice);
   cudaMalloc (&Cd, size);
   dim3 dBlock(32,32);
   dim3 dGrid(N/32,N/32);
   MatMulKernel<<<dGrid,dBlock>>> (Ad,Bd,Cd);
   cudaMemcpy (C,Cd,size, cudaMemcpyDeviceToHost);
}
int main() {
   Matrix A, B, C;
   A = (float *) malloc(N * N * sizeof(float));
   B = (float *) malloc(N * N * sizeof(float));
   C = (float *) malloc(N * N * sizeof(float));
   read_in(A); read_in(B);
   MatMul (A,B,C);
   write_out (C);
   cudaFree (Ad); cudaFree (Bd); cudaFree (Cd);
}
```

Fig. 7.10 CUDA program implementing a matrix multiplication of two N×N matrices A and B. The matrix size N is assumed to be a multiple of 32.

example program, which is a matrix–matrix multiplication. First, a CUDA program without tiling technique is given in Fig. 7.10, and then the tiling technique is introduced in Fig. 7.13.

Example: The CUDA program `matmult.cu` in Fig. 7.10 contains a host program and the kernel function `MatMulKernel()` implementing a matrix multiplication of two matrices A and B of size N×N. The result is stored in matrix C. The host function `MatMul()` captures the interaction of the CPU with the GPU using copy operations and the call of the kernel function `MatMulKernel()`. In the parallel implementation, each element of the result matrix C is computed by a separate thread, i.e., for computing an element of C in row i and column j, this thread computes a scalar product of the i-th row of A and the j-th column of B. Thus, the number of threads generated for the execution of `MatMulKernel` corresponds to the number of elements

in C. The execution configuration specifies two-dimensional blocks of size 32×32, i.e., the configuration `dim3 dBlock(32, 32)` is chosen. The size of the two-dimensional grid is defined such that the number of threads generated fits the number N×N of matrix elements, i.e., the grid size is `dim3 dGrid (N/32, N/32)`.

The arrays Ad, Bd, and Cd to be used on the GPU are allocated in the global memory of the GPU from which they are accessed by the threads. For the assignment of threads to the computation of an element of the result matrix, a specific row index `row` and a specific column index `col` are used based on the thread identifier. The variables `row` and `column` are private variables so that there exists a separate copy for each thread. The program uses a linear storage of the arrays in row-major order. The computation of one scalar product is implemented in a `for`-loop with the loop body

```
Cval= Cval+A [row*N+e]*B[e*N+col].
```

The value of the private variable `Cval` is then copied into the global array C at the appropriate position `row * N + col`. □

The matrix multiplication given in Fig. 7.10 accesses the data directly in the global memory, which results in a high data access time. This data access time can be hidden by a high number of threads so that always some threads can be executed while others access data. But still the bandwidth for accessing the global memory may limit the overall performance. As an example, we consider the NVIDIA GTX 680, which has a maximum memory bandwidth of 192 GB/sec, i.e., in each second, $192 * 10^9$ Bytes or equivalently $48 * 10^9$ floating-point values of single precision (32 bit) can be loaded from the global memory. The loop body of the `for`-loop in the kernel function `MatMulKernel()` in Fig. 7.10 requires two data accesses and performs two arithmetic operations. Since only $48 * 10^9$ floating-point values can be loaded per second, this means that a maximum of $48 * 10^9$ floating-point operations can be performed per second. However, the NVIDIA GTX 680 has a maximum performance of 3090 GFLOPS, see Table 7.1, which means that only about 1.6 % of the maximum performance is exploited.

The efficiency of a CUDA program can be increased by first loading data into the shared memory and then accessing them from this faster memory. This is advantageous if data are used several times by the threads of the same block. For the matrix multiplication, there is a multiple use of data, since the threads of a block of size 32×32 access 32 rows of matrix A and 32 columns of matrix B several times, see also [118]. The access pattern is illustrated in Fig. 7.11 for matrices of size 12×12 with 16 thread blocks of size 3×3. The thread blocks (0,0) and (1,2) are highlighted. As example, we first consider thread block (0,0). Thread (0,0) of this block computes `C[0,0]` and accesses row 0 of A and column 0 of B for this computation. These accesses are illustrated by the arrows in the figure. The arrows also indicate the access order to the rows of A and the columns of B. Thread (1,0) computing `C[0,1]` also accesses row 0 of A but column 1 of B, which is a multiple use of row 0 of A. Similarly, thread (2,0) also accesses row 0 of A for its computation of `C[0,2]`. Thus, for the computation of an entire block with nine elements of the result matrix C, the nine threads of thread block (0,0) need to access only six vectors, three rows of A (indicated by X), and three columns of B (indicated by Y). Similar accesses are performed for thread block (1,2) as well as the other thread blocks. Notice that the