Praktische Aspekte
der Informatik

Moritz Mühlhausen
Prof. Marcus Magnor
GPGPU-Programming

Cuda
Warning!
The following slides are meant to give you a very superficial introduction.

If you want to learn more, have a look at:
https://docs.nvidia.com/cuda/
Introduction

Programming Model

GPU Memory

Synchronization

NVCC
Introduction – Why GPU?

Benefits

• Highly parallel, multithreaded, manycore processors
• very high memory bandwidth
• well-suited for data-parallel computations
• waste less resources

Drawbacks

• memory synchronization between CPU and GPU
• synchronization between GPU threads
• extra compiler
• only for NVIDIA GPUs (Cuda)
Introduction – Why GPU?

Theoretical GFLOP/s at base clock

- NVIDIA GPU Single Precision
- NVIDIA GPU Double Precision
- Intel CPU Single Precision
- Intel CPU Double Precision

https://graphics.tu-bs.de/teaching/ss19/padi/
Introduction – Why GPU?

Theoretical Peak GB/s

- GeForce GPU
- Tesla GPU
- Intel CPU

https://graphics.tu-bs.de/teaching/ss19/padi/
## Introduction – Why GPU?

- **Element-wise sum of two vectors/matrices**

<table>
<thead>
<tr>
<th>A</th>
<th>+</th>
<th>B</th>
<th>=</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Introduction – Why GPU?

- Element-wise sum of two vectors/matrices
  - Naive implementation: simple for-loop over all indices

```
A
2 4 ...
1 7 ...
8 2 ...
2 7

B
5 1 ...
1 8 ...
1 2 ...

C
...
...
...
```

https://graphics.tu-bs.de/teaching/ss19/padi/
Introduction – Why GPU?

- **Element-wise sum of two vectors/matrices**
  - Naive implementation: simple for-loop over all indices

```
A
2 4 ...
1 7 ...
8 2 ...
2

B
5 1 ...
8 ...
1 7 ...

C
7 ...
= ...
```

https://graphics.tu-bs.de/teaching/ss19/padi/
Introduction – Why GPU?

• Element-wise sum of two vectors/matrices
  - Naive implementation: simple for-loop over all indices
Introduction – Why GPU?

- Element-wise sum of two vectors/matrices
  - Naive implementation: simple for-loop over all indices
Introduction – Why GPU?

- Element-wise sum of two vectors/matrices
  - Naive implementation: simple for-loop over all indices
  - Better: several threads iterate over all indices

```
\[
\begin{array}{cccc}
A & + & B & = \\
2 & 4 & 1 & 8 \\
... & ... & ... & ...
1 & 7 & 2 & 1 \\
8 & 2 & 1 & 7 \\
2 & & & \\
\end{array}
\]
```
Introduction – Why GPU?

- Element-wise sum of two vectors/matrices
  - Naive implementation: simple for-loop over all indices
  - Better: several threads iterate over all indices

\[
\begin{align*}
A &= \begin{pmatrix}
2 \\
4 \\
\ldots \\
1 \\
7 \\
\ldots \\
8 \\
2
\end{pmatrix} \\
B &= \begin{pmatrix}
5 \\
1 \\
\ldots \\
8 \\
2 \\
\ldots \\
1 \\
7
\end{pmatrix} \\
C &= \begin{pmatrix}
7 \\
\ldots \\
9 \\
\ldots \\
9 \\
\ldots \\
\ldots \\
\ldots
\end{pmatrix}
\end{align*}
\]
Introduction – Why GPU?

- Element-wise sum of two vectors/matrices
  - Naive implementation: simple for-loop over all indices
  - Better: several threads iterate over all indices

\[ A \begin{bmatrix} 2 \\ 4 \\ \vdots \\ 1 \\ 7 \\ \vdots \\ 8 \\ 2 \\ 2 \end{bmatrix} + B \begin{bmatrix} 5 \\ 1 \\ \vdots \\ 8 \\ 2 \\ \vdots \\ 1 \\ 7 \end{bmatrix} = C \begin{bmatrix} 7 \\ 5 \\ \vdots \\ 9 \\ 9 \\ \vdots \\ 9 \\ 9 \end{bmatrix} \]
Introduction – Why GPU?

