Parallel Systems Course: Chapter IV

GPU Programming

Jan Lemeire
Dept. ETRO
October 2017

Vrije Universiteit Brussel

Overview

1. GPUs for general purpose
2. GPU 'threads' executing kernels
3. Starting kernels from host (CPU)
4. Execution model & GPU architecture
5. Warps/wavefronts
6. Optimizing GPU programs
7. Analysis & Conclusions

Vectorization tonen!!!!!!!!

Overview

1. GPUs for general purpose
2. GPU 'threads' executing kernels
3. Starting kernels from host (CPU)
4. Execution model & GPU architecture
5. Warps/wavefronts
6. Optimizing GPU programs
7. Analysis & Conclusions


2010
350 Million triangles/second
3 Billion transistors GPU
1995
5.000 triangles/second
800.000 transistors GPU
2016
14.000 Million triangles/second
15 Billion transistors GPU
OGIO
HELMET


[Beta - MikeTheSempai]
70m
RELEASE


B SYNCHRONIZE


ELITEGROUP
GeForce
8800 GTX
NVIDIA

versus


intel
CORE i3
inside
intel

Supercomputing for free

- FASTRA at university of Antwerp

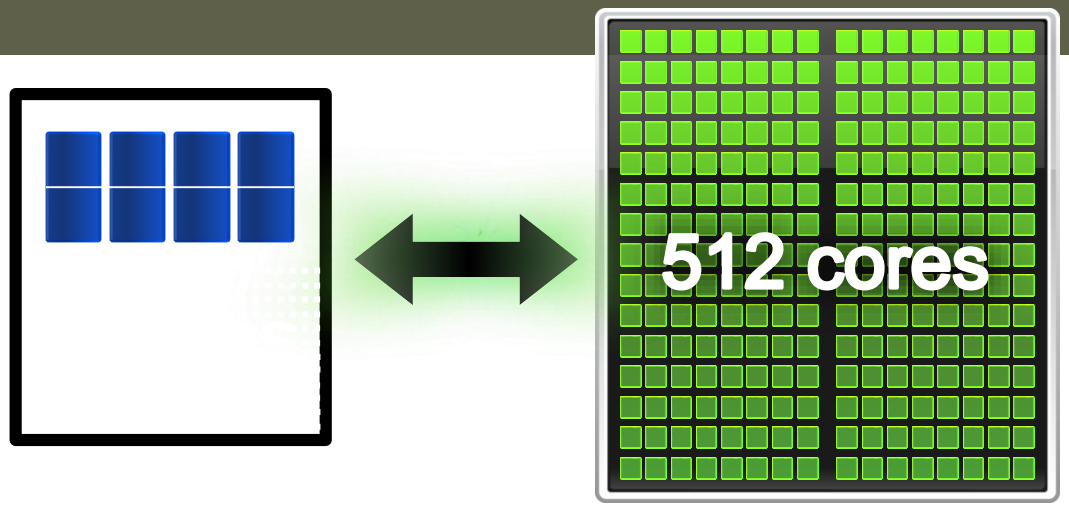
http://fastra.ua.ac.be

Collection of 8 graphical cards in PC

FASTRA 8 cards = 8x128 processors = 4000 euro

Similar performance as University's supercomputer (512 regular desktop PCs) that costed 3.5 million euro in 2005

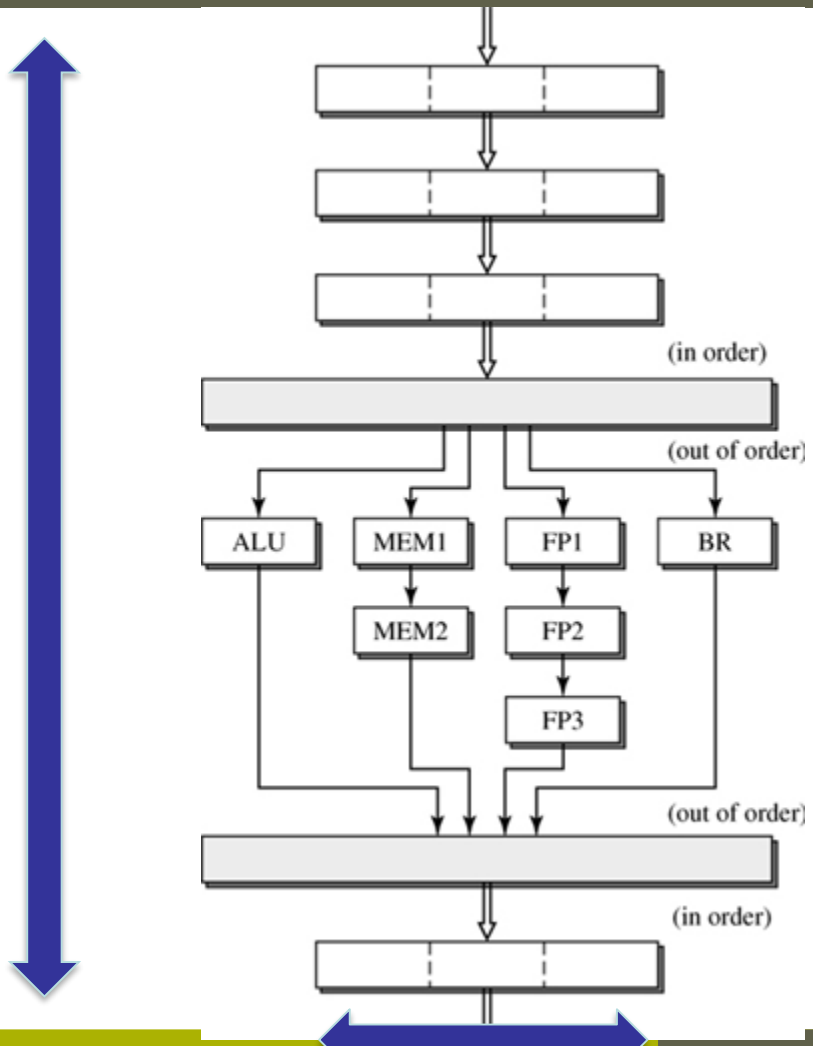
Why are GPUs faster?



GPU specialized for math-intensive highly parallel computation

So, more transistors can be devoted to data processing rather than data caching and flow control

‘Sequential’ processor: super-scalar out-of-order pipeline



Pipeline depth

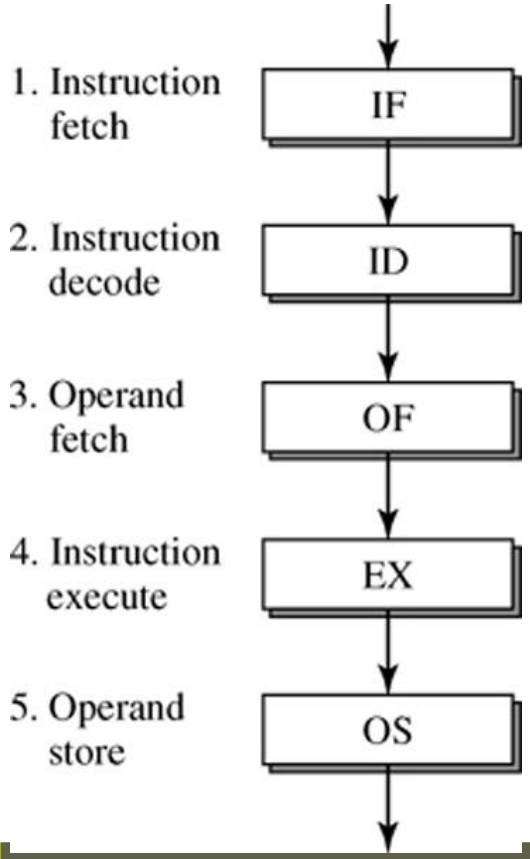
Pipeline width

Different processing units

Out-of-order execution
Branch prediction
Register renaming
…

Scalar Processor: Pipelined design

- Typically (CPU): five tasks in instruction execution
  - IF: instruction fetch
  - ID: instruction decode
  - OF: operand fetch
  - EX: instruction execution
  - OS: operand store, often called write-back WB
- GPU: 24 stages

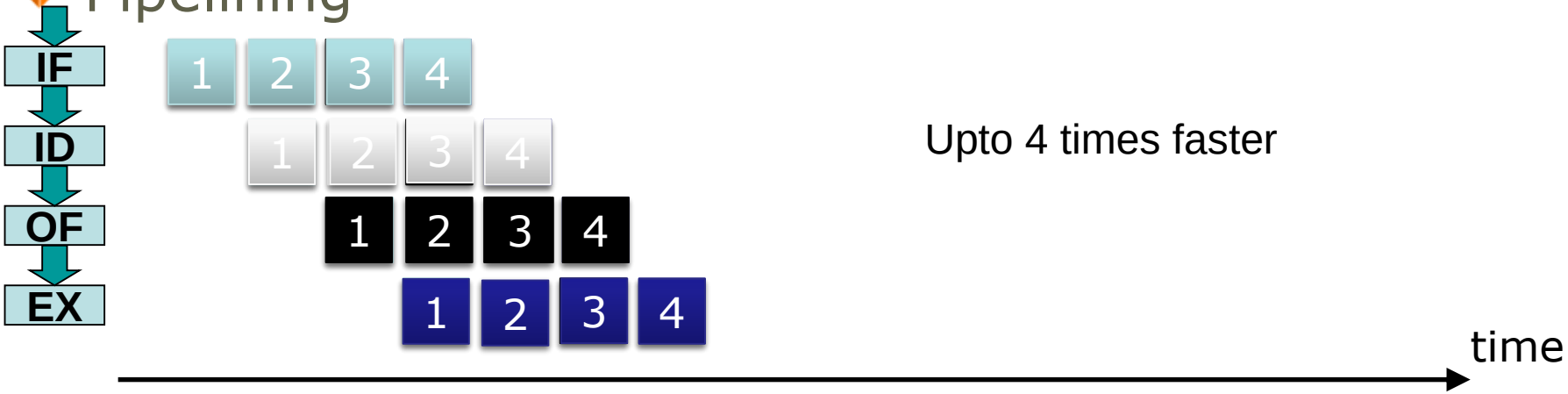


Pipelining Principle

- Long operations

- Combination of short operations

- Pipelining



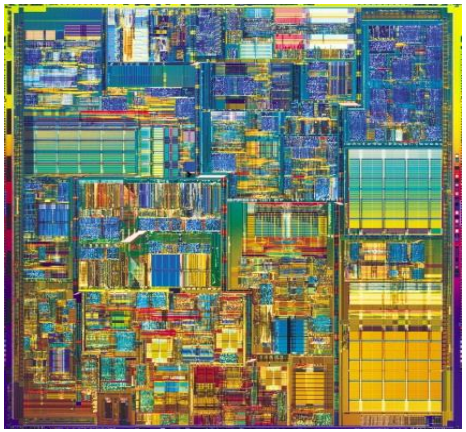
Upto 4 times faster

CPU computing

Algorithm

Implementation

Compiler



manual

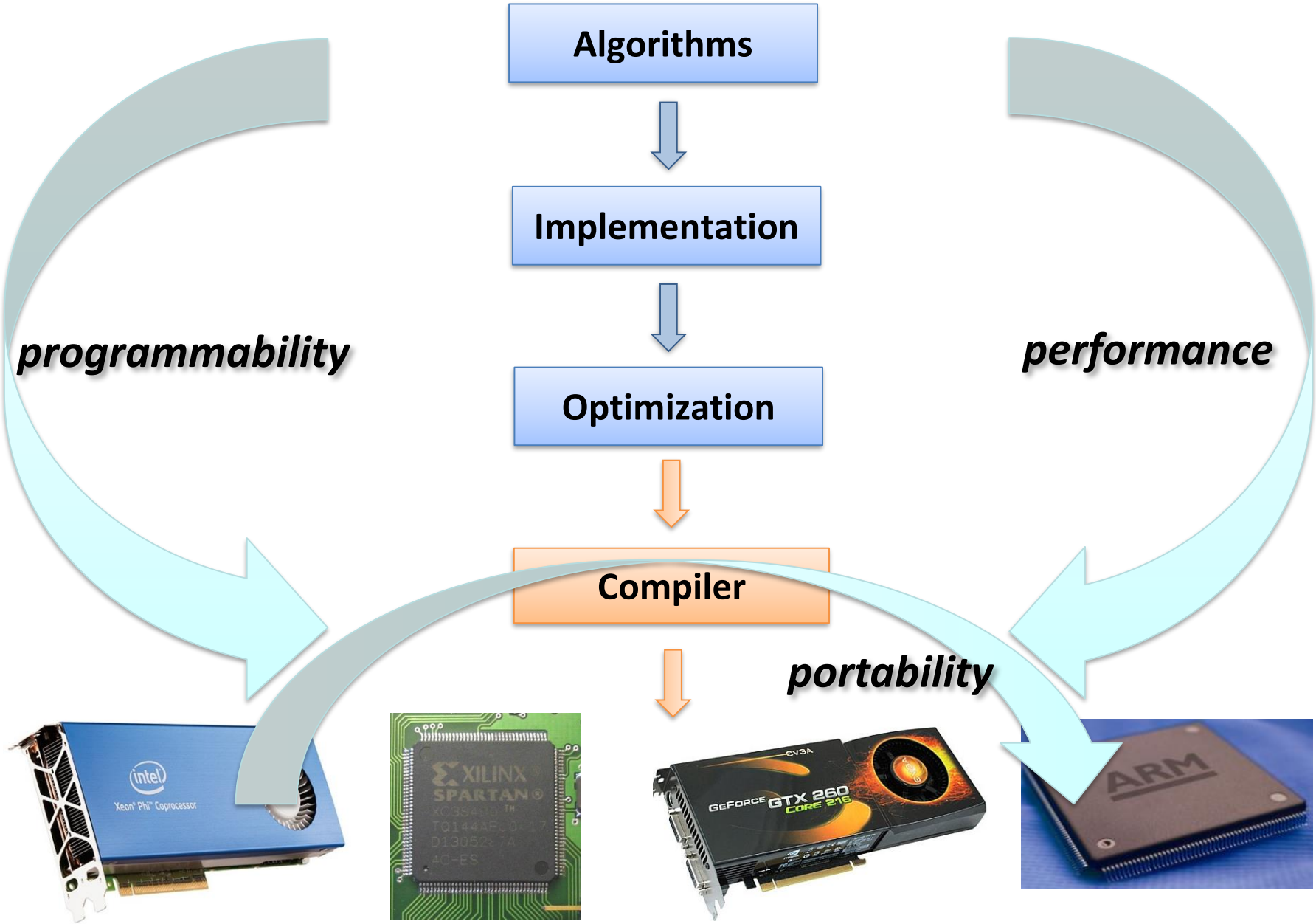
automatic

Write once
Run everywhere
efficiently!

Automatic optimization

Low latency of each instruction!

Challenges of GPU computing



GPU architecture strategy

- Light-weight threads, supported by the hardware
  - Thread processors, upto 96 threads per processing element
  - Switching between threads can happen in 1 cycle!
- No caching mechanism, branch prediction, …
  - GPU does not try to be efficient for every program, does not spend transistors on optimization
  - Simple straight-forward sequential programming should be abandoned…
- Less higher-level memory:
  - GPU: 16KB shared memory per SIMD multiprocessor
  - CPU: L2 cache contains several MB's
- Massively floating-point computation power
- Transparent system organization
  - ⇔ Modern (sequential) CPUs based on simple Von Neumann architecture

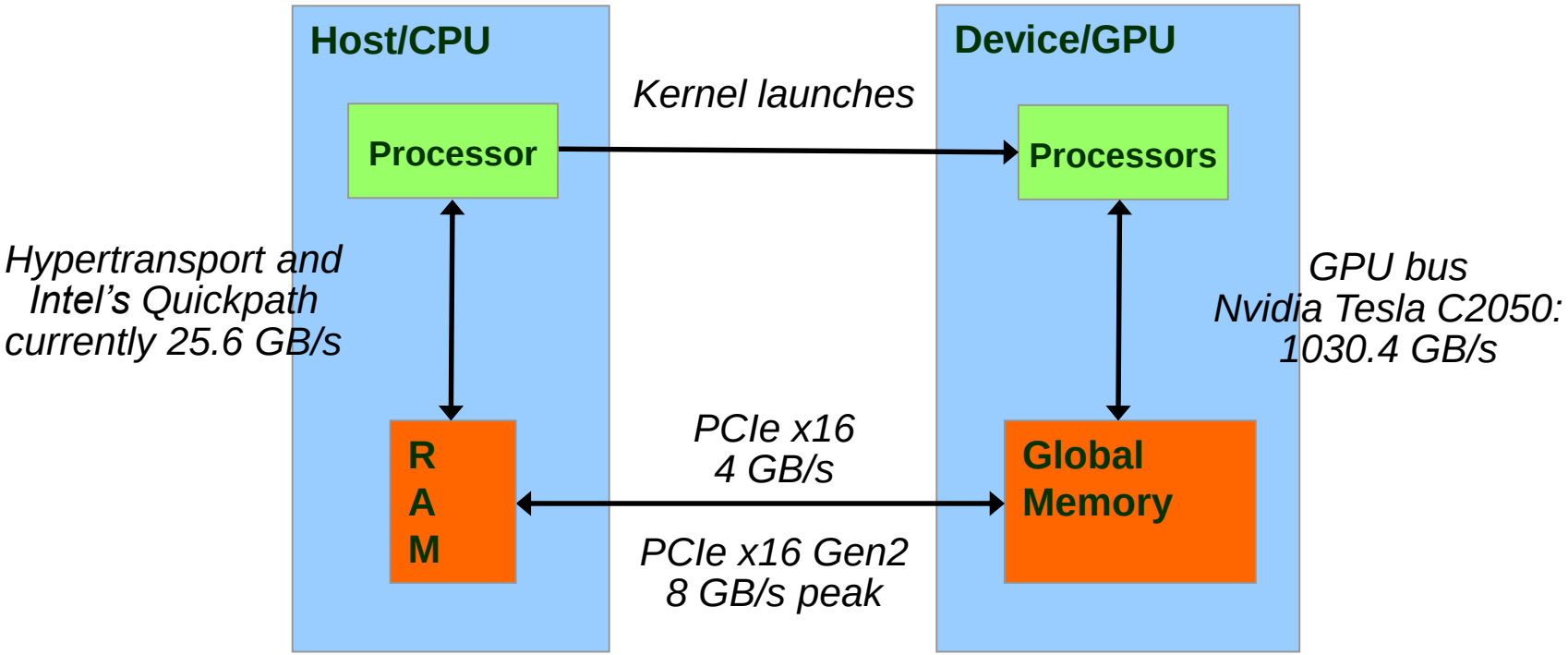
So…

GP-GPUs: Graphics Processing Units for General-Purpose programming

Usage

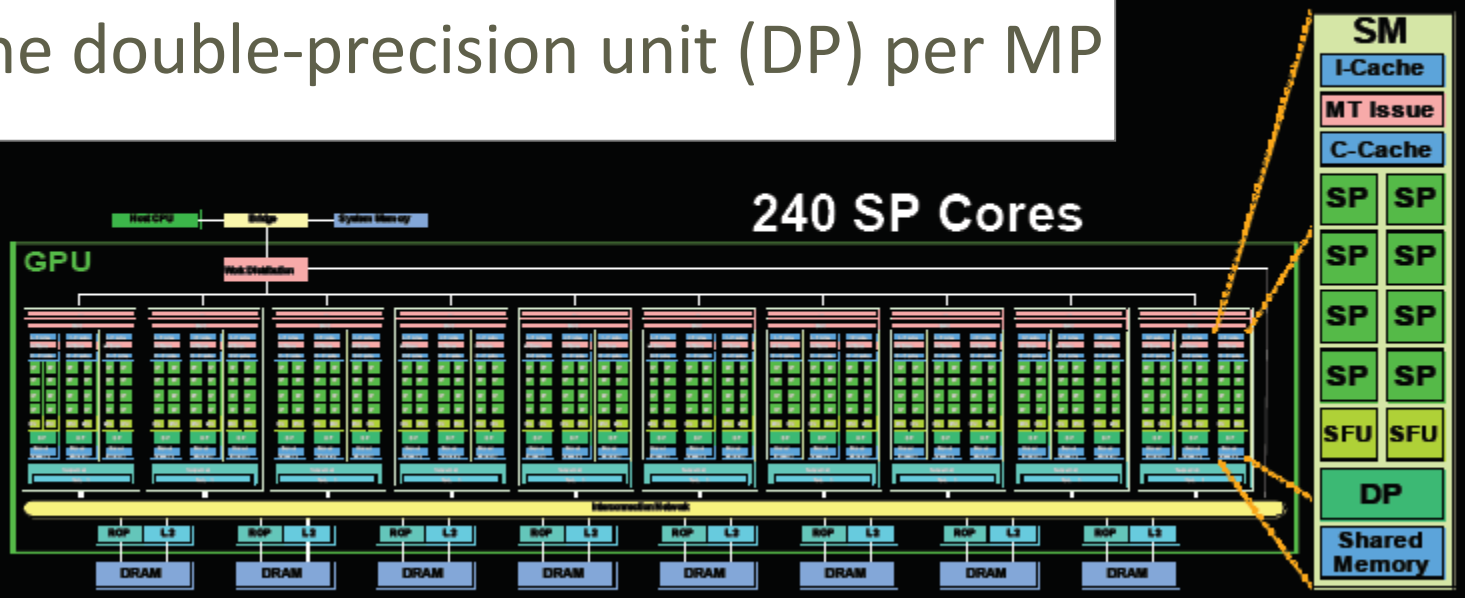
- Copy data from CPU to GPU
- Start kernel within CPU-program (C, java, Matlab, python, …)
  - Kernel = program executed on GPU by each 'thread'
  - Several kernels can be launched (pipelined)
  - Handled on the GPU one by one or in parallel
- Copy results back from GPU to CPU

Host (CPU) – Device (GPU)



