GPU Programming using CUDA Overview

Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

Course Overview

- Basics
  - Writing and executing GPU code
  - Parallel programming model
  - Transfer memory from/to device (video card)
  - Synchronization, error handling, …
- Performance Optimization I
  - Optimizing usage of parallelism in the hardware of the GPU
  - Single Instruction Multiple Thread (SIMT)
  - Optimization of bandwidth of device memory (global memory)
  - Transfer of data host↔device
  - Instruction throughput
- Shared Memory
  - What is shared memory?
  - Syntax & applications
  - Reductions
  - Exercise: scalar product
- Atomic Operations
  - What are atomic operations?
  - Syntax & applications
  - Exercises: reduction, particle sort
- Performance Optimization II
  - Optimizing shared memory accesses
  - CUDA-GDB
  - Visual Profiler
  - Kepler architecture
  - Exercise: matrix-matrix multiplications
- Advanced Features
  - Texture Memory & Constant Memory
  - Streams
  - Using multiple GPUs
  - Zero-copy host memory access
  - IEEE
  - Exercise: using multiple GPUs
- Numerical Libraries
  - CUBLAS, CUSPARSE, CUFFT, CURAND
- OpenACC
  - Alternatives to CUDA
  - Introduction to accelerator programming with OpenACC

- This course and its slides have been developed by
  Oliver Mangold
  and is now continued by
  Thomas Baumann and Mhd. Amer Wafai

GPU Programming using CUDA Basics

Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

Some Terms

- Host
  - The PC system containing the video card
  - Refers typically to **CPU + memory on the main board**
    - Host program, host memory, etc.
- Device
  - Video/accelerator card
  - Consists of **GPU + memory on the video board**
- Global memory/Device memory
  - Memory **on the video board**
  - **Off-chip** from the GPU
  - Not(!) host memory (on the main board)
- Multiprocessor
  - A ‘core’ (computing element with it’s own program flow) of the GPU
  - Multi: operates on multiple data at the same time
    - SIMT: single instruction multiple thread (similar to SIMD, details later)
- Compute Capability
  - Hardware ‘Version’ of the Graphics Processor
    - The Tesla C2050 (**Fermi architecture**) has compute capability **2.0**
    - The GTX 680 **(Kepler architecture**) has compute capability **3.0**
    - The K20x (**Kepler Architecture**) has compute capability **3.5**
    - See PG Appendix G for CC features, see Appendix A for CC of newer video cards

CUDA programs

- A CUDA program is usually organized into
  - A host program
    - Controls overall program flow
    - Manages memory transfers from/to GPU
  - One or multiple kernels running on the GPU
    - Contains the fast parallel computation code

- ‘Accelerator’ programming model
  - CPU = master
  - GPU = slave
  - CPU and GPU can work independently but CPU has control
    - Parallelism between CPU and GPU: synchronization is necessary → discussed later

Creating GPU code (C)

- **Kernels:**
  - A C function can be declared to be **device code callable from host** by adding the **__global__** declaration specifier:

```
__global__ void VectorSum(float* inputA,
                          float* inputB, float* output, int size);
```

  - Such functions are called **CUDA kernels**
  - **Remark:** kernels have to return void
- **Device functions:**
  - A C function is declared to be **device code callable from device** by adding the **__device__** declaration specifier:

```
__device__ float Max(float x, float y);
```

  - **Remark**: device functions cannot be recursive for devices of compute capability < 2.0
- **Other functions:**
  - Functions **without these declarations** are compiled as **host code**

Creating GPU code (Fortran)

- **Kernels:**
  - A subroutine can be declared to be **device code callable from host** by adding the **`global`** attribute:

```
attributes(global) subroutine VectorSum(a,b,output,size)
```

  - Such functions are called **CUDA kernels**
  - **Remark:** kernels cannot have dummy arguments with modifiers **`intent(out)`** and **`value`**

- **Device subroutines / functions:**
  - A C function is declared to be **device code callable from device** by adding the **`__device__`** declaration specifier:

```
attributes(device) real function Max(x,y)
```

  - **Remark**: device subroutines/functions cannot be recursive for devices of compute capability < 2.0

- **Other functions:**
  - Functions **without these declarations** are compiled as **host code**

Calling CUDA kernels

- Kernels can be called from host code as any other function
  - But require execution configuration `<<< >>>` specification

- **Remarks:**
  - Kernel invocations are asynchronous,
    function call **returns** immediately, **before execution on device is finished**
  - `cudaDeviceSynchronize()` can be used to wait for running kernels

- **Example (C):**

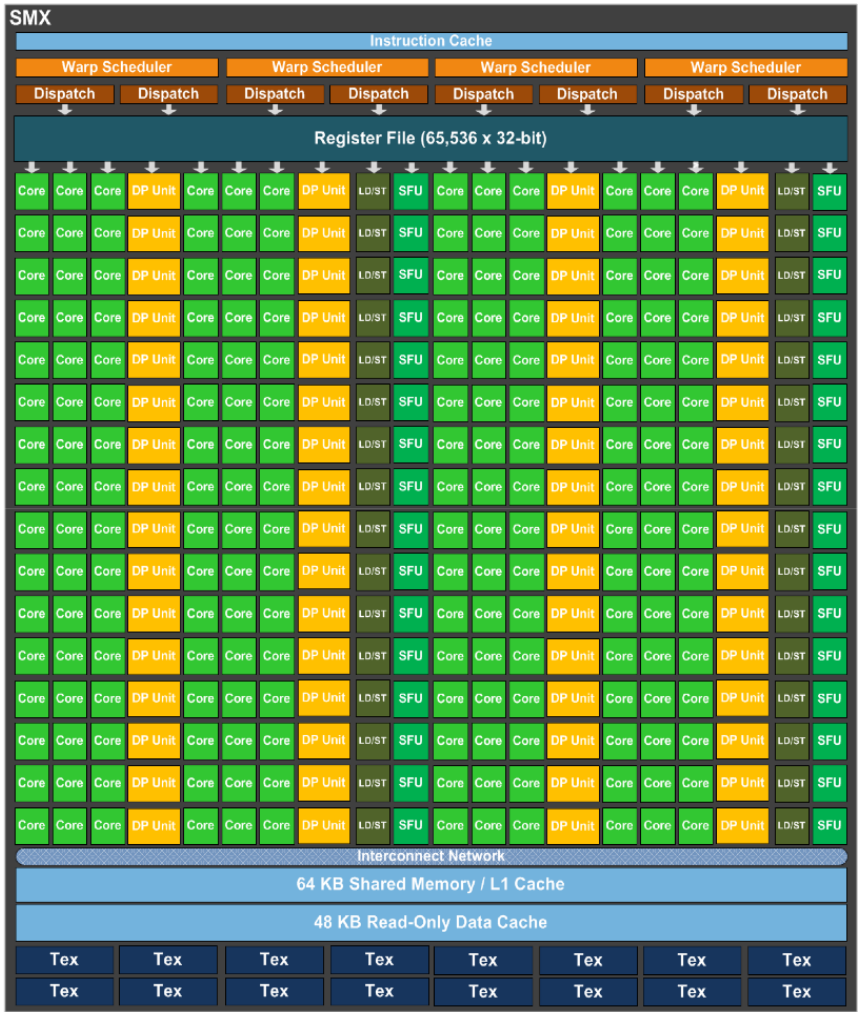
```
int main() {
    …
    VectorSum<<<gridSize, blockSize>>>(inputA, inputB,
                                       output, size);
    cudaDeviceSynchronize();
    …
    return 0;
}
```

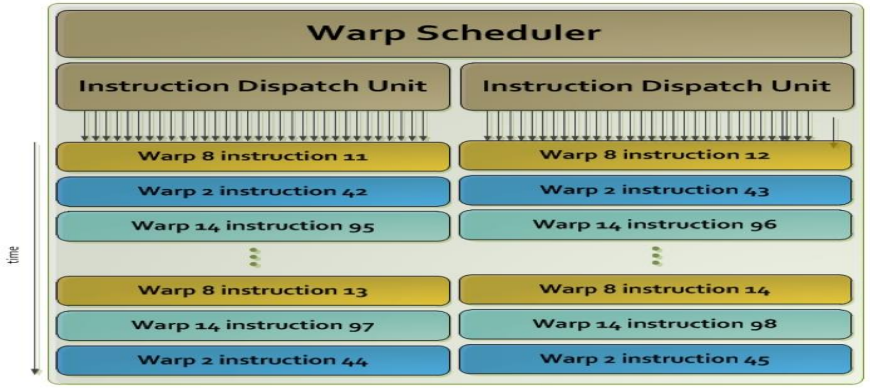
Calling CUDA kernels

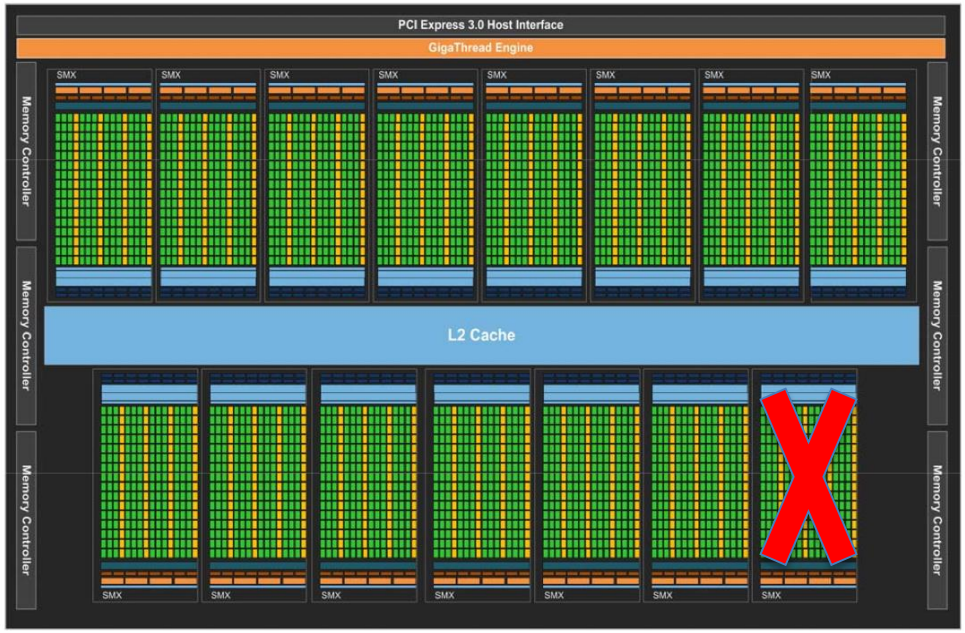
- **Example (Fortran):**

```
program example
     …
     call VectorSum<<<gridSize, blockSize>>>(inputA, inputB,
                                             output, size)
     error = cudaDeviceSynchronize()
     …
end program example
```

Hardware: Kepler Architecture (K20X)







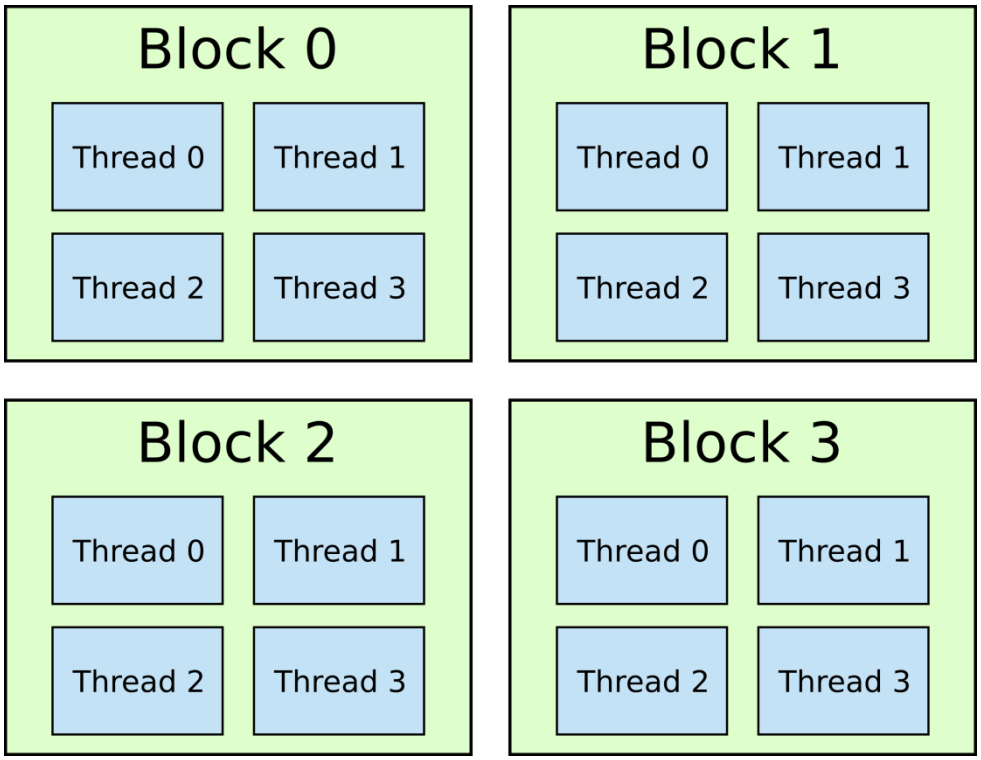
Kernel execution configuration

- Meaning of execution configuration:

```
<<<gridSize, blockSize>>>
```

  - **Block** of logical **thread**s
    - Are executed together (SIMT)
    - are distributed to different scalar processors within **the same** MP
  - **Grid** of **blocks**
    - Are executed independently in serial or parallel
    - are distributed to **different** multiprocessors

- On kernel invocation (‘function call’):
  - **All threads** in **all blocks** execute the **same function** in parallel



Threads & Blocks

- Advanced execution configuration syntax

```
dim3 blockSize(16,8,4); // total number of threads
                        // per block = 512
dim3 gridSize(64,32);   // total number of blocks = 2048

MyKernel<<<gridSize, blockSize>>>();
…
```

  - Threads and Blocks may be organized 1-, 2-, 3-dimensional
  - Unspecified components of **`dim3`** are initialized to 1 (overloaded C++ constructor)

- Get thread/block information from kernel code:

  - **`threadIdx.x`** (**`threadIdx.y`**, **`threadIdx.z`**) → index of this thread within block
  - **`blockDim.x`** (**`blockDim.y`**, **`blockDim.z`**) → number of threads per block
  - **`blockIdx.x`** (**`blockIdx.y`**, **`blockIdx.z`**) → index of block of this tread
  - **`gridDim.x`** (**`gridDim.y`**, **`gridDim.z`**) → number of blocks in grid

Threads & Blocks (Fortran)

- Advanced execution configuration syntax

```
type(dim3) :: blockSize = dim3(16,8,4) // total number of threads
                                       // per block = 512
type(dim3) :: gridSize = dim3(64,32,1) // total number of blocks = 2048

call MyKernel<<<gridSize, blockSize>>>()
…
```

  - Threads or blocks may be organized 1-, 2-, 3-dimensional
  - All components of `dim3` have to be specified (set unused components to 1)

- Get thread/block information from kernel code:

  - `threadIdx%x` (`threadIdx%y`, `threadIdx%z`) → index of this thread within block
  - `blockDim%x` (`blockDim%y`, `blockDim%z`) → number of threads per block
  - `blockIdx%x` (`blockIdx%y`, `blockIdx%z`) → index of block of this tread
  - `gridDim%x` (`gridDim%y`, `gridDim%z`) → number of blocks in grid

  - **Note:** all indices start counting **from 1 in Fortran** but **from 0 in C** (!)

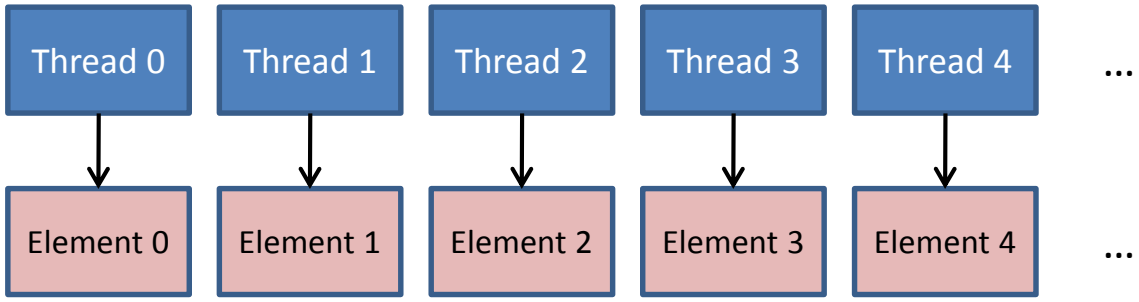
Limits

- The following limitations apply for the execution configuration
  - CC≥2.0: Maximum number of **threads per block** is 1024
    blockDim.x * blockDim.y * blockDim.z ≤ 1024
  - CC<2.0: Maximum number of **threads per block** is 512
  - Maximum number of threads in z-direction is 64
    blockDim.z≤64
  - Maximum number of **blocks per dimension** is
    - CC<3.0: 65535 in all directions
      gridDim.x≤65535, gridDim.y≤65535, gridDim.z≤65535
    - CC≥3.0: 2147483647 in x-direction, 65535 in y,z-directions
      gridDim.x≤2147483647, gridDim.y≤65535, gridDim.z≤65535
    - But overall number of blocks can be large

Example: add 2 vectors with one block

- If vector size = block size:

```
__global__ void VectorSum(float* inputA, float* inputB,
                          float* output) {
    output[threadIdx.x] = inputA[threadIdx.x] +
                          inputB[threadIdx.x];
}
```

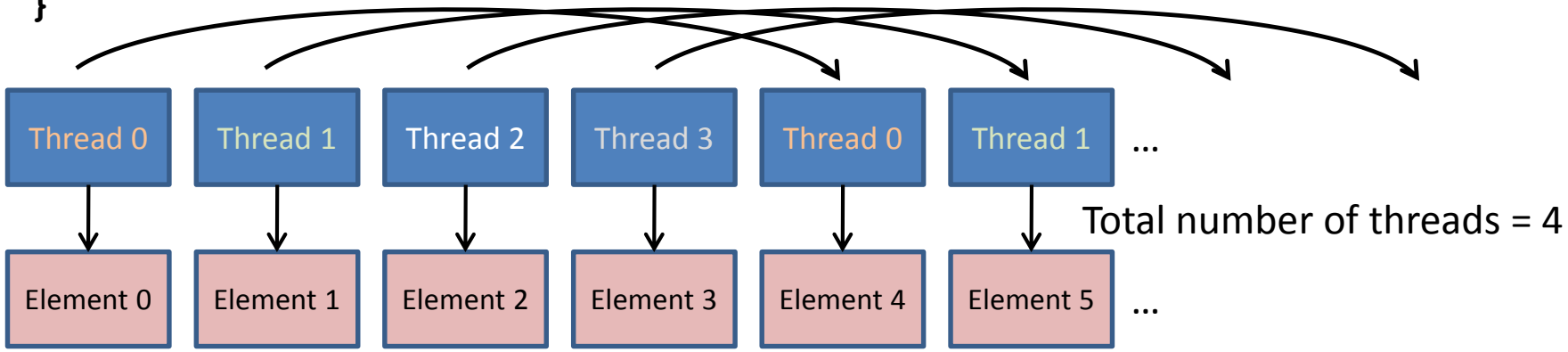


Each thread processes **exactly one array element *in total***

Example: add 2 vectors with one block

- **If vector size ≠ block size:**

```
__global__ void VectorSum(float* inputA, float* inputB,
                          float* output, int size) {
   for( int i=threadIdx.x; i<size; i+=blockDim.x ) {
     output[i] = inputA[i] + inputB[i];
   }
}
```

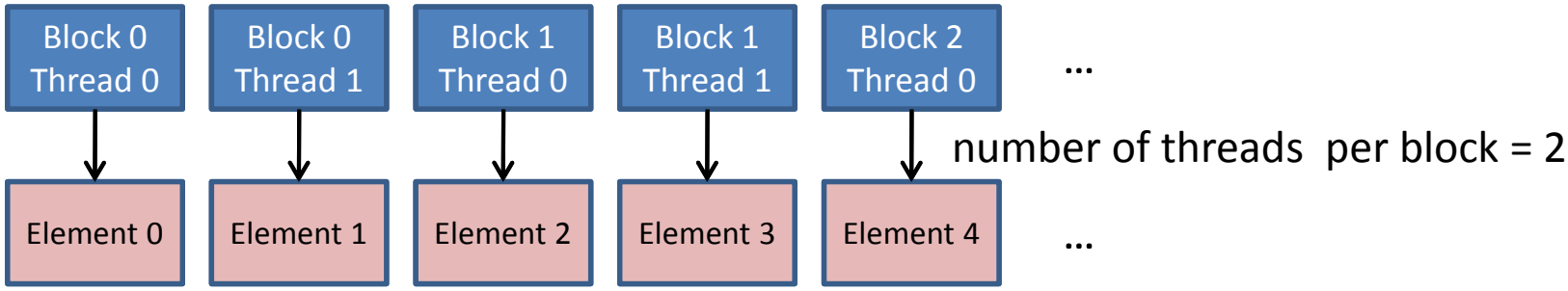


Each thread processes **one array element *per for loop iteration***
The elements are processed **in 'round robin' fashion**

Example: add 2 vectors with many blocks

- If vector size = block size * grid size:

```
__global__ void VectorSum(float* inputA, float* inputB,
                          float* output) {
    // local variables are local to thread
    int myIndex = blockIdx.x*blockDim.x + threadIdx.x;
    output[myIndex] = inputA[myIndex] + inputB[myIndex];
}
```

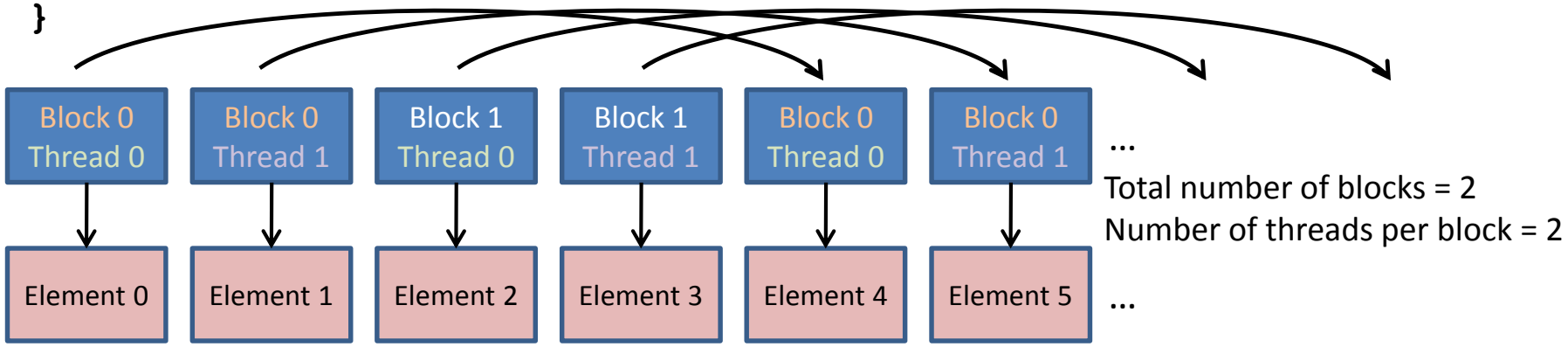


Each thread in each block processes **exactly one array element *in total***

Example: add 2 vectors with many blocks

- **If vector size ≠ block size * grid size:**

```
__global__ void VectorSum(float* inputA, float* inputB,
                          float* output, int size) {
   for( int i = blockIdx.x*blockDim.x + threadIdx.x;
        i<size; i+=blockDim.x*gridDim.x ) {
     output[i] = inputA[i] + inputB[i];
   }
}
```



Each thread in each block processes **one array element *per for loop iteration***
The elements are processed **in 'round robin' fashion**

Variable declarations and memory

- Variables declared outside of device functions reside on the host as usual
- Variables declared __device__ (Fortran: device) in global scope (outside of functions) reside in global device memory and are accessible from all threads of all blocks of all kernels
- Variables declared normally inside device functions reside in GPU registers or global device memory
  - these are local per thread
- Variables declared __constant__ (Fortran: constant) in global scope reside in constant memory and are readable from all threads of all blocks of all kernels
  - Constant memory is discussed later
- Variables declared __shared__ (Fortran: shared) inside device functions reside in shared memory and are local per **block**
  - Shared memory is discussed later

Allocate and copy device memory (C)

- **Allocate device memory:**
```
cudaError_t cudaMalloc( void** pointer, size_t size );
```

- **Free device memory:**
```
cudaError_t cudaFree( void* pointer );
```

- **Copy memory:**
```
cudaError_t cudaMemcpy( void* dst, const void* src, size_t count,
                        enum cudaMemcpyKind kind );
```

  kind may be
  - `cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice`
  - `cudaMemcpyDefault (with unified address space only)`

- **Allocate and free page-locked host memory:**
```
cudaError_t cudaMallocHost( void** pointer, size_t size );
cudaError_t cudaFreeHost( void* pointer );
```

- Why?
  - Faster copy from/to device:
    non-page-locked host memory will be buffered in PL memory during memcpy (!)
  - Access from device is possible
  - Asynchronous copy is possible

Allocate and copy device memory (Fortran)

- **Allocate device memory (static):**

```
real, dimension(0:size-1), device :: data
```

- **Allocate device memory (dynamic):**

```
real, allocatable, dimension(:), device :: data
allocate(data(0:size-1))
…
deallocate(data)
```

- **Copy memory:**
  simply use (array) assignment (works in all directions, except subarray copy from device to device):

```
data_device = data_host
data_host   = data_device
```

  - Note: device single array element accesses (`xxx=data_device(0)`), array initialization (`data_device=0`), and a few other operations work as well
  - Note: on subarray copy sometimes complete array is copied (compiler bug)

- **Allocate page-locked host memory:**

```
real, allocatable, dimension(:), pinned :: data
allocate(data(0:size-1))
 …
deallocate(data)
```

  - Note: this works only with allocatable arrays (!)

