GP-GPU

General Purpose GPU Programming

Salvatore Orlando
Manycore

- Manycore GPUs (Graphics Processing Units) are available in almost all current hardware platforms.

- Originally, these processors have been designed for graphics applications.
  - Because of the high potential of data parallelism in graphics applications, the design of GPU architectures has used several specialized processor cores (manycores).

- In addition to graphics processing, GPUs can also be employed for general non-graphics applications.
  - If data parallelism is large enough to fully utilize the high number of compute cores in a GPU.

- The trend to use GPUs for general numerical applications has inspired GPU manufacturers, such as NVIDIA, to develop the programming environment CUDA and OpenCL.
CUDA

- 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
  - then the same host program calls device functions to initiate the processing on the GPU.

- Device functions take the parallel architecture of the GPU into account
  - to appropriately map the data parallel computations to compute cores.
  - to appropriately consider memory organization of the GPU and the specific data transfer between CPU and GPU
GPU architecture

- A GPU comprises several multi-threaded SIMD processors
  - **SIMD processor** = Streaming Multiprocessors (SMs) composed of many Streaming Processors (SPs)
- Each of the SIMD processors has several function units (cores), which can execute the same SIMD instruction on different data
- The actual number of SIMD processors depends on the GPU model.
  - For example, the NVIDIA GTX480 GPU belonging to the family of the Fermi-architecture has up to 15 independent SMs
GPU architecture

• Each SM 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

• At first glance, it seems to be sufficient to provide one thread of control (SIMD thread) for each SIMD processor
  – parallelism results because each functional unit of the SIMD processor executes the same instruction on different data

• However, the data transfer operations will cause waiting times of uncertain length.
GPU architecture - multithreading

• To hide memory 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
GPU architecture - multithreading

- 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
  - contains for each SIMD thread the current instruction to be executed
  - 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
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 registers...
GPU architecture - multithreading

• The real design of a GPU of the Fermi-architecture is more complex than illustrated in Fig. 7.1
  – Each SIMD processor employs two SIMD thread schedulers (and not only one) with corresponding dispatch units for launching the SIMD instructions.

• 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.
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.

<table>
<thead>
<tr>
<th>GPU</th>
<th>GTX 285 (Tesla)</th>
<th>GTX 480 (Fermi)</th>
<th>GTX 580 (Fermi)</th>
<th>GTX 680 (Kepler)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistors</td>
<td>1, 4 $\cdot 10^9$</td>
<td>3, 2 $\cdot 10^9$</td>
<td>3, 0 $\cdot 10^9$</td>
<td>3, 54 $\cdot 10^9$</td>
</tr>
<tr>
<td>SIMD processors</td>
<td>30</td>
<td>15</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>SIMD cores per SIMD processor</td>
<td>8</td>
<td>32</td>
<td>32</td>
<td>192</td>
</tr>
<tr>
<td>total number of SIMD cores</td>
<td>240</td>
<td>480</td>
<td>512</td>
<td>1536</td>
</tr>
<tr>
<td>L2-Cache</td>
<td>/</td>
<td>768 KB</td>
<td>768 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>Performance</td>
<td>1063 GF</td>
<td>1344 GF</td>
<td>1581 GF</td>
<td>3090 GF</td>
</tr>
<tr>
<td>Bandwidth memory</td>
<td>159 GB/sec</td>
<td>177 GB/sec</td>
<td>192 GB/sec</td>
<td>192 GB/sec</td>
</tr>
<tr>
<td>clock rate memory</td>
<td>2484 MHz</td>
<td>3696 MHz</td>
<td>4008 MHz</td>
<td>6008 MHz</td>
</tr>
<tr>
<td>power consumption</td>
<td>204 W</td>
<td>250 W</td>
<td>244 W</td>
<td>195 W</td>
</tr>
</tbody>
</table>
2010 - Fermi Architecture

Diagram of the Fermi architecture showing the components: Instruction Cache, Scheduler, Dispatch, Register File, Core, Core, Core, Core, Load/Store Units x 16, Special Func Units x 4, Interconnect Network, 64K Configurable Cache/Shared Mem, Uniform Cache, DRAMIF, HOST IF, Clip Thread, L2.
2012 – Kepler Architecture
CUDA PROGRAMMING MODEL
CUDA basics

