ETRO VUB-DEPARTMENT OF ELECTRONICS AND INFORMATICS

Vrije Universiteit Brussel

GPGPU Training

Personal Super Computing Competence Centre PSC^3

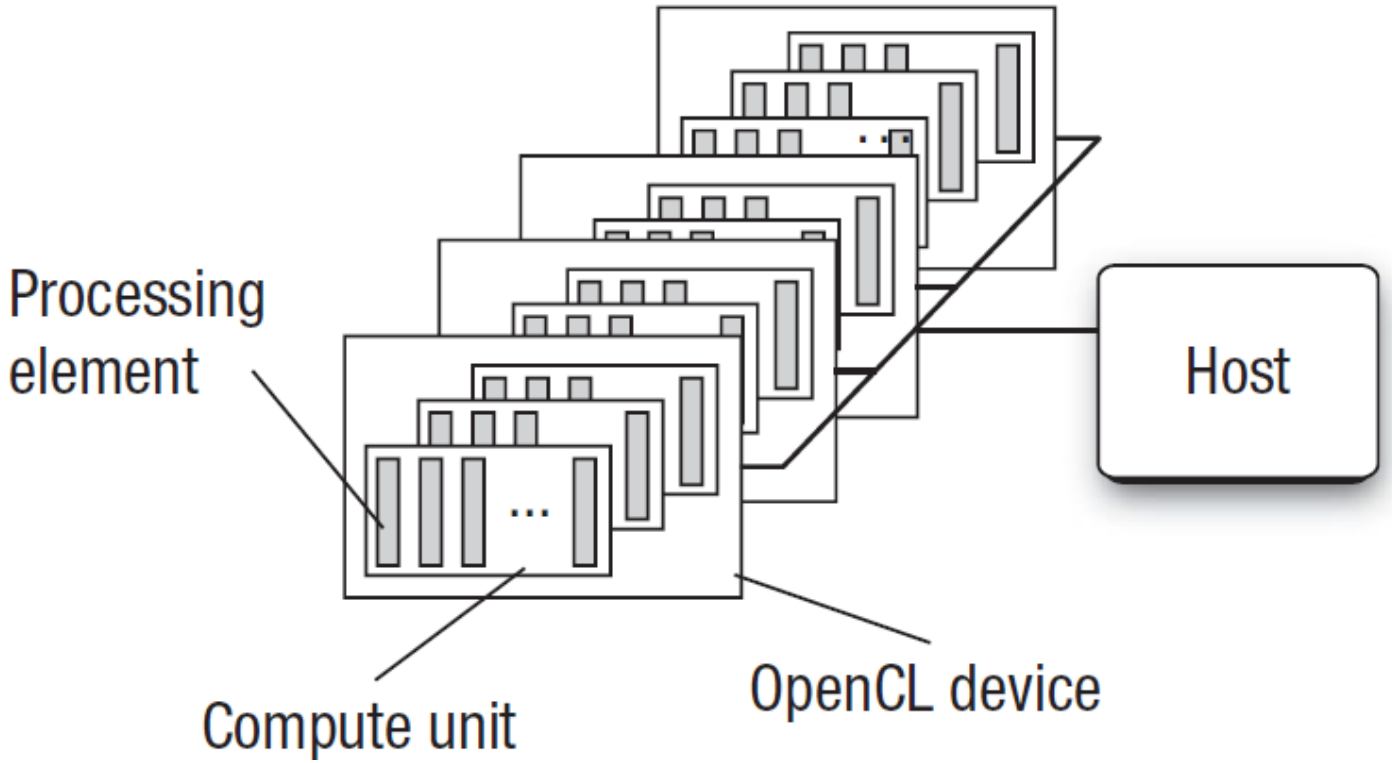
Jan G. Cornelis

Levels of Understanding

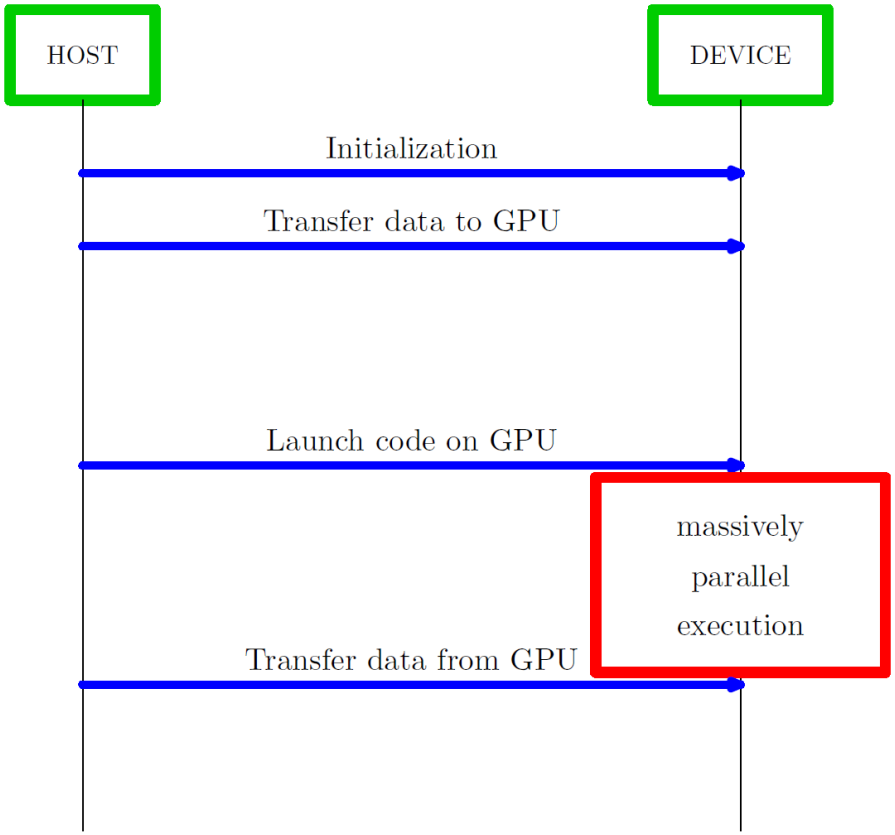
- Level 0
  - Host and device
- Level 1
  - Parallel execution on the device
- Level 2
  - Device model and work groups
- Level 3
  - Performance Considerations

Level 0
Host and Device

A Heterogeneous System
Host and Device



Typical Sequence of Events



OpenCL

OpenCL

- We need a way to
  - Modify our program to use accelerators
  - Specify the code that needs to run on the accelerators

- OpenCL
  - A host API
  - OpenCL C language
  - A model of
    - A heterogeneous system
    - An OpenCL device

- **https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/**

OpenCL Resources
A small sample

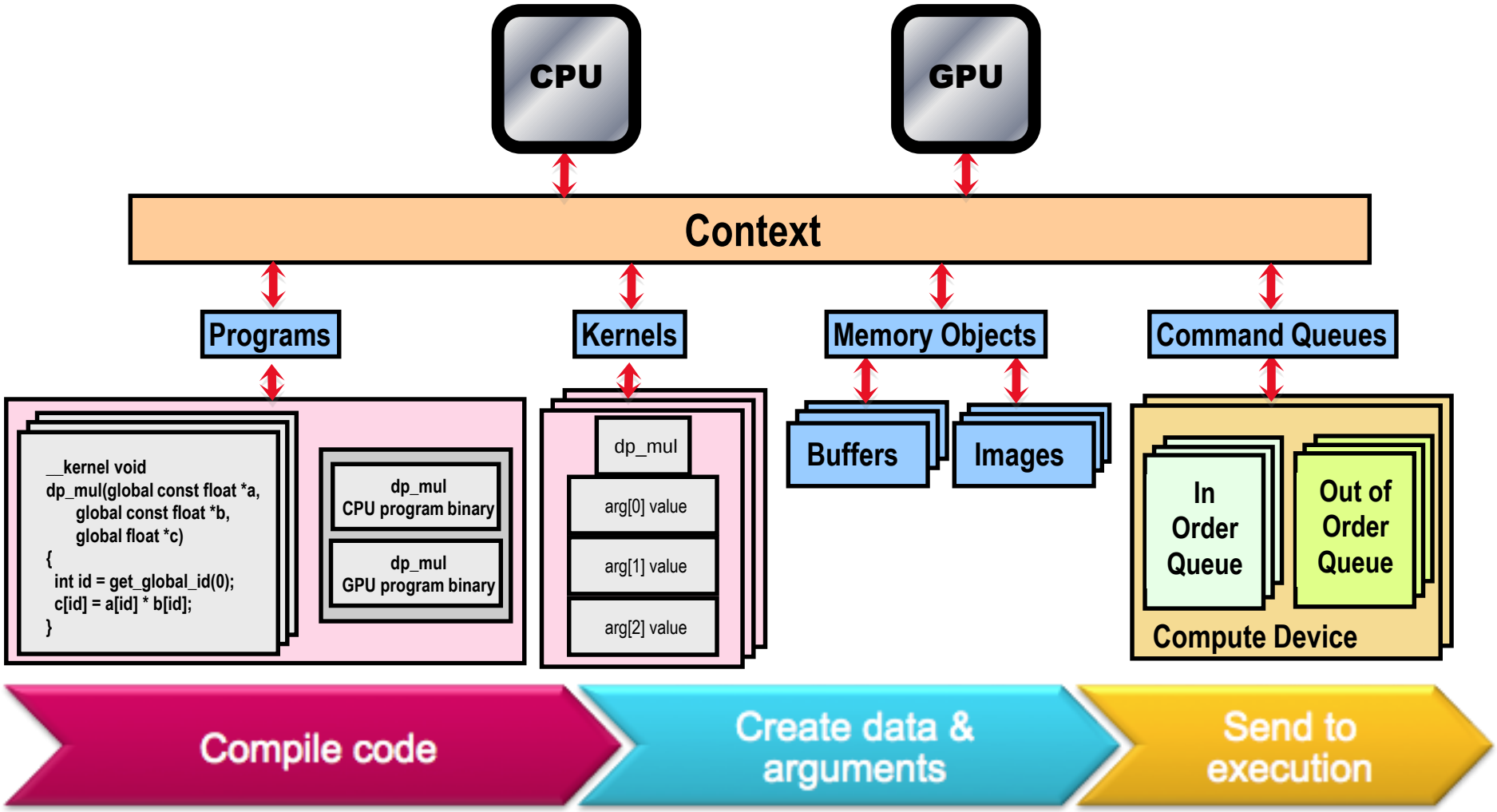
- www.khronos.org
- www.iwocl.org (*)
- www.streamcomputing.eu (*)
- developer.amd.com/tools-and-sdks/opencl-zone/
- www.eriksmistad.no/category/opencl/
- www.youtube.com
  - AJ Guillon