Pointers (C)

- Within host code:
  only pointers to host memory can be dereferenced.
- Within device code:
  - Since CUDA 4.0 unified address space:
    both host and device pointers can be dereferenced
  - Note: **only memory regions allocated with cudaMallocHost()**
    (or other CUDA functions) **can be accessed from the device!**
  - Conclusion: pointers **passed** to kernels **as arguments**
    must point to global memory (or unified address space)
- Note:
  - Pointers in device code may also point
    to shared memory or constant memory
- Fortran: compiler checks for most dereferencing errors
  - Note: unified address space not usable directly
    (compiler forbids passing `pinned` variables to kernel)

Passing arguments to kernels

- With CUDA 4.0 and CC 2.0 pointers and arrays (C) passed to kernels are assumed **to lie in the unified address space** of device memory and page-locked host memory
- Arrays (Fortran) passed to kernels are assumed to be stored in global device memory (declared **`device`**)
- Scalars like **`int`**, **`float`**, …, **`struct`** (C), **`integer`**, **`real`**, **`type`** (Fortran)
  - are passed by value to the kernel (C)
  - are passed by value to the kernel if declared **`value`**, or assumed to be passed **`device`** variables otherwise (Fortran)

- **Note:** no device memory allocation and copy is necessary to pass scalars to kernels by value
- **Note:** in Fortran kernel arguments may not be declared **`value`** and **`intent(out)`** or **`intent(inout)`** at the same time
- **Note:** passed by value kernel arguments are stored in shared memory (shared memory is discussed later)

Synchronization

- How can the execution of the threads and blocks be synchronized?
  - The device function (Fortran: subroutine)

    `__syncthreads()` (Fortran: `call syncthreads()`)

    synchronizes all threads within **the same block**
    - Note **all threads** of the block need to reach **the same** `__syncthreads()` call or a deadlock happens

  - The host function (Fortran: subroutine)

    `cudaDeviceSynchronize()` `(error=cudadevicesynchronize())`

    blocks the host execution until all GPU kernels and other operations are finished

  - Implicit barriers
    - Multiple kernel executions never overlap for devices of compute capability <2.0
      - Up to 32 (Fermi: 16) kernels can be executed simultaneously (depending on resource usage)
    - cudaMemcpy (Host → Device, Device → Host, Device → Device) does not overlap with kernel execution
      - But cudaMemcpy (Host → Host) does (!)
      - And cudaMemcpyAsync does also
    - cudaMemcpy blocks the host thread until the transfer is finished

Error handling

- The function

  ```
  cudaError_t cudaGetLastError();
  ```
  (Fortran: `integer function cudaGetLastError()`)

  returns the last error from a CUDA API call or kernel execution
- The function

  ```
  char* cudaGetErrorString(cudaError_t error);
  ```
  (Fortran: `character(len=*) function cudaGetErrorString(error)`)

  returns the error in printable form
- Most cuda functions return a **`cudaError_t`** directly
  - e.g. **`cudaMemcpy()`**, **`cudaDeviceSynchronize()`**
- If no error happened the returned value is **`cudaSuccess`**
- Note on errors during kernel execution:
  As kernel call is asynchronous, a cudaGetLastError() directly afterwards does not show error → next CUDA API call reports the error
  - Typically reported error: "**unspecified launch failure**", translation: kernel program **crashed**
    Common reason: **invalid access to memory** (array bounds violation, etc.)

Simplified error handling (C)

- The file "**cuda_utils.h**" provides a simpler interface to do error checking of cuda calls

- The macro **cudaVerify(x)** assumes expression x returns an **cudaError_t**.
  If the result is an error it prints the error string and aborts the program.
  - Appropriate for most CUDA library calls

- The macro **cudaVerifyKernel(x)** executes statement x and calls **cudaGetLastError()** afterwards.
  If the result is an error it prints the error string and aborts the program
  - Appropriate for kernel calls
  - Note: surround kernel calls with double '(' ')' parentheses. Otherwise the preprocessor gets confused
    cudaVerifyKernel((myKernel<<<…>>>(…)))

Example host program (C)

```
int main() {

  int vectorSize = 10000000;
  int memorySize = sizeof(float)*vectorSize;

  float *vectorHostA;
  float *vectorHostB;
  float* vectorHostResult;

  cudaMallocHost((void**)&vectorHostA,
                 memorySize);
  cudaMallocHost((void**)&vectorHostB,
                 memorySize);
  cudaMallocHost((void**)&vectorHostResult,
                 memorySize);

  ... initialize vectors A,B ...

  float* vectorDeviceA;
  float* vectorDeviceB;
  float* vectorDeviceResult;

  cudaMalloc((void**)&vectorDeviceA,
             memorySize);
  cudaMalloc((void**)&vectorDeviceB,
             memorySize);
  cudaMalloc((void**)&vectorDeviceResult,
             memorySize);
```

```
  cudaMemcpy(vectorDeviceA, vectorHostA,
             memorySize, cudaMemcpyHostToDevice);
  cudaMemcpy(vectorDeviceB, vectorHostB,
             memorySize, cudaMemcpyHostToDevice);

  VectorSum<<<gridSize, blockSize>>>
              (vectorDeviceA, vectorDeviceB,
               vectorDeviceResult, vectorSize);

  cudaMemcpy(vectorHostResult,
             vectorDeviceResult,
             memorySize, cudaMemcpyDeviceToHost);

  ... print result ...

  cudaFree(vectorDeviceA);
  cudaFree(vectorDeviceB);
  cudaFree(vectorDeviceResult);

  cudaFreeHost(vectorHostA);
  cudaFreeHost(vectorHostB);
  cudaFreeHost(vectorHostResult);

  return 0;
}
```

Example host program (Fortran)

```
program demo
    use device_code
    implicit none
    integer,parameter :: vectorSize = 10000000
    real,dimension(0:vectorSize-1) :: vectorHostA
    real,dimension(0:vectorSize-1) :: vectorHostB
    real,dimension(0:vectorSize-1) :: vectorHostResult
    real,dimension(0:vectorSize-1),device :: vectorDeviceA
    real,dimension(0:vectorSize-1),device :: vectorDeviceB
    real,dimension(0:vectorSize-1),device :: vectorDeviceResult

    ... initialize vectors A,B ...

    vectorDeviceA = vectorHostA
    vectorDeviceB = vectorHostB

    call VectorSum<<<gridSize, blockSize>>>(vectorDeviceA, vectorDeviceB, &
                                            vectorDeviceResult, vectorSize)

    vectorHostResult = vectorDeviceResult

    ... print result ...

end program demo
```

Example device program (C)

```
__global__ void VectorSum(float* inputA,
      float* inputB, float* output, int size) {
    for( int i = blockIdx.x*blockDim.x + threadIdx.x;
         i<size; i+=blockDim.x*gridDim.x ) {
          output[i] = inputA[i] + inputB[i];
    }
}
```

Example device program (Fortran)

```
module device_code
contains

attributes(global) subroutine &
                VectorSum(inputA, inputB, output, size)
  implicit none
  real, dimension(0:size-1) :: inputA,inputB,output
  integer, value :: size
  integer :: i
  do i=(blockIdx%x-1)*blockDim%x + threadIdx%x-1, &
        size, blockDim%x*gridDim%x
    output(i) = inputA(i) + inputB(i)
  end do
end subroutine VectorSum

end module device_code
```

What can be used inside kernels

- What can be used
  - All mathematical operators
  - Control flow constructs (if, for, while, case, goto)
  - Transcendent mathematical functions for single precision floating point (see Table D-1 in PG)
  - Calls to device functions/subroutines
  - Pointers (C)
  - Structs (C) and statically-sized arrays (C, Fortran)
  - Assumed-shape arrays as kernel arguments (Fortran)
  - CUDA-specific built-in functions
  - C++ templates
  - C++ function overloading

What can not be used inside kernels

- What can be used **only on newer devices**
  - Double precision floats and DP mathematical functions (compute capability ≥ 1.3)
  - Recursive function calls (C) (compute capability ≥ 2.0)
  - Function pointers (compute capability ≥ 2.0, CUDA ≥ 3.1)
  - C++ non-polymorphic classes (no virtual functions) (compute c. ≥ 2.0)
  - printf (compute capability ≥ 2.0, CUDA ≥ 3.1) (Fortran: print available, but buggy)
  - malloc, new (compute capability ≥ 2.0, CUDA ≥ 3.2) (no Fortran equivalent, yet)
  - C++ polymorphic classes (compute capability ≥ 2.0, CUDA ≥ 4.0)

- What can **not be used** at the moment **inside device code**
  - C99/C++ dynamically-sized '[]'-arrays
  - System calls, I/O, memory management
    - fopen, fprintf, system, ...
  - Long double floats
  - Allocatable arrays, pointers, value with intent(out) or intent(inout) (Fortran)
  - Recursive, pure, elemental function calls, optional function arguments (Fortran)
  - Save attribute (Fortran), static variables declared in device functions

GPU Programming using CUDA
Exercises 1+2: Vector-scalar-multiplication

Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

Exercise 1: serial function evaluations on GPU (C)

- Vector-scalar multiplication
  - CPU code (exercise01_cpu.c):

```
const int size = 10000;
float a[size];
const float b;
float c[size];
…
for(int i=0;i<size;i++) {
  c[i]=a[i]*b;
}
```

  - The computation should be done on the GPU but use only one thread and only one block
- Sketch:
  - Allocate an array on the device with **`cudaMalloc()`**, Allocate an array on the host with **`cudaMallocHost()`**
  - Use **`cudaMemcpy`** to copy the input vector to the GPU
  - Call kernel with gridSize = 1 and blockSize = 1
    - The kernel takes the address of the device memory as argument
  - Use **`cudaMemcpy`** to copy the results back to the host

Exercise 1: serial function evaluations on GPU (Fortran)

- Vector-scalar multiplication
  - CPU code (exercise01_cpu.f90):

```
subroutine eval(a,b,c,size)
  implicit none
  real, dimension(0:size-1) :: a,c
  real, value :: b
  integer, value :: size,I
  do i = 0, size-1
    c(i)=a(i)*b;
  end do
end subroutine eval
```

  - The computation should be done on the GPU but use only one thread and only one block
- Sketch:
  - Allocate an array on the device using the device modifier, Use an array assignment to copy the input vector to the GPU
  - Call kernel with gridSize = 1 and blockSize = 1
    - The kernel takes the array on the device memory as argument
  - Use an array assignment to copy the results back to the host

Exercise 2: parallel function evaluations on GPU

- Modify your solution of exercise 1
  to **use many threads in many blocks** which do the work in parallel
- Hint:
  - Use **`threadIdx.x`**, **`blockDim.x`**,
    `blockIdx.x` and **`gridDim.x`** to decide for each thread
    which array elements it has to process

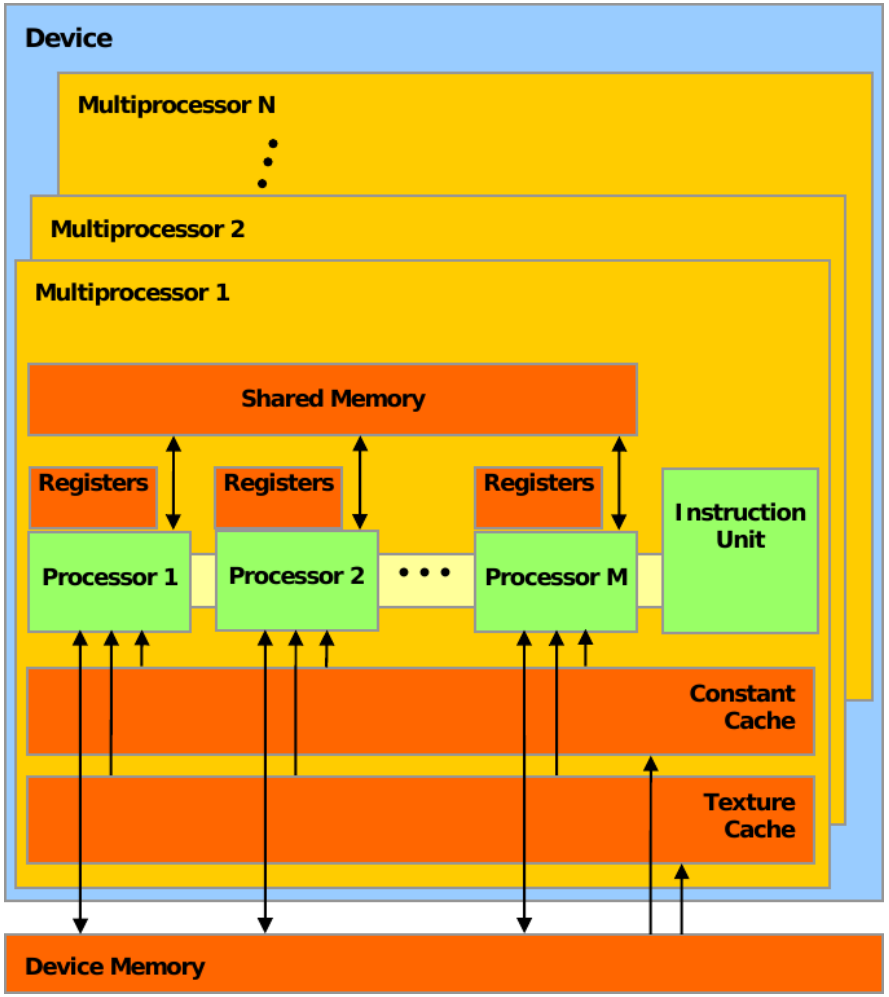
- Advanced exercise:
  - Can you modify your code in way
    that it does **not use a for loop** inside the kernel
    but uses a **large number of blocks**
    (depending on the vector size) instead?

GPU Programming using CUDA
Performance Optimization I

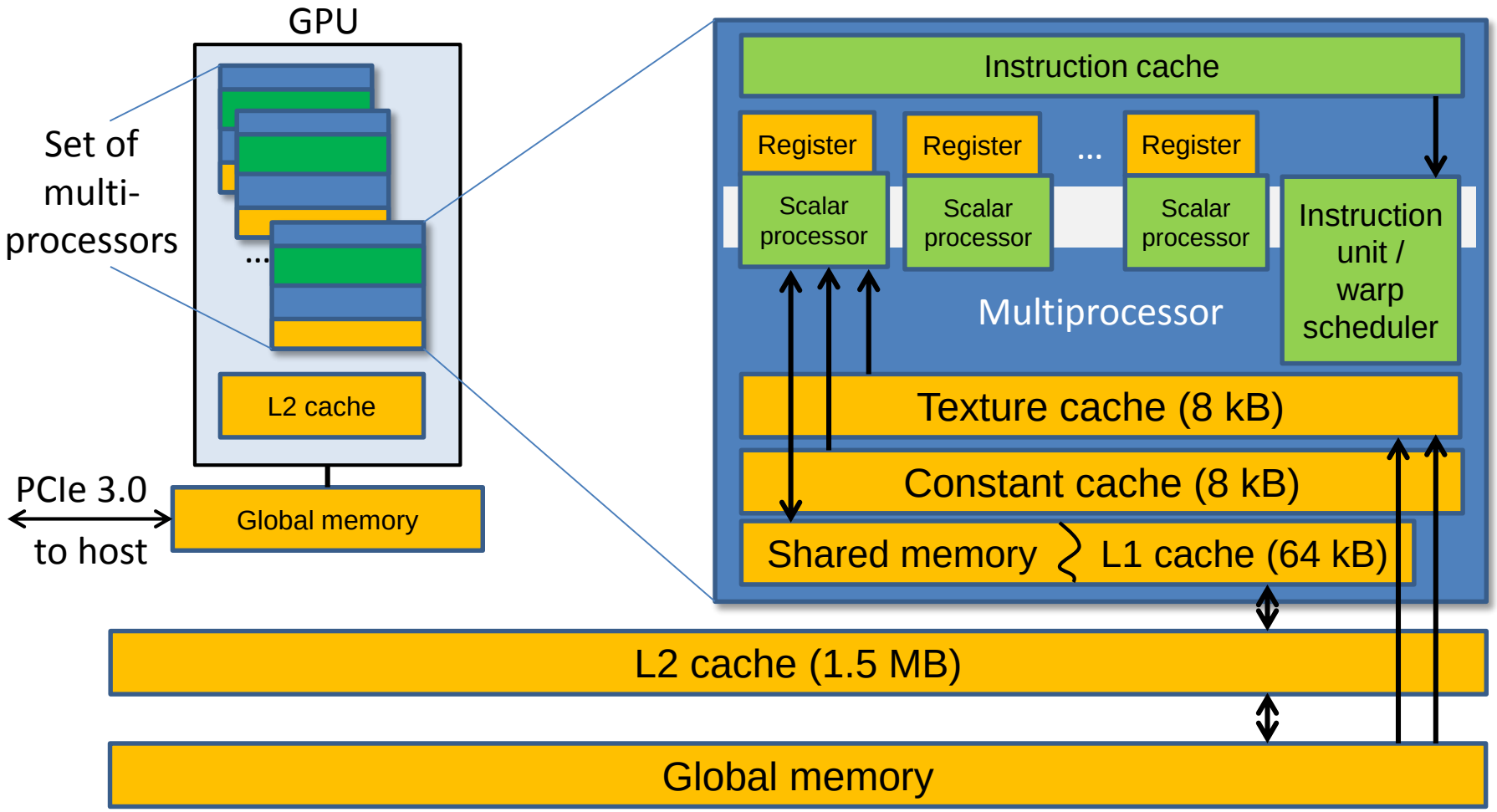
Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

GPU architecture

- Two-Stage parallelism
  - A GPU contains multiple **Multiprocessors (SMX)** (Kepler: 13 -15)
  - Each *Multiprocessor* (SMX) contains multiple **scalar processors** (Cores) (Kepler: 192)
- Programming of GPUs:
  - Work has to be distributed
    - To different multiprocessors
    - To different scalar processors within one multiprocessor



Hardware: Kepler architecture



Warps

- SIMT execution
  - Single instruction multiple thread
  - Each group of **32 threads** within the same block **execute the same instruction at the same time**
  - Those groups of threads are called **warps**
  - **Note:** some operations are per **half-warp**, this is either the first or second 16 threads of a warp
  - How are warps mapped to **multidimensional blocks**?
    - 3D-thread index is mapped to linear index, where threadIdx.x is fastest index
    - Example: blockDim=dim3(16,6,1)
      - 96 threads will result in 3 warps
        - warp 0: 0≤threadIdx.x ≤15, 0≤threadIdx.y ≤1
        - warp 1: 0≤threadIdx.x ≤15, 2≤threadIdx.y ≤3
        - warp 2: 0≤threadIdx.x ≤15, 4≤threadIdx.y ≤5

SIMT execution

- What happens when control flows diverge?

```
if ( threadIdx.x > 0 ) {
    output[threadIdx.x] = input[threadIdx.x] +
                          input[threadIdx.x-1];
} else {
    output[threadIdx.x] = input[threadIdx.x] + 1;
}
```

  - Answer: it works, but branches are executed in serial
  - It is handled efficiently: only branches which are taken by **at least one thread** are executed

Notes on SIMT execution

- How many threads per block should one use?
  - As threads are organized in groups of 32 threads (warps) which execute together
    - A block of **3 threads** needs **the same execution time as a block of 32 threads** (when working on GPU registers)
    - Ideally the number of threads is **a multiple of 32**
  - Maximum number of threads per block is 1024
  - Number of registers per multiprocessor is limited (Kepler: 65536)
    - Registers per thread * threads per block ≤ registers per MP

Memory model

- There are several different kinds of memory
  - Host memory: can only be accessed via PCIe bus (slow)
  - **Global memory**: main memory of the video board (Tesla K20x: 6 GiB), can be accessed from all multiprocessors
    - **High bandwidth** (> 100 GB/s)
    - **High latency** (several 100 cycles)
    - CC ≥ 2.0 : (16 or 48 kiB) L1 cache/shared memory per MP
    - CC ≥ 2.0 : shared L2 cache
  - **GPU registers**: used for data local to one thread
    - Can be accessed within 1 cycle
  - **Local memory**: physically resides global memory but is used for data local to one thread
  - **Shared**, **constant**, **texture memory**: discussed later

- Notes:
  - **Local variables of a kernel function** reside in **GPU registers** (if not declared otherwise)
    - Are **local to each thread**
    - **If no more registers are available** local variables are stored in **local memory**
  - **Kernel function arguments** reside in shared memory on devices with CC<2.0 and in constant memory otherwise(→ accessing kernel arguments is very fast)

Kernel dummy parameters (Fortran)

- In C kernel **scalar arguments** are passed **by value**, and pointers point to **global memory**
- In Fortran scalar arguments are by default stored **in global memory**
  - To reproduce the C behavior of passing scalar arguments by value in shared memory the 'value' modifier must be used:

```
attribute(global) subroutine my_kernel(my_param)
  real, value :: my_param
  …
```

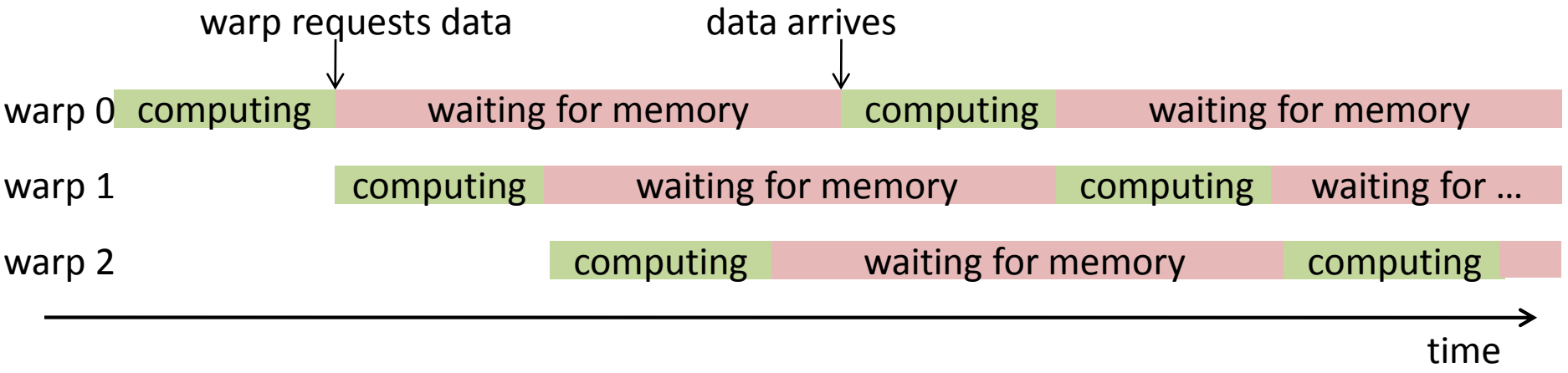
- **Note:** 'value' parameters are stored in shared memory (discussed later) which is faster than global memory
- **Array descriptors** are usually passed **in global memory**. This means using operators like **`size()`**, **`shape()`**, …, results in global memory accesses (slow!)
  - use sparingly or pass size information by value separately
  - **Note:** storing of array descriptors to global memory (extra memcpy) can be avoided by specifying sizes for dummy parameters (**`dimension(0:size-1)`** instead of **`dimension(:)`**)

Notes on block size and grid size

- How many blocks per grid should one use?
  - To use all multiprocessors,
    number of blocks ≥ number of multiprocessors
- As accesses to global memory have long latency,
  **to get full memory bandwidth, *latency hiding*
  is necessary(!)**
  - If the number of blocks is larger than the number of multiprocessors, **block execution will be overlapped** (multitasking)
- Equivalently for threads:
  if the number of threads is larger than the warp size (32)
  warp execution will be overlapped/multitasked

Latency hiding

- How does latency hiding work?
- Each multiprocessor **can switch/'multitask' between several warps** – even of different blocks
- If a thread/warp requests global memory it has to wait a very quite long (500-600 cycles!)
- During this time the multiprocessor executes code of a different warp



Notes on latency hiding / multitasking

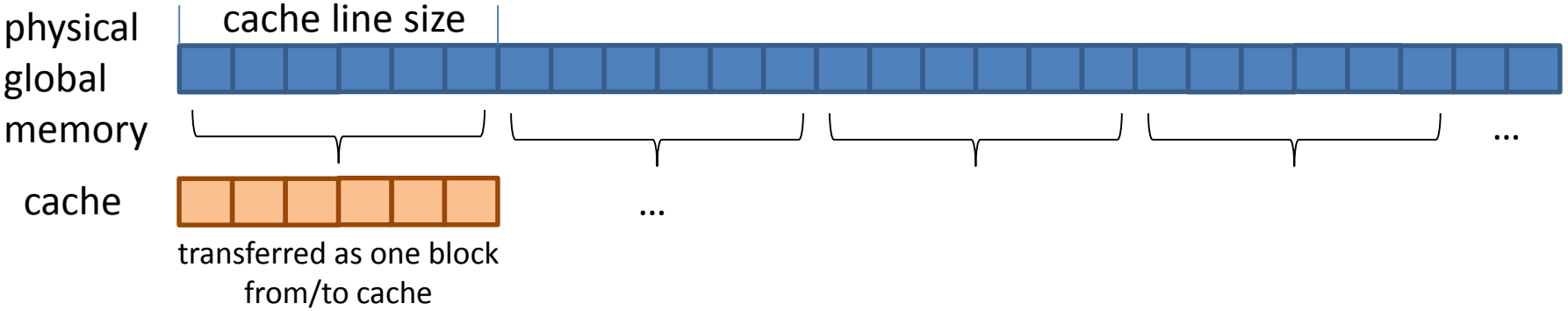
- The maximum number of **warps** than can be operated on at the same time **by one multiprocessor** is
  - 48 for Fermi architecture
  - 64 for Kepler architecture
- The maximum number of **blocks** than can be operated on at the same time **by one multiprocessor** is
  - 8 for Fermi architecture
  - 16 for Kepler architecture
- Conclusion: block sizes larger than 32 may be advantageous for latency hiding

Register usage

- Important note: warp switching/multitasking will only happen as far **as the available number registers** and the **vailable shared memory** in the multiprocessor **is enough for multiple warps/blocks**
  - Each multiprocessor has only a limited number of registers (!)
    - Kepler: 65536, Fermi: 32768, register size is 32 bits
- How do I find out **how many registers** per thread my kernel uses?
  - CUDA profiler or
  - `--ptxas-options -v or -Xptxas -v`
  - `See the difference if you compile with -arch=sm_(35, 20, 13, 10)`
- Can I limit the number of registers to use?
  - Yes: using `__launch_bounds__(maxThreadsPerBlock,minBlocksPerMP)` on kernel definition
  - or using '`--maxrregcount <N>`' on compilation
  - `Lower bound is 16 reg/thread`

Global memory accesses and caching

- Cache lines
  - Data is transferred to/from L1/L2 cache in blocks of 128 bytes
  - 128 consecutive bytes in global memory belong to the same cache line



- **Note:** caches are small:
  with 1024 concurrent threads per multiprocessor, around one third cache line/thread (!)
  - If each thread in the device accesses a different cache line in the same cycle, caching is useless!

