GPGPU
General Purpose Computing on Graphics Processing Units

These slides were created by David Rohr and Matthias Bach

Volker Lindenstruth (www.compeng.de) 14. Juli 2010— Copyright ©, Goethe Uni, alle Rechte vorbehalten

Roundup

- **Possibilities to increase computing performance**
  - **Increased clock speed**
  - **More complex instructions**
  - **Improved instruction throughput (caches, branch prediction, …)**
  - **Vectorization**
  - **Parallelization**

Possibilities and Problems

- **increased clock speed**
  - **Power consumption / cooling**
  - **Limited by state of the art lithography**
- **More complex instructions**
  - **Require more transistors / bigger cores**
  - **Negative effect on clock speed**
- **Caches, pipelining, branch prediction, out of order execution, …**
  - **Require many more transistors**
- **Vectorization / parallelization**
  - **Difficult to program**

Possible features for an HPC chip

- **Parallelism is obligatory**
  - **Both vectorization and many core seem reasonable**
  - **Huge vectors are easier to realize than a large number of cores (e.g. only 1 instruction decoder per vector processor)**
  - **Independent cores can process independent instructions which might be preferred for some algorithms**
- **Complex instructions, out of order execution, etc.**
  - **Hardware requirements are huge**
  - **Not suited for a many core design as the additional hardware is required multiple times**
- **Clock speed**
  - **Limited anyway, not so relevant in HPC as performance originates from parallelism**

Design Guideline for a GPU

- Use many cores in parallel
- Each core on its own has SIMD capabilities
- Keep the cores simple

  (Rather use many simple cores instead of fewer (faster) complex cores)
- This means: no out of order execution, etc.
- Use the highest clock speed possible, but do not focus on frequency
- Pipelining has no excessive register requirement and is required for a reasonable clock speed, therefore a small pipeline is used

Graphics Processing Unit Architectures

Volker Lindenstruth (www.compeng.de) 14. Juli 2010— Copyright ©, Goethe Uni, alle Rechte vorbehalten

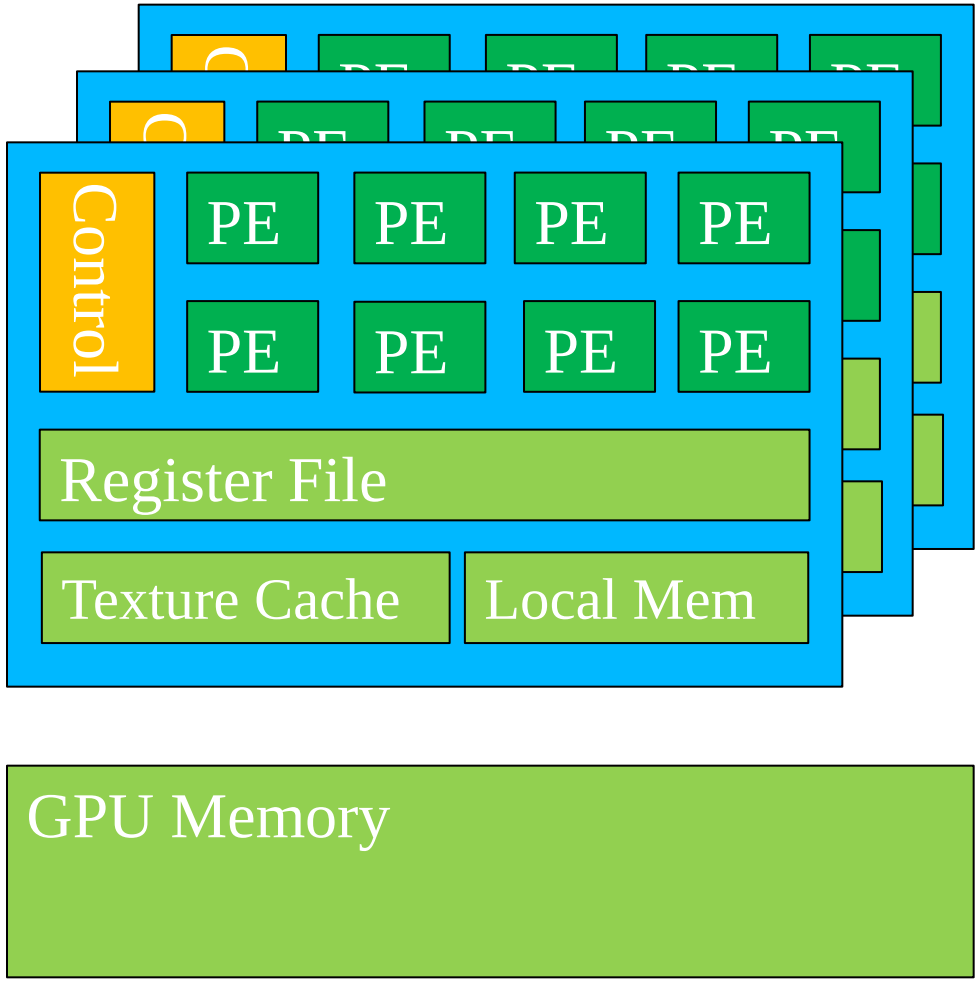
Todays graphics pipeline

- **Executed per primitive**
  - **Polygon, Vertex, Pixel**
  - **Highly parallel**
- **Everything but rasterization (and display) is software**

Volker Lindenstruth (www.compeng.de) 14. Juli 2010— Copyright ©, Goethe Uni, alle Rechte vorbehalten

A generic GPU architecture

- One hardware for all stages
- Modular setup
- Streaming Hardware
- Hardware scheduling
- Dynamic Register Count
- Processing Elements contain FPUs



Two application examples

- **Vector Addition**
- $\vec{c} = \vec{a} + \vec{b}$
- **Linear pseudo-code:**

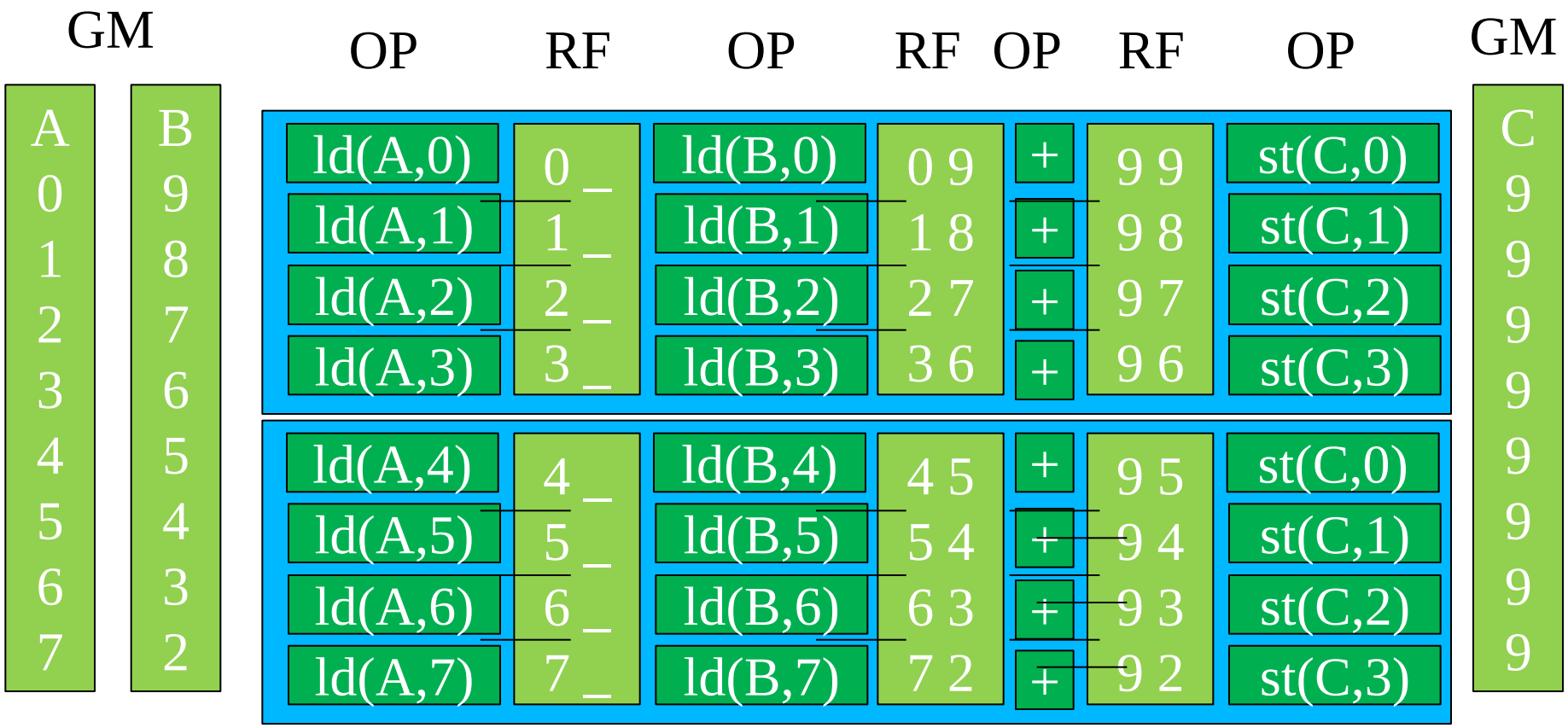
```
for i in lengthof(c)
    c[i] = a[i]
         + b[i]
```

- **Reduction**
- $s = \sum_{i=1}^{n} \vec{a_i}$
- **Linear pseudo-code:**

```
s = 0
for i in n
    s += a[i]
```

Volker Lindenstruth (www.compeng.de) 14. Juli 2010— Copyright ©, Goethe Uni, alle Rechte vorbehalten

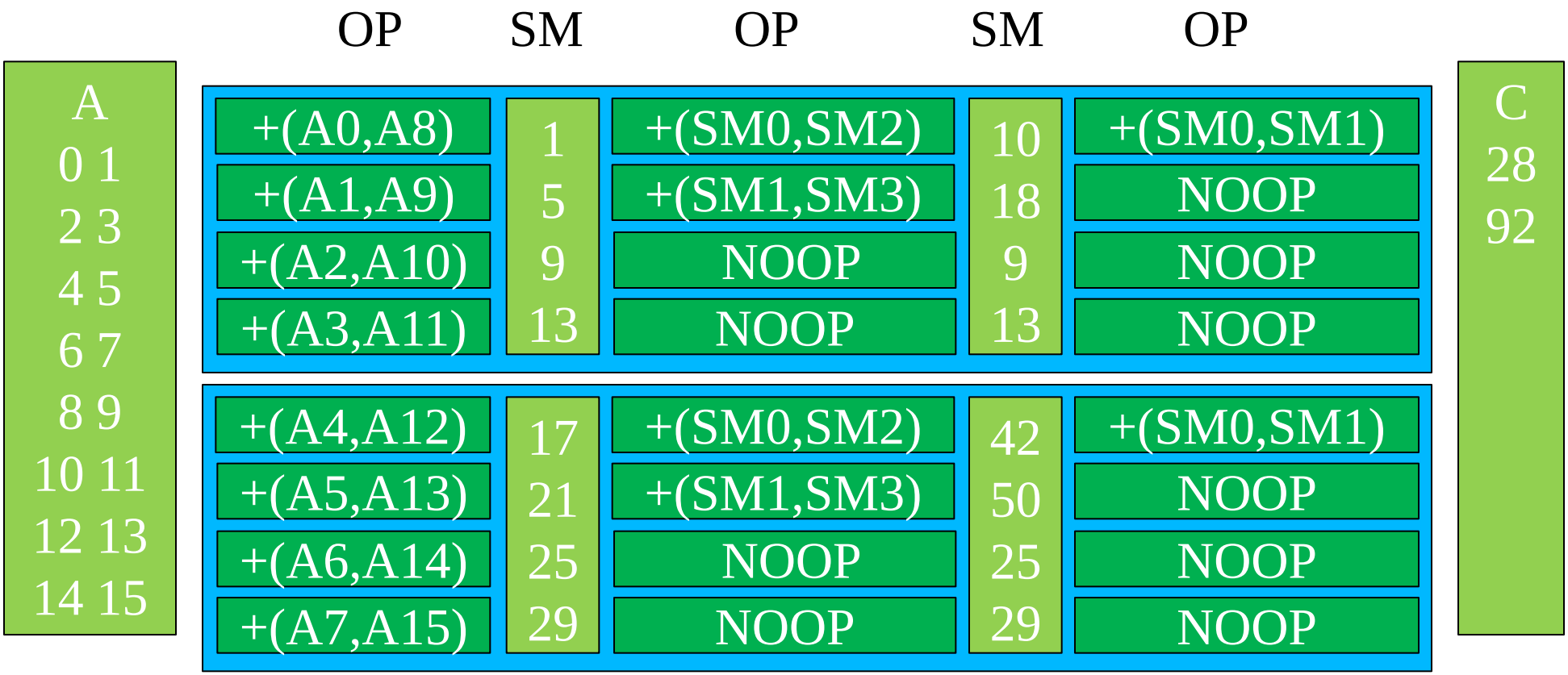
Generic architecture: Vector addition



The examples use only two compute units with 4 PEs each to make the visualization easier to overview.
We will also skip (imlicit) global memory ops and register file content in the next examples.

Volker Lindenstruth (www.compeng.de) 14. Juli 2010— Copyright ©, Goethe Uni, alle Rechte vorbehalten

Generic architecture: Reduction

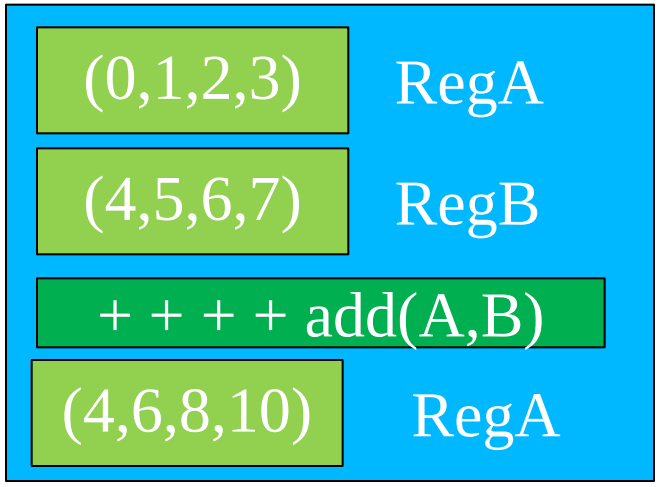


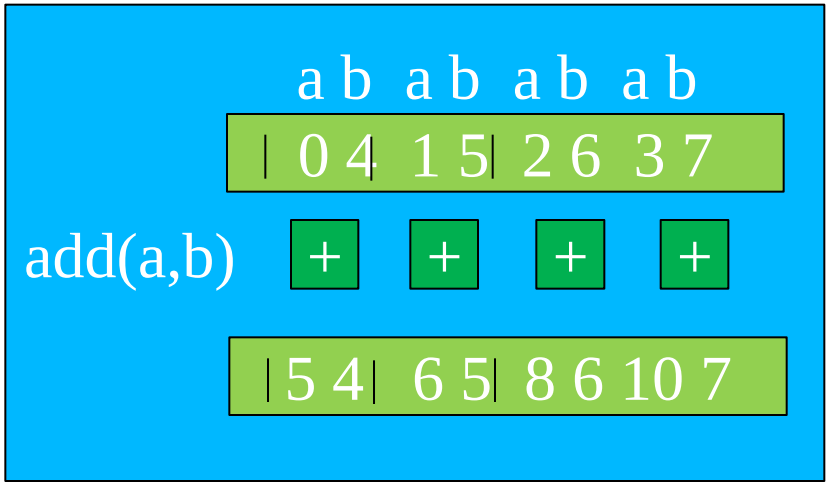
- No syncing between compute units
  - Second pass (with only one compute unit) to add results of compute units.