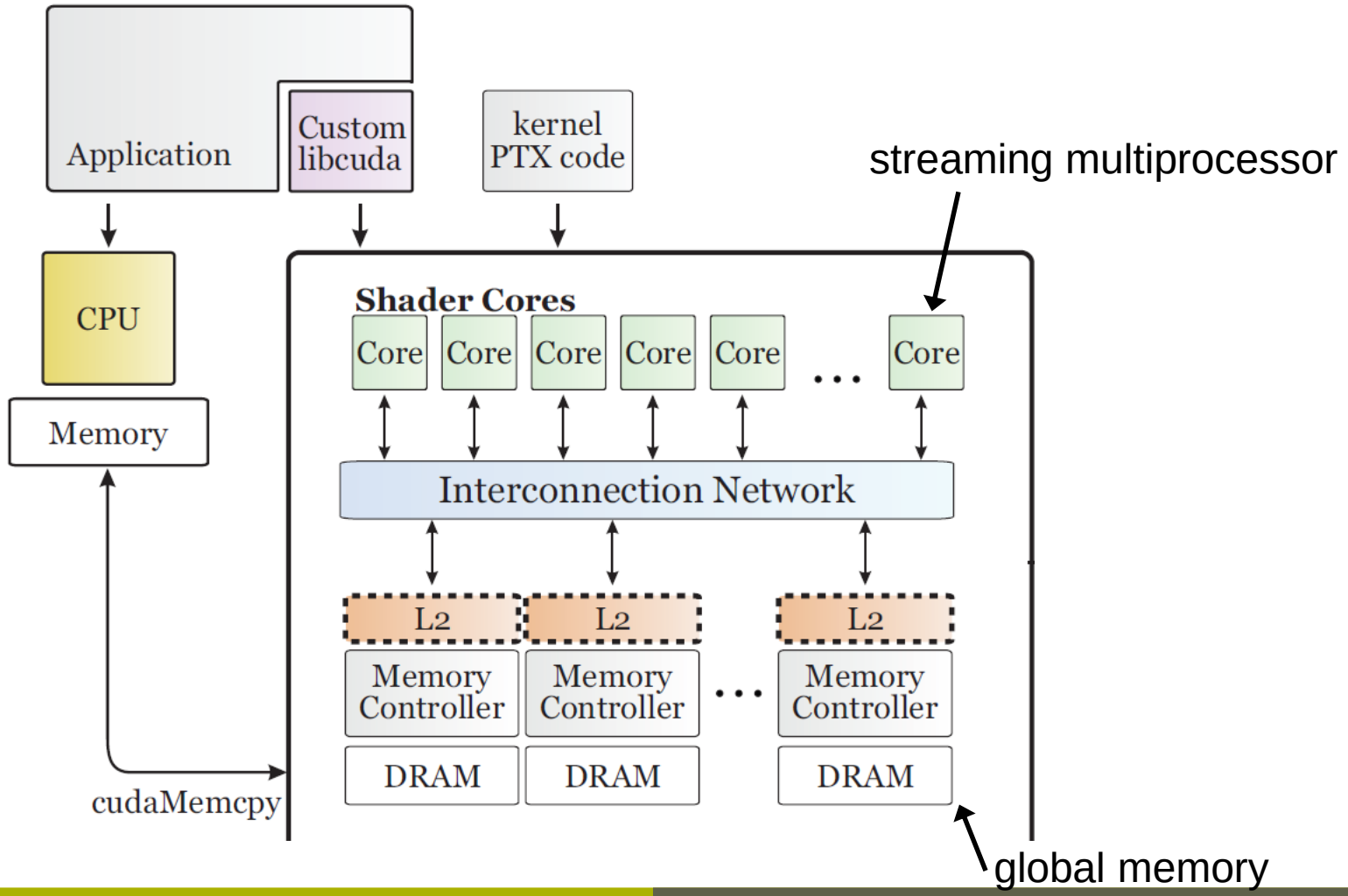
GPU Architecture

- In the GTX 280, there are 10 Thread Processing Clusters
  - Each has 3 Streaming Multiprocessors, which we will refer to as ***multiprocessors (MPs).***
  - Each MP has 8 Thread Processors. We will refer to these as ***Scalar Processors (SP).***
  - **240 processor cores and 30 MPs in total!**
- One double-precision unit (DP) per MP

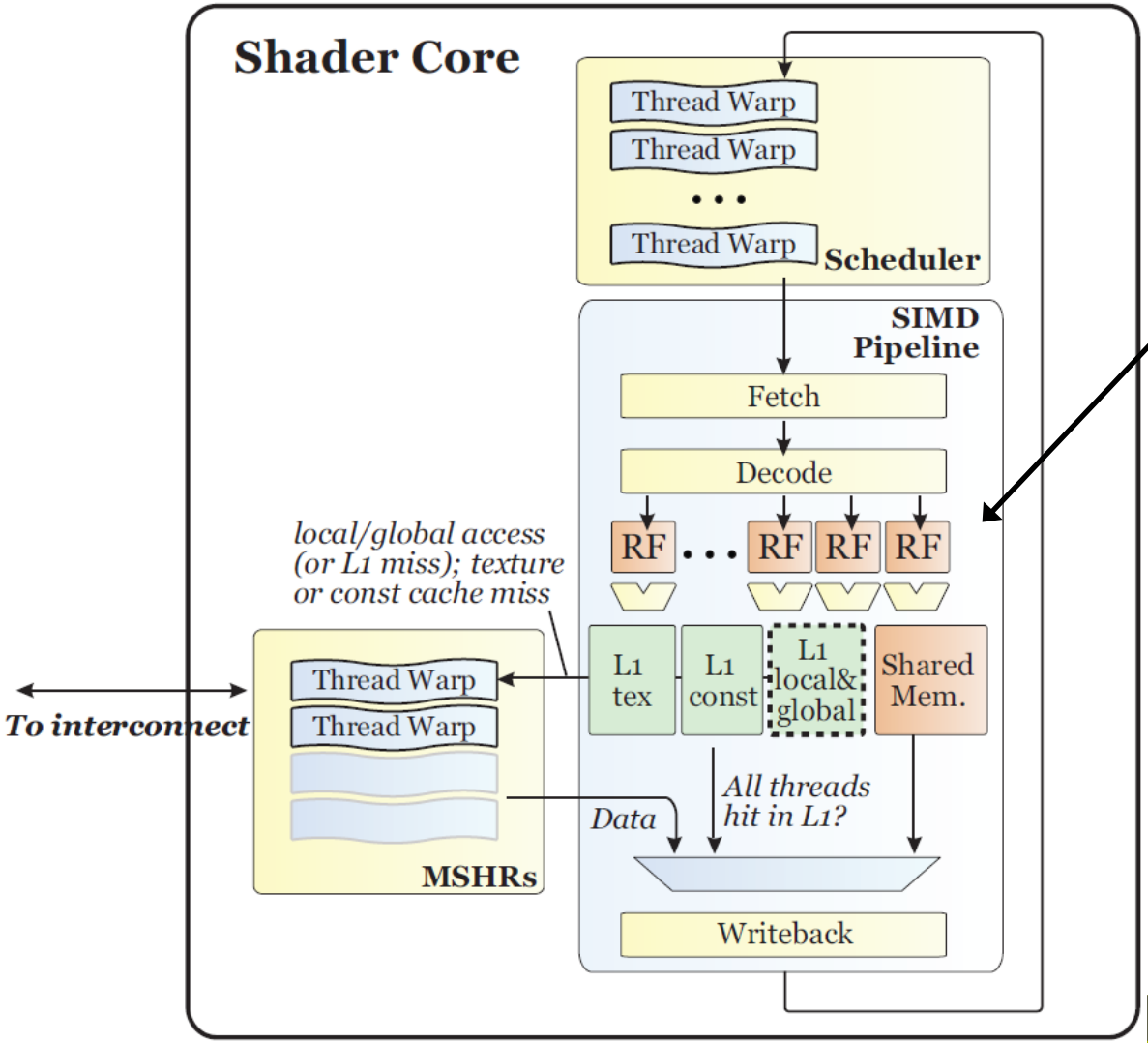


Source : NVIDIA

GPU Architecture



1 Streaming Multiprocessor



The Same Instruction is executed on Multiple Threads (SIMT)

width of pipeline: 8 – 32 – 192 - 128

GPU vs CPU: NVIDIA 280 vs Intel i7 860

	GPU	CPU[1]
Registers	16,384 (32-bit) / multi-processor[3]	128 reservation stations
Peak memory bandwidth	141.7 Gb/sec	21 Gb/sec
Peak GFLOPs	562 (float)/ 77 (double)	50 (double)
Cores	240 (scalar processors)	4/8 (hyperthreaded)
Processor Clock (MHz)	1296	2800
Memory	1Gb	16Gb
Local/shared memory	16Kb/TPC[2]	N/A
Virtual memory	None	

[1]http://ark.intel.com/Product.aspx?id=41316
[2]TPC = Thread Processing Cluster (24 cores)
[3]30 multi-processors in a 280

Performance: GFlops?

- GPUs consist of MultiProcessors (MPs) grouping a number of Scalar Processors (SPs)
- Nvidia GTX 280:
  - 30MPs x 8 SPs/MP x 2FLOPs/instr/SP x 1 instr/clock x 1.3 GHz

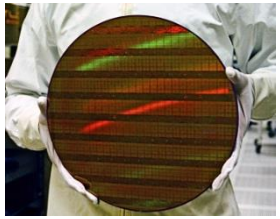
  = 624 GFlops
- Nvidia Tesla C2050:
  - 14 MPs x 32 SPs/MP x 2FLOPs/instr/SP x 1 instr/clock x 1.15 GHz (clocks per second)

  = 1030 GFlops

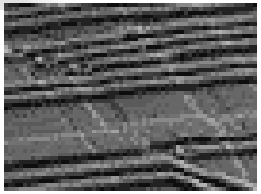
Other limit: bandwidth

- Nvidia GTX 280:
  - 1.1 GHz memory clock
  - 141 GB/s
- Nvidia Tesla C2050:
  - 1.5 GHz memory clock
  - 144 GB/s

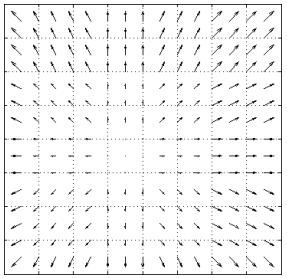
Example: real-time image processing



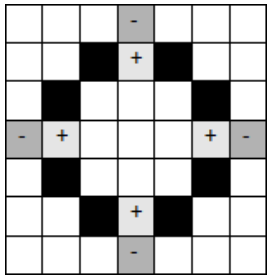
Images of 20MegaPixels



Pixel rescaling



lens correction



pattern detection

CPU gives only 4 fps
next generation machines need 50 fps
GPUs deliver 70 fps

Example: pixel transformation (FPN)

```
usgn_8  transform(usgn_8 in, sgn_16 gain, sgn_16 gain_divide,
sgn_8 offset)
{
   sgn_32 x;

   x = (in * gain / gain_divide) + offset;

   if (x < 0) x = 0;
   if (x > 255) x = 255;
   return x;
 }
```

Pixel transformation

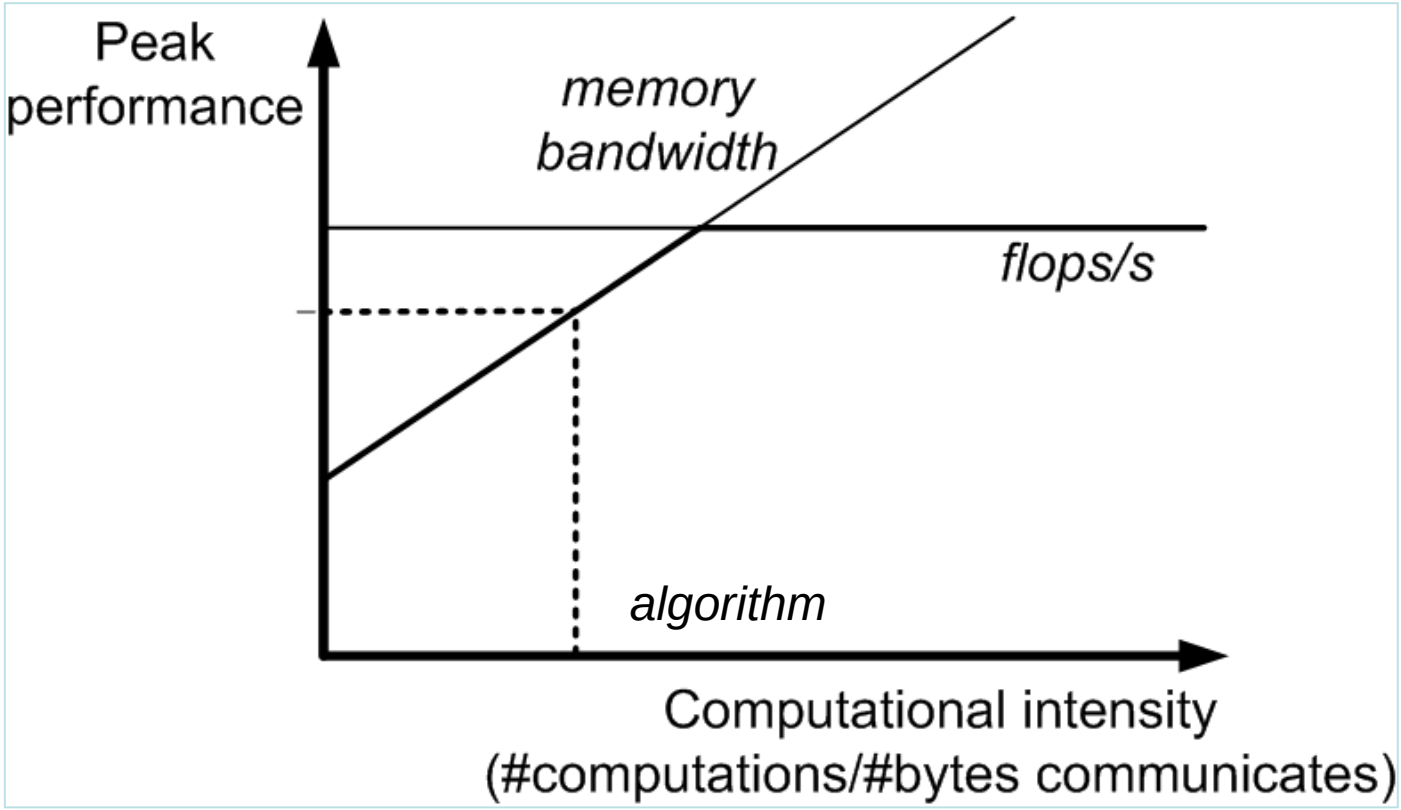
- Performance on Tesla C2050
- 1 pixel is represented by 1 byte [0-255]
  - Per pixel: read 4 bytes (pixel & gain & offset) and write 1 byte
- Integer operations: performance is half of floating point operations
- **Two different implementations**:
  - FPN1: 1 pixel per thread
  - FPN4: 4 pixels per thread (treat 4 bytes as 1 'word')

P_{mem} *(bytes/s)*	115 GB/s	**P_{ops}** *(ops/s)*	500 Gops/s	
bytes/pixel	5	*Ops/pix*	5+4 (FPN1) 5+1 (FPN4)	CI=1,8 CI=1,2
P_{mem}xCI *(pix/s)*	**23 Gpix/s**	**Pix/s**	**56 Gpix/s (FPN1)** **83 Gpix/s (FPN4)**	

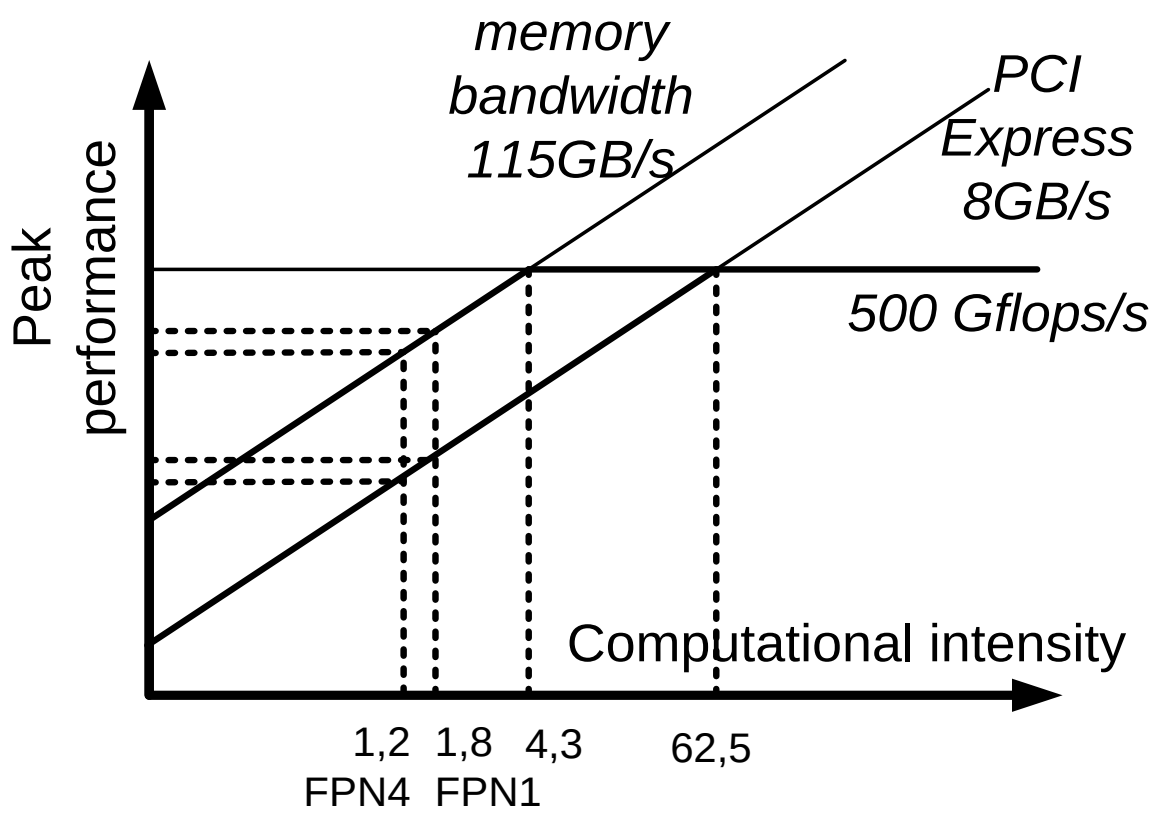
Memory-bound (take minimum)

CI = Computational Intensity

Roofline model



Roofline model applied



But… nothing is for free

- Harder to program!
  - Hardware architecture should be taken into account
  - Optimization is important
  - Additional complexity in code
  - Harder to debug, maintain, …
- Algorithms should contain inherently massively fine-grained parallelism

Overview

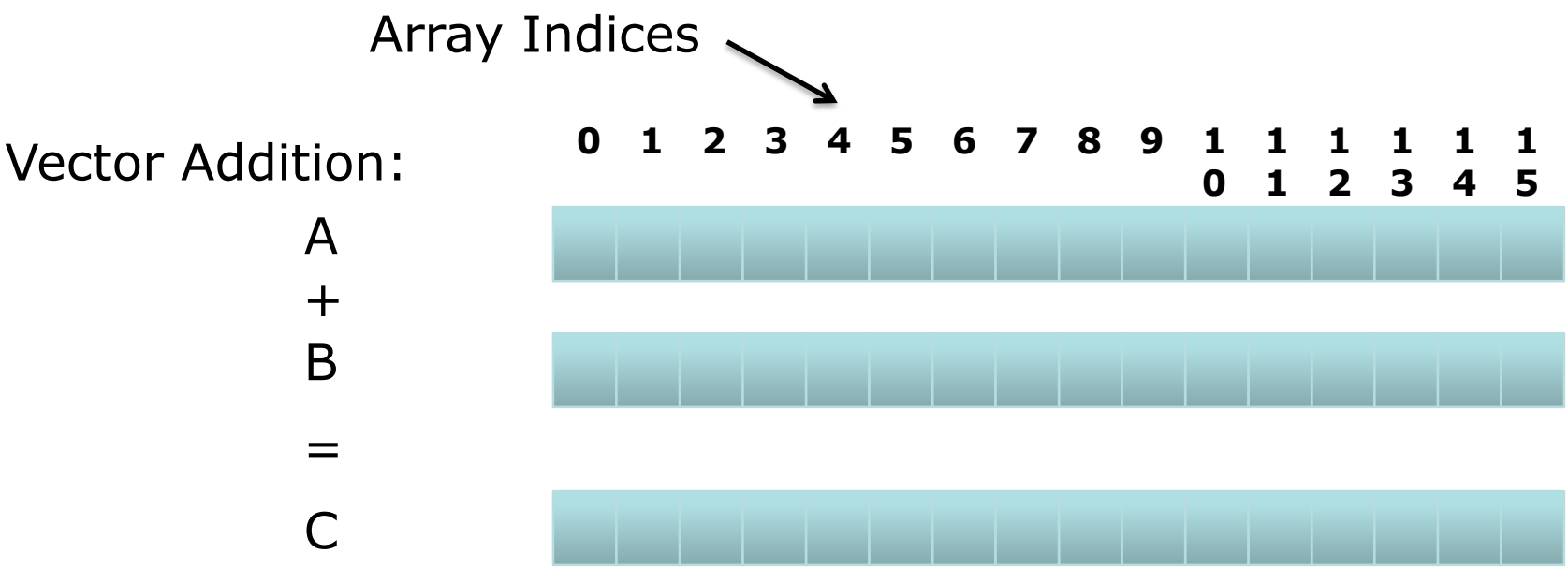
1. GPUs for general purpose
2. GPU 'threads' executing kernels
3. Starting kernels from host (CPU)
4. Execution model & GPU architecture
5. Warps/wavefronts
6. Optimizing GPU programs
7. Analysis & Conclusions

Thread Structure

- Massively parallel programs are usually written so that each thread computes one part of a problem
  - For vector addition, we will add corresponding elements from two arrays, so each thread will perform one addition
  - If we think about the thread structure visually, the threads will usually be arranged in the same shape as the data

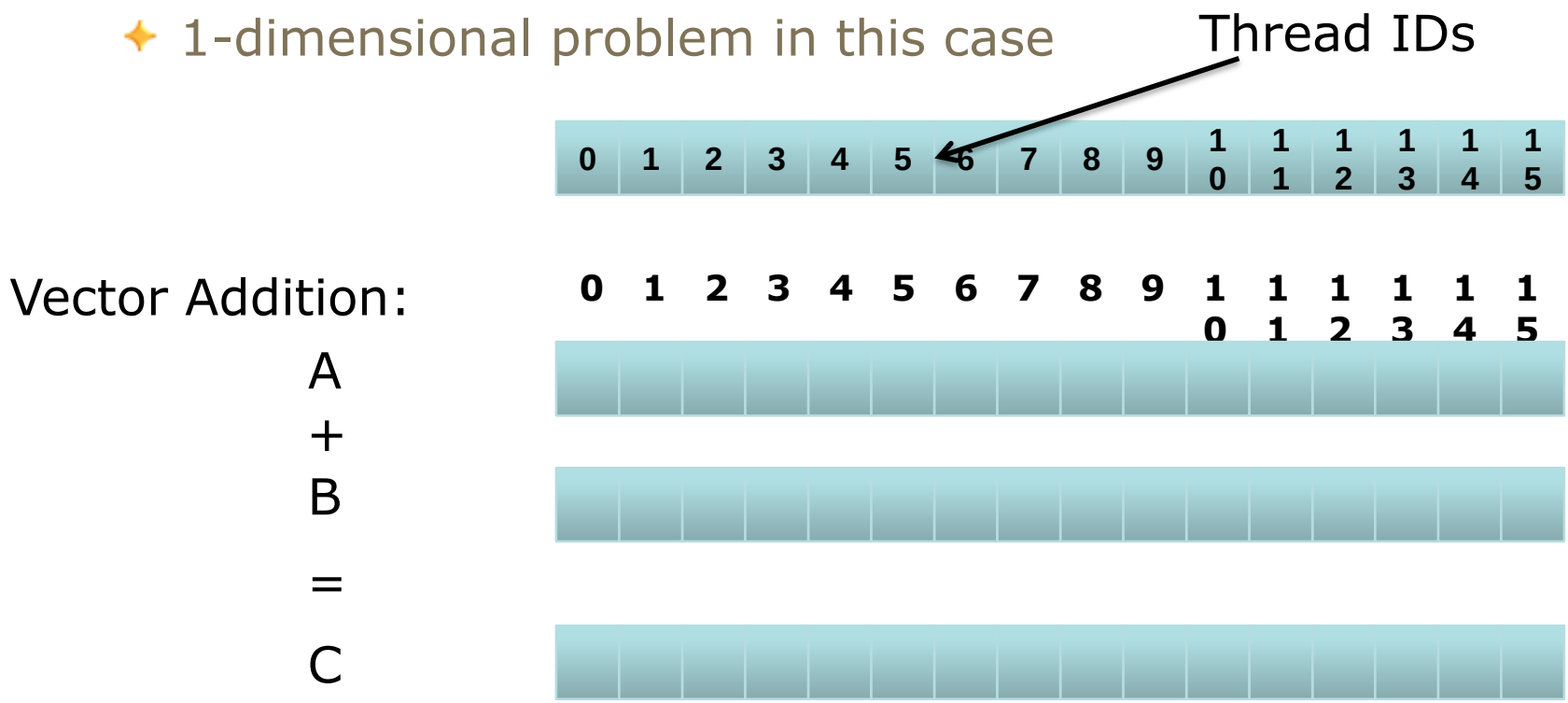
Thread Structure

- Consider a simple vector addition of 16 elements
  - 2 input buffers (A, B) and 1 output buffer (C) are required



Thread Structure

- Create thread structure to match the problem
  - 1-dimensional problem in this case