- The computing system consists of:
  - a **HOST** running serial or modestly parallel C code
  - one or more **DEVICES** running *kernel* C code
  - exploiting massive data parallelism
    - the program property where many operations can be safely performed on the distinct elements of data structures in a simultaneous manner

```
SERIAL CODE
(HOST)

PARALLEL KERNEL
(DEVICES)
```
CUDA Devices and threads

• A compute device
  – Is a coprocessor to the CPU or host
  – Has its own DRAM (device memory)
  – Runs many threads in parallel
  – Is typically a GPU but can also be another type of parallel processing device

• Data-parallel portions of an application are expressed as device kernels which run on many threads

• Differences between GPU and CPU threads
  – GPU threads are extremely lightweight
    • Very little creation overhead
  – GPU needs 1000s of threads for full efficiency
    • Multi-core CPU needs only a few
The thread hierarchy (bottom-up)

- A **KERNEL** is a data parallel function to be executed on the device. When called, it is executed $N$ times in parallel by $N$ different CUDA **THREADS**.

- Threads are organized in **BLOCKS**:
  - Threads in the same block (at most 2048 threads) share the **same SIMD processor (SP)** and its **resources**.
  - Threads in a block can share data through some **shared memory**
  - Threads have a 1/2/3-dimensional identifier:
    - threadIdx.x, threadIdx.y, threadIdx.z
The thread hierarchy (bottom-up)

- Blocks are organized in **GRIDS**:
  - The number of thread blocks is usually dictated by the problem size or the number of SIMD processors, which they can greatly exceed.
  - A thread block size of 16x16 (256 threads), although arbitrary in this case, is a common choice.
  - Blocks have a 1/2-dimensional identifier (till ver. 3):
    - blockIdx.x, blockIdx.y
  - Blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series.
    - This independence requirement allows blocks to be scheduled in any order across any number of cores
The thread hierarchy (top-down)

- **A GRID** is a piece of work that can be executed by the GPU
  - A 2D array of BLOCKs (3D since CUDA Version 3)

- **A BLOCK** is an independent sub-piece of work that can be executed in any order by a SIMD processor (**Streaming Multiprocessor SM**)
  - A 3D array of threads
  - Max no. of threads in a block: 512 threads (up to Version 2) and 1024 threads (since Version 3).

- **A THREAD** is the minimal unit of work.
  - All the threads execute the same **KERNEL** function
  - Threads are grouped in **WARPS** of 32 for scheduling, i.e. they are the minimal units of scheduling
Fake “Hello World!” example

```c
__global__ void kernel( void ) { }

int main( void ) {
    kernel<<<2,3>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

- **__global__**
  declare a kernel function to be run on the GPU
- **kernel<<<2,3>>>();**
  runs 2 blocks with 3 threads each executing the function `kernel`
Fake “Hello World!” example

__global__ void kernel( void ) { }

int main( void ) {
    // set the grid and block sizes
    dim3 dimGrid(3,1);  // a 3x1 array of blocks
    dim3 dimBlock(2,2,2); // a 2x2x2 array of threads

    // invoke the kernel
    kernel<<<dimGrid, dimBlock>>>();
    printf( "Hello, World!\n" );
    return 0;
}

• dim3 is the data type used to declare grid and block sizes
Memory

- **Global / Constant Memory** is the on-board device memory
  - Data transfers occur between **Host memory** and **Global memory**
  - It is accessible by any thread
  - It is (relatively) costly
  - **Constant memory** (64K) 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.

- **Shared Memory** is shared by the threads in the same block
  - Provides fast access but it is very limited in size

- **Registers** are private to threads
  - All scalar variables declared in a kernel are private and stored in registers if possible
Memory

• `cudaMalloc()`
  – Allocates memory in the device **Global Memory**
  – Parameters: Address of a pointer to the allocated object, Size of allocated object

• `cudaFree()`
  – Frees memory from device Global Memory
  – Parameters: Pointer to freed

• `objectcudaMemcpy()`
  – Memory data transfer
  – Parameters: Pointer to destination, Pointer to source, Number of bytes copied
    • Type of transfer: Host to Host, Host to Device, Device to Host, Device to Device
  – Asynchronous transfer
“Hello World!” example (1)