(*) These sites include references to books

HOST API
In this training

- We need only a little knowledge:
  1. Select the appropriate GPU.
  2. Allocate memory on the GPU.
  3. Transfer data between CPU and GPU.
  4. Compile and run code for/on the GPU.
- Understand what has to be modified.
- Seasoned programmers consult the manual pages
  https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml

The basic platform and runtime APIs in OpenCL (using C)

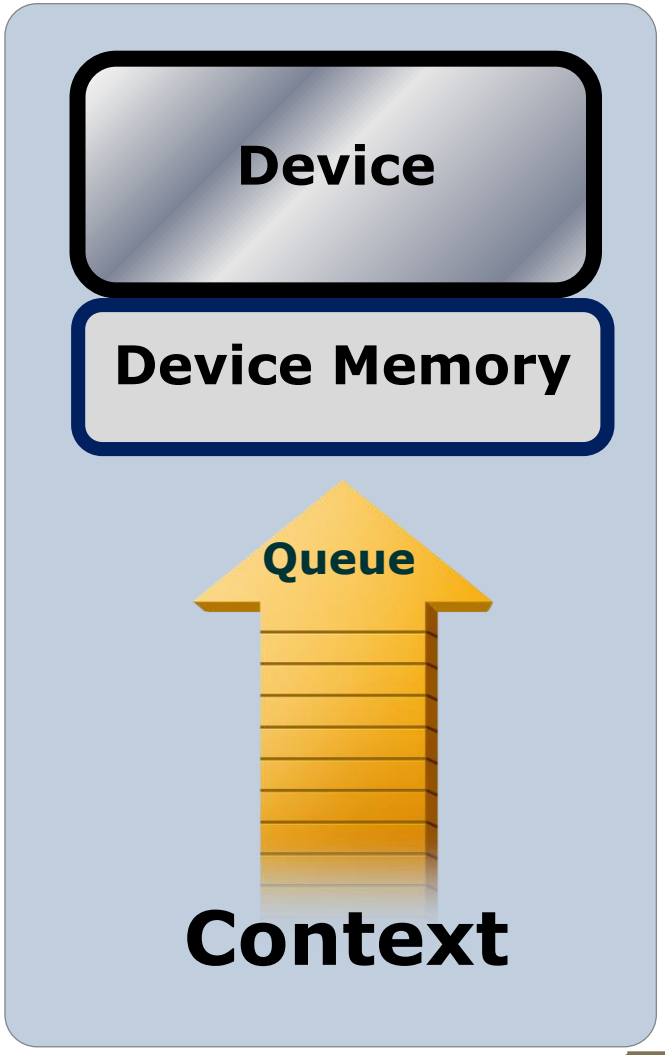


Host API Concepts

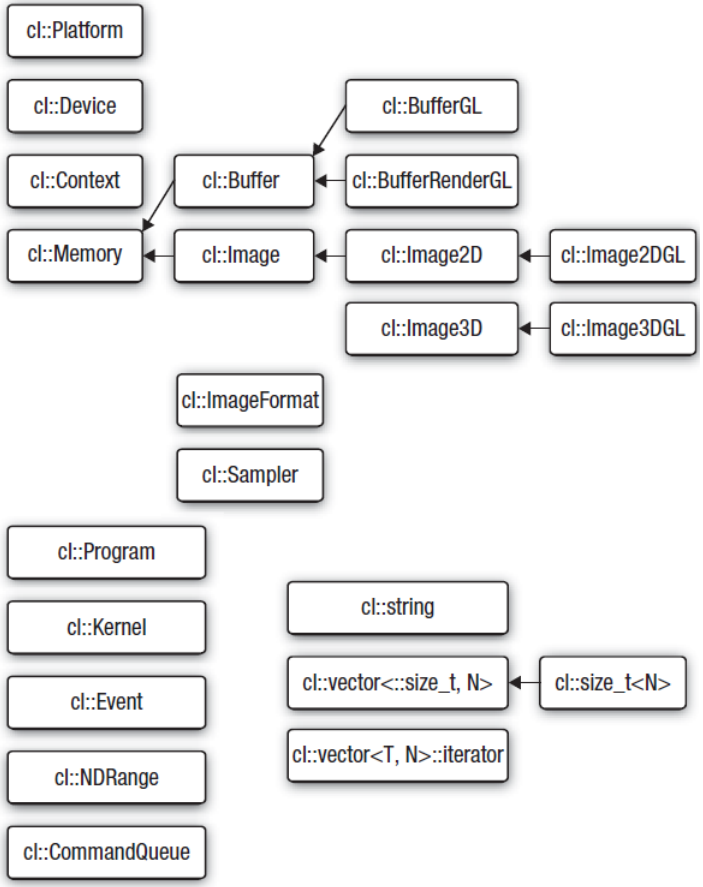
Platform	An OpenCL implementation e.g. AMD, Intel, NVIDIA, …
Device	An accelerator belonging to a platform
Context	A container object to deal with computation on the associated devices
Command Queue	Interface with a device. Used to send commands to the device
Program	A code container. Created from source or existing binaries
Kernel	A function to be run on a device
Buffer	A memory area on a device
NDRange	An execution configuration. See later

Context and Command-Queues

- ***Context***:
  - The environment within which kernels execute and in which synchronization and memory management is defined.
- The ***context*** includes:
  - One or more devices
  - Device memory
  - One or more command-queues
- All ***commands*** for a device (kernel execution, synchronization, and memory transfer operations) are submitted through a ***command-queue***.
- Each ***command-queue*** points to a single device within a context.



HOST API
C++ Wrapper



- Pros
  - Briefer
    - Exceptions instead of error handling
    - Certain methods equivalent to two C API function calls
  - Automatic cleanup
- Cons
  - May not be up to date
  - Man page mapping
    - Need some experience

OpenCL: What is needed?

- A device and a device driver that support OpenCL
- A static or dynamic library
- OpenCL header files
- Windows
  - A Visual Studio solution is provided
  - Includes library and headers
- Mac OSX
  - Supported out of the box. Compile with

    -framework OpenCL -DAPPLE
- Linux
  - https://wiki.tiker.net/OpenCLHowTo#How_to_set_up_OpenCL_in_Linux

Intermezzo

deviceQuery

Level 1

Parallel Execution on the Device

Parallel Execution on the Device
OpenCL Hello World

- A useless program?
  - Copy data from one buffer to another
- A very useful program!
  - Writing the first program is often a big hurdle
    - How to use the host API?
    - How to write the OpenCL C code
  - Helps to grasp the basic concepts
- Print-out of the code

The OpenCL Host API
OpenCL Hello World (1)

- Initializing OpenCL

```
std::vector<cl::Platform> platforms;
std::vector<cl::Device> devices;
cl::Platform::get(&platforms);
platforms[0].getDevices(CL_DEVICE_TYPE_GPU, &devices);

cl::Context context(devices);

cl::CommandQueue queue(context, devices[0], CL_QUEUE_PROFILING_ENABLE);
```

The OpenCL Host API
OpenCL Hello World (3)

- Allocating memory
- Transferring data

```
unsigned int size = data_count*sizeof(cl_float);

cl::Buffer source_buf(context, CL_MEM_READ_ONLY, size);
cl::Buffer dest_buf(context, CL_MEM_WRITE_ONLY, size);

queue.enqueueWriteBuffer(source_buf, CL_TRUE, 0, size, source);

// ...

queue.enqueueReadBuffer(dest_buf, CL_TRUE, 0, size, dest);
```

The OpenCL Host API
OpenCL Hello World (3)

- Compiling and executing code

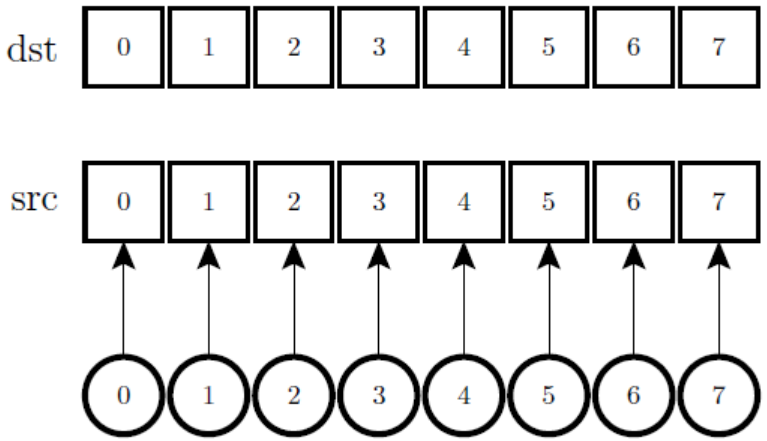
```
cl::Program program = jc::buildProgram(kernel_file, context, devices);
cl::Kernel kernel(program, kernel_name.c_str());

kernel.setArg<cl::Memory>(0, source_buf);
kernel.setArg<cl::Memory>(1, dest_buf);
kernel.setArg<cl_uint>(2, data_count);

cl_ulong t = jc::runAndTimeKernel(kernel, queue, cl::NDRange(data_count));
```

- *kernel_file*: name of text file containing OpenCL C code
- *kernel_name*: name of the kernel function

OpenCL C
OpenCL Hello World (4)

- *kernel_file* contains a function called *floatCopy*
- *floatCopy* specifies the work of a single work item

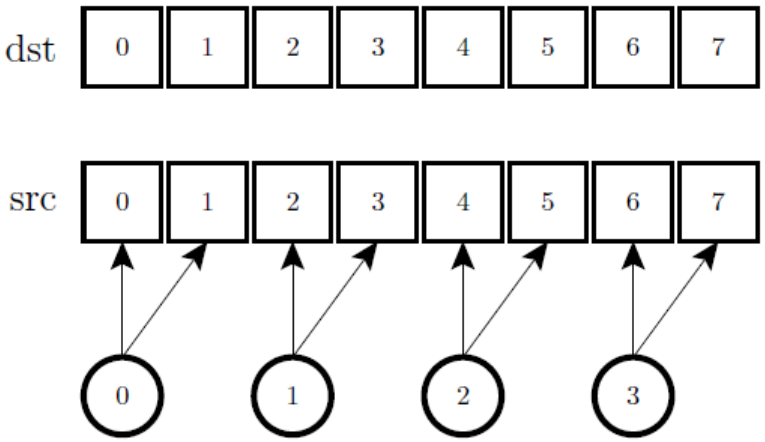
```
__kernel void floatCopy(
    __global float * source,
    __global float * dest,
    unsigned int     data_size
    )
{
    size_t index = get_global_id(0);

    if (index >= data_size) {
        return;
    }

    dest[index] = source[index];

    return;
}
```