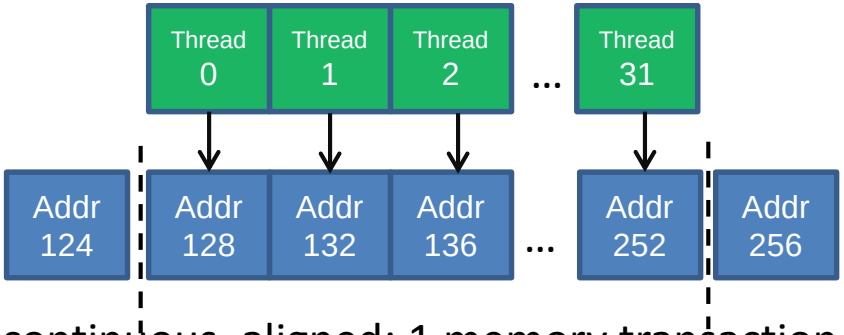
Coalesced accesses

- Coalesced accesses:
  - Combining global memory access improves performance
  - If multiple threads of a **warp** access the same cache line (128 bytes) the accesses are combined into one transfer
- **Note:** number of transferred cache lines per cycle is limited
  → it is slower to access multiple different cache lines
  at the same time from the same warp (!)
- **Note:** Kepler always transfers 128 byte blocks from/to memory/cache, regardless how much data of the block is needed

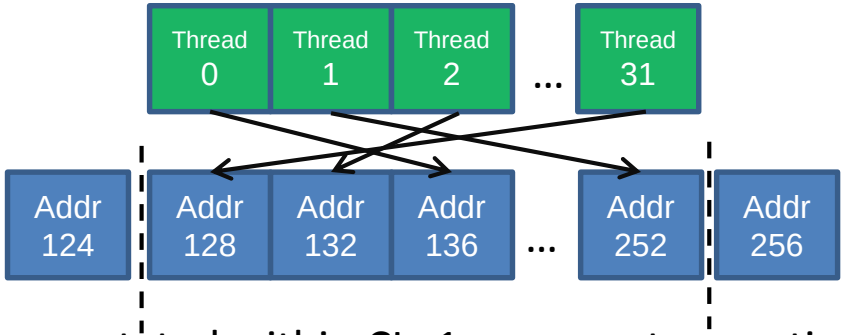
- Older hardware (T10):
  - pre-Fermi cards do not have L1/L2 cache
    → coalescing more important, as unneeded data is always wasted
  - T10 transfers 32-byte, 64-byte or 128-byte aligned blocks,
    as best suited (can be advantage over Fermi)
  - memory accesses are per half-warp
    (not per warp), so only first/second 16 threads combine transfers

Coalesced vs. uncoalesced access patterns

- Coalesced accesses:

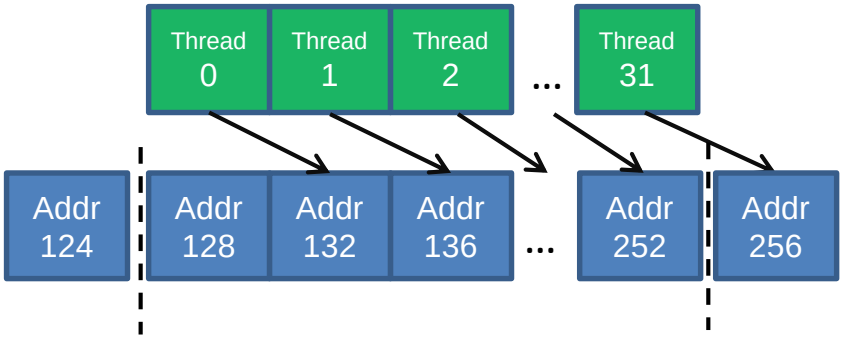


continuous, aligned: 1 memory transaction

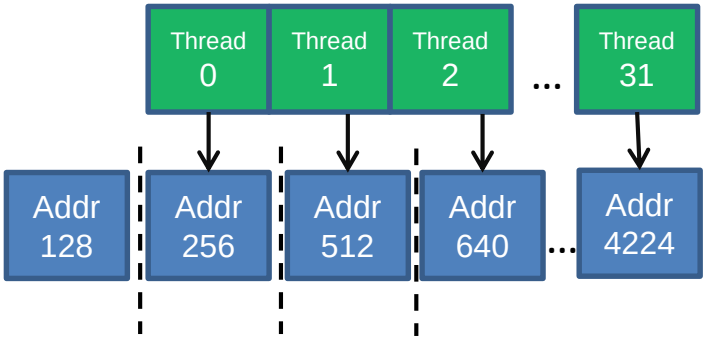


permutated within CL: 1 memory transaction

- Uncoalesced accesses:



continuous, unaligned: 2 memory transactions



strided (128 bytes): 32 memory transactions

Global memory writes on Fermi and Kepler

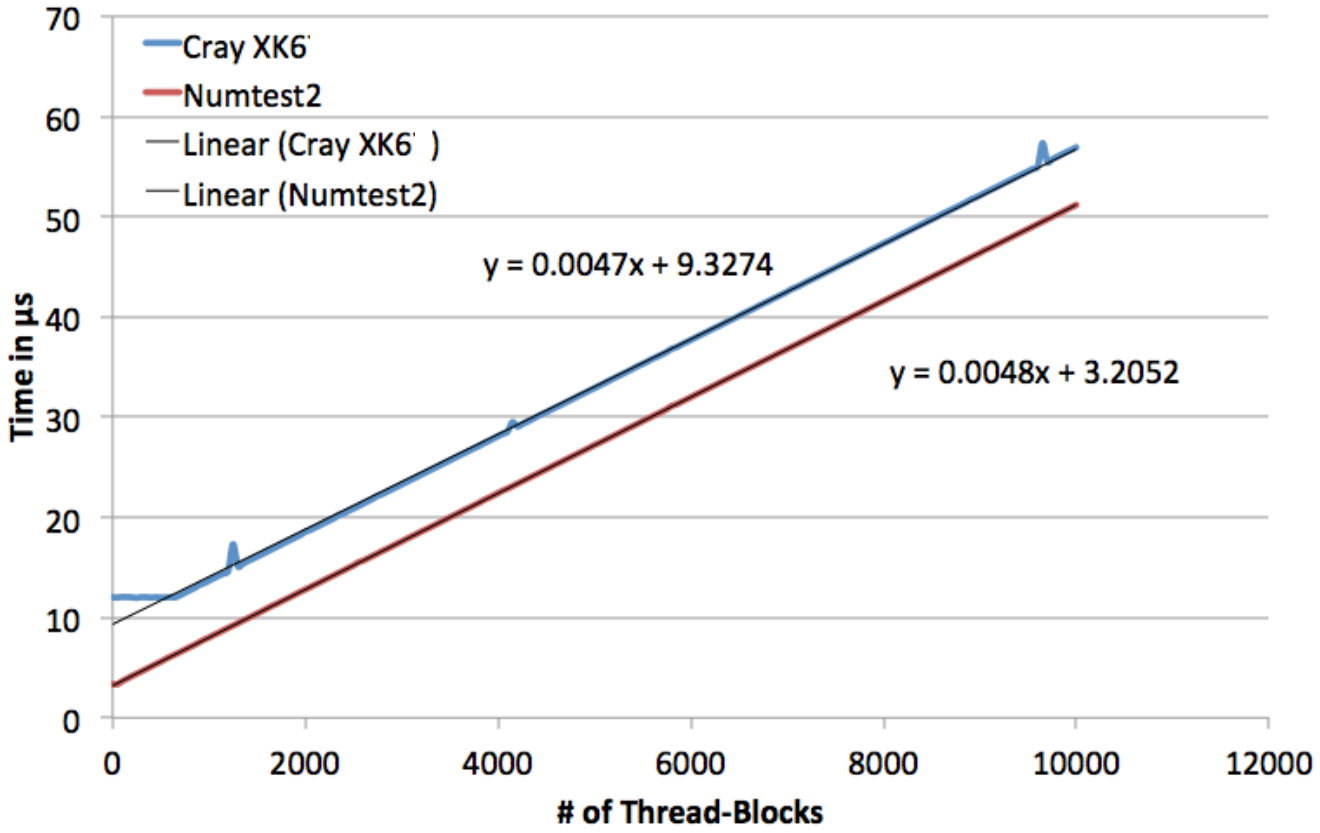
- **Note** on Fermi/Kepler and coalesced **write** accesses:
  - Cache architecture requires that after each write a cache line needs to be in a consistent state
  - When doing a **partial write** on a cache line in case its data contents were not present in cache before, the data has to be read from memory before the write can be done!
  - This is not the case if the cache line is written completely with one coalesced access
  - Effect: it is usually faster to write all data elements of a cache line with a coalesced access, instead of
    - writing only some elements
    - writing the elements sequentially
  - Example:

```
index = threadIdx.x+blockIdx.x*blockDim.x;
if ((threadIdx.x&31)!=0) {
  output[index] = inputA[index] + inputB[index];
}
```

    is slower than the version without the if clause!

CUDA kernel startup time

- How long does it take to start a kernel from the host?
  - Answer: about 3.2µs + 4.8ns*(grid size)
    (our measurement on our Tesla K20X, Fermi is at about 2.5-3µs)



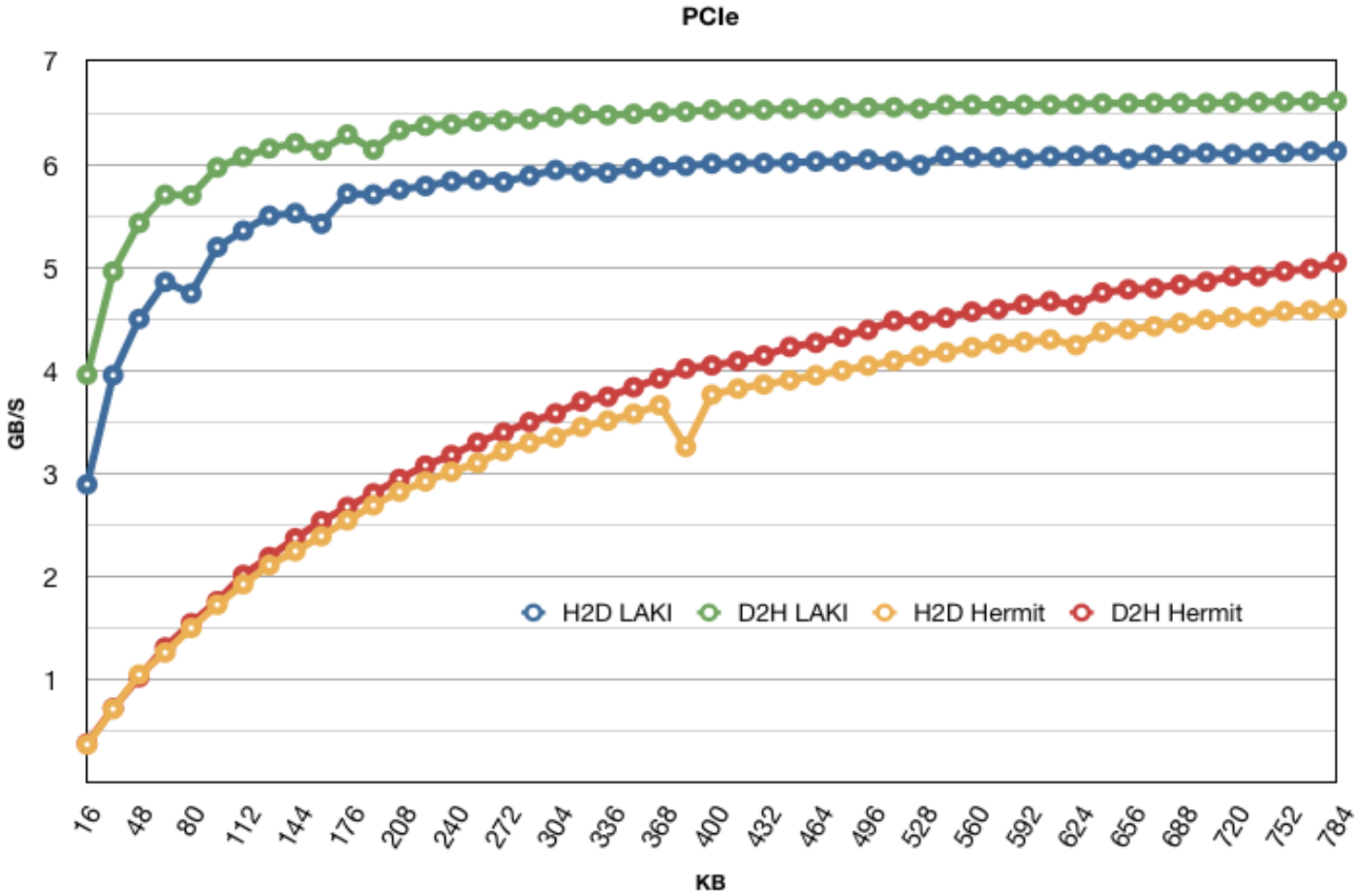
Notes on kernel startup time

- Conclusion:
  Overhead per block is only a few cycles
  → It is okay to use a large number of blocks instead of programming loops
  - This should be the preferred solution to allow for more latency hiding

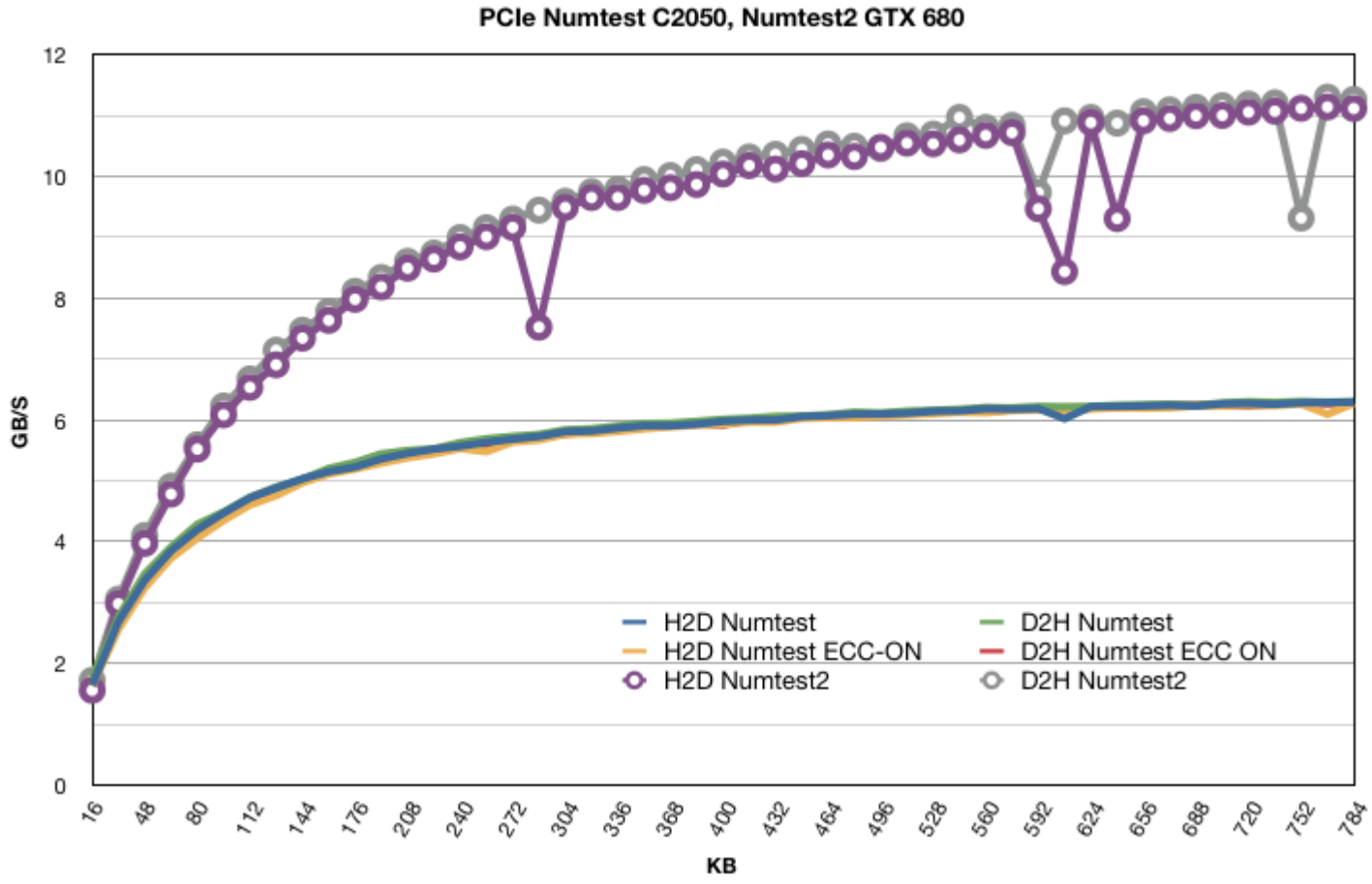
Memory transfers host ↔ device

- How long does it take to transfer memory from/to the device?
  - Answer: about 3.3 µs+(data size)/(6GB/s) for page-locked host memory (our measurement on our Tesla C1060s)
- Conclusion: keep data as long as possible on the device(!)

PCIe memory transfer speed



PCIe memory transfer speed



Is GPU always faster?

- First estimation:
  - The Time, the code needs to execute on CPU: t_{cpu}
  - The amount of data which needs to be copied to/from GPU: d
  - The Time needed for data copy: t_{copy}= d / (PCIe speed)
  - If $t_{copy} > t_{cpu}$ CPU execution will be faster(!)