Volker Lindenstruth (www.compeng.de) 14. Juli 2010— Copyright ©, Goethe Uni, alle Rechte vorbehalten

SIMT

- **Groups of threads executed in lock-step**
  - **Lock-step makes it similar to SIMD**
- **Own Register Set for each Processing Element**
  - **„Vector-Width“ given by FPUs, not register size**
  - **Gather/Scatter not restricted (see reduction example)**
  - **No masking required for conditional execution**
  - **More flexible register (re)usage**





Volker Lindenstruth (www.compeng.de) 14. Juli 2010— Copyright ©, Goethe Uni, alle Rechte vorbehalten

„HyperThreading“

- Hardware Scheduler
  - Zero-Overhead Thread Switch
  - Schedules thread groups onto processing units
- Latency Hiding
  - E.g. NVIDIA Tesla:
    - 400 to 500 cycles memory latency
    - 4 cycles per thread group
    - 100 thread groups (320 threads) on processing group to completely hide latency

1 Thread Group	READ								READ
6 Thread Groups	READ	READ	READ	READ	READ	READ			READ

In the example each thread issues a number of reads as required e.g. in vector addition example. The read latency is assumed equivalent to executing 8 thread groups. Colors distinguish groups. Thread groups are to be understood as (concurrently scheduled to the current processing unit)

Volker Lindenstruth (www.compeng.de) 14. Juli 2010— Copyright ©, Goethe Uni, alle Rechte vorbehalten

Stream Computing

- Limited execution control
  - Specify workitem count and thread group size
  - No synchronization between thread groups
    - Allows scaling over devices of different size
- Relaxed memory consistency
  - Memory only consistent within thread
  - Consistent for thread group at synchronization points
  - Not consistent between thread groups
    - No synchronization possibility anyway
    - Save silicon for FPUs.
  - Globally consistant memory only at end of execution.

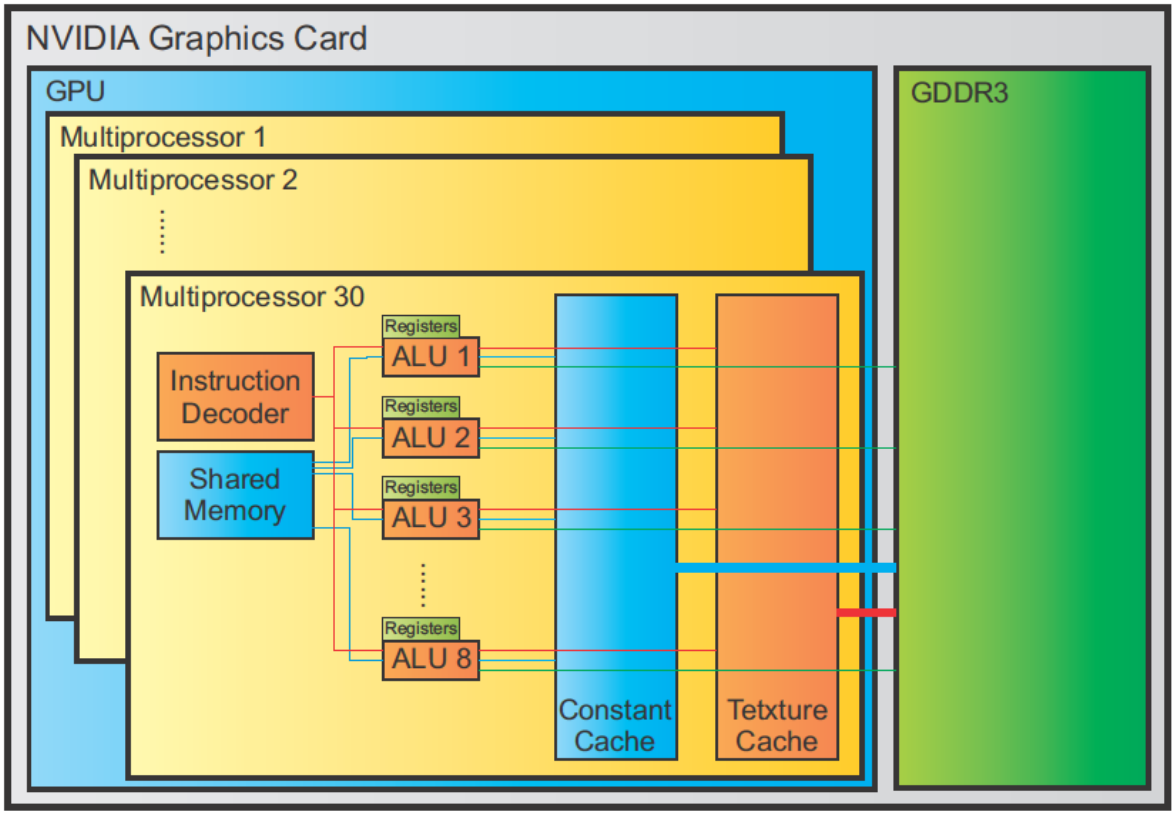
— Copyright ©, Goethe Uni, alle Rechte vorbehalten

Register Files / Local Memory / Caches

- Register File – Dynamic Register Set Size
  - Many threads with low register usage
    - Good to hide memory latencies
    - High throughput
  - Less threads with high register usage
    - Only suited for compute intense
- Local Memory – Data exchange within thread group
- Spatial locality cache
  - CPU caches work with temporal locality
  - Reduces memory transaction count for multiple threads reading “close“ addresses
  - 2D / 3D locality requires special memory layout

— Copyright ©, Goethe Uni, alle Rechte vorbehalten

Schematic of NVIDIA GT200b chip

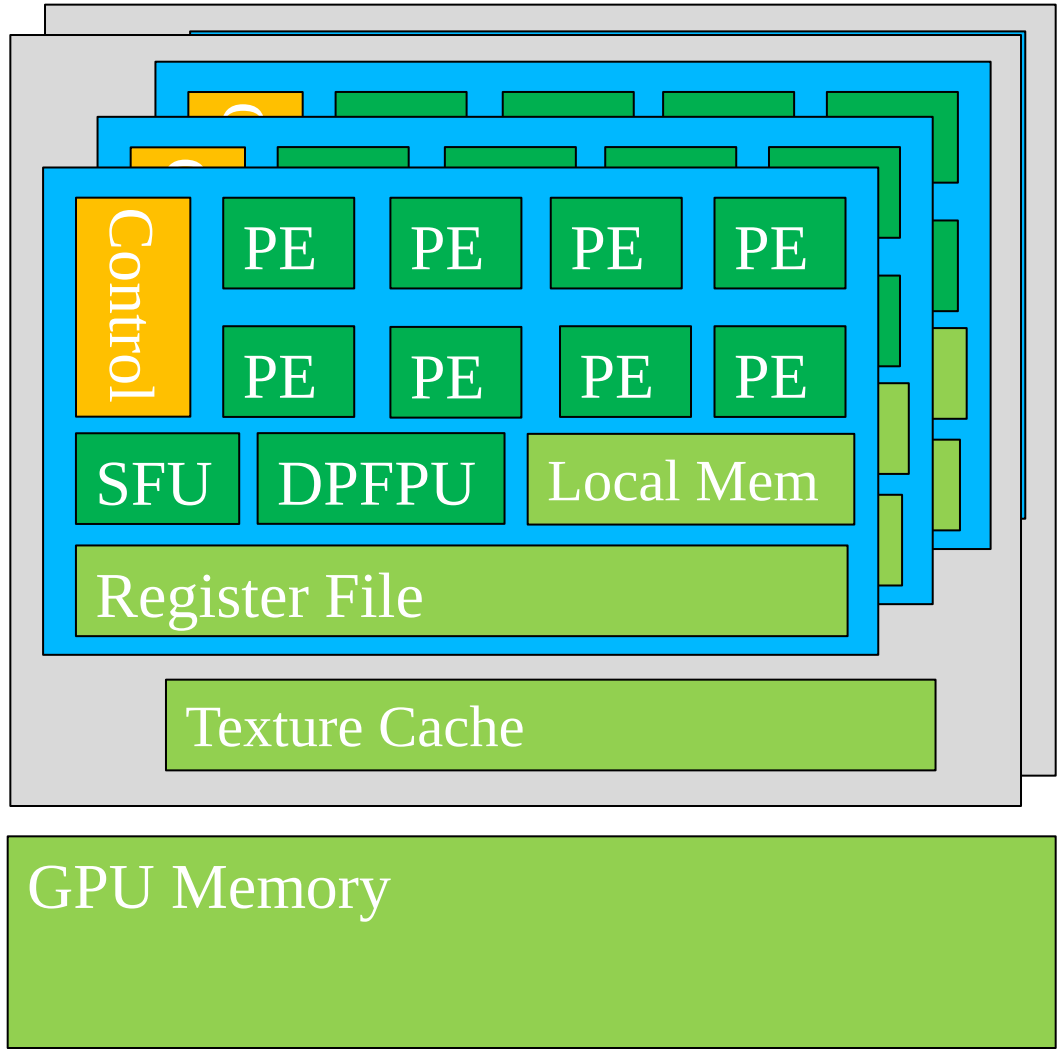


A full-featured Coherent read/write cache would be too complex. Instead several small special purpose caches are employed. Future generations have general purpose L2 Cache.

- **many core design (30 multiprocessors)**
- **8 ALUs per multiprocessor (vector width: 8)**

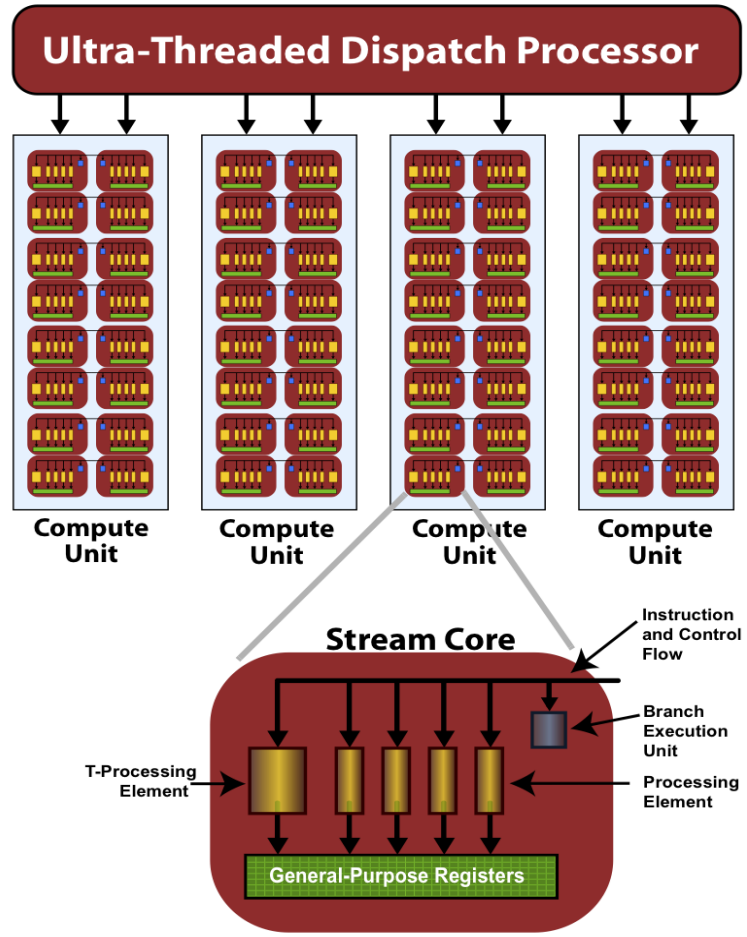
NVIDA Tesla Architecture

- **Close to generic architecture**
  - **Lockstep size: 16**
  - **1 DP FPU per Compute Unit**
  - **1 SFU per Compute Unit**
  - **3 Compute Units grouped into Thread Processing Cluster**
  - **Global Memory Atomics**



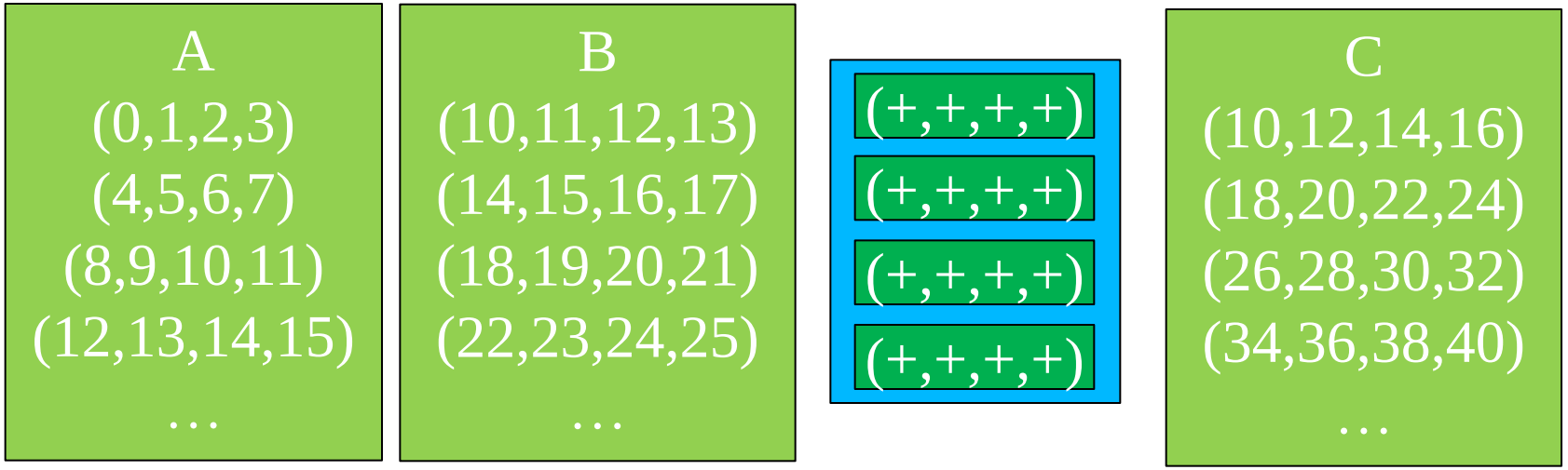
ATI Cypress Architecture

- **VLIW PEs**
  - 4 SP FPUs
  - 1 Special Function Unit
  - 1 to 2 DP ops per cycle
- **HD 5870**
  - 20 Compute Units
  - 16 Stream Cores each
  - 1600 FPUs total
  - Lockstep size: 64
- **Global Memory Atomics**