OpenCL C
OpenCL Hello World (5)

- The programmer specifies the number of work items
- Enough work items to handle all data items

```
__kernel void floatCopy(
    __global float * source,
    __global float * dest,
    unsigned int     data_size
    )
{
    size_t index1 = 2*get_global_id(0);
    size_t index2 = index1 + 1;

    dest[index1] = source[index1];
    if (index2 < data_size) {
        dest[index2] = source[index1];
    }

    return;
}
```



Beehive Metaphor

Memory – Data – Work items

Beehive – Honey – Bees

OpenCL C C?
A language based on C99

Extensions

- Function qualifiers
  `__kernel`
- Memory qualifiers
  `__global, __constant, __local, __private`
- Workspace query functions
  `get_global_id(dimidx), ...`
- Access qualifiers
  `__read_only, __write_only`

Limitations

- No recursion
- No function pointers
- No dynamic memory

Exercise

sumInts

- Implement the sum of lists
  - Assume three lists A, B and C
  - Element i of C:
    - $C_i = A_i + B_i$;
- Extension:
  - One work item processes more than one data item

Intermezzo

performance

Performance?
Informal Definition

- Performance of a program?
  - Linked to the run time
- Performance of a program is a function of
  - Hardware
  - Data on which the program is run
- Two implementations of the same algorithm: A and B

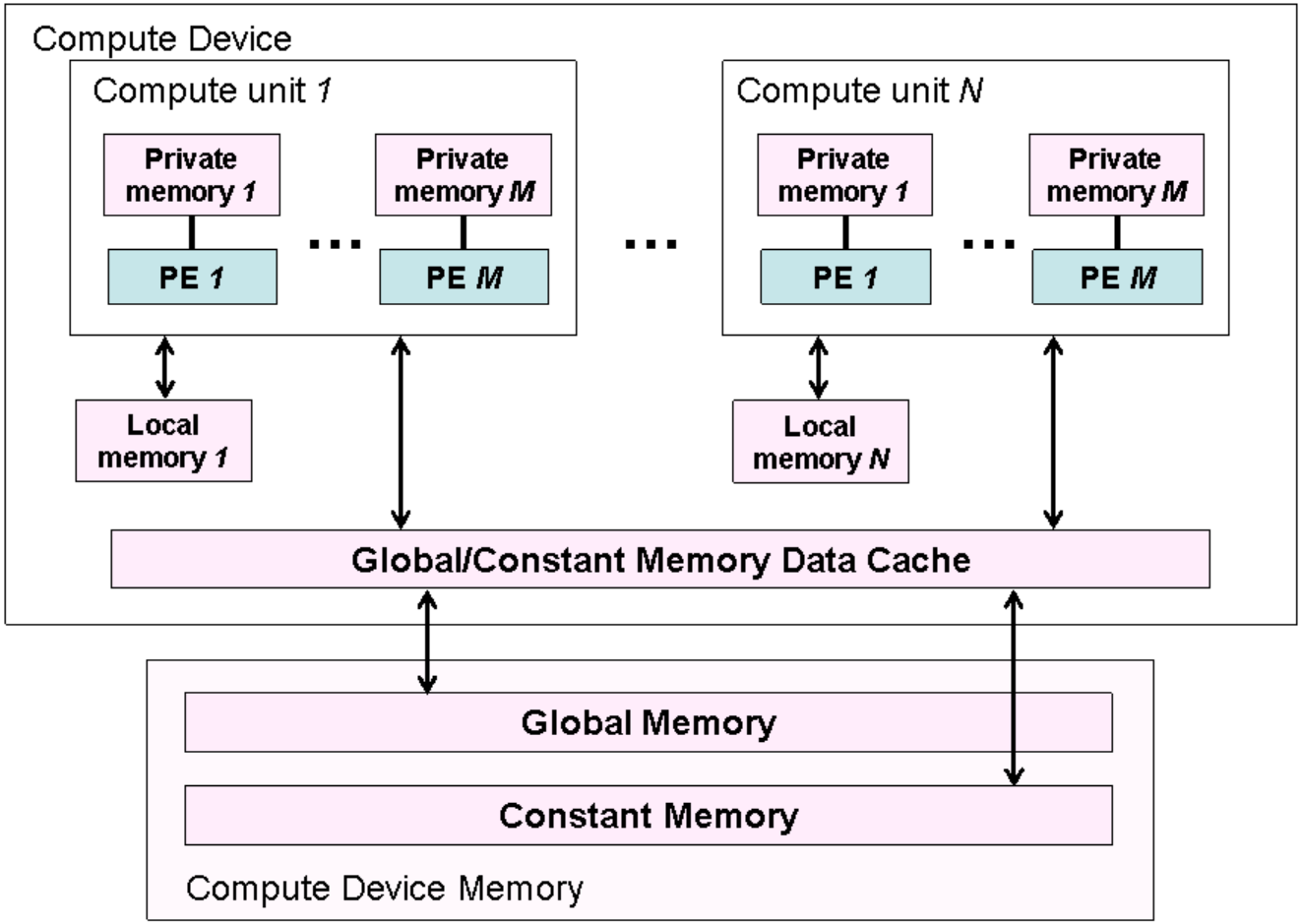
$$P(A, HW, D) > P(B, HW, D) \Leftrightarrow t(A, HW, D) < t(B, HW, D)$$

Performance?
Quantification

- Operations performed per second
  - For computationally bound code
- Bytes accessed per second
  - For memory bound code
- Compare to the platform peak performance
  - Memory bandwidth
  - Gflops/s, Ops/s
- More on day 3

Level 2
Device Model and Work Groups

OpenCL Device Model
How can we exploit this?



Work groups

- Work items are divided in work groups
- A work group is executed on one compute unit
  - From start to end
- Work items can share local memory
  - Kind of explicit cache
- Within a work group synchronization is possible
  - With the barrier statement.
- Work group size is determined by the programmer
  - One size for all work groups

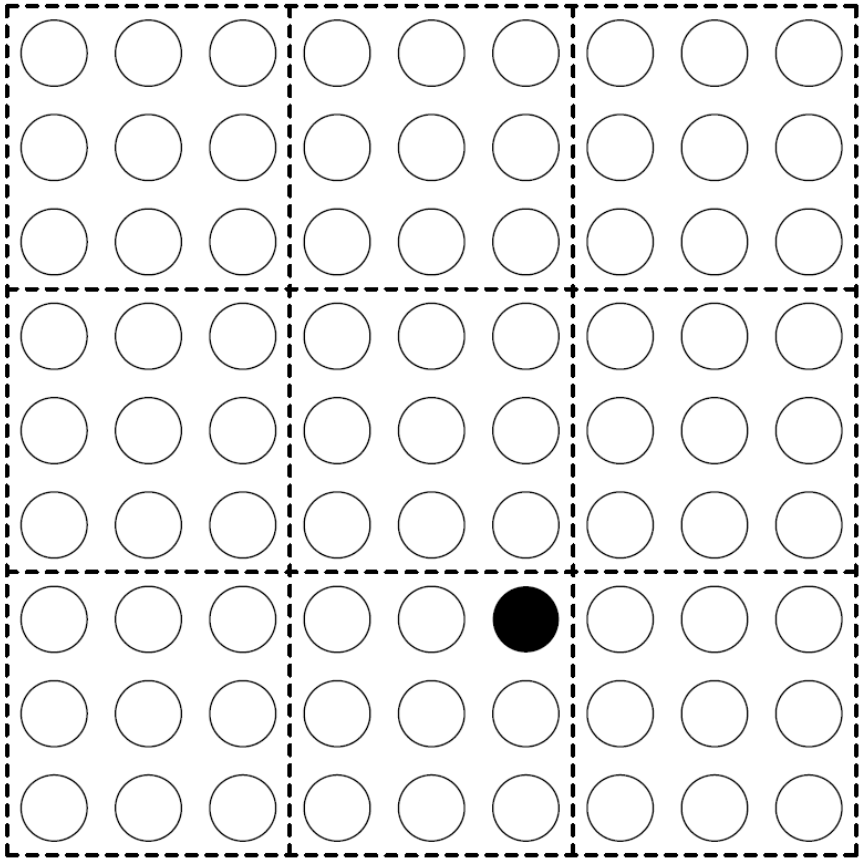
OpenCL Work Space
Terminology and query functions

- N-dimensional range
  - index space
  - 1-, 2- or 3-dimensional
- Global NDRange: configuration of ALL work items
- Local NDRange: configuration of a work group
- Note:
  - Global and Local ranges must have the same number of dimensions!
  - Work group size in a certain dimension must be a whole divisor of the global size in this direction

- Query functions

```
get_global_id(dimidx)
get_global_size(dimidx)
get_group_id(dimidx)
get_local_id(dimidx)
get_local_size(dimidx)
get_num_groups(dimidx)
get_work_dim()
```

OpenCL Work Space
Quick test



```
get_global_id(0) = _______
get_global_id(1) = _______
get_global_size(0) = _______
get_global_size(1) = _______
get_group_id(0) = _______
get_group_id(1) = _______
get_local_id(0) = _______
get_local_id(1) = _______
get_local_size(0) = _______
get_local_size(1) = _______
get_num_groups(0) = _______
get_num_groups(1) = _______
get_work_dim() = _______
```