Note on memory transfers host ↔ device

- Advanced feature: overlapping data transfer with computation
  - The function

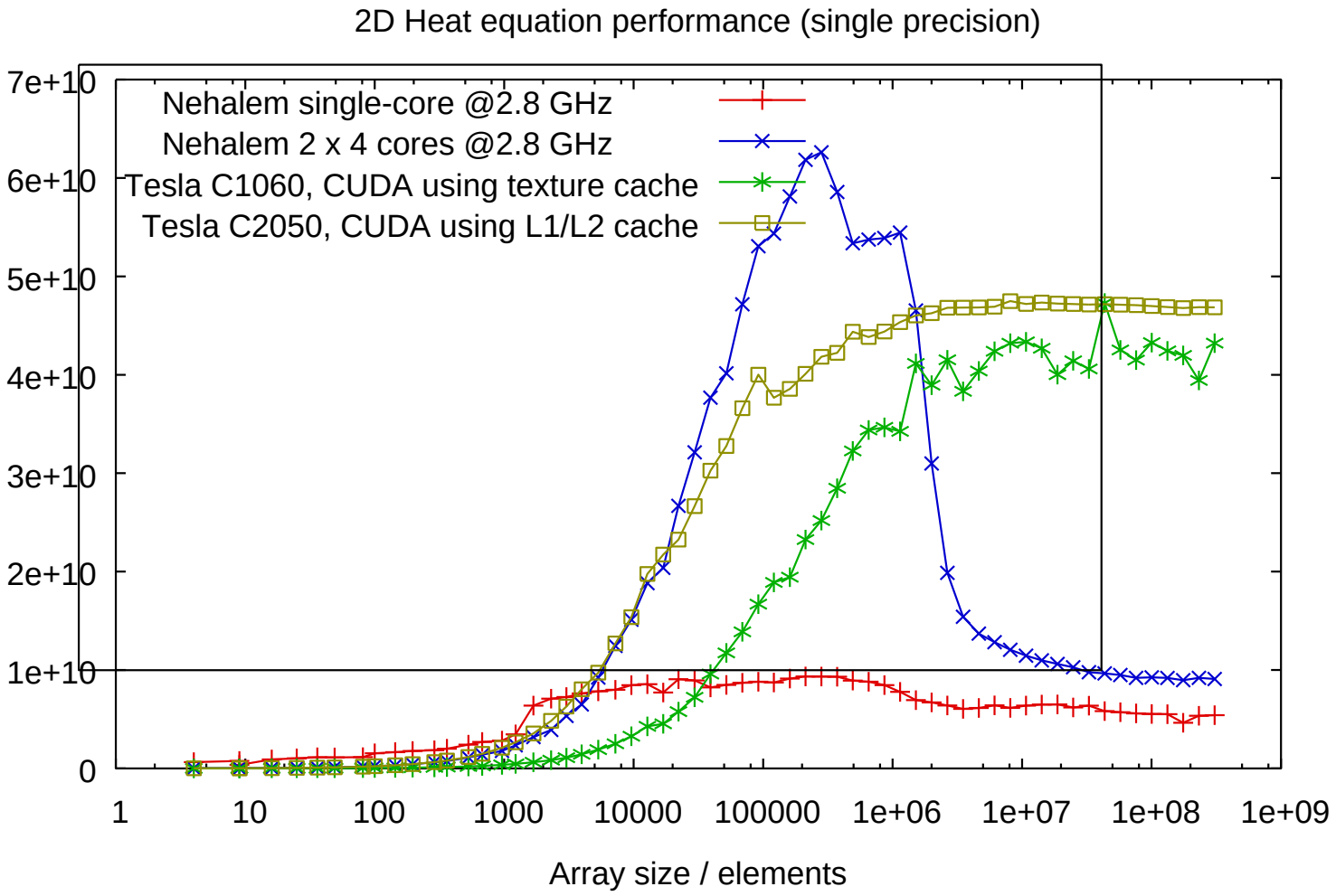
```
cudaError_t cudaMemcpyAsync( void* dst, const void* src,
                             size_t count, enum cudaMemcpyKind kind,
                             cudaStream_t stream );
```

    can be used to overlap transfers with CPU and/or GPU computations
  - Note: the usage of streams is required for this: Streams are discussed later

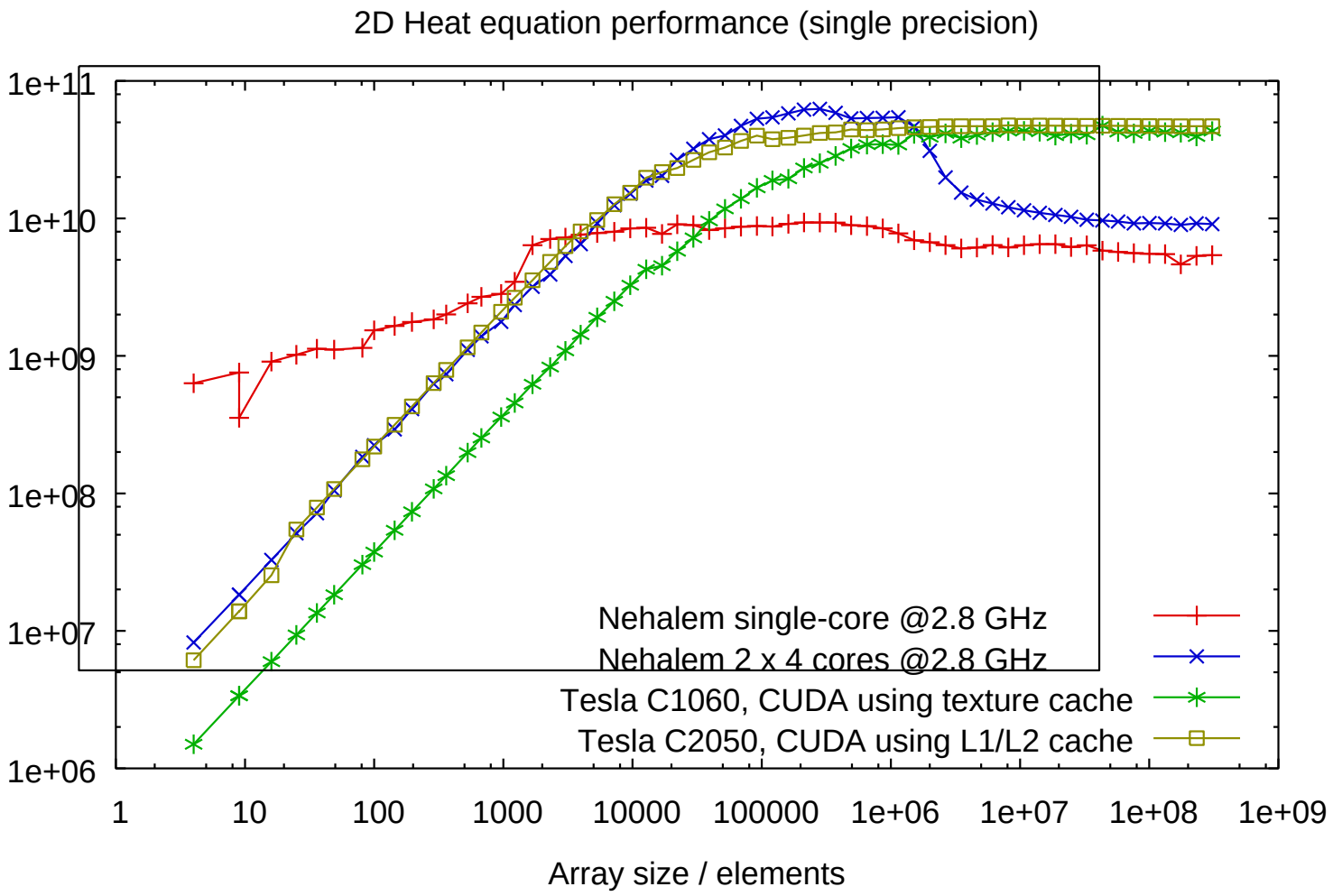
Notes on GPU Computation

- Problem size:
  - high memory latency and device parallelism
    → CPU faster for small problem size
- Non-parallelizable code parts
  - computation on CPU → PCIe data transfer costly
  - computation on GPU → single thread performance might be 100 times slower than CPU
    - stream addition: Nehalem: 9.2 GB/s, Fermi C2050: 72 MB/s
    - logistic map iteration:
      Nehalem: 740 MFLOPS, Fermi C2050: 43 MFLOPS

Performance: FDTD 2D heat equation



Performance: FDTD 2D heat equation



Double precision floating point numbers

- Double additions and multiplications **are slower by a factor of 8** on consumer Fermi cards (GTX 4xx/5xx), a **factor of 24** on consumer Kepler cards (GTX 6xx), a **factor of 2** on HPC Fermi cards (Tesla 20xx) and a **factor of 3** on HPC Kepler cards (Tesla K20x)
- Note: in many cases a factor 8 does not matter, when speed is limited by memory bandwidth (but a factor 24 usually does!)
- Note: to use doubles a device of **compute capability ≥ 1.3** is required. This means **'-arch=compute_13'**, **'-arch=compute_20' or** , **'-arch=compute_3?'** on compilation is required
- Note**: be careful with floating point constants**: in C/C++ constants without suffix '1.45' are doubles. **Use the 'f' suffix** if you wish your computation to be done in single precision: '1.45f'

Instruction throughput

- A multiprocessor can execute the following number of the given type of operation per cycle (Tesla K20x)
- Single precision float operations
  - add or multiply or multiply-add (192)
  - `1/x` (32)
  - `1/sqrtf(x)` (32)
  - `__sinf(x)`, `__cosf(x)`, `__log2f(x)`, `exp2f(x)` (32)
  - `sinf(x)`, `cosf(x)` (depends on argument, but slow)
    - Note: transcendental function throughput is still high compared to CPU
      C2050: 14G sinf operations/s, 64G __sinf operations/s
      Nehalem @2.8 GHz, 2x4 cores: 880M sinf operations/s
  - `__fdividef(x,y)` (faster than x/y)
  - atomic operation, L2, cache, no conflict (24)
- Double precision float operations
  - add or multiply or multiply-add (64)
  - `sin(x)`, `cos(x)`, `exp(x)` (faster than half speed of SP versions)
- 32-bit integer operations
  - add, `&`, `|`, `compare` (160)
  - shift (64)
  - multiply, multiply-add (32)
- `__syncthreads()` (128)

Pipelining and data dependencies

- **GPU can start executing a new instruction** of the same thread **before the last one is finished**
- **This can only work if there are no data dependencies(!)**
- Compare the following two code fragments

1)
```
float value1;
…
for(int j=0;j<iterations;j++) {
  value1=4.f*value1*(1.f-value1);
  value1=4.f*value1*(1.f-value1); // requires previous result
}
```

2)
```
float value1;
float value2;
…
for(int j=0;j<iterations;j++) {
  value1=4.f*value1*(1.f-value1);
  value2=4.f*value2*(1.f-value2); // no data dependency
}
```

- One finds that the second one runs considerably faster. Why?
  Because the 2 statements in the loop can be executed **independently** of each other.
  The processor **does not have to wait** for the results of the first statement

Summary on performance

- Use all scalar processors and all multiprocessors busy
- Avoid divergences in control flow
  of threads within the same warp
- Use many blocks but few registers and little shared memory
  for latency hiding
- Use coalesced accesses to global memory
  (or at least avoid accessing too many cache lines
  at the same time)
- Keep data on device as long as possible
- Use single precision floats
- Use fast __ mathematical operations when possible

GPU Programming using CUDA
Exercises 3-5: Coalesced accesses

Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

Exercise 3: Vector addition

- Compile the vector addition example below and measure the performance for different block/grid sizes
- Which is the optimal execution configuration?
- For which range of block/grid sizes do you get good performance and memory BW?
  - **Note:** you should see at least 70 GB/s on T10, 80 GB/s on Fermi and ~150 GB/s on K20x
- Kernel code (exercise03_template.cu):

```
__global__ void VectorSum(float* inputA,
                          float* inputB, float* output) {
    for( int i = blockIdx.x*blockDim.x + threadIdx.x;
i<size; i+=blockDim.x*gridDim.x ) {
          output[i] = inputA[i] + inputB[i];
    }
}
```

Timers (C)

- How to measure performance?
- The file "**cuda_utils.h**" provides a simple interface to the POSIX high resolution timers
- Usage:

```
Timer timer;
initTimer(&timer);
… code to measure …
double duration=getTimer(&timer);
```

- Add the linker option **-lrt** to the **nvcc** command line when compiling your program to link the POSIX real time library
- Performance (FLOPS) is (vector size)/duration for the vector addition kernel, as one float operation per vector component is needed

- Note: Remember that kernel invocations are asynchronous (!). A **cudaDeviceSynchronize**() call is needed before **getTimer()**

Timers (Fortran)

- How to measure performance?
- Fortran provides the built-in subroutine

```
call cpu_time(time)
```

  which returns a real-value timestamp in seconds

- Performance (FLOPS) is (vector size)/duration for the vector addition kernel, as one float operation per vector component is needed

- Note: Remember that kernel invocations are asynchronous (!). A **`cudaDeviceSynchronize`**() call is needed before **`getTimer()`**

Exercise 4: Uncoalesced access patterns

- Modify the vector addition kernel from exercise 3 to use different access patterns to the vector data
  a) Add an offset to the input vectors, so it is no longer aligned the way `cudaMalloc()` returns memory regions

C:

```
cudaMalloc((void**)&vector_x_alloc,
           (vector_size+offset)*sizeof(float));
vector_x_gpu=vector_x_alloc+offset;
…
free(vector_x_alloc)
```

Fortran:

```
real, dimension(0:vector_size-1+offset), device :: vector_x_gpu
…
call VectorSum<<<…>>>(vector_x_gpu &
                      (offset:vector_size+offset-1),…)
```

Which effect do you expect on performance?
On Kepler, do you see for offsets on **`vector_c`** a behavior different from that for **`vector_a`** and **`vector_b`**?

Exercise 4: Uncoalesced access patterns

- Modify the vector addition kernel from exercise 3 to use different access patterns to the vector data
  b) Rewrite the kernel so that each thread operates on **one continuous region** of vector indices (**stride is 1** instead of `blockDim.x*gridDim.x`):

  C:
  ```
  for(int i=my_start; i<my_end;i++)
  ```

  Fortran:
  ```
  do i=my_start,my_end-1,1
  ```

  Do you expect higher or lower performance from this?
  How large do you think the effect is?

Exercise 5 (optional): Diffusion equation in 1D

- Each field point requires the values of its neighbors.

$$f_i^{t+1} = f_i^t + \alpha (f_{i-1}^t + f_{i+1}^t - 2 f_i^t)$$

- CPU implementation:

```
void diffusion(float* input, float* output,
                                int size) {
  for( int i=1; i<size-1; i++ ) {
    output[i] = input[i]*(1.f-2.f*alpha) +
                alpha*(input[i-1]+input[i+1]);
  }
}
```

- Note: each input value is read 3 times,
  but caching in L1 is expected

Exercise 5: Diffusion equation in 1D

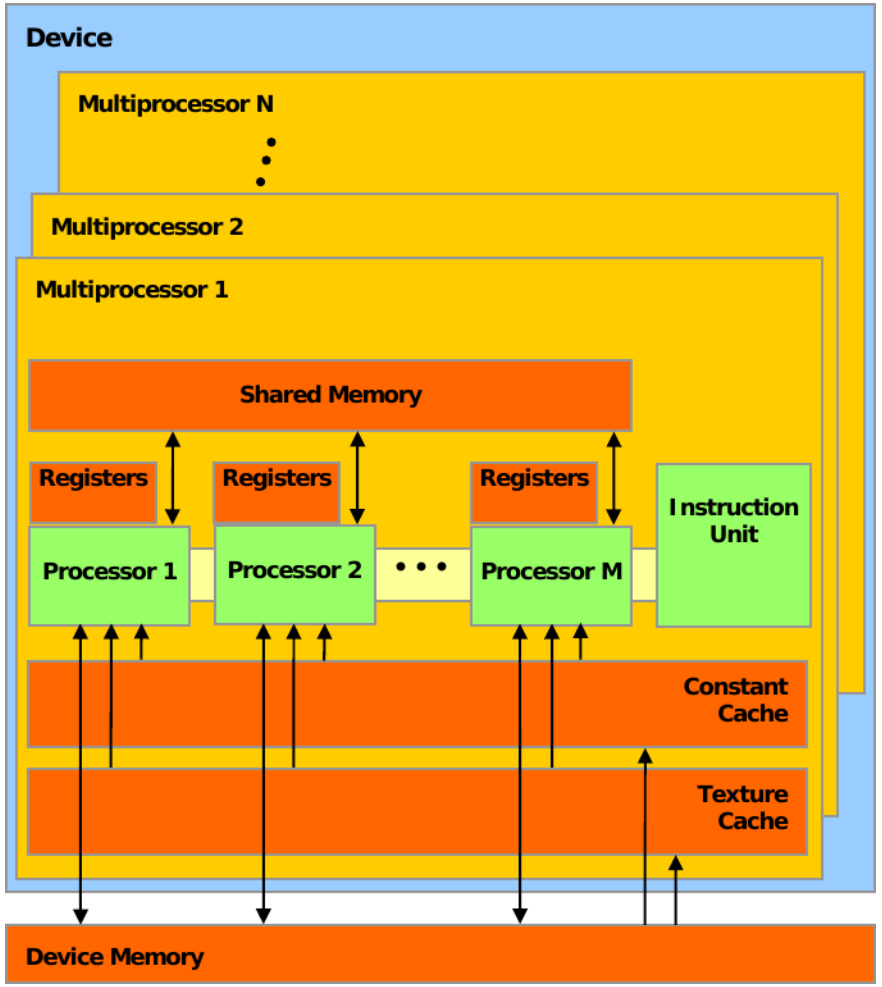
- Task:
  - Modify the kernel of exercise 2 or 3 implement a time step of the diffusion scheme on a 1D array (or just take the solution exercise05_solution.cu/.f90)
  - Re-adjust grid/block size for optimal performance
  - Compare the performance to exercise 3
- Note: please notice that the input array needs to be 1 element larger on the beginning and the end (2 in total) than the size of the computation.
  Do not forget to add offsets where necessary

GPU Programming using CUDA
Shared Memory

Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

What is shared memory?

- **Shared** between all threads in the **same block**
  - Scope is per block
- Kepler:
  - 16 kB or 48 kB per multiprocessor
- Organized into 32 banks
- **As fast as registers** (if no bank conflicts happen, discussed later)
- ‘user-managed cache’
- typical applications
  - intermediate buffer for global memory to improve memory reuse
  - Mechanism for fast exchange of data between threads of the same block



How can shared memory be used?

- In C local variables are being declared
  to reside in shared memory
  by the **__shared__** declaration specifier:

```
__global__ void myKernel() {
  __shared__ float data[size];
  ...
}
```

  In Fortran local variables are being declared
  to reside in shared memory
  by the **shared** modifier:

```
attributes(global) subroutine myKernel()
  real, shared, dimension(0,size-1) :: data
  ...
end subroutine myKernel
```

- Note: the **array size** has to be a **compile time constant**
  - Fortran: **be careful**, compiler **does not report the error**,
    but produces **undefined results** !

Pointers to shared memory in C

- Pointers can point to both shared or global memory without any special declaration:

```
__global__ void myKernel(float* input) {
  __shared__ float sharedData[size];
  float* a = input;
  float* b = sharedData;
  ...
}
```

- Restriction on devices of compute capability < 2.0: it must be possible to **decide at compile-time which memory a pointer points to** (the nvcc compiler produces a warning otherwise)

Dynamically allocated shared memory

- C: shared memory can be allocated dynamically
  by defining an extern **__shared__** array of unknown size:

```
__global__ void myKernel() {
  __shared__ float a[32];
  extern __shared__ char dynamicSharedData[];
  ...
}
```

- Fortran: shared memory can be allocated dynamically
  by defining an assumed size **shared** array:

```
attributes(global) subroutine myKernel() {
  real, shared, dimension(*) :: data
  ...
end subroutine myKernel
```

- The size of the dynamic shared memory (in bytes) must be provided in the execution configuration:

```
myKernel<<< gridSize, blockSize, dynamicSharedMemorySize >>>();
```

- Note: all dynamic shared arrays point to the same address.
  - If multiple shared arrays are required the addresses have to be corrected by hand (!)

Synchronization

- How to assure that a read operation to shared (or global) memory on one thread happens after a write operation on another thread?

```
__global__ void myKernel() {

  __shared__ float a[...];

  // thread 0 writes a[0], thread 1 writes a[1], ...
  a[threadIdx.x] = ...;

  // thread 0 reads a[1], thread 1 reads a[2], ...
  ... = a[threadIdx.x+1];
}
```

Synchronization

- Solution: the __syncthreads() function provides a barrier synchronization for all threads in the block. A thread which reaches the barrier waits until all threads of the block have also reached the barrier.

```
__global__ void myKernel() {

  __shared__ float a[...];

  // thread 0 writes a[0], thread 1 writes a[1], ...
  a[threadIdx.x] = ...;

  __syncthreads();

  // thread 0 reads a[1], thread 1 reads a[2], ...
  ... = a[threadIdx.x+1];
}
```

- Fortran:

```
call syncthreads()
```

Example: matrix transpose 1/3

- Naive implementation:

```
__global__ void transpose(float* input, float* output,
                          int sizeX,int sizeY) {

  int x=blockIdx.x*blockDim.x+threadIdx.x;
  int y=blockIdx.y*blockDim.y+threadIdx.y;
  output[x*sizeX+y]=input[y*sizeX+x];
}
```

- Problem: either read or write operation is not coalesced

Example: matrix transpose 2/3

- Implementation using shared memory (block size = 16,16):

```
__global__ void transpose(float* input, float* output,
                          int sizeX, int sizeY) {

  __shared__ float sharedMem[16][16];

  int xIn = blockIdx.x*blockDim.x+threadIdx.x;
  int yIn = blockIdx.y*blockDim.y+threadIdx.y;
  sharedMem[threadIdx.y][threadIdx.x] = input[yIn*sizeX+xIn];

  __syncthreads();

  int xOut = blockIdx.y*blockDim.x+threadIdx.x;
  int yOut = blockIdx.x*blockDim.y+threadIdx.y;
  output[yOut*sizeY+xOut] =
                          sharedMem[threadIdx.x][threadIdx.y];
}
```

- **Note:** if sizeX or sizeY not multiple of 16, additional checks are needed

Example: matrix transpose 3/3

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

0 **1** **2** **3** **4** **5** **6** **7** **8**

0	6	12	18	24	30
1	7	13	19	25	31
2	8	14	20	26	32
3	9	15	21	27	33
4	10	16	22	28	34
5	11	17	23	29	35

0 **3** **6** **1** **4** **7** **2** **5** **8**

Reduction operations

- Reduction operation
  - Combine a set values
    (which is distributed over multiple threads or blocks)
    into one value
    - E.g. by addition, multiplication, min, max

$$A = \sum_{k=0}^{N-1} a_i \qquad A = \prod_{k=0}^{N-1} a_i \qquad A = \max(a_0, a_1, ..., a_{N-1})$$

Reduction operations

- How to do this in CUDA?
  - Operations are associative (in approximation), so summation can be split
  - Build partial sums for each thread separately
  - Build partial sums for each block using shared memory
  - Combine the per block results into end result
    - Separate kernel is needed because kernel borders are the only (sensible) way to synchronize multiple blocks
      - Each block writes partial sum to global memory
      - Separate kernel reduces the partial sums
    - Note: Atomic operations can be used if the values are integers
      - E.g. `atomicAdd(int* globalSum, int value);`
      - See PG Appendix B.12
      - Kepler and Fermi also have `atomicAdd()` for single precision floats

Configuring Fermi/Kepler shared memory size

- Fermi and Kepler have in total 64 kB of memory per multiprocessor which contains both L1 cache and shared memory
- It can be configured as
  - 48 kB L1 cache and 16 kB shared memory
  - 16 kB L1 cache and 48 kB shared memory
  - 32 kB L1 cache and 32 kB shared memory (only Kepler)
- The function

```
cudaError_t cudaDeviceSetCacheConfig
                        (enum cudaFuncCache cacheConfig);
```

  sets this configuration.

- **`cacheConfig`** may be:
  - **`cudaFuncCachePreferNone`**: automatically selected (default)
  - **`cudaFuncCachePreferShared`**: 48 kB shared memory
    `cudaFuncCachePreferEqual`: 32 kB / 32 kB (Kepler only)
  - **`cudaFuncCachePreferL1`**: 48 kB L1 cache

GPU Programming using CUDA
Exercises 6+7: Scalar product

Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

Exercise 6: Scalar product using one block

- Write a CUDA program which computes the scalar product between 2 vectors using just one multiprocessor
- Follow the structure from the shared memory session:
  - Sum over all elements one thread processes
  - Copy the ‘per thread’ results to shared memory
  - Reduce the data in shared memory to one end result

- CPU code (exercise06_cpu.c):

```
float ScalarProduct(float* inputA,
                    float* inputB, int size) {
  float sum=0;
  for( int i=0;i<size;i++) {
    sum += inputA[i] * inputB[i];
  }
  return sum;
}
```

Exercise 7: Scalar product using multiple blocks

- Modify your program from exercise 6 to work with multiple blocks
- **Note:**
  You have to create a separate kernel to reduce the partial results of the separate blocks to the end result to be sure that multiprocessor synchronization is done correctly
- **Note:**
  For real world applications (especially when using cache), reductions usually work also without using shared memory.

GPU Programming using CUDA
Atomic Operations

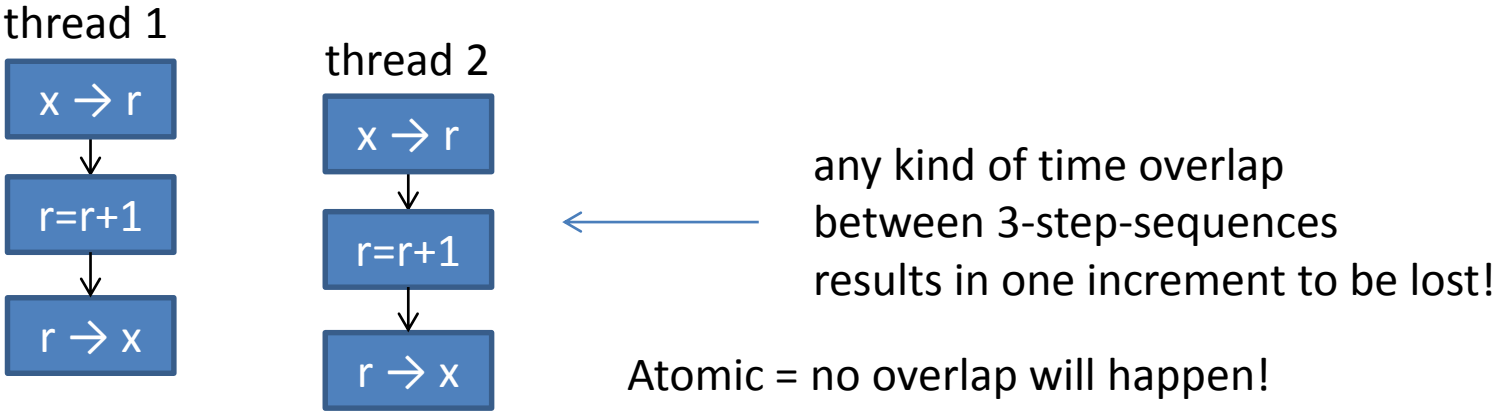
Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

Atomic operations

- Synchronization between cores
  not provided by programming model
  - only complete device synchronization at kernel boundary
  - CPU interoperation needed
    - PCIe latency
  - tightly coupled problems difficult to implement
- But device has atomic operations (add, min, max, CAS)
  - What do atomic operations do?
  - Answer: they allow a **read-modify-write** sequence
    on a single variable in memory
    without interference of other threads

Atomic operations

- Example: 2 threads want to do x = x + 1 on the same variable x at the same time
- What happens actually within machine code (of one thread)?
  - x is read from memory to register
  - value in register is incremented by 1
  - value is written back from register to memory of x



Atomic operations

- Atomics often allow easy and efficient parallelization of codes which are very hard to parallelize otherwise(!)
  - assembly of FEM sparse matrices
  - sorting algorithms
- Performance characteristics:
  - highly efficient, if no conflict happens
    - example: stream addition (a[i]=a[i]+1) on Tesla K20x
      bandwidth normal addition: 154 GB/s
      bandwidth atomic add: 139 GB/s
  - Much slower in case of conflict
    → suitable for cases, when conflicts can happen, but are rare
    - Slow down factor of up to 80, depending on number of conflicts
  - Kepler architecture: greatly improved speed of atomics
    - 32-bit integer atomicAdd() faster(!) than a[i]=a[i]+1
      (in case no conflict happens)

Atomic operations

- The following atomic operations are available in CUDA *device code*
  - **`atomicAdd`** (int32, uint32, uint64, float)
  - **`atomicSub`** (int32, uint32)
  - **`atomicExch`** (int32, uint32, uint64, float)
    `atomicCAS` (int32, uint32, uint64)
    = compare and swap
    (swap is only executed if data equals a given compare value)
  - **`atomicMin`**, **`atomicMax`** (int32, uint32)
  - **`atomicInc`**, **`atomicDec`** (uint32)
  - **`atomicAnd`**, **`atomicOr`**, **`atomicXor`** (int32, uint32)
- 32-bit operations are available for both shared and global memory on all devices with CC ≥ 1.2 but **64-bit** operations on **shared memory** require **CC ≥ 2.0**
- Note: for floating point, only atomicAdd and only in single precision available (no min/max and no DP)

Compare-and-swap

- What to do if different kind of atomic operation is required (e.g. atomicAdd for double precision)?