Thread Structure

- Each thread is responsible for adding the indices corresponding to its ID

Vector Addition:

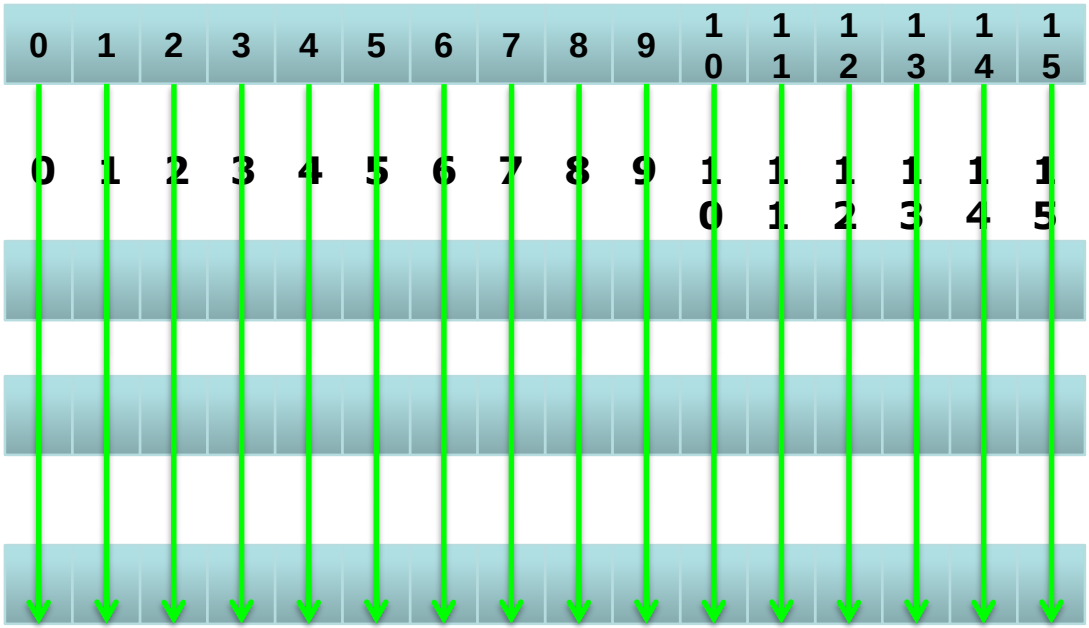
A

+

B

=

C



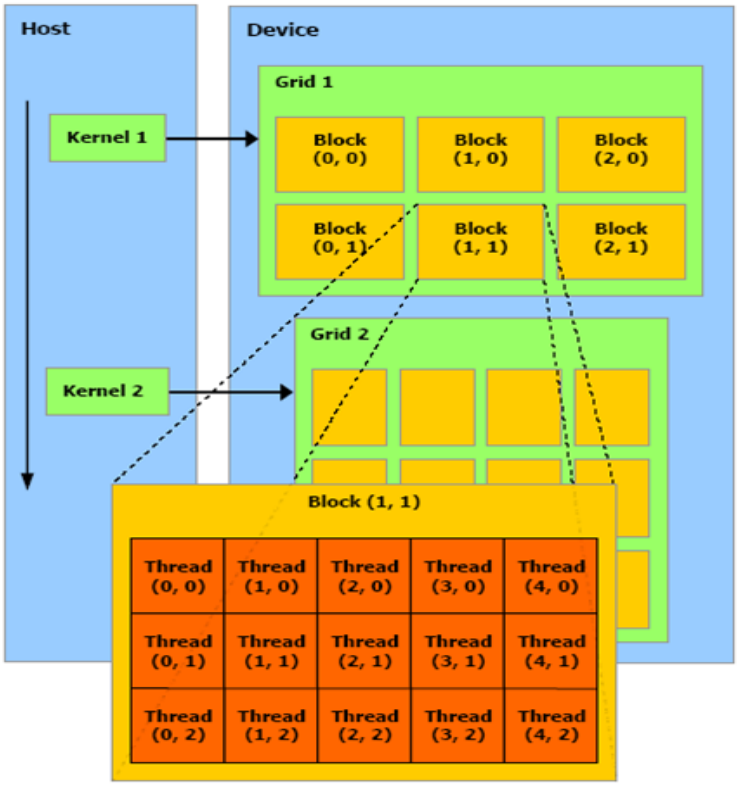
OpenCL Kernel code

```
__kernel void vectorAdd(__global const float * a,
__global const float * b, __global float * c)
{
  // Vector element index
  int nIndex = get_global_id(0);
  // addition
  c[nIndex] = a[nIndex] + b[nIndex];
}
```

- OpenCL kernel functions are declared using "__kernel".
- __global refers to global memory
- get_global_id(0) returns the ID of the thread in execution

Kernel launch

- Execution environment
- Grid
- Work groups/thread blocks in 2 or 3 dimensions
- Specify at launch time: grid of work groups of work items (threads)
- Query in kernel at run time
- Impact on performance!

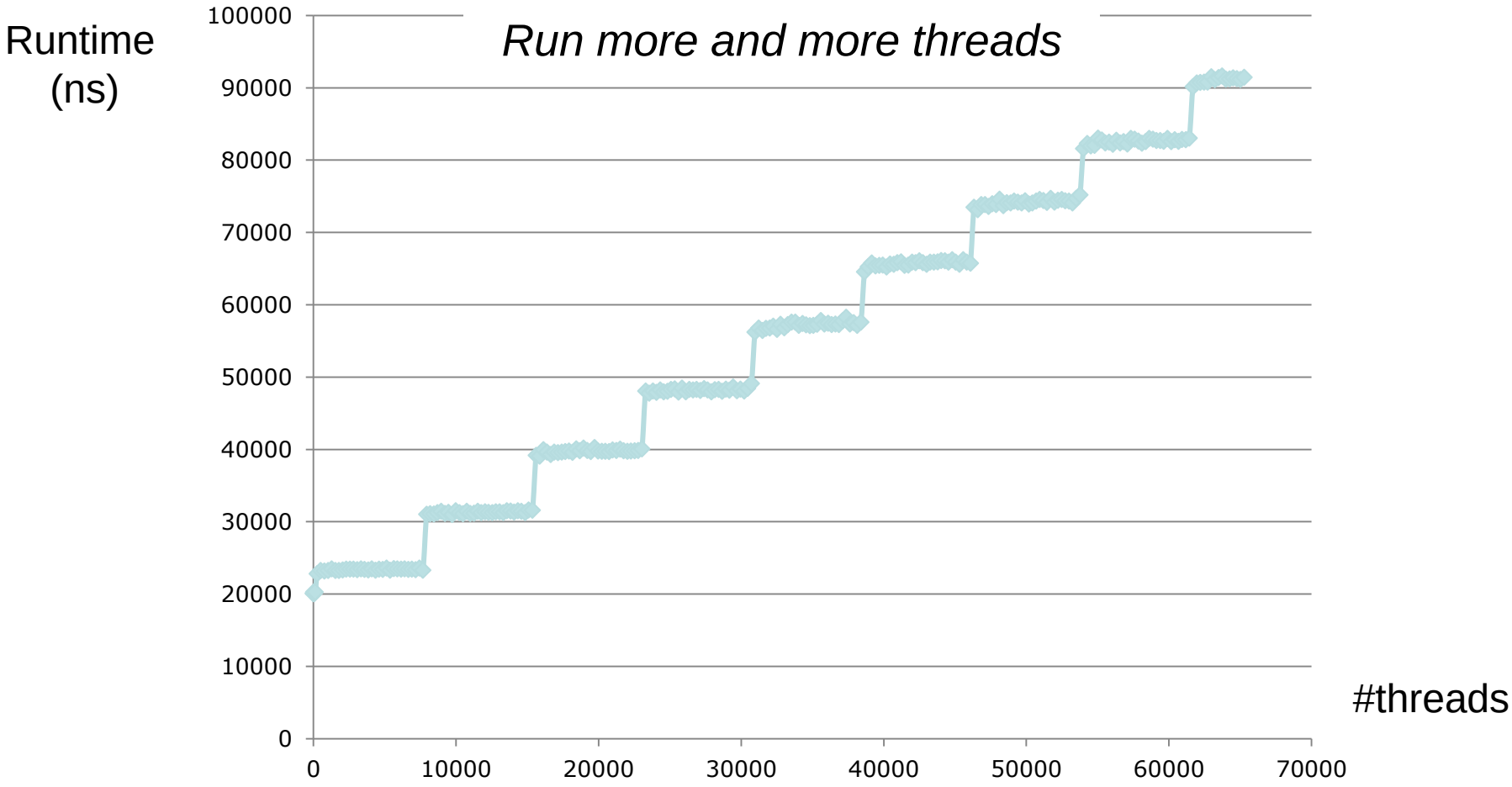


The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks

Work items/threads

- API calls allow threads to identify themselves and their data
- Threads can determine their global ID in each dimension
  - get_global_id(dim)
  - get_global_size(dim)
- Or they can determine their work-group ID and ID within the workgroup
  - get_group_id(dim)
  - get_num_groups(dim)
  - get_local_id(dim)
  - get_local_size(dim)
- get_global_id(0) = column, get_global_id(1) = row
- get_num_groups(0) * get_local_size(0) == get_global_size(0)

The effect of parallelism



Occupancy

- Keep all processing units busy!
  - Enough threads
- All Multiprocessors (MPs)
- All Scalar Processors (SPs)
- Full pipeline of scalar processor
  - Pipeline of 24 stages (*see later*)

So: Power is within reach?

Unfortunately…

- It's not that simple…
- It's not what we are used in the CPU-world
  - CPU: multicores also requires us to program differently
- If we want the speed, we will have to pay for it…

Overview

1. GPUs for general purpose
2. GPU 'threads' executing kernels
3. **Starting kernels from host (CPU)**
4. Execution model & GPU architecture
5. Warps/wavefronts
6. Optimizing GPU programs
7. Analysis & Conclusions

OpenCL Working Group

- **Diverse industry participation**
  - Processor vendors, system OEMs, middleware vendors, application developers
- **Many industry-leading experts involved in OpenCL's design**
  - A healthy diversity of industry perspectives
- **Apple initially proposed and is very active in the working group**
  - Serving as specification editor
- **Here are some of the other companies in the OpenCL working group**

3DLABS SEMICONDUCTOR · ACTIVISION BLIZZARD · AMD · ARM · BROADCOM · EA · codeplay · ERICSSON · freescale semiconductor · HI CORP. · IBM · intel · Imagination TECHNOLOGIES · MOTOROLA · movidia · NOKIA · NVIDIA · QNX SOFTWARE SYSTEMS · RAPIDMIND · SAMSUNG · Seaweed SYSTEMS · TAKUMI · TEXAS INSTRUMENTS · UNIVERSITET

CUDA Working Group

OpenCL Keywords & functions

- Address space qualifiers:
  - __global, __local, __constant and __private
- Function qualifiers:
  - __kernel
- Access qualifiers for images:
  - __read_only, __write_only, and __read_write

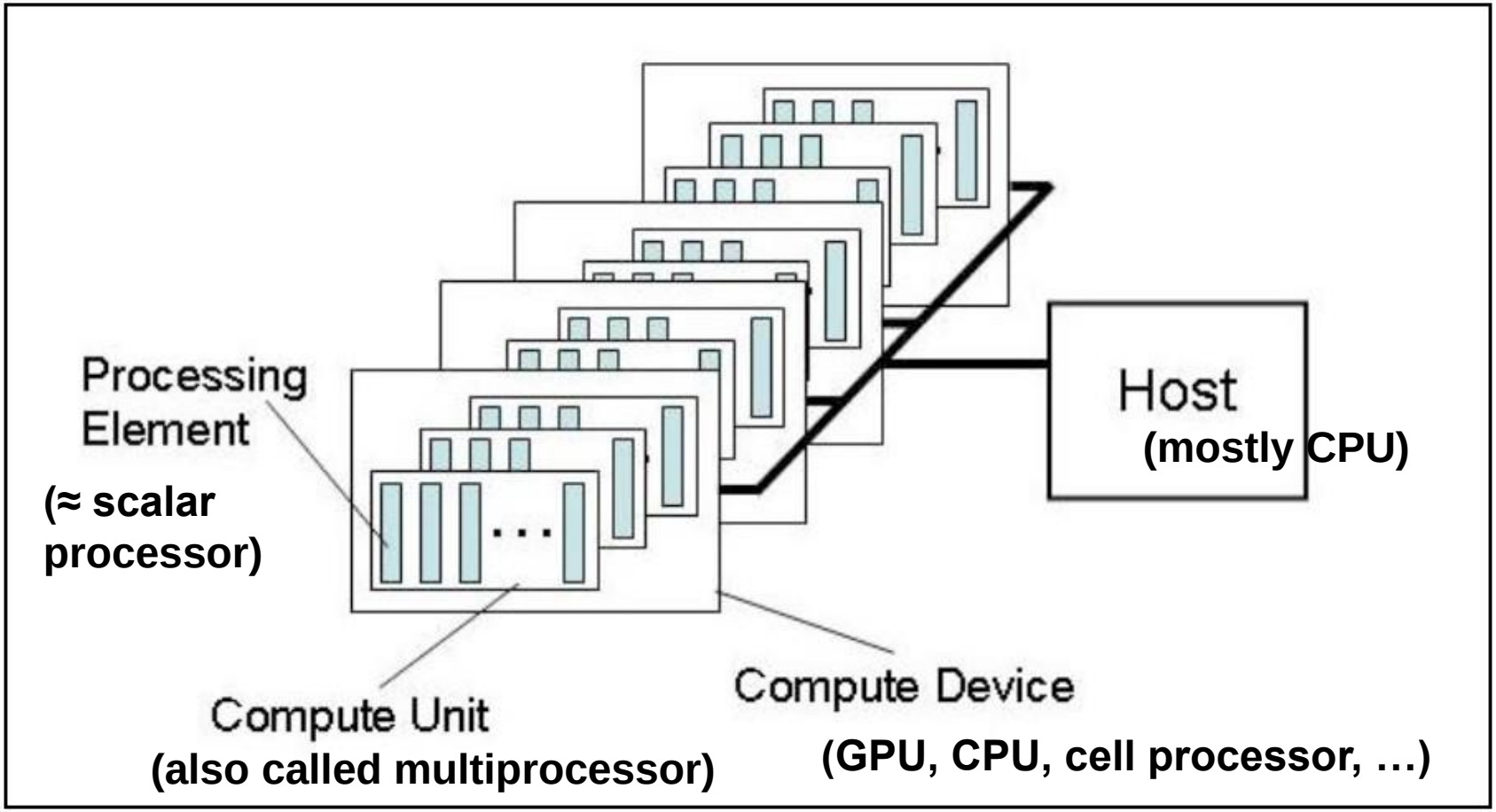
- **OpenCL functions:** start with *cl* prefix

OpenCL Kernel code

```
__kernel void vectorAdd(__global const float * a,
__global const float * b, __global float * c)
{
  // Vector element index
  int nIndex = get_global_id(0);
  // addition
  c[nIndex] = a[nIndex] + b[nIndex];
}
```

- OpenCL kernel functions are declared using "__kernel".
- __global refers to global memory
- get_global_id(0) returns the ID of the thread in execution

Architecture – Computing elements



OpenCL Software Stack

- **Host program**
  - Query compute devices
  - Create contexts
  - Create memory objects associated to contexts
  - Compile and create kernel program objects
  - Issue commands to command-queue
  - Synchronization of commands
  - Clean up OpenCL resources
- **Kernels**
  - C code with some restrictions and extensions

Platform Layer

Runtime

Language

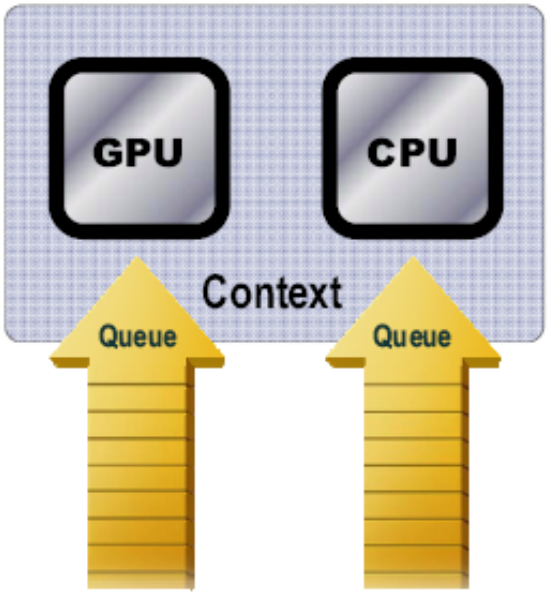
Shows the steps to develop an OpenCL program

On Host: platform layer

- **Creating the basic OpenCL run-time environment**
  - **Select Platform:** collection of devices managed by the OpenCL framework that allow an application to share resources and execute kernels on devices in the platform
    - **OpenCL framework ≈ OpenCL implementation: NIVDIA, AMD, Intel, …**
  - **Device:** hardware such as GPU, multicore, cell processor
  - **Context:** defines the entire environment, including kernels, devices, memory management, command-queues, etc.
  - **Command-Queue:** object where OpenCL commands are enqueued to be executed by the device.

Setup

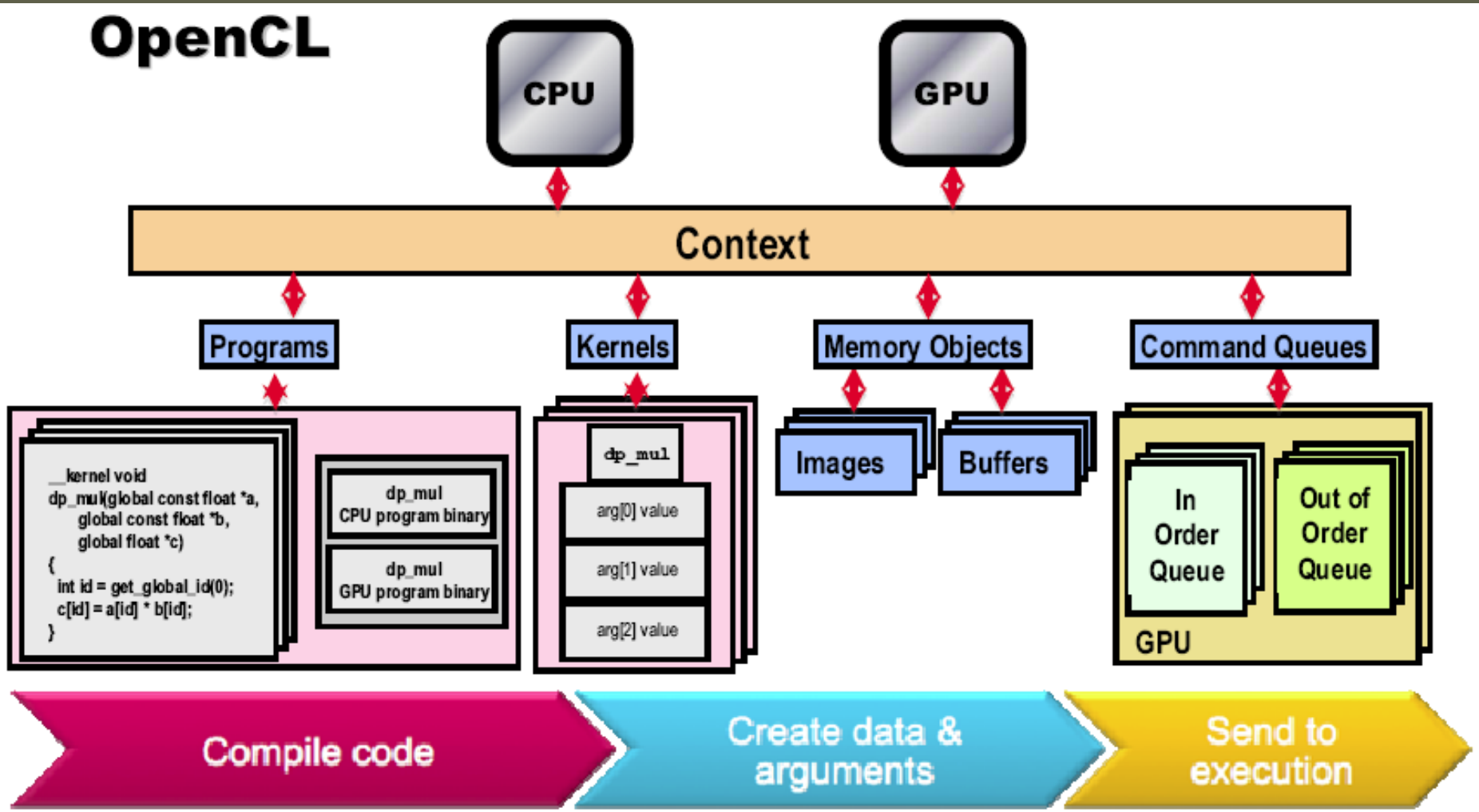
1. Get the device(s)
2. Create a context
3. Create command queue(s)



platform is set to NULL

```
cl_uint num_devices_returned;
cl_device_id devices[2];
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1,
                     &devices[0], num_devices_returned);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1,
                     &devices[1], &num_devices_returned);

cl_context context;
context = clCreateContext(0, 2, devices, NULL, NULL, &err);

cl_command_queue queue_gpu, queue_cpu;
queue_gpu = clCreateCommandQueue(context, devices[0], 0, &err);
queue_cpu = clCreateCommandQueue(context, devices[1], 0, &err);
```

Within context

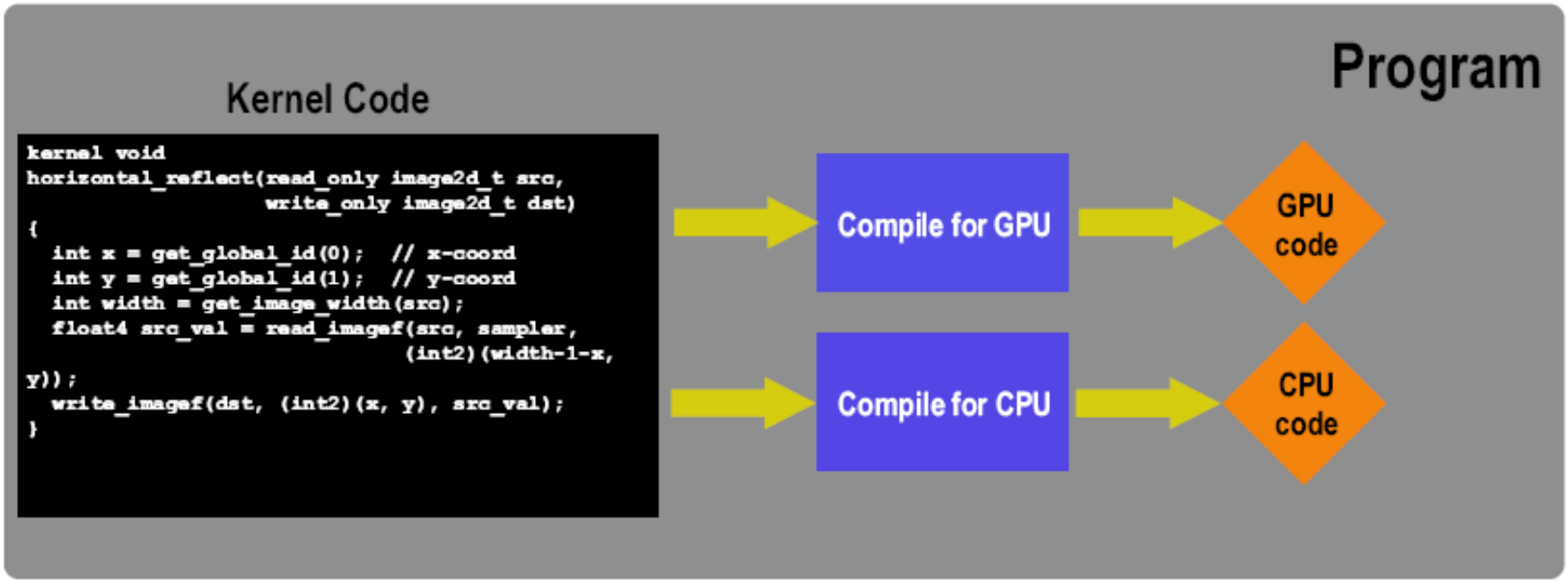


Data movement & kernel calls

- Create **buffers** for this ***context***.
  - In global memory
- Data movement
  - Host => device: clEnqueueWriteBuffer()
  - Device => host: clEnqueueReadBuffer()
- Create **program** using ***input file*** for this ***context***.
  - build this ***program***.
- Create **kernel** from ***program***.