VLIW = Very Long Instruction Word

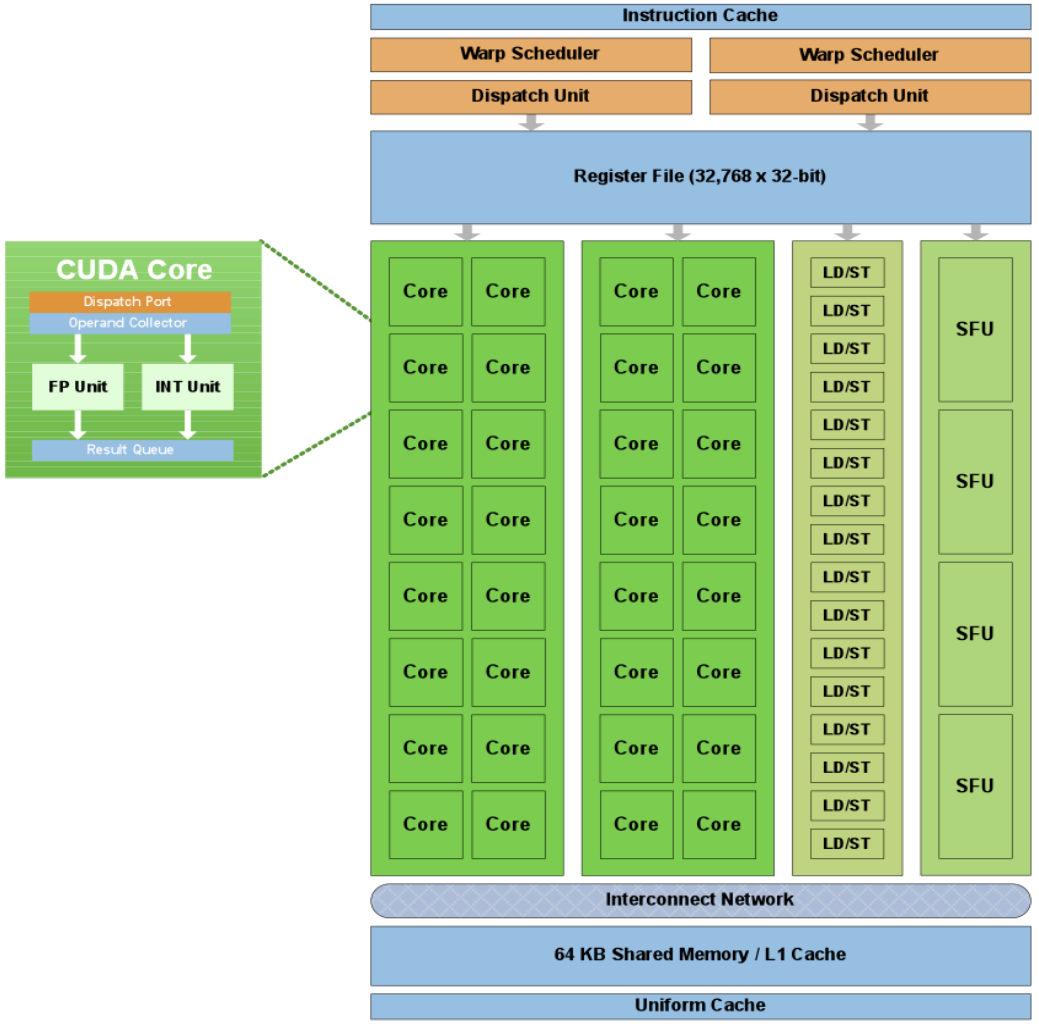
- **VLIW PE is similar to SIMD core**
  - **FPUs can execute different ops**
  - **Data for FPUs within VLIW must be independent**
    - **Compiler need to detect this to generate proper VLIW**
    - **Often results in SIMD style code / using vector types, e.g. float4**



Volker Lindenstruth (www.compeng.de) 14. Juli 2010— Copyright ©, Goethe Uni, alle Rechte vorbehalten

NVIDIA Fermi Architecture

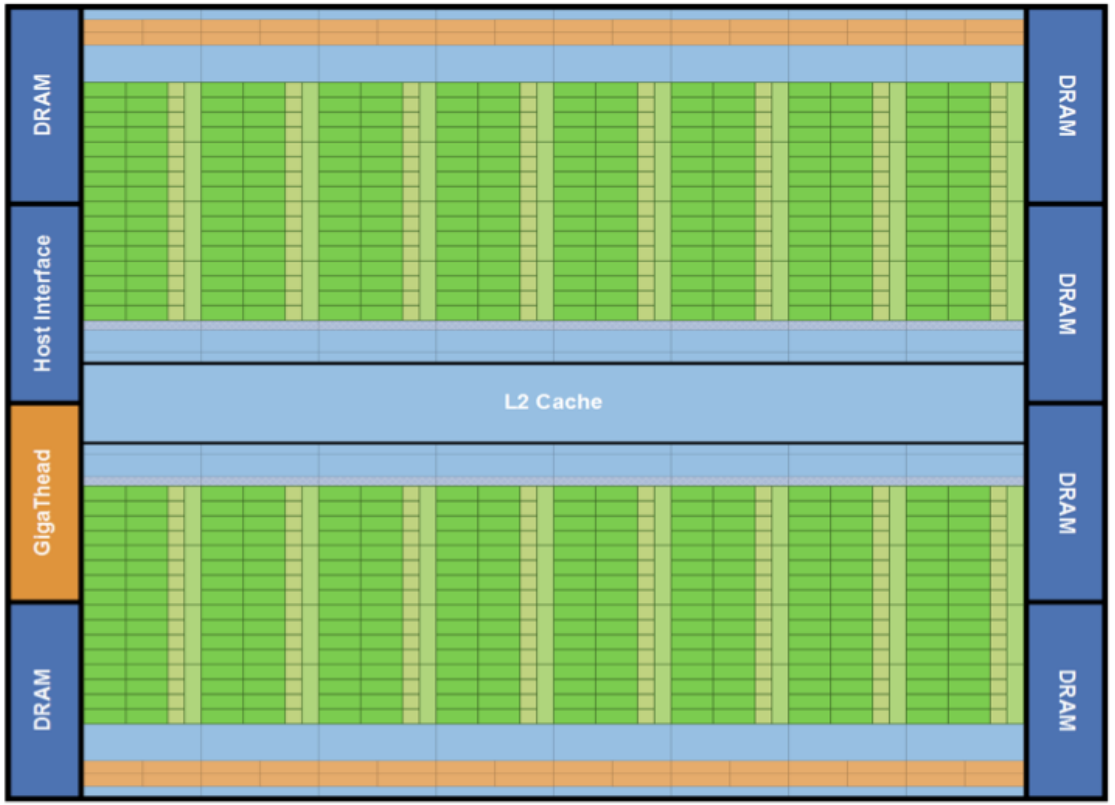
- PE = CUDA core
  - 2 Cores fused for DP ops
- 2 Instruction Decoders per Compute Unit
- Lockstep size: 32



Volker Lindenstruth (www.compeng.de) 14. Juli 2010— Copyright ©, Goethe Uni, alle Rechte vorbehalten

NVIDIA Fermi Architecture

- **Large L2 cache**
  - **Unusual**
  - **Shared**
  - **Read-Write**
- **No synchronization between Compute Units**
  - **Global Memory Atomics exist**



Volker Lindenstruth (www.compeng.de) 14. Juli 2010— Copyright ©, Goethe Uni, alle Rechte vorbehalten

GPUs in Comparison

	NVIDIA Tesla	NVIDIA Fermi	AMD HD5000
FPUs	240	448	1600
Performance SP / Gflops	933	1030	2720
Performance DP / Gflops	78	515	544
Memory Bandwidth / GiB/s	102	144	153.6
Local Scratch Memory / KiB	16	16 – 48	32
Cache (L2) / MiB	N/A (Texture only)	10.5	N/A (Texture only)

SIMD

- **Recall SIMD (Single Instruction Multiple Data)**
  - **one instruction stream processed multiple data streams in parallel**
  - **often called vectorization**

SIMD v.s. SIMT

- **new programming model introduced by NVIDIA SIMT: Single Instruction Multiple Threads**
  - **resembles programming a vector processor**
  - **instead of vectors threads are use**
  - **BUT: as only 1 instruction decoder is available all threads have to execute the same instruction**
  - **SIMT is in fact an abstraction for vectorization**
  - **SIMT code looks like many core code**
  - **BUT: the vector-like structure of the GPU must be kept in mind to achieve optimal performance**

SIMT example

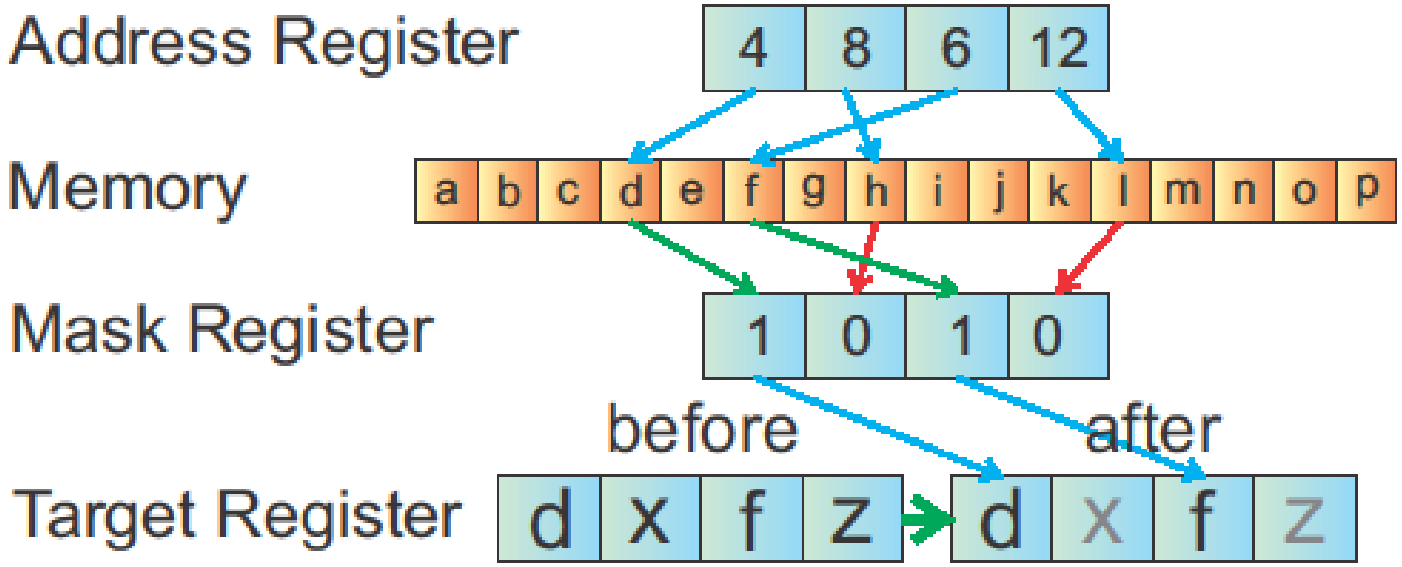
- how to add 2 vectors in memory
- the corresponding vectorized code would be:
  dest = src1 + src2;
- the SIMT way:
  - each element of the vector in memory is processed by an independent thread
  - each thread is assigned a private variable (called thread_id in this example) determining which element to process
  - SIMT code:

    ```
    dest[thread_id] = src1[thread_id] + src2[thread_id];
    ```

  - dest, src1, and src2 are pointers not vectors of course
  - a number of threads equal to the vector size is started executing the above instruction in parallel

SIMD v.s. SIMT examples

- **masked vector gather as example**
  (Remember VC)



SIMD v.s. SIMT examples

- **SIMD masked vector gather (vector example)**
  - **int_v dst;**
  - **int_m mask;**
  - **int_m *addr;**
  - **code: dst(mask) = load_vector(addr);**
  - **only one instruction executed by one thread on a data vector**
- **SIMT masked vector gather**
  - **int dst;**
  - **bool mask;**
  - **int *addr;**
  - **code: if (mask) dst = addr[thread_id];**
  - **multiple instructions executed by the threads in parallel**
  - **source is a vector in memory, target is a set of registers but no vector-register**

SIMD v.s. SIMT comparison

- **why use SIMT at all**
  - **SIMT allows if-, else-, while-, and switch-statements etc. as commonly used in scalar code**
  - **no masks required**
  - **this makes porting code to SIMT easier**
  - **especially code that has been developed to run on many core systems (e.g. using OpenMP, Intel TBB) can easily be adopted (see next example)**
- **SIMT primary (dis)advantages**
  - **+ easier portability / more opportunities for conditional code**
  - **– implizit vector nature of chip is likely to be not dealt with resulting in poor performance**

SIMT threads

- **threads within one multiprocessor**
  - **usually more threads than ALUs are present on each multiprocessor**
  - **this assures a good overall utilization (latency hiding: threads waiting for memory accesses to finish are replaced by the scheduler with other threads without any overhead)**
  - **Thread count per multiprocessor is usually a multiple of the ALU count (only a minimum thread count can be defined)**
- **threads of different multiprocessors**
  - **As only one instruction decoder is present, threads on one particular multiprocessor must execute common instructions**
  - **threads of different multiprocessors are completely independent**

Porting OpenMP Code

- **simple OpenMP code**
  - **#pragma omp parallel for**

    for (int i = 0;i < max;i++) {

    //do something

    }
- **SIMT code**

  int i = thread_id;

  if (i < max) {

  //do something

  }

  - **Enough threads are started so that no loop is necessary**
  - **the check for i < max is needed**

Languages for GPGPU

- **OpenGL / Direct3D**
  - **first GPGPU approaches tried to encapsulate general problems in 3D graphics calculation, representing the source data by textures and encoding the result in the graphics rendered by the GPU**
  - **not used anymore**
- **CUDA (*Compute Unified Device Architecture)***
  - **SIMT approach by NVIDIA**
- **OpenCL**
  - **open SIMT approach by the Khronos Group that is platform independent (compare OpenCL)**
  - **very similar to CUDA (CUDA still has more features)**
- **AMD / ATI Stream**
  - **Based on Brook, recently AMD focuses in OpenCL**

Languages for GPGPU

- OpenGL / Direct3D / Stream seem out-dated
- this course will focus primarily on OpenCL
  - OpenCL is favored because it is an open framework
  - more importantly OpenCL is platform independent, not even restricted to GPUs but also available for CPU (with auto-vectorization support)
- some notes will be made about CUDA
  - especially where CUDA offers features OpenCL not available in OpenCL, such as:
  - Full C++ support

    (CUDA offered limited functionality for C++ started from the beginning. Full C++ support is available as of version 3.0)
  - this strongly suggest the application of CUDA when porting C++ codes
  - support for DMA transfer using page-locked memory

OpenCL Introduction

— Copyright ©, Goethe Uni, alle Rechte vorbehalten

OpenCL Introduction

- **OpenCL distinguishes between two types of functions**
  - **regular host functions**
  - **kernels (functions executed on the computing device)**

    “__kernel” - keyword

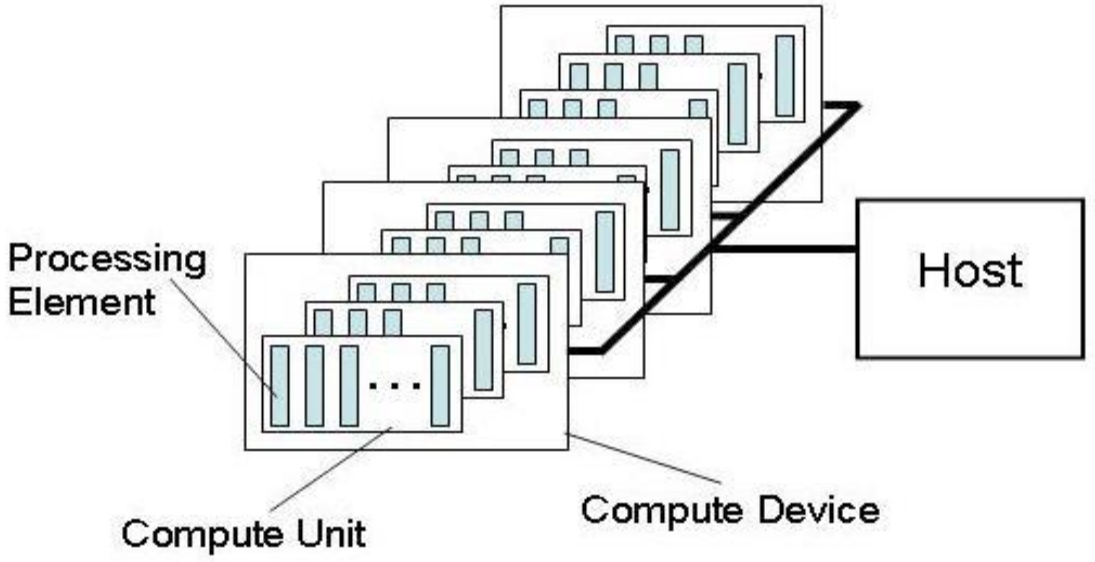
  - **in the following “host” will always refer to the CPU and the main memory**
  - **“device” will identify the computing device and its memory, usually the graphics card (also a CPU can be the device when running OpenCL code on CPU. Then both host and device code executes in different threads on the CPU. The host thread is responsible for administrative tasks while the device threads do all the calculations)**