```c
// Host function
int main(int argc, char** argv) {
    // desired output
    char str[] = "Hello World!";
    // mangle contents of output
    // the null character is left intact
    for(int i = 0; i < 12; i++)
        str[i] -= i;

    // allocate memory on the device
    char *d_str;
    int size = sizeof(str);
    cudaMalloc((void**)&d_str, size);

    // copy the string to the device
    cudaMemcpy(d_str, str, size,
               cudaMemcpyHostToDevice);

    // set the grid and block sizes
    dim3 dimGrid(3);  // 3 blocks
    dim3 dimBlock(4); // 4 threads

    // invoke the kernel
    helloWorld<<< dimGrid,
                dimBlock >>>(d_str);

    // retrieve the results
    cudaMemcpy(str, d_str, size,
                cudaMemcpyDeviceToHost);

    // free allocated memory
    cudaFree(d_str);
    printf("%s\n", str);
    return 0;
}
```
`__global__
void helloWorld(char* str)
{
    // determine where in the thread grid we are
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // unmangle output
    str[idx] += idx;
}
SUMMARY ON THREADS, BLOCKS, ETC.
Function declaration

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float DeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong> void KernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong> float HostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

__global__ defines a kernel function

__device__
- No recursion
- No static variables inside the function
- No variable number of arguments

__device__ and __host__ can be used together, in which case the function is compiled for both the host and the device.
Invoking a kernel function

```c
__global__ void KernelFunc(...);
dim3 DimGrid(100, 50);    // 5000 blocks
dim3 DimBlock(4, 8, 8);    // 256 threads per block
KernelFunc<<< DimGrid, DimBlock >>>(...);
```

- **Kernel calls are asynchronous**
- `gridDim.x/y` `blockIdx.x/y` `blockDim.x/y/z` `threadIdx.x/y/x` identify threads and blocks in a grid within kernel function

- **Sizes and limitations (Compute capability 1.0):**
  - Warp size = 32; Threads per block = 512;
  - Warps per SM = 24; Blocks per SM = 8;
  - Threads per SM = 768
Automatic (Transparent) Scalability

- Do we need to take care of the device computing power (number of SM)?
  - No, because the block scheduler can re-arrange blocks accordingly
  - A Grid contains a set of independent blocks, which can be executed in any order
VECTOR ADD EXAMPLE
Objective

Blocks

Threads
// Device kernel
__global__
void blockthreadAdd( int* a,
                     int* b,
                     int* c)
{
    // position in the block/grid
    int i = threadIdx.x +
            blockIdx.x*blockDim.x;
    c[i] = a[i] + b[i];
}
// Host function
int main(int argc, char** argv) {
    int *a, *b, *c;  // vectors
    int N = 1024;    // vectors length
    int size = N*sizeof(int);  // memory size

    // allocate memory
    a = new int[N];
    b = new int[N];
    c = new int[N];

    // initialize
    for (int i=0; i<N; i++) {
        a[i] = rand() % 1000;
        b[i] = rand() % 1000;
    }

    Allocate memory on host
    Initialize with random numbers
Vector add (3)

// allocate memory on device
int *dev_a, *dev_b, *dev_c;
cudaMalloc((void**)&dev_a, size);
cudaMalloc((void**)&dev_b, size);
cudaMalloc((void**)&dev_c, size);

// copy input on device
cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

// launch kernels with 8 threads each
blockthreadAdd<<<N/8,8>>>(dev_a, dev_b, dev_c);

// copy result on host
cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

- Allocate memory on device
- Copy input from host memory to device memory
- Invoke kernel
  - the number of threads depend on the input size
- Get results back
```cpp
// free memory on device
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);