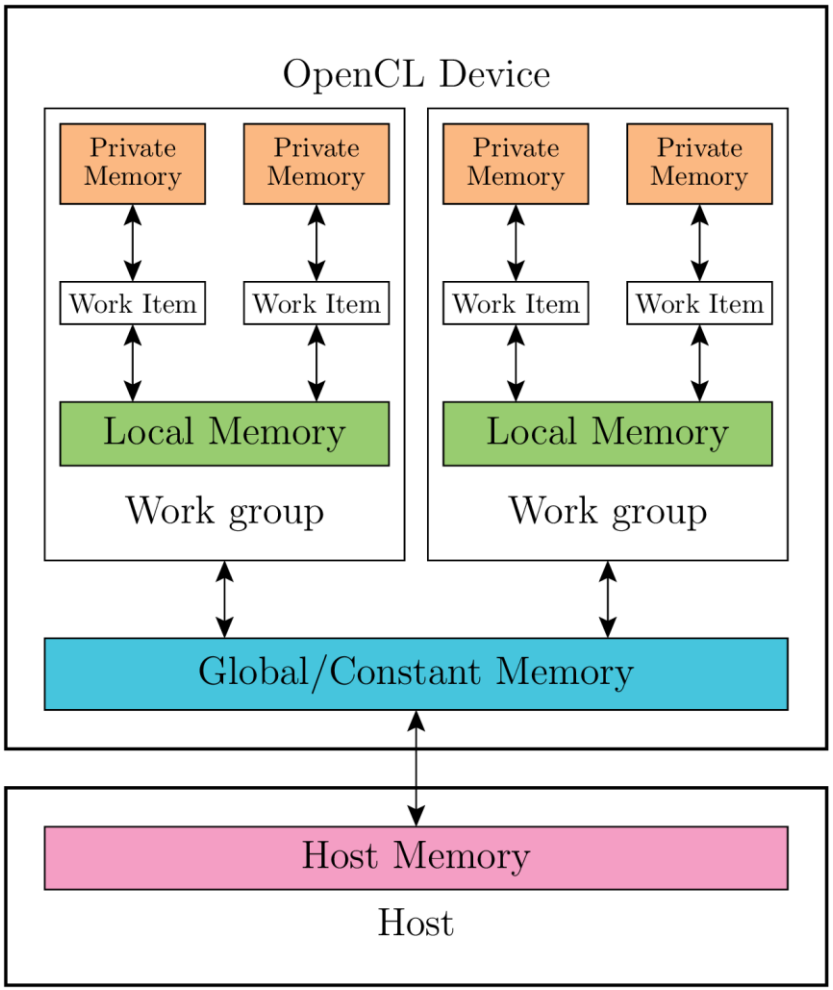
OpenCL Work Space – Exercise
queryWorkSpace

- Write the result of the query functions to global memory
- Visualize the resulting matrix from the host
  - Tip: for a readable result use small matrices and small workgroup sizes
    - E.g. 16x16 matrix 4x4 work group

```
Usage: queryWorkSpace.exe <kernel_file> <kernel_name> <data_width> <data_height> <wg_width> <wg_height>

$ ./queryWorkSpace.exe kernels.ocl queryWorkSpace 16 16 2 4
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

Which query function did I call?

OpenCL Memory Model
Explicit Memory Hierarchy



- In your kernel code:

```
__kernel void similarity_constant_local
(
    __global float    * tags_min,          //  0
    __global float    * tags_max,          //  1
    __constant float  * query_min,         //  2
    __constant float  * query_max,         //  3
    __global float    * shifted_weights,   //  4
    __global float    * scores,            //  5
    __global uint     * indices,           //  6
    __global int      * offsets,           //  7
     unsigned int   tags_size,             //  8
     unsigned int   num_windows,           //  9
     unsigned int   index_resolution,      // 10
    __local float     * l_scores,          // 11
    __local uint      * l_indices,         // 12
    __local float     * l_tags_min,        // 13
    __local float     * l_tags_max,        // 14
    unsigned int        tag_count          // 15
)
{
    // kernel body omitted
}
```

OpenCL Memory Model
using local memory

In the kernel body

```
#define N 256

__kernel void similarity_constant_local
(
    __global float * in,
    __global float * out
    unsigned int   size
)
{
    unsigned int index = get_global_id(0);
    __local float shared[N];
    // populate
    shared[get_local_id(0)] =
        index < size ? In[index] : 0;
    barrier(CLK_LOCAL_MEM_FENCE);
    // use local memory
    // …
}
```

As a kernel argument

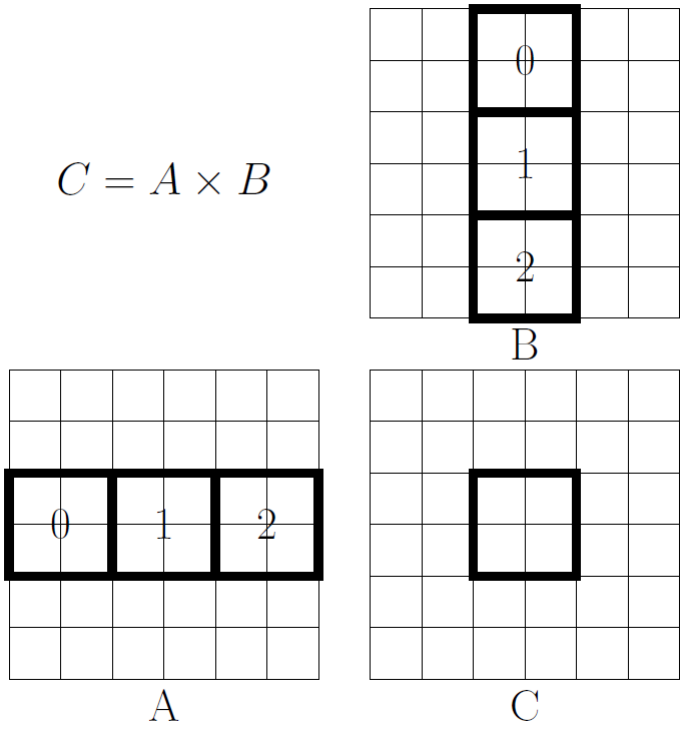
```
__kernel void similarity_constant_local
(
    __global float * in,
    __global float * out,
    __local  float * shared,
    unsigned int   size
)
{
    unsigned int index = get_global_id(0);

    // populate
    shared[get_local_id(0)] =
        index < size ? In[index] : 0;
    barrier(CLK_LOCAL_MEM_FENCE);
    // use local memory
    // …
}
```

```
kernel.setArg<cl::LocalSpaceArg>(2, cl::__local(N)); // N can be variable
```

OpenCL Memory Model
using local memory – example

Matrix Multiplication

$$C = A \times B$$



Device code

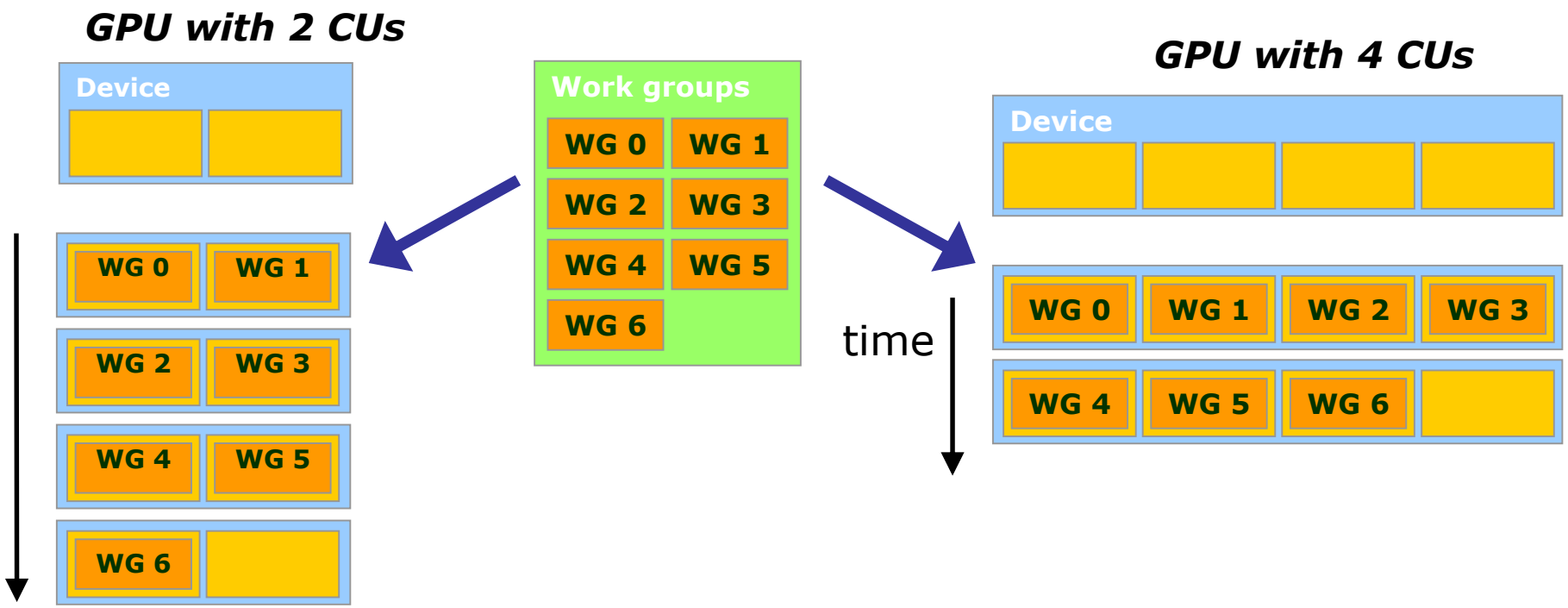
```
__kernel void mul(__global int *A, __global int *B, __global int *C, int
size)
{
    __local int sharedA[16][16];
    __local int sharedB[16][16];

    int sum    = 0;
    int aStart = get_global_id(1)*size + get_local_id(0);
    int aEnd   = aStart + size;
    int bStart = get_local_id(1)*size + get_global_id(0);
    int aStep  = 16;      // move 16 colums
    int bStep  = 16*size; // move 16 rows

