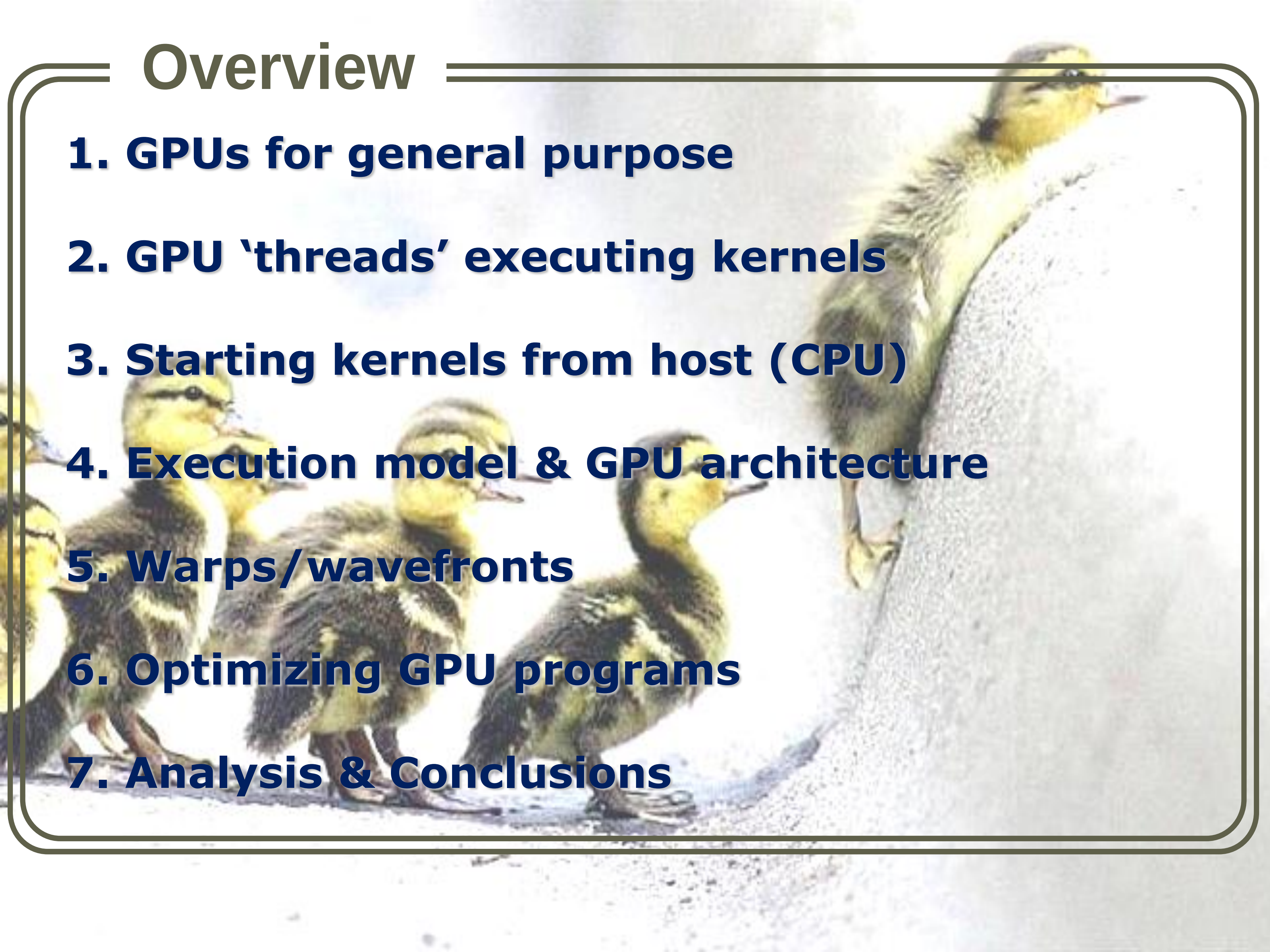
Parallel Systems Course: Chapter IV
Advanced Computer Architecture

GPU Programming

Jan Lemeire
Dept. ETRO
September 28th 2012

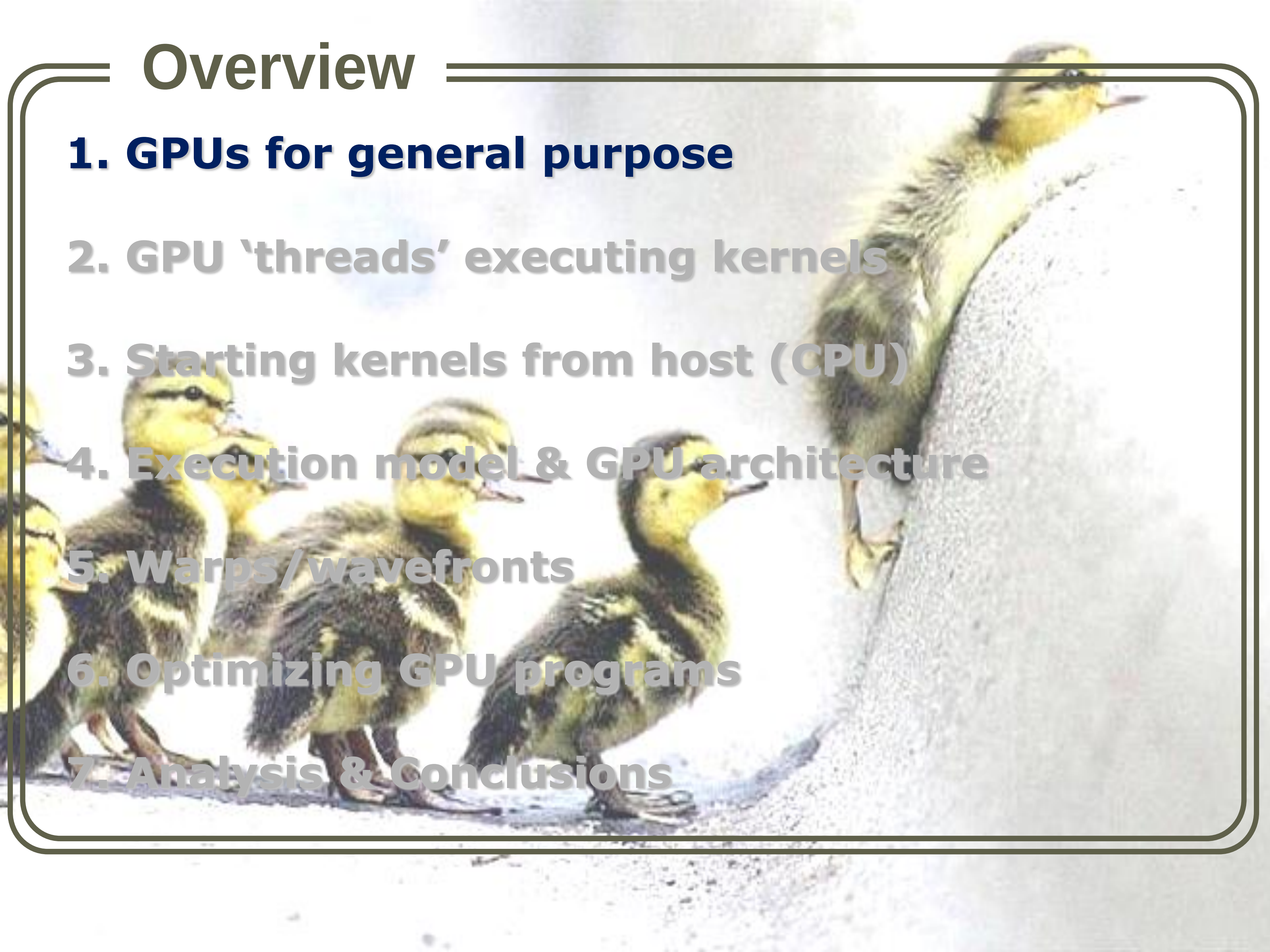
Vrije Universiteit Brussel



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

Overview



2010

350 Million triangles/second
3 Billion transistors GPU

1995

5,000 triangles/second
800,000 transistors GPU

Supercomputing for free

- FASTRA at university of Antwerp

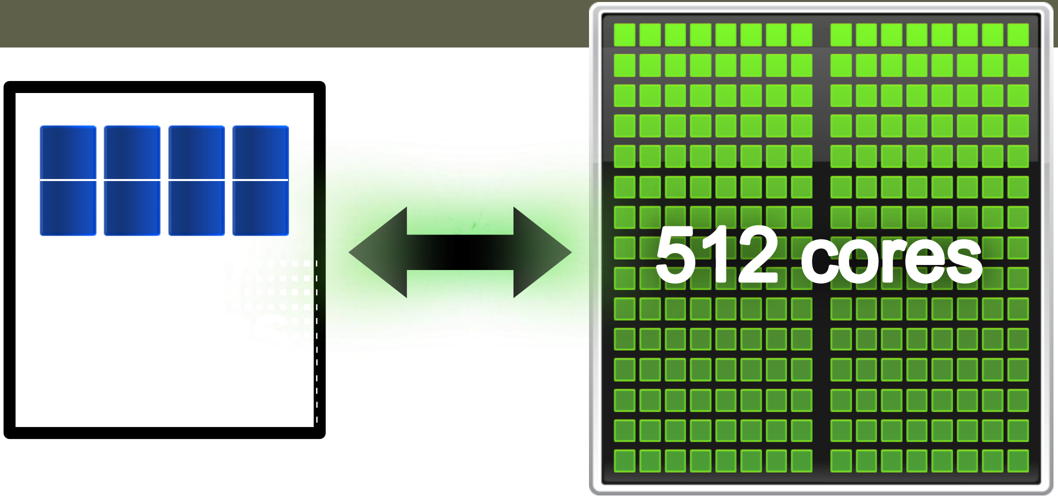
http://fastra.ua.ac.be

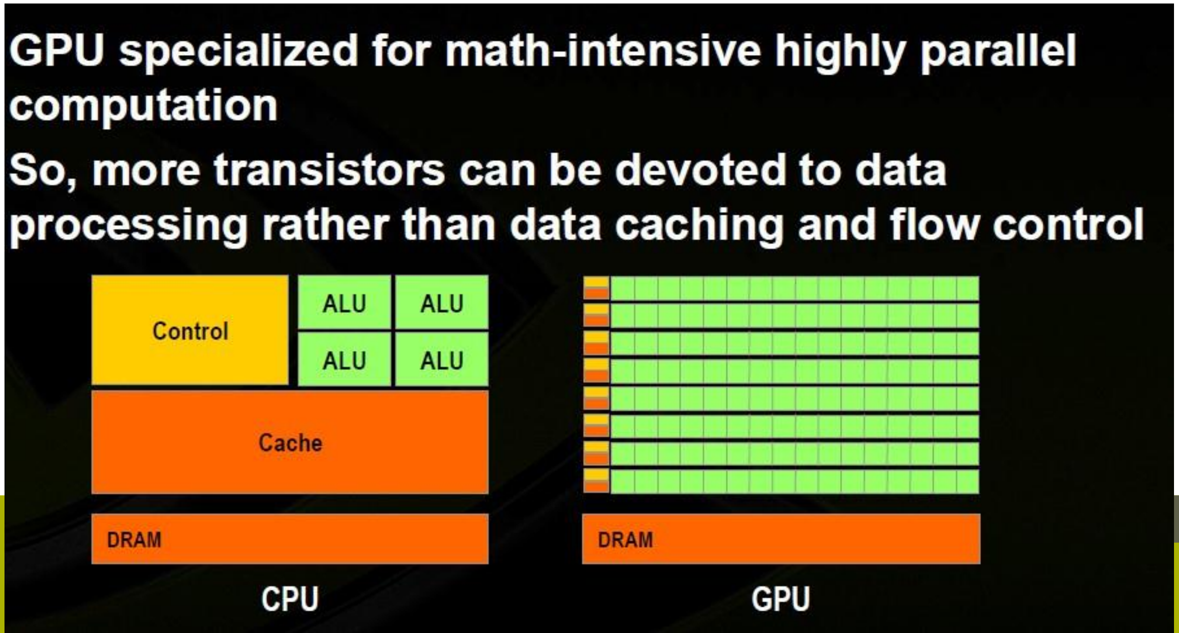
Collection of 8 graphical cards in PC

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

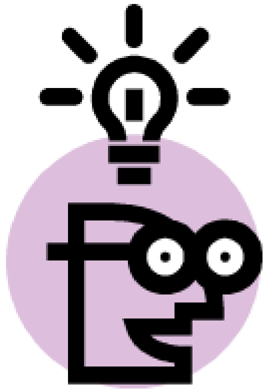
So, more transistors can be devoted to data processing rather than data caching and flow control

GPU architecture strategy

- Light-weight threads, supported by the hardware
  - Thread processors, upto 96 threads per processor
  - Context switch 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…

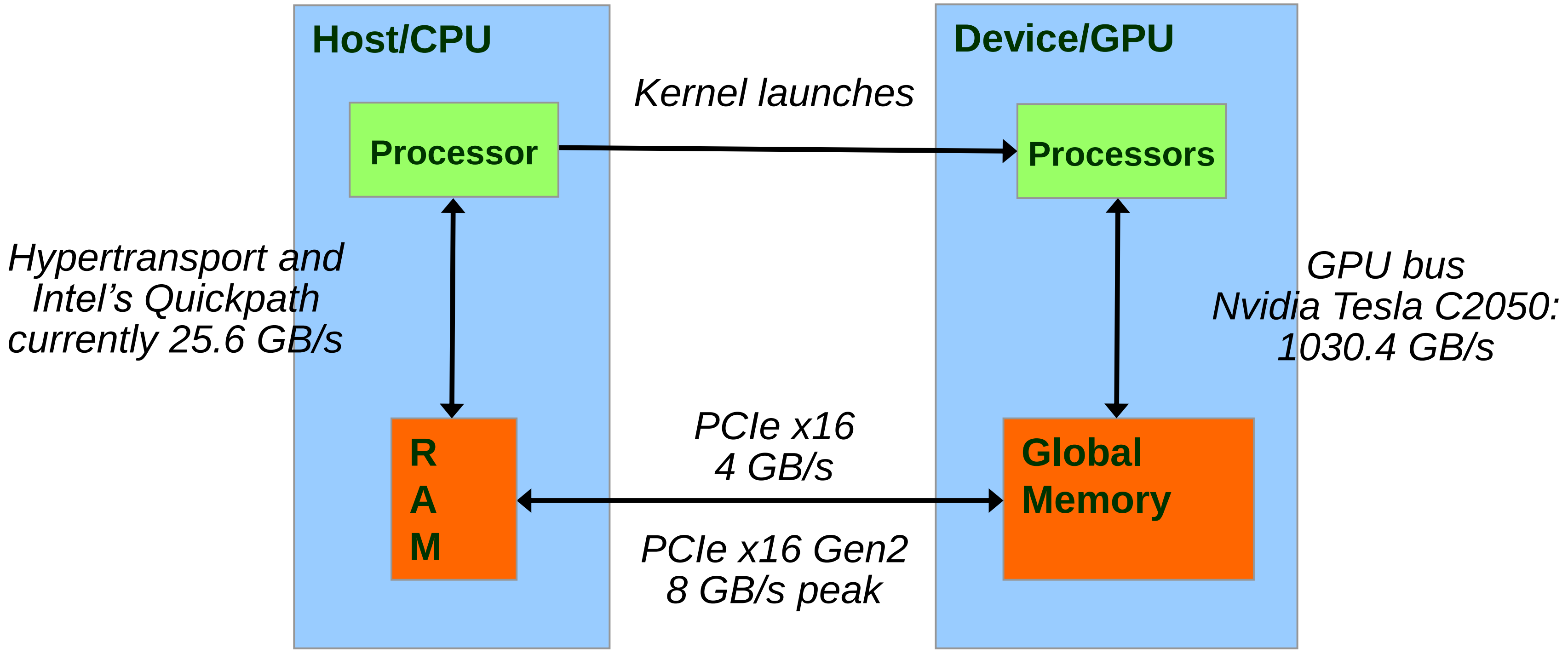
GP-GPUs: Graphics Processing Units for General-Purpose programming



Usage

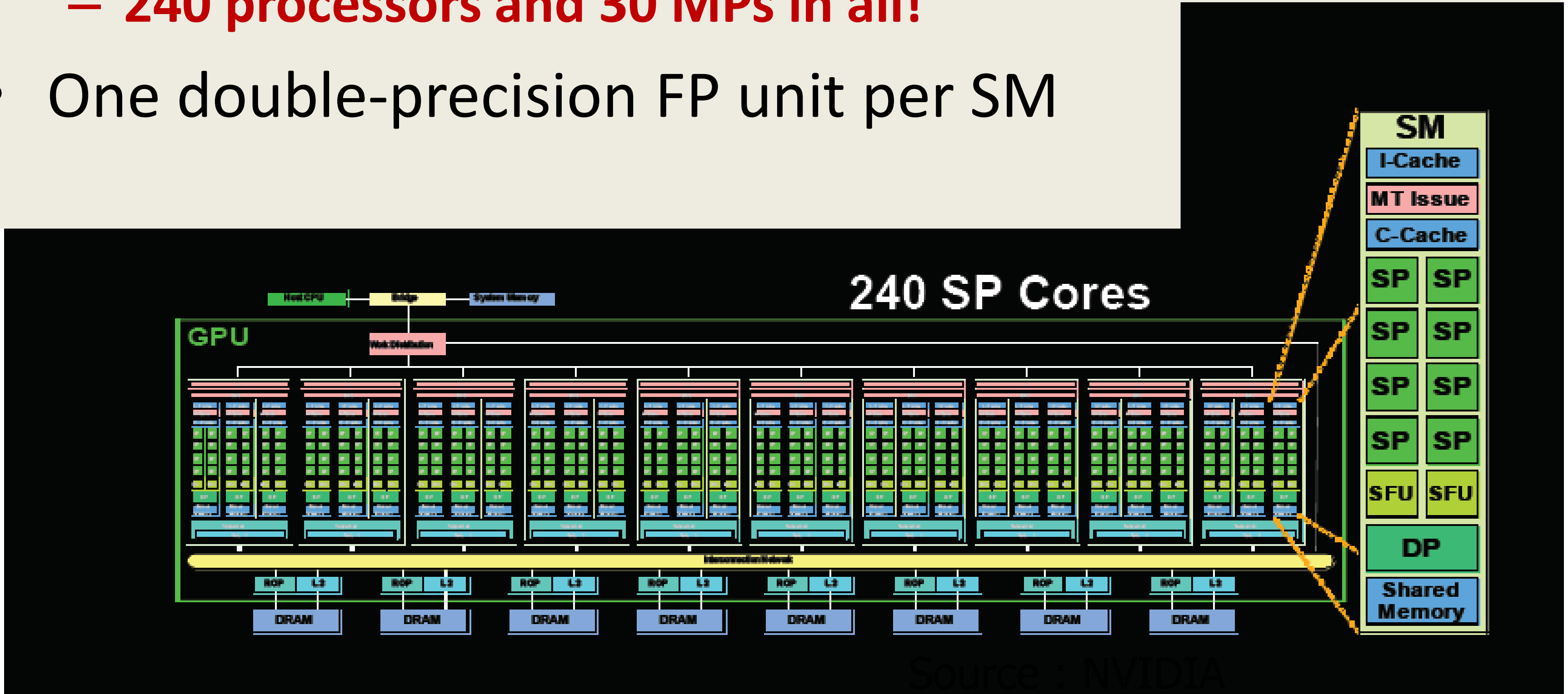
- Copy data from CPU to GPU
- Start kernel within CPU-program (C, java, Matlab, python, …)
    - Several kernels can be launched (pipelined)
    - Handled on the GPU one by one or in parallel
- Figure

Host (CPU) – Device (GPU)


Host/CPU
Device/GPU
Kernel launches
Processor
Processors
Hypertransport and Intel's Quickpath currently 25.6 GB/s
GPU bus Nvidia Tesla C2050: 1030.4 GB/s
R A M
Global Memory
PCIe x16 4 GB/s
PCIe x16 Gen2 8 GB/s peak

GPU Architecture

- In the GTX 280, there are 10 Thread Processing Clusters
  - Each has 3 Streaming Multiprocessors (SMs), which we will refer to as ***multiprocessors*** (MPs)
  - Each MP has 8 Streaming Processors (SPs) or Thread Processors (TPs). We will refer to these as ***processors.***
  - **240 processors and 30 MPs in all!**
- One double-precision FP unit per SM



Source : NVIDIA

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
[redacted]	[redacted]	[redacted]