Executing Code

- Programs build executable code for multiple devices
- Execute the same code on different devices



Cleanup

```
// release kernel, program, and memory objects
DeleteMemobjs (cmMemObjs, 3);
free (cdDevices);
clReleaseKernel (ckKernel);
clReleaseProgram (cpProgram);
clReleaseCommandQueue (cqCommandQue);
clReleaseContext (cxMainContext);
```

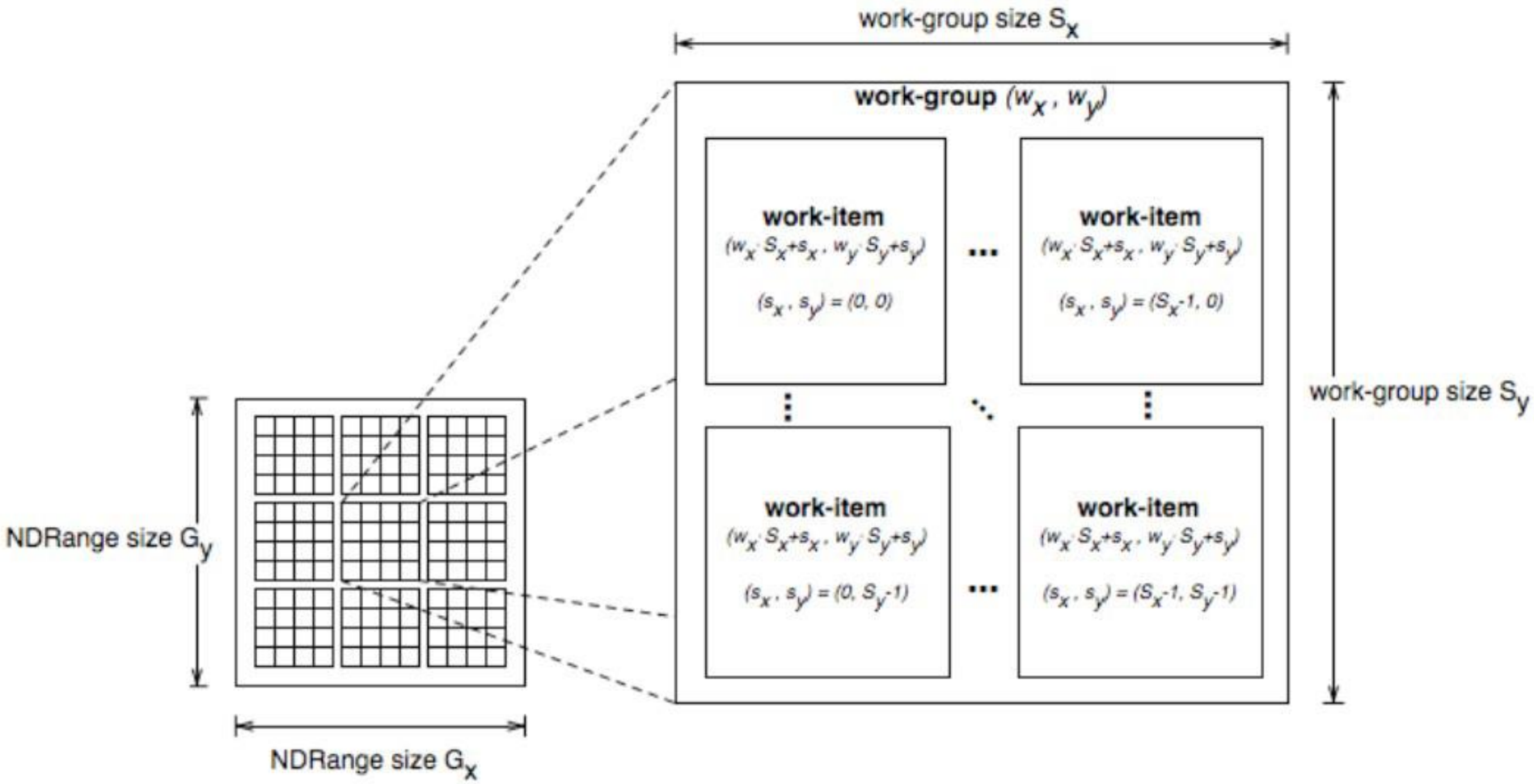
Overview

1. GPUs for general purpose
2. GPU 'threads' executing kernels
3. Starting kernels from host (CPU)
4. **Execution model & GPU architecture**
5. Warps/wavefronts
6. Optimizing GPU programs
7. Analysis & Conclusions

Execution Model

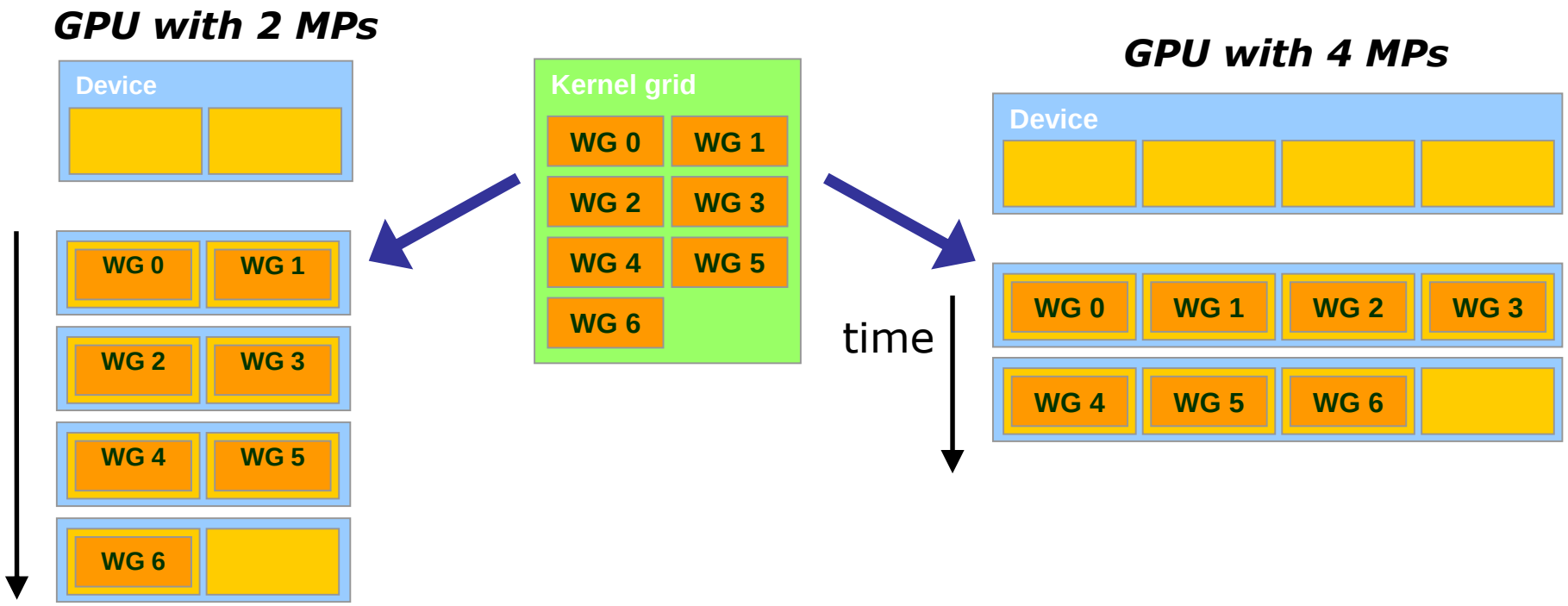
- Kernel = smallest unit of execution, like a C function, executed by each **work item** (≈ thread)
- Data parallelism: kernel is run by a grid of work groups
- **Work group** consist of instances of same kernel: work items
- Different data elements are fed into the work items of the work groups
  - We talk about *stream computing*

Architecture – Execution Model



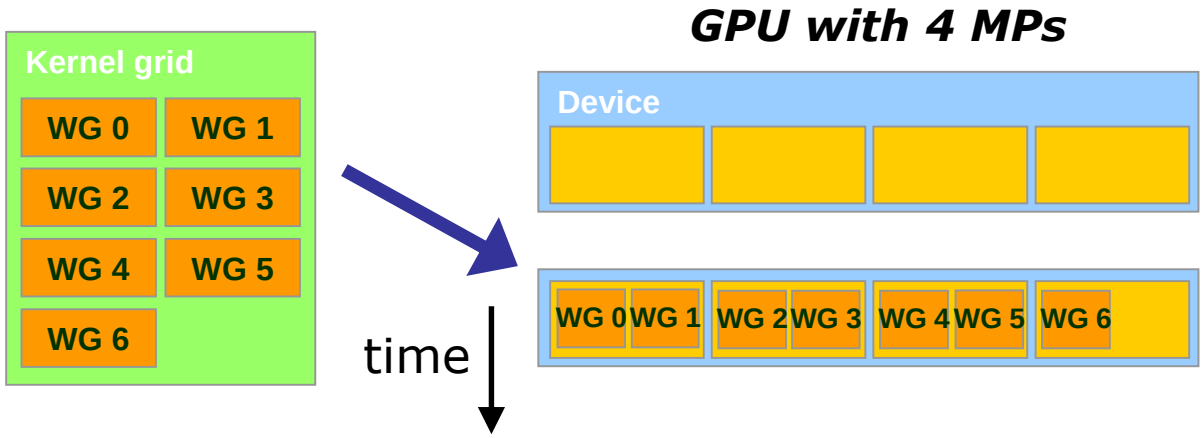
Kernel execution

- Simple scheduler
  - Assigns work groups to available streaming MultiProcessors (MPs)
  - Basically, a waiting queue for work groups
- Work groups (WGs) execute independently
  - Global Synchronization among work groups is not possible!

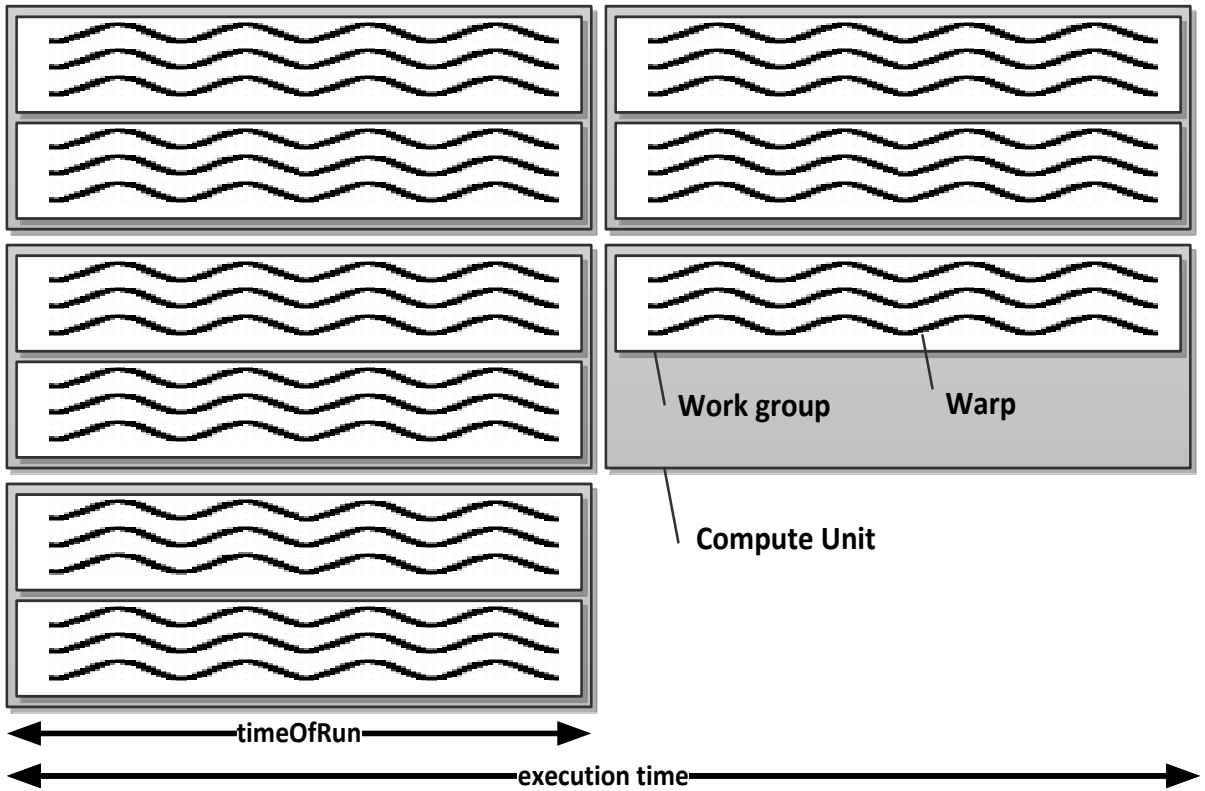


Multiple WGs per MP

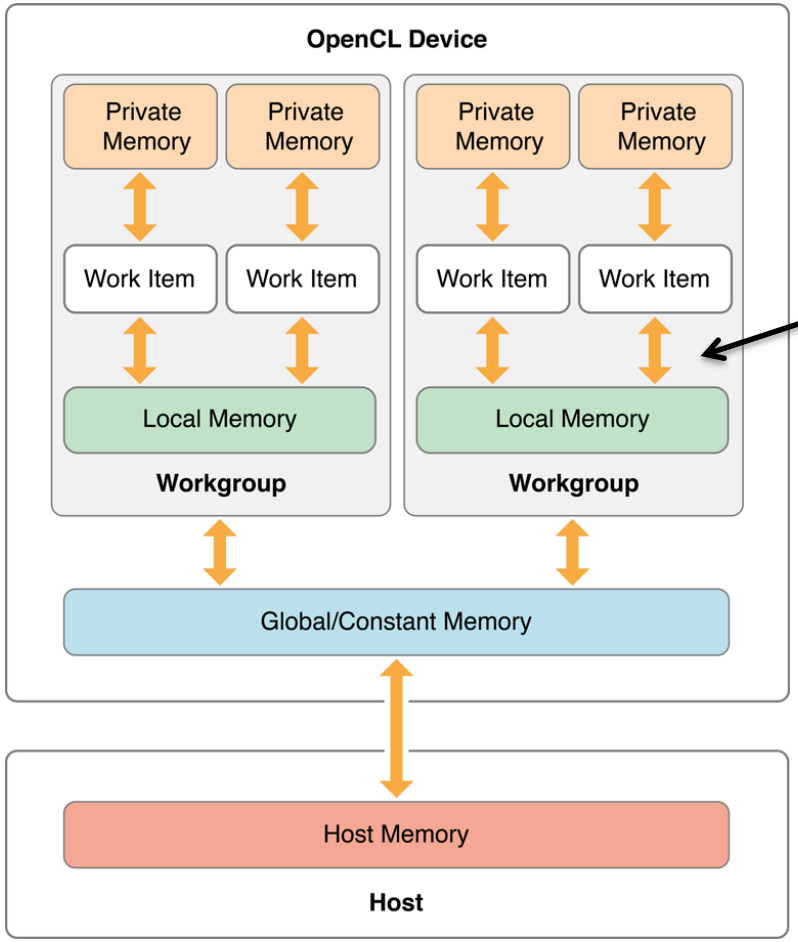
- One MP can execute work groups concurrently
- Determined by available resources (hardware limits):
  - *Max. work groups simultaneously on MP: 8*
  - *Private memory (registers) per MP: 16/48KB*
  - *Local (shared) memory per MP: 16/32KB*



The execution on a GPU



Architecture – Memory Model



Multiprocessor

OpenCL Memory Model on NVIDIA

Software

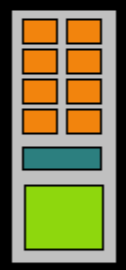
__private

Hardware

Scalar Processor

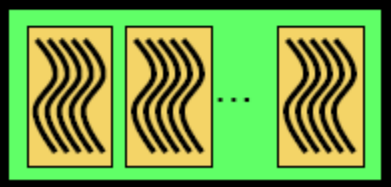
- Each hardware thread has a dedicated __private region for stack

__local



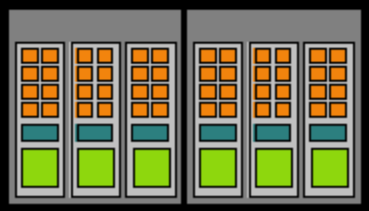
Multiprocessor

- Each multiprocessor has dedicated storage for __local memory and __constant caches
- Work-items running on a multiprocessor can communicate through __local memory



__global

_constant



Device

- All work-groups on the device can access __global memory
- Atomic operations allow powerful forms of global communication

Memory requirements

- # local variables per thread (registers)
- # work items per work group
- => memory per work group

Work group execution

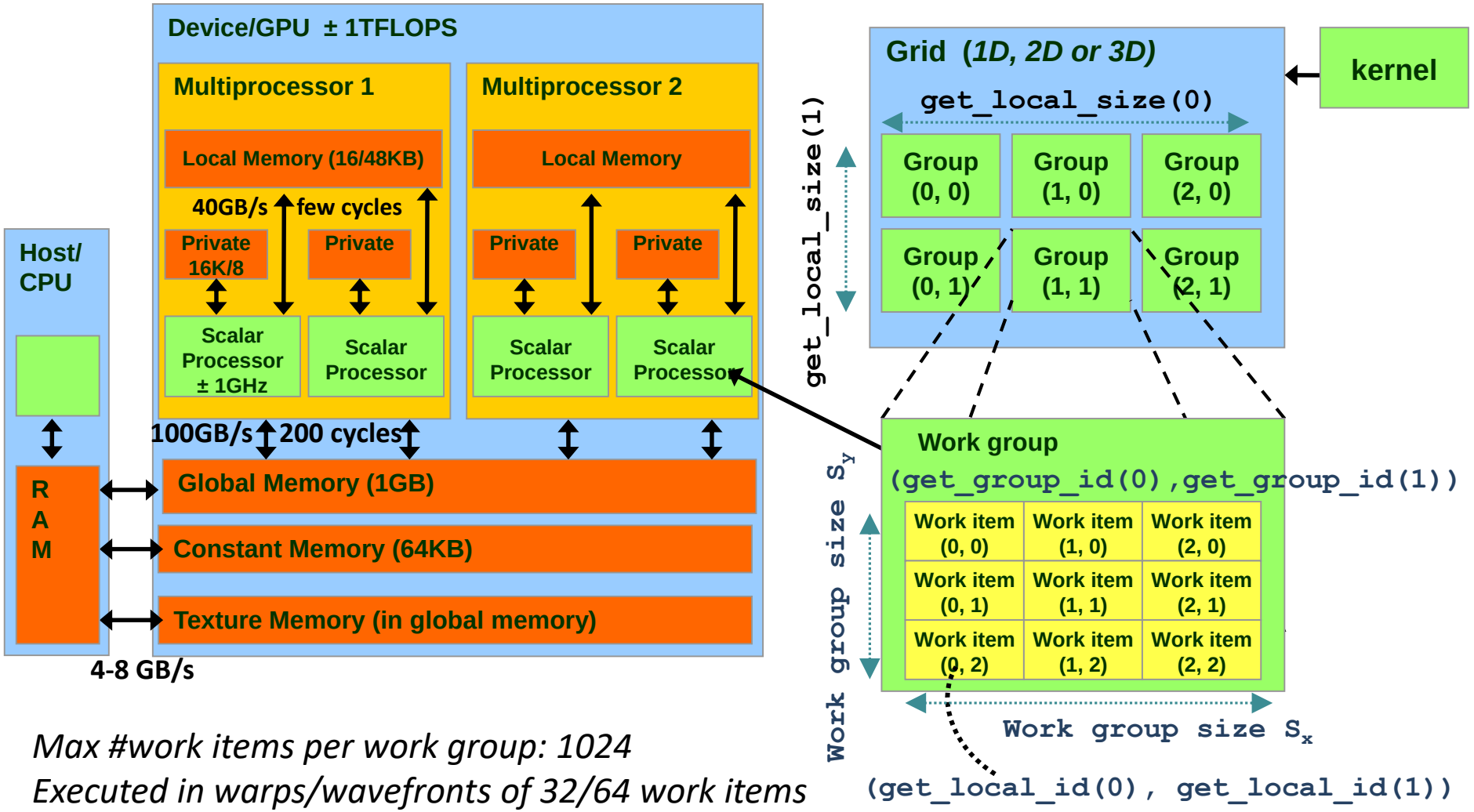
- Work items can synchronize <u>within a work group</u>

```
barrier(CLK_LOCAL_MEM_FENCE);  // barrier synchronization
```

- Work items can share on-chip local memory
  - Local memory is on MultiProcessor (MP)
  - Visible to work group only

```
__local int shr[NUMBER_OF_ROWS][NUMBER_OF_COLS];
```

GPU Programming Concepts



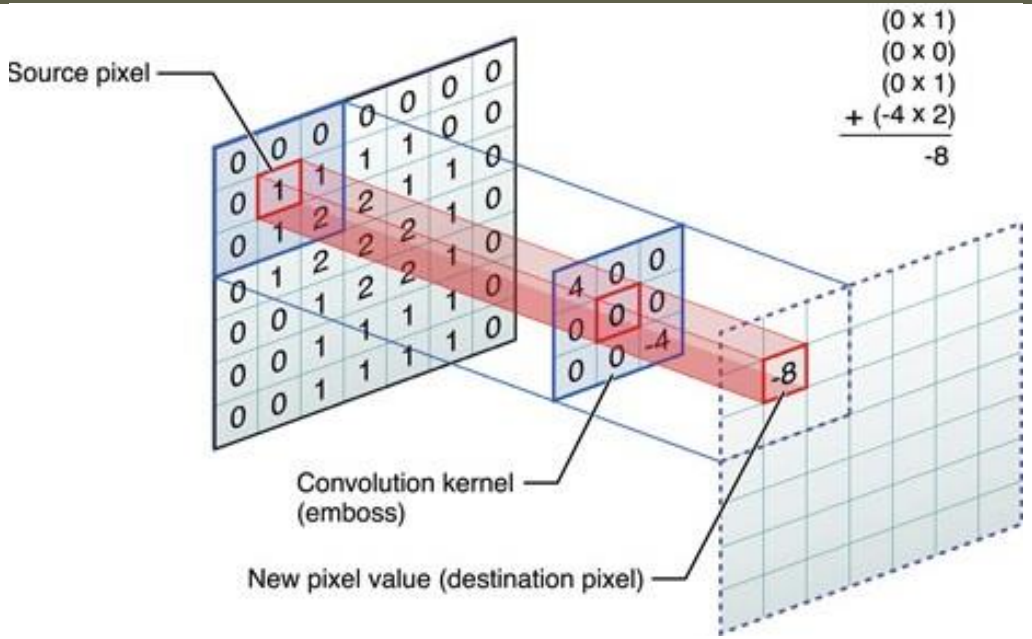
Max #work items per work group: 1024
Executed in warps/wavefronts of 32/64 work items
Max work groups simultaneously on MP: 8
Max active warps/wavefronts on MP: 24/48