// free memory
delete [] a;
delete [] b;
delete [] c;

return 0;
}
```

- Clean up memory on device and on host
- The end.
Compilation & Execution

- 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 also provides a set of machine commands that ensure the compatibility of different generations of NVIDIA GPUs.
Compilation & Execution

• The execution of a CUDA program starts by first running 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.

• In order to execute a kernel function on a device, the data have to reside in the device memory.
DOT PRODUCT EXAMPLE
Dot Product: Objective

- How to split the work among blocks and threads?
  1. Every block should compute a partial sum
  2. Sum the partial sums
Dot Product

// Device kernel
__global__ void dotproduct( int* a, int* b, int* c) {
    // position in the block/grid
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    // compute product and store it in block-shared memory
    // wait all threads to compute their product
    // make one thread in the warp to compute the block sum
    // increase safely the global current sum
}
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. Lifetime of shared variables is restricted to the execution of the kernel function. Access to shared variables is very fast, so use these variables for data that are accessed frequently in the computation.
#define THREADS_PER_BLOCK 512

// Device kernel
__global__ void dotproduct( int* a, int* b, int* c) {
    __shared__ int temp[THREADS_PER_BLOCK];
    // position in the block/grid
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    // compute product and store it in block-shared memory
    temp[threadIdx.x] = a[i]*b[i];
    // wait all threads to compute their product
    __syncthreads();
    // make one thread in the warp to compute the block sum
    // increase safely the global current sum
}
Dot Product

```
#define THREADS_PER_BLOCK 512

// Device kernel
__global__ void dotproduct( int* a, int* b, int* c) {
    __shared__ int temp[THREADS_PER_BLOCK];
    // position in the block/grid
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    // compute product and store it in block-shared memory
    temp[threadIdx.x] = a[i]*b[i];
    // wait all threads to compute their product
    __syncthreads();
    // make one thread in the warp to compute the block sum
    if (threadIdx.x==0) {
        int sum = 0;
        for (int j=0; j<THREADS_PER_BLOCK; j++)
            sum += temp[j];
    }
    // increase safely the global current sum
}
```
#define THREADS_PER_BLOCK 512

// Device kernel
__global__ void dotproduct( int* a, int* b, int* c) {
    __shared__ int temp[THREADS_PER_BLOCK];
    // position in the block/grid
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    // compute product and store it in block-shared memory
    temp[threadIdx.x] = a[i]*b[i];
    // wait all threads to compute their product
    __syncthreads();
    // make one thread in the warp to compute the block sum
    if (threadIdx.x == 0) {
        int sum = 0;
        for (int j=0; j<THREADS_PER_BLOCK; j++)
            sum += temp[j];
        // increase safely the global current sum
        atomicAdd(c, sum);
    }
}
Summary

• **Atomic functions**
  - perform a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory

  ```
  atomicAdd()  atomicSub()  atomicExch()
  atomicMin()  atomicMax()  atomicInc()
  atomicDec()  atomicAnd()  atomicOr()  atomicXor()
  ```

• `__syncthreads()`
  - Barrier for threads in the same block, possibly in different warps.
• Each thread can:
  – Read/write per thread registers
    • Automatic variables
  – Read/write per block shared memory
    __shared__ qualifier
  – Read/write per grid global memory
    __device__ qualifier
  – Read per grid constant memory
    __constant__ qualifier
const int N = 32 * 1024;
const int N1 = 1024
const int threadsPerBlock = 256;
const int n-blocks = N1 / threadsPerBlock;

__global__ void scal_prod (float *a, float *b, float *c) {
    // perform dot product, where all the thread of the
    // block participates in the reduce computation
}


__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, t_Idx = threadIdx.x;
    float part-res = 0;
    for (i = tid; i < N; i += blockDim.x * gridDim.x) {
        part-res += a[i] * b[i];
        part-prod[t_Idx] = part-res;
    }
    __syncthreads();
    size = blockDim.x/2;
    while (size != 0) {
        if (t_Idx < size) {
            part-prod[t_Idx] += part-prod[t_Idx + size];
            __syncthreads();
            size = size/2;
        }
    }
    if (t_idx == 0)
        c[blockIdx.x] = part-prod[0];
}
int main (void) {
    a = (float*) malloc (N*sizeof(float));
    b = (float*) malloc (N*sizeof(float));
    part-c = (float*) malloc( n-blocks*sizeof(float));
    cudaMemcpy ((void **) &ad, N*sizeof(float));
    cudaMemcpy ((void **) &bd, N*sizeof(float));
    cudaMemcpy ((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);
    scaled 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]; // final red.
    write-out (c);
    cudaFree (ad);  cudaFree (bd);  cudaFree (part_c);
}

Dot Product 2
Dot Product 2

Vector size $N=3\times16$ and an execution configuration $<<<2,8>>$, i.e., $2\times8=16$ threads, in the call of kernel function `scal_prod`. 

Matrix multiplication and data layout