[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: pixel transformation

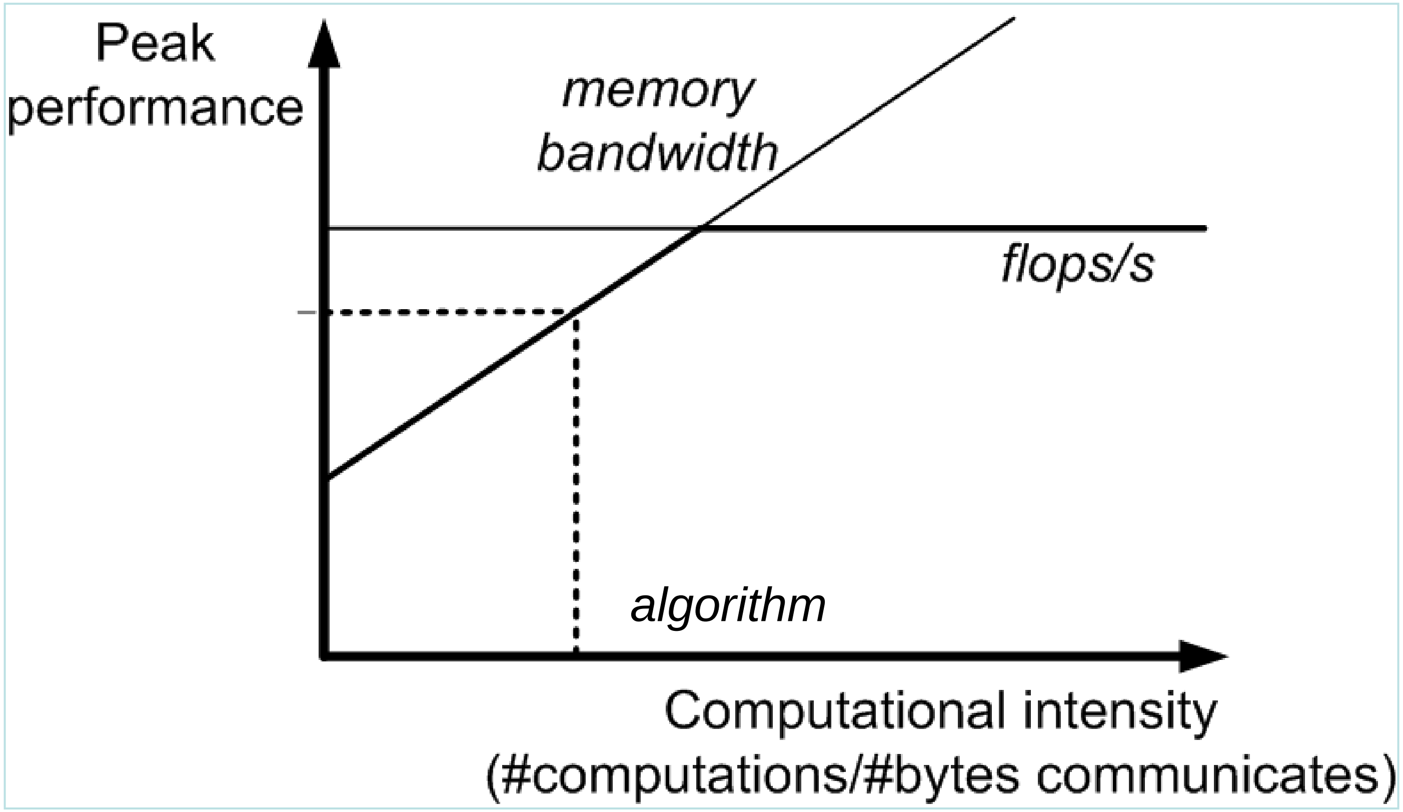
```
usgn_8  transform(usgn_8 in, sgn_16 gain, sgn_16 gain_divide,
sgn_8 offset)
{
    sgn_32 x;

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

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

Pixel transformation

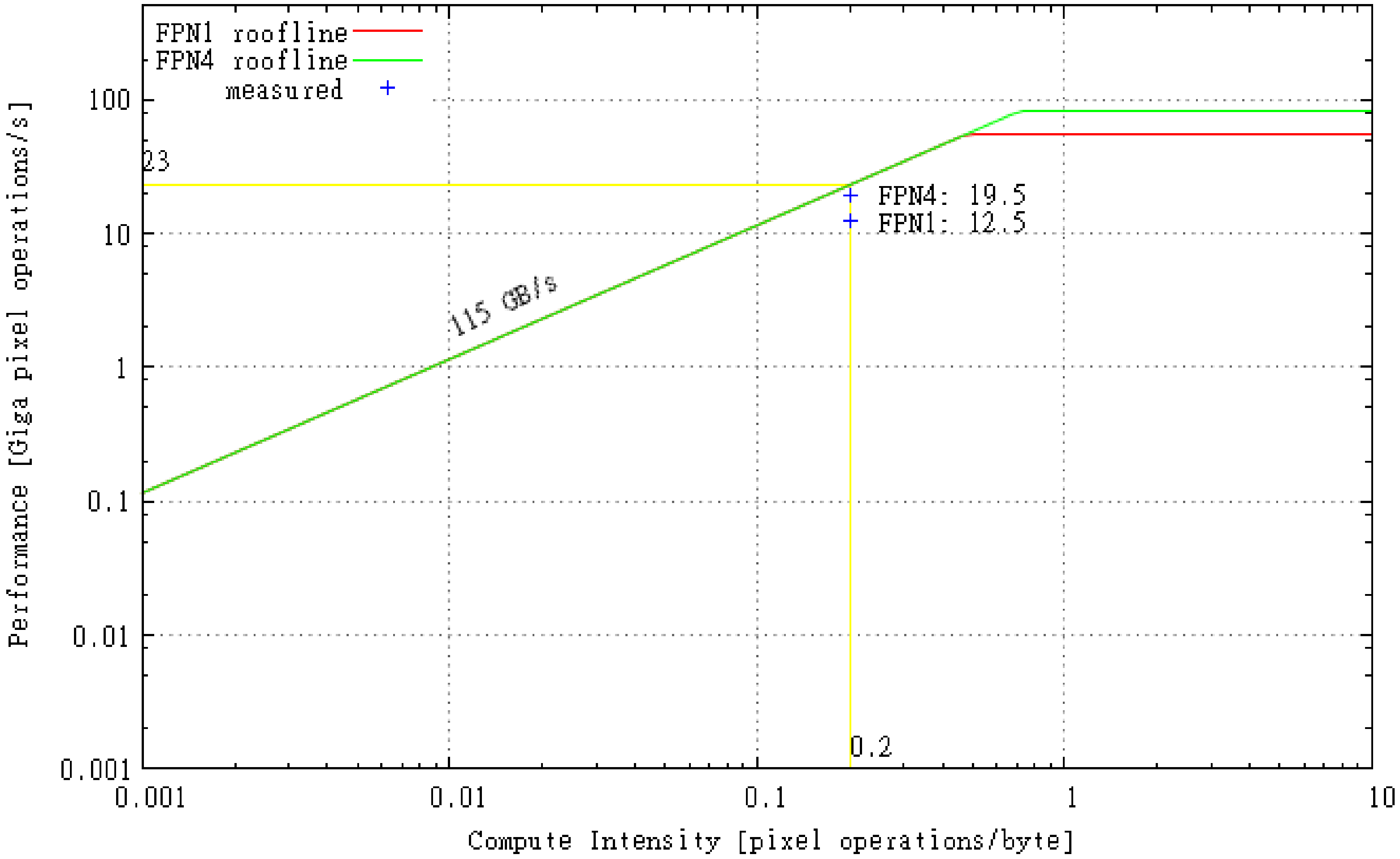
- Performance on Tesla C2050
- 1 pixel is represented by 1 byte [0-255]
- Integer operations: performance is half of floating point operations
- **Two different implementations**:
  - FPN1: 1 pixel per thread
  - FPN4: 4 pixels per thread (treat 4 bytes as 1 'word')

$\mathbf{P}_{\mathbf{mem}}$ *(bytes/s)*	115 GB/s	$\mathbf{P}_{\mathbf{ops}}$ *(ops/s)*	500 Gops/s
CI *(bytes/pix)*	1/5	*Ops/pix*	5+4 (FPN1) 5+1 (FPN4)
$\mathbf{P}_{\mathbf{mem}}$**xCI** *(pix/s)*	**23 Gpix/s**	$\mathbf{P}_{\mathbf{ops}}$**/(Ops/pix)**	56 Gpix/s (FPN1) 83 Gpix/s (FPN4)

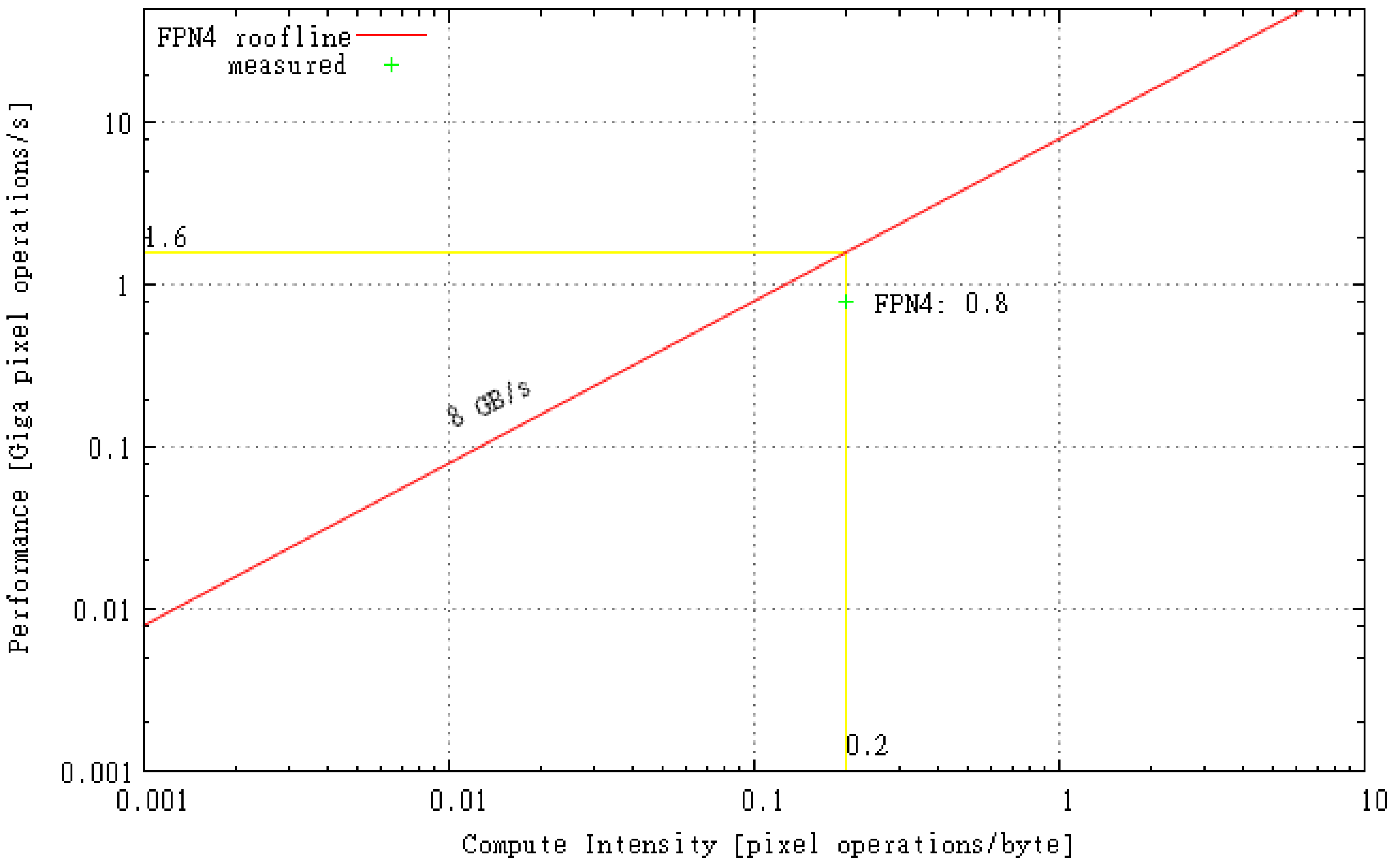
Roofline model



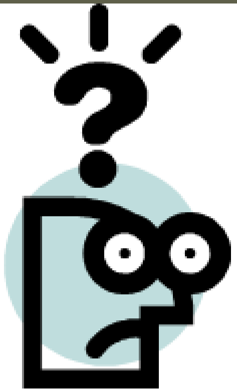
Roofline model applied
kernel only



Roofline model applied
PCI Express included



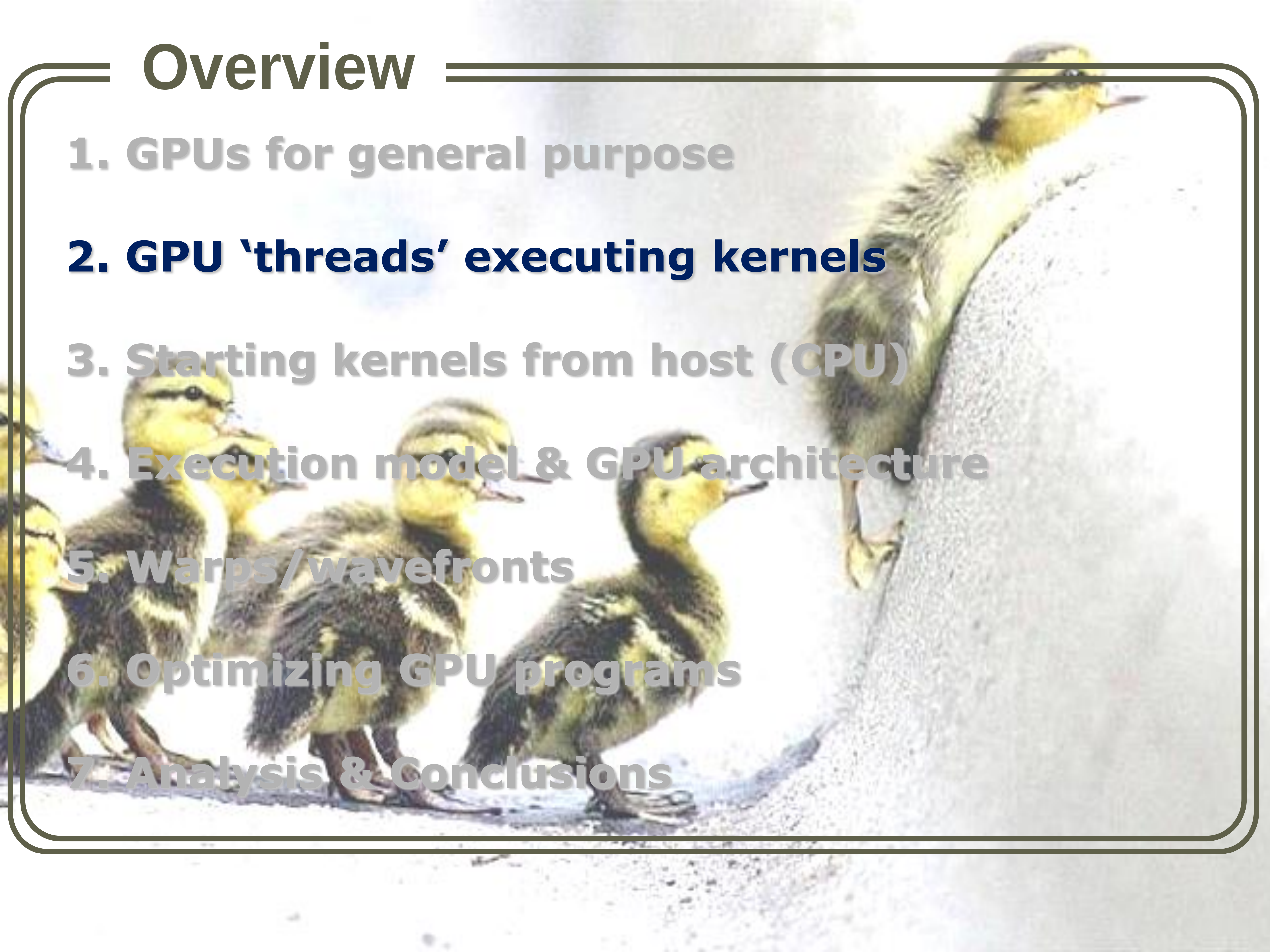
But… nothing is for free



- Harder to program!
  - Hardware architecture should be taken into account
  - Optimization is important
  - Additional complexity in code
  - Harder to debug, maintain, …

- Algorithms should be inherently massively-parallel

Overview

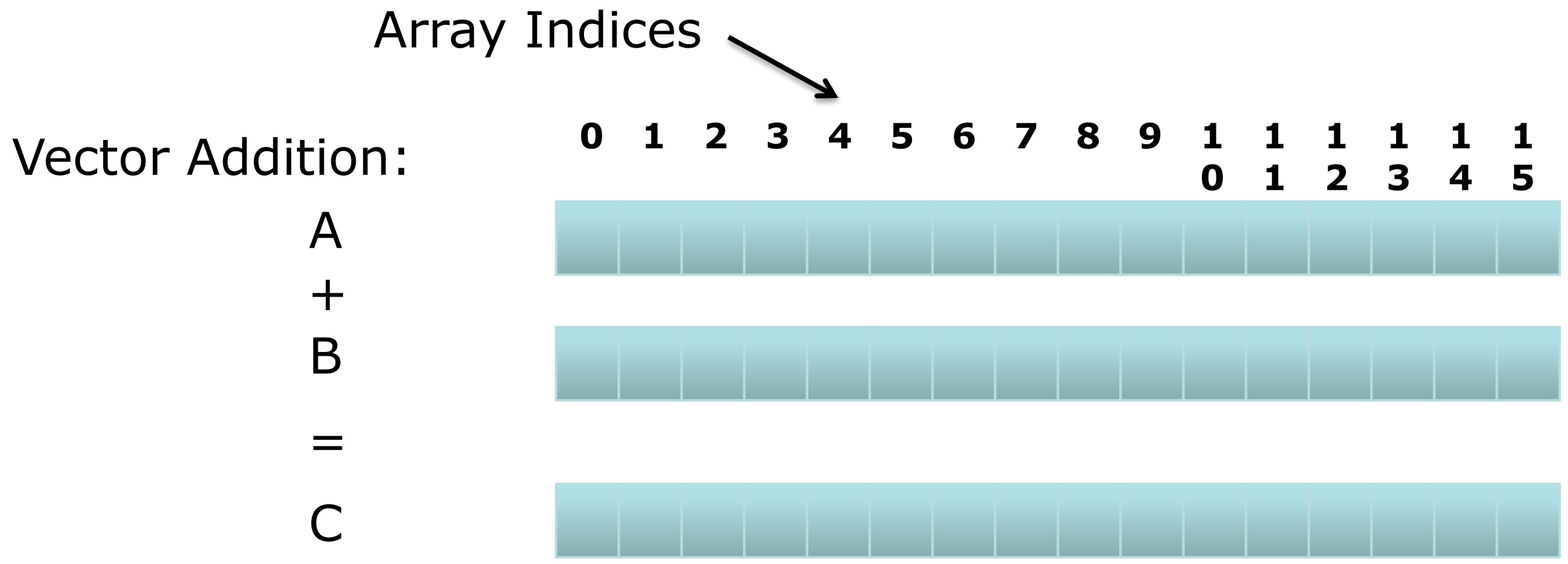


Thread Structure

- 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

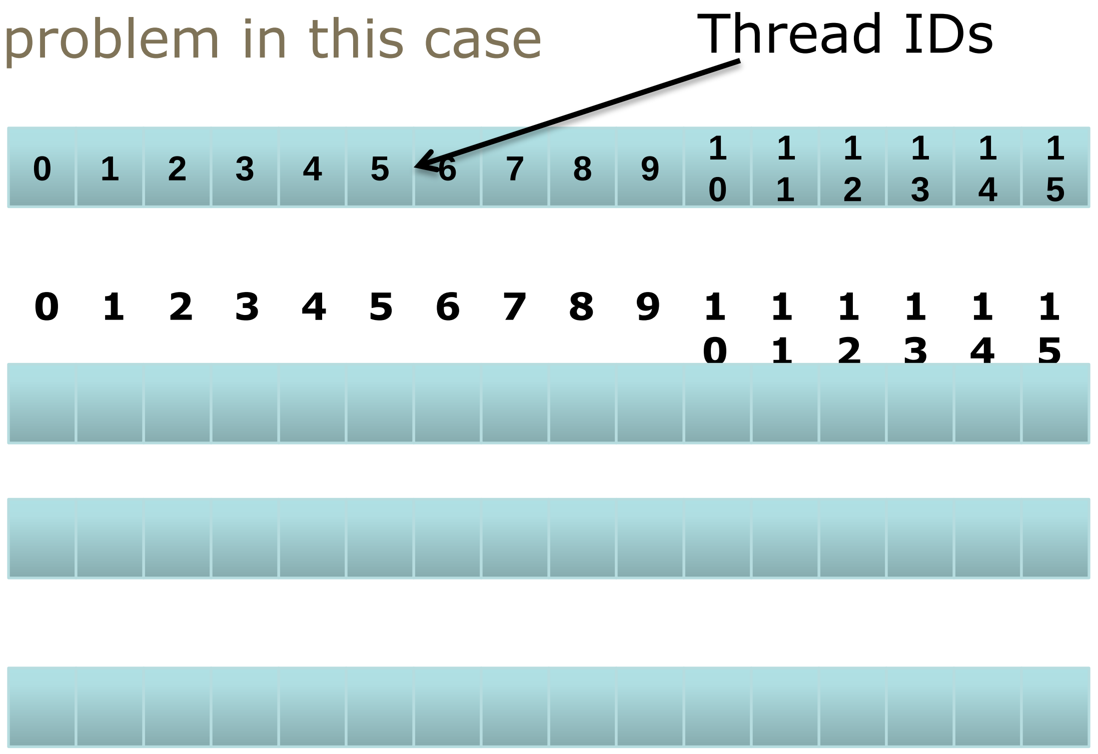
Thread Structure

- 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



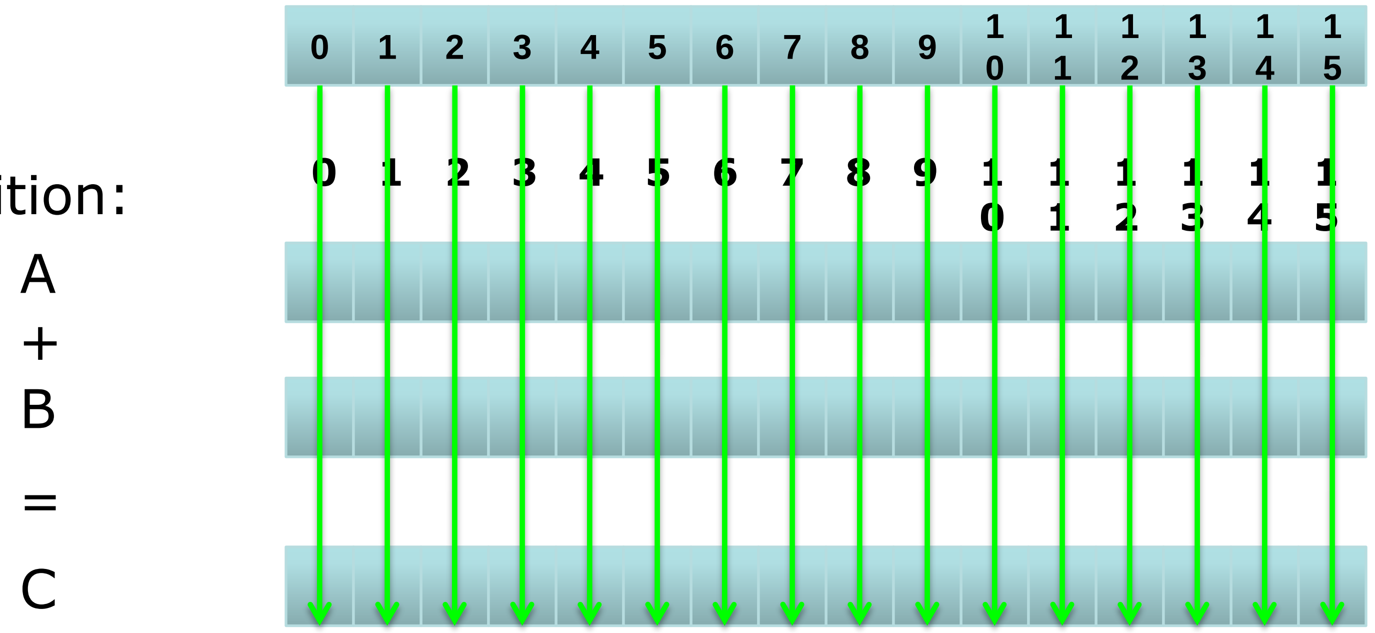
Vector Addition:

A + B = C

Thread Structure

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

Vector Addition:



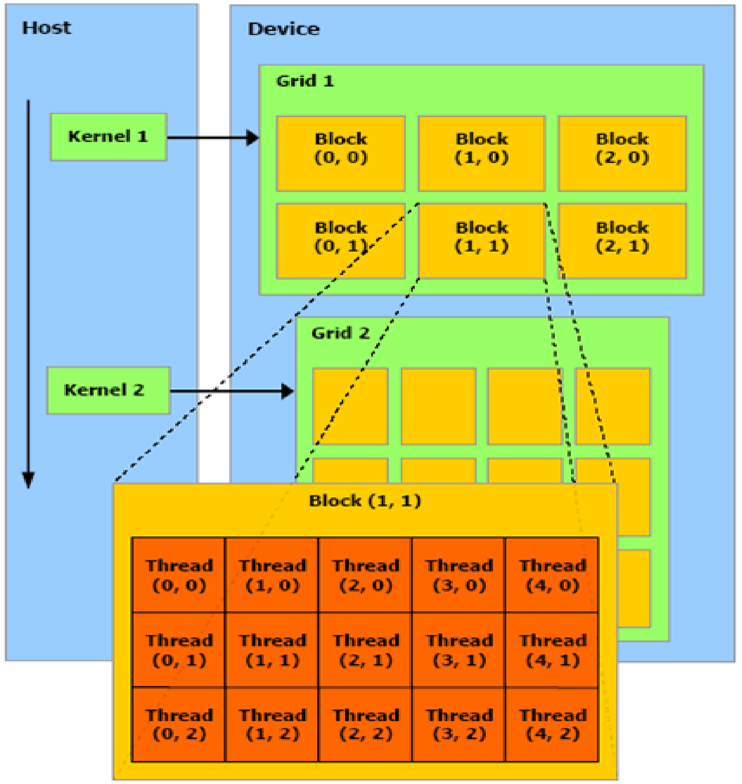
OpenCL Kernel code

```
__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!