OpenCL Kernels / Subroutines

- **Subroutines**
  - **to initiate a calculation on the computing device a kernel must be called by a host function**
  - **kernels can call other functions on the device but can obviously never call host functions**
  - **the kernels are usually stored in plain source code and compiled at runtime (functions called by the kernel must be contained there too), then transferred to the device where they are executed (see example later)**
  - **several third party libraries simplify this task**
- **Compilation**
  - **OpenCL is platform independent and it is up to the compiler how to treat function calls. Usually calls are simply inlined**

OpenCL Devices

- **in OpenCL terminology several compute devices can be attached to a host (e.g. multiple graphics cards)**
- **each compute device can possess multiple compute units (e.g. the multiprocessors in the case of NVIDIA)**
- **each compute unit consists of multiple processing elements, which are virtual scalar processors each executing one thread**



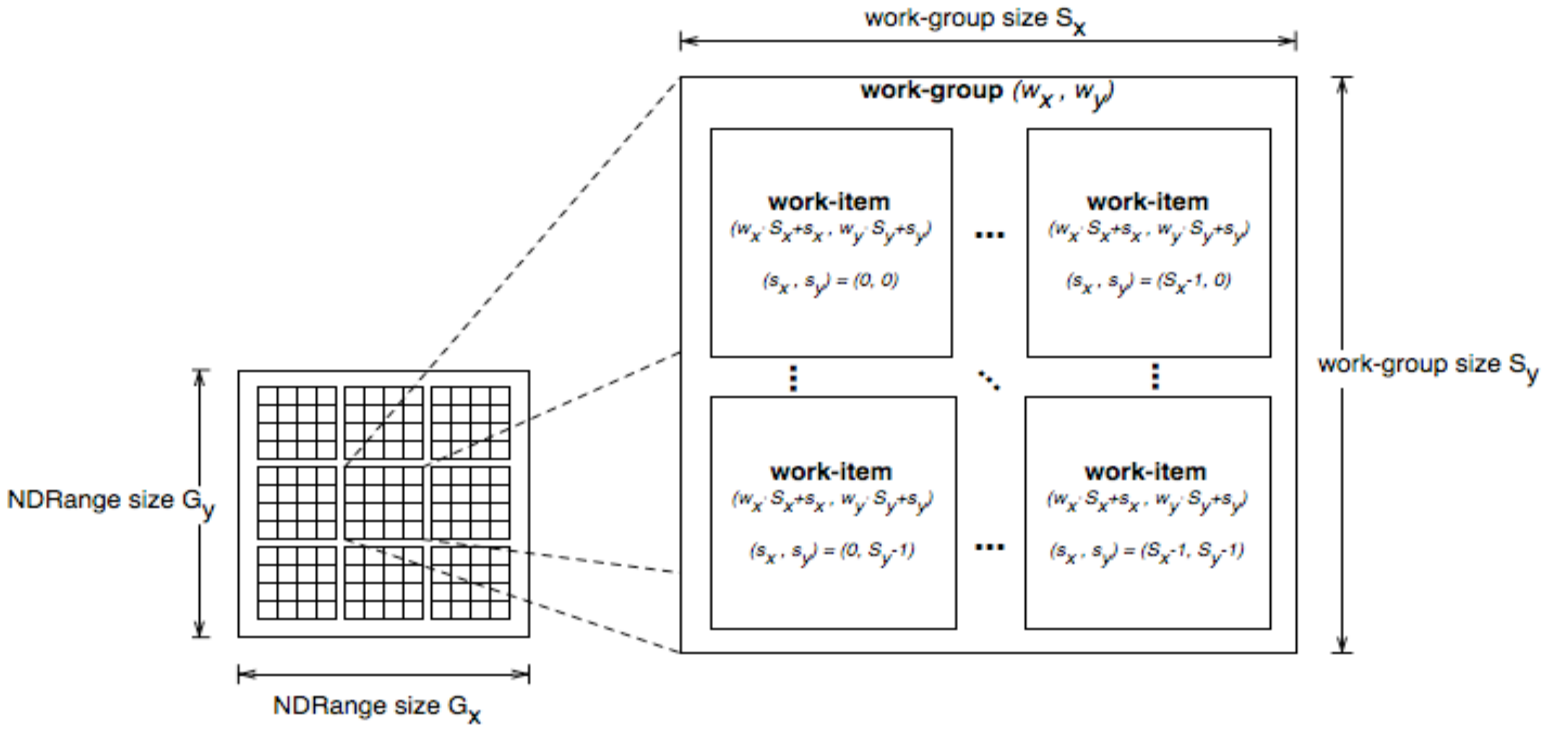
Volker Lindenstruth (www.compeng.de) 14. Juli 2010— Copyright ©, Goethe Uni, alle Rechte vorbehalten

OpenCL Execution Configuration

- **Kernels are executed the following way**
  - **n*m kernel instances are created in parallel, which are called work-items (each is assigned a global ID 0…(n*m-1))**
  - **work-items are grouped in n work-groups**
  - **work grouped are indexed (0…(n-1))**
  - **each work-item is further identified by a local work-item-ID inside its work group (0…(m-1))**
  - **thus the work-item can be uniquely identified using the global id or both the local- and the work-group-ID**
- **The work-groups are distributed as follows**
  - **all work-items within one work-group are executed concurrently within one compute unit**
  - **Different work-groups may be executed simultaneously or sequentially on the same or different compute unit where the execution order is not well defined**

More Complex Execution Configuration

- **OpenCL allows the indexes for the work-items and work-groups to be N-dimensional**
  - **often well suited for some problems, especially image manipulation (recall that GPU originally render images)**



Command Queues

- OpenCL kernel calls are assigned to a command queue
- command queues can also contain memory transfer operations and barriers
- execution of command queues and the host code is asynchronous
- barriers can be used to synchronize the host with a queue
- tasks issued to command queues are executed in order

Realization of Execution Configuration

- consider n work-groups of m work-items each
  - each compute unit must uphold at least m threads
  - m is limited by the hardware scheduler (on NVIDIA GPUs the limit varies between 256 and 1024)
  - if m is too small the compute unit might not be well utilized
  - multiple work-groups (say k) can then be executed in parallel on the same compute unit (which then executed k*m threads)

  - each work-item has a certain requirement for registers, memory, etc
  - say each work-items requires l registers, then in total m*k*l registers must be available on the compute-unit
  - this further limits the maximal number of threads

Platform Independent Realization

- **register limitation and platform independence**
  - **OpenCL code is platform independent and compiled at runtime**
  - **apparantly this solves the problem with limited registers, because the compiler knows how many work-items to execute and can create code with reduced register requirement (up to a certain limit)**
  - **no switch or parameter available that controls the register usage of the compiler, everything is decided by runtime**
  - **HOWEVER: register restriction leads to intermediate results being stored in memory and thus might result in poor performance**

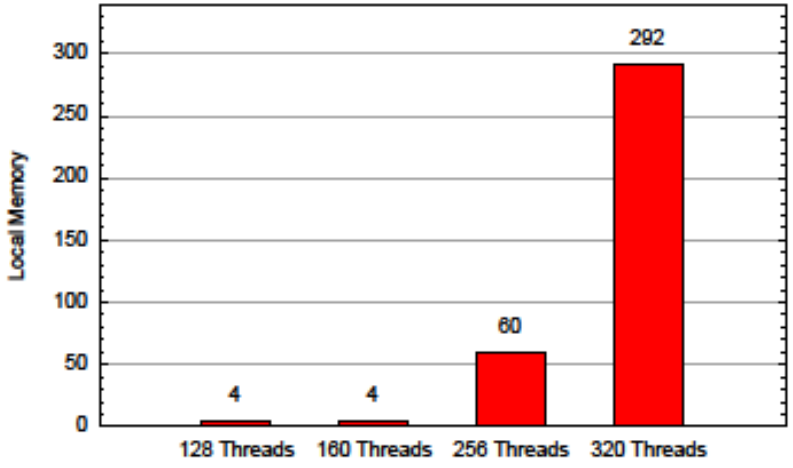
Register / Thread Trade-Off

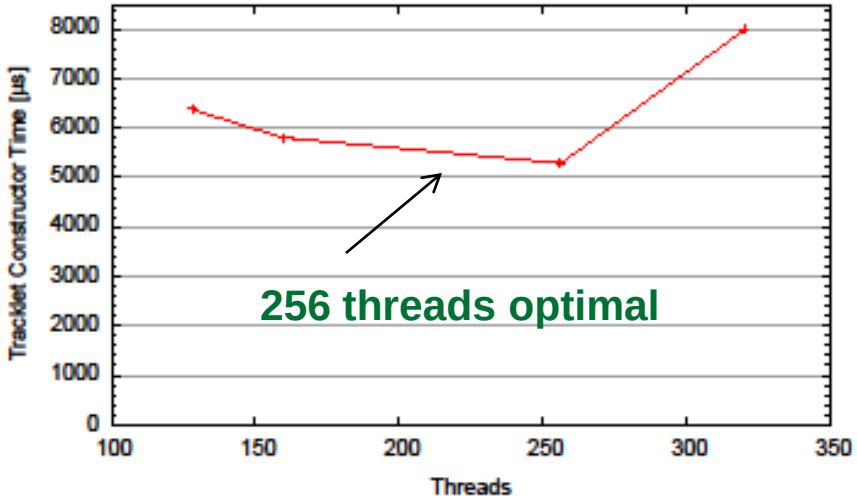
- **this can be discussed more concretely in the CUDA case, here the compiler is platform dependent and its behavior is well defined**
  - **more registers result in faster threads**
  - **more threads lead to a better overall utilization**
  - **the best parameter has to be determined experimentally**

Performance Impact of Register Usage

- **Real World CUDA HPC Application (later in detail) ALICE HLT Online Tracker on GPU**
  - **Performance for different thread- / register-counts**
  - **Register and thread count is related as follows**

Registers	48	64	96	128
Threads	320	256	160	128





Summary of register benchmarks

- **Optimal parameter was found experimentally**
- **It depends on the hardware**
- **Little influence possible in OpenCL code**
  (as it is platform independent)
- **CUDA allows for better optimizations**
  (as it is closer to the hardware)
- **OpenCL optimizations possible on compiler side**
  (e.g. just in time recompilation, compare JAVA)

OpenCL kernel sizes

- **Recall that functions are usually inlined**
- **Kernel register requirement commonly increases with amount of kernel source code**

  (the compiler tries to eliminate registers at its best but often cannot assure the independence of variables that can share a register)

→Try to keep kernels "small"

- **multiple small kernels executed sequentially usually perform better than one big kernel**
- **split tasks in small steps as possible**

One more theoretical part: Memory

- no access to host main memory by device
  - possible using special functions but slow
- device memory itself divided into:
  - global memory
  - constant memory
  - local memory
  - private memory
- before the kernel gets executed the relevant data must be transferred from the host to the device
- after the kernel execution the result is transferred back
  - special function can be used to do this automatically

Device Memory in Detail

- **global memory**
  - **global memory is the main device memory (such as main memory for the host)**
  - **can be written to and read from by all work-items and by the host through special runtime functions**
  - **global memory may be cached depending on the device capabilities but should be considered slow**

    (Even if it is cached, the cache is usually not as sophisticated as a usual CPU L1 cache. “Slow” still means transfer rates of more than 150 Gb/sec (for the newest generation NVIDIA Fermi cards). Random access however should be avoided at any case. “Coalescing Rules” to achieve optimal performance on NVIDIA cards will be explained later.)
  - **Global memory is not coherent.**

Device Memory in Detail

- **constant memory**
  - **a region of global memory that remains constant during kernel execution**
  - **often this allows for easier caching**
  - **constant memory is allocated and written to by the host**

Device Memory in Detail

- **local memory**
  - **special memory that is shared among all work-items in one work-group**
  - **local memory is generally very fast**
  - **atomic operations to local memory can be used to synchronize and share data between work-items**
  - **when global memory is too slow and no cache is available it is a general practice to use local memory as an explicit (non transparent) global memory cache**

Device Memory in Detail

- **private memory**
  - **as the name implies this is private for each work-item**
  - **private memory is usually a region of global memory**
  - **each thread requires its own private memory, so when executing n work-groups of m work-items each n*m*k bytes of global memory is reserved (with k the amount of private memory required by one thread)**
  - **as global memory is usually big compared to private memory requirements, the available private memory is usually not exceeded**
  - **if the compiler is short of registers it will swap register content to private memory**

OpenCL Memory Summary

	global memory	constant memory	local memory	private memory
host	dynamic allocation, read/write	dynamic allocation, read/write	dynamic allocation, no access	no allocation, no access
device	no allocation, read/write	static allocation read ony	static allocation read/write	static allocation read/write

Correspondance OpenCL / CUDA

- **As already stated OpenCL and CUDA resemble each other. However terminology differs:**

OpenCL	CUDA
host / compute device / kernel	host / device / kernel
compute unit	multiprocessor
global memory	global memory
constant memory	constant memory
local memory	shared memory
private memory	local memory
work-item	thread
work-unit	block
keyword for (sub)kernels: kernel	__global (__device)
command queue	stream

Be carefull with local memory, as it refers to different memory types

Memory Realization

- **the OpenCL specification does not define memory sizes and type (speed, etc.)**
- **we look at it in the case of CUDA (GT200b chip)**

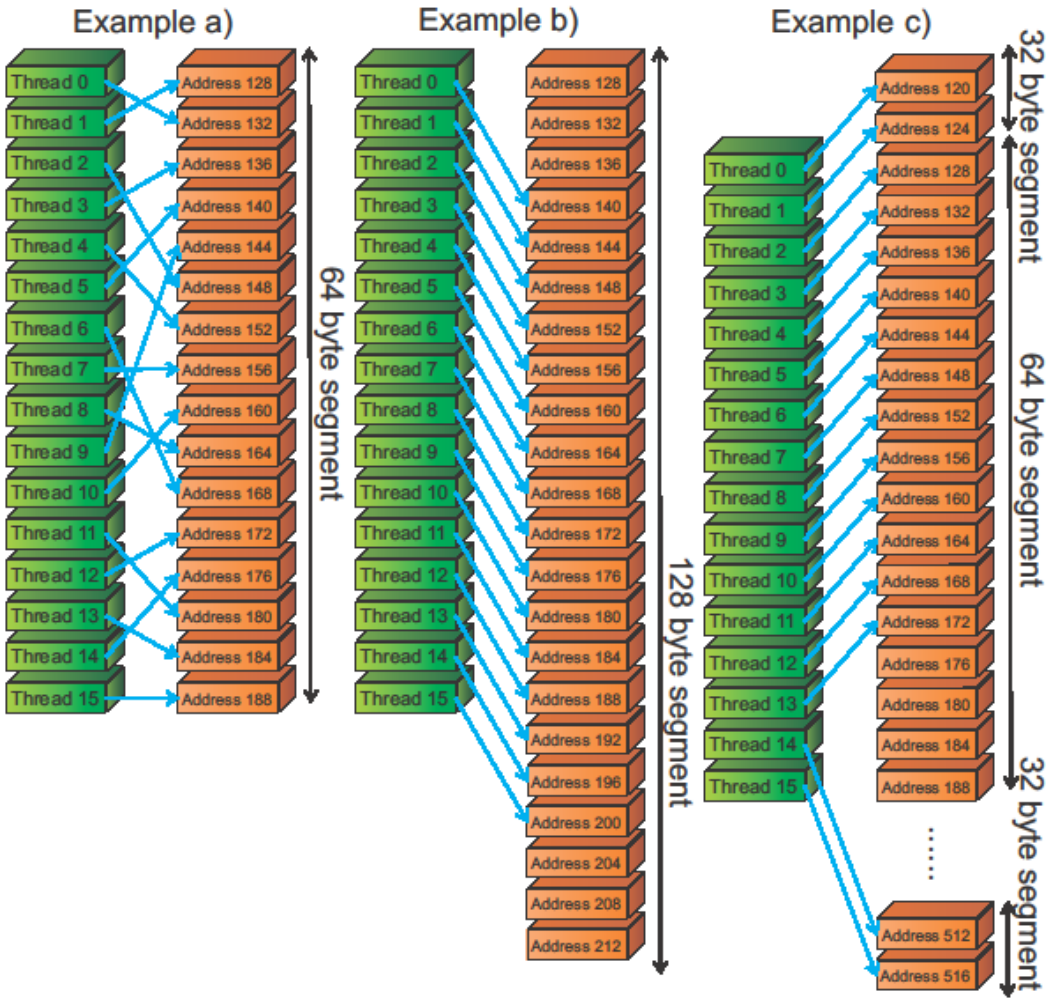
memory (OpenCL terminology)	Size	Remarks
global memory	1 GB	not cached, 100 Gb/sec
constant memory	64 kB	cached
local memory	16 kB / multiprocessor	very fast, when used with correct pattern as fast as registers
private memory	-	part of global memory, considered slow

Memory Guidelines

- **the following guidelines refer to the GT200b chip**

  (for different chips the optimal memory usage might differ)
  - **store constants in constant memory wherever possible to benefit from the cache**
  - **try not to use too many intermediate variables to save register space, better recalculate values**
  - **try not to exceed the register limit, swapping registers to private memory is painful**
  - **avoid private memory where possible**
  - **use local memory where possible**
  - **big datasets must be stored in global memory anyway, try to realize a streaming access,follow coalescing rules (see next sheet), and try to access the data only once**

NVIDIA Coalescing Rules



Application of coalescing rules:

a) Results in one 64-byte access

(Random access can still be coalesced if it is restricted to one segment. Cards with compute capability below 1.2 would issue 16 memory requests.)

b) Results in one 128-byte access

(The whole 128-byte segment is read even though only 64 byte of the data is used.)