    for (int a = aStart, b = bStart; a < aEnd; a += aStep, b += bStep){
        sharedA[get_local_id(1)][get_local_id(0)] = A[a];
        sharedB[get_local_id(1)][get_local_id(0)] = B[b];
        barrier(CLK_LOCAL_MEM_FENCE);
        #pragma unroll
        for (int j = 0; j < 16; ++j)
            sum += sharedA[get_local_id(0)][j] *
                   sharedB[j][get_local_id(0)];
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    C[y*size + x] = sum;
}
```

OpenCL Execution Model

- Execution of N work groups of m work items
- Work groups are assigned to compute units
  - A work group stays there until it completes
- Compute units may execute multiple work groups concurrently
  - See later
- Work groups not yet assigned to a compute unit must wait
- The order in which work groups execute is non-deterministic

- Consequences
  - There can be no interaction between work groups
  - OpenCL code scales inherently

Inherent Scaling



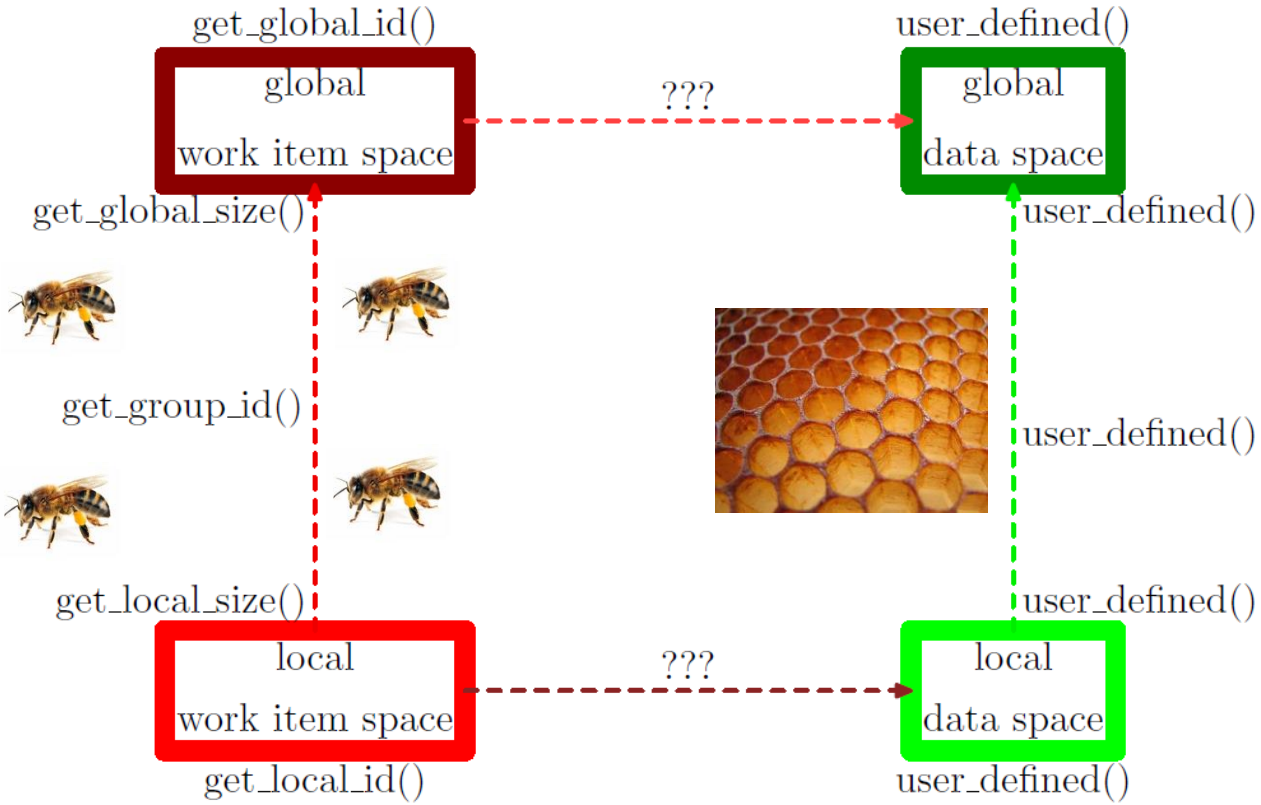
Advanced OpenCL

- OpenCL is a large topic.
- You cannot know everything in 3 days:
  - Images and OpenGL interoperability
  - Runing code on multiple devices
  - Atomic operations
  - Mapped memory
  - Streaming
  - Events
  - …
- Extend your knowledge as needed.
- But don’t try to run before you can walk!

Runtime math library

- Two ways to compute standard mathematical functions
  - func(): slow but precise
  - native_func(): less precise but fast
- For example
  - cos(), native_cos()
  - sqrt(), native_sqrt()
- Special hardware for native functions

The Main Challenge of OpenCL



Exercise: Matrix Vector Operation

matrixVector

- Matrix A mxn
- Vector B n

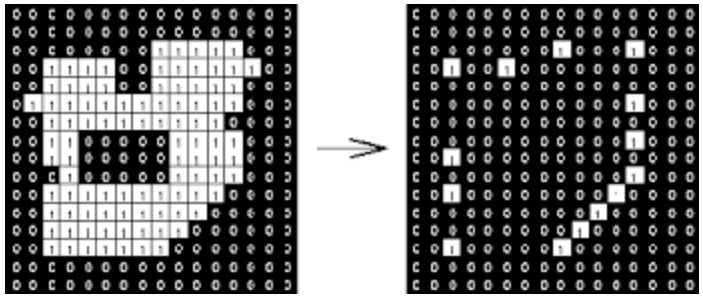
- Computation?
  - Repeat N times:
    - A[i,j] = A[i,j] + A[i,j]*B[j]

- Observe
  - Data throughput in function of N
  - Computational throughput in function of N

Exercise: Erosion

listErosion – matrixErosion

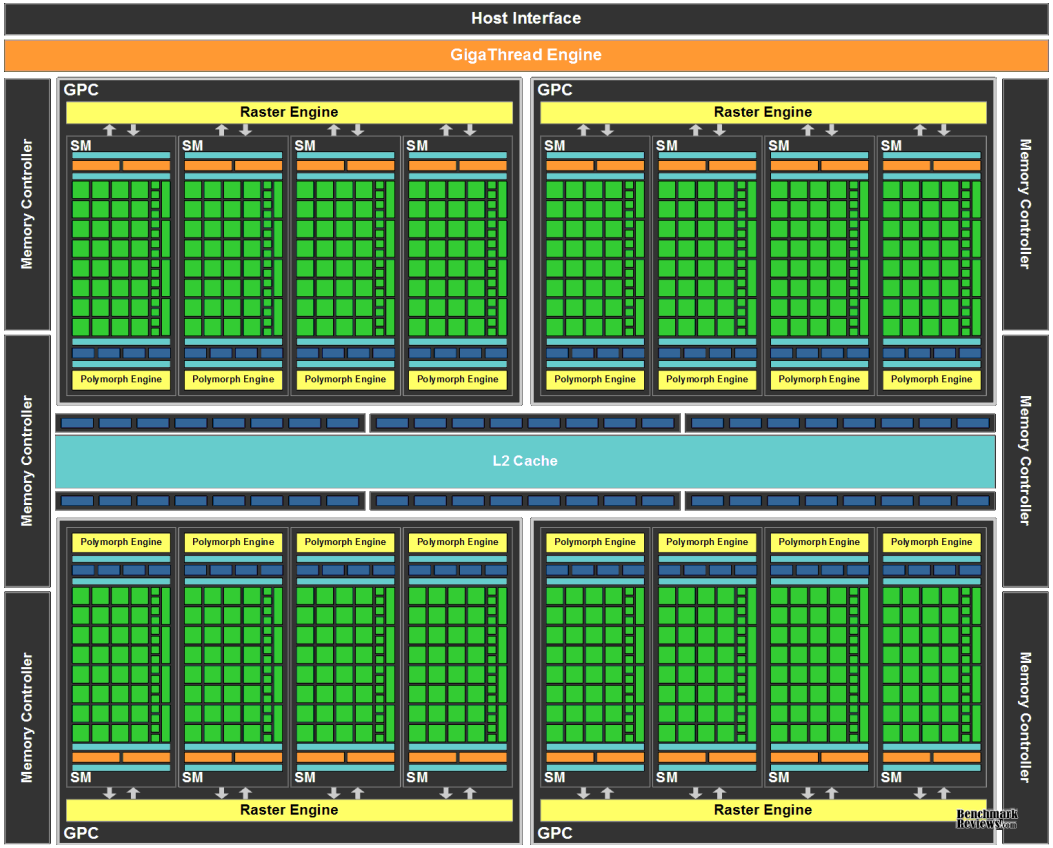
- Typical operation in image processing
- Given an input pixel the value of the corresponding output pixel is the minimum of values of pixels under a mask centered on the input pixel
- Example Erosion with a 3x3 mask on a binary image:



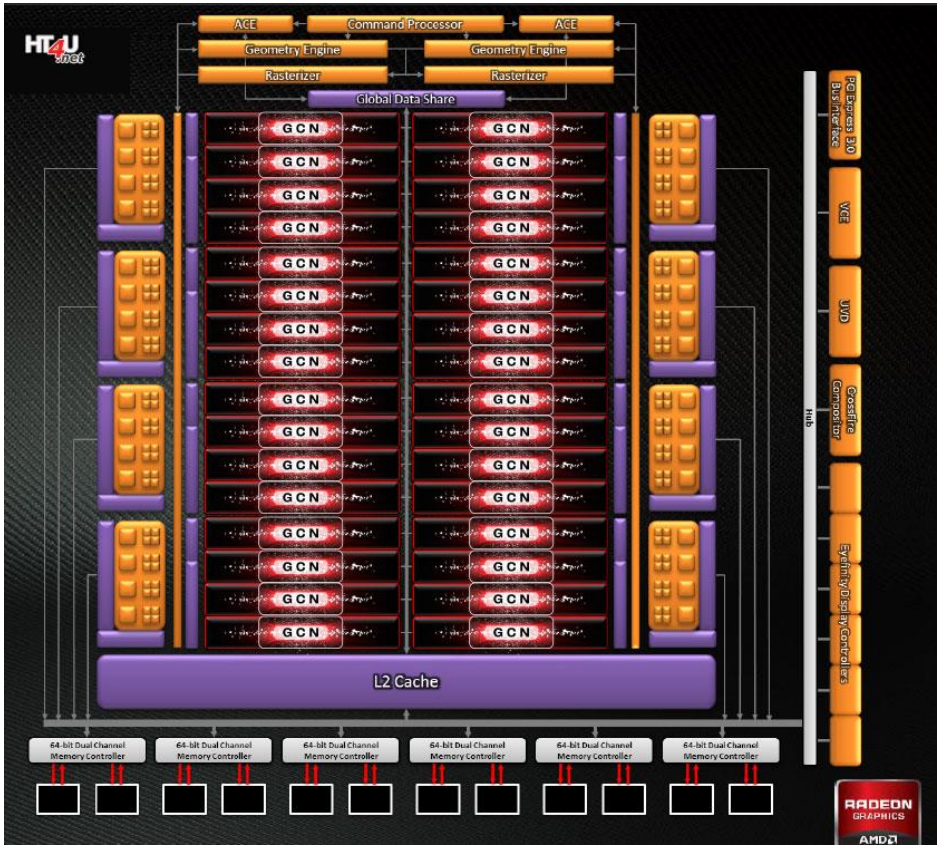
- Implement erosion for one-dimensional data for a parameterizable mask width
  1. Doing everything in global memory
  2. Using local memory
- Try two-dimensional erosion

Level 3
Performance Considerations

NVIDIA GPU
Fermi Architecture



AMD GPU
GCN Architecture



NVIDIA Compute Unit
a.k.a. Streaming Multiprocessor

- SM:
  - 32 cores
    - Processing elements
  - 4 special function units
  - 64 KB local memory/cache
    - __local memory
  - 32K 32 bit registers
    - __private memory
- SMX:
  - 192 cores
  - 32 special function units
  - 64 KB local memory/cache
  - 64K 32 bit registers

Execution of Work groups

- Work group executed on a compute unit
- Groups of 32/64 work items operate together
  - NVIDIA: warp consists of 32 work items
  - AMD: wavefront consists of 64 work items
- It is necessary to think in terms of warps or wavefronts to obtain optimal performance