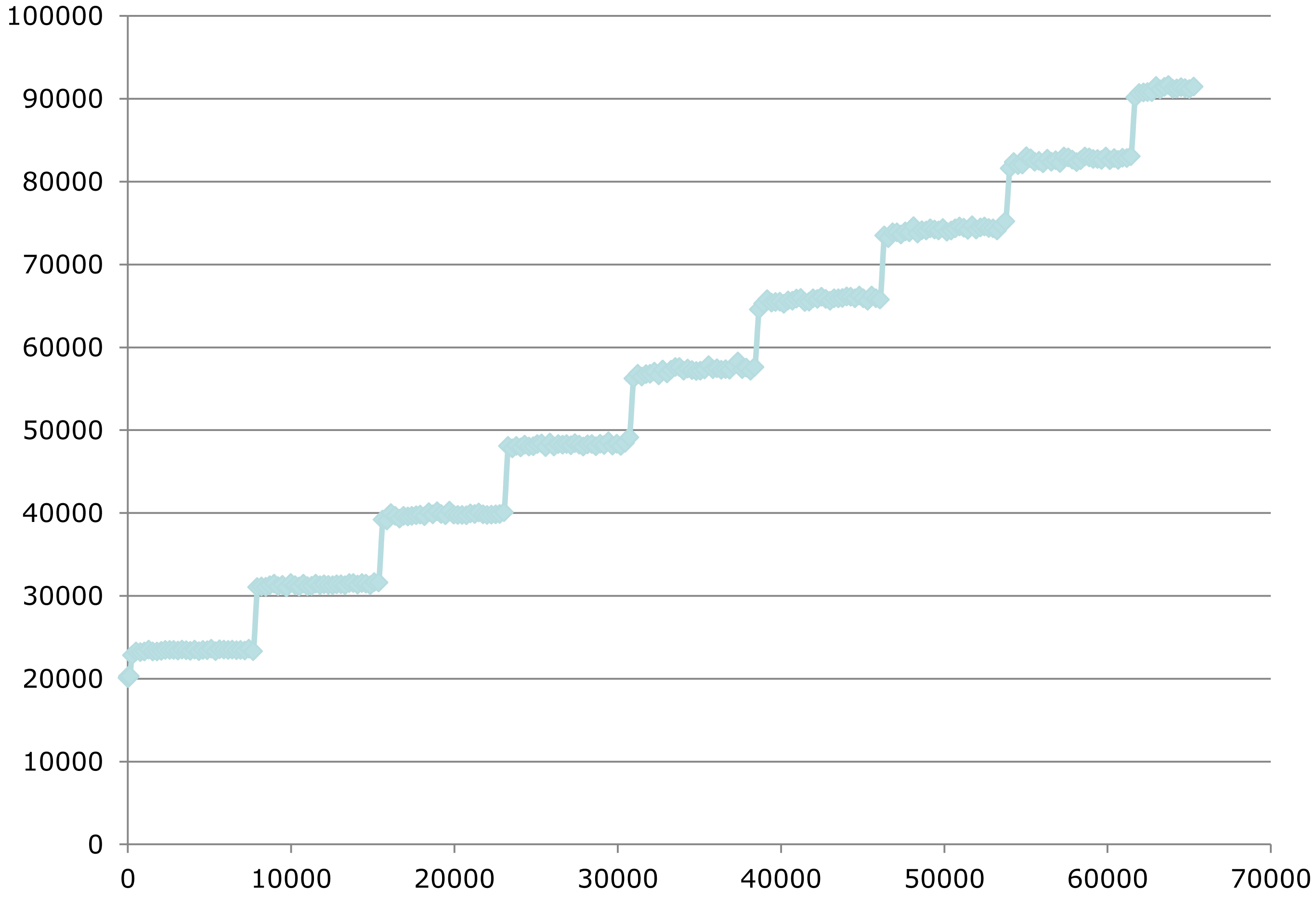


The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks

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)

Runtime (ns) i.f.o. #threads



Occupancy

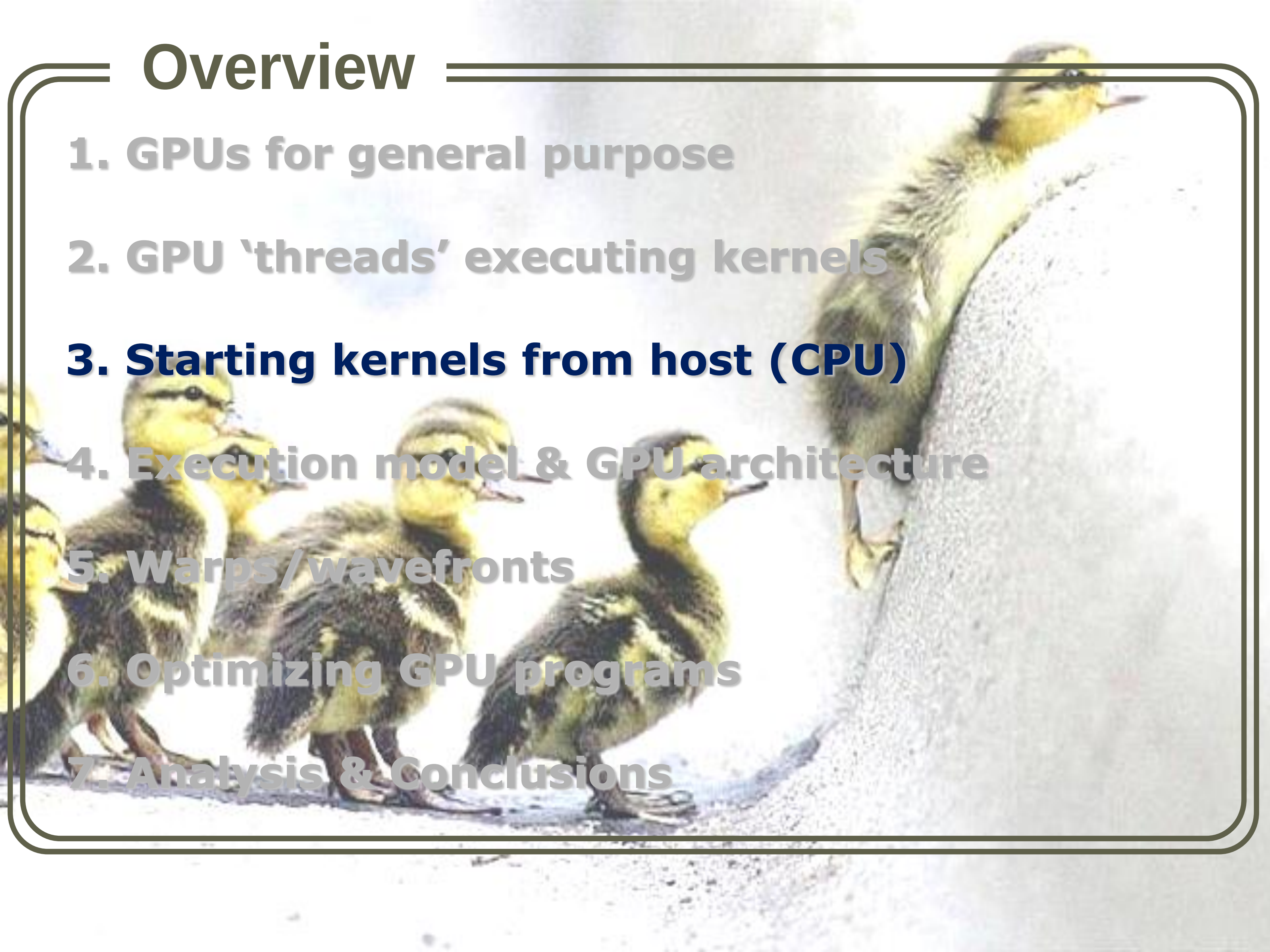
- Keep all processing units busy!
  - Enough threads
- All Multiprocessors (MPs)
- All Scalar Processors (SPs)
- Full pipeline of scalar processor
  - Pipeline of 24 stages (*see later*)

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 you want the speed, you 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 UMEÅ 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

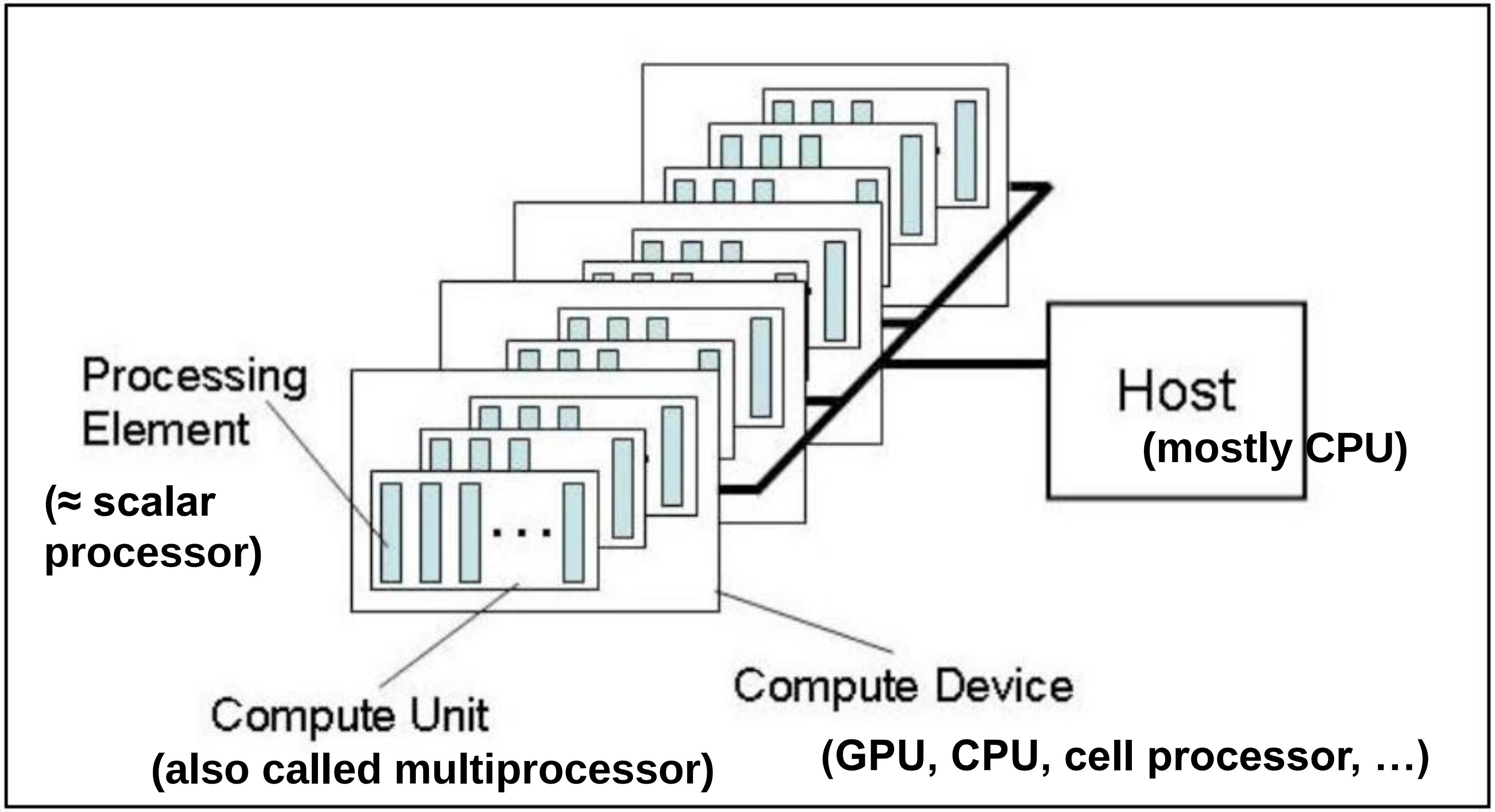
- **OpenCL functions:** start with *cl* prefix

OpenCL Kernel code

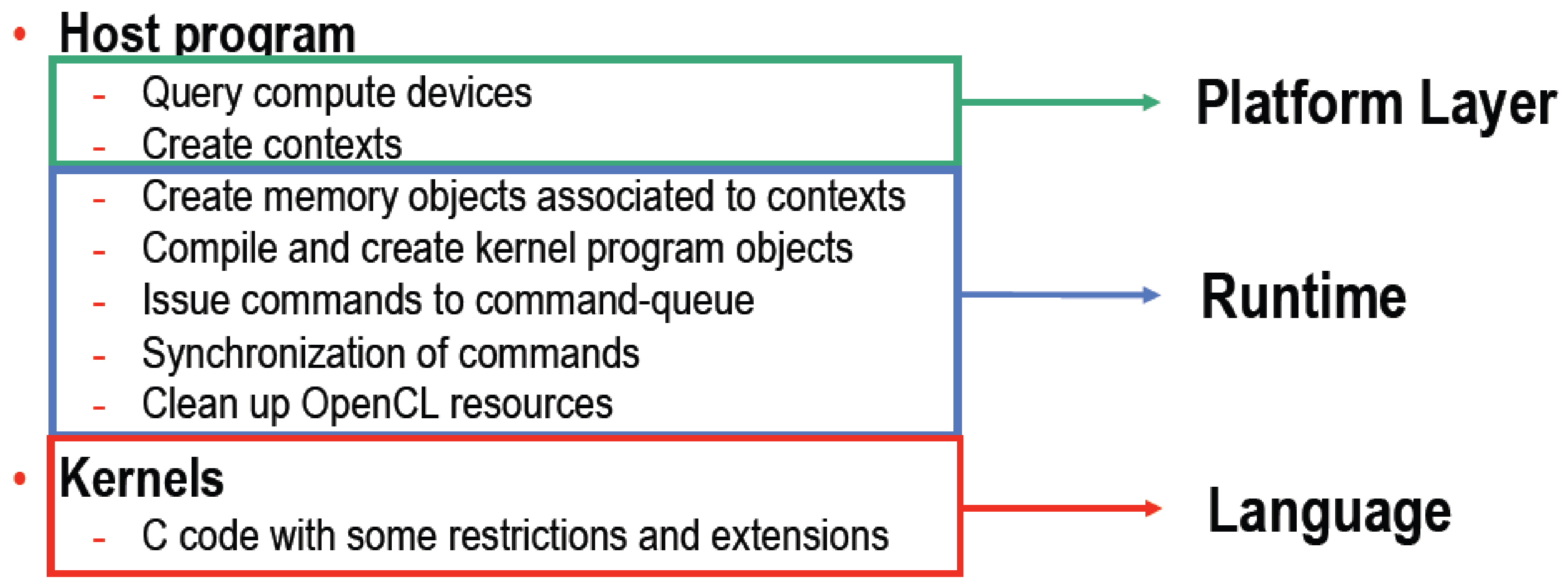
```
__kernel void vectorAdd(__global const float * a,
__global const float * b, __global float * c)
{
  // Vector element index
  int nIndex = get_global_id(0);
  // addition
  c[nIndex] = a[nIndex] + b[nIndex];
}
```

- 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

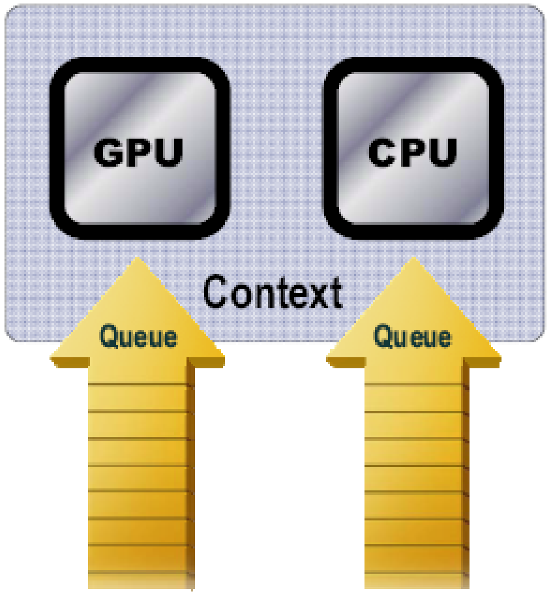
Shows the steps to develop an OpenCL program

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

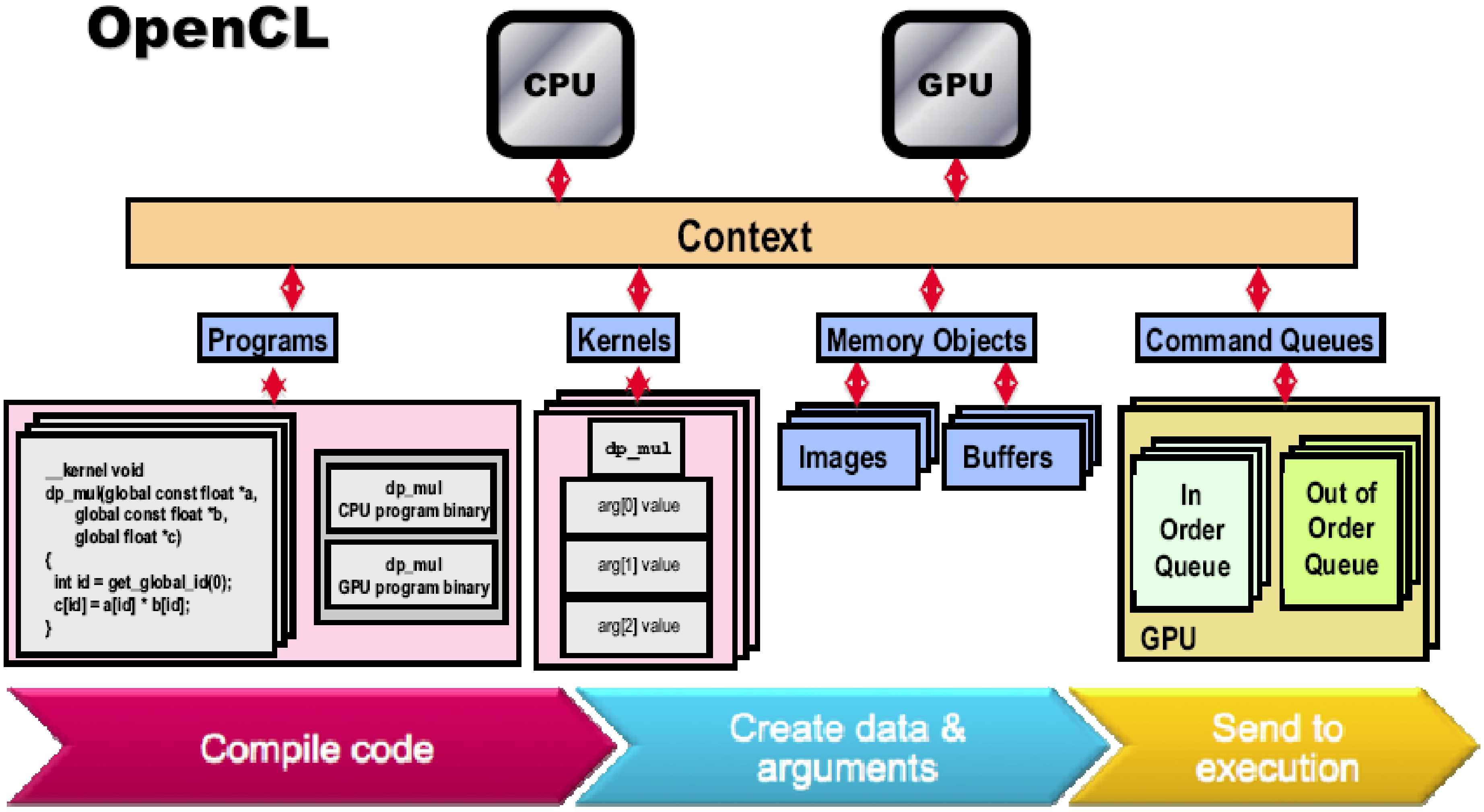
1. Get the device(s)
2. Create a context
3. Create command queue(s)



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



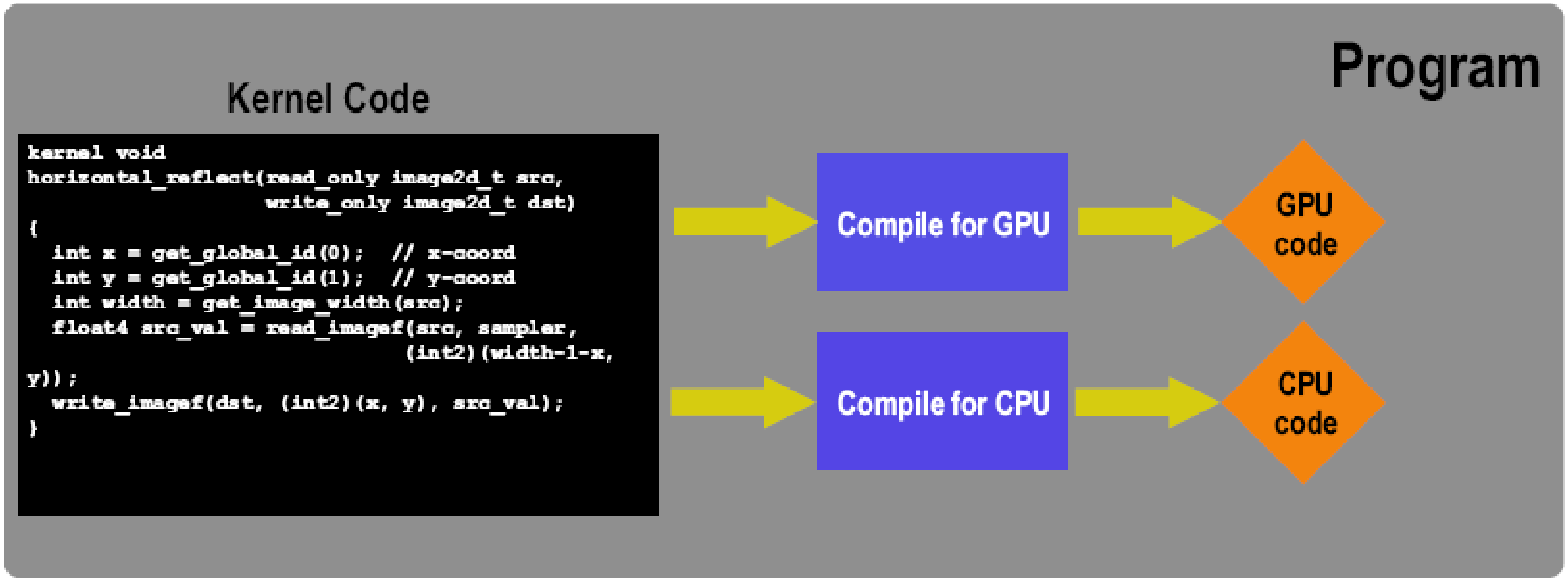
```
__kernel void
dp_mul(global const float *a,
      global const float *b,
      global float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
```

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

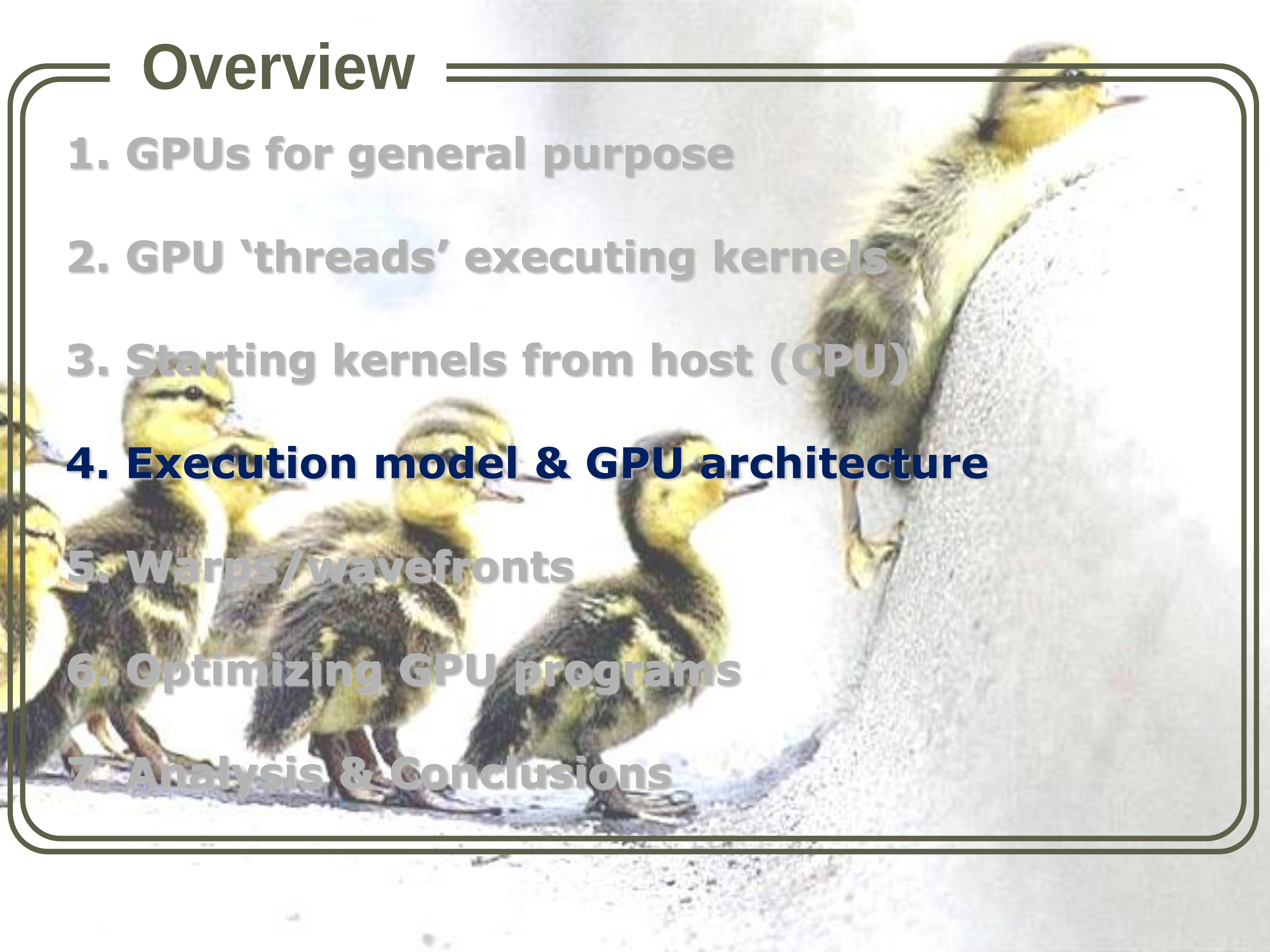


```
kernel void
horizontal_reflect(read_only image2d_t src,
                   write_only image2d_t dst)
{
  int x = get_global_id(0);  // x-coord
  int y = get_global_id(1);  // y-coord
  int width = get_image_width(src);
  float4 src_val = read_imagef(src, sampler,
                               (int2)(width-1-x,
y));
  write_imagef(dst, (int2)(x, y), src_val);
}
```