- Element-wise sum of two vectors/matrices
  - Naive implementation: simple for-loop over all indices
  - Better: several threads iterate over all indices
  - GPU can handle more threads concurrently!
Introduction – Why GPU?

- Element-wise sum of two vectors/matrices
  - Naive implementation: simple for-loop over all indices
  - Better: several threads iterate over all indices
  - GPU can handle more threads concurrently!

1. allocate GPU memory

```
A  B  C
2  5  
4  1  
... ...
1  8  
7  2  
... ...
8  1  
2  7  

A' B' C'
?  ?  ?
?  ?  ?
... ...
?  ?  ?
?  ?  ?
... ...
?  ?  ?
?  ?  ?
```
Introduction – Why GPU?

- Element-wise sum of two vectors/matrices
  - Naive implementation: simple for-loop over all indices
  - Better: several threads iterate over all indices
  - GPU can handle more threads concurrently!

A
\[
\begin{array}{c}
2 \\
4 \\
... \\
1 \\
7 \\
... \\
8 \\
2 \\
\end{array}
\]

B
\[
\begin{array}{c}
5 \\
1 \\
... \\
8 \\
2 \\
... \\
1 \\
7 \\
\end{array}
\]

C
\[
\begin{array}{c}
... \\
... \\
... \\
... \\
... \\
\end{array}
\]

A′
\[
\begin{array}{c}
2 \\
4 \\
... \\
1 \\
7 \\
... \\
8 \\
2 \\
\end{array}
\]

B′
\[
\begin{array}{c}
5 \\
1 \\
... \\
8 \\
2 \\
... \\
1 \\
7 \\
\end{array}
\]

C′
\[
\begin{array}{c}
? \\
? \\
... \\
? \\
? \\
... \\
? \\
? \\
\end{array}
\]

1. allocate GPU memory
2. copy A and B values

https://graphics.tu-bs.de/teaching/ss19/padi/
Introduction – Why GPU?

• Element-wise sum of two vectors/matrices
  ▪ Naive implementation: simple for-loop over all indices
  ▪ Better: several threads iterate over all indices
  ▪ GPU can handle more threads concurrently!

A

B

C

A'  B'  C'

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

1. allocate GPU memory
2. copy A and B values
3. Execute adding kernel

https://graphics.tu-bs.de/teaching/ss19/padi/
Element-wise sum of two vectors/matrices

- Naive implementation: simple for-loop over all indices
- Better: several threads iterate over all indices
- GPU can handle more threads concurrently!
1. allocate GPU memory
2. copy A and B values
3. Execute adding kernel
4. copy C from GPU
Introduction

Programming Model

GPU Memory

Synchronization

NVCC
### GPGPU Overview

1. **allocate GPU memory**

2. **copy A and B values**

3. **Execute adding kernel**

4. **copy C from GPU**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A'</td>
<td>B'</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>9</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>9</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>9</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

### Steps:

1. **allocate GPU memory**

2. **copy A and B values**

3. **Execute adding kernel**

4. **copy C from GPU**

https://graphics.tu-bs.de/teaching/ss19/padi/
• CPU callable function to start computations on the GPU

• executed N times in parallel by N different CUDA threads

• Defined using __global__ declaration specifier

• Number of CUDA threads are specified using a new <<<...>>> execution configuration

• Asynchronous from CPU code
Parallel element-wise sum of two matrices

```c
// Kernel definition
__global__ void MatAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    MatAdd<<<1, N>>>(A, B, C);
    ...
}
```

__global__ specifier defines a kernel function, only `void` return.

MatAdd<<<1, N>>>(A, B, C) calls the kernel function with `N` threads.

`threadIdx.x` specifies the id of the thread. This way each thread is handling a different matrix element.