OpenCL terminology

GPU Threads v/s CPU Threads

- GPU work items or *threads*:
  - **Lightweight,** small creation and scheduling overhead, extremely **fast switching between threads**
    - No context switch is required
  - **Need to issue 1000s of GPU threads** to hide global memory latencies (600-800 cycles)
    - GPU=Thread processor, upto 96 threads per processor
- CPU threads:
  - Heavyweight, large scheduling overhead, **slow context switching** (processor state has to be saved)

Example: convolution



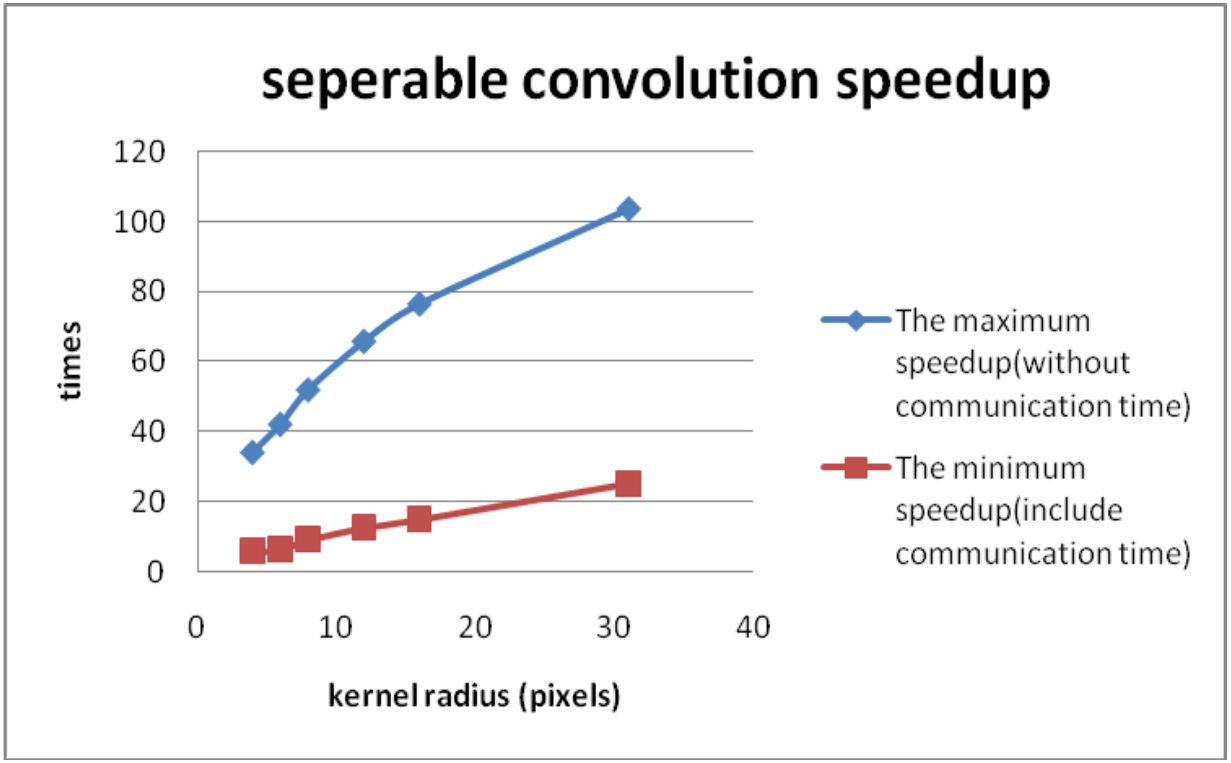
Parallelism: +++
Locality: ++
Work/pixel: ++

3x3 kernel (also called *filter* or *mask*) is applied to each pixel of the image

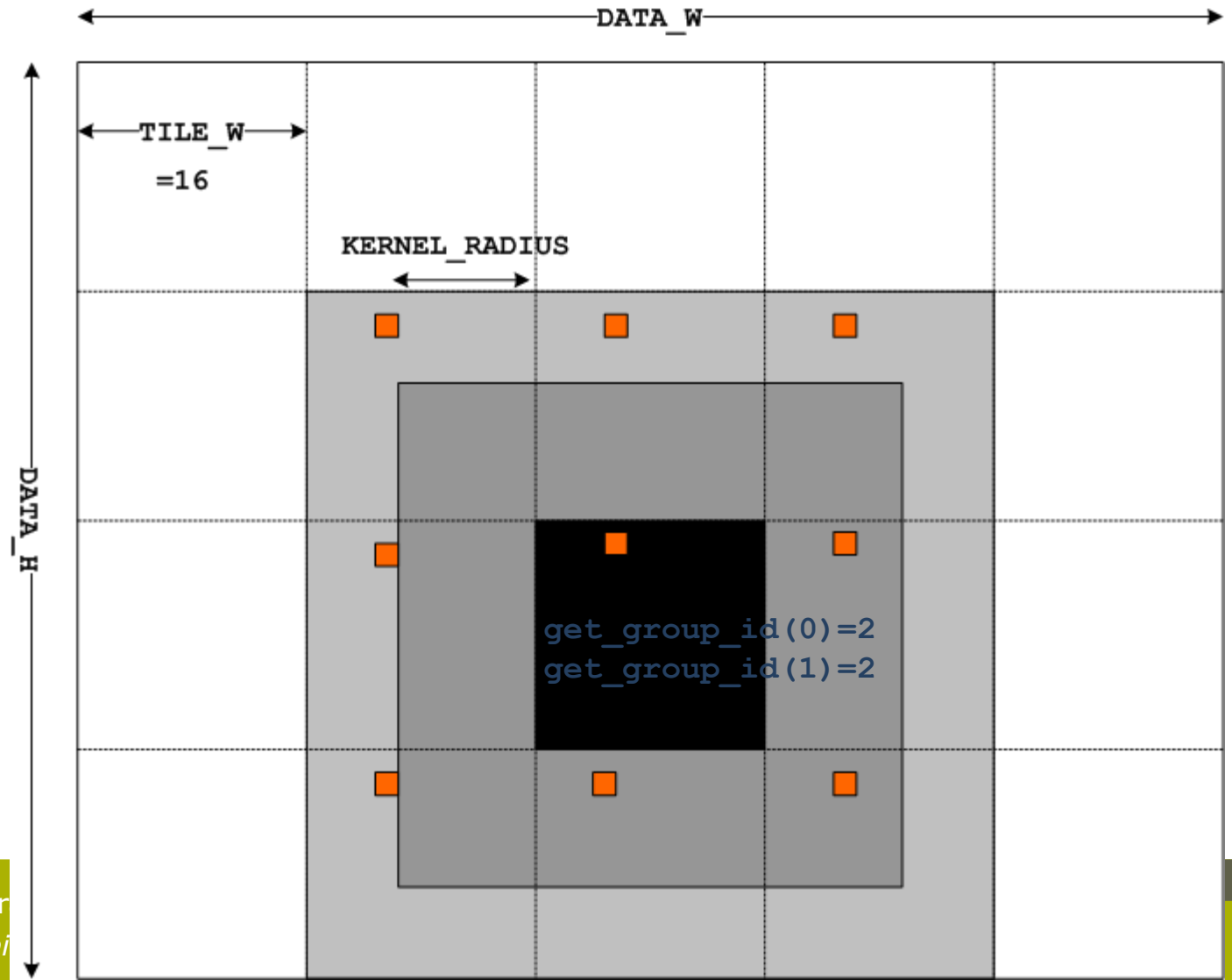
Examples of convolution

Edge detection
with sobel filter

Speedup


seperable convolution speedup
120
100
80
60
40
20
0
times
0
10
20
30
40
kernel radius (pixels)
The maximum speedup(without communication time)
The minimum speedup(include communication time)

Convolution on GPU

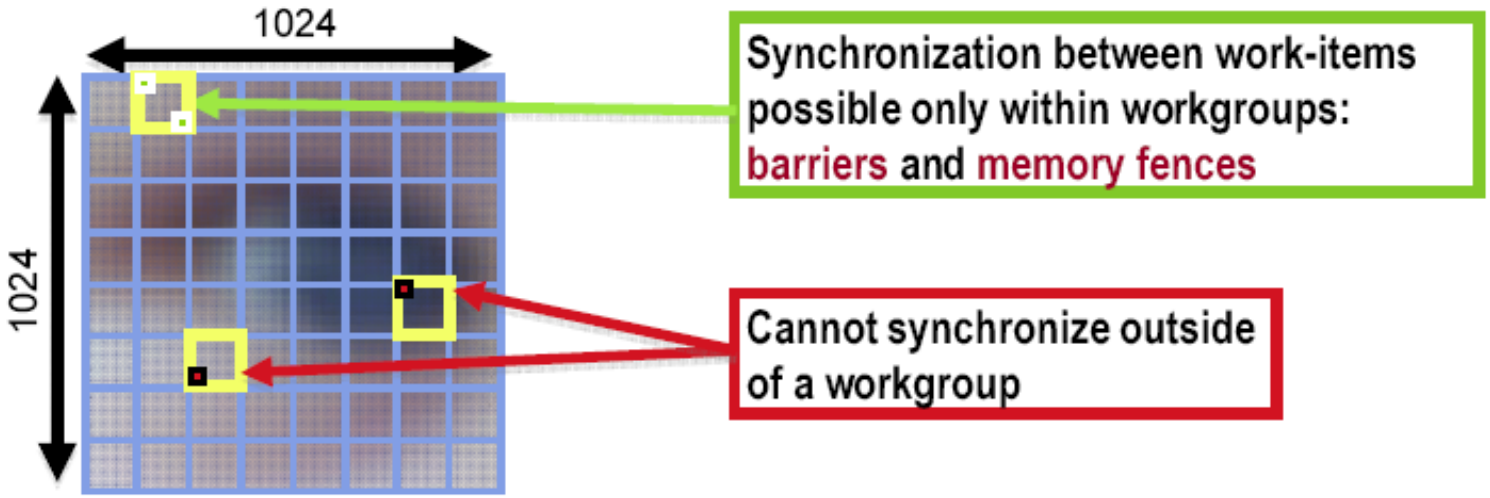


Convolution Kernel Code

```
__kernel void convolutionUsingSharedMemory(
        __global int *in, __global int *out, __local  int *in_local, __constant int *filter, int
filter_height, int filter_width)
{
    uint row = get_global_id(1);
    uint col = get_global_id(0);

    in_local[get_local_id(1) * get_local_size(0) + get_local_id(0)] =
        in[row * get_global_size(0) + col];
    … // copy 9 pixels to local

    barrier(CLK_LOCAL_MEM_FENCE);
    int sum=0;
    for (int i = 0; i< filter_height; ++i)
        for (int j = 0; j< filter_width; ++j)
            sum += filter[…] * in_local[…];

    out[row * get_global_size(0) + col] = sum;
}
```

Global and Local Dimensions

- Global Dimensions: 1024 x 1024 (whole problem space)
- Local Dimensions: 128 x 128 (executed together) by a work group)



- Choose the dimensions that are "best" for your algorithm

Overview

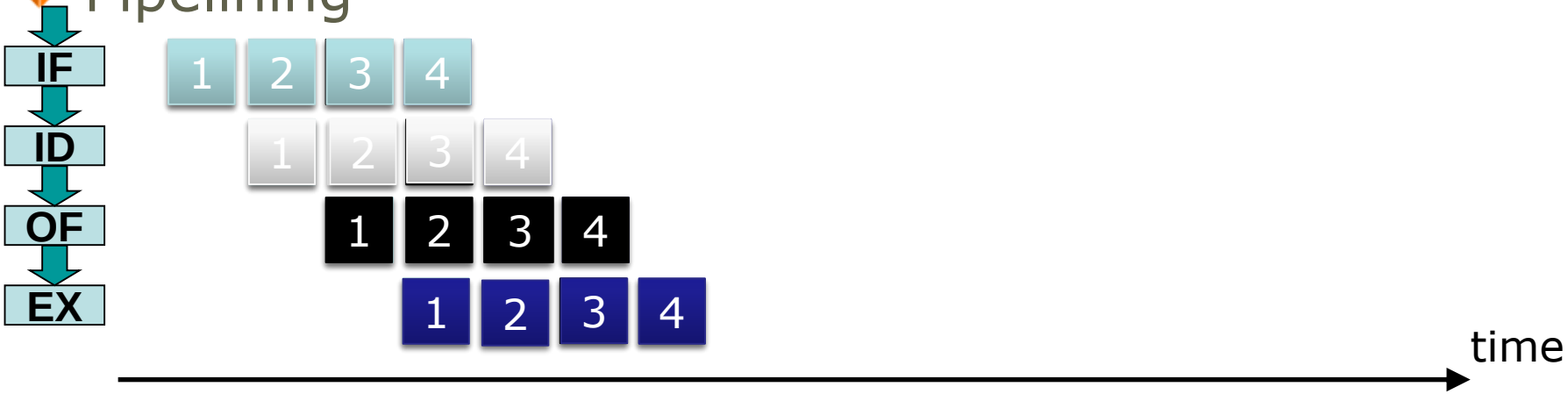
1. GPUs for general purpose
2. GPU 'threads' executing kernels
3. Starting kernels from host (CPU)
4. Execution model & GPU architecture
5. **Warps/wavefronts**
6. Optimizing GPU programs
7. Analysis & Conclusions

Pipelining Principle

- Long operations

- Combination of short operations

- Pipelining

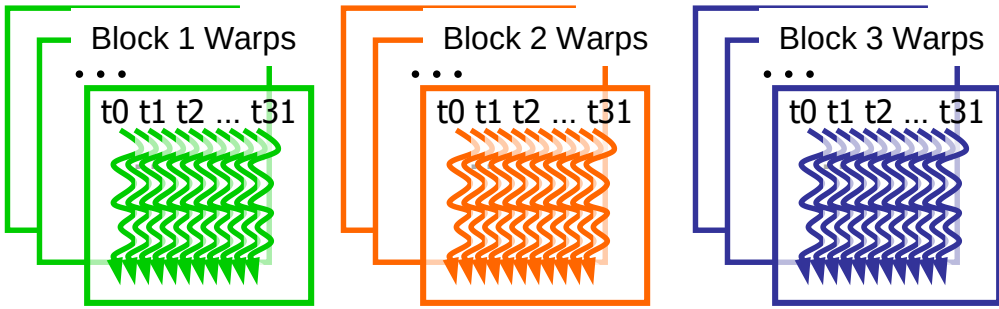


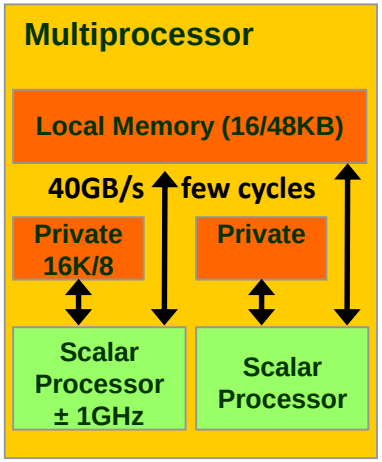
Pipelining

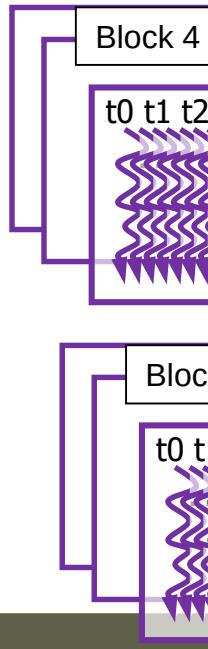
- On GPU: 24 stages
- Multiple instructions simultaneously in flight
- Higher throughput, *except* if dependencies between threads
  - E.g.: If instruction 2 depends on the outcome of instruction 1, then instruction 2 can only proceed in pipeline after the termination of instruction 1
  - **Pipeline stall**
- GPU: instructions of *different* threads in flight

Work group execution

- Work items are executed in NVIDIA-**warps**/AMD-**wavefronts**, they are the scheduling units in the MP.
- Groups of 32/64 work items that execute in lockstep: they execute the same instruction.



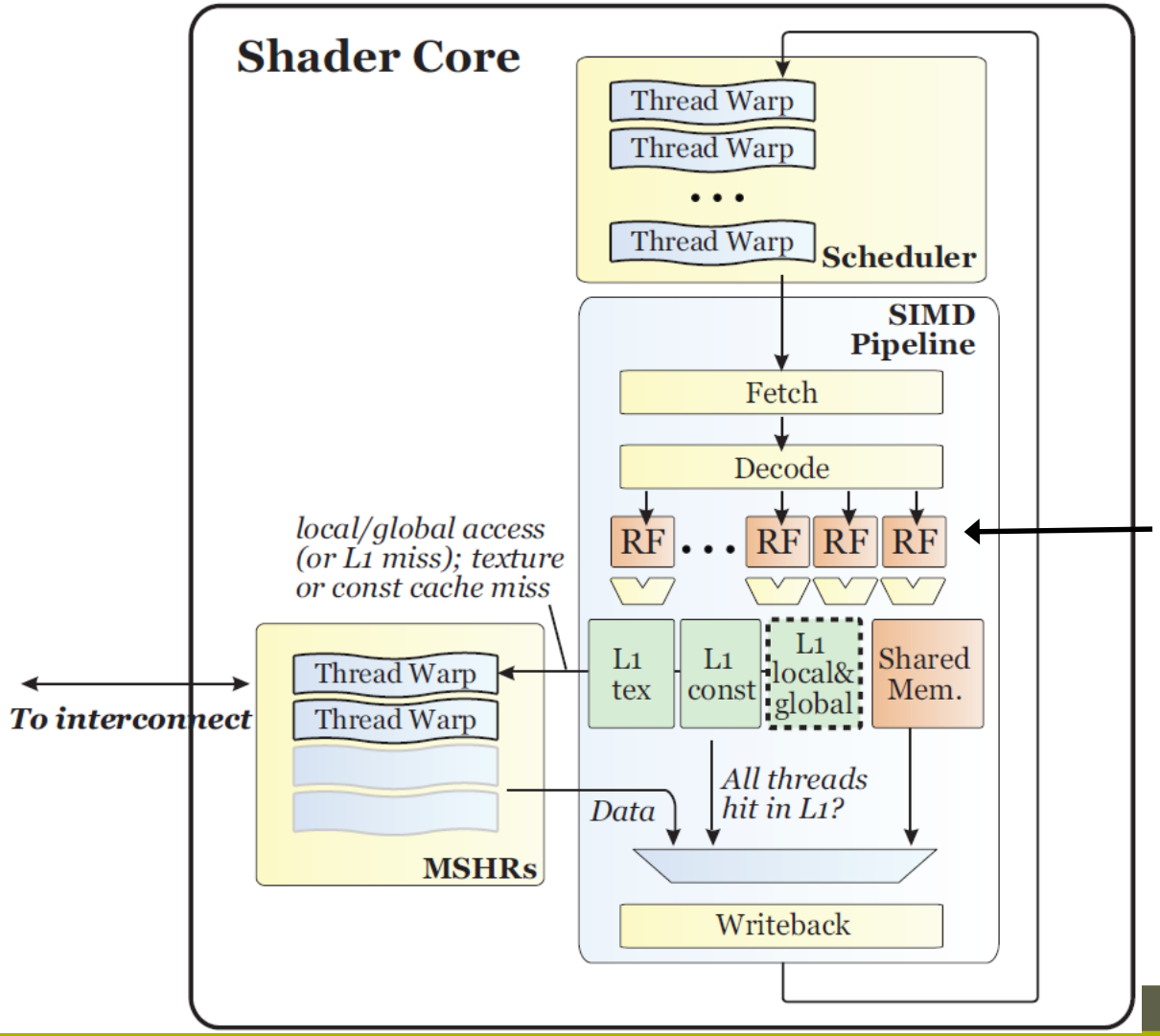




Example: 3 work groups on MP, each group has 256 work items, how many Warps are there in the MP?

- Each group is divided into 256/32 = 8 Warps
- There are 8 * 3 = 24 Warps

Warp scheduling

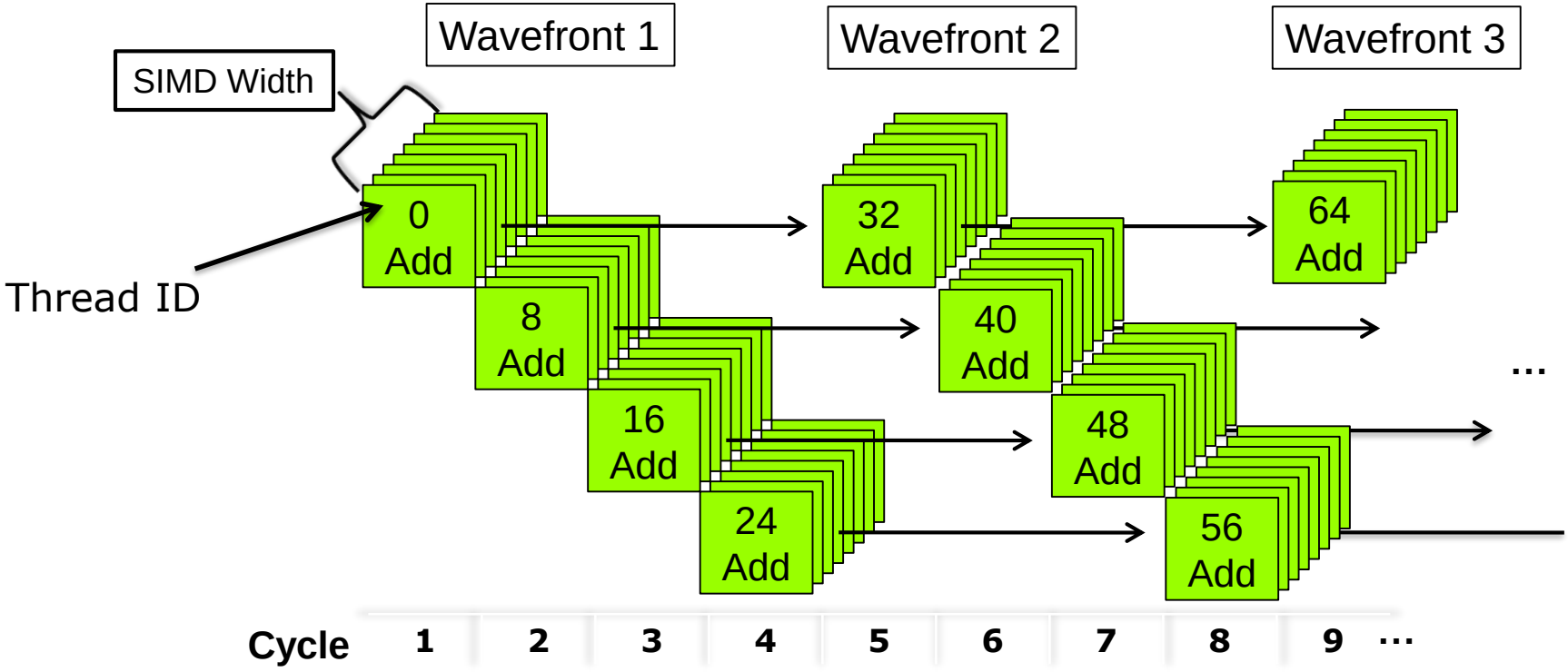


The Same Instruction is executed on Multiple Threads (SIMT)

width of pipeline: 8 – 32 – 192 - 128

Warp/wavefront execution

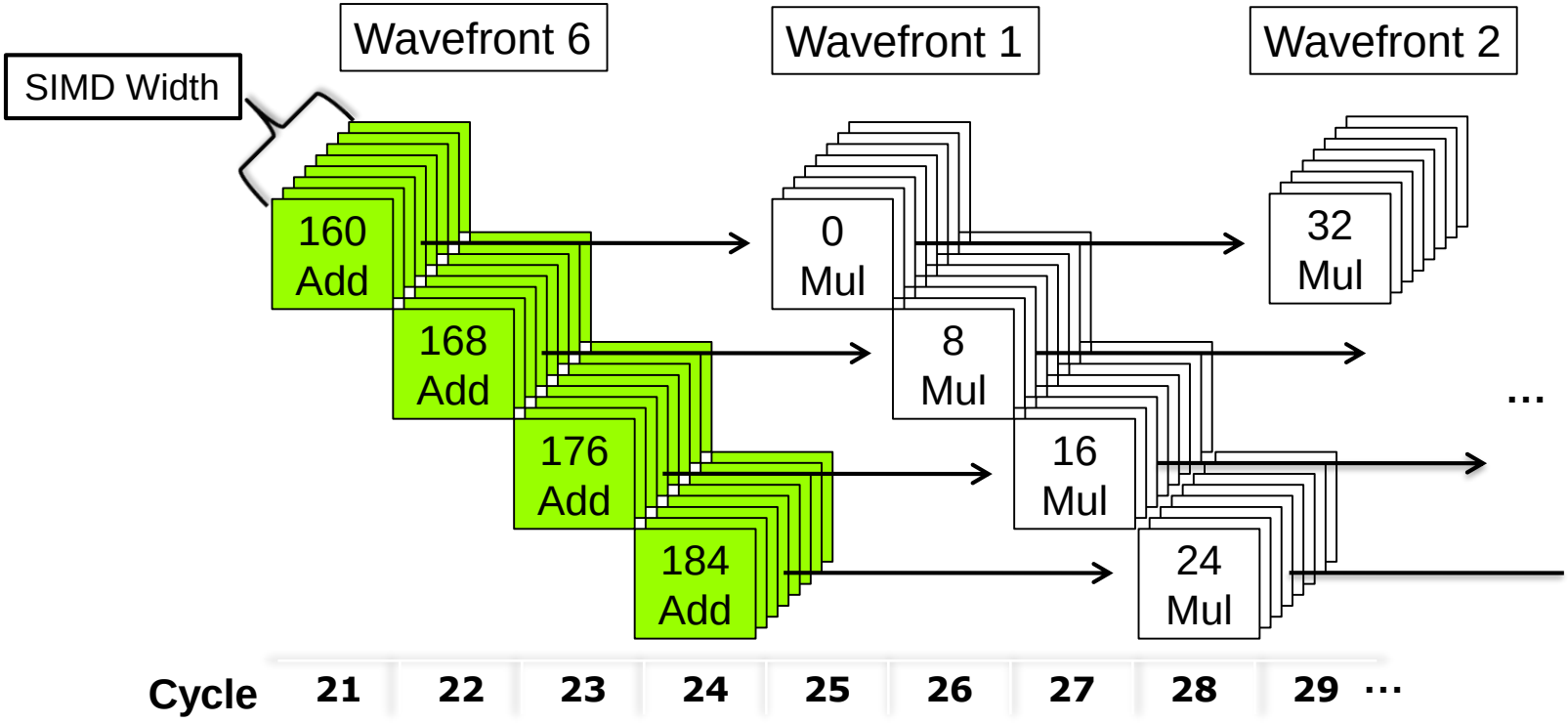
- Work items are sent into pipeline grouped in warp /wavefront
  - ALUs all execute the same instruction: Single Instruction, Multiple Threads (SIMT)
  - 32 work items / 8 SPs => 4 cycles



Warp/wavefront execution

- Kernels proceeds to next instruction if all warps are in the pipeline
  - If 192 work items => 6 warps => 24 cycles needed

  ⇒ pipeline has independent instructions => no stalling



Why do we have to consider warps/wavefronts?