Cleanup

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

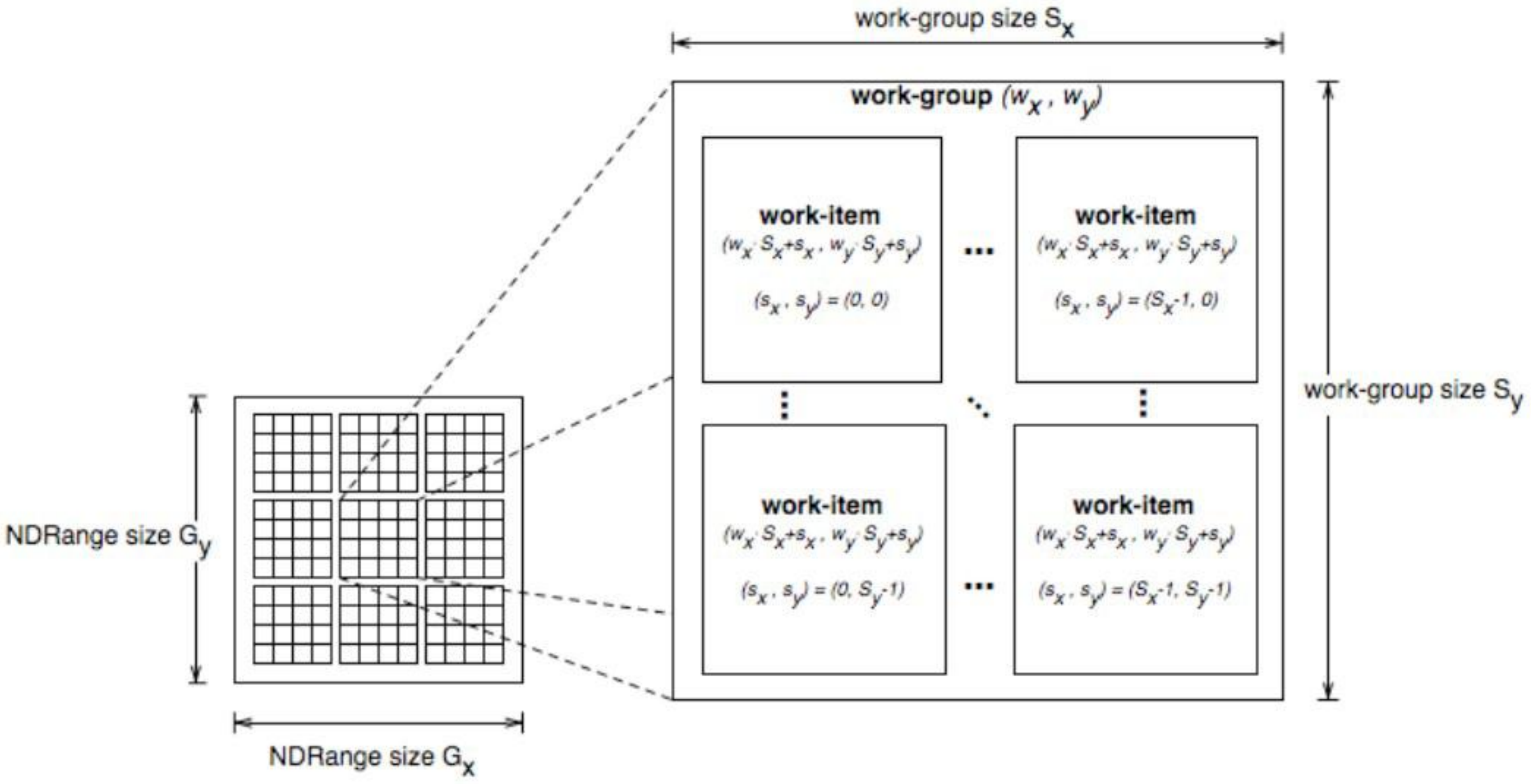
Overview



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
  - Also called *stream computing*

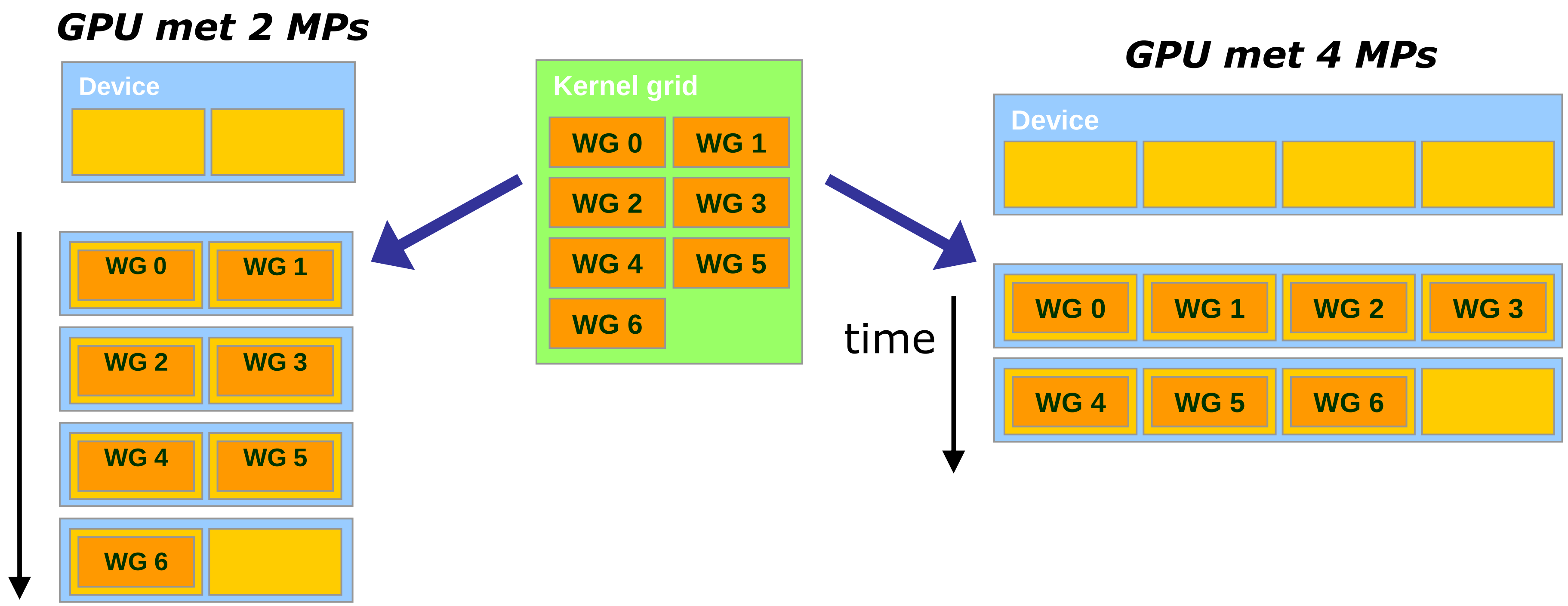
Architecture – Execution Model



work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

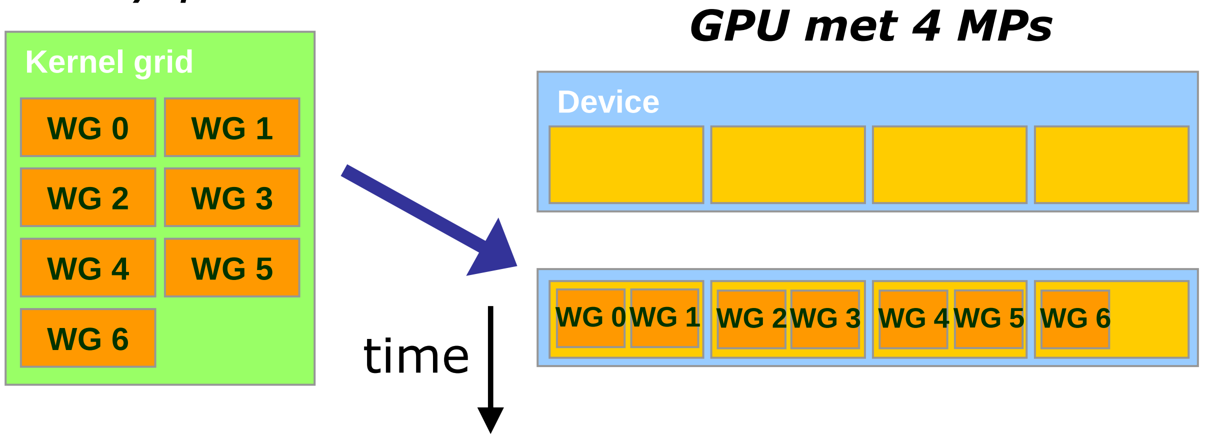
Kernel execution

- 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!

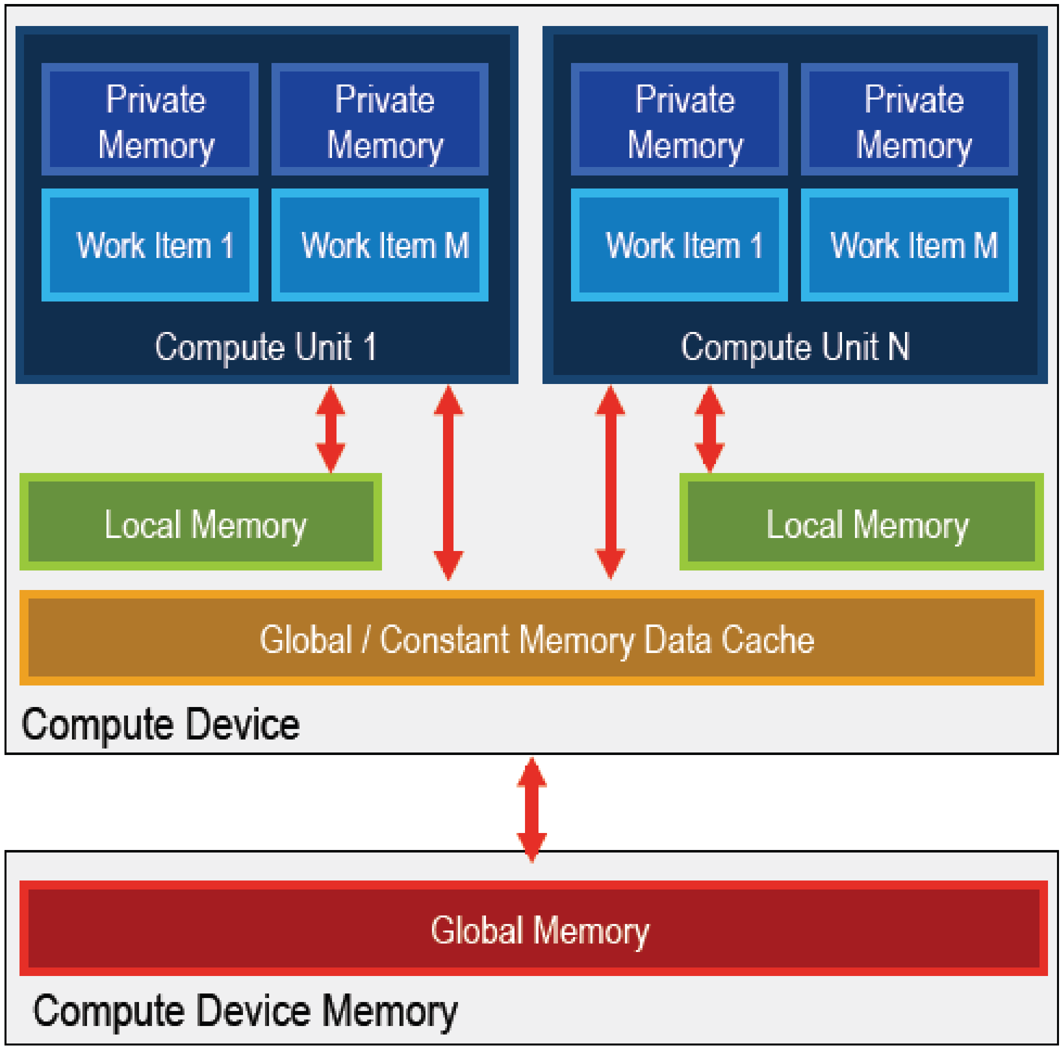


Multiple WGs per MP

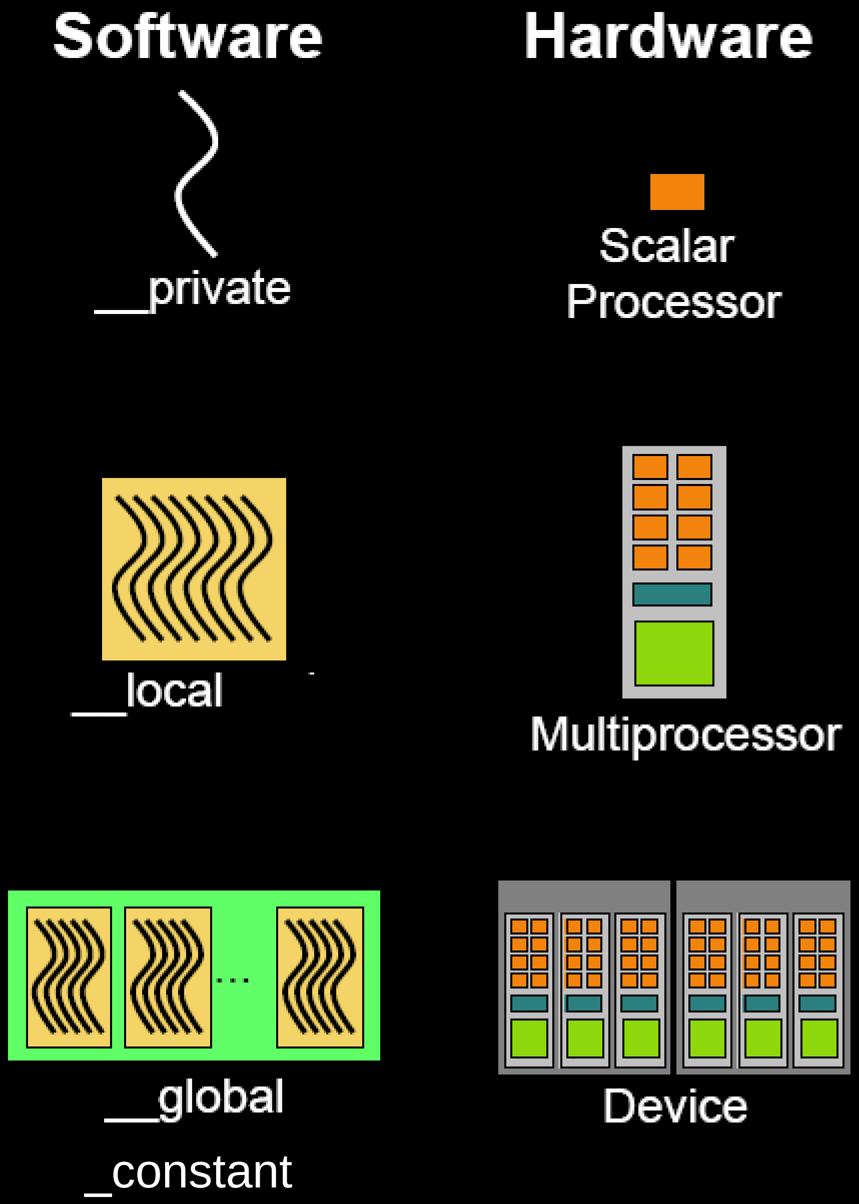
- One MP can execute work groups concurrently
- Determined by available resources (hardware limits):
  - *Max. work groups simultaneously on MP: 8*
  - *Private memory per MP: 16/48KB*
  - *Local memory per MP: 16/32KB*



Architecture – Memory Model



OpenCL Memory Model on NVIDIA



- Each hardware thread has a dedicated __private region for stack

- Each multiprocessor has dedicated storage for __local memory and __constant caches