**Kernels** can only return `void`, but they can alter global GPU memory.
Parallel element-wise sum of two matrices

```c
// Kernel definition
__global__ void MatAdd(float* A, float* B, float* C, int N)
{
    int i = threadIdx.x;
    int j = threadIdx.y;
}

int main()
{
    ...
}

    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
```

__global__ specifier defines a kernel function, only void return.

MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C) calls the kernel function with N * N threads.

threadIdx.x & .y specifies the id of the thread. This way each thread is handling a different matrix element.

**Kernels** can only return void, but they can alter global GPU memory.

**Threads** can be identified by a one-, two- or three-dimensional threadIdx.

- Convenient for computations for a vector, matrix or volume
Programming Model – Threads

GPGPU Overview

1. allocate GPU memory
2. copy A and B values
3. Execute adding kernel
4. copy C from GPU

A: 2 4 1 7 8
    ... 1
    8
B: 5 1 8 2 1
    ... 9
    9
C: 7 5 9 9

A': 2 4 1 7 8
    ... 1
    8
B': 5 1 8 2 1
    ... 9
    9
C': 7

Thread 1

Thread N
Parallel element-wise sum of two matrices

// Kernel invocation with one block of N * N * 1 threads
int numBlocks = 1;
dim3 threadsPerBlock(N, N);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);

Number of threads is limited.
• reside on the same processor core
• share limited memory resources
• depending on the GPU up to 1024 threads

Kernel can be executed by multiple equally-shaped thread blocks.
• organized in one-, two- or three-dimensional grid (dim3 or int)
• required to execute independently
1. allocate GPU memory
2. copy A and B values
3. Execute adding kernel
4. copy C from GPU
Parallel element-wise sum of two matrices

```c
// Kernel definition
__global__ void MatAdd(float* A, float* B, float* C, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

}

int main()
{
    ...

    // Kernel invocation with N * N * 1 blocks of N * N * 1 threads
    dim3 numBlocks(N, N);
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

**MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C)** calls the kernel function in **N * N** blocks with **N * N** threads.

**blockIdx.x & .y** specifies the id of the block.

**blockDim.x & .y** holds the number of threads in the respective dimension.

**Block** can be identified by a one-, two- or three-dimensional blockIdx.
Parallel element-wise sum of two matrices

```c
// Kernel definition
__global__ void MatAdd(float* A, float* B, float* C, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

}

int main()
{
    ...

    // Kernel invocation with N * N * 1 blocks of N * N * 1 threads
    dim3 numBlocks(N, N);
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C, N);
    ...
}
```

MatAdd\(\text{<<<numBlocks, threadsPerBlock>>}>>(\text{A, B, C})\) calls the kernel function in \(N \times N\) blocks with \(N \times N\) threads.

`blockIdx.x & .y` specifies the id of the block.

`blockDim.x & .y` holds the number of threads in the respective dimension.

**Block** can be identified by a one-, two- or three-dimensional `blockIdx`.

What happens if it is still not enough?
GPGPU Overview

1. allocate GPU memory
2. copy A and B values
3. Execute adding kernel
4. copy C from GPU
Parallel element-wise sum of two matrices

```
// Kernel definition
__global__ void MatAdd(float* A, float* B, float* C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int iStride = blockDim.x * gridDim.x;
    int jStride = blockDim.y * gridDim.y;

    for(int y = j; y < N; y += jStride) {
        for(int x = i; x < N; x += iStride) {
            C[y * N + x] = A[y * N + x] + B[y * N + x];
        }
    }
}
```

**Loop** over all indices.

**Memory distance** of threads within a block should be minimized.

Allows the GPU to utilize cached memory and therefore minimize global memory lookups.

**Loop** prevents the kernel of accessing indices outside the pointer.

**Block-** and **Gridsize** can be independent from matrix/vector size.
1. allocate GPU memory
2. copy A and B values
3. Execute adding kernel
4. copy C from GPU

A

B

C

A' + B' = C'

Block 1

Block N

Block ...
Device functions can only be called from global or other device functions.

Any return type allowed.

Split your device code into small functions for readability!
• **__host__** specifies a host (CPU) callable function.
  - Default specifier, any return type possible

• **__device__** specifies a device (GPU thread) callable function.
  - Threads may call these functions
  - Not only void as return type allowed

• **__host__ __device__** specifies a function callable by the host (CPU) and the device (GPU thread)
  - No need to write functions for CPU and GPU separately

• **__global__** can not be combined with any of these
Outline

Introduction

Programming Model

GPU Memory

Synchronization

NVCC

https://graphics.tu-bs.de/teaching/ss19/padi/
• Per-thread local memory
  - No other thread or block have access to read or write
GPU Memory – Local Memory

Pre-thread local memory