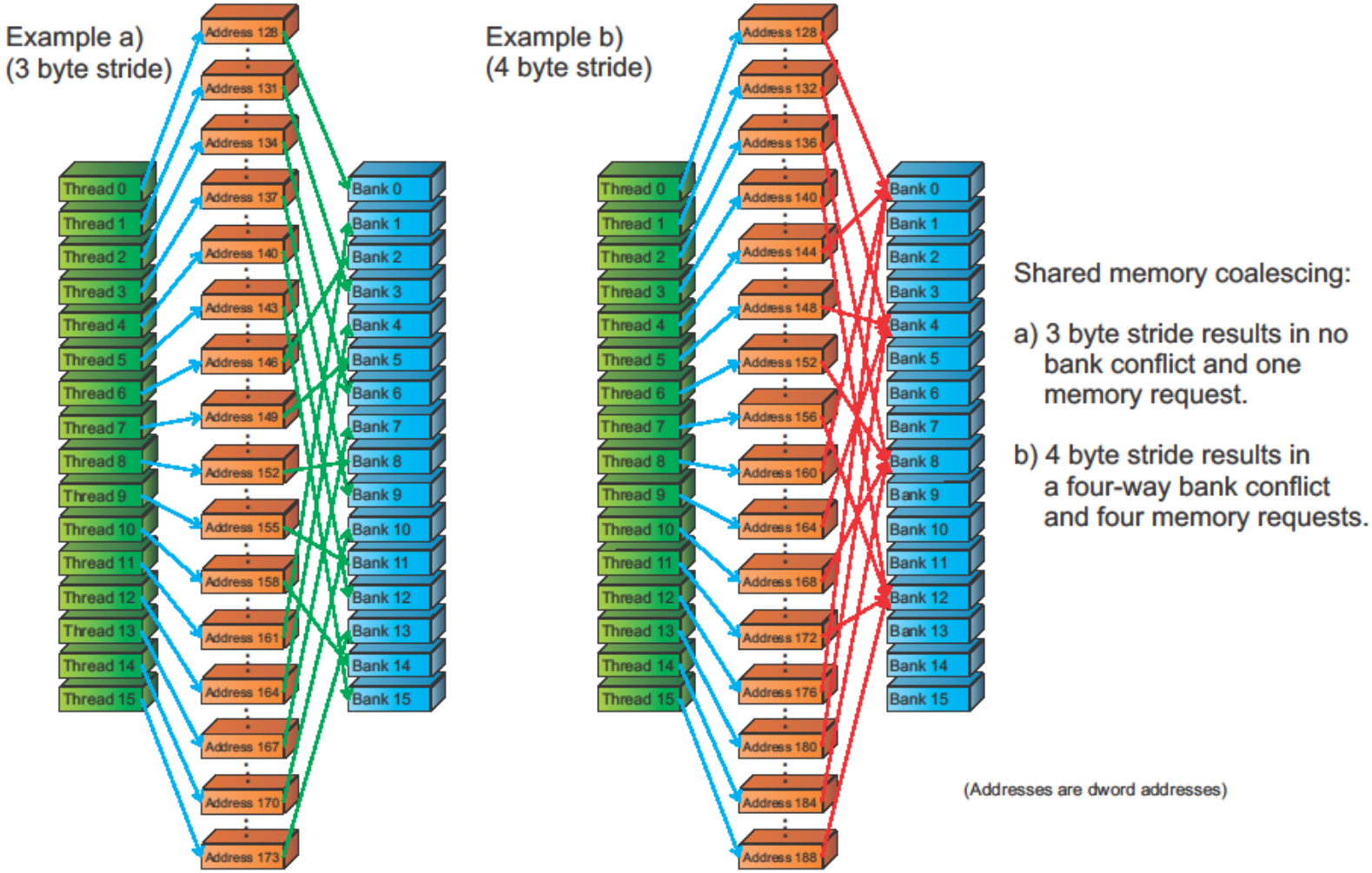
c) Results in one 64-byte and two 32-byte accesses

(The upper two segments are within 128 bytes but cannot be coalesced as they are misaligned)

Analysing Coalescing Rules

- example A resembles an aligned vector fetch with a swizzle
- example B is an unaligned vector fetch
- both access patterns commonly appear in SIMD applications
- as for vector-processors random gathers cause problems
- the vector-processor-like nature of the NVIDIA-GPU reappears

NVIDIA Local Memory Coalescing



Memory Consistency

- **GPU memory consistency differs from what one is used from CPUs**
  - **load / store order for global memory is not preseved among different compute-units**
  - **the correct order can be ensured for threads within one particular compute-unit using synchronization / memory fences**
  - **global memory coherence is only ensured after a kernel call is finished (when the next kernel starts, memory is consitent)**
  - **there is no way to circumvent this!!!**
  - **HOWEVER: different compute units can be synchronized using atomic operations**
- **As inter work-group synchronization is very expensive, try to divide the problem in small parts that are handled independently**

From theory to application

- **tasks required to execute an OpenCL kernel**
  - **create the OpenCL context**
    - **query devices**
    - **choice device**
    - **etc.**
  - **load the kernel source code (usually from a file)**
  - **compile the OpenCL kernel**
  - **transfer the source data to the device**
  - **define the execution configuration**
  - **execute the OpenCL kernel**
  - **fetch the result from the device**
  - **uninitialize the OpenCL context**
- **third party libraries encapsulate these tasks**
- **we will look at the OpenCL runtime functions in detail**

OpenCL Runtime

- **OpenCL is plain C currently**
- **Upcoming C++ interface for the host**
- **C++ for the device might appear in future versions**
- **The whole runtime documentation can be found at: http://www.khronos.org/opencl/**
- **The basic functions to create first examples will be presented in the lecture**
- **Some features will just be mentioned, have a look at the documentation to see how to use them!!!**

OpenCL Runtime Functions (Context)

```
//Set OpenCL platform, choose between different implementations / versions
cl_int clGetPlatformIDs (cl_uint num_entries, cl_platform_id *platforms,
   cl_uint *num_platforms)

//Get List of Devices available in the current platform
cl_int clGetDeviceIDs (cl_platform_id platform, cl_device_type device_type,
   cl_uint num_entries, cl_device_id *devices, cl_uint *num_devices)

//Get Information about an OpenCL device
cl_int clGetDeviceInfo (cl_device_id device, cl_device_info param_name,
   size_t param_value_size, void *param_value, size_t *param_value_size_ret)

//Create OpenCL context for a platform / device combination
cl_context clCreateContext (const cl_context_properties *properties,          cl_uint
num_devices, const cl_device_id *devices, void              (*pfn_notify)(const char
*errinfo, const void *private_info, size_t cb,      void *user_data), void *user_data, cl_int
*errcode_ret)
```

Runtime Functions (Queues / Memory)

//Create a command queue kernels will be assigned to later

cl_command_queue clCreateCommandQueue (cl_context *context,* cl_device_id *device,* cl_command_queue_properties *properties,* cl_int **errcode_ret)

//Allocate memory on the device

cl_mem clCreateBuffer (cl_context *context,* cl_mem_flags *flags,* size_t *size,* void **host_ptr,* cl_int **errcode_ret)

- **flags regulate read / write access for kernels**
- **host memory can be defined as storage, however the OpenCL runtime is allowed to cache host memory in device memory during kernel execution**

- **device memory can be allocated as buffer or as image**
- **buffers are plain memory segments accessible by pointers**
- **images are 2/3-dimensional objects (textures / frame buffers) accessed by special functions, storage format opaque for user**

Runtime Functions (Memory)

```
//Read memory from device to host
cl_int clEnqueueReadBuffer (cl_command_queue command_queue, cl_mem
            buffer, cl_bool blocking_read, size_t offset, size_t cb, void *ptr,
            cl_uint num_events_in_wait_list, const cl_event *event_wait_list,  cl_event
*event)

//Write to device memory from host
cl_int clEnqueueWriteBuffer (…)                    //same parameters
```

- reads / writes can be blocking / non-blocking
  (Blocking commands are enqueued, the host process wait for the command to finish before it continues. Non blocking commands do not pause host execution)
- the “event” parameters force the operation to start only after specified events occurred on the device
- events occur for example when kernel executions finish, they are used for synchronization

Runtime Functions (Kernel creation)

```
//Load a program from a string
cl_program clCreateProgramWithSource (cl_context context, cl_uint count,
    const char **strings, const size_t *lengths, cl_int *errcode_ret)

//Compile the program
cl_int clBuildProgram (cl_program program, cl_uint num_devices,
const cl_device_id *device_list, const char *options, void (*pfn_notify)(cl_program,
    void *user_data), void *user_data)

//Create an executable kernel out of a kernel function in the compiled program
cl_kernel clCreateKernel (cl_program program, const char *kernel_name,
    cl_int *errcode_ret)

//Define kernel parameters for execution
cl_int clSetKernelArg (cl_kernel kernel, cl_uint arg_index, size_t arg_size,
    const void *arg_value)
```

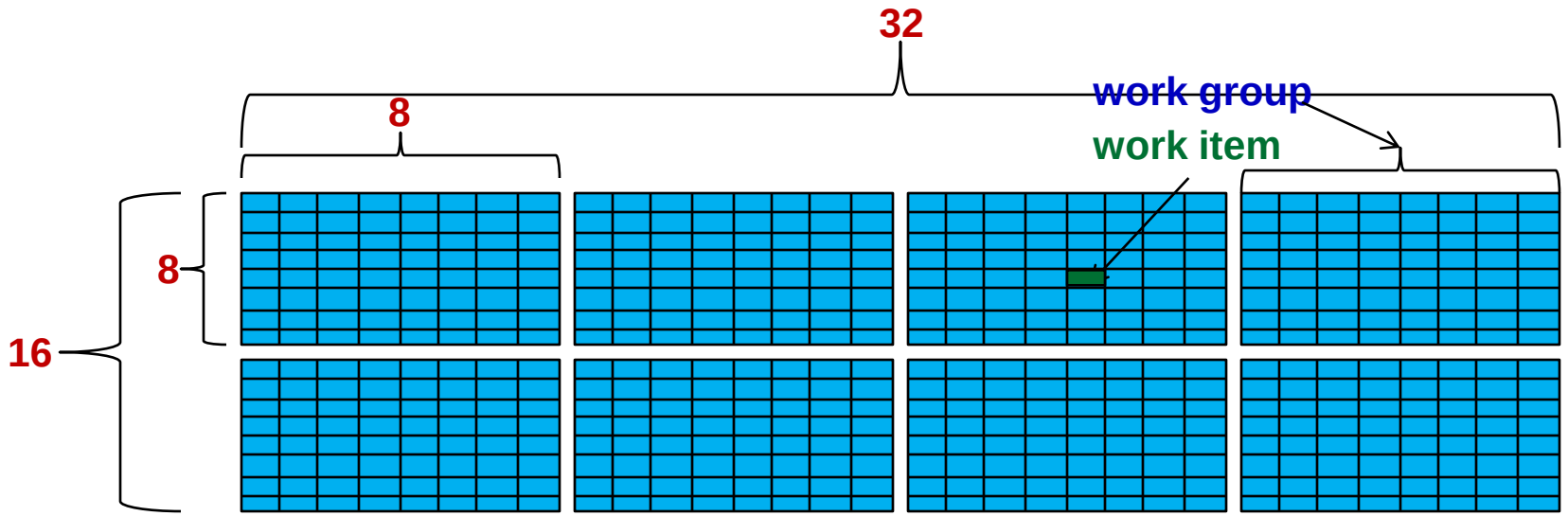
Runtime Functions (Kernel execution)

```
//Load a program from a string
cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,
    cl_kernel kernel, cl_uint work_dim, const size_t *global_work_offset,
    const size_t *global_work_size, const size_t *local_work_size,
    cl_uint num_events_in_wait_list, const cl_event *event_wait_list,
    cl_event *event)
```

- work_dim is the dimensionality of the work groups
- global_work_size and local_work_size are the number of work items globally and in a work group respectively.
- these parameters are array ranging from 0 to work_dim – 1 to allow multi-dimensional work-groups
- the local_work_size parameters must evenly divide the global_work_size parameters

Runtime Functions (Kernel execution)

- **examples for kernel execution configurations**
  - **simple 1-dimensional example:**
    2 work groups of 16 work_items each
    → work_dim = 1, local_work_size = (16), global_work_size = (32)
  - **more complex 2-dimensional example:**
  - **4*2 work groups of 8*8 work_items each**
  - **→ work_dim = 1, local_work_size=(4,4), global_work_size = (32, 16)**



Memory Access by Kernels

- **Memory objects can be accesses by kernels**
- **New keywords for kernel parameters**
  - **“__global”** **//pointer to global memory**
  - **“__constant”** **//pointer to constant memory**

  - **Assigning a buffer object to a __global variable will result in a pointer to the address**

    //Used for images not buffers, see reference for details
  - **“__read_only”**
  - **“__write_only”**

OpenCL First Example, Vector Addition

„addvector.cl“:

```
__kernel void addVector(__global int *src_1, __global int *src_2, __global int *dst,
    cl_int vector_size)
{
    for (int i = get_global_id(0);i < vector_size;i += get_global_size(0))
    {
            dst [i] = src_1[i] + src_2[i];
    }
}
```

- global id and size can be obtained by get_global_id / …_size
- consider how the work is distributed amont the threads

  (consecutive threads access data in adjacent memory addresses,
  → following the coalescing rules)