- Work-items running on a multiprocessor can communicate through __local memory

- All work-groups on the device can access __global memory

- Atomic operations allow powerful forms of global communication

Example

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

Work group execution

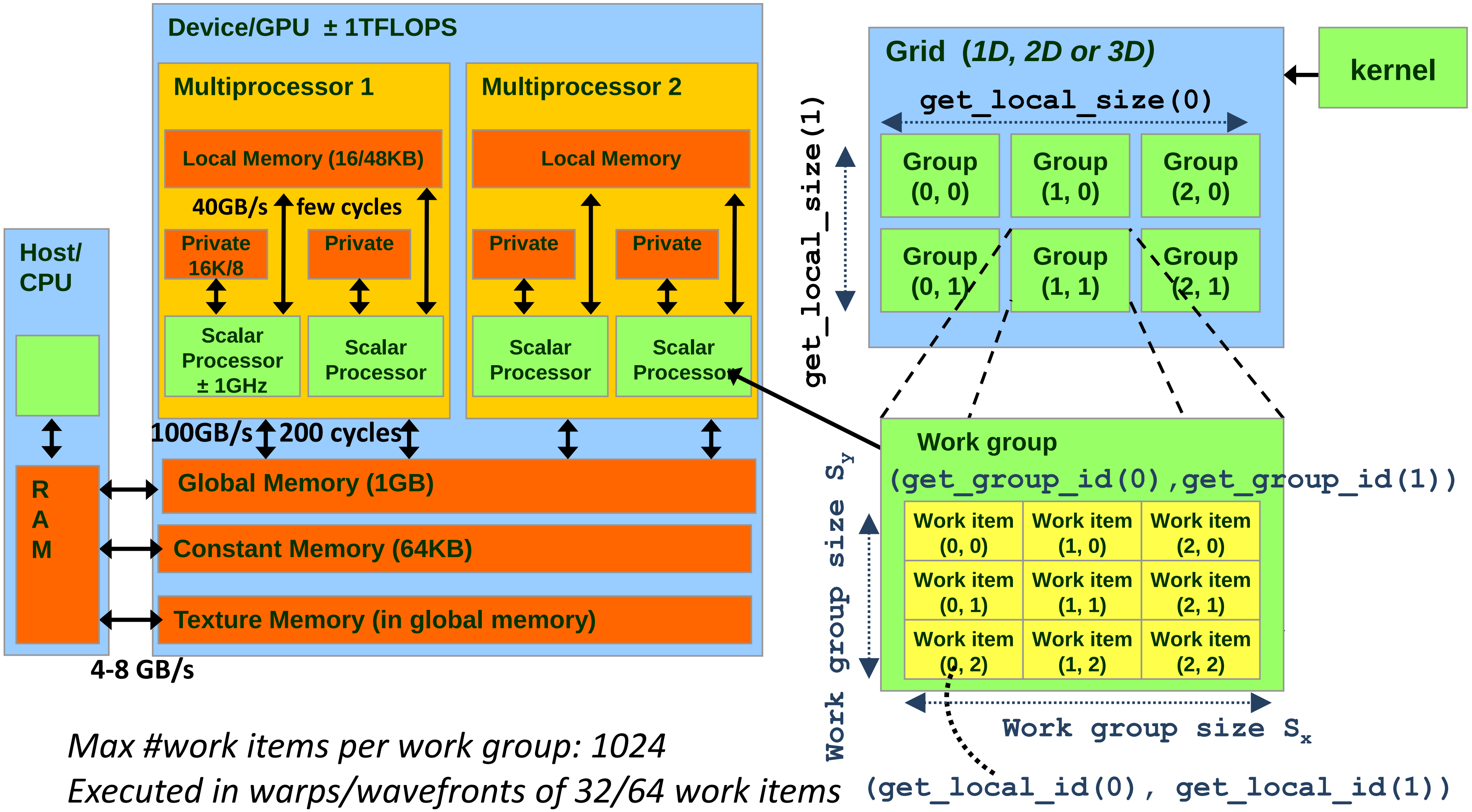
- Work items can synchronize within a work group

```
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

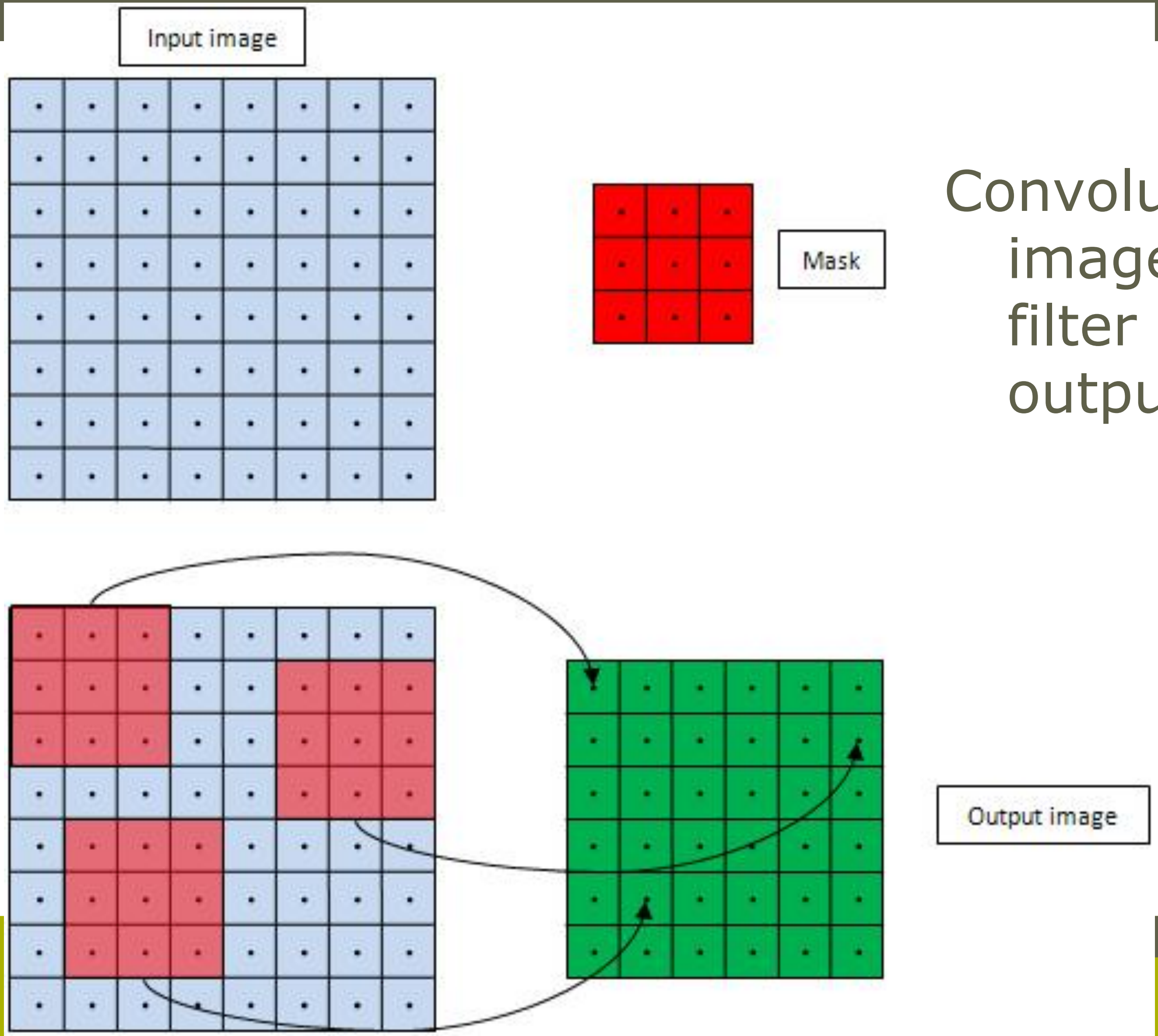
Max active warps on MP: 24/48

OpenCL terminology

GPU Threads v/s CPU Threads

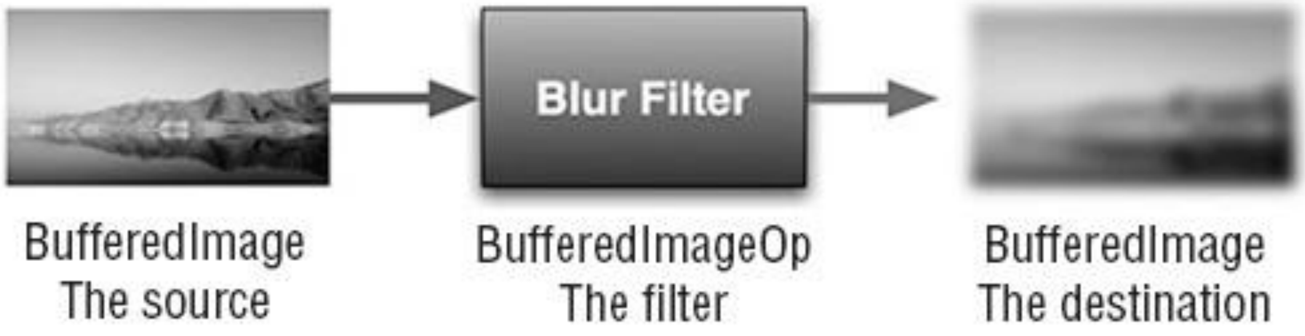
- GPU work items or *threads*:
  - **Lightweight,** small creation and scheduling overhead, extremely **fast switching between threads**
    - No context switch is required
  - **Need to issue 1000s of GPU threads** to hide global memory latencies (600-800 cycles)
    - GPU=Thread processor, upto 96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

Convolution of images

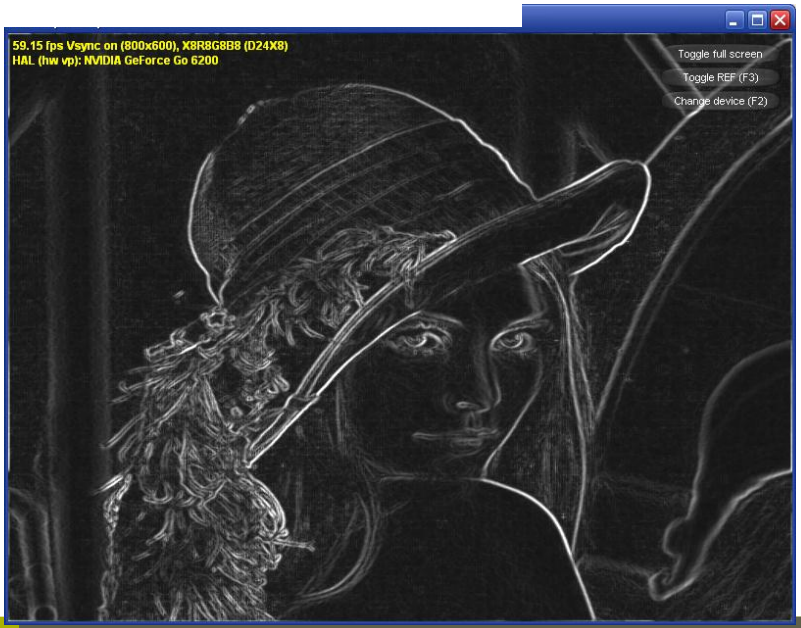


Convolution of a 8x8 image with a 3x3 filter to yield a 6x6 output image

Examples of convolution

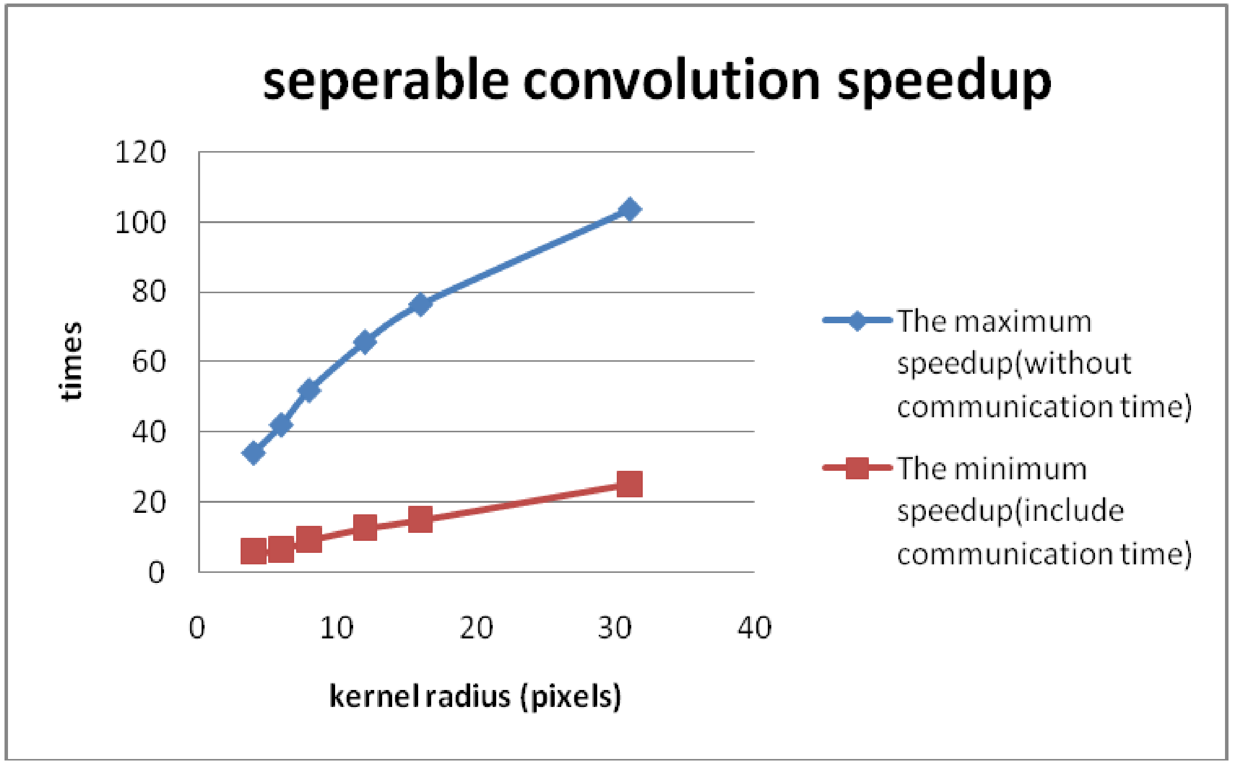


Edge detection with sobel filter

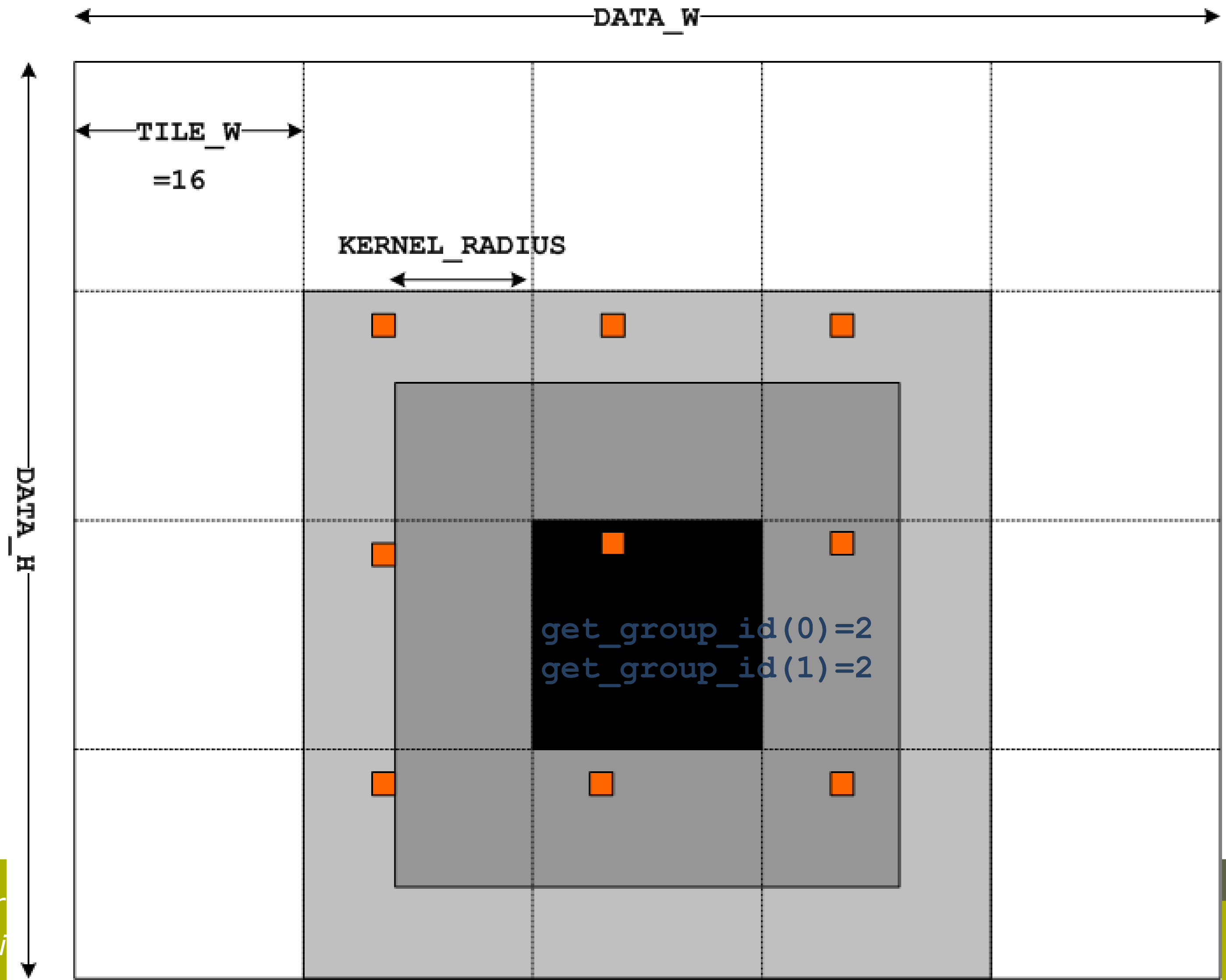


```
59.15 fps Vsync on (800x600), X8R8G8B8 (D24X8)
HAL (hw vp): NVIDIA GeForce Go 6200
```

Speedup


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

Convolution on GPU

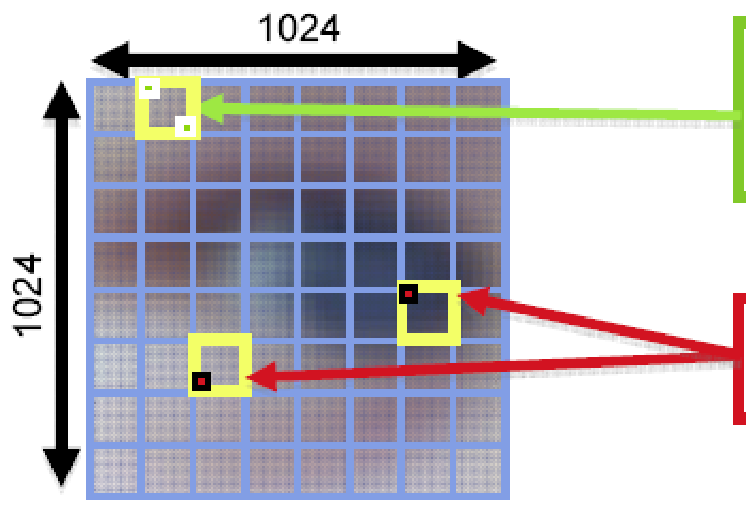


Convolution Kernel

```
__kernel void bad_shared(
        __global int *in, __global int *out, __local  int *in_local, __constant int *filter)
{
    uint row = get_group_id(1) * get_local_size(1) + get_local_id(1);
    uint col = get_group_id(0) * get_local_size(0) + get_local_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_width; ++i)
         for (int j = 0; j< filter_height; ++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)

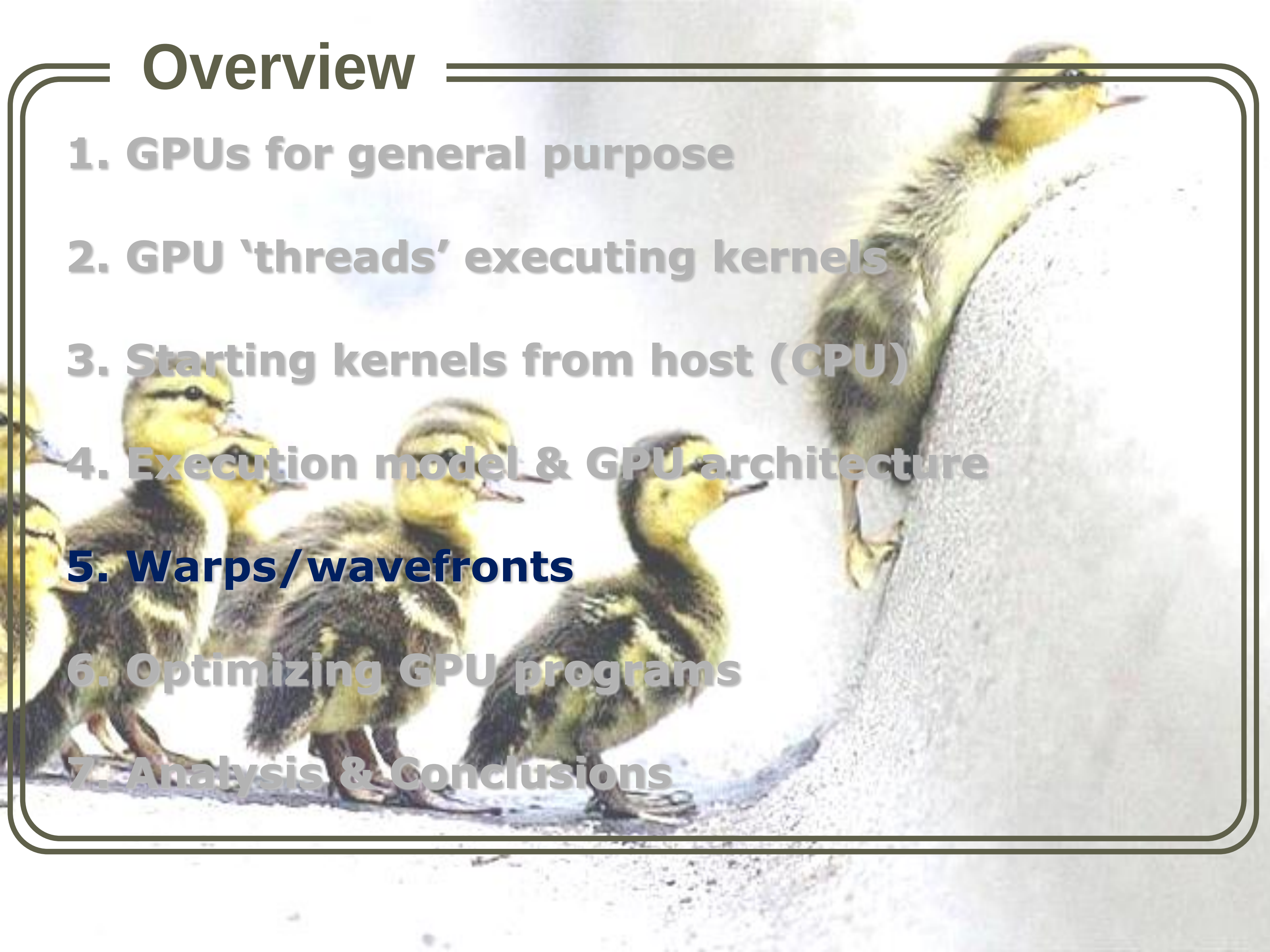


Synchronization between work-items possible only within workgroups: barriers and memory fences

Cannot synchronize outside of a workgroup

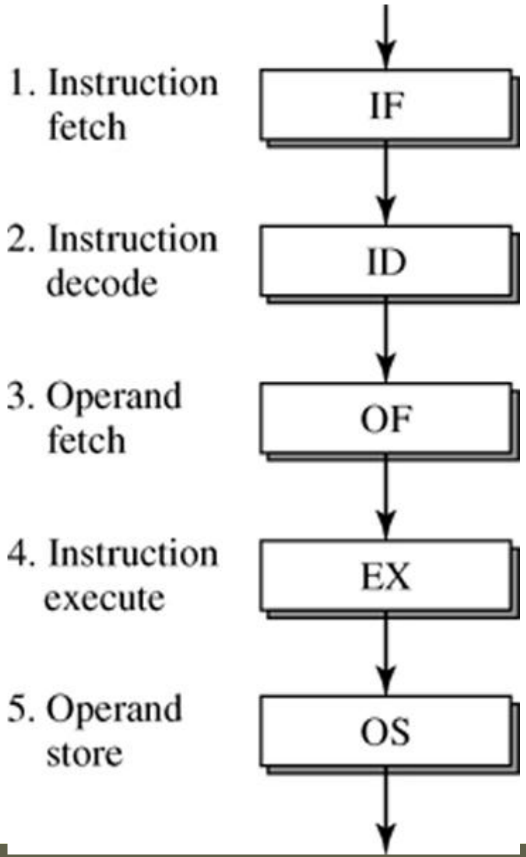
- Choose the dimensions that are "best" for your algorithm

Overview



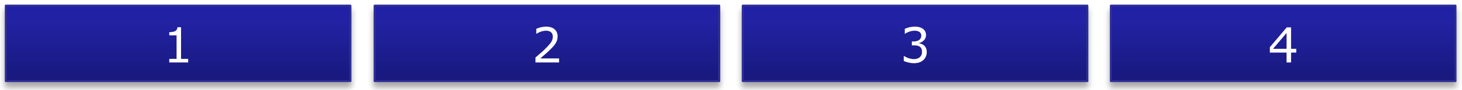
Scalar Processor: Pipelined design

- 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

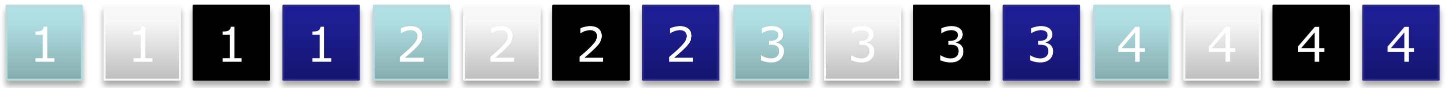


Pipelining Principle

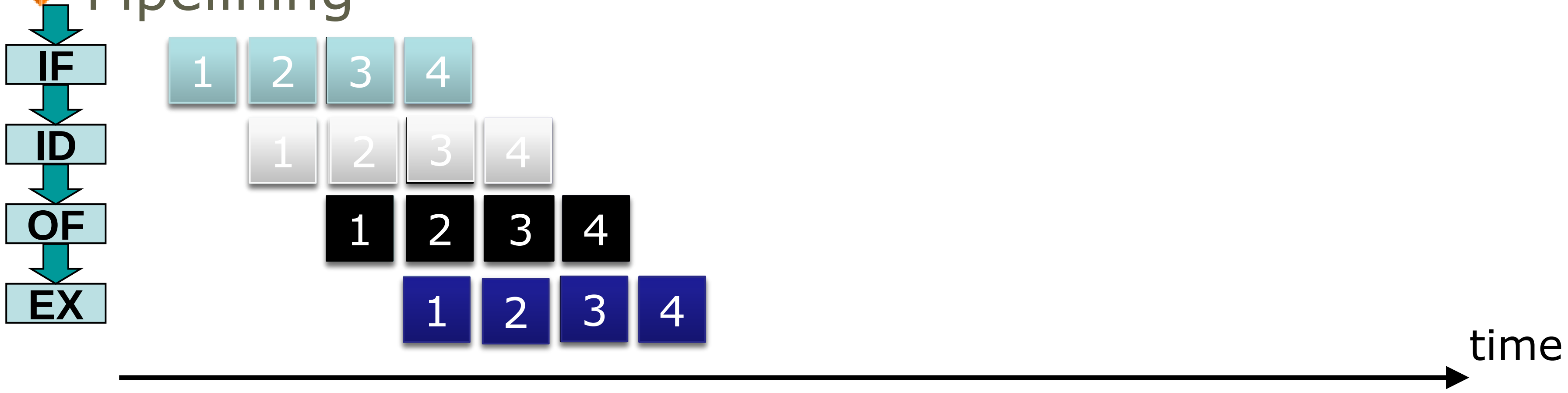
- Long operations



- Combination of short operations



- Pipelining

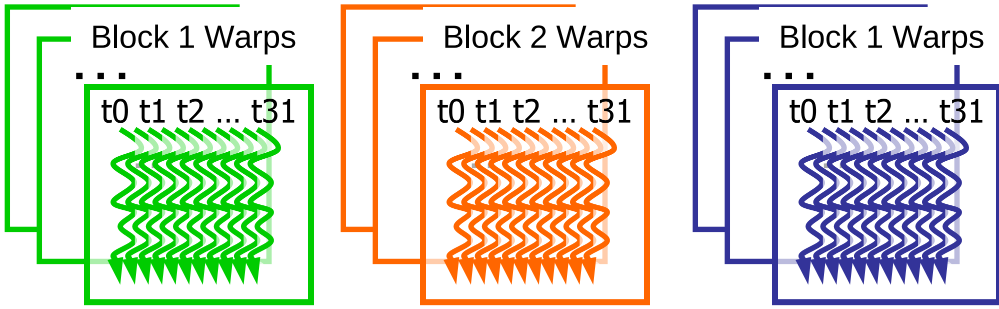


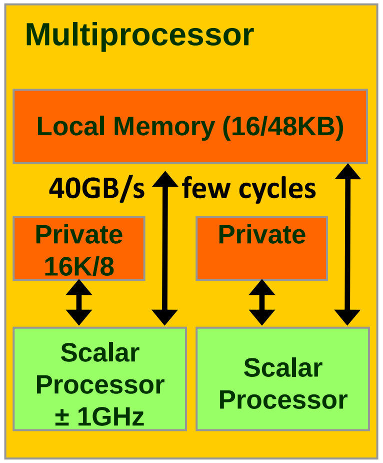
Pipelining

- 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.