Occupancy

- Occupancy $= \frac{\#concurrent\ warps\ on\ a\ Compute\ Unit}{maximum\ \#concurrent\ warps\ on\ a\ Compute\ Unit}$
- Limited resources may limit the occupancy:
  - Registers needed per work group
  - Local memory needed per work group
  - Maximum number of concurrent work groups
- The most constrained resource determines the occupancy
- A higher occupancy means more work can be scheduled

Pipeline model for performance analysis

Our research

- To understand several aspects influencing the performance, one should understand the behavior of *pipelined processors*

- Our performance analysis is based on the simulation of a dual pipeline model
  - The GPU is modeled with 2 pipelines: one for the computational units, another for the memory units
  - *It does not intend to reflect a hardware accurate model nor a cycle accurate simulation*

http://parallel.vub.ac.be/pipeline/

Single Pipeline

- One warp and only dependent instructions

Model '3 computations (all dependent)'

  - Completion latency (Λ) determines performance

    = length of the pipeline

- Several warps or independent instructions:

Increase #warps/#WG and/or #concurrent work groups

Model '3 computations (two independent instructions)'

  - latency hiding
  - Issue latency (λ) determines performance

    = 1 cycle for simple pipeline

    Determines the peak performance:

Parameters #multiprocessors and #work items in 1 warp/wavefront

Dual Pipeline

- Computation and communication (Memory access)
- Memory access is modeled as a single pipeline

Model '3 computations and communications (all dependent)'

- $\Lambda_{mem} >> \Lambda_{comp}$ and $\lambda_{mem} >> \lambda_{comp}$
  ➔ More concurrency needed for peak performance
- Communication vs memory bound

Models 'balanced graph', 'communication-bound graph' and 'computation-bound graph'

- The cost of barrier synchronization

Compare models with and without barrier

Real GPU is not a simple pipeline

- NVIDIA generations
  - Tesla: 8 cores ➔ 1 warp every 4 clock cycles
  - Fermi: 32 cores ➔ 1 warp every clock cycle
  - Kepler: 192 cores ➔ 6 warps every clock cycle
  - Maxwell: 128 cores ➔ 4 warps every clock cycle
- Pipeline model
  - one computation pipeline λ_{comp} = f(generation)
    - λ_{comp}(Tesla) = 4 clock cycles
    - λ_{comp}(Fermi) = 1 clock cycles
    - λ_{comp}(Kepler) = 1/6 clock cycles
    - λ_{comp}(Maxwell) = 1/4 clock cycles
  - One communication pipeline
    - Latencies depend on type of memory request
      - Longer for non-ideal memory access

Programming for Performance
Minimizing the overall run time

- Minimize idle time
  - Maximize parallelism
  - Minimize dependencies
  - Minimize synchronization
- Minimize software and hardware overheads
  - Memory access
    - Data placement
    - Global memory access patterns
    - Local memory access patterns
  - Computation
    - Minimize excess computations
    - Minimize branching
- **Remembering data access is slow and computation fast**

Maximize Parallelism
On the device

- Number of work groups:
  - A multiple of the number of compute units