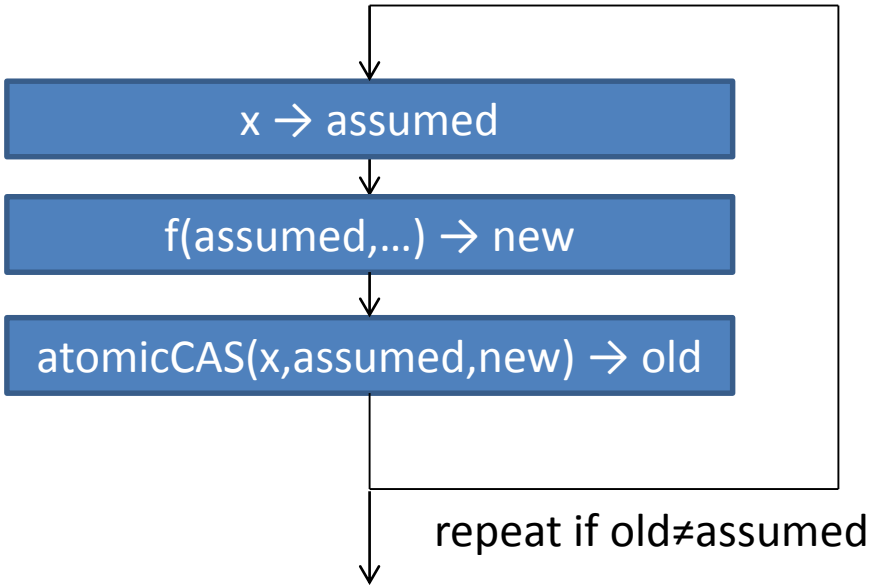
- CAS-operation, equivalent to atomically done function:

```
int atomicCAS(int* address,int compare,int value)
{
  int old=*address;
  if (old==compare)
    *address=value;
  return old;
}
```

- This is known to allow to create arbitrary atomic operations

Compare-and-swap

- Custom atomic operation using CAS:



- Principle: interference by other threads is detected, and operation is repeated if needed

Compare-and-swap

- atomicCAS exists only for integer types,
  but reinterpretation functions (__…_as_…) help.

  Example (atomicAdd for double precision):

```
typedef unsigned long long ull;

__device__ double atomicAdd(double* address, double inc)
{
  ull *addressUll = (ull*)address;
  ull oldValue=*addressUll;
  ull assumedValue;
  do {
    assumedValue=oldValue;
    ull newValue = __double_as_longlong
                (__longlong_as_double(assumedValue)+inc);
    oldValue=atomicCAS(addressUll,assumedValue,newValue);
  }
  while(oldValue!=assumedValue);
  return __longlong_as_double(oldValue);
}
```

Example: Particle sort using atomic adds

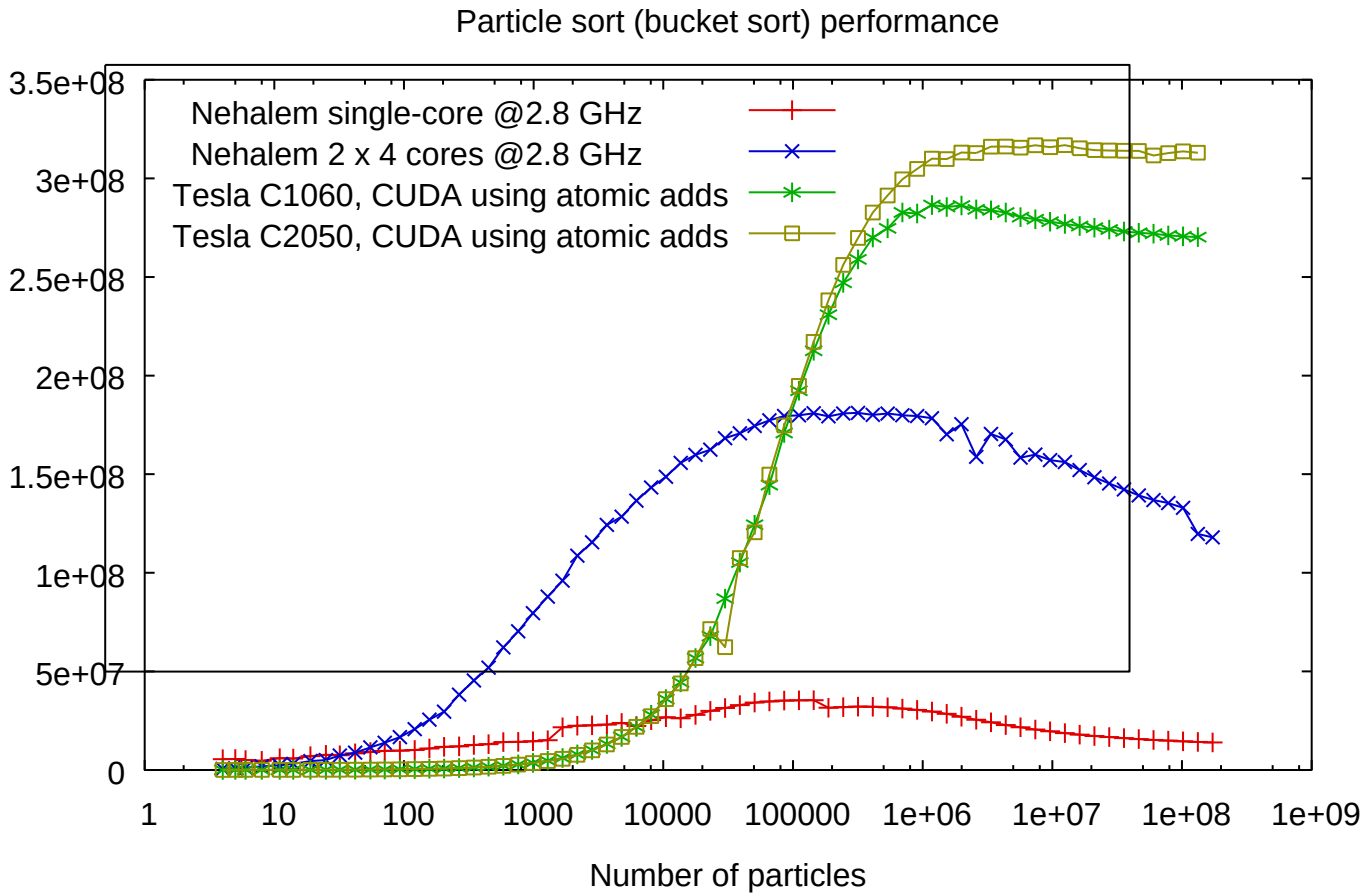
- Particle codes: sorting particles in spacial ‘cells’,
  i.e. collecting particles for each cell in a list:

  Pseudocode:

```
for(p in particles) {
  cellList[int(p.position/cellSize)].append(p.index);
}
```

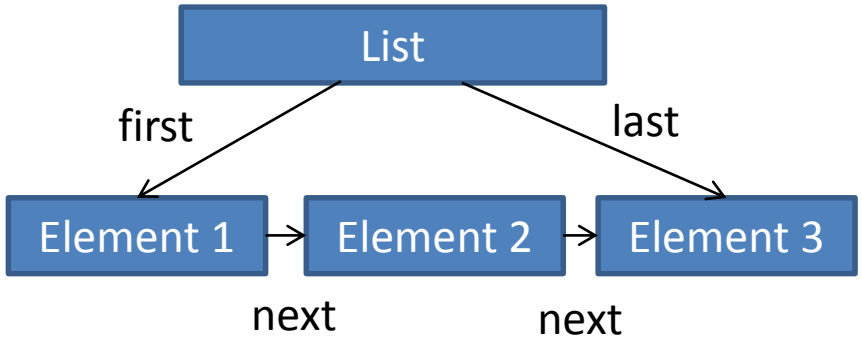
- Parallelize over particles
  → each list might be changed by any thread
- atomicAdds allows performant parallel implementation
  → exercise 9

Bucket sort using atomic adds



List operations

- Linked-list:

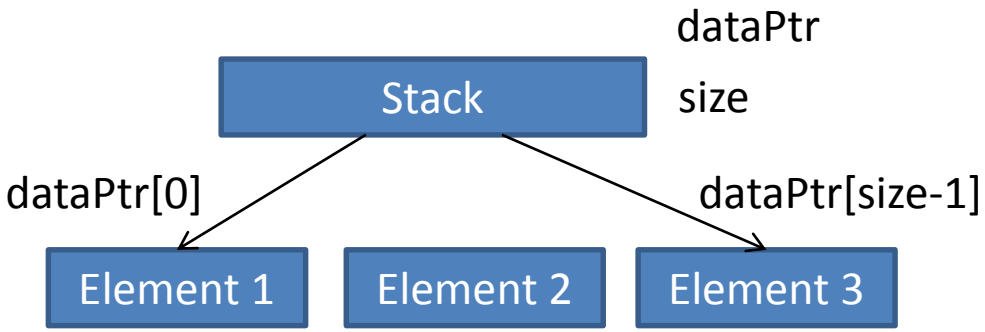


- Append operation using **`atomicExch`** (Pseudocode)

```
newElement = new Element;
previousElement = atomicExch(list.last,newElement);
previousElement.next = newElement;
```

Stack/Queue operations

- Stack:



- Append operation using **`atomicAdd`** (Pseudocode)

```
newElement = new Element;
index = atomicAdd(stack.size,1);
stack.dataPtr[index] = newElement;
```

- Note: this kind of stacks/queues can be used to implement dynamic work scheduling over all blocks/threads

Synchronizing multiple blocks

- Atomic operations can be used
  for synchronization between blocks → exercise 10
- Notes:
  - Not all blocks execute in parallel for large block numbers (limited by registers, shared memory usage, number of concurrent warps and blocks per multiprocessor), so inter-block barrier synchronization can cause deadlocks!
  - To make sure data you wrote is visible to other blocks the `__threadfence()` function needs to be called

GPU Programming using CUDA
Exercises 8-10: Atomic Operations

Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

Exercise 8: reduction using atomic add

- Task:
  - Modify the solution of exercise 7 to use atomic adds instead of synchronization mechanisms
  - Try 3 different versions
    - add all sum terms directly (atomically) to global variable
    - add up separate per-thread sums in register, then add the per thread results to global result variable
    - add per-thread results atomically to per-block result in shared memory, then add per-block results atomically to global result
    - make sure that type real is “double” otherwise you will get a rounding error (Fortran: set real kind “RK = 8”)
  - How large are the performance differences?

Exercise 9: Sort particles using atomic adds

- Problem:
  - An array of particles with coordinates (x,y) should be sorted into a 2D grid of cells with width=1 each in both directions
  - After the sorting operation, each cell shall have a list of indices of the particles which reside there
  - The lists are implemented as arrays of fixed size, with an additional size variable per list (which stores the actual current length):

```
int list[maxLength];
int listLength;

int myPosition = atomicAdd(&listLength,1);
list[myPosition] = …
```

Exercise 9: Sorting particles using atomicAdds

- CPU code:

```
for(int i=0;i<particleCount;i++) {
  int x = (int)positionsX[i];
  int y = (int)positionsY[i];
  int oldListSize =
                cellParticleCounts[y*gridSizeX+x]++;
  particleLists[(y*gridSizeX+x)
                +gridSizeX*gridSizeY*oldListSize]=i;
}
```

- To parallelize over the particles (loop i) requires to increment the list length (cellParticleCounts[…]) atomically, because multiple particles may have the same cell

- **Note:** in the particlesLists array the list of one cell is not stored consecutively but wit stride gridSizeX*gridSizeY (the same list positions for different particles are stored consecutively)

Exercise 9: Sorting particles using atomicAdds

- Task:
  - Complete the kernel in exercise09_template.cu/.f90 by creating a parallel CUDA version of the CPU implementation in exercise09_cpu.c/.f90
  - Use atomicAdd() or atomicInc() to avoid race conditions when incrementing the list size
  - Replace the atomic increment with a plain one.
    Although the kernel now produces wrong results,
    run it to see how large the performance impact
    of using atomic operations is.
- Advanced exercise (C only):
  - Replace the global memory reads of the particle positions and the writes of the particle indices into the list
    with uncached versions using PTX inline assembly
  - Do you see a performance difference?

Exercise 10 (optional): synchronization of blocks

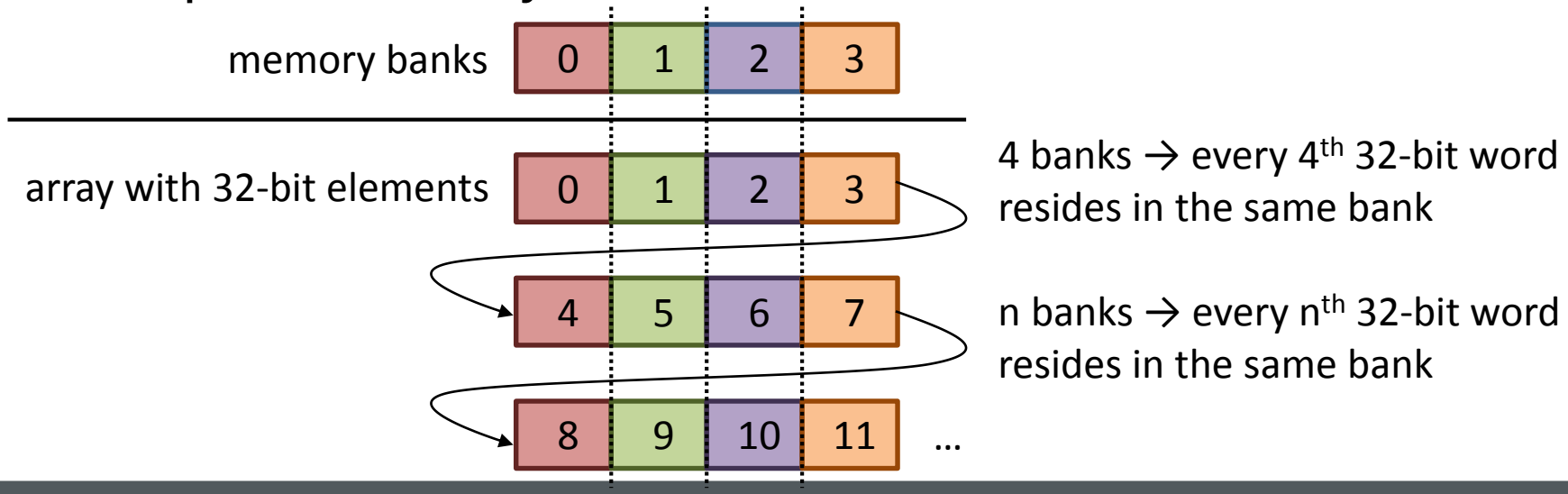
- Task:
  write a device function which barrier-synchronizes multiple blocks (as `__syncthreads()` does for multiple threads in the same block)
  - Use the exercise10_template.cu/.f90 file to verify that your function works.
- Notes:
  - You can use `atomicAdd()` to count how many blocks already reached the barrier.
  - If you create a spinlock
    ```
    while(*syncval!=blockDim.x) {
    }
    ```
    you need declare the variable that is waited on as volatile!
  - Include a call to `__threadfence()` in your function to make sure all data is visible to other blocks
  - In Fortran volatile attribute is not translated correctly in GPU code (!), but for reading `syncval`, also an atomic can be used (e.g. add zero)

GPU Programming using CUDA
Performance Optimization II

Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

Shared memory banks

- **Shared memory** is organized into banks:
  - **Consecutive 32 bit words *in shared memory* belong to consecutive memory banks**
  - If there are no bank conflicts
    shared memory is as fast as registers
- Example: 4 memory banks

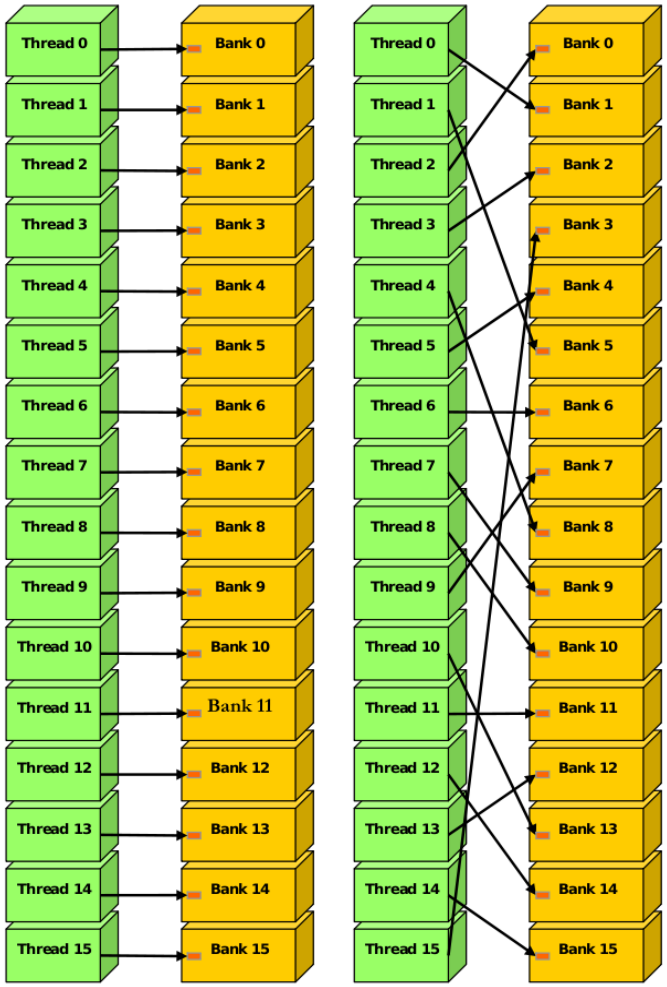


Bank conflicts

- What is a bank conflict?
  - each bank can not deliver/store more than one 64-bit/32-bit word per clock cycle
- Fermi/Kepler architecture:
  shared memory is organized in 32 banks
  - accesses are per warp,
    bank conflicts can happen between all threads in the same warp
- Older GPUs:
  shared memory is organized in 16 banks
  - accesses are per half-warp,
    bank conflicts can only happen between threads in the same half-warp
- Rule for unconflicted accesses:
  - Within a warp/half-warp no 2 threads access the same bank
  - Exception:
    - broadcast mechanism:
      one 32-bit/64-bit word can be read from arbitrarily many threads
    - (CC ≥ 3.x) access to sub-words of 32-, 64-bit words.
    - (CC ≥ 3.x) access to consecutive 32-bit words aligned to 64-bit segments.

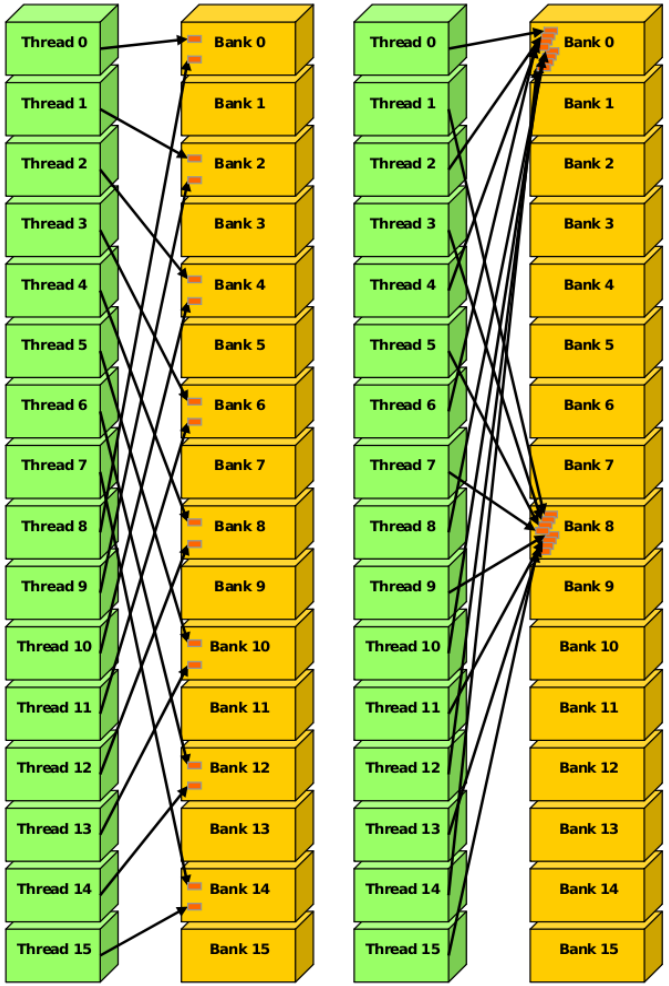
Unconflicted shared memory accesses

- Unconflicted accesses:
  - Each thread accesses a different bank
  - Right figure: arbitrary permutations are allowe



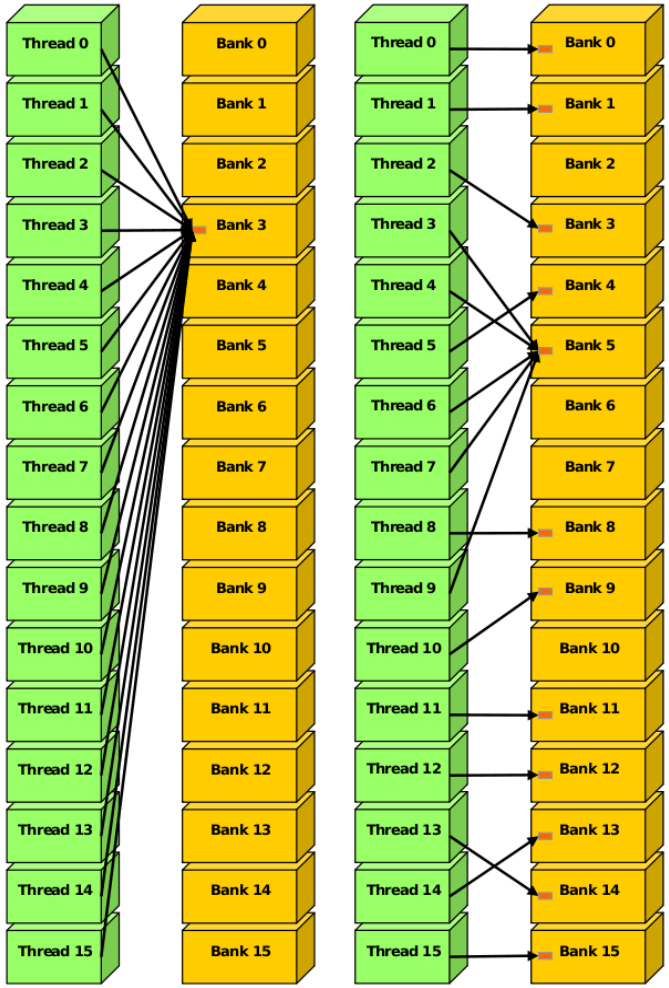
Conflicted shared memory accesses

- Conflicted accesses:
  - multiple threads of the same half-warp access the same bank
  - left figure: 2-way conflict
  - right figure: 8-way conflict



Broadcasted shared memory accesses

- Broadcast mechanism:
  - multiple threads access the same 32 bit word
  - right figure: broadcast may be mixed with unconflicted accesses to other banks
  - is as fast as unconflicted access
  - CC ≤ 1.3: **only one word** can be broadcast per cycle
  - CC ≥ 2.0: **multiple words** can be broadcast per cycle