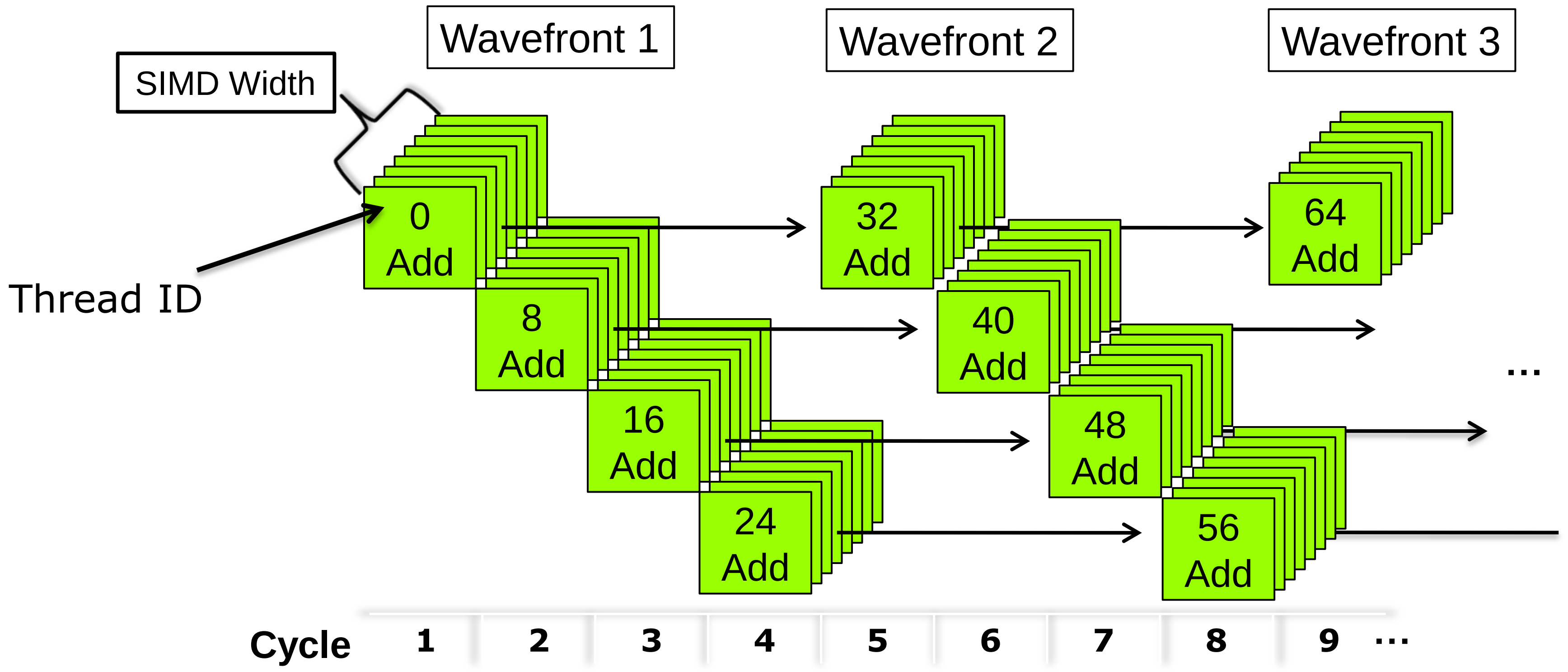


Example: 3 work groups on MP, each group has 256 work items, how many Warps are there in the MP?

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

Warp/wavefront execution

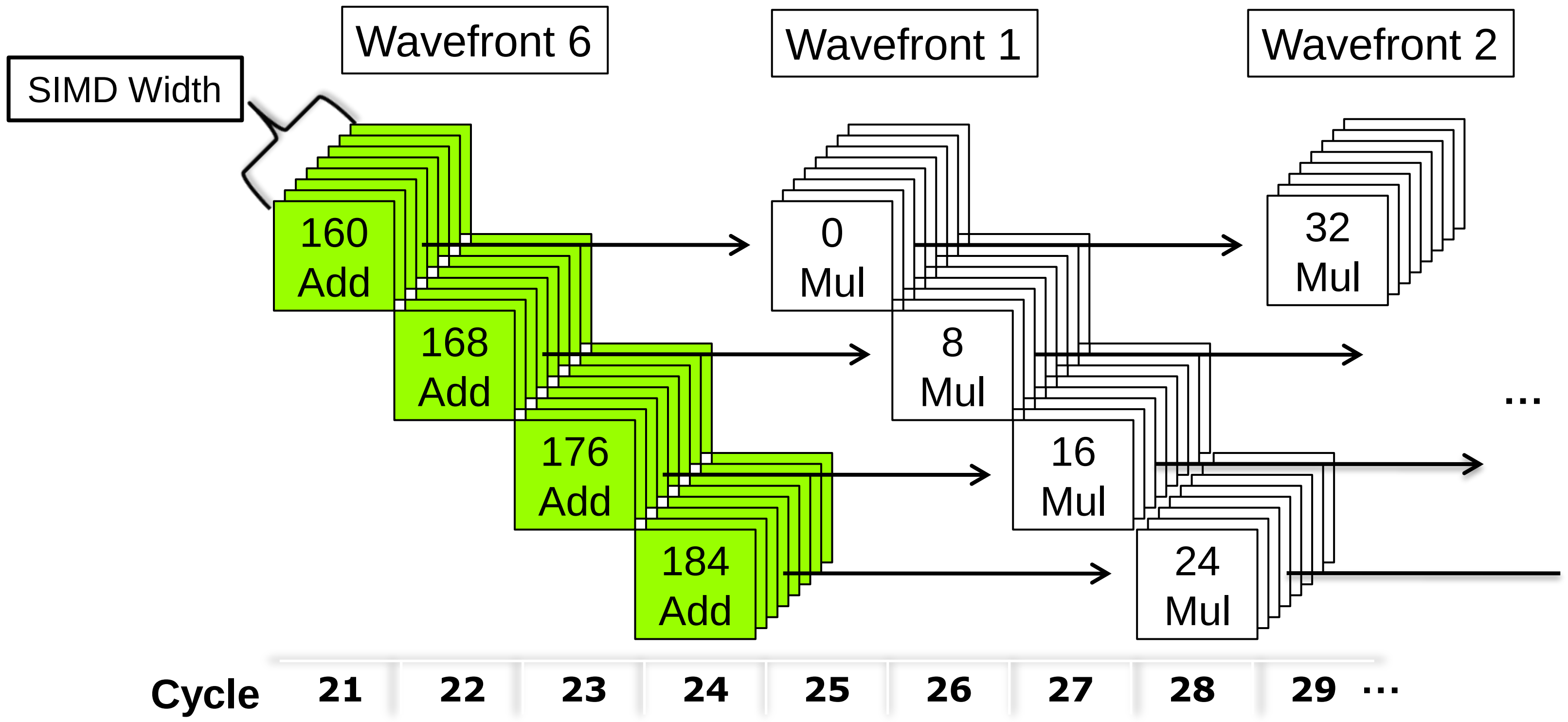
- Work items are sent into pipeline grouped in warp /wavefront
  - ALUs all execute the same instruction: Single Instruction, Multiple Threads (SIMT)
  - 32 work items / 8 SPs => 4 cycles



Warp/wavefront execution

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

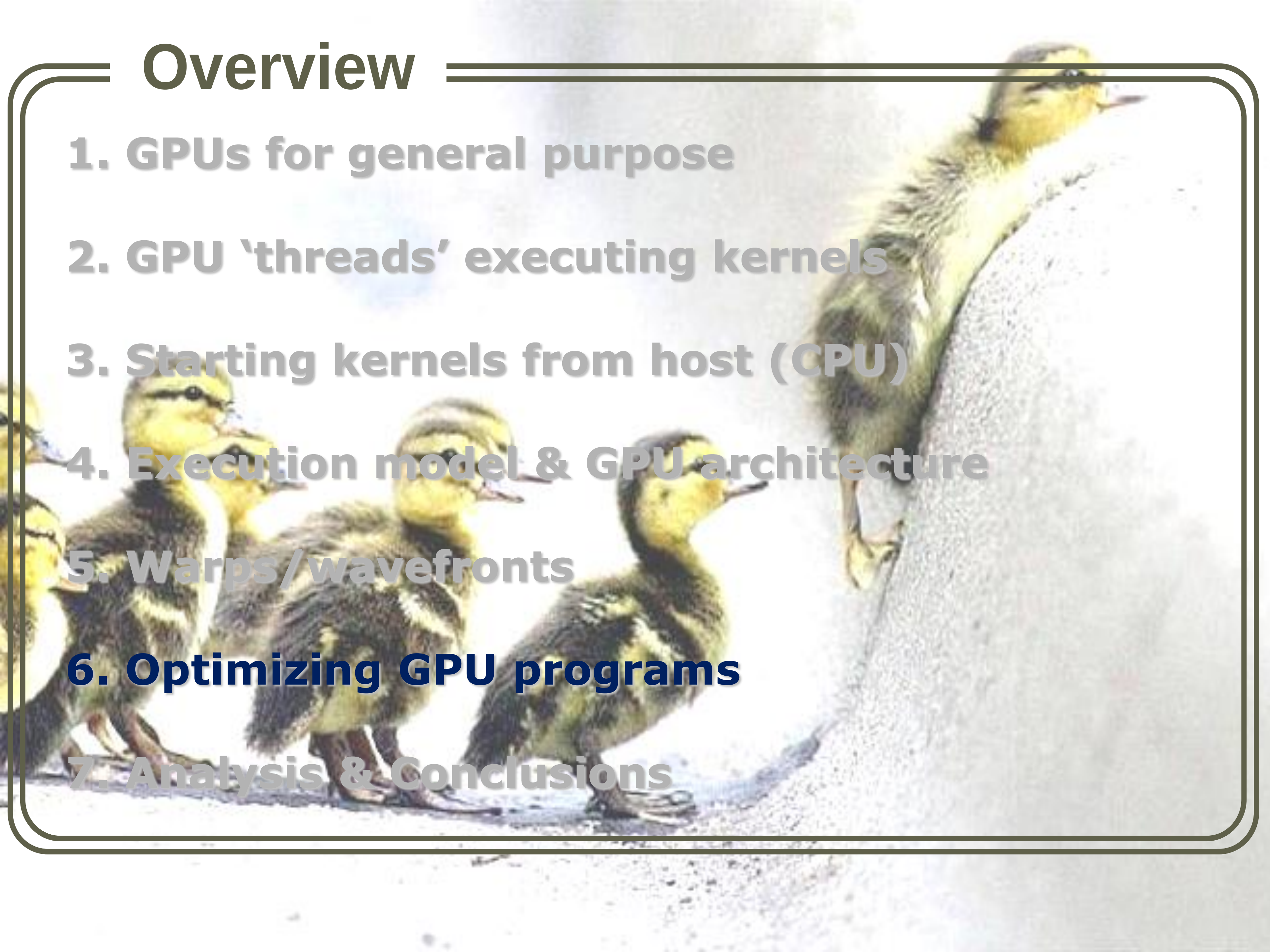
  ⇒ pipeline has independent instructions => no stalling

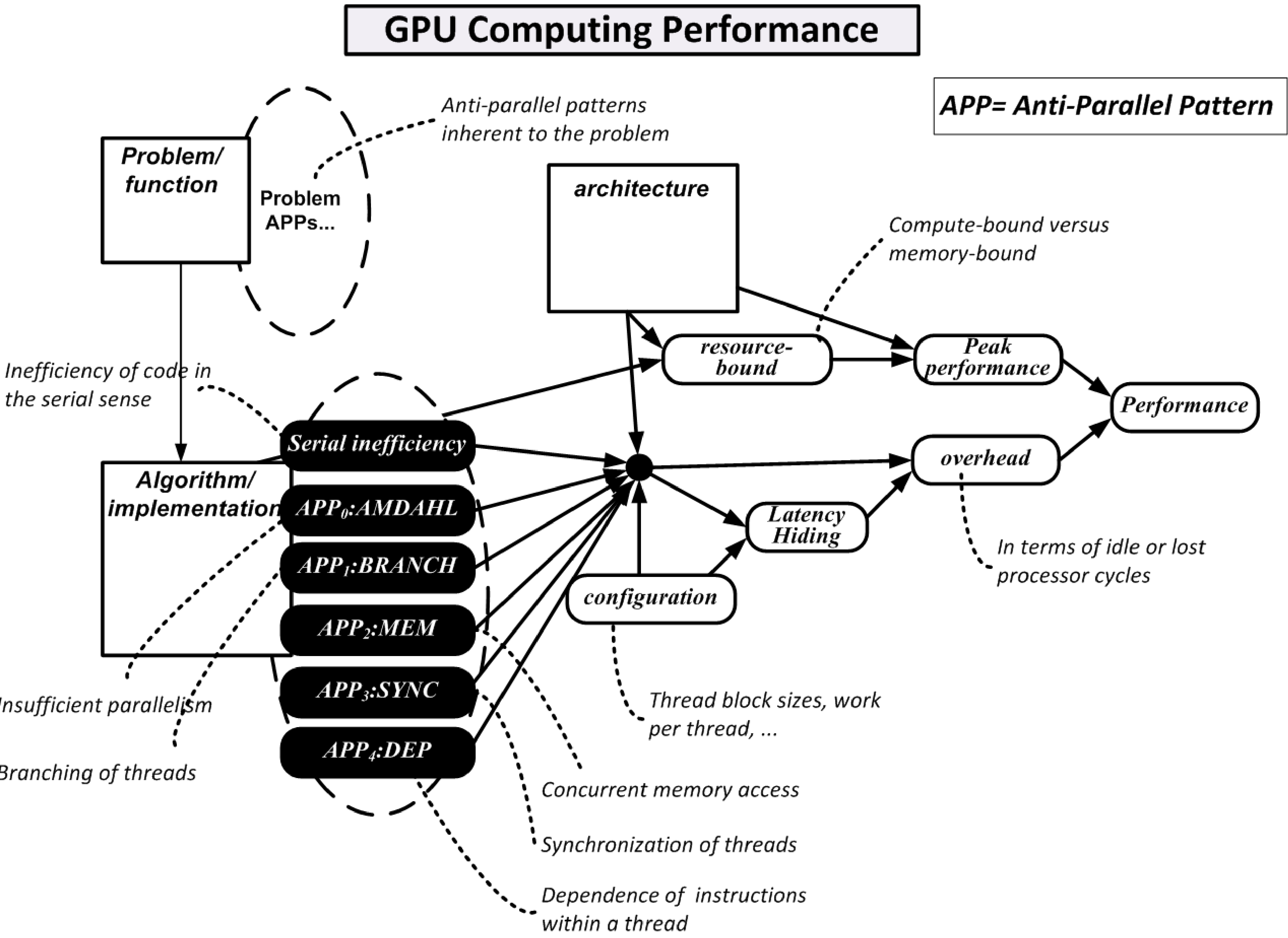


Why do we have to consider warps/wavefronts?

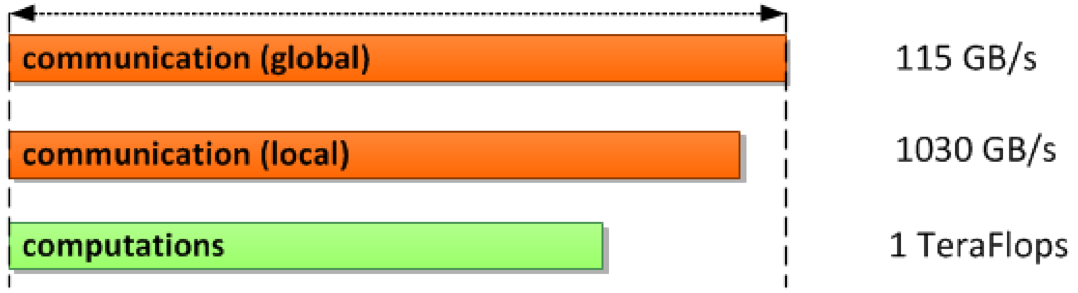
- Different **branching** of threads within a warp incurs ‘lost cycles’
  - SIMT execution happens per warp/wavefront, in lockstep
- **Memory access pattern** of a warp should be considered
  - Memory access also happens per warp
  - Not all access patterns can happen concurrently (*see further*)

Overview



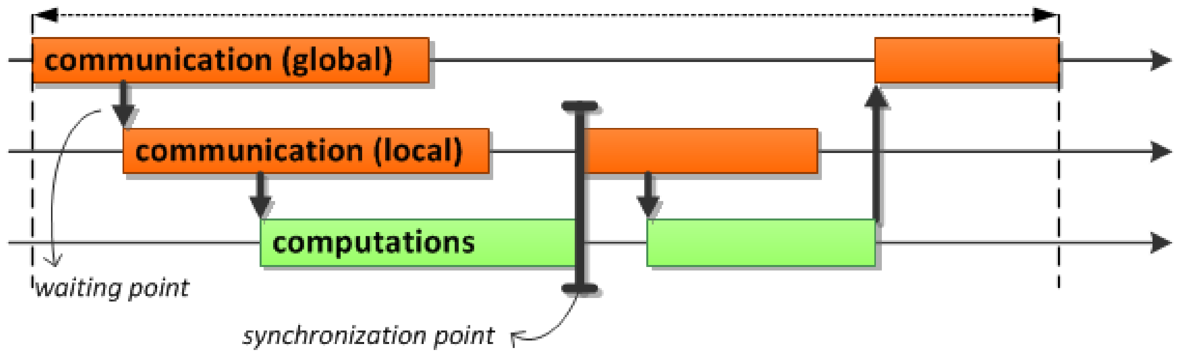

GPU Computing Performance
APP= Anti-Parallel Pattern
Anti-parallel patterns inherent to the problem
Problem/ function
Problem APPs...
architecture
Compute-bound versus memory-bound
resource-bound
Peak performance
Performance
Inefficiency of code in the serial sense
Serial inefficiency
Algorithm/ implementation
APP$_0$:AMDAHL
APP$_1$:BRANCH
APP$_2$:MEM
APP$_3$:SYNC
APP$_4$:DEP
overhead
Latency Hiding
configuration
In terms of idle or lost processor cycles
Thread block sizes, work per thread, ...
Insufficient parallelism
Branching of threads
Concurrent memory access
Synchronization of threads
Dependence of instructions within a thread

A. Peak Performance



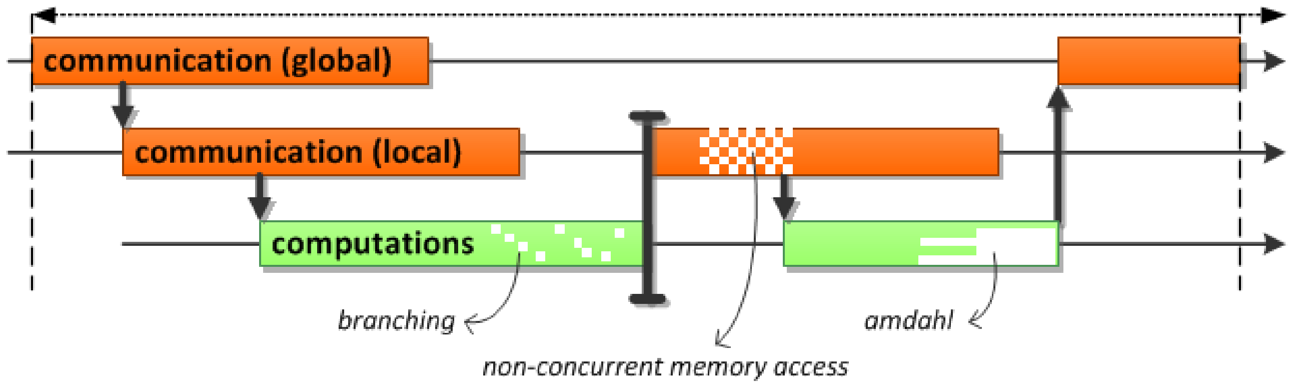
Roofline model

B. Non-overlap



Non-overlap factors

C. Anti-parallel interactions



Anti-parallel patterns & model for latency hiding

General approach

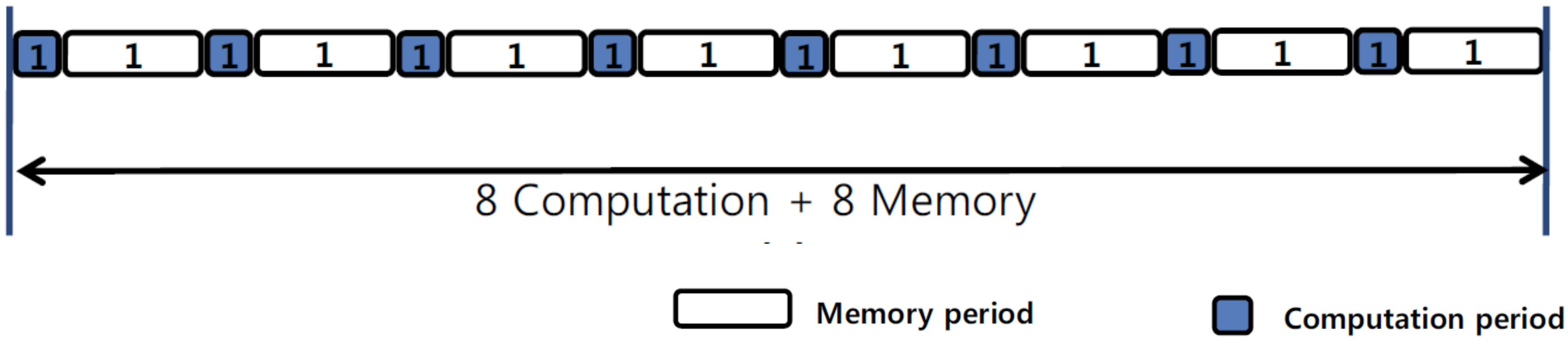
- Estimate a performance bound for your kernel
  - ***Compute bound***: t_1 = #operations / #operations per second
  - ***Data bound***: t_2 = # memory accesses / # accesses per second
  - $t_{min} = \min(t_1, t_2)$ *expressed by roofline model*
- Measure the actual runtime
  - $t_{actual} = t_{min} + t_{delta}$
- 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

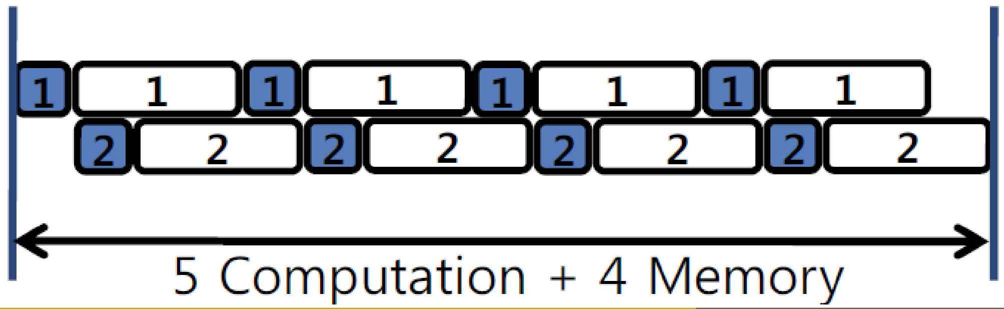
- Definition: *Common parts of kernel code that work against the available parallelism*.
  - Can be inferred from the source code
  - Map parts of the source to parallel overhead
- Systematic way to categorize performance topics
- Systematic way to optimize kernels
- *PhD research of Jan G. Cornelis*

Latency Hiding

- 1 warp, without latency hiding



- 2 warps running concurrently

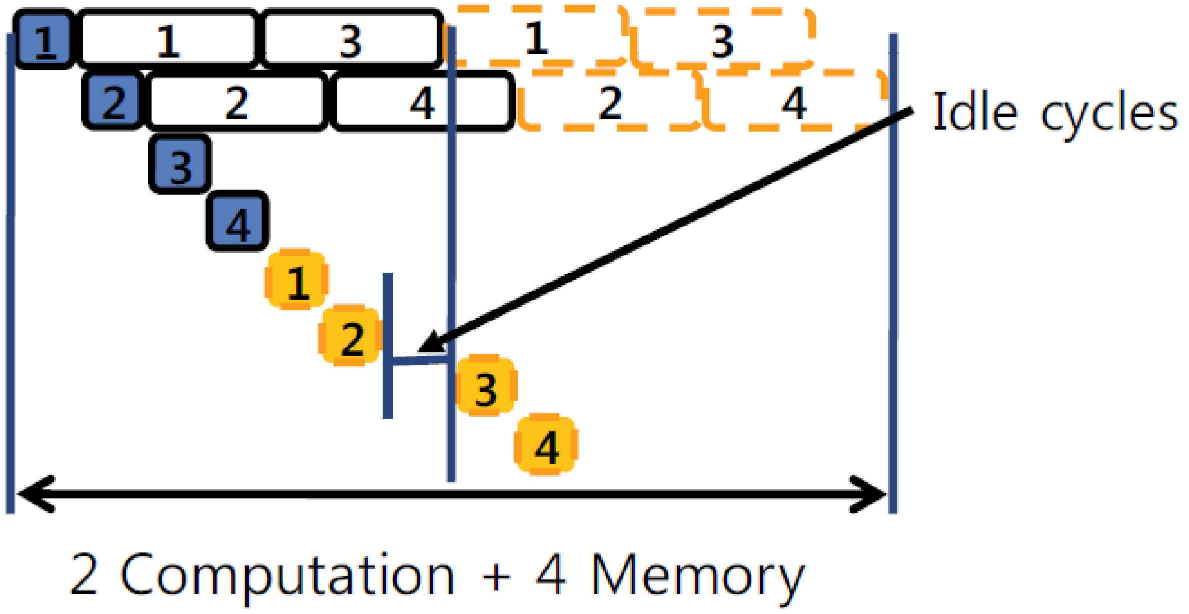


Latency Hiding for Memory Accesses

- 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

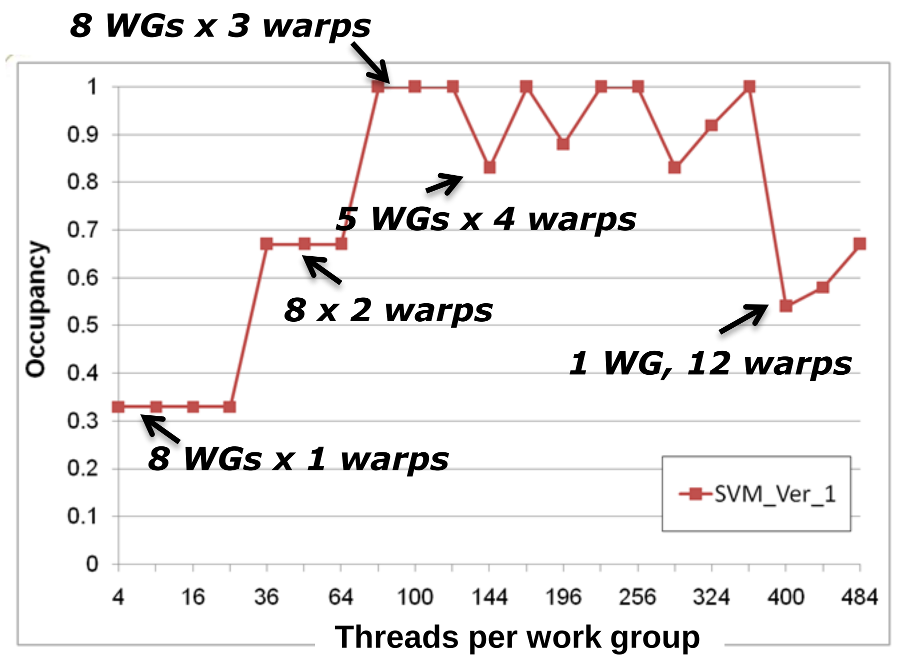
- But only 2 concurrent memory transactions