```c
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);
}
```
Matrix multiplication and data layout

#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);
    cudaMemcpy (&Cd, size);
    dim3 dBlock(32,32);
    dim3 dGrid(N/32,N/32);
    MatMulKernel<<<dGrid,dBlock>>> (Ad, Bd, Cd);
    cudaMemcpy (C, Cd, size, cudaMemcpyDeviceToHost);
}

- A specific row index \(row\) and a specific column index \(col\) are used based on the thread identifier.

- Data are directly accessed in the global memory: the for-loop accessing the global memory may limit the overall performance.

- The for-loop in the kernel function requires two data accesses and two arithmetic operations.
Matrix multiplication and data layout

Fig. 7.11 Illustration of the access pattern of a matrix-matrix multiplication with input matrices $A$ and $B$ and result matrix $C$ of size 12 × 12 and 16 thread blocks of size 3 × 3. Array indices are denoted using the standard matrix indices, but the indices of the threads in the thread blocks use the CUDA-specific numbering, as given in Fig. 7.5.

To take advantage of this data reuse for the matrix multiplication using 32 × 32 thread blocks, sub-arrays of size 32 × $N$ of $A$ and of size $N$ × 32 of $B$ could be preloaded into the shared memory so that all data accesses of a thread block address the shared memory and, thus, the memory access time is reduced. However, since the size of the shared memory is limited, a sub-array of size 32 × $N$ might be too large to fit into this memory. In such cases, the sub-array to be reloaded should be smaller while being still large enough to provide data accesses from the shared memory. This can be achieved by the tiling programming technique, see Sect. 4.6.

For the matrix multiplication from Fig. 7.10, the tiling technique can be implemented by subdividing the sub-array of size 32 × $N$ further into tiles of size 32 × 32, which are preloaded into the shared memory one after another. To process such a tile before other data of the same sub-array are needed, the matrix multiplication program has to be restructured in such a way that data in the preloaded tile are accessed consecutively, i.e., locality of data accesses is required, see Sect. 2.7. This is achieved by dividing the scalar product to be computed by each tile into sub-scalar products using only the vector of length 32 loaded with the tile. An appropriate ordering of elements to be loaded can even exploit the coalescing of the hardware to further increase the efficiency.
Matrix multiplication and data layout

• The efficiency of a CUDA program can be increased
  – By first loading data into the shared memory
  – Then accessing them from this faster memory.
• Advantageous if data are used several times by the threads of the same block
• For matrix multiplication, multiple use of data
  – the threads of a block of size $32 \times 32$ access 32 rows of matrix A and 32 columns of matrix B several times
Matrix multiplication and data layout

• To take advantage of this data reuse for the matrix multiplication using $32 \times 32$ thread blocks
  – sub-array of size $32 \times N$ of $A$
  – sub-array of size $N \times 32$ of $B$

could be preloaded into the shared memory, so that all data accesses of a thread block address the very fast shared memory

• However, since the size of the shared memory is limited
  – a sub-array of size $32 \times N$ might be too large to fit into shared memory.

• Tiling technique to solve this issue
  – subdivide the sub-array of size $32 \times N$ further into tiles of size $32 \times 32$, which are preloaded into the shared memory one after another
Matrix multiplication : tiling and data reuse

#define TILE_WIDTH 32
__global__ void MatMulTileKernel (Matrix A, Matrix B, Matrix C) {
    --shared-- float Ads[TILE_WIDTH][TILE_WIDTH];
    --shared-- float Bds[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float Cval = 0.0;
    for (int m = 0; m < N/TILE_WIDTH; m++) { /* loop over tiles */
        Ads[ty][tx] = A[Row*N + (m*TILE_WIDTH + tx)];
        Bds[ty][tx] = B[(m*TILE_WIDTH + ty)*N + Col];
        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; k++) /* loop within tile */
            Cval += Ads[ty][k] * Bds[k][tx];
        __syncthreads();
    }
    C[Row * N + Col] = Cval; /* write back to global memory */
}

• The tiles Ads and Bds have size TILE_WIDTH × TILE_WIDTH. Each thread is responsible for the computation of one element of the result matrix C.
Matrix multiplication: tiling and data reuse

Fig. 7.12 Illustration of the matrix-matrix multiplication with tiling technique for tiles of size $3 \times 3$. Figure 7.12 illustrates the tiling technique for $12 \times 12$ matrices using $3 \times 3$ tiles, i.e., there are four tiles in each dimension of the matrices. In the figure, tile $X$ of matrix $A$ and tile $Y$ of matrix $B$ are used for the sub-computation of tile $T$ of the output matrix $C$. However, these are not all the tiles needed for the computation of $T$ and $S$. For tile $R$ of matrix $C$, all tiles of matrix $A$ and of matrix $B$ that are needed for the complete computation of tile $R$ are highlighted. As shown in the figure, these are the tiles 1, 2, 3, and 4 of $A$ and of $B$.

Example: The CUDA kernel function in Fig. 7.13 implements the tiling technique for the matrix multiplication by preloading tiles $A_{ds}$ and $B_{ds}$ into the shared memory, see [118]. The tiles $A_{ds}$ and $B_{ds}$ have size $TILE_{\text{WIDTH}} \times TILE_{\text{WIDTH}}$. Each thread is responsible for the computation of one element of the result matrix $C$. The indices of the element to be computed by a thread is determined according to their block and thread identifier. Since these values are accessed often during the computation, they are loaded into private variables $bx$, $by$, $tx$, and $ty$, for which each thread has private copies to be stored in its registers.
ADVANCED THREAD AND MEMORY MANAGEMENT
Warp & SIMT/SIMD

• 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.
  - A two-dimensional block is first linearized in a rowmajor manner and then consecutive threads are combined for a warp.
  - ....

• 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 (**SIMD**) and only then proceeds to the next instruction.

• If threads of the same warp have different control flow paths (**Thread Divergence**) in an `if-then-else` construct, the different control flow paths have to be executed one after another in the SIMT model.
Thread divergence