Shared memory - Changes in Kepler

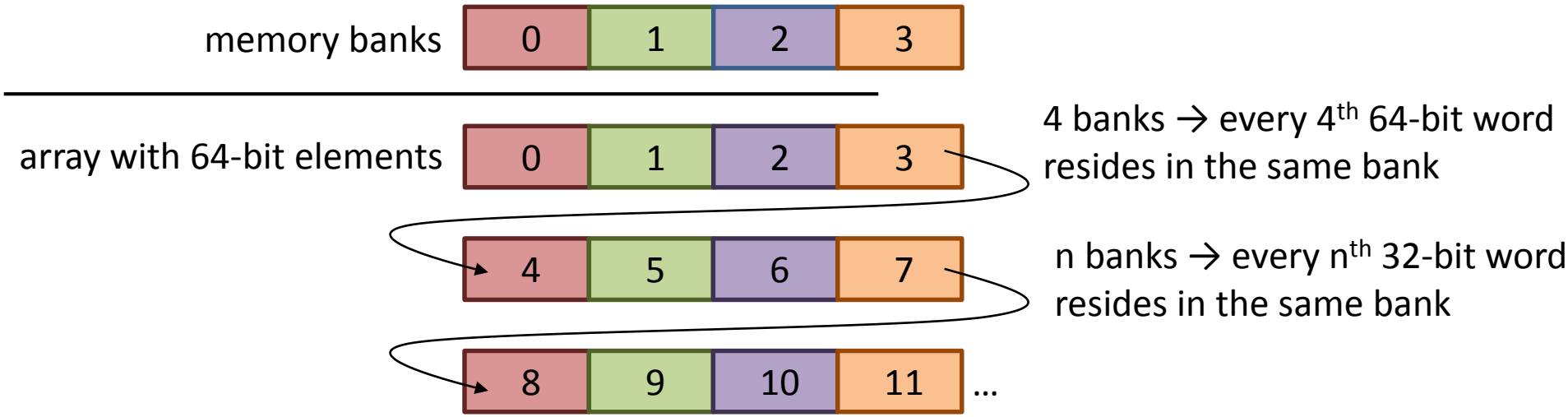
- Kepler allows configuration of shared memory

- Can be set globally and per device function
  - **`cudaDeviceSetSharedMemConfig(config)`**
  - **`cudaFuncSetSharedMemConfig(func, config)`**

- The supported bank configurations are:
  - **`cudaSharedMemBankSizeDefault`** (currently 32 bit)
  - **`cudaSharedMemBankSizeEightByte`** (64 bit mode)
  - **`cudaSharedMemBankSizeFourByte`** (32 bit mode)

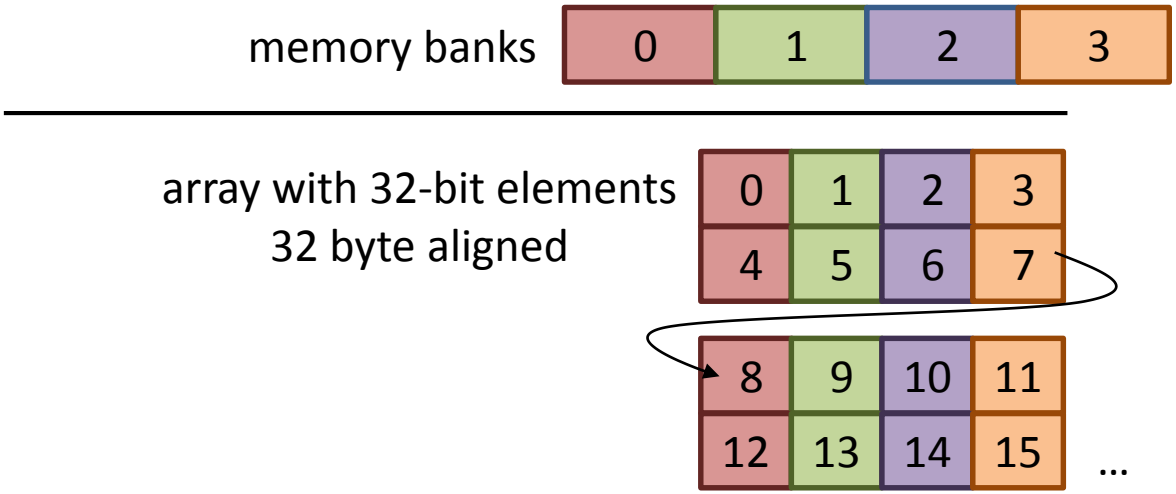
Shared memory in Kepler - 64 bit mode

- Consecutive **64 bit** words in shared memory belong to consecutive memory banks
  - Suitable for 64-bit data elements, e.g. double precision
- Conflict determination also based on 64-bit words
- Example: 4 memory banks



Shared memory in Kepler - 32 bit mode

- Consecutive **32 bit** words in shared memory belong to consecutive memory banks (as for older devices)
- But within **same 256 byte segment** (64 32-bit words) addresses with **a distance of 32 32-bit words** cause no bank conflict
- Example: 4 memory banks



Strided access to shared memory

- Note: if you access shared memory in a **strided fashion** on CC ≤ 3.0

```
__shared__ float x[blockDim.x*n];
…x[threadIdx.x*n]…;
```

you will get bank conflicts if n is a multiple of 2.

- The same for is true for **2-dimensional arrays**
  when using threadIdx.x on the first array index (and n is again a multiple of 2)

```
__shared__ float x[blockDim.x][n];
…x[threadIdx.x][j]…;
```

- How to avoid bank conflicts if strided access patterns are necessary?
- Standard solution: choose the stride to be not a multiple of 2
  by **padding you data with one unused element** per stride, e.g.

```
__shared__ float x[blockDim.x*(n+1)];
…x[threadIdx.x*(n+1)]…;
```

or for 2-dimensional arrays

```
__shared__ float x[blockDim.x][n+1];
…x[threadIdx.x][j]…;
```

Shared memory applications

- Clever splitting of the problem and buffering in shared memory can speed up things enormously
- General strategy:
  - Copy data from global memory to shared memory
  - Do (much) computation on data in shared memory
  - Copy data back to global memory
- Examples
  - Dense Matrix-Matrix-Multiplication (see PG)
  - FFT

Note on shared memory usage

- As accesses to global memory have long latency to get full memory bandwidth *latency hiding* is necessary
  - If the number of blocks is larger than the number of multiprocessors, block execution will be overlapped (multitasking)
    as long as the available registers
    and the available shared memory is enough
    for multiple blocks
- Effect: using shared memory can result in performance loss (!)
- Advice: decide carefully about the amount of shared memory per block you request

Texture memory

- Access to texture memory is not as fast as access to shared memory
- But texture fetches are always coalesced and cached
- 2D or 3D textures
  - Cache is optimized for access patterns localized in 2D or 3D
  - Cache lines are not organized linearly but in 2D or 3D blocks
  - Example application: matrix transpose (although shared memory is faster)

Texture memory applications

- L1 cache is typically faster than texture cache
  - per default try to use L1/L2 cache instead of texture cache
- But L1/L2 cache still requires coalesced accesses to be efficient
- The texture cache shows performance characteristics completely different from the L1/L2 cache
  → **uncoalesced accesses, which result in good cache reuse**, e.g. continuous regions per thread as in exercise 4b or exercise 8 **are faster via texture memory (!)**
- Other application: prevent cache pollution
  - Use texture memory to prevent data to be cached in L1

Unroll pragma

- Control flow overhead is a more serious issue as on CPU
- So unrolling of loops is recommended for computation bound problems
  - Note on Fermi/Kepler effect is smaller than for older devices
- The unroll pragma can help here

```
#pragma unroll 8
for(int i=0;i<n;i++) {
   …
}
```

- In Fortran this does not work, but compilation with –O3 sometimes results in automatic unrolling (check PTX assembler output)

PTX Assembler Output

- The CUDA compiler creates output in the intermediate device-independent (assembler-like) language PTX which is compiled further to device-specific CUBIN code
  - E.g. PTX uses virtual registers which are assigned to physical registers during the PTX-to-CUBIN step
- Viewing the PTX output may help to identify performance issues (the CUDA compiler produces not always optimal code)
- The option '**`--ptx`**' forces the Nvidia compiler to produce a PTX output file
- Fortran: add option keepptx to **`-Mcuda`**, e.g. **`-Mcuda=cuda5.0,cc35,keepptx`**

```
.entry _Z6kernelPfS_ (
                .param .u64 __cudaparm__Z6kernelPfS__a,
                .param .u64 __cudaparm__Z6kernelPfS__b)
        {
        .reg .u16 %rh<4>;
        .reg .u32 %r<5>;
        .reg .u64 %rd<8>;
        .reg .f32 %f<5>;
        .loc    15      5       0
$LBB1__Z6kernelPfS_:
        .loc    15      7       0
        cvt.u32.u16     %r1, %tid.x;
        mov.u16         %rh1, %ctaid.x;
        mov.u16         %rh2, %ntid.x;
        mul.wide.u16    %r2, %rh1, %rh2;
        add.u32         %r3, %r1, %r2;
        cvt.u64.s32     %rd1, %r3;
        mul.lo.u64      %rd2, %rd1, 4;
        ld.param.u64    %rd3, [__cudaparm__Z6kernelPfS__b];
        add.u64         %rd4, %rd3, %rd2;
        ld.global.f32   %f1, [%rd4+0];
        mov.f32         %f2, 0f41300000;        // 11
        add.f32         %f3, %f1, %f2;
        ld.param.u64    %rd5, [__cudaparm__Z6kernelPfS__a];
        add.u64         %rd6, %rd5, %rd2;
        st.global.f32   [%rd6+0], %f3;
        .loc    15      8       0
        exit;
$LDWend__Z6kernelPfS_:
        } // _Z6kernelPfS_
```

PTX inline assembly / cache operators

- CUDA 5.0 supports writing inline assembly in C device code
- Example: load and store instructions have modifiers to specify caching behavior
  - .ca=cache on all levels
  - .cg=cache only in L2
  - .cs=do not cache in L1/L2

- Load 32-bit float uncached from global memory:

```
float* address;
float v;
asm("ld.global.cs.f32 %0,[%1];\n"
        : "=f"(v) : "l"(address) : );
```

- Store 32-bit int uncached to global memory:

```
asm("st.global.cs.s32 [%0],%1;\n"
      ::"l"(address),"r"(v));
```

- Note: setting the caching mode globally can be achieved with the nvcc compiler option **`-Xptxas -dlcm=cg`** (or **`cs`**)

Notes on CUDA Fortran

- The PGI compiler uses intermediate C code
  (CUDA Fortran→CUDA C→PTX Assembler)
  → sometimes performance issues
  - Example: multidimensional arrays
    are sometimes slower than index translation by hand
  - Note: the intermediate CUDA C code can be reached
    with the compiler option 'keepgpu':
    `pgfortran -Mcuda=5.0,keepgpu` …
- Some features of newer CUDA versions
  are not available in Fortran, yet:
  - full unified address space support
    - **`pinned`** arrays cannot be passed to kernels
  - allocating memory from GPU code
  - printing from GPU code (has bugs)

NVIDIA Visual Profiler

- Records various kinds of GPU performance-related information during program execution
- Usage:
  - **`nvvp (Cuda ≥ 4.0), computeprof (Cuda < 4.0)`**
  - File→New
  - Select project name and directory
  - In tab ‘Session’ enter
    - Executable path (‘Launch’)
    - Working directory, program arguments (optional)
  - In tabs ‘Profiler Counters’ and ‘Other Options’ select the information you want to record
  - Click ‘Start’

NVIDIA Visual Profiler

- Meaning of profiler counters:

gld uncoalesced	Number of non-coalesced global memory loads
gld coalesced	Number of coalesced global memory loads
gld request	Number of global memory load
gld_32/64/128b	Number of 32 byte, 64 byte and 128 byte global memory load transactions
gst uncoalesced	Number of non-coalesced global memory stores
gst coalesced	Number of coalesced global memory stores
gst request	Number of global memory store requests
gst_32/64/128b	Number of 32 byte, 64 byte and 128 byte global memory store transactions
local load	Number of local memory loads
local store	Number of local memory stores
tlb hit	Number of instruction or constant memory cache hits
sm cta launched	Number of instruction or constant memory cache misses
branch	Number of threads blocks launched on a multiprocessor
divergent branch	Number of divergent branches within a warp
instructions	Number of instructions executed
warp serialize	Number of thread warps that serialize on address conflicts to either shared or constant memory
cta launched	Number of threads blocks executed

Notes on CUDA Profiler

- The Hardware has a limited amount of profiling registers
  - The program will be run multiple times to collect all profiling information
  - Attention: the results may become useless if the program does not run the same way each time
    - Random Number Generators
    - OpenMP dynamic scheduling
- Most interesting: GPU time … plot
  - Quick overview over time usage in complex application with many kernels
  - Computation time vs. data transfer time
  - Fortran and accelerator models: can be used to check for unnecessary data copy
- More interesting fields
  - Profiler Output
    - Static shared memory per block
    - Registers per thread
    - Occupancy = number of concurrent warps / maximum number of concurrent warps
      - Note: it assumes only 1 MP in the GPU, it is not considered if there are enough blocks to keep all MPs busy !
    - Branch vs. divergent branch
  - Summary Table
    - Global mem … throughput

Command line profiling

- CUDA profiling can be enabled manually without the visual profiler
  - e.g. for profiling MPI-parallel programs
- Either **nvprof <options> <executable> <arguments>**
- or set the environment variable **COMPUTE_PROFILE=1**
  - Profiling information will be written into a log file
  - Additional options
    - **COMPUTE_PROFILE_CSV=1** (switch output from 'human readable' to CSV format for import into visual profiler)
    - **COMPUTE_PROFILE_LOG=**<filename> explicitly set logfile name
    - **COMPUTE_PROFILE_CONFIG=**<filename> set profiler configuration file

Profiler configuration file

- The profiler configuration file is simply a list of the records / counters the profiler should log (with one entry per line)
- The following entries may be allowed in the configuration file, (depending on the compute capability of your hardware)

```
timestamp, gpustarttimestamp, gpuendtimestamp, streamid,
gridsize,threadblocksize,dynsmemperblock,stasmemperblock,regperthread,
memtransferdir, memtransfersize, memtransferhostmemtype,
local_load, local store, gld_request, gst_request,
divergent_branch, branch, sm_cta_launched,
gld_incoherent, gld_coherent, gld_32b, gld_64b, gld_128b,
gst_incoherent, gst_coherent, gst_32b, gst_64b, gst_128b,
instructions, warp_serialize, cta_launched,
prof_trigger_00...prof_trigger_07,
tex_cache_hit, tex_cache_miss, shared_load, shared_store,
inst_issued, inst_executed, warps_launched, threads_launched,
l1_global_load_hit, l1_global_load_miss
```

- Note: for devices of compute capability 1.x only 4 device counters can be used at the same time!
- Note: a detailed description of these records can be found in the *Command Line Profiler User Guide*
- Note: an example ‘profile.conf’ exists that records basic timing information

Kernel time function

- In kernel code the function clock() returns the value of the GPU time counter.
- Fortran: **`call gpu_time(value)`**
- The result is in GPU core clock cycles
- The clock rate returned by **`cudaGetDeviceProperties()`** can be used to convert the results to seconds

Note for the Fermi and Kepler architecture

- Fermi supports 64 bit address space
  but can only do 32 bit integer operations natively
  - → Performance penalty on address operations
  - If no more than 4 GB of memory is needed,
    the compiler option ‘-m32’
    can be used to switch back to 32 bit address mode
- **Note:** CUDA 4.0 unified addressing works only
  in 64 bit mode

Features of Kepler

- New multiprocessor design (SMX)
  - A SMX contains 192 compute cores (old SM: 32)
    and can execute within one cycle instructions from up to 4 warps
    (up to 2 instructions per warp)
  - Roughly half GPU clock rate (C2050: 1150 MHz, GTX680: 700 MHz)
  - Note: warp size stays 32 threads!
- Kepler I (e.g. GTX680): only 8 SMX (Fermi 14-16 SM)
  - Multiprocessors became larger, but fewer
  - Little DP performance (1/24 of SP, Fermi C20xx: 1/2)
  - Very fast atomics
  - Ratio of registers and shared memory to compute cores smaller
    - Fermi: 32k registers, 16/48 kB shared memory, 32 compute cores per SM
    - Kepler I: 64k registers, 16/32/48 kB shared memory, 192 compute cores per SMX
- Kepler II (Tesla K20X)
  - Again high DP performance (1/3)
  - Higher memory bandwidth (250 GB/s)

Summary of optimization tactics

- Stencil-based operators (FDTD, Lattice Boltzmann)
  - Fermi L1 cache
- Reduction
  - 2 Kernels:
    - A: reduction per thread, then per multiprocessor
    - B: reduction over the per multiprocessor results
  - Atomic adds
- Sparse Matrix-Vector-Multiplication
  - Padding of matrix data for coalesced accesses (ELLPACK-R)
  - In some cases caching of input vector in texture memory is of advantage
- Dense Matrix-Matrix-Multiplication
  - Blocking in Shared Memory
- Fast Fourier Transform
  - Blocking in Shared Memory
- Sorting
  - Bitonic Merge Sort with blocking in shared memory
  - Bucket sort / radix sort can be done using atomic adds
- Assembling of FEM matrices
  - Atomic adds

GPU Programming using CUDA
Exercise 11: Small matrix-matrix multiplications

Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

Computation task

- A multiplication of 2 matrices A (N x N), B (N x N)

$$\underline{\underline{C}} = \underline{\underline{A}} \cdot \underline{\underline{B}}$$

can be done independently for each element of C:

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} \cdot B_{kj}$$

- The operation can be done easily using one CUDA block of sizes N x N
- **Note:** the problem is of course too small to use all multiprocessors efficiently
  - But if many such matrix multiplications need to be done each CUDA block can process one MM

$$\underbrace{C_{ij}^{(0)} = \sum_{k=0}^{N-1} A_{ik}^{(0)} \cdot B_{kj}^{(0)}}_{\text{Block } 0} \quad \underbrace{C_{ij}^{(1)} = \sum_{k=0}^{N-1} A_{ik}^{(1)} \cdot B_{kj}^{(1)}}_{\text{Block } 1} \quad \ldots \quad \underbrace{C_{ij}^{(M)} = \sum_{k=0}^{N-1} A_{ik}^{(M)} \cdot B_{kj}^{(M)}}_{\text{Block } M} \quad \ldots$$

Optimized version

- Disadvantage of naïve CUDA implementation: each element of A and B is read N times
- Buffer the matrices A,B in shared memory
- Algorithm:
  - Each thread copies one element of the N x N matrices A and B to shared memory
  - Each thread computes the sum for one element of matrix C

$$C_{ij} = \sum_{l=0}^{N-1} A_{il} \cdot B_{lj}$$

- Advantage: each element of A or B is only read once from global memory

Task

- Complete the template program exercise11_template.cu
  to copy matrices A,B to shared memory before computation.
- Use 2D static arrays for the shared data:

```
__shared__ float sharedA[T][T];
__shared__ float sharedB[T][T];
```

- Note: the program computes MMs for many matrices of the same size N x N
  - Each block processes one matrix multiplication
  - The multiple matrices $A^{(i)}$ (or $B^{(i)}$ or $C^{(i)}$) are stored consecutively in memory
- Note: the matrices in global memory A_{uv}, B_{uv}, C_{uv} are stored
  in a linear array with the mapping

  $index_{linear} = m*N^2 + u*N + v$

  where m is the index of the small NxN matrix
  in the list of the many small matrices to be multiplied

Advanced Exercise

- Is it better to store the matrix elements B_{ij} as `B[i][j]` or `B[j][i]` in shared memory? Why? How is it for A_{ij}?
  - Hint: think about bank conflicts during the accesses to shared memory
  - Can you think of a trick to use the ‘bad’ storage ordering for B and still get performance as fast as for the good ordering?
- Try to change the shared arrays to 1-dimensional arrays (Fortran only)
  Do you see a performance change? Why?

Answers

- Is it better to store the matrix elements B_{ij} as **`B[i][j]`** or as **`B[j][i]`**?
  - Answer: **`B[i][j]`** is considerably faster
- Why?
  - Answer: because the shared array is accessed as **`shared_b[l][threadIdx.x]`** when using **`B[i][j]`** and as **`shared_b[threadIdx.x][l]`** when using **`B[j][i]`**. The second way will cause **a bank conflict on every access**
- How is it for A_{ij}?
  - Answer: it matter much less because shared_a is read either as **`shared_a[threadIdx.y][l]`** or as **`shared_a[l][threadIdx.y]`** which **does not depend on threadIdx.x**.

    The only location it matters is when reading A from global to shared memory which happens much fewer times.

Answers

- Can you think of a trick to use the ‘bad’ storage ordering for B and still get performance as fast as the for the good ordering?
  - Yes, use the standard trick for strided shared memory accesses and allocate shared_b as

```
__shared__ float
  shared_b[matrixSize][matrixSize+1];
```

- Try to change the shared arrays to 1-dimensional arrays (Fortran only)
  Do you see a performance change? Why?
  - Because the Cuda Fortran to Cuda C compiler also converts Fortran 2D arrays to C 1D arrays, but the Cuda C code is often more efficient if used on 1D arrays directly in Fortran

GPU Programming using CUDA
Advanced Features

Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

Getting hardware information

- Number of devices in system:
  ```
  cudaGetDeviceCount(int * count);
  ```

- Name and properties of device:
  ```
  cudaGetDeviceProperties(cudaDeviceProp* prop,
                          CUdevice dev);
  ```

- Note: try running the example program deviceinfo.cu, it prints most of the information returned by **`cudaGetDeviceProperties()`**

Getting hardware information

- Example: GTX 680 hardware information (deviceinfo.cu results):

device name: GeForce GTX 680
global memory: 2147155968 bytes
maximum shared memory per block: 49152 bytes
maximum number of registers per block: 65536
warp size: 32
maximum number of threads per block: 1024
maximum block dimensions : (1024,1024,64)
maximum grid dimensions: (2147483647,65535,65535)
maximum number of threads per multiprocessor: 2048
GPU clock frequency: 705.500000 MHz
memory clock frequency: 3004.000000 MHz
total constant memory: 65536 bytes
compute capability: 3.0
can execute multiple kernels concurrently: yes
can overlap data transfer and computation: yes
can overlap data transfers from/to device: no
can map host memory: yes
unified address space enabled: yes
number of multiprocessors: 8
compute mode: default (multiple processes/threads per device)
kernel timeout enabled: no
ECC memory enabled: no
is integrated device: no
memory bus width: 256 bits
size of L2 cache: 524288 bytes

Unified address space

- All types of memory the device can access reside in the same address space.
  Pointers to all of these memories can directly be passed to kernels.
  - Global memory
  - Shared memory
  - Constant memory
  - Page-locked host memory
  - Global memory of other GPUs on the same host (‘GPUdirect’)
- Prerequisites:
  - Fermi GPU (CC ≥ 2.0)
  - 64 bit address space
  - CUDA 4.0

- **Note:** in Fortran not really supported, yet
  - Consequence: host memory can not be accessed directly
    - **`pinned`** arrays cannot be passed to kernels
    - but access of memory of other GPUs should be possible

Zero-copy host memory access / peer-to-peer access

- Zero-copy host memory access
  - direct dereferencing of pointers to CPU memory on device
  - works only on page-locked host memory
  - with unified address space no further conditions
  - older devices:
    - enable zero-copy access for device:
      `cudaSetDeviceFlags(cudaDeviceMapHost);`
    - allocate page-locked host memory with device mapping:
      `cudaHostAlloc(&ptr,size,cudaHostAllocMapped);`
    - translate CPU address to GPU address:
      `cudaHostGetDevicePointer(&gpu_ptr,cpu_ptr,0);`
  - Fortran: same API functions, but works on `type(c_ptr)`, `type(c_devptr)`
    → is necessary as CUDA Fortran is not aware of unified addressing
- Peer-to-peer-access
  - Prerequisites: unified address space, both GPUs on same PCIe bus
  - Enable peer-to-peer-access from active device to peer device:
    `cudaDeviceEnablePeerAccess(peer_device,0);`

Streams

- Streams are a mechanism to allow and manage concurrent execution of multiple kernels or memory copy operations and kernels
- cudaStream_t stream;
- Basic principle of operation:
  - Each operation on the device is attached to a stream (calls that do not provide a stream handle are attached to the default stream, called ‘stream 0’)
  - Operations on the same stream are always executed in serial
  - Operations on different streams may be executed in parallel

Usage of streams

- Creation
```
cudaStream_t stream;
cudaStreamCreate(&stream);
```
- Cleanup
```
cudaStreamDestroy(stream);
```
- Asynchronous copy attached to a stream
```
cudaMemcpyAsync(dest, src, size, type, stream);
```
  - Note: concurrency only works with page-locked memory!
  - Note: PCIe bus is bidirectional. CC ≥ 3.0 can do 2 copies (1 to device / 1 to host) concurrently
  - Note for Fortran: memcpy operations exist also
    - but do not need a type (direction is derived from src and dest types)
    - size is in array elements, not in bytes (!)
- Kernel call attached to a stream
```
MyKernel<<<grid, block, shared_mem, stream>>>(…);
```
- Waits for all operation of a specific stream
```
cudaStreamSynchronize(stream);
```
- Wait for all operations of all streams (including stream 0)
```
cudaDeviceSynchronize();
```

Notes on streams

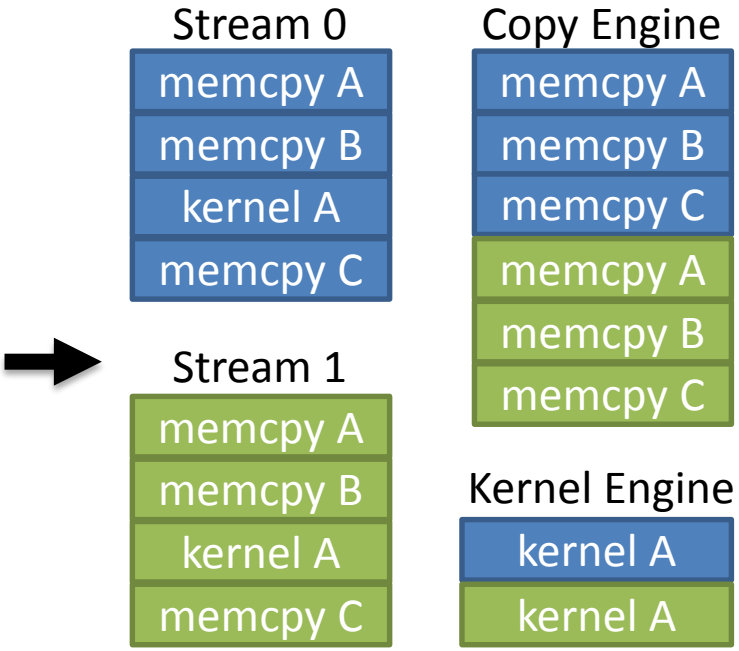
- Concurrency between kernel execution and copy from/to device is possible exactly if the device property ‘deviceOverlap’ is set
- Concurrency between multiple kernels is possible exactly if the device property ‘concurrentKernels’ is set
  - Note: very useful if many small kernels need to be executed!
- Some tasks (page-locked host memory allocation, device memory allocation, device-to-device memory copy, operations to stream 0) disable concurrency between an operation before and after them

Executing parallel streams

- Special care needs to be taken when executing streams

```
cudaMemcpyAsync(dev_a0,…,stream0);
cudaMemcpyAsync(dev_b0,…,stream0);
kernel<<<block,grid,0,stream0>>>
    (dev_a0,dev_b0,dev_c0);
cudaMemcpyAsync(host_c0,…,stream0);

cudaMemcpyAsync(dev_a1,…,stream1);
cudaMemcpyAsync(dev_b1,…,stream1);
kernel<<<block,grid,0,stream1>>>
    (dev_a1,dev_b1,dev_c1);
cudaMemcpyAsync(host_c1,…,stream1);
```



With one copy engine second kernel has to wait for first one to finish.