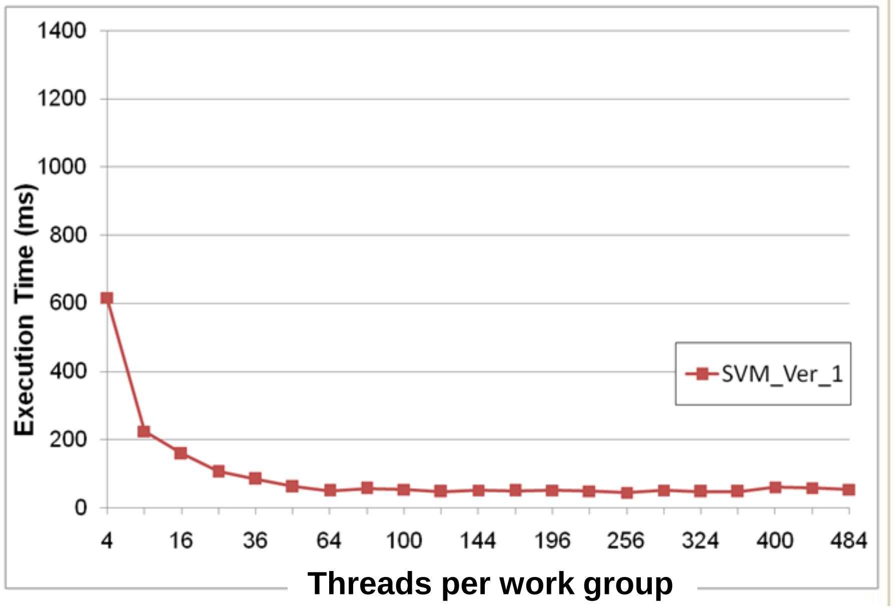


▭	1st Memory period	🟦	1st Computation period
⬚	2nd Memory period	🟨	2nd Computation period

Keep occupancy high

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

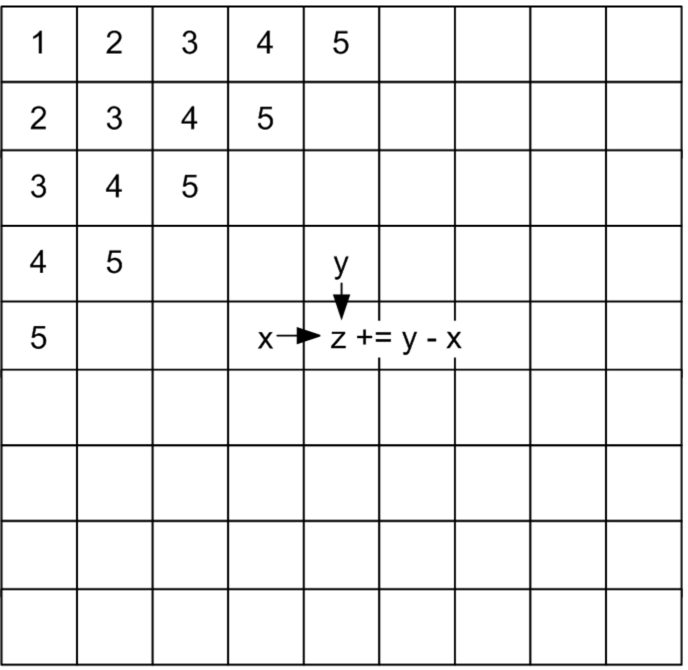




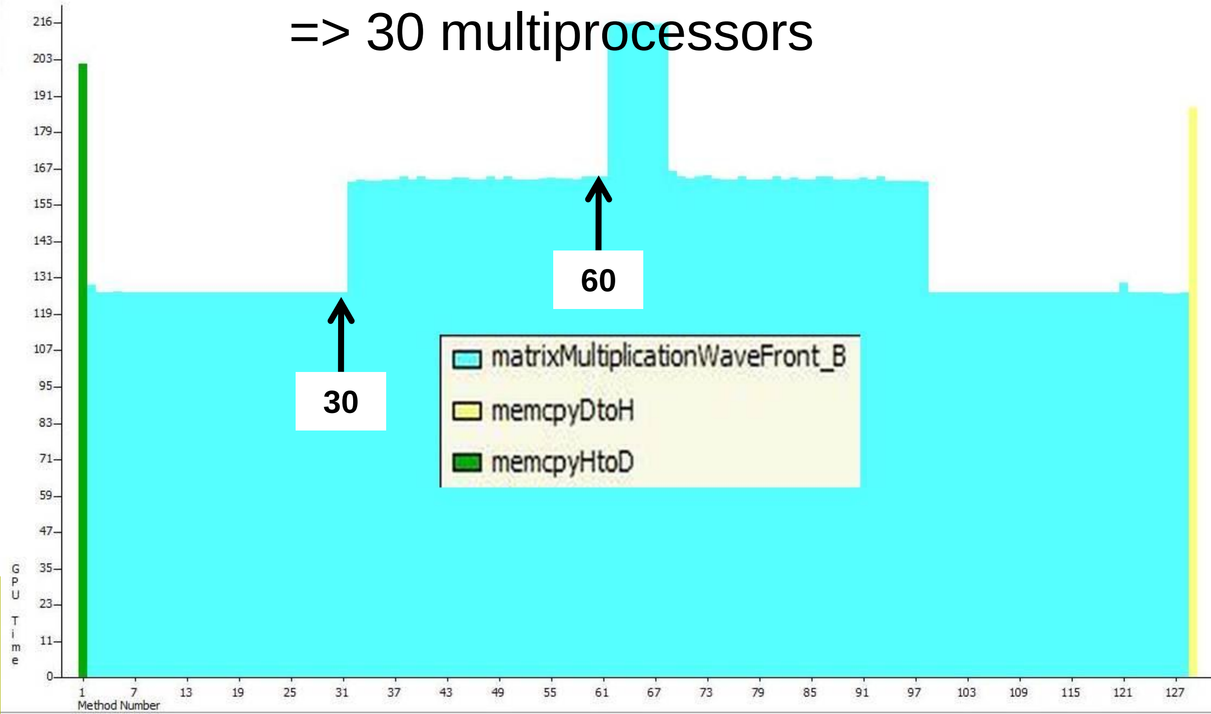
- Conclusion: in general, higher occupancy leads to a better performance

APP0: AMDAHL

Wavefront algorithm



- 512x512 image divided into work groups of 8x8, handled by 1 work group => 64 x 64 work groups
- 1 work group depends on results of 2 work groups => global synchronization necessary => a kernel call per wave
- On GTX280: 240 cores => 30 multiprocessors

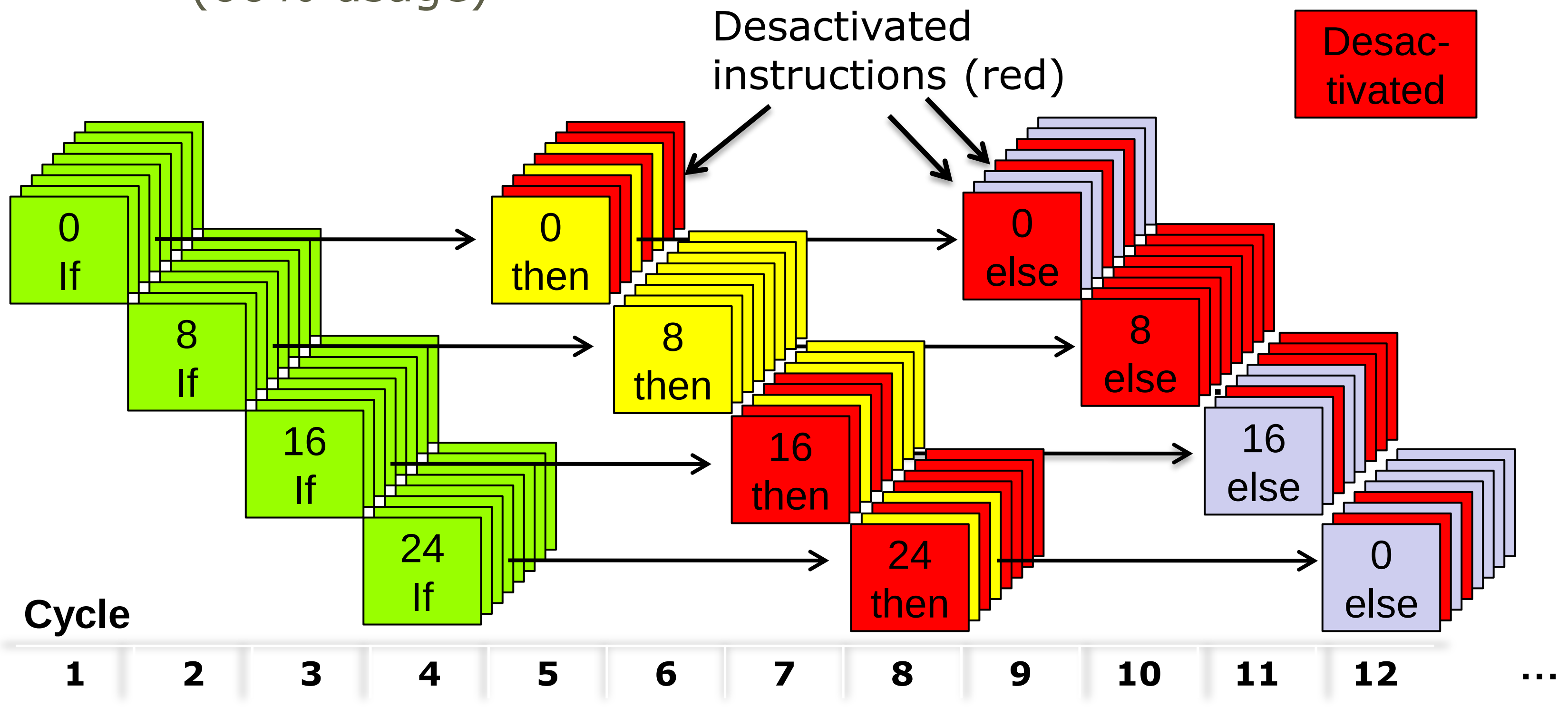


SIMT Conditional Processing

- 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:

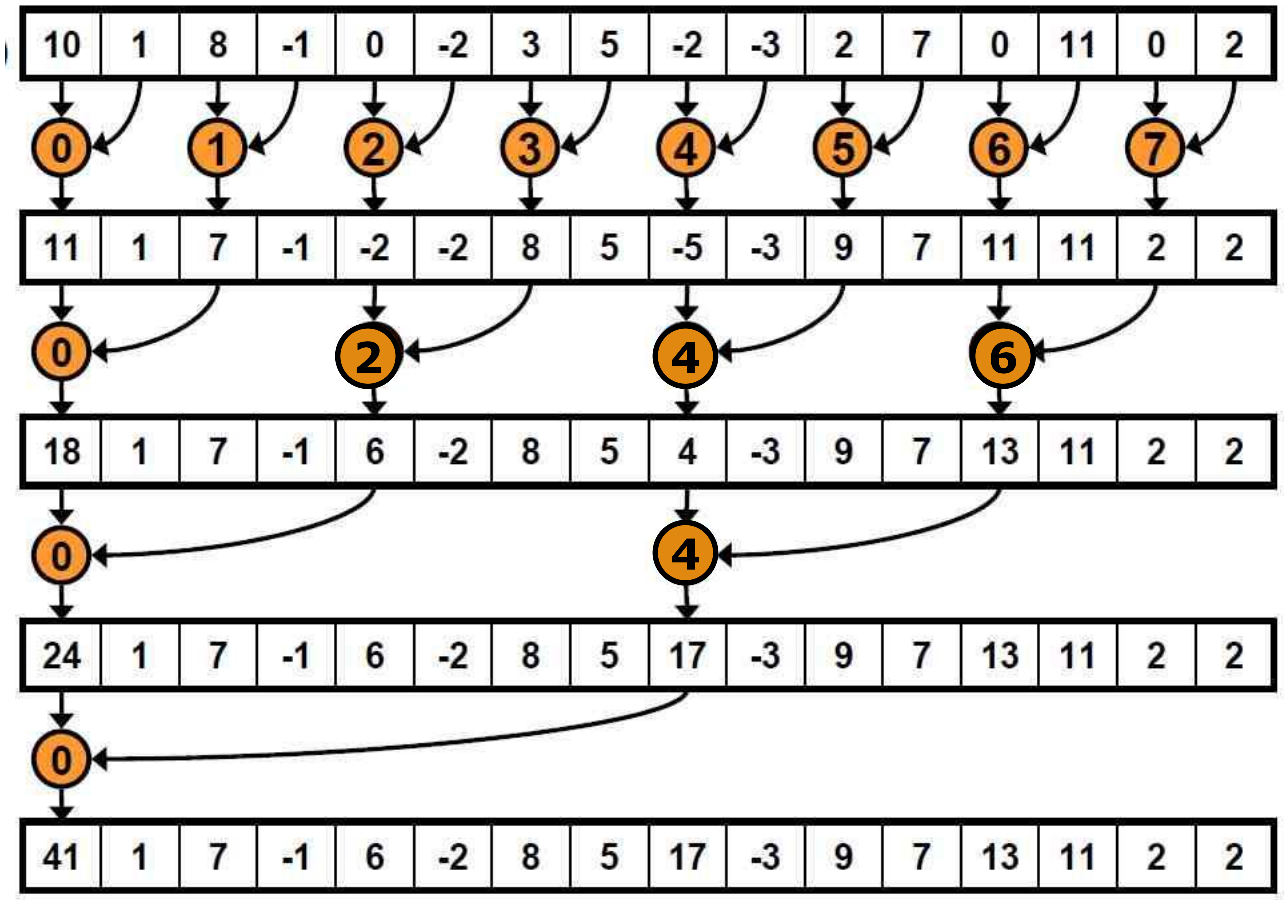
```
if (x < 5)  y = 5; else y = -5;
```

  - SIMD performs the 3 steps
  - `y = 5;` is only executed by threads for which `x < 5`
  - `y = -5;` is executed by all others

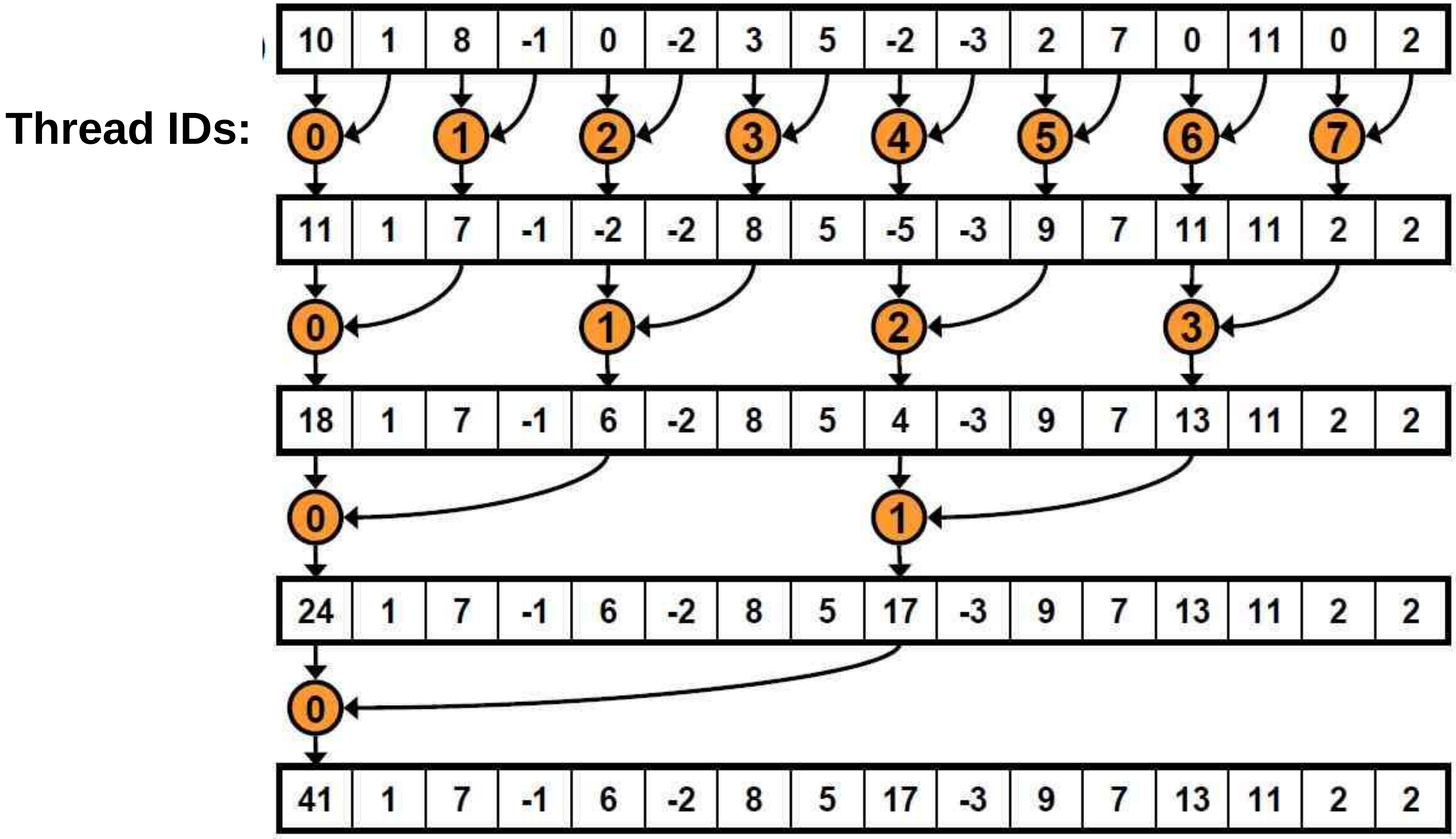
- *Warp branch divergence* decreases performance: <u>cycles are lost</u>
  - Solution: statically or dynamically reorder threads
- No latency hiding possible

Reduction operation (sum)

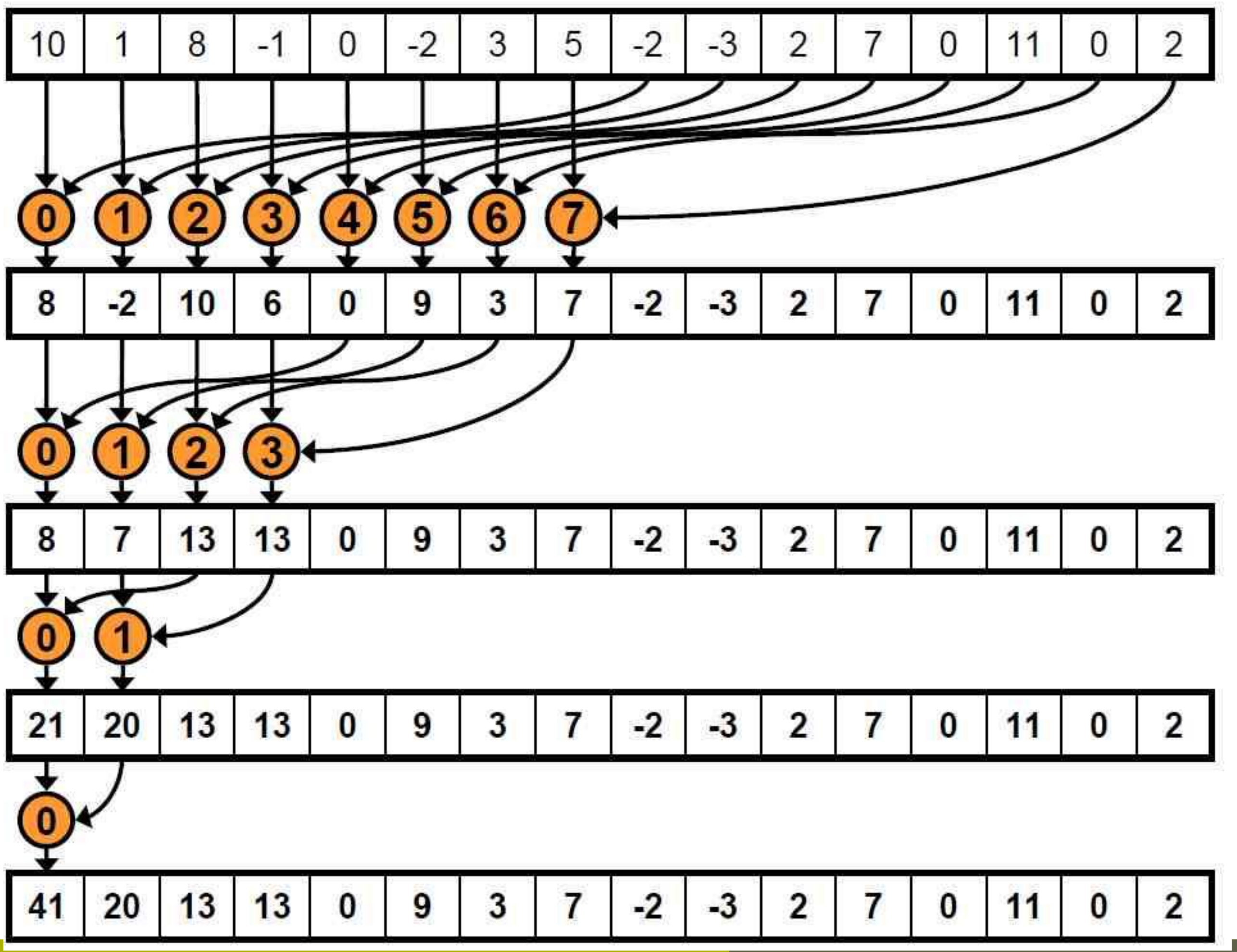
Thread IDs:



V2: group active threads



V3: Sequential addressing

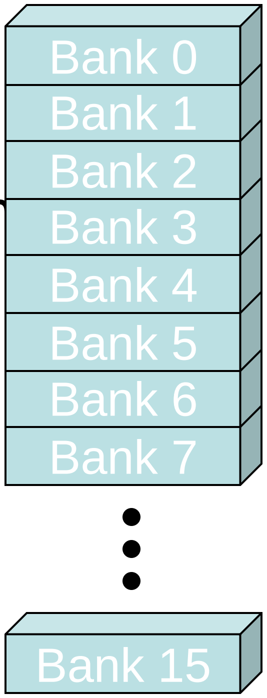


Global memory

- Memory coalescing for half-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