```c
// Kernel definition
__global__ void MatAdd(float* A, float* B, float* C, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int iStride = blockDim.x * gridSize.x;
    int jStride = blockDim.y * gridSize.y;

    for(int y = j; y < N; y += jStride)
    {
        for(int x = i; x < N; x += iStride)
        {
            C[y * N + x] = A[y * N + x] + B[y * N + x];
        }
    }
}
```

**All parameters** are in local memory.

**Pointer** are in **local memory**, but point to **global memory**!

Comparable to CPU: pointer are on STACK but point to HEAP

**Standard Variables** defined in **global or device functions** are in local memory.

**Local memory** is not accessible by any other thread.

**Local variables** will allocate memory for each thread!
• **Pre-thread local memory**
  - No other thread or block have access to read or write

• **Per-block shared memory**
  - All threads within a block have access to the same shared memory
  - Faster memory access than global memory
Pre-block shared memory

__shared__ specifier defines shared memory.

**Static shared memory** array size can be explicitly declared.

**Dynamic shared memory** array size must be given as an optional third execution configuration parameter.

Shared memory is accessible to all threads within the same block.

Shared variables will allocate memory for each block!
GPU Memory – Memory Hierarchy

- **Pre-thread local memory**
  - No other thread or block have access to read or write

- **Per-block shared memory**
  - All threads within a block have access to the same shared memory
  - Faster memory access than global memory

- **Global memory**
  - All threads within all blocks have access to this memory
  - Slowest memory access, try to copy to shared memory if possible
Global memory is not directly accessible.

Pointer can point to global memory!

Global memory allocation needs to be done before the kernel call!

Global memory is accessible by any thread within any block if they have a pointer pointing to it.

Global memory allocation is done once for each cudaMalloc(...) call!
1. allocate GPU memory

2. copy A and B values

3. Execute adding kernel

4. copy C from GPU

### GPGPU Overview

- **A**:
  - 2
  - 4
  - ...
  - 1
  - 7
  - ...
  - 8
  - 2
  - 7

- **B**:
  - 5
  - 1
  - ...
  - 8
  - 1
  - ...
  - 1
  - 1
  - 7

- **C**:
  - 7
  - 5
  - ...
  - 9
  - 9
  - ...
  - 9
  - 9
  - 9

- **A'**:
  - 2
  - 4
  - ...
  - 1
  - 7
  - ...
  - 8
  - 2
  - 7

- **B'**:
  - 5
  - 1
  - ...
  - 8
  - 1
  - ...
  - 1
  - 1
  - 7

- **C'**:
  - 7
  - 5
  - ...
  - 9
  - 9
  - ...
  - 9
  - 9
  - 9
GPU Memory – Global Memory Allocation

• Global GPU memory has to be allocated and freed with special commands
  ▪ `cudaMalloc(void** ptr, size_t sizeInBytes)` allocate memory on the GPU
  ▪ `cudaFree(void*)` will free the allocated memory

• GPU memory allocation does **not initialize** the memory!
  ▪ **Always** initialize it yourself before you read from it
1. allocate GPU memory
2. copy A and B values
3. Execute adding kernel
4. copy C from GPU
Global memory is not initialized during allocation!

cudaMemcpy(...) copies values from/to global memory.
Last parameter specifies direction.

cudaMemset(...) sets bytes to the specified int.

cudaMemcpy(void* dst, void* src, size_t sizeInBytes, cudaMemcpyKind kind)
cudaMemset(void* dst, int value, size_t sizeInBytes)

Both methods lock the CPU thread!
-> cudaMemcpyAsync(...), cudaMemcpyAsync(...)
GPU Memory – More memory types

• For standard 2D or 3D global memory it can be beneficial to add padding for faster access
  ▪ cudaMallocPitched(...), cudaMalloc3D(...) does add padding automatically
  ▪ cudaMemcpyPitched(...), cudaMemcpy3D(...) are then used to copy values

• Constant memory also resides in device memory
  ▪ Cached in constant cache
  ▪ Read-only memory, but lookup can be faster

• Texture and Surface memory also resides in device memory
  ▪ Cached in texture cache
  ▪ Optimized for 2D spatial locality, higher bandwidth can be achieved
  ▪ Read-only memory

https://graphics.tu-bs.de/teaching/ss19/padi/
Outline

Introduction
Programming Model
GPU Memory

Synchronization

NVCC
Synchronization – CPU and GPU

- Independent tasks that can operate concurrently
  - Computation on the host
  - Computation on the device
  - Memory transfer from the host to the device
  - Memory transfer from the device to the host
  - Memory transfers within the memory of a given device
  - Memory transfers among devices

- Kernel calls are asynchronous by default
  - Host code will continue

- `cudaMemcpy()`, `cudaMemset()` etc. also have asynchronous calls
  - `cudaMemcpyAsync()`, `cudaMemsetAsync()`, ...

- `cudaDeviceSynchronize()` can be used to synchronize GPU and CPU
  - Will lock host-code until all called device code finished

https://graphics.tu-bs.de/teaching/ss19/padi/
Synchronization – CPU and GPU – Streams

- Synchronization between kernel calls and copies?
  - Kernel, which alters global memory and a memcpy from this global memory are started
  - Copy should wait for the kernel to finish…
Synchronization – CPU and GPU – Streams

• Synchronization between kernel calls and copies?
  - Kernel, which alters global memory and a memcpy from this global memory are started
  - Copy should wait for the kernel to finish...

• Streams are used for device call synchronization
  - Same stream calls run sequentially in call order

• Without specification default stream (0) is used
  - All kernel and copy calls are synchronized
  - No need to care about this

• cudaStreamSynchronize(streamid) synchronizes CPU and GPU with respect to a specific stream

• Use streams when you have calls that are independent from others

https://graphics.tu-bs.de/teaching/ss19/padi/
Vector reduction

Sum of all elements in a vector

A

2
4
...
1
7
...
8
2

+ → 85
Synchronization – Threads

Vector reduction