- Different **branching** of threads within a warp incurs 'lost cycles' (*see next section*)
  - SIMT execution happens per warp/wavefront, in lockstep
- **Memory access pattern** of a warp should be considered
  - Memory access also happens per warp
  - Not all access patterns can happen concurrently (*see further*)

Overview

1. GPUs for general purpose
2. GPU 'threads' executing kernels
3. Starting kernels from host (CPU)
4. Execution model & GPU architecture
5. Warps/wavefronts
6. **Optimizing GPU programs**
7. Analysis & Conclusions

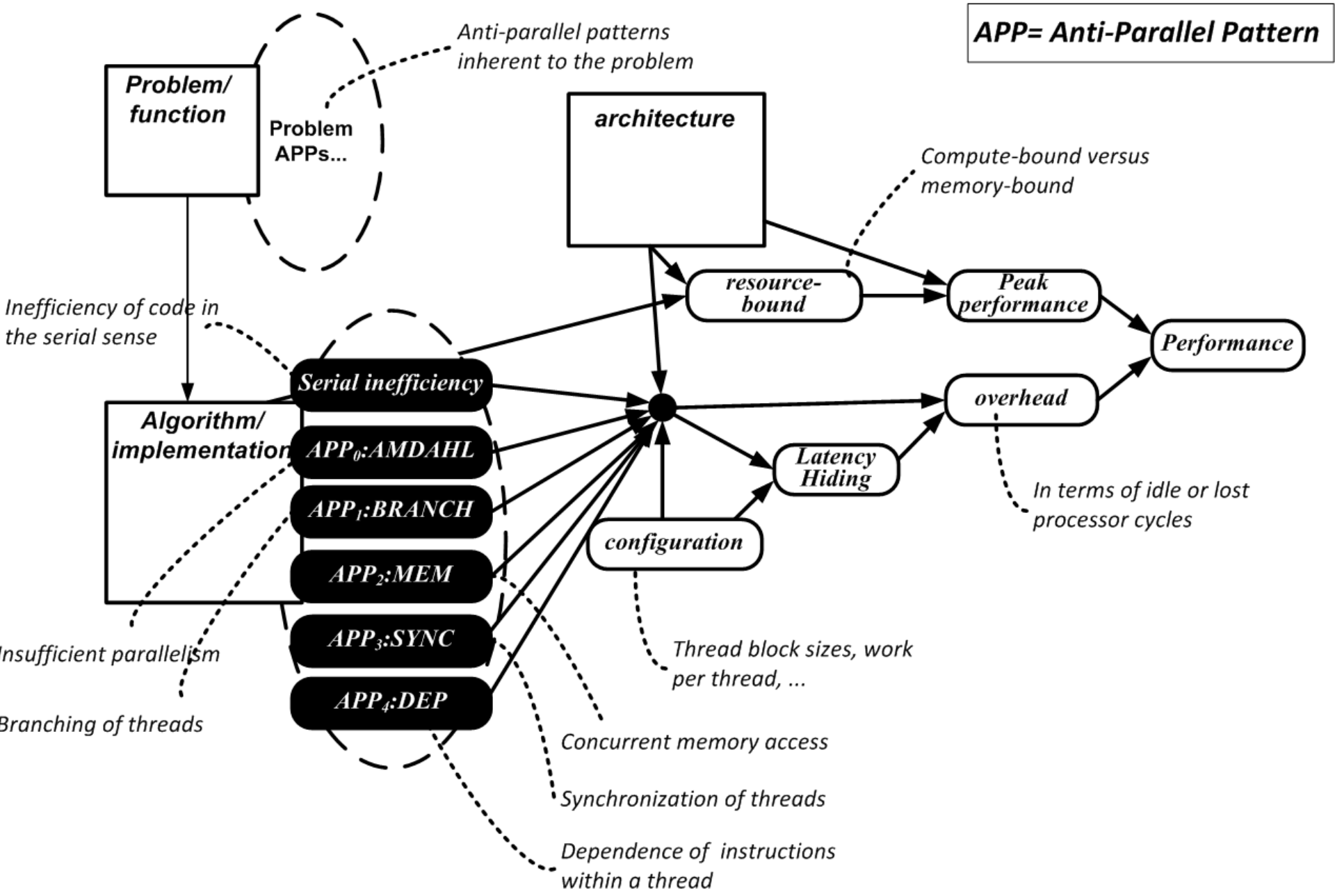
Example of optimization process starting with a naïve version

Performance for 4M element reduction

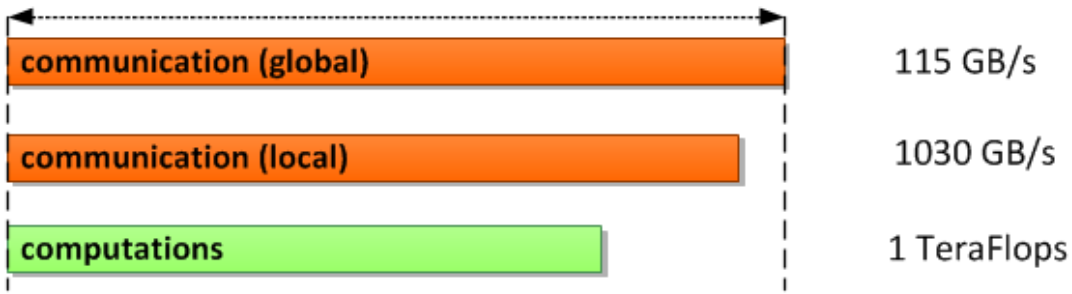
	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Kernel 7 on 32M elements: 72 GB/s!

GPU Computing Performance

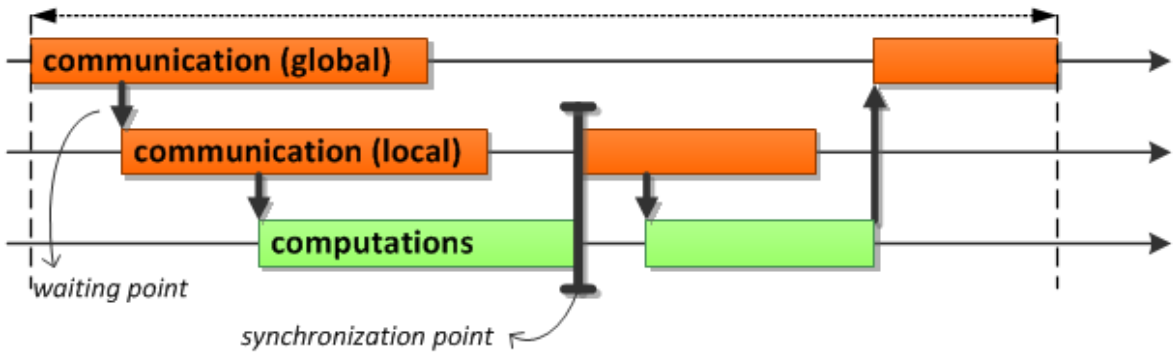


A. Peak Performance



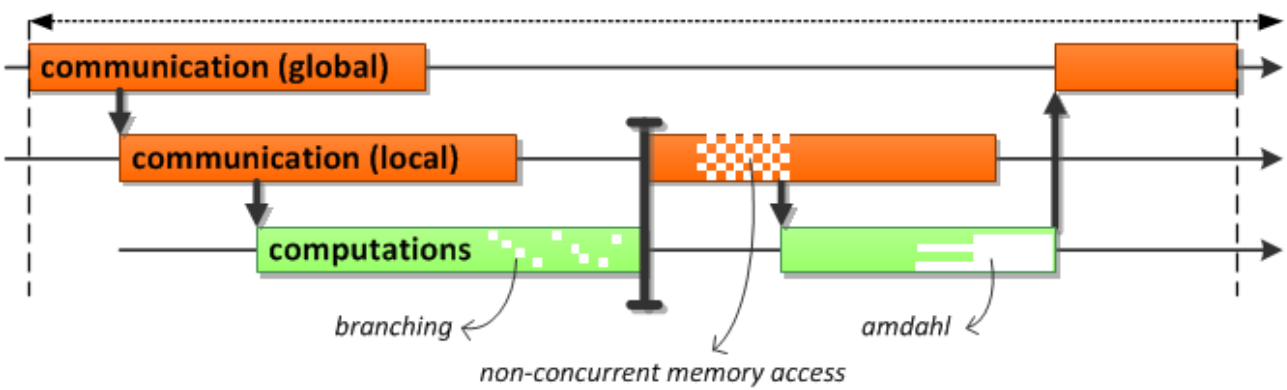
Roofline model

B. Non-overlap



Non-overlap factors

C. Anti-parallel interactions



Anti-parallel patterns & model for latency hiding

General approach

- Estimate a performance bound for your kernel
  - ***Compute bound***: t_1 = #operations / #operations per second
  - ***Data bound***: t_2 = # memory accesses / # accesses per second
  - Minimal runtime $t_{min} = \max(t_1, t_2)$
    <u>*expressed by roofline model*</u>
- Measure the actual runtime
  - $t_{actual} = t_{min} + t_{delta}$
- Try to account for and minimize t_{delta}
  - Due to non-overlap of computation and communication
  - Due to *anti-parallel patterns (APPs)*
  - Consult remedies for APPs

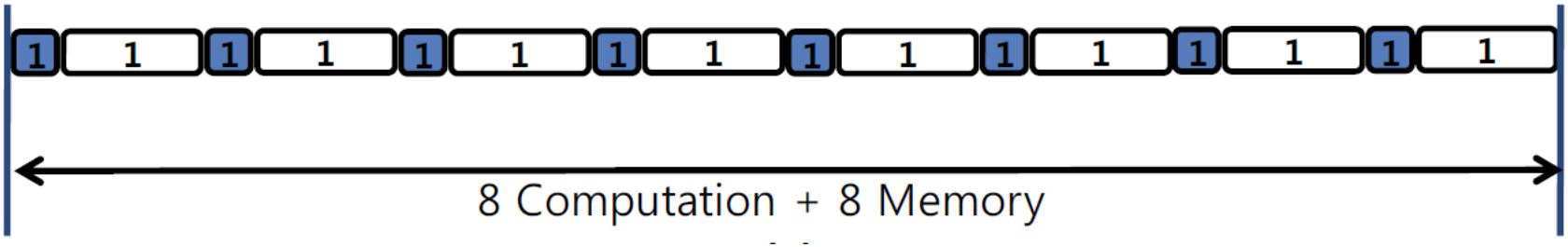
Anti-parallel Patterns

- Definition: *Common parts of kernel code that work against the available parallelism*.
  - Can be inferred from the source code
  - Map parts of the source to parallel overhead
- Systematic way to categorize performance topics
- Systematic way to optimize kernels
- *PhD research of Jan G. Cornelis*

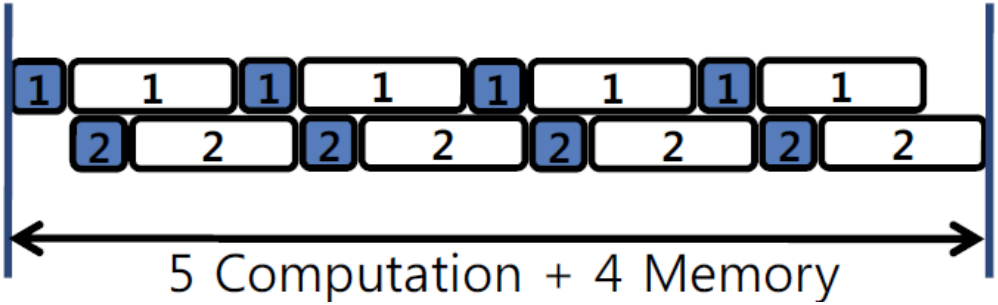
Anti-parallelism results in sequential execution; not all parallel resources are used

Latency Hiding

- 1 warp, without latency hiding



- 2 warps running concurrently



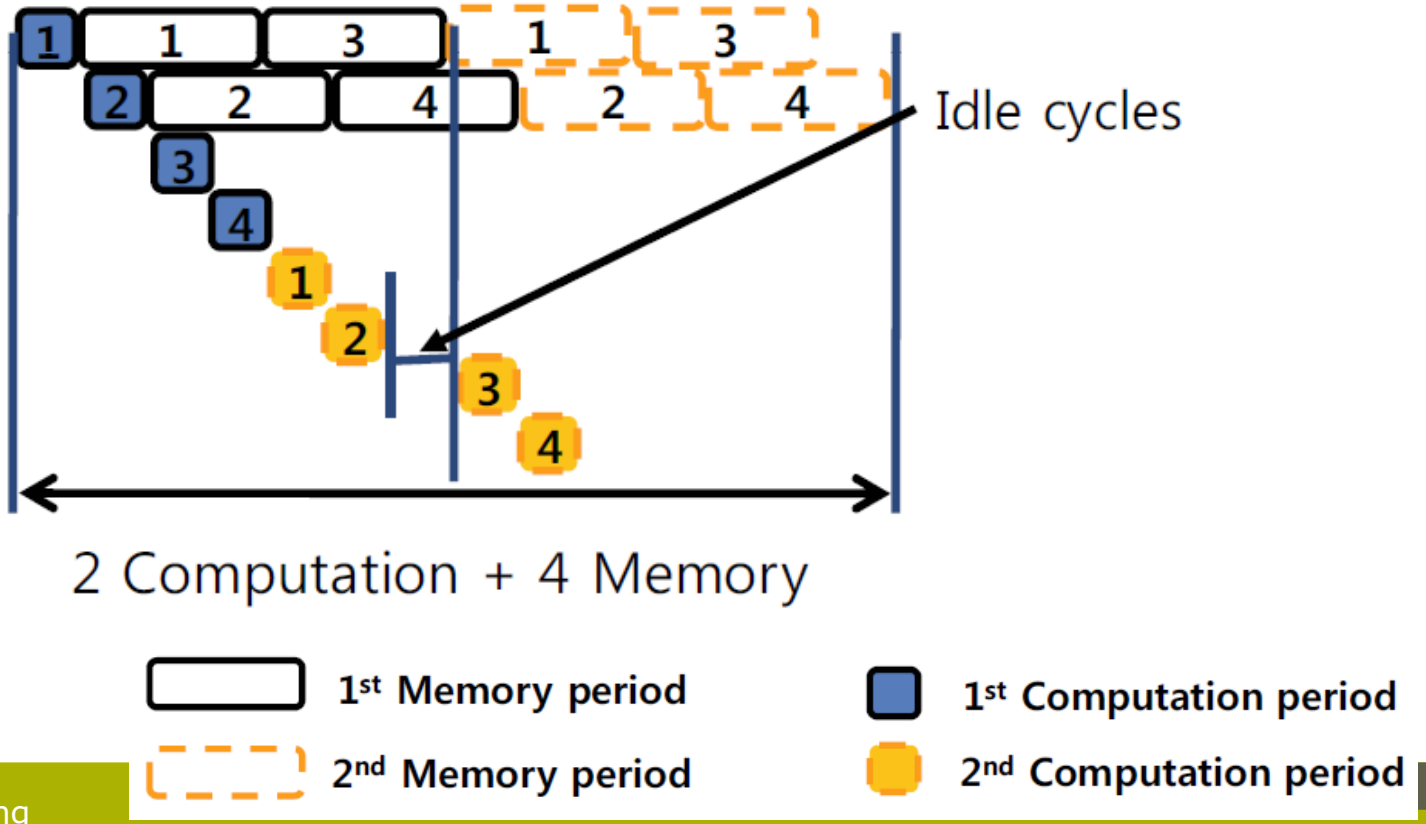
- 4 warps running concurrently: full latency hiding

Latency Hiding for Memory Accesses

- Latency Hiding
  - During global to local memory copying
  - During local memory reads
- Keep multiprocessors busy with a huge amount of threads
  - 1 multiprocessor can simultaneously execute multiple work group of maximal 512/1024 work items
  - Is limited by amount of local and register memory needed by each work item
  - ***Maximize occupancy***= Number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently

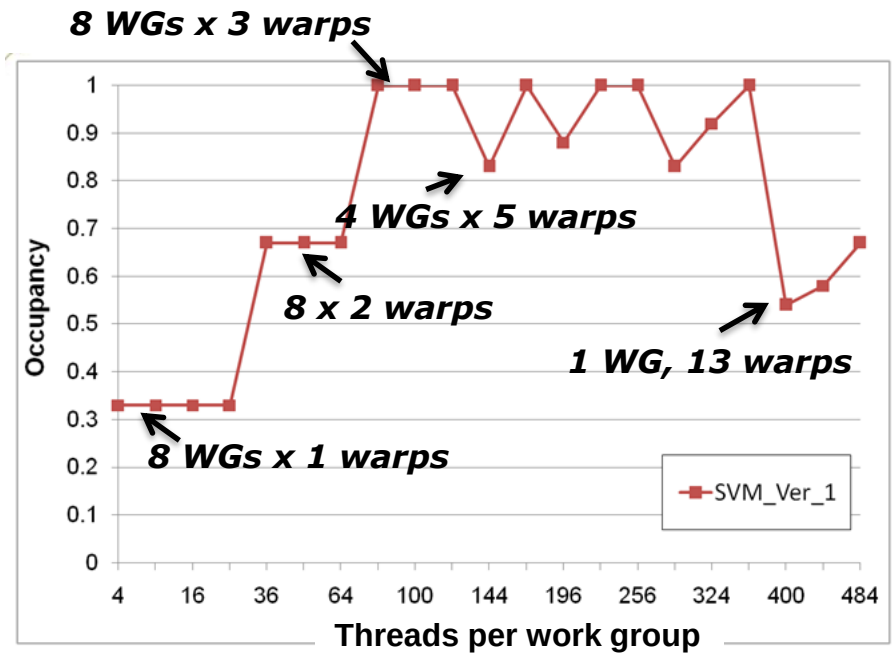
4 warps running concurrently

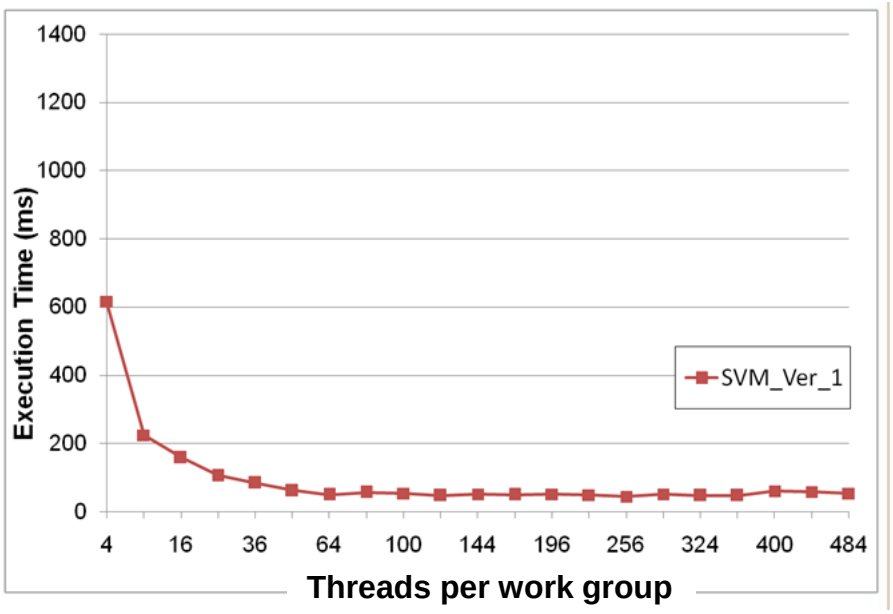
But only 2 concurrent memory transactions…