  - A multiple of the number of compute units times the occupancy in work group count
  - In practice: a very large number

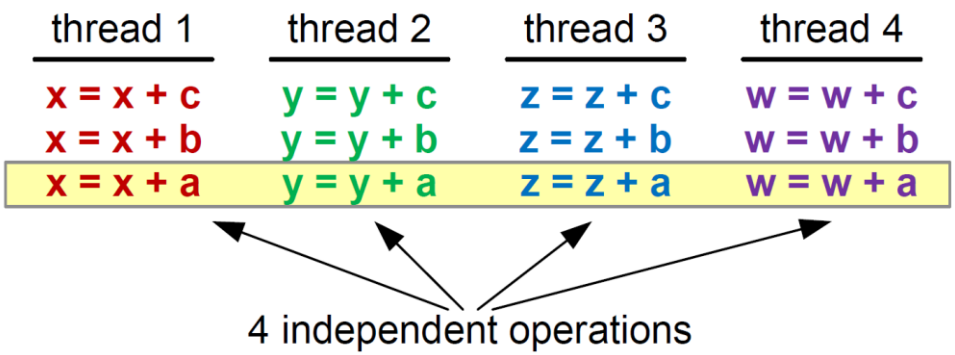
- Work group size:
  - Not too large: could limit occupancy
  - A multiple of the warp/wavefront size
  - In practice: 256 is a good number

Maximize Parallelism
On the compute unit

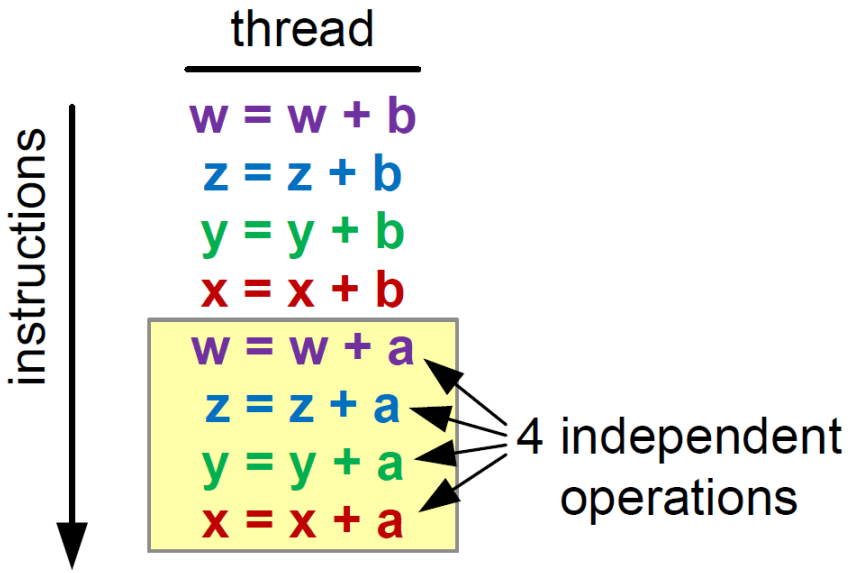
- Maximize occupancy
  - Scheduler has more choice
- Instruction Level Parallelism can help
  - Independent instructions within one warp
  - Can be executed concurrently
- Data Level Parallelism can help
  - Independent memory requests for one warp
  - Can be serviced concurrently
- Peak performance is reached for fewer warps if the ILP and MLP are increased

Minimize Dependencies
ILP and MLP

Thread-Level Parallelism
- Independent threads



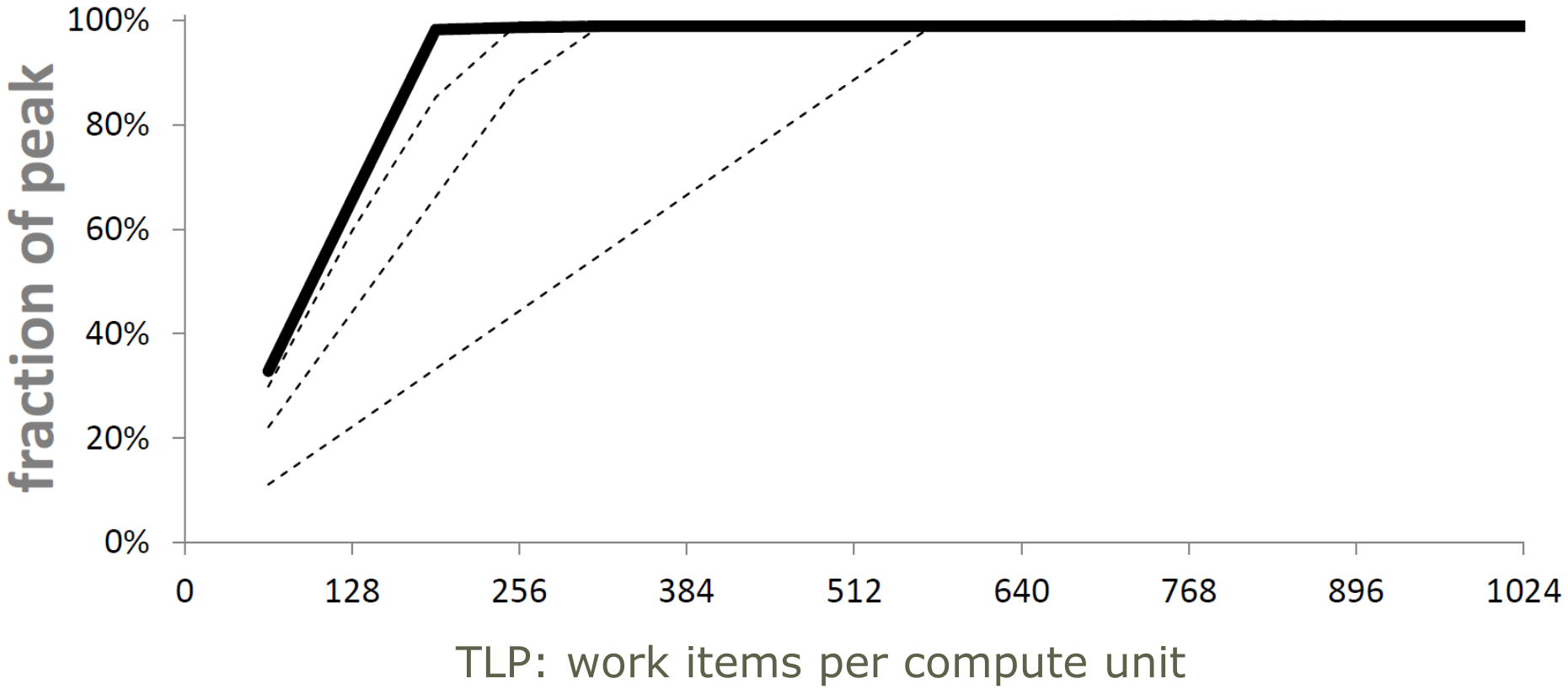
Instruction-Level Parallelism
- Independent instructions



Memory-Level Parallelism
- One thread reading / writing 2, 4, 8, 16, … floating point values

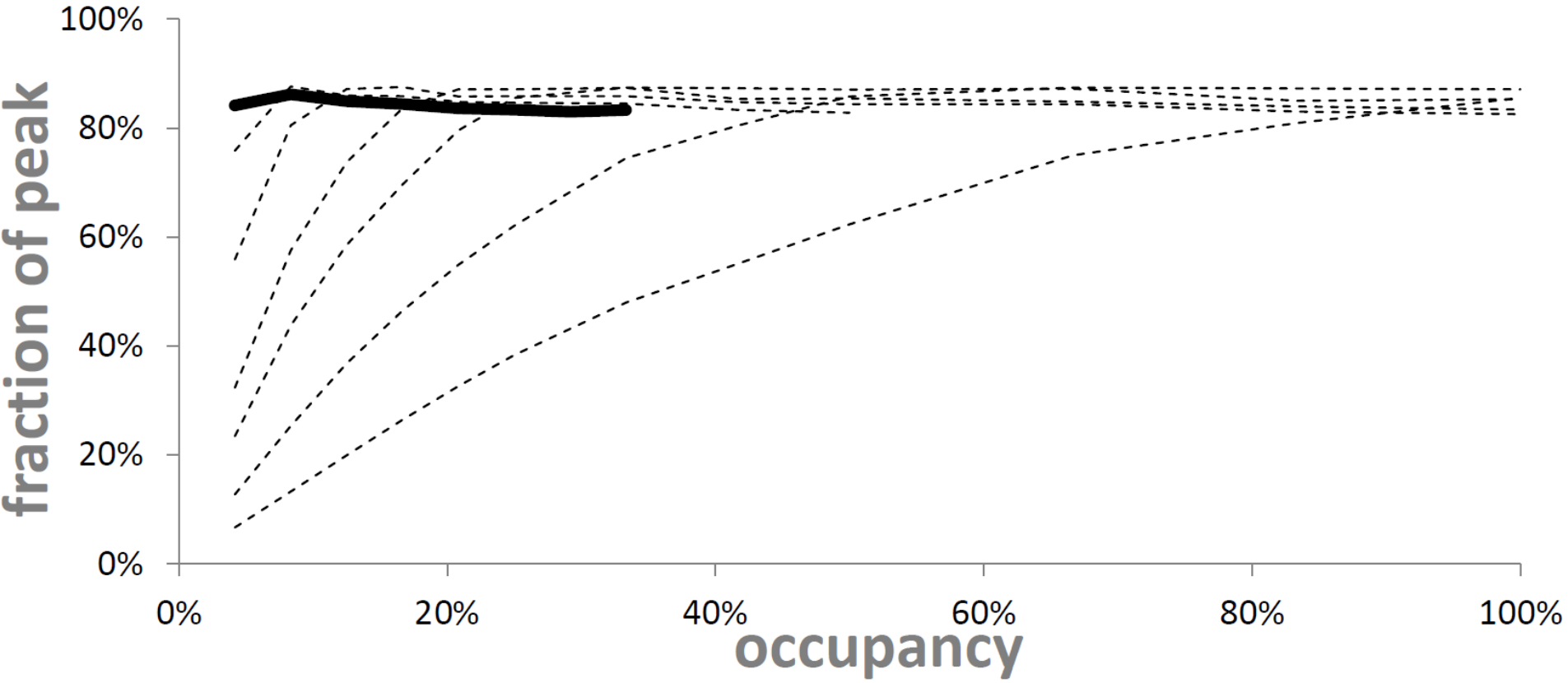
Computational Performance
A function of TLP and ILP

- ILP = 1, 2, 3, 4



Memory throughput
A function of TLP and MLP

- MLP: 1 float, 2 float, 4 float, 8 float, 8 float2, 8 float4 and 14 float4
- TLP: occupancy



Memory Access Overhead
Data Placement (1)

- Data placement is crucial for performance
- Use the memory hierarchy:
  - Global memory
    - Share data between GPU and CPU
    - Large latency and low throughput
      - ➔ Access should be minimized
    - Cached in L2-cache
  - Constant memory
    - Share read-only data between GPU and CPU
    - Is cached in L1 cache
    - Limited size. Typically 64 KB
    - Prefer it to local memory for small read-only data

Memory Access Overhead
Data Placement (2)

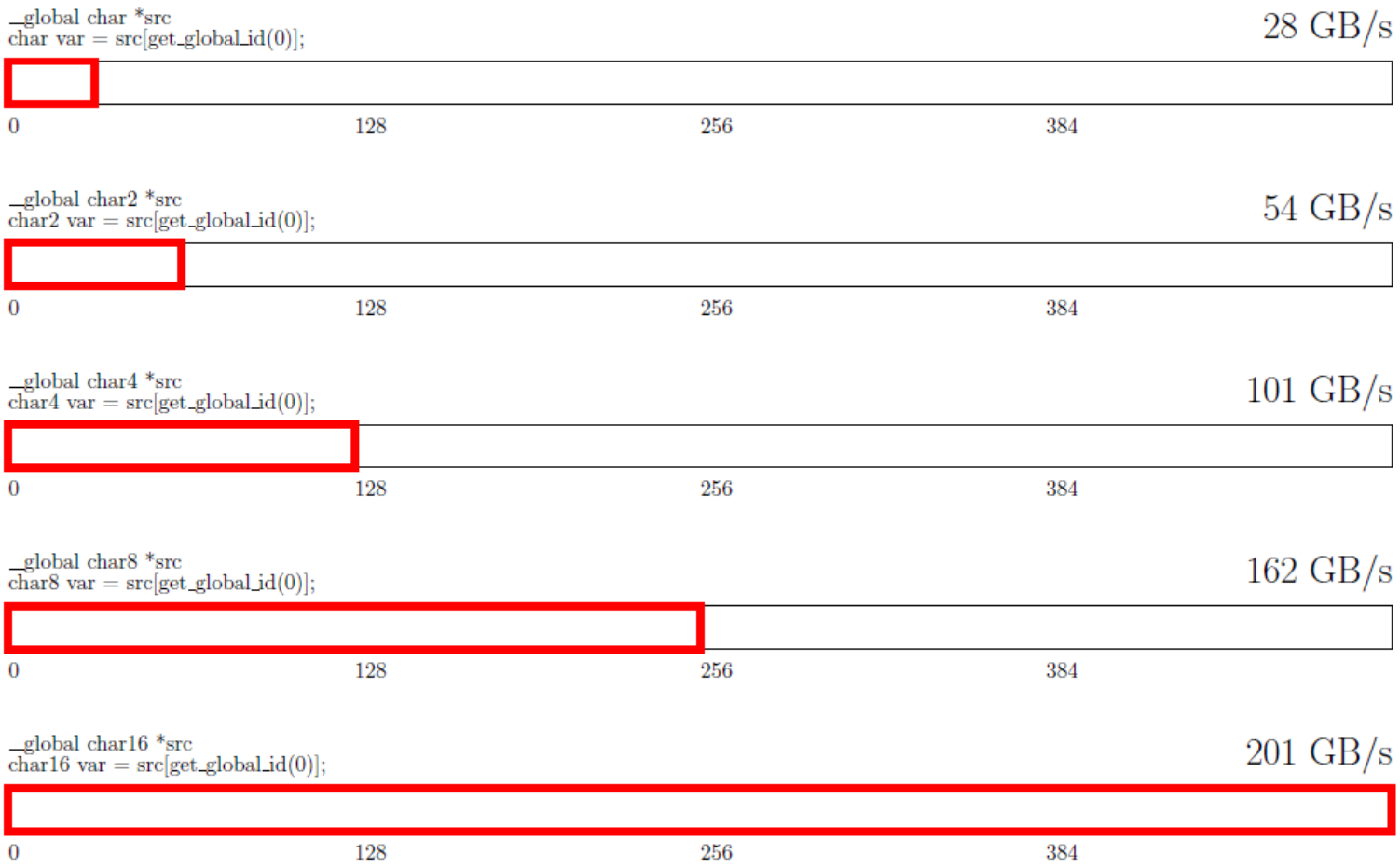
- Texture memory
  - Like global memory but 2D and 3D caching
  - Discussion on images
- Local memory
  - Share data within a work group
  - Use it if the same data is used by multiple work items in the same work group
- Private memory (registers)
  - Lowest latency highest throughput
  - ! Private arrays will be stored in global memory
    - Cached in L1-cache

Memory Access Overhead
Global Memory Access (1)

- Global memory is organized in segments
- Memory requests of warp are handled together
- Ideal situation:
  - The number of bytes that need to be accessed to satisfy a warp memory request is equal to the number of bytes actually needed by the warp for the given request
- A few examples will clarify this

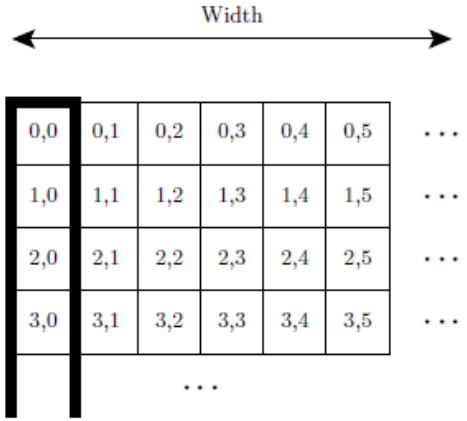
Global Memory Access
Impact of size of accessed elements

- Multiple copy kernels on an AMD Radeon HD 7950

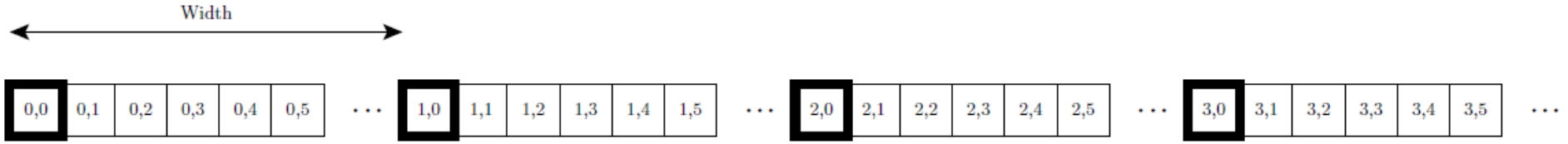


Global Memory Access
Impact of strided access

- 2-D and 3-D data stored in flat memory space
  - Strided access is not a good idea e.g. access columns



Quadro K620	AMD HD 7950
Aligned: 26 GB/s	Aligned: 170 GB/s
Strided: 7 GB/s	Strided: 4 GB/s



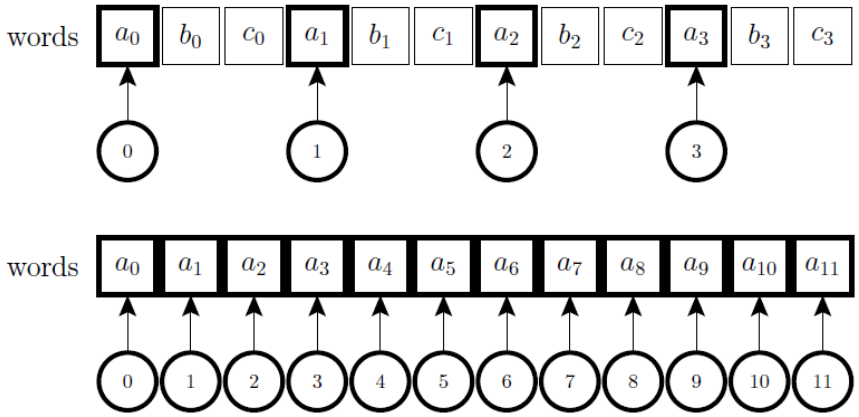
Memory Access Overhead
Global Memory Access (3)

- Array of struct vs struct of arrays