```c
unsigned int index = ( blockDim.x * blockIdx.x ) + threadIdx.x;
float value = 0.0f;

if ( threadIdx.x % 2 == 0 )

    value = PathA( src );

value = PathB( src );

dst[index] = value;
```
Efficient Memory Access

- **Global memory accesses are expensive**
  - data should be copied into the shared memory or the registers, which have a much faster access time

- **Memory coalescing**
  - Threads of a warp execute the same instruction.
  - If this instruction is a load/store operation, then the hardware can detect whether the parallel operations address neighboring memory locations in the global memory.
  - The hardware then combines many operations directed to neighboring memory locations to only one memory access
  - This is much faster than several single memory accesses.
Efficient Memory Access

• How do we obtain memory coalescing?

• One-dimensional array
  – $n$ threads $T0, T1, ..., Tn-1$
  – They should access consecutive memory locations $M, M+1, ..., M+n-1$
    where $M$ denotes the address of the 1st element of an array, $M+1$ the address of the 2nd element, and so on.

• Two-dimensional arrays
  – Supposing row-major order memory data layout, threads with consecutive thread identifiers should access neighbouring elements in the rows.

• The data layout of CUDA programs should be designed such that coalescing is possible
Shared Memory (2.0)

- 32 memory banks organized such that successive words reside in different banks
  - INTERLEAVING MEMORY ORGANIZATION
- Each bank has a bandwidth of 32 bits per 2 clock cycles
- 32 adjacent words are accessed in parallel from 32 different memory banks
- A possible bank conflict occurs if two threads access to different words within the same bank
- When multiple threads access the same word
  - A broadcast occurs in case of read (OK)
  - Only one threads writes (which is undetermined)