Keep occupancy high

- Maximal warps: 24, maximal Work Groups (WGs): 8





- Conclusion: in general, higher occupancy leads to a better performance

Insufficient parallelism

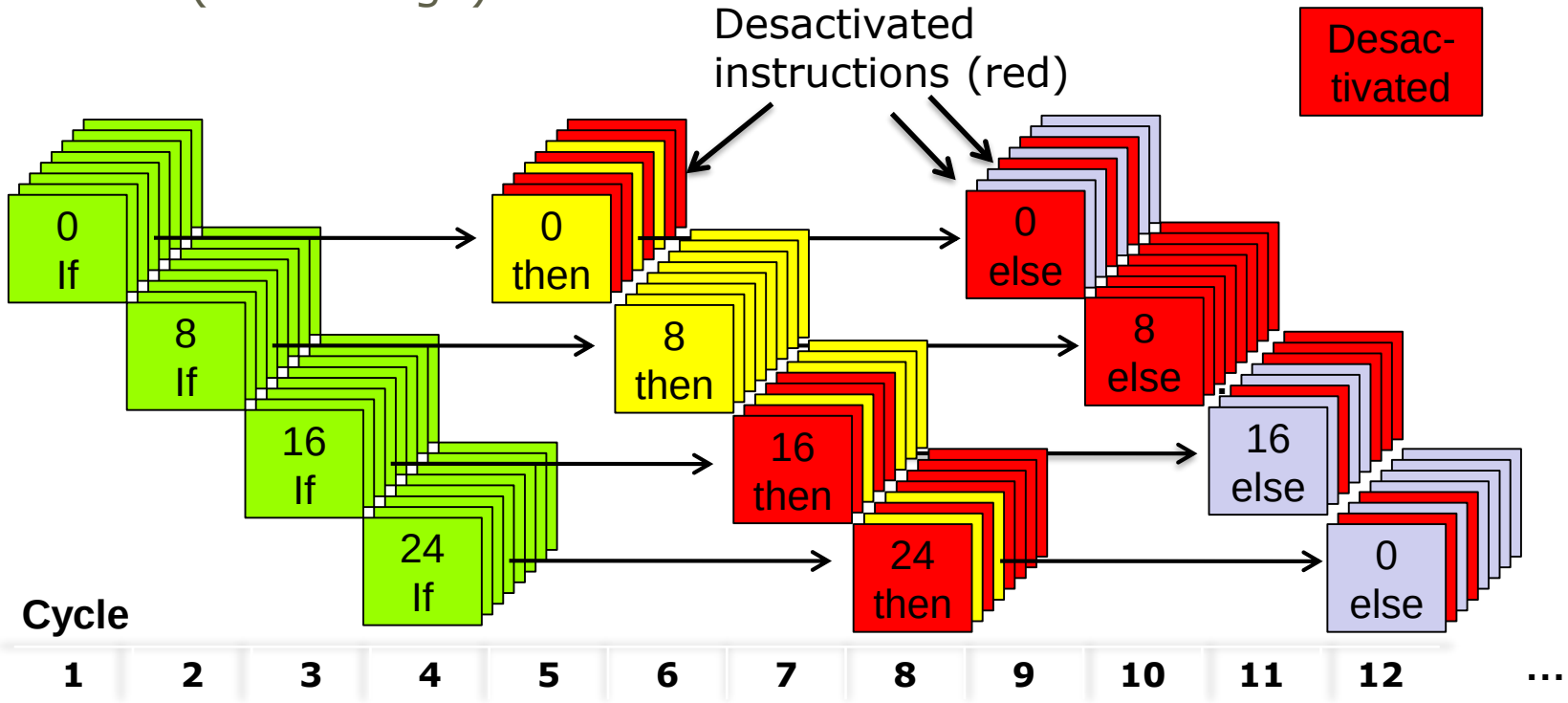
- Processor needs sufficient work groups/work items to keep the system busy, to keep all pipelines full; to get full performance.
- We'll discuss Amdahl and the law attributed to him in more depth in the lecture on Performance Analysis
- if GPU is not fully used, additional work can be scheduled without cost
  - See earlier slide with graph of runtime in function of the number of threads for vector addition
    - the runtime does not increases as long as GPU is not full.
    - ➡ function shaped as a staircase
    - only just before the jump to the next step the GPU is fully busy

SIMT Conditional Processing

- Unlike threads in a CPU-based program, **SIMT programs cannot follow different execution paths**
  - All threads of a warp/wavefront are executing the same instruction
- **Ideal scenario:**
  - **All GPU threads of a work group follow the same execution path**
  - **All processors continuously active**
- If divergent paths within a warp/wavefront, **the *then*- and *else*-instructions are scheduled executed** for all threads, but only executed for the correct threads, dependent on the condition
  - Program flow cannot actually diverge, a bit is used to enable/disable processors based on the thread being executed (*instruction predication*)
- **Parallelism is reduced**, impacting performance… (see later)

SIMT Conditional Processing

- **Example**: assume only one warp, one instruction in if-clause, one in then-clause
  - 12 cyles in which 64 instructions are executed, 32 lost cycles (66% usage)



Branching

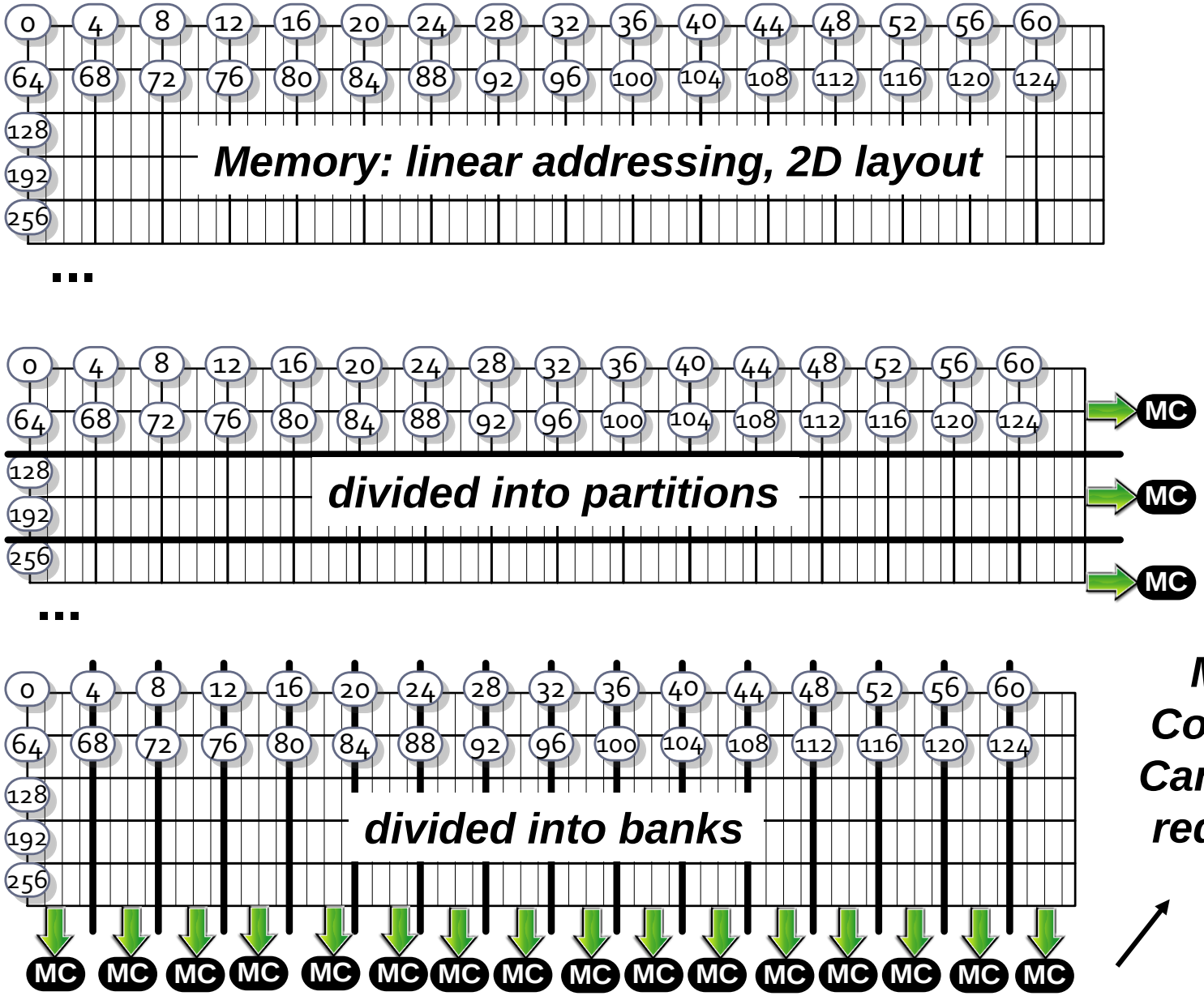
- Threads of the same warp/wavefront (32/64 threads) are run in lockstep
- For example:

```
if (x < 5)  y = 5; else y = -5;
```

  - SIMD performs the 3 steps
  - `y = 5;` is only executed by threads for which `x < 5`
  - `y = -5;` is executed by all others
- *Warp branch divergence* decreases performance: <u>cycles are lost</u>
  - **Possible solution**: statically or dynamically reorder threads such that all threads of a warp follow same branch
- No latency hiding possible

Global memory

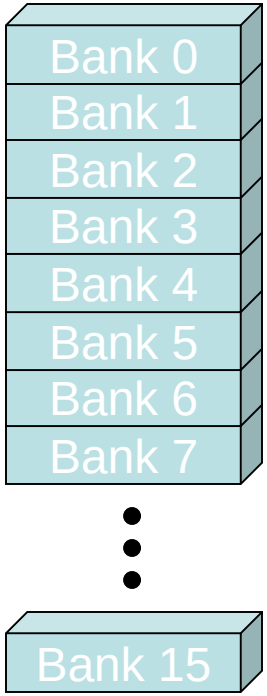
- Memory coalescing for warps
  - Accessed elements belong to same aligned segment
  - Older cards: sequential threads access sequential locations
  - Newer cards: not necessary anymore
- Global memory is a collection of partitions
  - 200 series and 10 series NVIDIA GPUs have 8 partitions of 256 bytes wide
  - Partition camping when different thread Work groups access the same partition



Memory Controllers: Can handle 1 request at a time

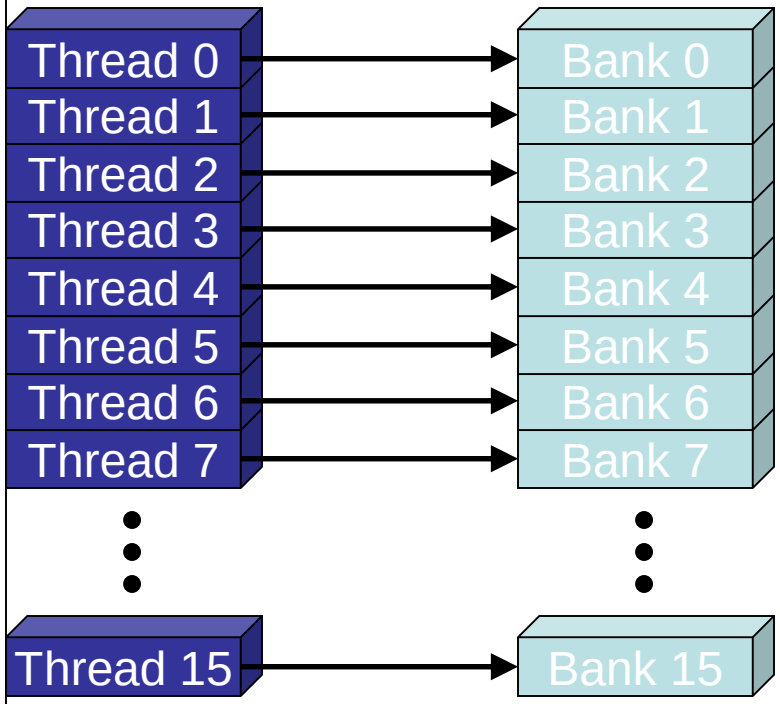
Local/Shared memory

- Local/Shared memory is divided into banks
- Each bank can service one address per cycle
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized
  - Cost = max # simultaneous accesses to single bank
- No bank conflict:
  - all threads of a half-warp access different banks,
  - all threads of a half-warp access identical address, (broadcast)

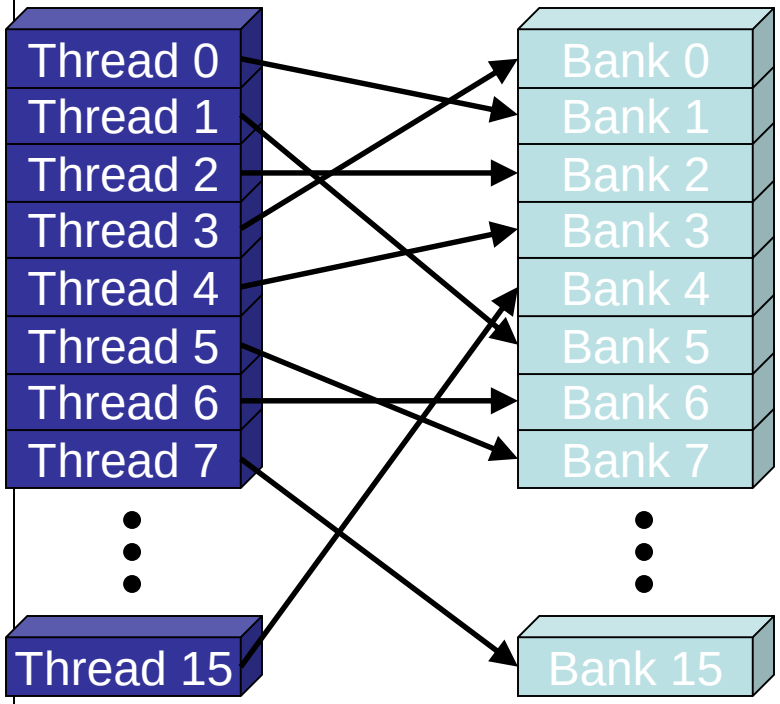


Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride of 1

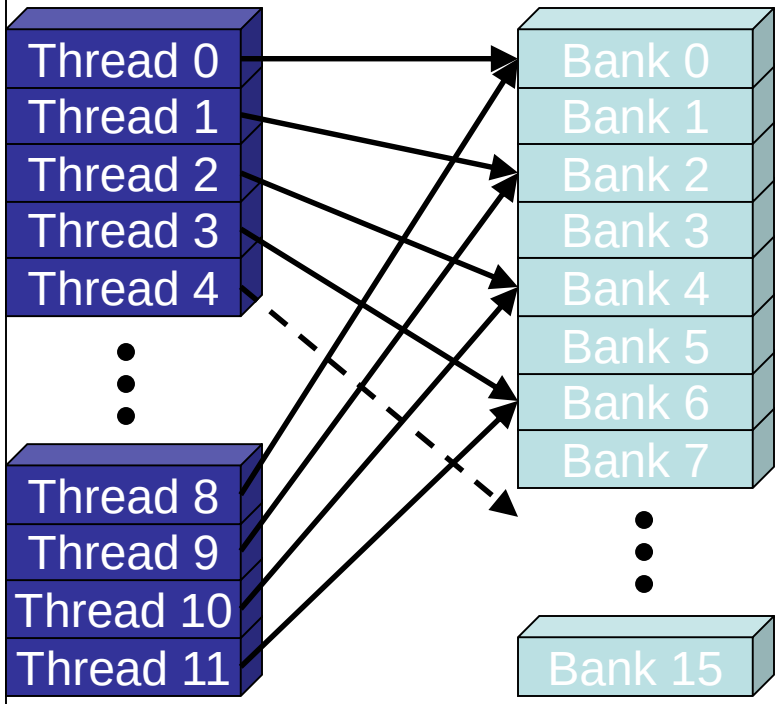


- No Bank Conflicts
  - Random 1:1 Permutation

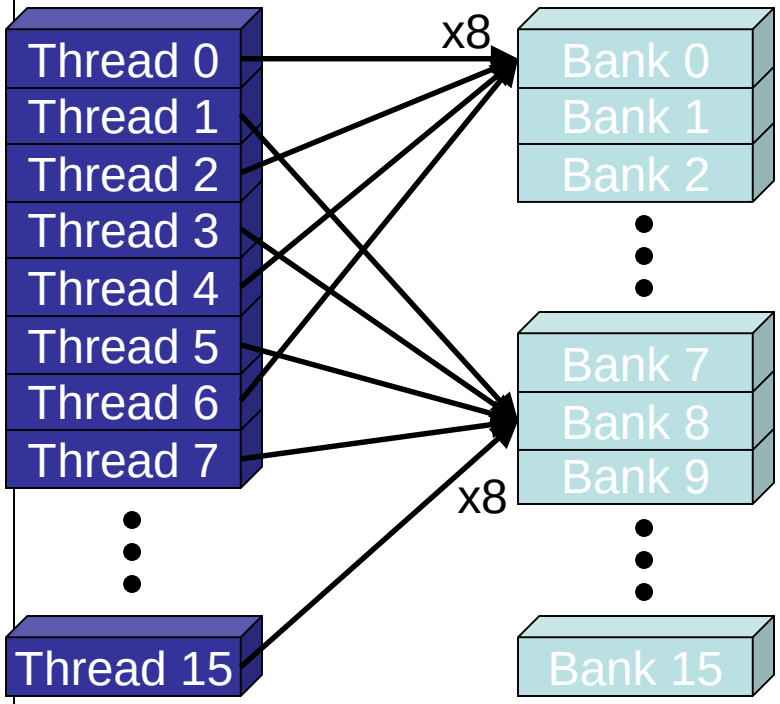


Bank Addressing Examples

- 2-way Bank Conflicts
  - Linear addressing stride of 2



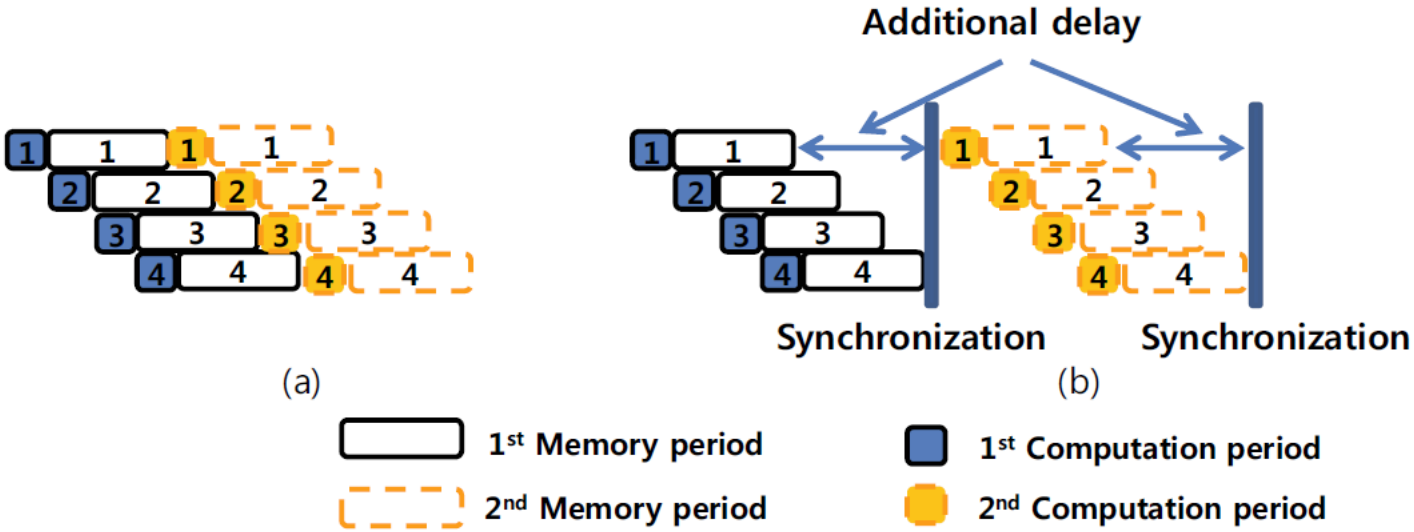
- 8-way Bank Conflicts
  - Linear addressing stride of 8



Synchronization

- Barrier synchronization within a work group
  - barrier(CLK_LOCAL_MEM_FENCE);
  - Work items that reached the barrier must wait
- Global synchronization should happen across kernel calls
  - Work groups that have completed
- Greater instruction dependency
  - ➔ less potential for latency hiding
- Thus: try to minimize synchronization

Lost cycles due to synchronization



Tree traversal

- Each work item follows a different path in a tree, from root to leave.
  - While-loop
- If not all leaves are at the same depth: the highest depth determines the execution time of a warp/wavefront
- Imbalances in trees result in many lost cycles

Dependent Code

- Well-known fact: latency is hidden by launching other threads
- Less-known fact: one can also exploit *instruction level parallelism* in one thread.
  - Data level parallelism in one thread.
- Anti-parallel pattern?
  - Dependent instructions can not be parallelized.
  - Dependent memory accesses can not be parallelized.

AMD's static kernel analyzer

File Edit Help

Source Code

Function: main

```
1 inline void waveReduce(__local volatile EL
2     #if WGSIZE >= 128
3     if (lid < 64) {
4         shared_v[lid] += shared_v[lid + 64
5     #endif
6
7     #if WGSIZE == 64
8     if (lid < 32) {
9     #endif
10
11        shared_v[lid] += shared_v[lid + 32
12        shared_v[lid] += shared_v[lid + 16
13        shared_v[lid] += shared_v[lid + 8]
14        shared_v[lid] += shared_v[lid + 4]
15        shared_v[lid] += shared_v[lid + 2]
16        shared_v[lid] += shared_v[lid + 1]
17    }
18 }
19
20 // |-----------------------------------
21 // - load and directly sum elements from g
22 //   local memory (balances ALU:Fetch)
23 // - reduce in local memory
24 // - write back to host memory
25 // ! requires a warp/wavefront size of at
26 //   synchronization for work items within
27 __kernel __attribute__((reqd_work_group_si
28 void main(__global ELTYPE* source, __local
29     const size_t gid = get_global_id(0);
```

Compile

Source type: OpenCL

OpenCL Compiler

Options: -fbin-amdil

Macro Definitions

Symbol	Value
WGSIZE	256u
SUMONLOAD	8u
ELTYPE	uint4

Object Code

Format: Radeon HD 5870 (Cypress) Assembly

```
1 ; -------- Disassembly ----------------
2 00 ALU: ADDR(32) CNT(17) KCACHE0(CB0:0-15
3       0  w: LSHL          ____,  R0.x,  4
4          t: ADD_INT       R0.w,  R1.x,  KC0
5       1  y: ADD_INT       R0.y,  KC1[1].x,
6          t: MULLO_INT     ____,  R1.x,  KC0
7       2  y: ADD_INT       R10.y,  PV1.y,  4
8          z: ADD_INT       ____,  R0.x,  PS1
9          w: ADD_INT       R10.w,  PV1.y,  1
10         t: ADD_INT       R0.z,  PV1.y,  8
11      3  x: ADD_INT       ____,  PV2.z,  KC
12      4  y: LSHL          ____,  PV3.x,  4
13      5  z: ADD_INT       R2.z,  KC1[0].x,
14      6  x: LSHR          R1.x,  PV5.z,  4
15 01 TEX: ADDR(288) CNT(1)
16      7  VFETCH R1, R1.x, fc175  FORMAT(3
17         FETCH_TYPE(NO_INDEX_OFFSET)
18 02 ALU: ADDR(49) CNT(3)
19      8  x: LDS_WRITE_REL (offset: 1) ___
20      9  x: LDS_WRITE_REL (offset: 1) ___
21     10  x: GROUP_BARRIER  ____
22 03 ALU: ADDR(52) CNT(27)
23     11  x: LDS_READ2_RET QAB,  R10.y,  R
24         y: ADD_INT       ____,  R2.z,  409
25         z: ADD_INT       ____,  R2.z,  819
26         w: ADD_INT       ____,  R2.z,  163
27         t: ADD_INT       T0.y,  R2.z,  122
28     12  x: LDS_READ2_RET QAB,  R10.w,  R
29         y: LSHR          R1.y,  PV11.y,  4
```

Compiler Statistics (Using CAL 11.7)

Name	GPR	Scratch Reg	Min	Max	Avg	ALU	Fetch	Write	Est Cycles	ALU:Fetch	BottleNeck	Thread\Clock	Throughput
Radeon HD 5870	11	0	3.20	11.10	7.05	111	8	1	7.05	2.20	ALU Ops	4.54	3858 M Threads\Sec
Radeon HD 5770	11	0	3.20	11.10	7.05	111	8	1	7.05	2.20	ALU Ops	2.27	1929 M Threads\Sec
Radeon HD 5670	11	0	6.00	22.20	14.10	111	8	1	14.10	4.41	ALU Ops	0.57	440 M Threads\Sec
Radeon HD 5450	11	0	15.00	55.50	35.25	111	8	1	35.25	8.81	ALU Ops	0.11	74 M Threads\Sec
Radeon HD 6970	12	0	2.75	9.58	6.17	115	8	1	6.17	2.31	ALU Ops	5.19	4566 M Threads\Sec
Radeon HD 6870	11	0	4.57	15.86	10.07	111	8	1	10.07	2.20	ALU Ops	3.18	2860 M Threads\Sec
Radeon HD 6670	11	0	5.00	18.50	11.75	111	8	1	11.75	4.41	ALU Ops	1.36	1089 M Threads\Sec
Radeon HD 6450	11	0	15.00	55.50	35.25	111	8	1	35.25	8.81	ALU Ops	0.45	340 M Threads\Sec