OpenCL First Example, Vector Addition

„addvector.cpp“:

```
cl_int ocl_error, num_platforms = 1, num_devices = 1, vector_size= 1024;
clGetPlatformIDs(num_platforms, &platforms NULL);
clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, num_devices, &device,
        NULL);
cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL,
        &ocl_error);
cl_command_queue command_queue =      clCreateCommandQueue(context,
device, NULL, &ocl_error);
cl_program program = clCreateProgramWithSource(context, 1, (const     char**)
&sourceCode, NULL, &ocl_error);
clBuildProgram(program, 1, &device, „-cl-mad-enable“, NULL,  NULL);
cl_kernel kernel = clCreateKernel(program, "addVector", &ocl_error);
cl_mem vec1 = clCreateBuffer(context, CL_MEM_READ_WRITE,
        vector_size* sizeof(int), NULL, &ocl_error);
```

OpenCL First Example, Vector Addition

```
clEnqueueWriteBuffer(command_queue, vec1, CL_FALSE, 0, vector_size*
        sizeof(int), host_vector_1, 0, NULL, NULL);
clSetKernelArg(kernel, 0, sizeof(cl_mem), &vec1);
//…
//....    Vector 2, Destination Memory
//…
clSetKernelArg(kernel, 3, sizeof(cl_int), vector_size);
size_t local_size = 8;
size_t global_size = 32;
clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,   &global_size,
&local_size, 0, NULL, NULL;
clEnqueueReadBuffer(command_queue, vec_result, CL_TRUE, 0,
        vector_size * sizeof(int), host_vector[2], 0, NULL, NULL);
```

OpenCL First Example, Vector Addition

- Vector addition admittedly very simple example
- OpenCL overhead for creating the kernel etc. seems huge
- Extended / documented source code available on the lecture homepage
- Can now be easily extended to more complex kernels (will be done in the turorials)

OpenCL Optimizations

— Copyright ©, Goethe Uni, alle Rechte vorbehalten

Computing Device Utilization

- **Using OpenCL the primary computing device is no longer the CPU (still the CPU can contribute to the total computing power or the CPU can be the OpenCL computing device on its own)**
- **Therefore the main objective is to keep the OpenCL computing device as busy as possible**
- **This includes two objectives**
  - **Firstly: Ensure the device is totally utilized during kernel execution (This includes the prevention of latencies due to memory access as well as the utilization of all threads of the vector-like processor)**
  - **Secondly: Make sure that there is no delay between kernel executions**

Computing Device Utilization

- **We will now discuss some criteria that should be fulfilled to ensure both of the previous requisitions**
- **Bad device utilization during kernel execution mostly originates from:**
  - **Memory latencies when the device waits for data from global memory**
  - **Non coalesced memory access where multiple memory accesses have to be issued by the threads instead of only a single access**
  - **work-group serialization: As only one instruction decoder is present, performance decreases when different work-items follow different branches in conditional code**

Memory Latencies

- **OpenCL devices have an integrated feature to hide memory latencies**
- **The number of threads started greatly exceeds the number of threads that can be executed in parallel**
- **For each instruction cycle without any overhead the scheduler selects threads that are ready to execute**
  - **→Try to use a large number of parallel threads**
- **OpenCL devices do not necessarily have a general purpose cache**
  - **→ Random memory access is very expensive, streaming access is even more important than for usual CPUs**

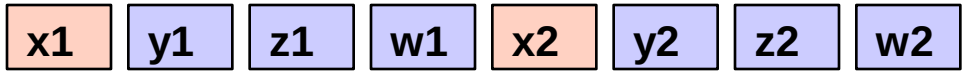
Memory Coalescing

- **Streaming access can usually be achieved by following coalescing rules**
- **Often data structures must be changed to allow for coalescing**
  - **→Often arrays of structures should be replaced by structures of arrays**

Memory Coalescing Data Structures

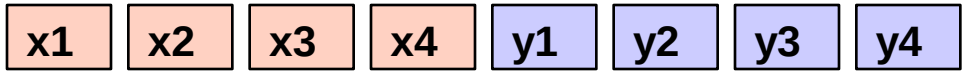
- **Consider the following examples**

```
struct int4{int x, y, z, w};
int4 data[thread_count];
kernel example1
{
   data[thread_id].x++;
   data[thread_id].y--;
}
```



Access to x[thread_id] skips 3 out of 4 memory addresses

```
int x[thread_count], y[thread_count], z[thread_count], …
kernel example2
{
   x[thread_id]++;
   y[thread_id]--;
}
```



Access to x[thread_id] affects a continous memory segment

Example 1 requires 4 times the amount of accesses example 2 needs (for a thread count of 4), the ratio is worse for higher thread counts

Random memory access

- **Many algorithms have random access schemes restricted to a bounded memory segment**
- **If this segment fits in local memory it can be cached**
- **Random memory access to local memory is almost as fast as sequential access (e.g. except for the possibility of bank conflicts for NVIDIA chips)**
- **Caching to local memory can be performed in a coalesced way**

Work-group serialization

- **Consider the following code**

```
int algorithm(data_structure& data, bool mode)
{
    ....
    if (mode)
        data.x++;
    else
        data.x--;
    …
}

kernel example(data_structure* data)
{
    if (data[thread_id].y > 1)
        algorithm(data[thread_id], true);
    else
        algorithm(data[thread_id], false);
}
```

Work-group serialization

- **Analyzing the example**
  - **The check for data[thread_id].y > 1 might lead to different results and thus different branches for different threads**
  - **As only one instruction decoder is present, both branches have to be executed one after another**
  - **Only if the result is the same for all work-items in a work group, execution is restricted to a single branch**
  - **This problem is called work-group serialization**
  - **Except for the different behavior depending on the mode flag both branches involve identical code**
  - **In the given example it will possibly halve the performance**
  - **This might become worse with more complex branches**

  → conditional execution where branches contain complex code should be avoided

Work-group serialization

- **Improved version of above example**

```
int algorithm(data_structure& data, bool mode)
{
    ....
    if (mode)
        data.x++;
    else
        data.x--;
    …
}

kernel example(data_structure* data)
{
    algorithm(data[thread_id], data[thread_id] > 1);
}
```

This clearly gives the same result, but the outer branch was removed

Work-group serialization

- **Even more improved version of above example**

```
int algorithm(data_structure& data, bool mode)
{
    ....
    data.x += mode;
    …
}

kernel example(data_structure* data)
{
    algorithm(data[thread_id], data[thread_id] > 1);
}
```

Often conditions can be exchanged by other statements without branches

Work-group serialization

- **Allowed branches**

  At the end it shall be noted, that conditions where different work-groups execute different branches do not influence the performance

```
kernel example()
{
   if (work_group_id % 2)
      algorithm1();
   else
      algorithm2();
}
```

  As different work groups do not execute simultaneously on one compute unit, the restriction to one instruction decoder is not relevant

Common Source

- **Common Source refers to a programming paradigm simplifying application development instead of an OpenCL performance optimization**
- **If different versions of an algorithm shall be created for OpenCL and another platform, it is desirable to stay with one common source code**
- **This is possible by placing the actual algorithm in a function that is included twice, in the OpenCL file and another C++ file**
- **Only two different wrapper functions are required and must be maintained**
- **Changes to the algorithm itself are done only once**

Common Source Example

include.cpp

```
int algorithm(data_structure& data)
{
    ....
}
```

opencl.cl

```
#include include.cpp
kernel void example(data_structure* data)
{
    algorithm(data[thread_id]);
}
```

other_version.cpp

```
#include include.cpp
void example(data_structure* data)
{
    for (i = 0;i < DATA_SIZE;i++) algorithm(data[i]);
}
```

Local Variable Usage

- **Always try not to use too many local variables**
- **Reordering the code might result in less local variables required in parallel**

Local Variable Example

- **Consider the following example**
  (The first loop can be vectorized on a CPU)

```
for(o = 0;o < 6;o++)
{
    for(p = 0;p < 4;p++)
    {
            AV[p] += aarray[o] * barray[o].bgr[p];
            CV[p] += carray[o] * barray[o].bgr[p];
    }
}
for(p = 0;p < 3;p++)
{
    DIFFERENTIALCOMPAREC(p);
}
```

Local Variable Example

- **Reordered code of the same example**

```
for(p = 0;p < 3;p++)
{
    for(o = 0;o < 6;o++)
    {
            AV[p] += aarray[o] * barray[o].bgr[p];
            CV[p] += carray[o] * barray[o].bgr[p];
    }
    DIFFERENTIALCOMPAREC(p);
}
```

Local Variable Benchmarks

- **Different versions show different performance on CPU / GPU**

Renderer	Time	Speedup
CPU (Code 1)	98.718	1,000
CPU (Code 2)	109.476	0,902
GPU (Code 1)	50.254	1,000
GPU (Code 2)	15.882	3,164

Real World HPC application

— Copyright ©, Goethe Uni, alle Rechte vorbehalten

ALICE HLT TPC Online Tracker

- **We will now look at a real world HPC application that was portet to GPGPU**
- **The ALICE HLT Online Tracker is responsible for real-time track reconstruction for the ALICE experiment at LHC (CERN)**
- **Problems emerging when porting the code and solutions will be presented**

ALICE HLT TPC Online Tracker

- **First question: Which language to choose**
  - **Some facts:**
  - **A C++ tracker CPU tracker code already existed**
  - **It was desired to have a common code base for CPU / GPU**
  - **The CPU code relies on AliROOT (which relies on ROOT) and therefore C++ was obligatory**

- **Using CUDA for the GPU tracker seemed the appropriate solution (OpenCL was also far away from a stable release)**

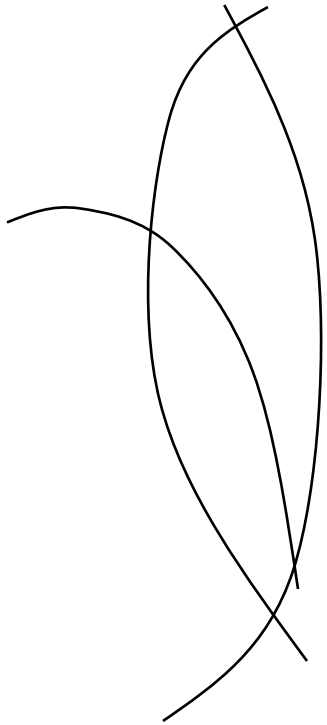
Track Reconstruction

- **What is tracking**
  - **In the LHC particles (protons or heavy-ions) are accelerated to yet unreach energies and afterwards collide in the center of the detectors.**
  - **In the collision (imagine an explosion) lots of daughter particles are produced that shall be analysed**
  - **Measuring the trajectories of the particles is a crucial part of this analyzation**

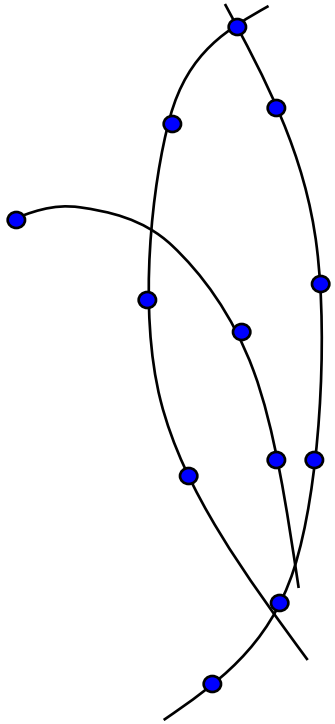
- **How to measure the trajectories**
  - **It is impossible to measure the particle trajectory directly**
  - **Instead only some discrete points of the trajectories can be measured, these are called clusters**
  - **The crucial part now, is to regain the trajectories from these clusters**

Track Reconstruction

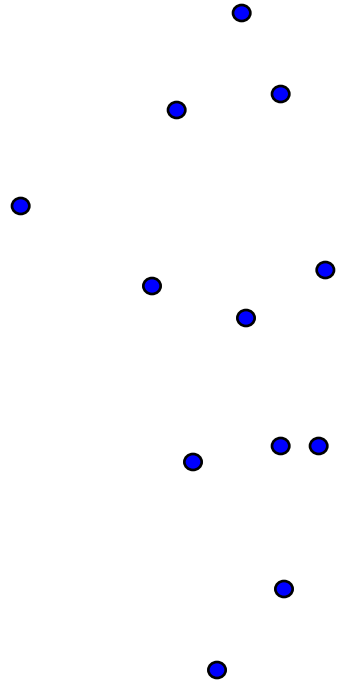
trajectories



trajectories and cluster



only the clusters



The third graph shows the input data for the tracking algorithm.
It is now a combinatorial challenge to restore the original tracks.

Tracking Algorithm

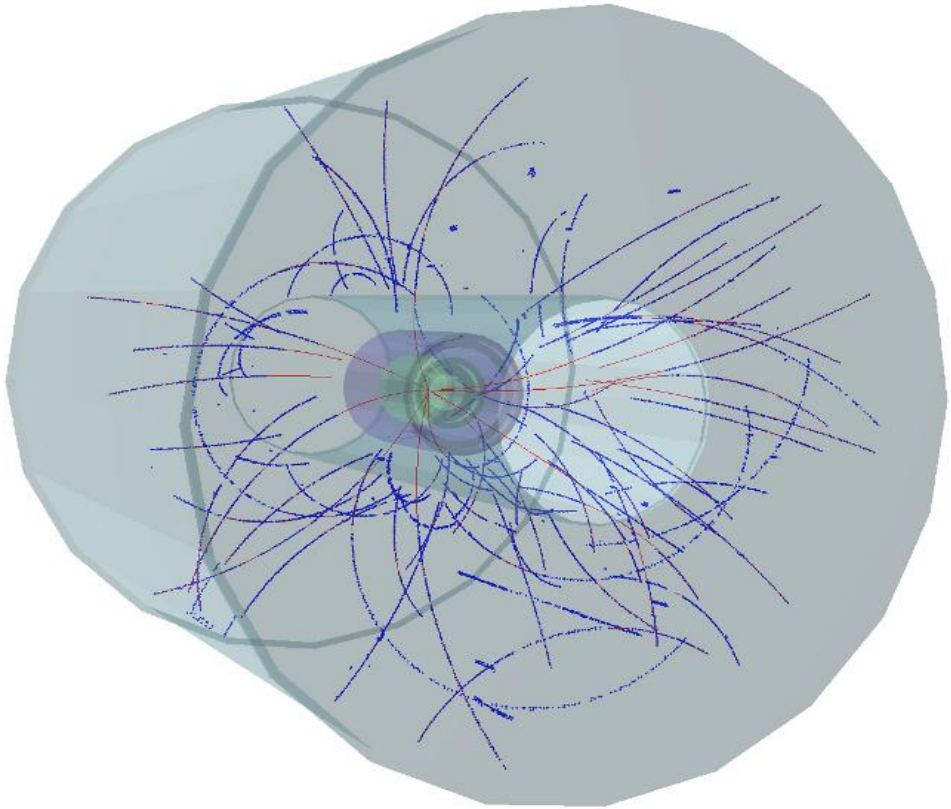
- **How to extract trajectories from 3-dimensional space-points**
  - **The CPU tracking algorithm was originally designed to run on parallel computer**
  - **The Algorithm first determines some track candidates (tracklets), by fitting straight lines to 3 adjacent clusters**
  - **These candidates can then be extrapolated, and new clusters close to the extrapolated tracks can be added**
  - **Extrapolation and fitting for different candidates can be performed in parallel**
  - **This seems ideal for massive parallel computing, the more tracks the better**

How to fit / extrapolate

- **Fitting and extrapolating in a free 3-dimensional space is complicated**
- **The following assumption is made to simplify and speed up the tracking**
  - **As tracks are supposed to originate from the interaction point fitting and extrapolation is done in radial direction only**
  - **radial space coordinates are discrete with 159 values possible (called rows)**
  - **angular coordinates in every row are continuous**
  - **Extrapolation is done in two steps: upwards and downwards (to the next / to the previous row) to extend the initial candidate in both directions**
  - **In each extrapolation step the continuous angular coordinates for the next / previous row are calculated**