```c
// Kernel definition
__global__ void reduce(int const* A, int* sum, int N)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = gridDim.x * blockDim.x;
    sum[0] = sum[0] + A[tid];
}

int *a_gpu, *sum;
cudaMalloc(&a_gpu, n * sizeof(int));
cudaMalloc(&sum, sizeof(int));
cudaMemcpy(a_gpu, b, n * sizeof(int), cudaMemcpyHostToDevice);
cudaMemset(sum, 0, sizeof(int));
Reduce<<<1, N>>>(a_gpu, sum, N);
```

Sum of all elements in a vector

Threads may not write to the same memory address in parallel

- Undefined behaviour

Threads may not read a memory address while another thread writes to it

- Undefined which value is read!
Synchronization – Threads – Atomics

Vector reduction

```c
// Kernel definition
__global__ void reduce(int const* A, int* sum, int N)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * blockDim.x;
    atomicAdd(sum, A[tid]);
}

int *a_gpu, *sum;
cudaMalloc(&a_gpu, n * sizeof(int));
cudaMalloc(&sum, sizeof(int));
cudaMemcpy(a_gpu, b, n * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(sum, 0, sizeof(int), cudaMemcpyHostToDevice);
Reduce<<<1, N>>>(a_gpu, sum, N);
```

Sum of all elements in a vector

Threads may not write to the same memory address in parallel
• Undefined behaviour

Threads may not read a memory address while another thread writes to it
• Undefined which value is read!

Atomics are guaranteed to be performed without interference from other threads!
Atomics guarantee execution without interference from other threads
- Even across blocks!

Several atomic-functions are implemented directly
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

In principle it is possible to implement a mutex for threads and make any functionality ‘atomic’

When atomic functions are used and another thread is accessing the memory it will have to wait for the atomic to end

Try to resolve synchronization elsewise if possible!

https://graphics.tu-bs.de/teaching/ss19/padi/
Synchronization – Threads

Vector reduction

Sum of all elements in a vector.

Each thread only sum 2 elements in each iteration -> less active waiting for a write/read operation!

Synchronization between iterations must be done!
Synchronization – Threads

Vector reduction

Sum of all elements in a vector.

Each thread only **sum 2 elements** in each iteration -> **less active waiting** for a write/read operation!

**Synchronization** between **iterations** must be done!
Vector reduction

Sum of all elements in a vector.

Each thread only **sum 2 elements** in each iteration -> **less active waiting** for a write/read operation!

**Synchronization** between **iterations** must be done!