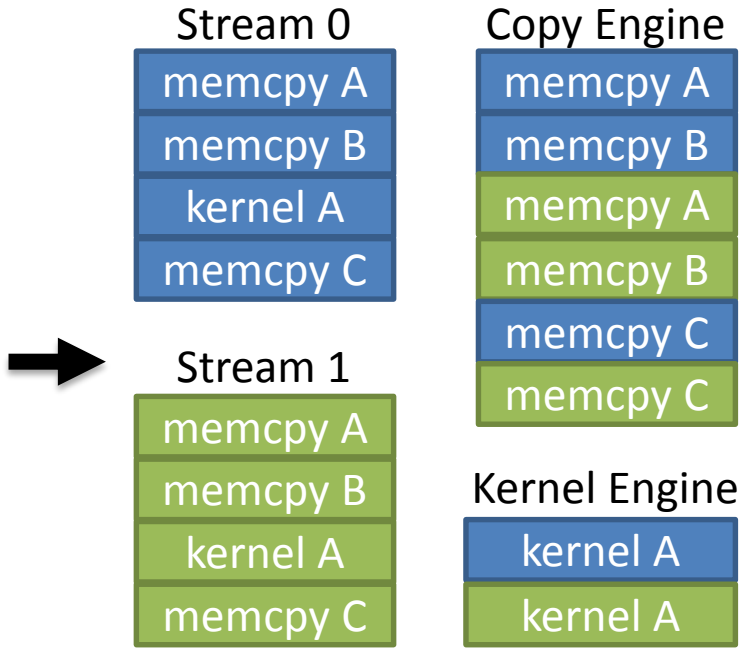
Executing parallel streams II

- Solution: Reorder copy operations

```
cudaMemcpyAsync(dev_a0,…,stream0);
cudaMemcpyAsync(dev_b0,…,stream0);
kernel<<<block,grid,0,stream0>>>
    (dev_a0,dev_b0,dev_c0);

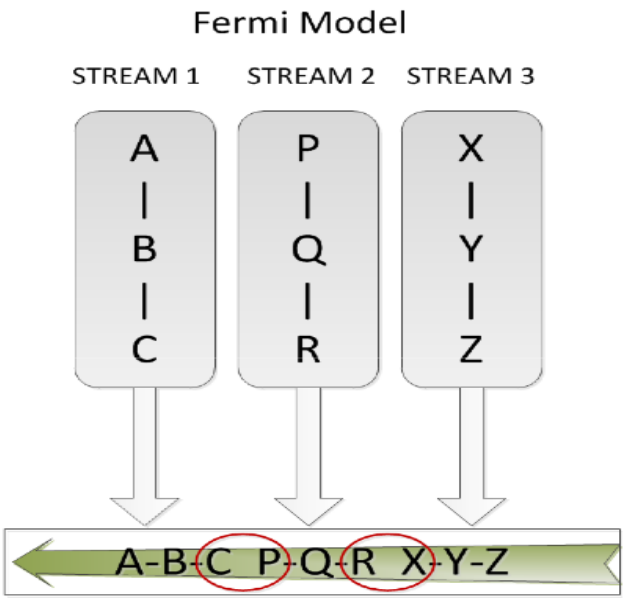
cudaMemcpyAsync(dev_a1,…,stream1);
cudaMemcpyAsync(dev_b1,…,stream1);
kernel<<<block,grid,0,stream1>>>
    (dev_a1,dev_b1,dev_c1);

cudaMemcpyAsync(host_c0,…,stream0);
cudaMemcpyAsync(host_c1,…,stream1);
```

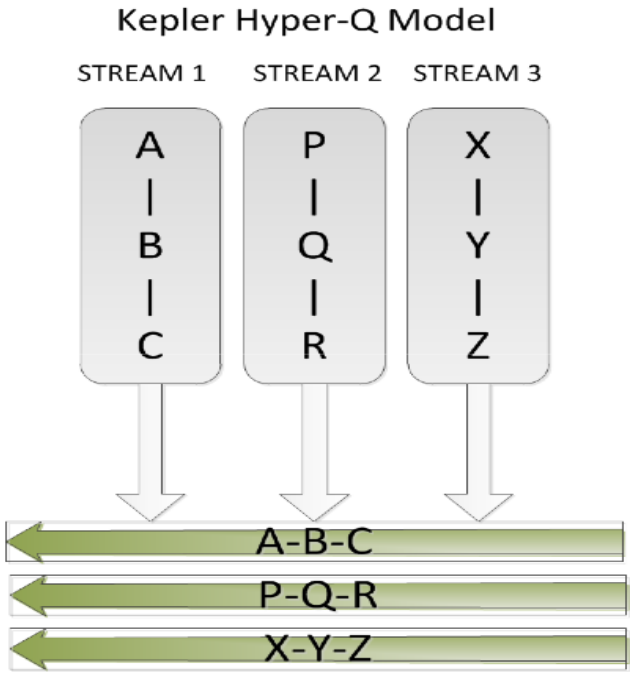


Kernels can run in parallel now.

Hyper-Q in Kepler K20x



- Only C&P and R&X run concurrently
- Only one work queue



- All streams can run concurrently
- 32 work queues

Notes:

- CUDA 5.5 and CC ≥ 3.5 allow 32 MPI-processes to use the device concurrently in proxy mode.
- On CRAY: export CRAY_CUDA_PROXY=1

Using multiple GPUs

- With CUDA 4.0 or higher:
  - `cudaSetDevice(int device_index);` can be used to switch between devices
  - All following CUDA API calls are using the active device
    - Memory allocations
    - Stream creation
    - …
  - Note: streams are always bound to one device
    → it is not possible to attach operations on device A to a stream belonging to device B or vice versa
  - Note: `cudaDeviceSynchronize()` waits only for operations on the active device (!)
  - When programming multithreaded, the active device (‘device context’) is set for the calling thread only, each thread may have a different active device

Using multiple GPUs

- CUDA 4.0 example for 2 GPUs:

```
cudaSetDevice(0);
…start some asynchronous operations for device 0…
…(cudaMemcpyAsync or kernel calls)…

cudaSetDevice(1);
…start some asynchronous operations for device 1…

cudaSetDevice(0);
cudaDeviceSynchronize();
cudaSetDevice(1);
cudaDeviceSynchronize();
```

Using multiple GPUs

- Older CUDA versions:
  - `cudaSetDevice(int device_index);`
    can be used to set device for host thread
  - Restriction:
    after usage of device device context cannot be changed again for the current thread
    (except by usage of low-level driver API cuCtx… functions)
  - Simplest solution:
    create multiple host threads with OpenMP,
    each thread operates only on one device

Using multiple GPUs with OpenMP

- OpenMP Example for 2 GPUs:

```
#pragma omp parallel
{
    int thread=omp_get_thread_num();
    if (thread==0) {
      cudaSetDevice(0);
      …cuda code for device 0…
      cudaDeviceSynchronize();
     …
    } else if (thread==1) {
      cudaSetDevice(1);
      …cuda code for device 1…
      cudaDeviceSynchronize();
     …
    }
  }
```

- Note: Even with CUDA 4 sometimes using multiple (OpenMP) host threads when working with multiple GPUs has advantages:
  - No need to switch between devices all the time
  - No need to use asynchronous copies

Using multiple GPUs with OpenMP (Fortran)

- OpenMP Example for 2 GPUs:

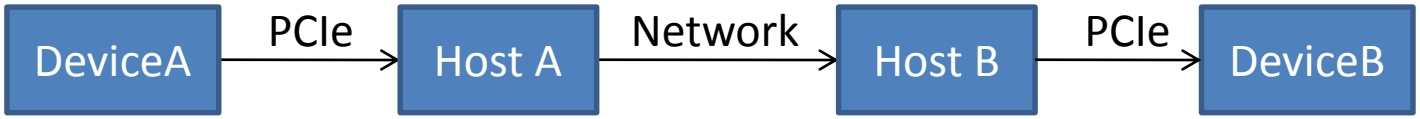
```
!$omp parallel
thread = omp_get_thread_num()
if (thread==0) then
  print *,'starting computation on GPU 0'
  error = cudaSetDevice(0)
  …cuda code for device 0…
else if (thread==1) then
  print *,'starting computation on GPU 1'
  error = cudaSetDevice(1)
  …cuda code for device 1…
end if
!$omp end parallel
```

Data transfer between devices

- With unified address space:
  - **`cudaMemcpy()`** with **`kind = cudaMemcpyDefault`**
    works for all kinds of copies including peer-to-peer copy
  - Note: **`kind = cudaMemcpyDeviceToDevice`**
    does not work for peer-to-peer copy!
- With CUDA 4.0 but CC < 2.0
  - **`cudaMemcpyPeer(void* destAddress, int destDevice, void* srcAddress, int srcDevice, size_t bytes);`**
  - works always, but direct peer-to-peer copy is only done
    if hardware supports it
  - **Note:** call is **asynchronous**,
    but serialized with other operations on stream 0
    (like kernel calls without stream specification) !

Using multiple GPUs on different nodes

- What if you want to use **more GPUs than are available to one host / node**?
- Answer: distributed memory parallelism, e.g. with **MPI** (Message Passing Interface)
  - There are separate courses on MPI at HLRS
- Principle: multiple CPUs in a cluster cooperate by sending messages over a fast local network (Infiniband, etc.)
- **MPI+CUDA**: no problem in principle, but for cooperation **messages have to be transferred 3 times**:



- **Compilation of MPI+CUDA** programs: both MPI and CUDA have their own compiler drivers (mpicc and nvcc)
  - Suggestion: linking of CUDA code to other executables is quite simple
    - **compile** CUDA code with nvcc **to an object file with '-c'**
    - link the CUDA object file into the executable with mpicc: for **linking only the library libcudart is required** ('-lcudart')

Texture memory

- Resides in global memory
  - But read only
    - Note: with CUDA ≥ 3.1 and Fermi ‘surface memory’ allows read/write
  - And with **8 kB cache** per multiprocessor
- typical applications:
  - Heavily reused read only data with irregular access patterns
    - sometimes faster than Fermi L1 cache
    - Note: Access patterns unsuitable for L1 cache perform sometimes better with texture cache (example: pattern as in exercise 4 a)
- Can be configured to use optimized caching for 1-, 2- or 3-dimensional locality in access patterns

How can texture memory be used?

- A region of global memory can be bound as texture memory by using texture references:
- **Step 1:** declaring the texture reference

```
texture<Datatype, Dimension> texRef;
```

  Note: the parameter **`Dimension`** is optional
- Note: texture references (**`texRef`**) **need to be defined as plain global variables** (not: arrays, local variables) and cannot be passed as kernel arguments
- **Note:** texture/surface memory is **not implemented** in Cuda **Fortran**

How can texture memory be used?

- **Step 2:** telling the texture reference which array binding the array of global memory to the texture reference:
  - For 1D textures

```
cudaChannelFormatDesc channelDesc =
  cudaCreateChannelDesc<Datatype>();
cudaBindTexture( NULL, &texRef, memoryPtr,
&channelDesc, memorySize );
```

with

```
memorySize = sizeof(Datatype) * arraySize
```

- Note: the cudaChannelFormatDesc is a helper struct to tell the bind function which datatype
- Note: binding means an existing region of global memory can be accessed through the texture reference.
  The original data is not copied(!)

How can texture memory be used?

- for 2D-textures

```
cudaChannelFormatDesc channelDesc =
        cudaCreateChannelDesc<Datatype>();
cudaBindTexture2D( NULL, &texRef, memoryPtr,
       &channelDesc, arraySizeX, arraySizeY, pitch );
```

where **pitch** specifies the **offset in bytes to move from one 'line' to the next** in memory.
It is used to calculate the address of an element:

```
address = baseAddr +
          pitch*indexY + sizeof(Datatype)*indexX
```

- Note: the `pitch` **is required to be a multiple** of a certain page-size given by `cudaDeviceProp::texturePitchAlignment` (Kepler: 512 bytes).
  If your arraySizeX is not already aligned, padding must be added(!)
- If your arraySizeX is already aligned simply choose
  `pitch = sizeof(Datatype) * arraySizeX`

How can texture memory be used?

- **Step 3:** now the texture memory can be used from inside the kernel:

```
void __global__ myKernel() {
   ...
   ... = ... tex1D( tex_ref_1d, index ) ... ;
   ... = ... tex1Dfetch( tex_ref_1d, index ) ... ;
   ... = ... tex2D( tex_ref_2d, indexX, indexY ) ... ;
   ... = ... tex3D( tex_ref_3d,
                    indexX, indexY, indexZ ) ... ;
   ...
}
```

- Note: the functions **`tex1D`**, **`tex2D`**, **`tex3D`** take a 32-bit float value as index, **`tex1Dfetch`** takes a 32-bit int value as index.
  - for large 1D-arrays tex1Dfetch should be used to avoid rounding problems (!)

Advanced features of texture memory

- **Linear filtering:**
  - **It is possible to configure texture memory to do linear/bilinear/trilinear interpolation of array data**

Constant memory

- Resides in separate constant memory space
  - Read only
  - With 8 kB cache per multiprocessor
- Restrictions:
  - Not dynamically allocable
  - Maximum size is 64 kB per device

How can constant memory be used?

- Global variables or arrays are being declared to reside in constant memory by the **__constant__** declaration specifier:

```
__constant__ float data[size];
```

Accessing constant memory from GPU code

- Constant memory can be accessed normally from device code:

```
__constant__ float data[size];

void __global__ myKernel() {
   ...
   = ...data[...]...;
   ...
}
```

- **Note:** on Fermi/Kepler constant memory is also used implicitly to store kernel ‘function call’ arguments

Accessing constant memory from host code

- Constant memory cannot be accessed directly from host code, but copied:

```
cudaError_t cudaMemcpyToSymbol(const char* symbol,
const void* src, size_t count, size_t offset,
enum cudaMemcpyKind kind);

cudaError_t cudaMemcpyFromSymbol(void* dst,
const char* symbol, size_t count, size_t offset,
enum cudaMemcpyKind kind);
```

- Note: in host code constant memory variables or arrays are not variables or arrays but **symbols**
  - A symbol is a string identifier (type = const char*) known by the CUDA runtime
  - Do not try to read, write or use pointer arithmetic on constant memory symbols (!)

C++ classes

- For devices of compute capability ≥ 2.0 it is easily possible to use **non-polymorphic** C++ classes
  - non-polymorphic = no virtual functions

- Syntax:

```
class foo {

  int x,y,z;
  float *a,*b;

  __device__ foo(…) { // constructor as device function
    …
  }
  __device__ void f(…) { // method as device function
    …
  }
  …
};
```

- **Note:** kernels (`__global__`) functions may not be static class members
- **Note:** static class data members cannot be accessed from device code

Polymorphic C++ classes

- are possible with CC ≥ 2.0 but has an important restriction
- Objects allocated **in host code** can only be used **on host** !
- Objects allocated in **device code** can only be used **on device** !
  - Remark: use **`new`** and **`delete`** operators in device code
- Reason: virtual function table is different for host and device classes
- Note: STL containers do not work on device, as methods are not declared as device functions
  - Thrust template library (open source) can do some container operations (called from host, with data on GPU)

IEEE-exact arithmetic

- Default behavior:
  - rounding-mode: round-to-nearest-even
  - T10: SP denormals are flushed to zero,
    Fermi: SP denormals are kept
    (use **`-ftz=true`** for old behavior)
    - DP denormals are kept on both architectures
    - Note: denormal computations are full speed, contrary to x86!
  - Float additions and multiplications **are sometimes replaced by fused-multiply-adds (FMA)** by choice of the compiler!
    - This can be suppressed by replacing * and + with
      __fadd_rn(x,y), __dadd_rn(x,y)
      __fmul_rn(x,y), __dmul_rn(x,y)
      __fmaf_rn(x,y,z), __fma_rn(x,y,z)

IEEE-exact arithmetic

- Controlling rounding modes:
  - Can be controlled per function, suffix specifies rounding:
    _rn: round to nearest even
    _rz: round towards zero
    _ru: round upwards (towards positive infinity)
    _rd: round downwards (towards negative infinity)

  - Functions:
    __fadd_r?(x,y), **__dadd_r?(x,y)**: add **x+y**
    __fmul_r?(x,y), **__dmul_r?(x,y)**: multiply **x*y**
    __fmaf_r?(x,y,z), **__fma_r?(x,y,z)**: fused-multiply-add **x*y+z**
    __frcp_r?(x), **__drcp_r?(x)**: reciprocal **1/x**
    __fdiv_r?(x,y), **__ddiv_r?(x,y)**: division **x/y**
    __fsqrt_r?(x), **__dsqrt_r?(x)**: square root of **x**

- Note: see Appendix D in PG for more information on IEEE arithmetic functions

GPU Programming using CUDA
Exercise 12: Advanced features

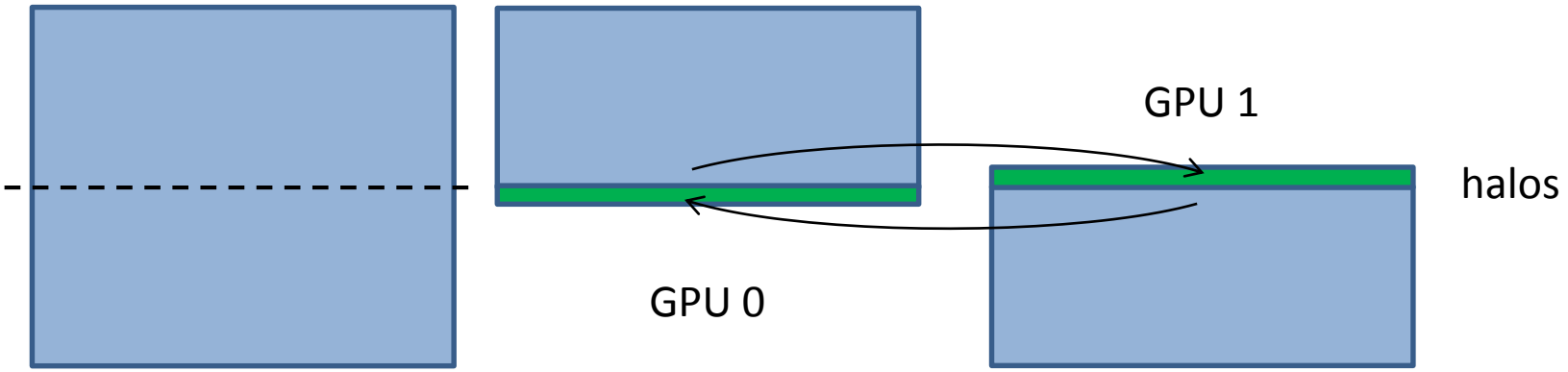
Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

Exercise 12: Using 2 GPUs using MPI

- Task:
  - Implement a 2D finite difference code on 2 GPUs
  - Algorithm:

$$f_{i,j}^{t+1} = f_{i,j}^{t} + \alpha \left(f_{i-1,j}^{t} + f_{i+1,j}^{t} + f_{i,j-1}^{t} + f_{i,j+1}^{t} - 4 f_{i,j}^{t} \right)$$

    - each field point requires the values of each its left, right, top, bottom neighbor
  - Parallelization technique: domain decomposition



first/last calculated line is copied to other GPU after each time step

Exercise 12: Using 2 GPUs using MPI

- Single-GPU CUDA kernel for heat equation in 2D:

```
void __global__ diffusion(int sizeX,int sizeY,
                          float* input,float* output) {
  int myX=blockIdx.x*blockSizeX+threadIdx.x+1;
  int myY=blockIdx.y*blockSizeY+threadIdx.y+1;
  if (myX>=sizeX-1 || myY>=sizeY-1)
    return;
  int index=myY*sizeX+myX;

  output[index] = input[index]*(1.-4.*alpha) +
    alpha*(input[index-1] + input[index+1] +
           input[index-sizeX] + input[index+sizeX]);

}
```

Exercise 12: Using 2 GPUs with MPI (C)

- The exercise12_mpi_template.cu contains the distribution of the field to two processes with one GPU each.
  Only the host/device copy of boundary is missing.
- **Note:** Use GNU compiler environment for Cray wrappers
  - module swap PrgEnv-cray PrgEnv-gnu
- **Note:** To compile use CUDA compiler for .cu files and Cray MPI compiler wrapper for host code and linking
  - nvcc -c -arch=compute_35 exercise12_mpi_template.cu
  - cc exercise12_mpi_main.c exercise12_mpi_template.o
- **Note:** To access one GPU with two MPI processes concurrently
  - export CRAY_CUDA_PROXY=1
  - aprun -n 2 ./a.out

Exercise 12: Using 2 GPUs with MPI (Fortran)

- The exercise12_mpi_template.f90 contains the distribution of the field to two processes with one GPU each.
  Only the host/device copy of boundary is missing.