Tracking Proton Events

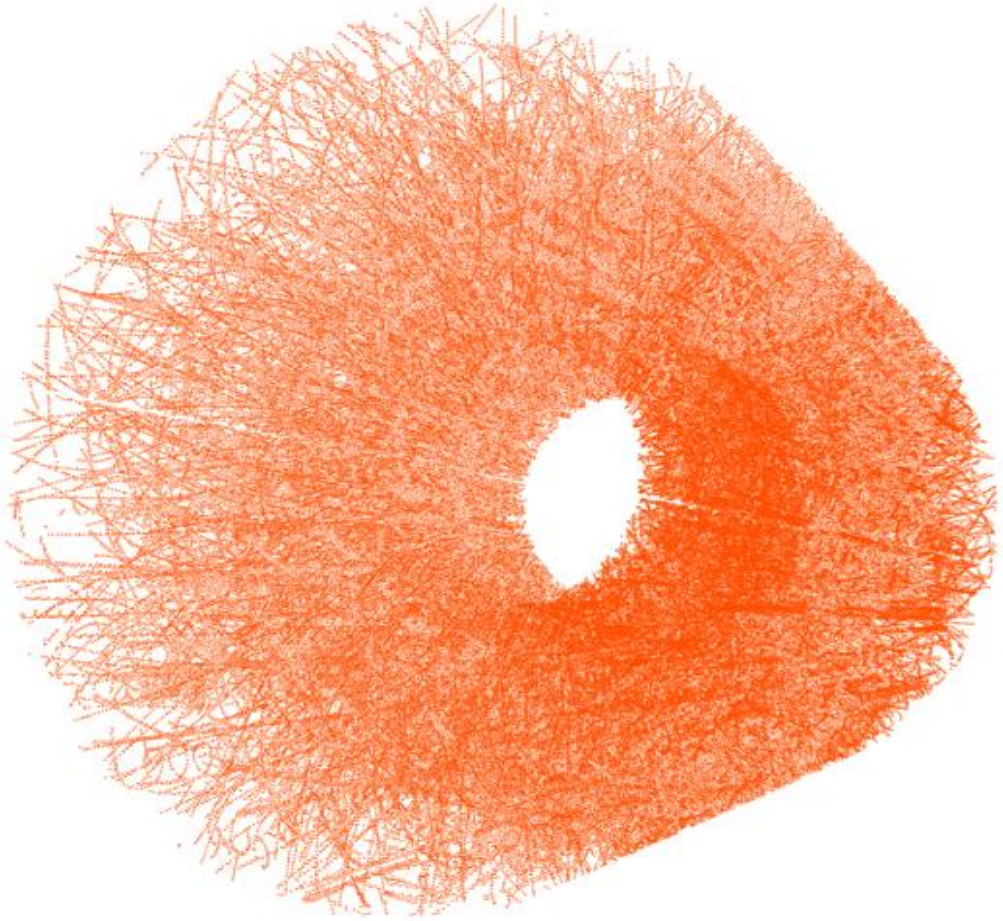
- **Real Proton-Proton collision at ALICE**



Does not look so complex?

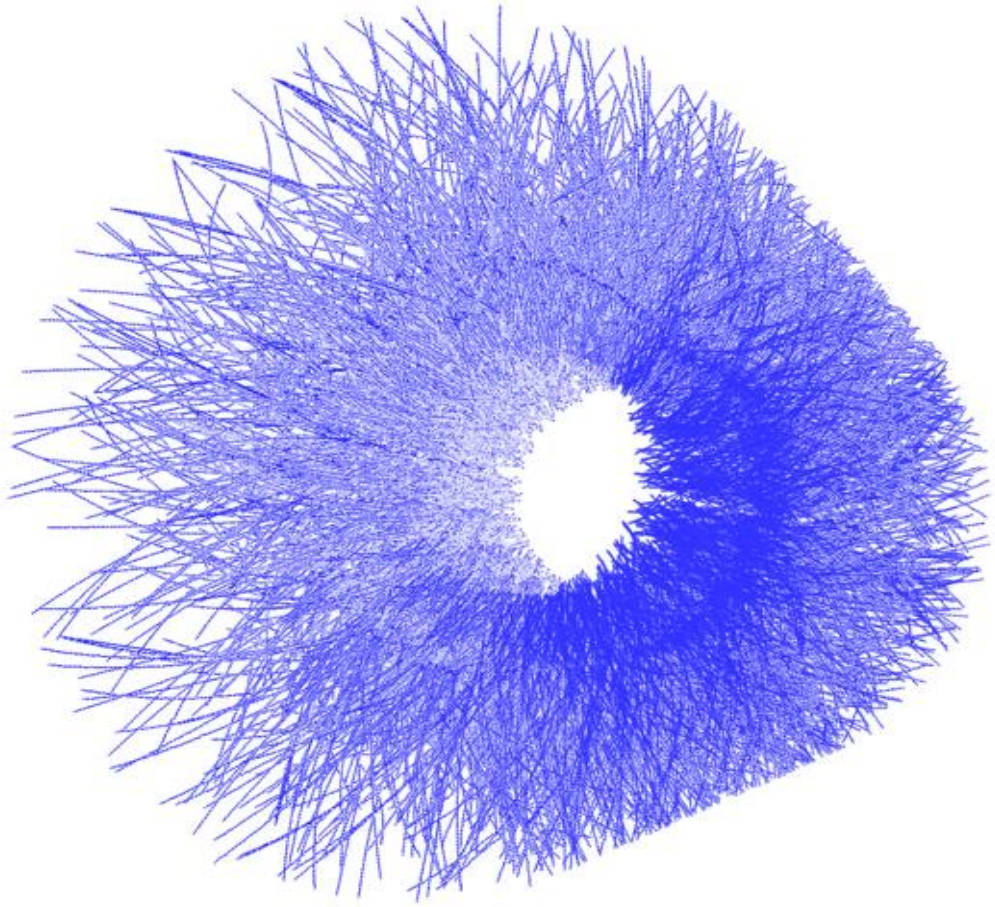
Tracking Heavy-Ion Events

- **Simulated lead-lead collision:**



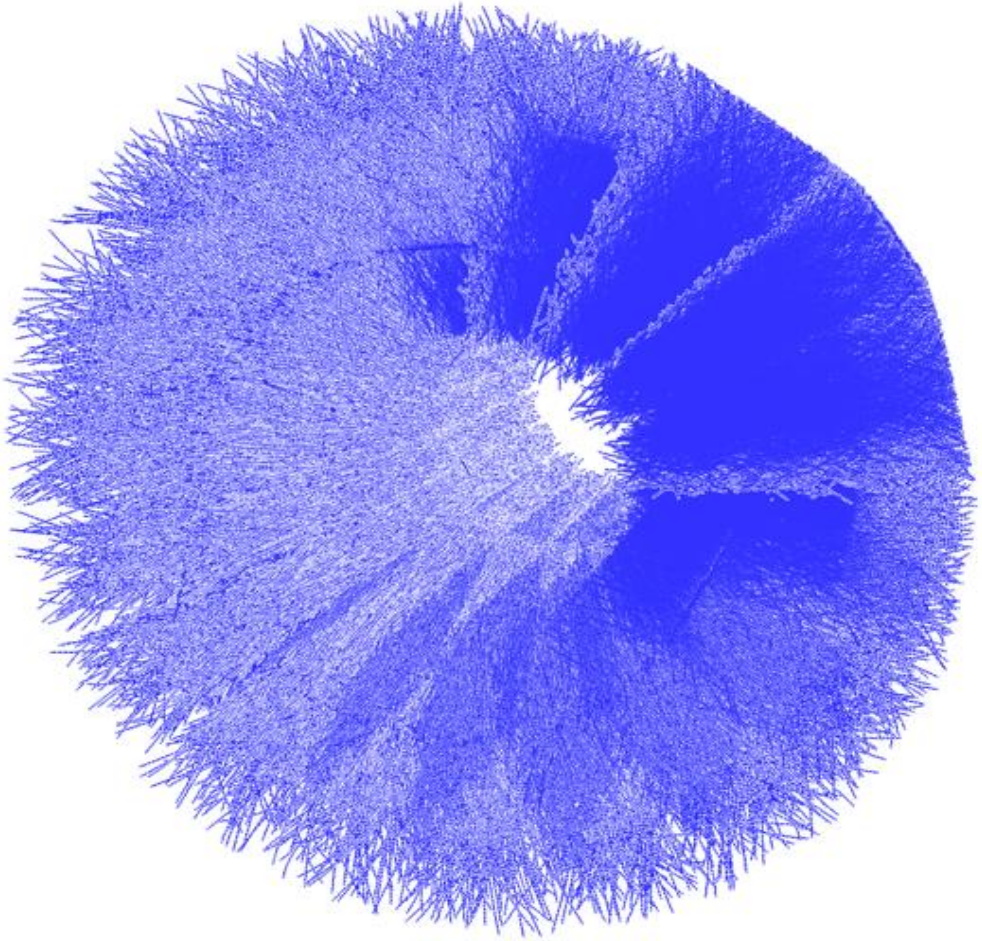
Tracking Heavy-Ion Events

- **Tracks reconstructed in lead-lead collision**



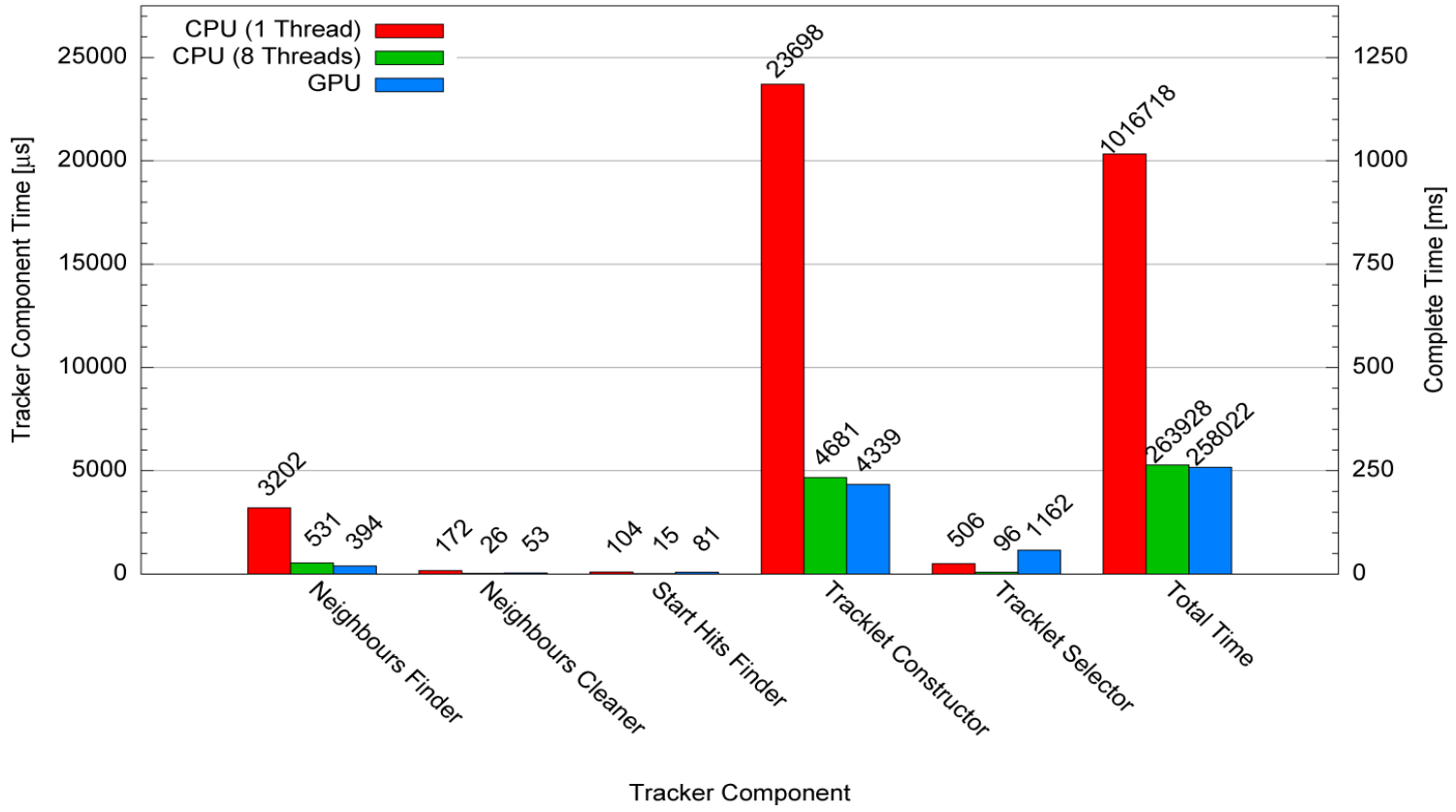
Tracking Heavy-Ion Events

- **To make things even worse: central pb-pb**



Tracking on GPU

- **Initial GPU tracker performance (first working port GTX285 v.s. Nehalem 3.2 GHz)**



Performance analysis

- **GPU tracker is not faster than a state-of-the-art CPU**
- **In contrast the speed-ups presented by NVIDIA this looks frustrating**
- **BUT: This is an optimized CPU application, and no 10 year old fortran code, so no speedup of 10+ should be expected**
- **FURTHER: This was the very first try, so let us examine how to improve the tracker**

Performance analysis

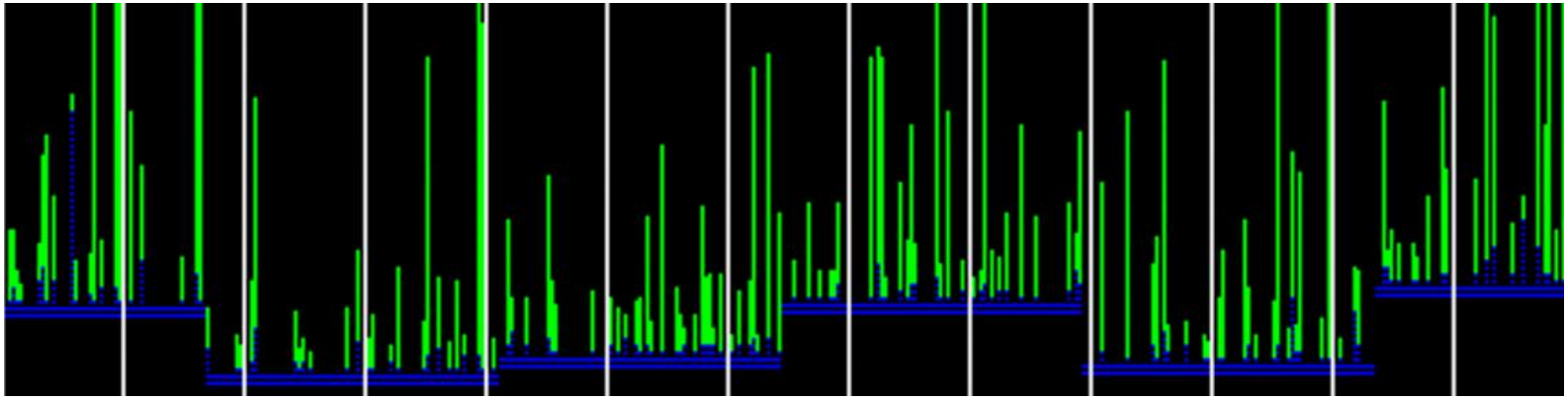
- **Remember GPU paradigm:**

  Keep all GPU processors active
  - **Ensure a high workload during kernel execution**
  - **Make sure that there is always a kernel in the queue that can be executed**

- **We will now apply these criteria to the tracker**

Initial GPU Utilization

Active GPU Threads for the First Implementation



Active GPU Threads: 19%

Colors represent Tracklet Constructor Steps:
Black: Idle
Blue: Track Fit
Green: Track Extrapolation