AMD's dynamic profiler

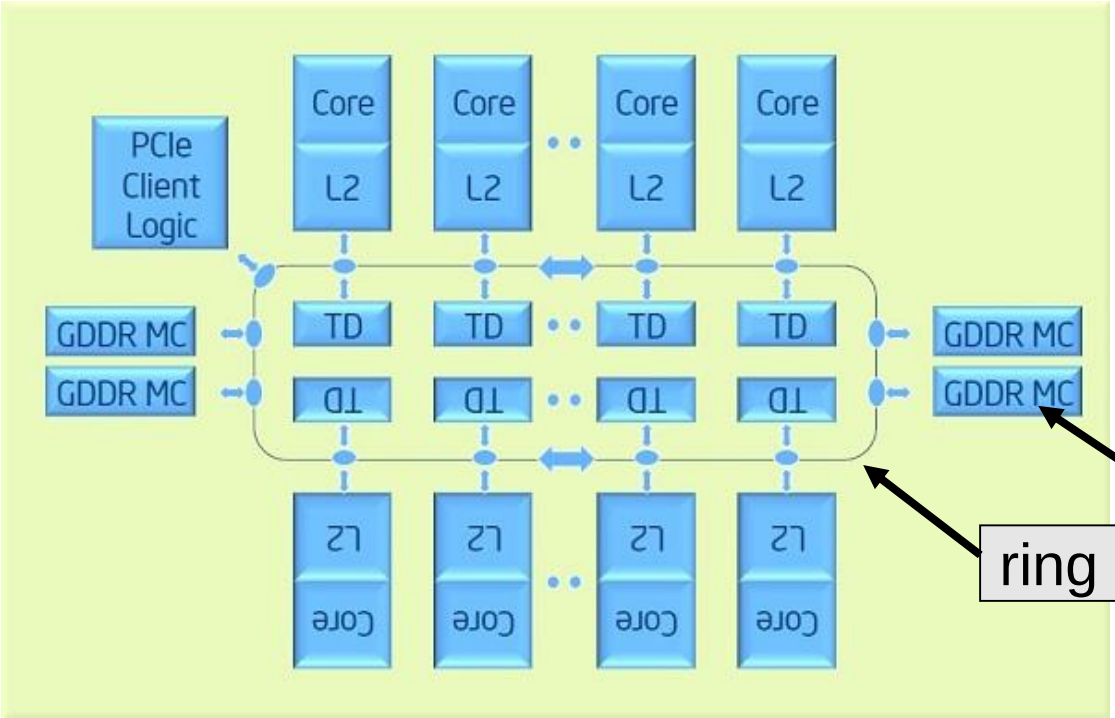
View Options

☑ Show Kernel Dispatch ☑ Show Data Transfer ☑ Show Zero Column

Method	ExecutionOrder	ThreadID	CallIndex	GlobalWorkSize	GroupWorkSize	Time	LocalMemSize	DataTransferSize	GPRs	ScratchRegs	FCStacks	Wavefronts	ALUInsts	FetchI
WriteBufferAsynch	1	3492	80			31,87611		65536,00						
main__kl_Cypress1	2	3492	89	{ 524288 1 1}	{ 256 1 1}	0,23722	4096		11	0	1	8192,00	51,25	8,0
ReadBuffer	3	3492	95			0,16278		0,06						
main__kl_Cypress1	4	3492	96	{ 256 1 1}	{ 256 1 1}	0,01067	4096		11	0	1	4,00	51,25	8,0
ReadBuffer	5	3492	102			0,22911		0,06						

Intel Xeon Phi

Intel's Xeon Phi coprocessor



Intel's response to GPUs…

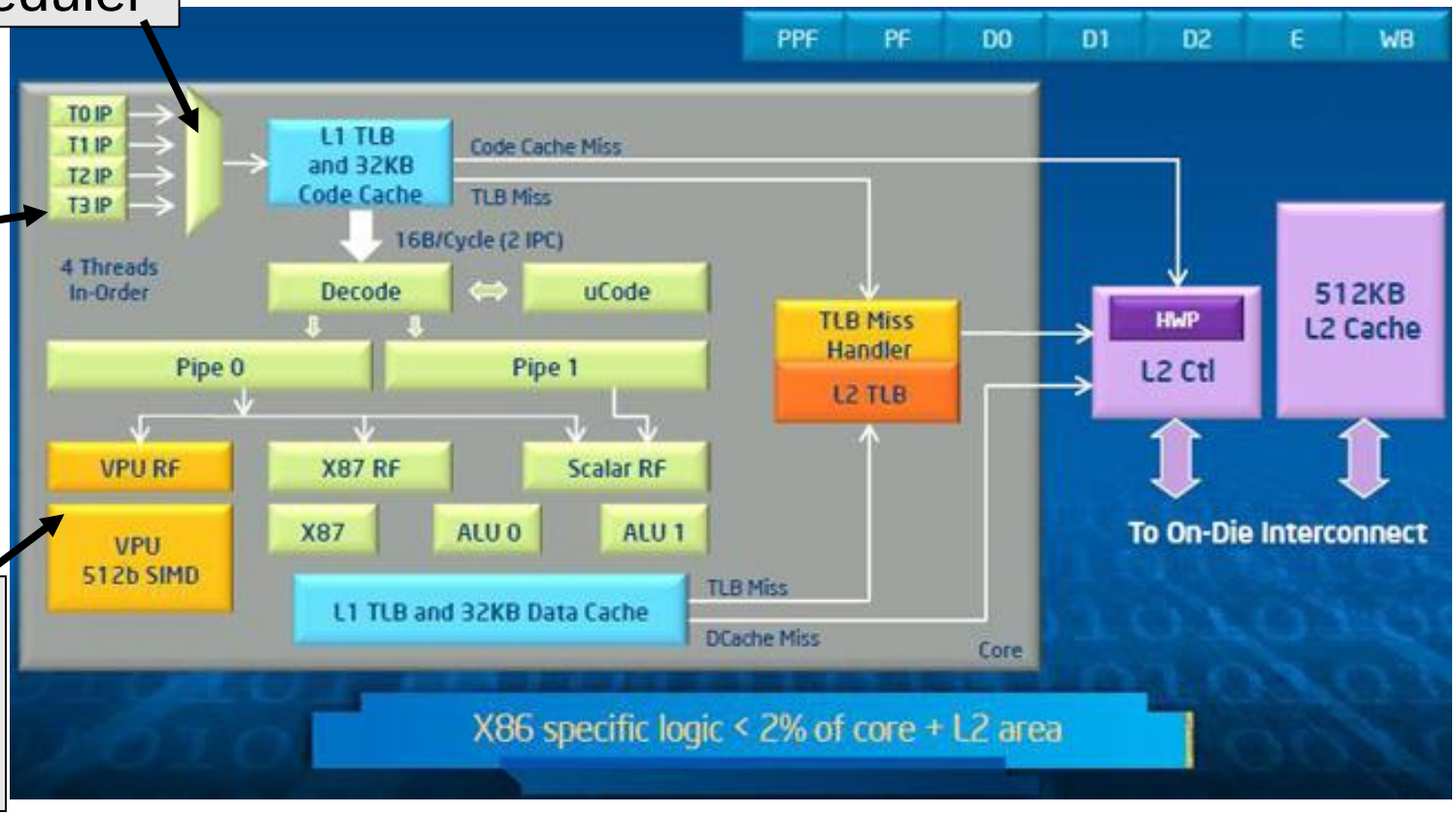
60 cores

Intel's Xeon Phi's core

Thread scheduler

4 hardware threads

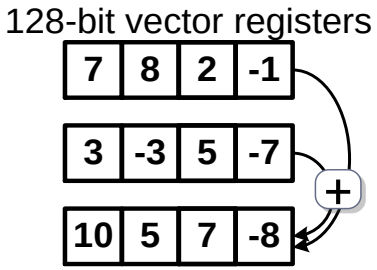
512-bit Vector unit (SIMD)



Usage of the coprocessor

- As MPI-node
- Off-load from main processor
- As standalone processor
- Common c-programming
  - Pthreads
  - Openmp
  - Intel threading building blocks

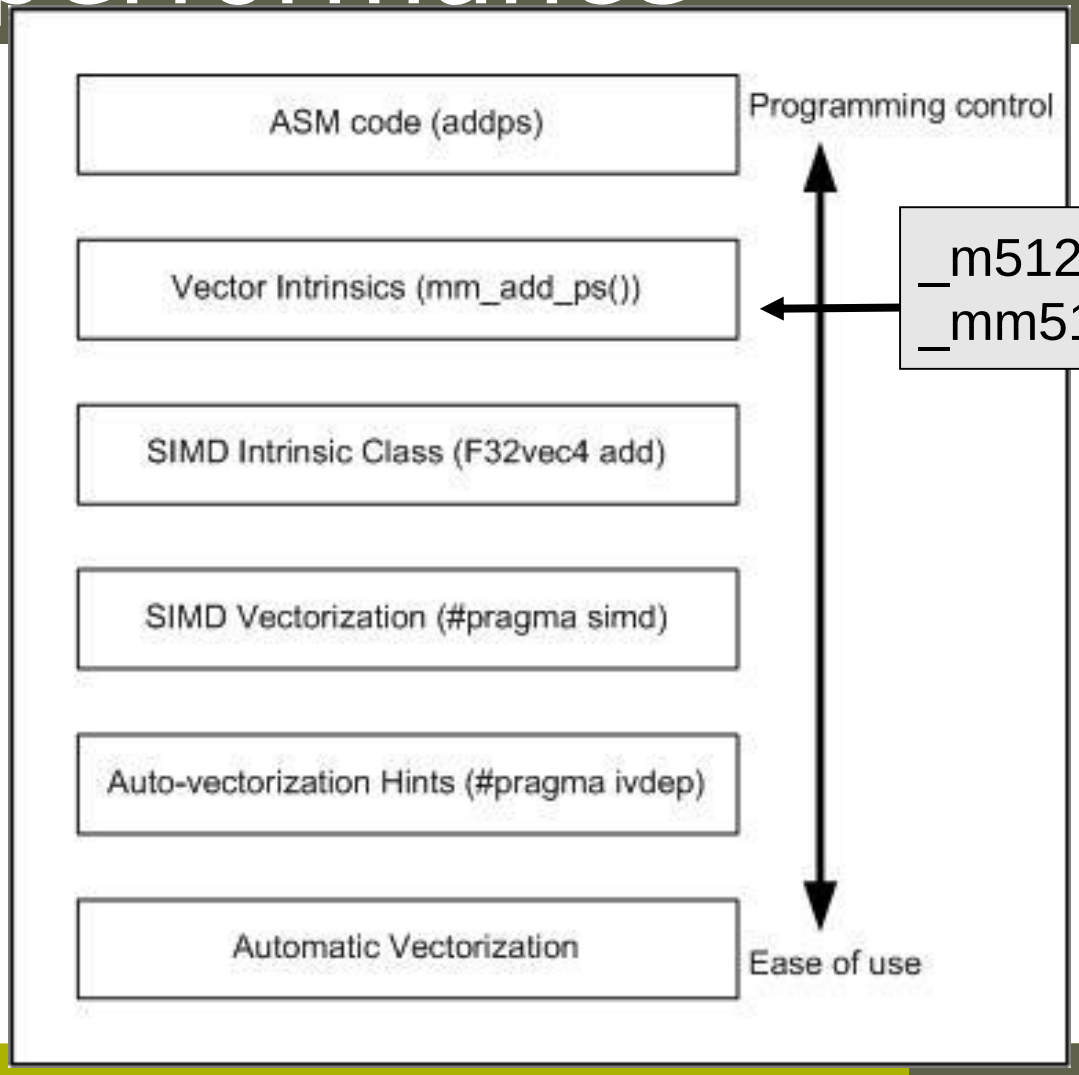
Vector processors (SIMD)



Instructions can be performed at once on all elements of vector registers

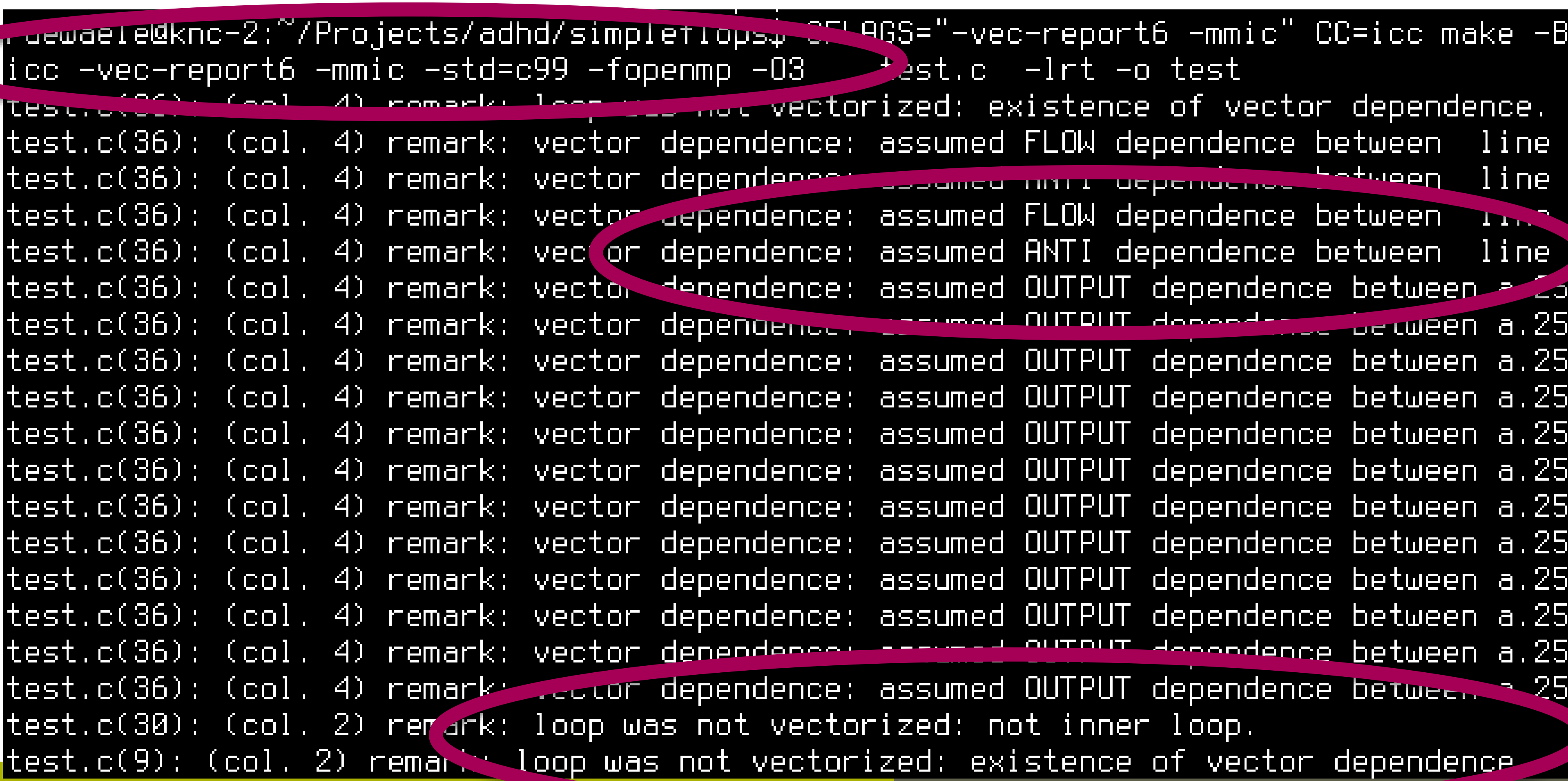
- Has long be viewed as the solution for high-performance computing
  - Why always repeating the same instructions (on different data)? => just apply the instruction immediately on all data
- However: difficult to program
- Is SIMT (OpenCL) a better alternative??

Vectorization needed for pea performance



_m512 register
_mm512_add operations

Auto-vectorization



```
dewaele@knc-2:~/Projects/adhd/simpleflops$ CFLAGS="-vec-report6 -mmic" CC=icc make -B
icc -vec-report6 -mmic -std=c99 -fopenmp -O3    test.c  -lrt -o test
test.c(36): (col. 4) remark: loop was not vectorized: existence of vector dependence.
test.c(36): (col. 4) remark: vector dependence: assumed FLOW dependence between  line
test.c(36): (col. 4) remark: vector dependence: assumed ANTI dependence between  line
test.c(36): (col. 4) remark: vector dependence: assumed FLOW dependence between  line
test.c(36): (col. 4) remark: vector dependence: assumed ANTI dependence between  line
test.c(36): (col. 4) remark: vector dependence: assumed OUTPUT dependence between a.25
test.c(36): (col. 4) remark: vector dependence: assumed OUTPUT dependence between a.25
test.c(36): (col. 4) remark: vector dependence: assumed OUTPUT dependence between a.25
test.c(36): (col. 4) remark: vector dependence: assumed OUTPUT dependence between a.25
test.c(36): (col. 4) remark: vector dependence: assumed OUTPUT dependence between a.25
test.c(36): (col. 4) remark: vector dependence: assumed OUTPUT dependence between a.25
test.c(36): (col. 4) remark: vector dependence: assumed OUTPUT dependence between a.25
test.c(36): (col. 4) remark: vector dependence: assumed OUTPUT dependence between a.25
test.c(36): (col. 4) remark: vector dependence: assumed OUTPUT dependence between a.25
test.c(36): (col. 4) remark: vector dependence: assumed OUTPUT dependence between a.25
test.c(36): (col. 4) remark: vector dependence: assumed OUTPUT dependence between a.25
test.c(36): (col. 4) remark: vector dependence: assumed OUTPUT dependence between a.25
test.c(30): (col. 2) remark: loop was not vectorized: not inner loop.
test.c(9): (col. 2) remark: loop was not vectorized: existence of vector dependence
```

SIMD pragma to indicate p

```
void dflops(double * restrict a) {
  const double c = 1.;
  const double x = 0.9;
#pragma simd
  for (long long i = 0; i < niterations; i += 16) {
    a[0] = a[0] * x + c;
    a[1] = a[1] * x + c;
    a[2] = a[2] * x + c;
    a[3] = a[3] * x + c;
    a[4] = a[4] * x + c;
    a[5] = a[5] * x + c;
    a[6] = a[6] * x + c;
    a[7] = a[7] * x + c;

    a[8] = a[8] * x + c;
    a[9] = a[9] * x + c;
    a[10] = a[10] * x + c;
    a[11] = a[11] * x + c;
    a[12] = a[12] * x + c;
    a[13] = a[13] * x + c;
    a[14] = a[14] * x + c;
    a[15] = a[15] * x + c;
  }
}
```

Successful vectorization

```
rdewaele@knc-2:~/Projects/adhd/simpleflops/simpleflops$ CFLAGS="-vec-re
icc -vec-report6 -mmic -std=c99 -O3 -fopenmp -funroll-loops -vec-report
test.c(84): (col. 5) remark: vectorization support: reference sa has un
test.c(84): (col. 5) remark: vectorization support: unaligned access us
test.c(83): (col. 4) remark: LOOP WAS VECTORIZED.
test.c(76): (col. 3) remark: loop was not vectorized: not inner loop.
test.c(74): (col. 2) remark: loop was not vectorized: not inner loop.
test.c(79): (col. 4) remark: SIMD LOOP WAS VECTORIZED.
test.c(13): (col. 2) remark: SIMD LOOP WAS VECTORIZED.
test.c(38): (col. 2) remark: SIMD LOOP WAS VECTORIZED.
```

SIMD (vectorisation)

Versus

SIMT (Single Instruction Multiple Thread – OpenCL/CUDA)

Overview

1. GPUs for general purpose
2. GPU 'threads' executing kernels
3. Starting kernels from host (CPU)
4. Execution model & GPU architecture
5. Warps/wavefronts
6. Optimizing GPU programs
7. **Analysis & Conclusions**

Link 1: white paper

GPU Strategy

- Don't write explicitly threaded code
  - Compiler handles it => no chance of deadlocks or race conditions
- Think differently: analyze the **data** instead of the algorithm.
- In contrast with modern superscalar CPUs: programmer writes sequential code (single-threaded), processor tries to execute it in parallel, through pipelining etc. (instruction parallelism). But by the data and resource dependencies more speedup cannot be reached with > 4-way superscalar CPUs. *1.5 Instructions per cycles seems a maximum*.
- Programming models have to make a delicate balance between **opacity** (making an abstraction of the underlying architecture) and **visibility** (showing the elements influencing the performance). It's a trade-off between productivity and implementation efficiency.

Results

- Performance doubling every 6 months!
- 1000s of threads possible!
- High Bandwidth
  - PCI Express bus (connection GPU-CPU) is the bottleneck
- Enormous possibilities for latency hiding
- Matrix Multiplication 13 times faster on a standard GPU (GeForce 8500GT) compared to a state-of-the art CPU (Intel Dual Core)
  - 200 times faster on a high-end GPU, 50 times if quadcore.
- Low threshold (especially Nvidia's CUDA):
  - C, good documentation, many examples, easy-to-install, automatic card detection, easy-compilation

How to get maximal performance, or call it ... limitations

- Create many threads, make them 'aggressively' parallel
- Keep threads busy in a warp
- Align memory reads
  - Global memory <> Shared/local memory
  - Using shared memory
- Limited memory per thread
- Close to hardware architecture
  - Hardware is made for exploiting data parallelism

Disadvantages

- Maintenance…
- CUDA = NVIDIA
  - Alternatives:
    - ***OpenCL***: a standard language for writing code for GPUs and multicores. Supported by ATI, NVIDIA, Apple, …
    - RapidMind's Multicore Development, supports multiple architectures, less dependent on it
    - AMD, IBM, Intel, Microsoft and others are working on standard parallel-processing extensions to C/C++
    - Intel's Xeon Phi: combining processing power of GPUs with programmability of x86 processors **Links in Scientific Study section**
- CUDA/OpenCL promises an abstract, scalable hardware model, but will it remain true?

Link 1: white paper

Heterogeneous Chip Designs

- Augment standard CPU with attached processors performing the compute-intensive portions :
  - Graphics Processing Unit (GPU)
  - Field Programmable Gate Array (FPGA)
  - Xeon Phi coprocessor
  - Cell processor, designed for video games

Go parallel: take decisions now based on expectations of the future.

- But future is unclear…
  - Parallel world is evolving.
- What do Intel, NVIDIA & Riverside tell us?
  - Workshop in Ghent, May 16 2011: "Challenges Towards Exascale Computing"
- <u>They agree on:</u>
  - Heterogeneous hardware is the future
  - Data movement will determine the cost (power & cycles)
  - Power consumption & Programmability are the challenges
  - Commodity products & programming languages
  - Hope for a programming model expressing *parallelism* and *locality*

The future… II

- They do not agree on:
  - Intel sticks to x86 architecture
    - That's what programmers know & they won't change
    - Intel platforms have to support legacy code
    - New architecture: Knights Ferry & Knights Corner (cf Larrabee)
  - GPU: stream processor with high throughput, latency is hidden by massively multithreading
  - CPU: one-thread processor with low latencies
  - Riverside sees reconfigurable hardware as the sole solution: no data movement necessary.
  - NVIDIA envisages that the CPU will still be on board… in a corner of the chip ;-)