- **Note:** To compile use Cray MPI compiler wrapper
  - ftn -Mcuda,cc35,cuda5.5 exercise12_mpi_template.f90

- **Note:** To access one GPU with two MPI processes concurrently
  - export CRAY_CUDA_PROXY=1
  - aprun -n 2 ./a.out

GPU Programming using CUDA
CUDA Libraries

Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

Overview

- NVIDIA provides with CUDA 5 important libraries for
  - simple dense linear algebra (CUBLAS)
  - sparse linear algebra (CUSPARSE)
  - Fast Fourier Transform (CUFFT)
  - pseudorandom number generation (CURAND)
- Mode of operation:
  - GPU memory allocation and copy is done manually by user with the normal CUDA API (cudaMalloc(), cudaMemcpy())
  - Library functions are called from host and get passed pointers to global device memory
  - Exception: CURAND contains device functions to generate random numbers from GPU code

API initialization (CUBLAS and CUSPARSE)

- All libraries require certain data structures to be initialized before they can be used
- CUBLAS and CUSPARSE require handles to be created for use with all following library calls

- CUBLAS:

```
cublasHandle_t cublasHandle;
cublasStatus_t status = cublasCreate(&cublasHandle);
…
cublasDestroy(cublasHandle);
```

- CUSPARSE:

```
cusparseHandle_t cusparseHandle;
cusparseStatus_t status =
                          cusparseCreate(&cusparseHandle);
…
cusparseDestroy(cusparseHandle);
```

API initialization (CUFFT)

- CUFFT: a plan needs to be created, which contains the data needed for doing FFTs of a certain dimension, size and type:

```
cufftResult
cufftPlan1d(cufftHandle* plan, int size, cufftType type, int batch);
cufftResult
cufftPlan2d(cufftHandle* plan, int sizeX, int sizeY, cufftType type);
cufftResult
cufftPlan3d(cufftHandle* plan, int sizeX, int sizeY, int sizeZ,
            cufftType type);
```

- Type may be:
  - **`CUFFT_R2C`** = Real to complex (interleaved)
  - **`CUFFT_C2R`** = Complex (interleaved) to real
  - **`CUFFT_C2C`** = Complex to complex, interleaved
  - **`CUFFT_D2Z`** = Double to double-complex
  - **`CUFFT_Z2D`** = Double-complex to double
  - **`CUFFT_Z2Z`** = Double-complex to double-complex
- **`batch`** allows to do multiple FFTs with one call (using consecutive array elements)

- Plans have to be released with the **`cufftDestroy()`** function:

```
cufftResult cufftDestroy(cufftHandle plan);
```

API initialization (CURAND)

- CURAND host API: a ‘generator’ needs to be created:

```
curandStatus_t curandCreateGenerator
          (curandGenerator_t* generator, curandRngType_t type)
```

- Type specifies the used method of random number generation. Possible values are:

CURAND_RNG_PSEUDO_DEFAULT
CURAND_RNG_PSEUDO_XORWOW
CURAND_RNG_QUASI_DEFAULT
CURAND_RNG_QUASI_SOBOL32
CURAND_RNG_QUASI_SCRAMBLED_SOBOL32
CURAND_RNG_QUASI_SOBOL64
CURAND_RNG_QUASI_SCRAMBLED_SOBOL64

- Generators need to be freed with **`curandDestroyGenerator()`**:

```
curandStatus_t
curandDestroyGenerator(curandGenerator_t generator)
```

- CURAND device API: **`curand_init(…,&state)`** is used to initialize the generator state. No destroy is necessary with device API.
State can be different data structures, depending on generation mechanism.

Streams

- The libraries allow setting the CUDA stream for a handle:

```
cufftResult
cufftSetStream( cufftHandle plan, cudaStream_t stream )

cublasStatus_t
cublasSetStream(cublasHandle_t handle, cudaStream_t streamId)

cusparseStatus_t
cusparseSetKernelStream(cusparseHandle_t handle,
                        cudaStream_t streamId )

curandStatus_t
curandSetStream (curandGenerator_t generator,
                 cudaStream_t stream)
```

- **Note:** if many small kernels are executed, e.g. with CUBLAS, setting streams allows overlapping concurrent execution with Fermi devices

CUBLAS functions

- BLAS standard is completely implemented
  - Level 1: vector operations
  - Level 2: matrix-vector operations
  - Level 3: matrix-matrix operations
- Naming conventions:
  all functions for mathematical operations are of the form

```
cublasStatus_t cublas<T><operation>(cublasHandle_t handle, ...)
```

  - A single letter ‘T’ is used for the datatype
    (S=float, D=double, C=complex float, Z=complex double)
  - Example: compute 2-norm of vector (single or double precision)

```
cublasStatus_t cublasSnrm2(cublasHandle_t handle, int n,
                           const float* x, int incX, float* result)
cublasStatus_t cublasDnrm2(cublasHandle_t handle, int n,
                           const double* x,
                           int incX, double* result)
```

  - Note: most functions have stride (increment) specifications of array elements (here: **incX**) for strided storage of vectors

CUBLAS functions

- 2-norm of vector

$$\sqrt{\sum_k x_k^2}$$

```
cublasStatus_t cublas<T>nrm2(cublasHandle_t handle, int n,
                             const T* x, int incX, T* result)
```

- Sum of absolute values of vector

$$\sum_k \left| x_k \right|$$

```
cublasStatus_t cublas<T>asum(cublasHandle_t handle, int n,
                             const T* x, int incX, float *result)
```

- Scalar product

$$\sum_k x_k y_k$$

```
cublasStatus_t cublas<T>dot (cublasHandle_t handle, int n,
                             const T* x, int incX,
                             const T* y, int incY,
                             T* result)
```

CUBLAS functions

- Vector addition with scaling

$$\underline{y} = \alpha \underline{x} + \underline{y}$$

```
cublasStatus_t cublas<T>axpy(cublasHandle_t handle, int n,
                             const T* alpha,
                             const T* x, int incX,
                             T* y, int incY)
```

- Matrix-vector multiplication

$$\underline{y} = \alpha \underline{\underline{A}} \underline{x} + \beta \underline{y}$$

```
cublasStatus_t cublas<T>gemv(cublasHandle_t handle,
                             cublasOperation_t transpose,
                             int m, int n,
                             const T* alpha,
                             const T* A, int pitchA,
                             const T* x, int incX,
                             const T* beta,
                             T* y, int incy)
```

CUBLAS functions

- Matrix-matrix multiplication

$$\underline{\underline{C}} = \alpha \underline{\underline{A}}\,\underline{\underline{B}} + \beta \underline{\underline{C}}$$

```
cublasStatus_t
cublas<T>gemm(cublasHandle_t handle,
              cublasOperation_t transposeA,
              cublasOperation_t transposeB,
              int m, int n, int k,
              const T* alpha,
              const T* A, int pitchA,
              const T* B, int pitchB,
              const T* beta,
              T* C, int pitchC)
```

CUSPARSE csr storage format

- Sparse matrix is stored in compressed sparse row (CSR) format
- Matrix A =

	1		4	5
2	7			
		3		6

- Nonzero elements are stored row-wise in 1D-array

matrixAValues =

1	4	5	2	7	3	6

- Column index array stores positions in row in original matrix

matrixAColumnIndices =

2	4	5	1	2	3	5

First row NZ elements are in 2nd, 4th, 5th column

- Row pointer array stores beginning of original matrix rows in matrixAValues and matrixAColumnIndices

matrixARowPointers =

1	4	6	8

2nd row elements start at 4th entry, 3rd row elements start at 6th entry

CUSPARSE functions

- Sparse matrix-vector multiplication with matrix in compressed sparse row format

$$\underline{y} = \alpha \underline{\underline{A}}\,\underline{x} + \beta \underline{y}$$

```
cusparseStatus_t
cusparse<T>csrmv(cusparseHandle_t handle,
                 cusparseOperation_t transpose,
                 int m, int n, int nnz, T* alpha,
                 const cusparseMatDescr_t descrA,
                 const T* matrixAValues,
                 const int* matrixARowPointer,
                 const int* matrixAColumnIndices,
                 const T* x, T* beta, T* y)
```

CUSPARSE matrix description

- CUSPARSE matrices need a matrix description data structure

```
typedef struct {
    cusparseMatrixType_t MatrixType;
    cusparseFillMode_t   FillMode;
    cusparseDiagType_t   DiagType;
    cusparseIndexBase_t  IndexBase;
} cusparseMatDescr_t;
```

- The data structure should be initialized with

```
cusparseStatus_t
cusparseCreateMatDescr( cusparseMatDescr_t *descrA )
```

  and released with

```
cusparseStatus_t
cusparseDestroyMatDescr( cusparseMatDescr_t descrA )
```

- The fields of **`cusparseMatDescr_t`** specify details of the matrix storage
  - compact symmetric, hermitian, triangular matrix storage
  - 0-based or 1-based indexing
- **Note:** **`cusparseCreateMatDescr()`** initializes the description to unsymmetrical matrix with 0-based indexing

CUFFT functions

- A Fourier Transform can be executed according to the created plan by the functions

```
cufftResult
cufftExecC2C(cufftHandle plan, cufftComplex *idata,
             cufftComplex *odata, int direction );

cufftResult
cufftExecR2C(cufftHandle plan, cufftReal *idata,
             cufftComplex *odata );

cufftResult
cufftExecC2R(cufftHandle plan, cufftComplex *idata,
             cufftReal *odata );

cufftResult
cufftExecZ2Z(cufftHandle plan, cufftDoubleComplex *idata,
             cufftDoubleComplex *odata, int direction );
```

- The direction parameter specified forward or reverse Fourier transform. It may be
  - CUFFT_FORWARD
  - CUFFT_INVERSE

CURAND host functions

- CURAND host functions can generate arrays of random numbers in global device memory
  - 32 and 64 bit signed integers and floats and doubles are supported
- Integers:

```
curandStatus_t
curandGenerate(curandGenerator_t generator,
               unsigned int * outputPtr, size_t num);
curandStatus_t
curandGenerateLongLong
        (curandGenerator_t generator,
         unsigned long long * outputPtr, size_t num);
```

  - Produces random numbers over the full range of the integer type (all bits are random)

- Floating point:

```
curandStatus_t
curandGenerateUniform(curandGenerator_t generator,
                      float* outputPtr, size_t num)
curandStatus_t
curandGenerateUniformDouble(curandGenerator_t generator,
                            double* outputPtr, size_t num)
```

  - Produces random numbers in (0,1]

CURAND host functions

- For floating point normal and log-normal distributions can also be generated:

```
curandStatus_t
curandGenerateLogNormal(curandGenerator_t generator,
                        float* outputPtr, size_t n,
                        float mean, float stddev);
curandStatus_t
curandGenerateLogNormalDouble(curandGenerator_t generator,
                              double* outputPtr, size_t n,
                              double mean, double stddev);
curandStatus_t
curandGenerateNormal(curandGenerator_t generator,
                     float* outputPtr, size_t n,
                     float mean, float stddev);
curandStatus_t
curandGenerateNormalDouble(curandGenerator_t generator,
                           double* outputPtr, size_t n,
                           double mean, double stddev);
```

CURAND device functions

- From device code single random numbers can be generated with a device function call:

```
__device__ unsigned int curand (…* state);
__device__ unsigned long long curand (…* state);
__device__ float curand_uniform (…* state);
__device__ double curand_uniform_double (…* state);
__device__ float curand_normal (…* state);
__device__ double curand_normal_double (…* state);
__device__ float curand_log_normal (…* state,
                                    float mean, float stddev);
__device__ double curand_log_normal_double (…* state,
                                   double mean, double stddev);
```

- **Note:** when generating random numbers in parallel in multiple threads, each thread should have a different state data structure!

GPU Programming using CUDA

Exercises 13: Conjugate Gradient with CUBLAS and CUSPARSE

Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

Exercise 13: Conjugate Gradient

- Conjugate Gradient is an iterative solver for symmetric positive definite matrices. It is typically used to solve sparse linear equation systems

$$\underline{\underline{A}}\,\underline{x} = \underline{b} \qquad \underline{\underline{A}}^{T} = \underline{\underline{A}} \qquad \underline{\underline{A}} > 0$$

- The only matrix or vector operations it needs is:
  - copy of vectors
  - sparse matrix-vector-multiplication
  - scaling of vectors (vector-scalar-multiplication), addition of vectors and combinations thereof (saxpy, daxpy)
  - scalar product, 2-norm of vector
- exercise13_cpu.c contains a CPU implementation already using the csr data storage format of CUSPARSE

Exercise 13: Conjugate Gradient

- Task:
  - in exercise13_template.cu
    the needed initializations of CUBLAS, CUSPARSE,
    GPU memory allocation and copy of A, x and b
    are already done
  - replace the CPU code in the matrix/vector operation helper functions with calls to CUBLAS or CUSPARSE
    - csrMatrixVectorMultiplication
    - axpy
    - scaleVector
    - scalarProduct
    - vector2NormSquare
- Note: the template file in the initial state will crash,
  because above functions already get passed GPU pointers,
  but still contain the CPU code from exercise13_cpu.c
- Advanced exercise: check in profiler which kernels take the most time.
  Can you gain performance by replacing some library calls
  with your own kernels?

Exercise 13: Conjugate Gradient

- Algorithm:

$\underline{r} = \underline{b} - \underline{\underline{A}}\,\underline{x}$ — sparse matrix vector multiplication

$\underline{p} = \underline{r}$ — vector copy

$\alpha_{old} = \left|\underline{r}\right|^2$ — vector 2-norm

repeat until α is small enough {

$\underline{v} = \underline{\underline{A}}\,\underline{p}$ — sparse matrix vector multiplication

$\lambda = \dfrac{\alpha_{old}}{\left\langle p \middle| v \right\rangle}$ — scalar product

$\underline{x} = \underline{x} + \lambda\,\underline{p} \qquad \underline{r} = \underline{r} - \lambda\,\underline{v}$ — saxpy/daxpy

$\alpha = \left|\underline{r}\right|^2$ — vector 2-norm

$\underline{p} = \underline{r} + \dfrac{\alpha}{\alpha_{old}}\,\underline{p}$ — vector scaling/addition

$\alpha_{old} = \alpha$

}

Exercise 13: Conjugate Gradient

- **Notes**
  - For the sparse matrix vector multiplication the functions **`cusparseScsrmv()`** (single precision) and **`cusparseDcsrmv()`** (double precision) can be used
    - The matrix is already in the correct csr format, no conversion is necessary
  - For scalar product and norm **`cublasSdot()`**, **`cublasDdot()`**, **`cublasSnrm2()`**, **`cublasDnrm2()`** can be used
    - Warning: **`cublasSnrm2()`** has a bug.
      It returns NAN when the vector contains denormalized numbers.
      Better use **`cublasSdot()`** for vector 2-norm!
  - For vector addition and scaling **`cublasSscal()`**, **`cublasDscal()`**, **`cublasSaxpy()`**, **`cublasDaxpy()`** can be used
  - The files **`cublas_utils.h`** and **`cusparse_utils.h`** contain the functions **`cublasVerify()`** and **`cusparseVerify()`** for simplified error handling.
  - Link executable with **`-lcublas -lcusparse`**

GPU Programming using CUDA
Introduction to OpenACC

Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

Introduction to OpenACC

- Accelerator programming model
  - Device code is written like (serial) CPU code
  - Data is not *explicitly* copied to GPU (looks like working on CPU data structures)
  - Compiler tries automatic parallelization (loop vectorization, etc.)
  - Intermediate CUDA C or OpenCL code (CAPS, PGI) or PTX assembly (Cray) is generated

- Simple example:

```
void VectorAdd(unsigned int size,
               float* a, float* b, float* c) {
  #pragma acc parallel loop
  for(int i=0;i<size;i++) {
    c[i]=a[i]+b[i];
  }
}
```

Parallel construct

- What is the parallel construct?
  - Requests that code is to be executed on accelerator device
    - C: next statement or ‘{ }’-block
    - Fortran: region between ‘parallel’ and ‘end parallel’
  - Without further pragmas code is to be executed in serial on device
  - ‘manual’ mode of parallelization
    - parallelization of code has to be specified explicitly
    - No attempt is made, e.g. to find parallelizable loops

Parallel construct

- Syntax:
  - C:

```
#pragma acc parallel
{
   ...
}
```

  - Fortran:

```
!$acc parallel
...
!$acc end parallel
```

Kernels construct

- What is the kernels construct?
  - Requests that code is to be executed on accelerator device
    - C: next statement or ‘{ }’-block
    - Fortran: region between ‘kernel’ and ‘end kernel’
  - ‘automatic’ mode of parallelization
    - Compiler tries to find parallelizable code (loops, etc.) and automatically distributes work to CUDA blocks/threads
    - ‘kernels’ → multiple kernels may be generated (by compiler decision)
    - But further pragmas are still possible (to guide compiler)

Kernels construct

- Syntax:
  - C:

```
#pragma acc kernels
{
   ...
}
```

  - Fortran:

```
!$acc kernels
...
!$acc end kernels
```

Loop construct

- Within parallel construct:
- Tells the compiler that loop iterations are independent and shall be executed in parallel

C:

```
#pragma acc parallel
{
  #pragma acc loop
  for(int i=0;i<size;i++) {
    c[i]=a[i]+b[i];
  }
}
```

Fortran:

```
!$acc parallel
!$acc loop
do i=1,size
  c(i)=a(i)+b(i)
end do
!$acc end loop
!$acc end parallel
```

Combined parallel/loop construct

- A parallel region which consists wholly of a single loop can be specified directly using a combined directive:

C:

```
#pragma acc parallel loop
for(int i=0;i<size;i++) {
  c[i]=a[i]+b[i];
}
```

Fortran:

```
!$acc parallel loop
do i=1,size
  c(i)=a(i)+b(i)
end do
!$acc end parallel loop
```

Independent clause within kernels region

- On a **loop construct** within a **kernels** region
  - Tells the compiler that **loop iterations** are **independent**
    (**only a hint**, for the case compiler cannot proof this independence by itself)
  - Compiler still decides about parallelization

C:

```
#pragma acc kernels
{
  #pragma acc loop independent
  for(int i=0;i<size;i++) {
    c[i]=a[i]+b[i];
  }
}
```

Fortran:

```
!$acc kernels
!$acc loop independent
do i=1,size
  c(i)=a(i)+b(i)
end do
!$acc end loop
!$acc end kernels
```

gang, worker and vector clauses

- On a loop construct it is possible to specify over which hardware entities parallelization should happen
  - gang = group of workers
    ```
    #pragma acc loop gang
    #pragma acc loop gang(n)
    ```
  - worker = single ‘thread’, but might contain SIMD vector unit
    ```
    #pragma acc loop worker
    #pragma acc loop worker(n)
    ```
  - vector clause: do SIMD vectorization of loop
    ```
    #pragma acc loop vector
    #pragma acc loop vector(n)
    ```
  - In CUDA terms
    - Gang = GPU / Grid (Group of multiprocessors)
    - Worker (Vector?) = Multiprocessor / Block (Group of threads)

num_gangs, num_workers and vector_length

- On a **parallel of kernels construct** it is possible to specify the ‘execution configuration’ of the called kernel(s)
- Syntax

```
#pragma parallel num_gangs(n) num_workers(n) \
                 vector_length(n)
```

- Note: this is an imported feature for performance tuning, as should be clear from CUDA tests
- Note: mapping of these clauses to grid size/block size may be implementation-specific

Data management

- How does **data copy from/to device** work?

```
#pragma acc parallel loop
for(int i=0;i<size;i++) {
  c[i]=a[i]+b[i];
}
```

  - Per default **all arrays** a[], b[], c[] are **copied to device before** loop is executed and **back to host afterwards**
    - Compiler might detect need to copy, but at the moment it is rarely the case

- Simple method to configure copy operations: **`copyin`**/**`copyout`** clauses

```
#pragma acc parallel copyin(a[0:size],b[0:size]) \
                     copyout(c[0:size])
{
  ...
}
```

- Problem: data is **still copied** to/from host **between** parallel/kernels **regions**

Data regions

- Code within data region is executed on host
  (outside of parallel or kernels regions)
- The specified variables / arrays are copied from/to device
  only at begin/end of data region
  (even if the data regions contains many parallel/kernel regions)
- ```
  #pragma acc data copyin(a[0:size],b[0:size]) \
                   copyout(c[0:size])
  {
    ...
  }
  ```
- '**copyin**' clause
  - Meaning data is copied at start of data region to device
- '**copyout**' clause
  - Meaning data is copied at end of data region to host
- '**copy**' clause
  - Meaning data is copied at start of data region to device
    and at end of data region back to host
  ```

Array size specification

- In C with ‘pointer as array’ or Fortran with ‘(*)’-syntax compiler does not know array size
- In these cases size specification is required on data clauses
- Syntax
  - C:

```
#pragma acc data copyin(a[0:size],b[0:size]) \
                 copyout(c[0:size])
{
  ...
}
```

    - Note: it is `[begin:size]` in C, but `(begin:end)` in Fortran (!!!)

  - Fortran:

```
!$acc data copyin(a(1:size),b(1:size)) &
           copyout(c(1:size))
...
!$acc end data
```

Further data clauses

- '**`create`**' clause
  - Memory is allocated on device, but no copy is done at all
- '**`present`**' clause
  - Data is already present on device
- '**`present_or_copyin`**', '**`present_or_copy`**' clauses
  - Framework checks if data is already present on device, if not it is copied
  - '**`present_or_copy`**' also copies data back at end of data region
- '**`present_or_create`**', '**`present_or_copyout`**',
  - Framework checks if data is already present on device, if not it is allocated
  - '**`present_or_copyout`**' also copies data back at end of data region
- '**`deviceptr`**' clause
  - array is a pointer to device memory (created with some other allocation mechanism)

Finer control: update directive

- Variables which are already present on device (but may have outdated data) can be manually copied from/to device
- **`#pragma acc update host(variable list) device(variable list)`**
  - **`host`**: variables are copied from device to host
  - **`device`**: variables are copied from host to device

Note on variable scoping

- Are variables in device code local per thread or globally visible?
- Default behavior:
  - Arrays are globally visible
  - Variables appearing in a data clause are globally visible
  - Variables accessed both inside and outside of a parallel/kernels region are globally visible
  - Variables accessed only within one parallel/kernels region are thread local
- Explicit control: **`private`** clause (on parallel, kernels, loop)
  - variable is private for each thread (or iteration for loop clause)

Asynchronous execution

- Per default the host is expected at parallel/kernels regions
- Asynchronous execution ('streams') is possible, though
- Mechanism
  - Add an async clause to parallel of kernels construct

```
#pragma acc parallel ... async
{
   ...
}
#pragma acc parallel ... async(n)
{
  ...
}
```

  - To wait for asynchronous regions the wait directive is used
    - wait on all asynchronous regions:

```
#pragma acc wait
```

    - Wait only on regions with the same integer index `n` in the async clause:

```
#pragma acc wait(n)
```

Runtime library

- Beside pragmas there is a runtime library containing several useful functions
  - Include with
    C: **`#include <openacc.h>`**
    Fortran: **`use openacc`**
- Functions, e.g.:
  - Select between multiple devices:
    `acc_set_device_num()`
  - Test for finished asynchronous regions, without blocking:
    `acc_async_test()`
  - Manual device memory management
    (to be used with **`deviceptr`**):
    `acc_malloc()`, **`acc_free()`**

More features

- Private
- Reduction on loops

GPU Programming using CUDA
Exercises 14+15: OpenACC

Thomas Baumann, Oliver Mangold, Mhd. Amer Wafai

Exercise 14: Vector operation with OpenACC

- Task:
  - Create a GPU version of the vector-scalar multiplication CPU code exercise01_cpu.c/.f90
    by adding the necessary ‘acc’ pragmas
  - Check with COMPUTE_PROFILE=1
    that the code is actually executed on the GPU.
  - How many memory copies are done?
- Advanced exercise
  - Create an ‘acc data’ region around your ‘acc parallel’ region and modify the data clauses so that the arrays are copied when entering/leaving the data region instead of the parallel region
  - Add timers around the ‘acc parallel’ region
    so you can measure the duration of the kernel
    without the duration of the array copies between host and device
  - Note: do not forget async/wait (with the Cray compiler)

Exercise 15: 2D Heat equation with OpenACC

- Task:
  - exercise15_template.cl/.f90 contains the CPU version of the 2D finite difference heat equation scheme from exercise 12
  - Move the computation on the GPU with acc pragmas
  - Make sure no data copy from/to host is happening between the multiple kernel calls (no copy inside the other loop over the time steps i=0,steps)
- Notes:
  - You can check if accelerator code is generated by using the Cray compiler option '**`-h list=m`**' and looking at the '.lst' file
  - The field data is copied from array a to b, then back from b to a in a loop. Which data clauses do you actually need to save as many unnecessary memory copies as possible?

Timers with Cray Fortran Compiler

- The subroutine **`cpu_time()`** of the Cray compiler is not suitable for timing GPU code, as it has second resolution
- Instead there is an alternative **`cpu_time_cray()`** available
- To use it you have to compile **`cray_fortran_timer.c`** and **`timer.f90`** and link it to your code:

```
cc -c cray_fortran_timer.c
ftn -c timer.f90
ftn -h pragma=acc <source file> \
  cray_fortran_timer.o timer.o  \
  -o <executable>
```

Thank you!

- Thank you for listening!