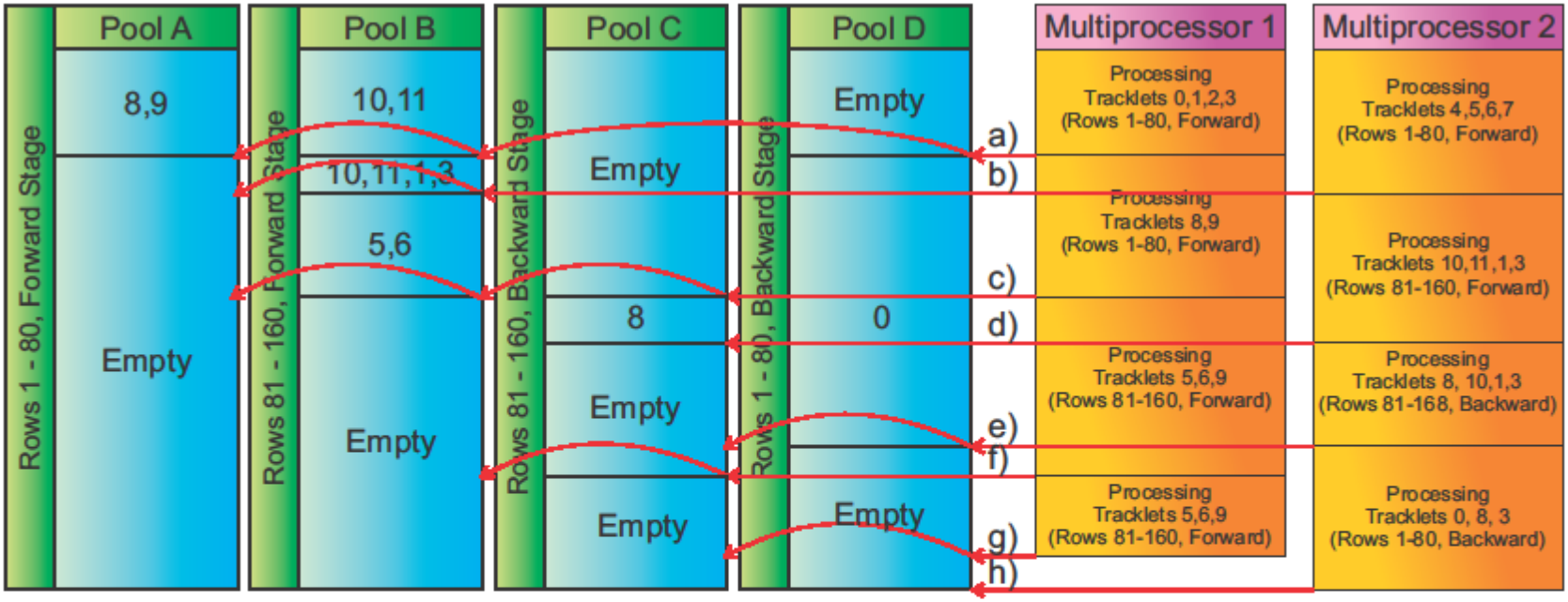
Problem with First Implementation

- The tracks greatly differ in length
- Some candidates are immediately dropped
- Some candidates are extended to up to 100 clusters
- In the first implementation every track candidate was handled by a different thread
- When some threads in a work-group have already dropped their track candidate while others have not, they are idling, because only one single instruction decoder is present
- It is better to stop the tracking in between, and exchange dropped candidates with new ones

Fixing GPU Utilization / Scheduling

- **Idea: Introduce Tracklet Pools**
  - **The set of 159 rows is divided into n row-blocks**
  - **Tracklets are fitted and extrapolated for rows in one row-block only**
  - **Afterwards the extrapolation is interrupted**
  - **All tracklets that are still active (the track has not ended yet) are stored in the tracklet pool of the next row block. However, if the extrapolation is finished, the track is stored**
  - **The threads then fetch new tracklets from a tracklet pool**
  - **This way, after every interrupt, the active tracklets are redistributed among the threads, and no threads have to idle when their track has ended**

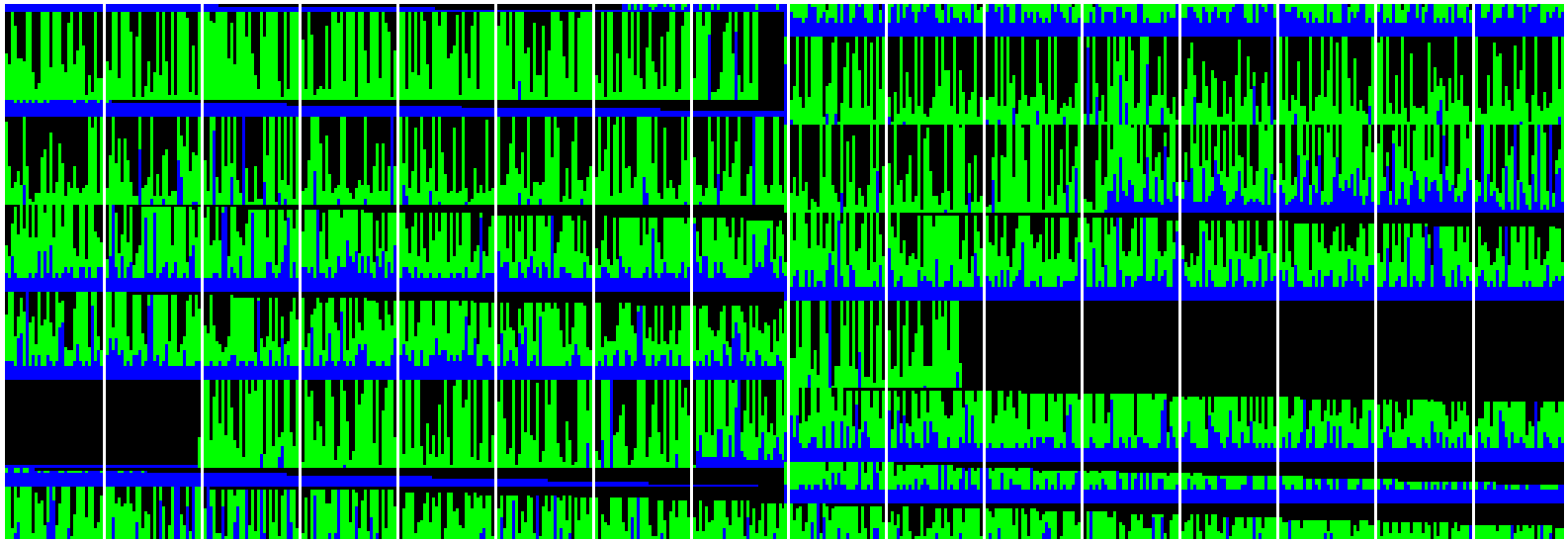
Fixing GPU Utilization / Scheduling



a) Tracklets 0 stored to pool D, tracklets 1,3 stored to pool B, tracklet 2 dropped, tracklets 8,9 read from pool A
b) Tracklets 5,6 stored to pool B, tracklets 4,7 dropped, no tracklets found in pool A, tracklets 10,11,1,3 read from pool B
c) Tracklet 8 stored to pool C, tracklet 9 stored to pool B, no tracklets found in pool A, tracklets 5,6,9 read from pool B
d) Tracklet 10,1,3 stored to pool C, tracklet 11 dropped, no tracklets found in pool B, tracklet 8,10,1,3 read from pool C
e) Tracklets 8,3 stored to pool D, tracklets 10,1 dropped, no tracklets found in pool C, tracklets 0,8,3 read from pool D
f) Tracklets 5,6,9 stored to pool C, no tracklets in pool B, tracklets 5,6,9 read from pool C
g) Tracklets 5,6,9 dropped, no tracklets found in pool C, no tracklets found in pool D
h) Tracklets 0,8,3 dropped, no tracklets found in pool D

Improved GPU Utilization

Active GPU Threads using Dynamic Scheduling



Active GPU Threads: 62%

Colors represent Tracklet Constructor Steps:
Black: Idle
Blue: Track Fit
Green: Track Extrapolation

Trackled Scheduling Summary

- **Tracklet Scheduling**
  - **Before many threads were idling after a track has ended, but other threads in the same work-group were still extrapolating other tracks**
  - **This could be overcome by introducing a tracklet scheduler, which redistributes the tracklets among the threads**

- **Tracklet Scheduling Performance**
  - **Without the scheduling only 19% of the threads were active**
  - **Occupancy rose to 62% using the scheduler**

  - **GPU utilization increased by 226%**

Initialization / Output

- **Tracker Initialization / Tracklet Output**
  - **Some steps of the tracking algorithm cannot be easily ported to run on GPU**
  - **The cluster data is fetched from the network and the final tracks are sent to other nodes in the network**
  - **The initial data received is converted to a format suitable for the tracking algorithm**
  - **The conversion process touches every bit only one time rendering a transfer to the GPU non-performant**
  - **The same holds for the conversion of the tracklet output data**

- **Running Initialization, Tracking and Output in a loop iterating over events is not optimal**
  - **The GPU is idling during initialization and output wasting a lot of computational performance**

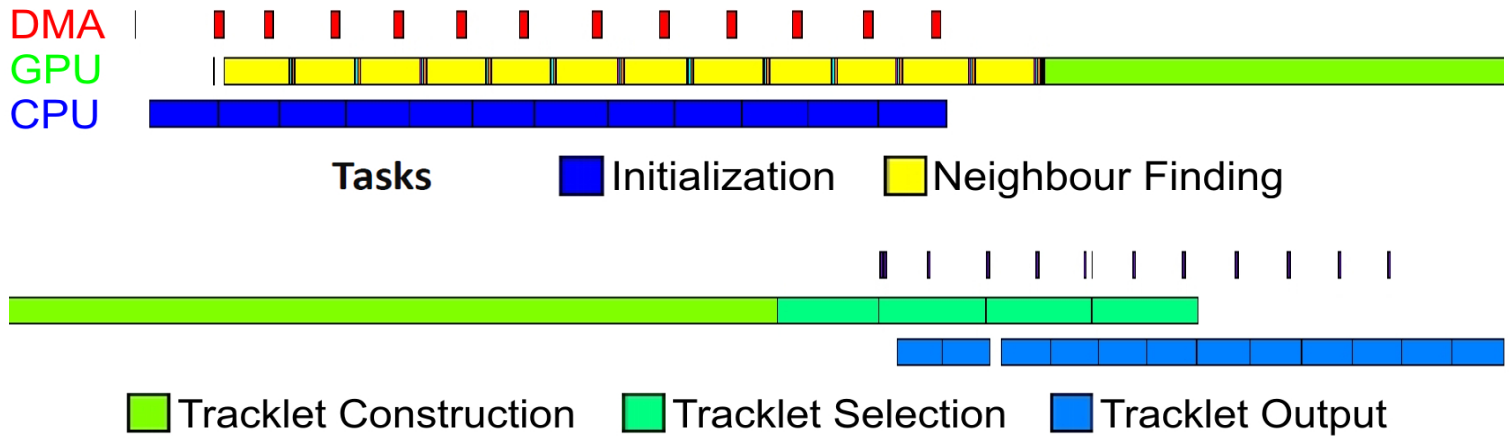
CPU / GPU pipelining

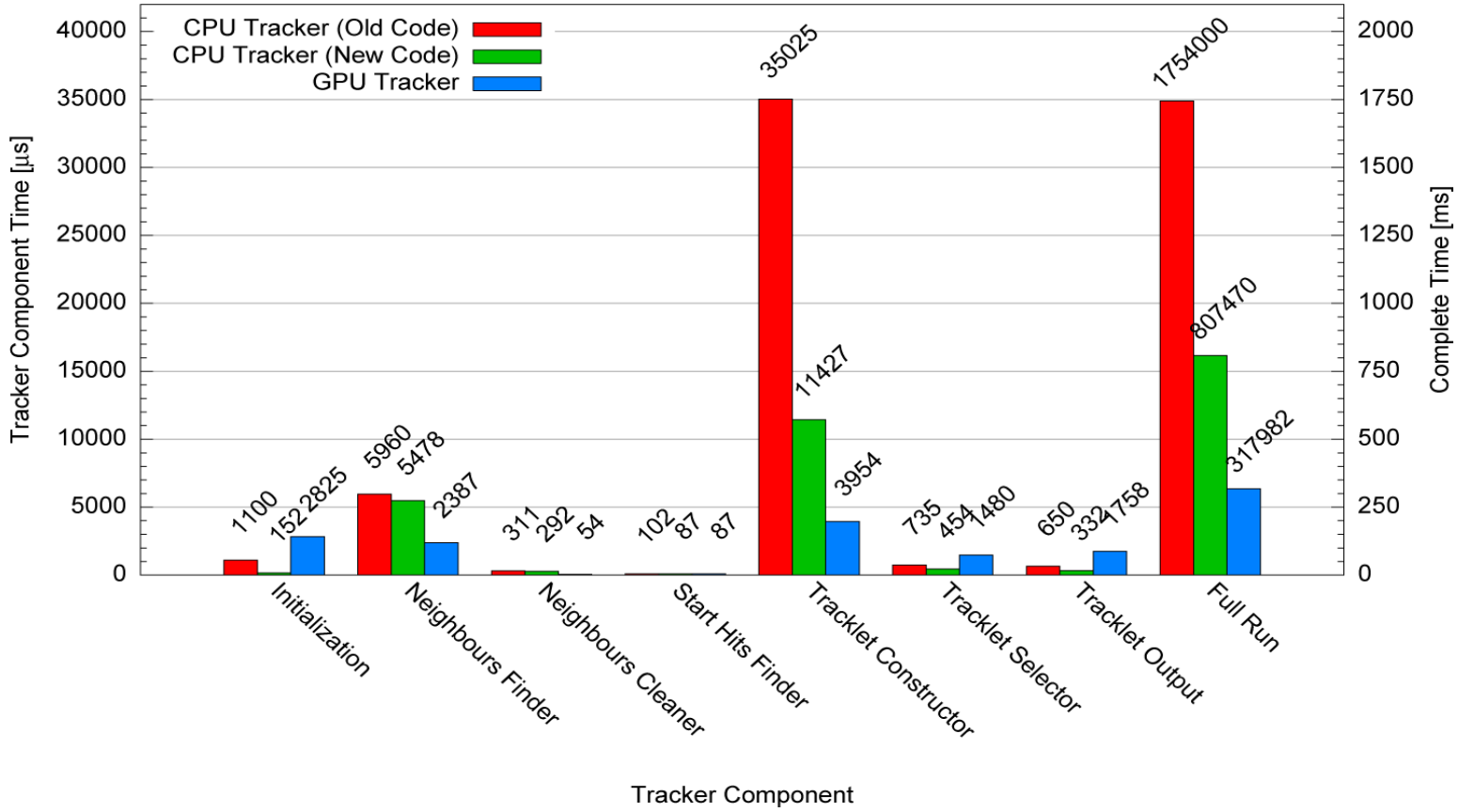
- **Asynchronous Calculation**
  - **The initialization / output of events on the CPU can overlap with the tracking of another event on the GPU**
  - **There is no need that the transfer of data to the GPU is governed by the CPU**
  - **Instead the data should be transferred using Direct Memory Access (DMA)**
  - **This way 3 tasks can be performed in parallel**

- **Pipelining**
  - **A pipeline was introduced in the tracking algorithm**
  - **The GPU performs the tracking for event n, while event n-1 is transferred to the GPU, while event n-2 is initialized by the CPU**
  - **It is only necessary to ensure, that the GPU is always fully utilized, as the GPU now is the main processor for computing**

Pipelining visualization

- **The diagram below shows the tracking of 12 slices in a pipelined way**
- **Only during the initialization of the first slice the GPU is idling**



GPU Tracker Speedup

- **The following final speedup was achieved:**