```
A

2  4  ...
\rightarrow 6  \rightarrow 20
\rightarrow ... \rightarrow 24
1  7  ...
\rightarrow 8  \rightarrow 26
\rightarrow ... \rightarrow 15
8  2
\rightarrow 10  \rightarrow 15
```

sync
Vector reduction

Sum of all elements in a vector.

Each thread only sum 2 elements in each iteration -> less active waiting for a write/read operation!

**Synchronization** between iterations must be done!
Vector reduction

Sum of all elements in a vector.

Each thread only \textbf{sum 2 elements} in each iteration -> \textbf{less active waiting} for a write/read operation!

\textbf{Synchronization} between iterations must be done!
Vector reduction

Sum of all elements in a vector.

Each thread only **sum 2 elements** in each iteration -> **less active waiting** for a write/read operation!

**Synchronization** between iterations must be done!
Synchronization – Threads

Vector reduction

Sum of all elements in a vector.

Each thread only sum 2 elements in each iteration -> less active waiting for a write/read operation!

**Synchronization** between iterations must be done!

https://graphics.tu-bs.de/teaching/ss19/padi/
Vector reduction

```c
// Kernel definition
__global__ void reduce(int const* A, int* sum, int N) {
    extern __shared__ int sdata[];
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = gridDim.x * blockDim.x;
    sdata[tid] = A[tid];
    __syncthreads();

    for(int s=1; s < blockDim.x; s*=2) {
        if(tid % (2*s) == 0)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }

    if(tid==0) sum[0] = sdata[0];
}

Reduce<<<1, N, N*sizeof(int)>>>(a_gpu, sum, N);
```

Sum of all elements in a vector

`__syncthreads()` will synchronize all threads within the same block

No synchronization across blocks
- may not run in parallel

For even more speedups (up to 30x):

Use the visual profiler
Outline

Introduction

Programming Model

GPU Memory

Synchronization

NVCC
• CUDA files should end with .cu and compiled with NVCC

• NVCC – NVidia Cuda Compiler
  ▪ Proprietary compiler by Nvidia for CUDA

• Separates the code into two parts
  ▪ Host code, which is forwarded to a C compiler
  ▪ Device code, which is further compiled by NVCC

• CMake will automatically use NVCC for .cu files
  ▪ Needs CMake 3.8 or newer
  ▪ For older versions use find_package() and cuda_add_executable()
• Debugging device code can be difficult due to concurrency

• Threads can call printf() in general for basic ’debugging’
  ▪ Be aware that the output can be in any order within a call due to concurrency

• CUDA API calls return a cudaError that can be checked
  ▪ cudaMemcpy(), cudaMemcpy(), …
CUDA API calls return a cudaError that can be checked
• cudaMalloc(), cudaMemcpy(), ...

CUDA kernel calls do not return any value
• asynchronous calls cannot know whether an error will occur
• After execution it can be checked if an error occurred with cudaGetLastError()

Error message can be checked with cudaGetErrorString()
• Debugging device code can be difficult due to concurrency

• Threads can call printf() in general for basic 'debugging'
  ▪ Be aware that the output can be in any order within a call due to concurrency

• CUDA API calls return a cudaError that can be checked
  ▪ cudaMemcpy(), cudaMemcpy(), ...

• NVIDIA Nsight CUDA Debugger enables debugging gpu code
  ▪ Breakpoints, insight to variables, step through device code, etc.