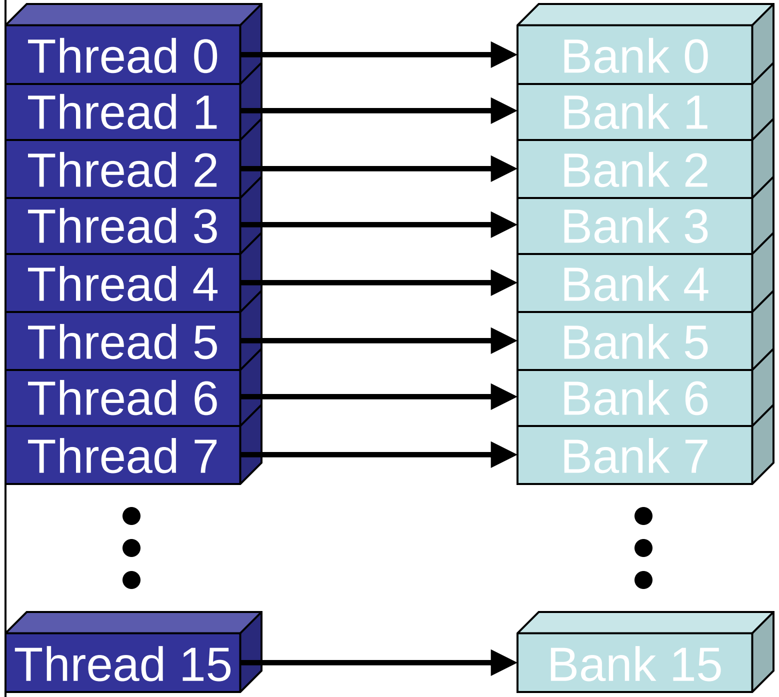
Local/Shared memory

- 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 a single ban
- No bank conflict:
  - all threads of a half-warp access different banks,
  - all threads of a half-warp access identical address,
    (broadcast)

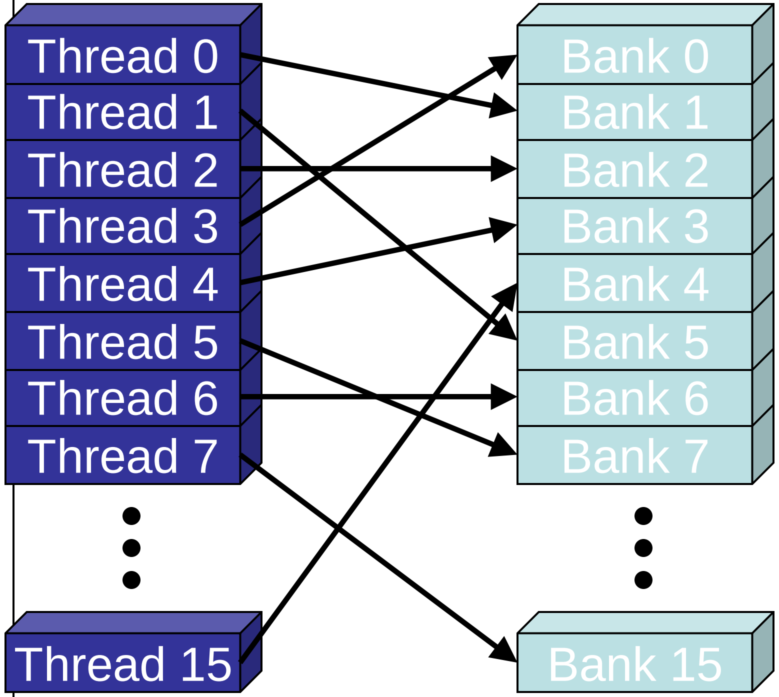


Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride of 1

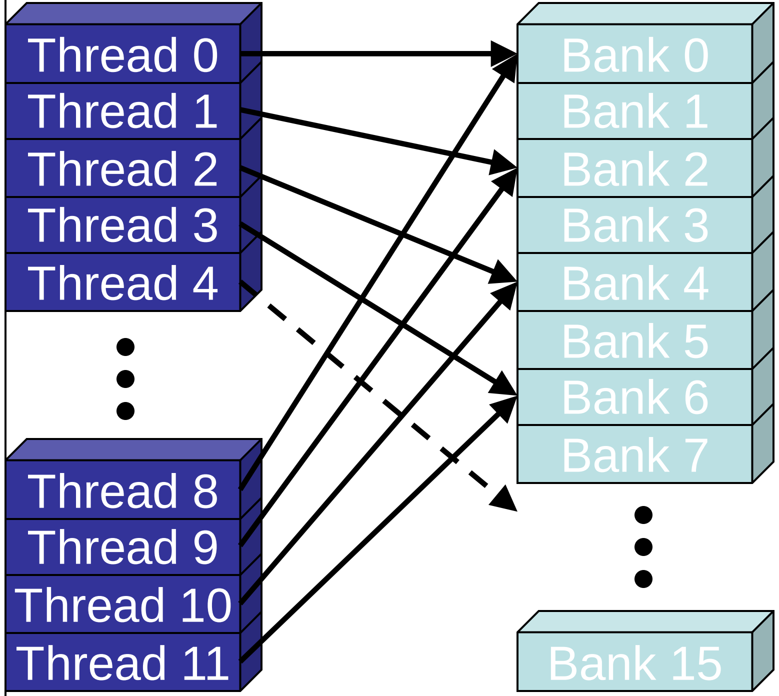


- No Bank Conflicts
  - Random 1:1 Permutation

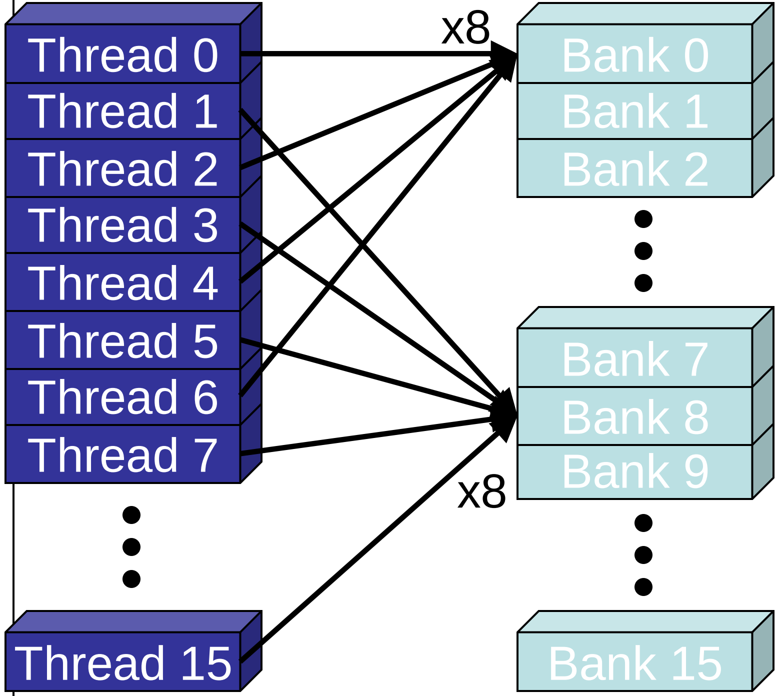


Bank Addressing Examples

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



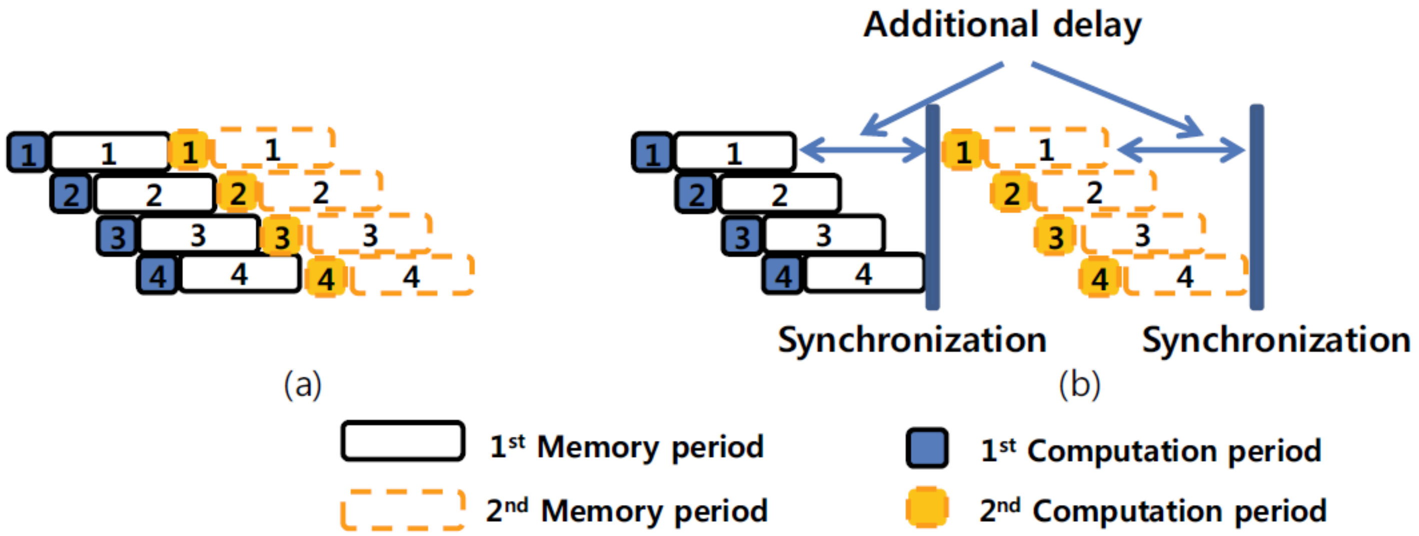
- 8-way Bank Conflicts
  - Linear addressing stride of 8



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

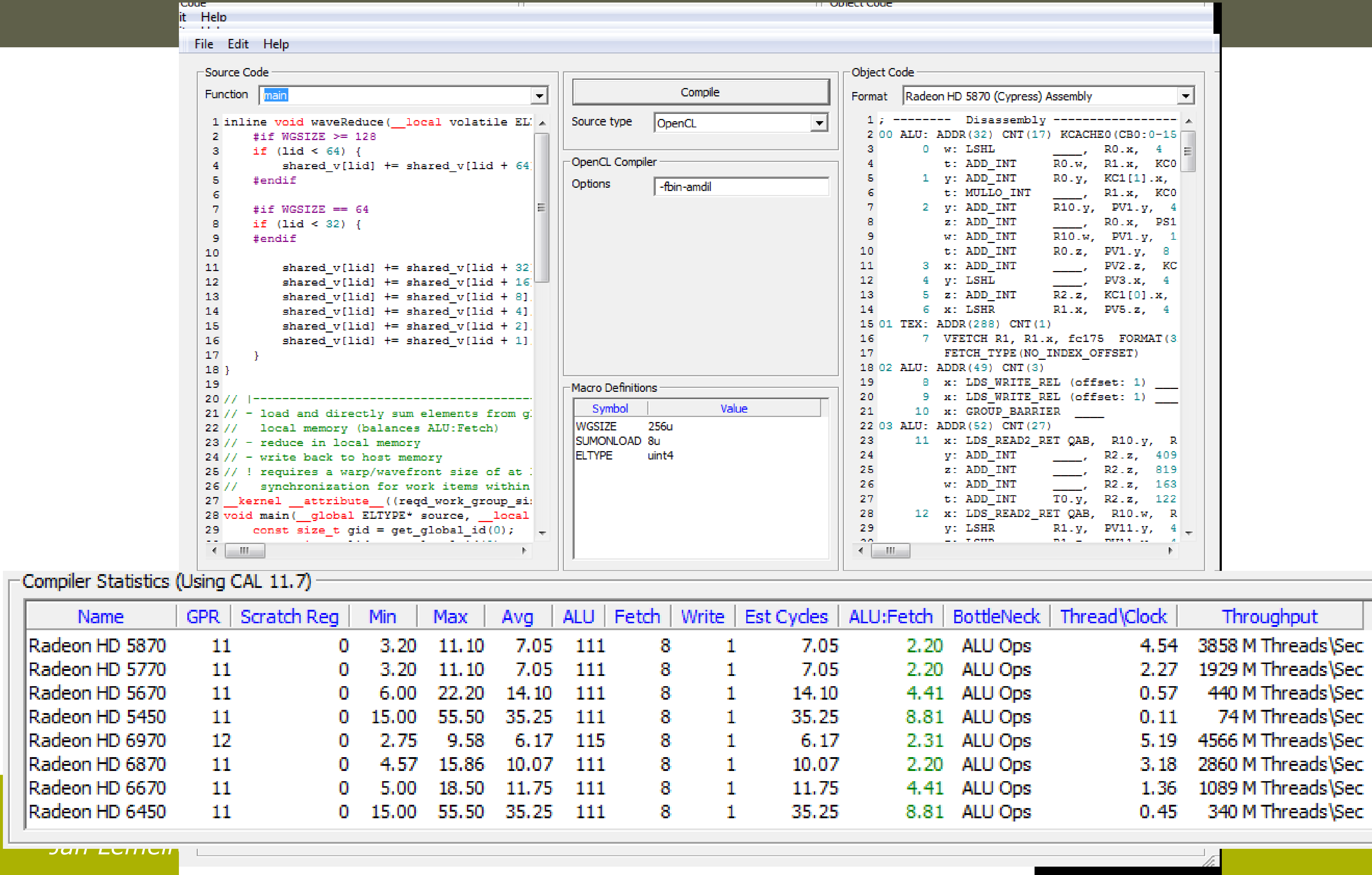
Lost cycles due to synchronization



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



```
inline void waveReduce(__local volatile EL
    #if WGSIZE >= 128
    if (lid < 64) {
        shared_v[lid] += shared_v[lid + 64
    #endif

    #if WGSIZE == 64
    if (lid < 32) {
    #endif

        shared_v[lid] += shared_v[lid + 32
        shared_v[lid] += shared_v[lid + 16
        shared_v[lid] += shared_v[lid + 8]
        shared_v[lid] += shared_v[lid + 4]
        shared_v[lid] += shared_v[lid + 2]
        shared_v[lid] += shared_v[lid + 1]
    }
}

// |----------------------------------------
// - load and directly sum elements from g
//   local memory (balances ALU:Fetch)
// - reduce in local memory
// - write back to host memory
// ! requires a warp/wavefront size of at
//   synchronization for work items within
__kernel __attribute__((reqd_work_group_si
void main(__global ELTYPE* source, __local
    const size_t gid = get_global_id(0);
```

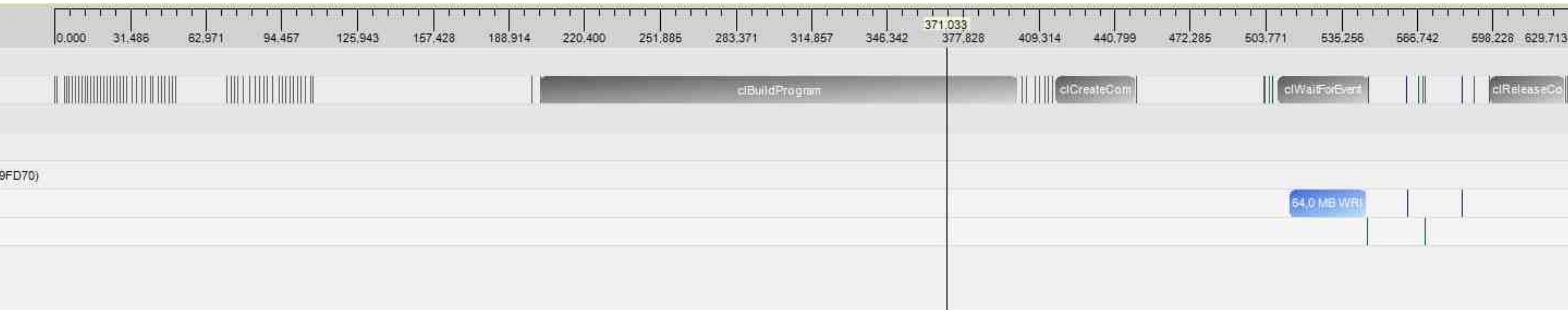
Symbol	Value
WGSIZE	256u
SUMONLOAD	8u
ELTYPE	uint4

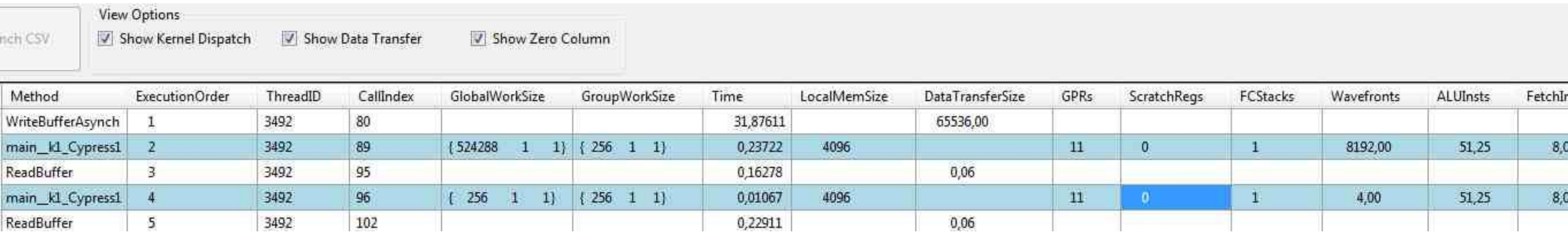
```
; -------- Disassembly ----------------
00 ALU: ADDR(32) CNT(17) KCACHE0(CB0:0-15
     0  w: LSHL        ____,   R0.x,   4
        t: ADD_INT     R0.w,   R1.x,   KC0
     1  y: ADD_INT     R0.y,   KC1[1].x,
        t: MULLO_INT   ____,   R1.x,   KC0
     2  y: ADD_INT     R10.y,  PV1.y,  4
     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

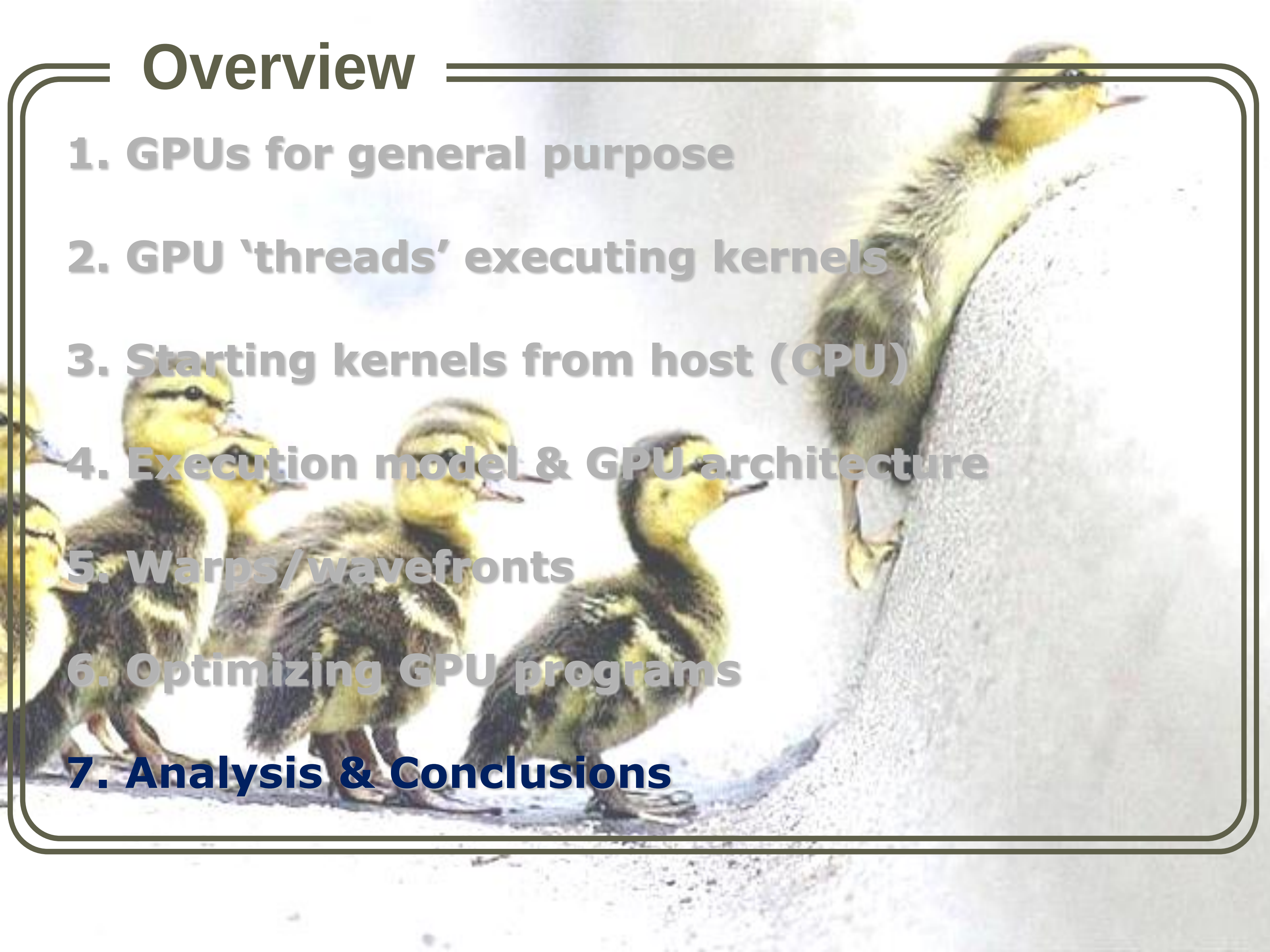




☑ Show Kernel Dispatch ☑ Show Data Transfer ☑ Show Zero Column

Method	ExecutionOrder	ThreadID	CallIndex	GlobalWorkSize	GroupWorkSize	Time	LocalMemSize	DataTransferSize	GPRs	ScratchRegs	FCStacks	Wavefronts	ALUInsts	FetchIr
WriteBufferAsynch	1	3492	80			31,87611		65536,00						
main__k1_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__k1_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						

Overview



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++
    - Larrabee: combining processing power of GPUs with programmability of x86 processors

Links in Scientific Study section

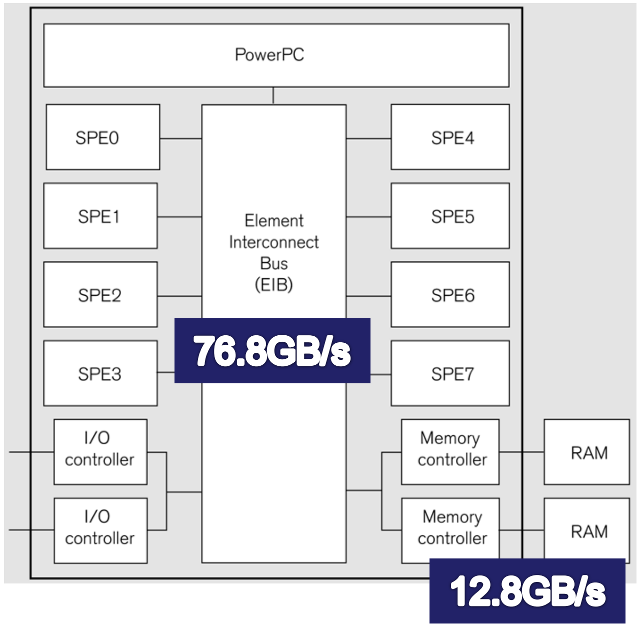
- CUDA/OpenCL promises an abstract, scalable hardware model, but will it remain true?

Link 1: white paper

Heterogeneous Chip Designs

- Augment standard CPU with attached processors performing the compute-intensive portions:
  - Graphics Processing Units (GPUs)
  - Field Programmable Gate Arrays (FPGAs)
  - Cell processors, designed for video games

Cell processor



- 8 Synergistic Processing Elements (SPEs)
  - 128-bit wide data paths
  - for vector instructions
  - 256K on-chip RAM
- No memory coherence
  - Performance and simplicity
  - Programmers should carefully manage data movement

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"
- They agree on:
  - 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 ;-)