```
typedef struct {
          float a;
          float b;
          float c;
} triplet_t;

__kernel void aos(__global triplet_t *triplets,
                  // ... )
{
    float a = triplets[get_global_id(0)].a;
    // ...
}

__kernel void soa(__global float *as,
                  __global float *bs,
                  __global float *cs, // ... )
{
    float a = as[get_global_id(0)];
    // ...
}
```

AOS introduces strides

If elements are visited at different moments



SOA removes strides

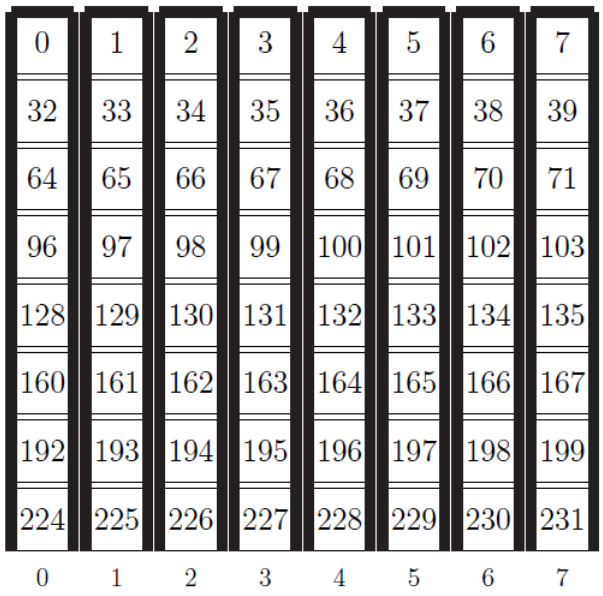
Memory Access Overhead
Local Memory access (1)

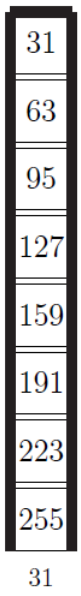
- Local memory is organized in banks
- Each bank can service one address per cycle
- Simultaneous access by work items of same warp of the same bank is a **bank conflict**
  - Accesses are serialized
  - Maximum cost = maximum bank conflict degree
  - No bank conflicts when
    - All work items of warp access another bank
    - All work items of warp read the same address

- AMD avoids bank conflicts in hardware

Memory Access Overhead
Local Memory access (2)

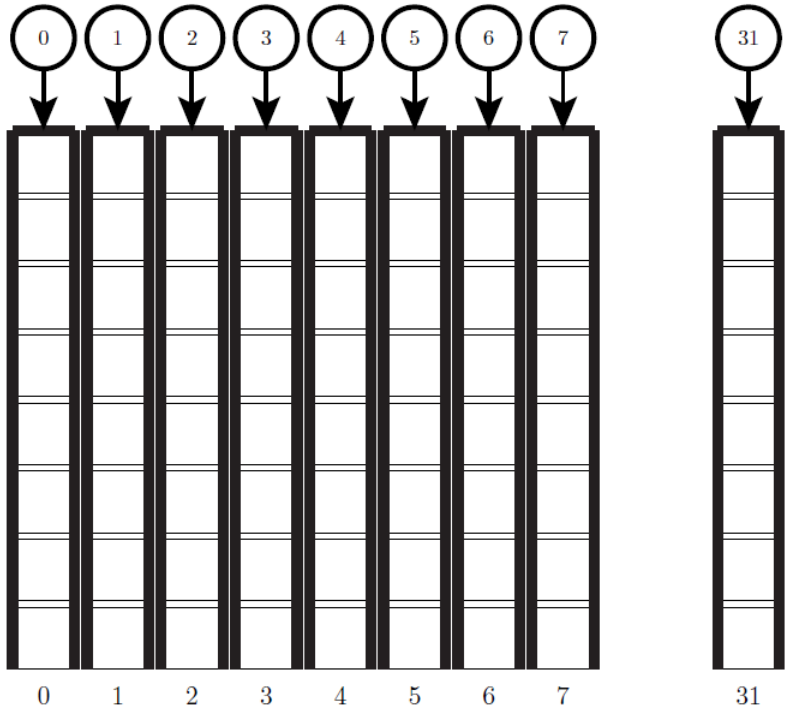
- Word storage order:
  - Banks are 4 bytes wide
  - Next word in next bank modulo 32





- Row access

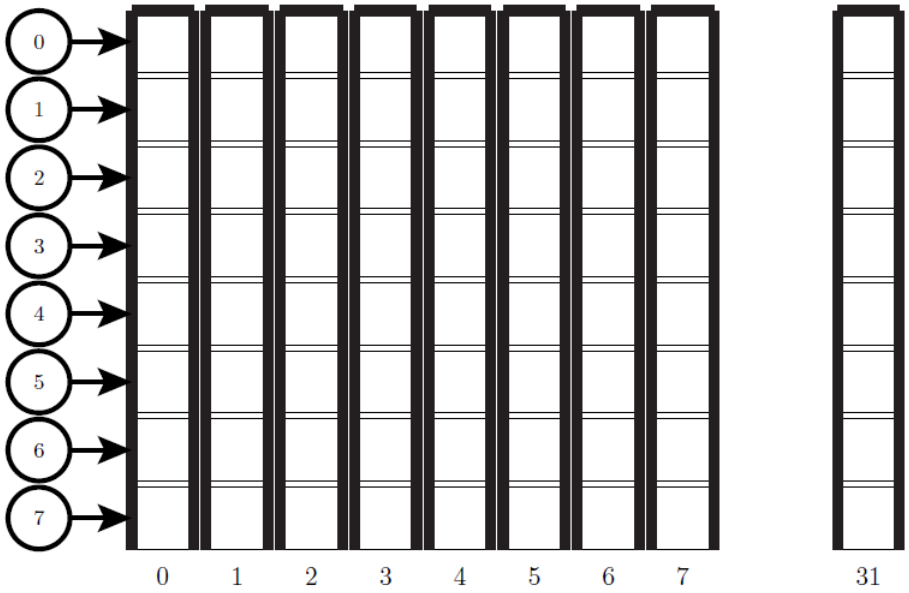
```
__local float sh[32][32];
```



Memory Access Overhead
Local Memory access (3)

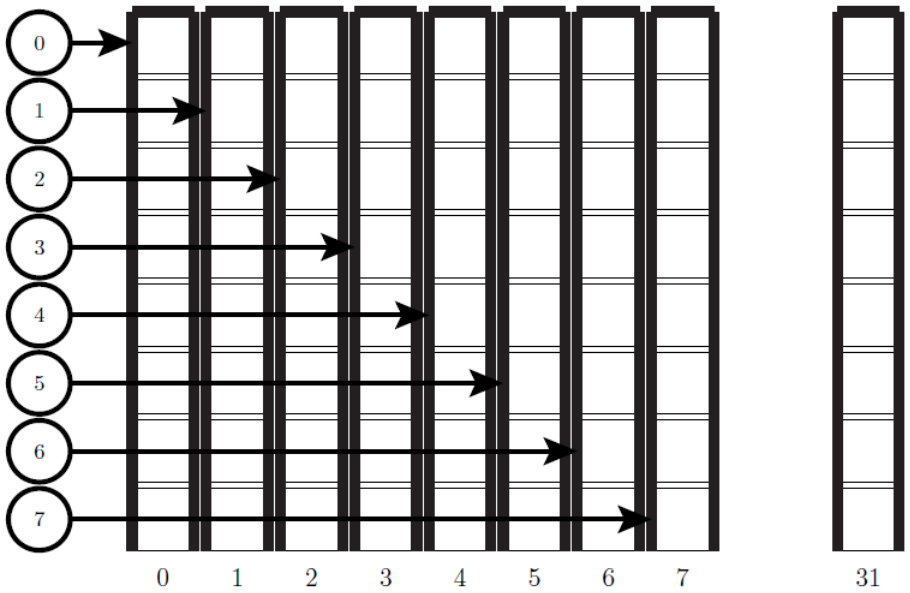
- Column access

```
__local float sh[32][32];
```



- Column access

```
__local float sh[32][33];
```



Computation Overhead
Excess Computation

- Unroll loops with a fixed number of iterations
  - Removes loop overhead
    - Index computations and tests
  - Increases ILP and DLP
  - Use `#pragma unroll`

- Let one work item process multiple data items
  - Thread index calculation overhead is ammortized
  - ILP and DLP will increase
  - Extra potential for loop unrolling

Computation Overhead
Branching – Definition

- A warp|wavefront runs in lockstep
  - 32|64 work items execute the same instruction
- For example:

```
if (x<5) y = 5; else y *= 2;
```

  - SIMD performs the 3 steps:
    - Test condition
    - ***then*** branch executed for threads for which condition holds
    - ***else*** branch executed for threads for which condition doen't hold

- Branch divergence decreases performance!

Computation Overhead
Branching – Remedies

- Lookup table
- Static thread reordering
  - Typical in reduction operations
  - See extended example
- Dynamic thread reordering
  - Reorder at runtime
  - Time lost reordering < time won due to reordering

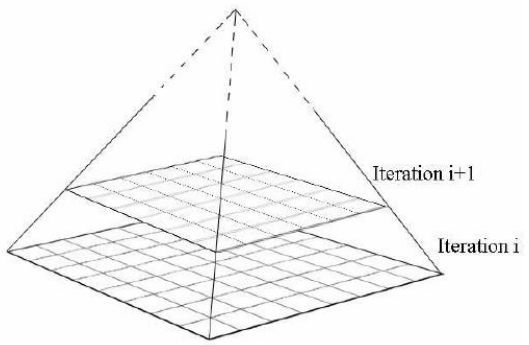
Minimize Idling
Local and global synchronization (1)

- Local sycnhronization:
  - Work items of the same group can synchronize:
    `barrier(CLK_LOCAL_MEM_FENCE);`
  - Work items that reach the barrier must wait
    - Cannot be chosen by the scheduler
    - ➔ Less potential for latency hiding

- Global synchronization
  - A new kernel must be launched!
  - Data must be written to and read from global memory

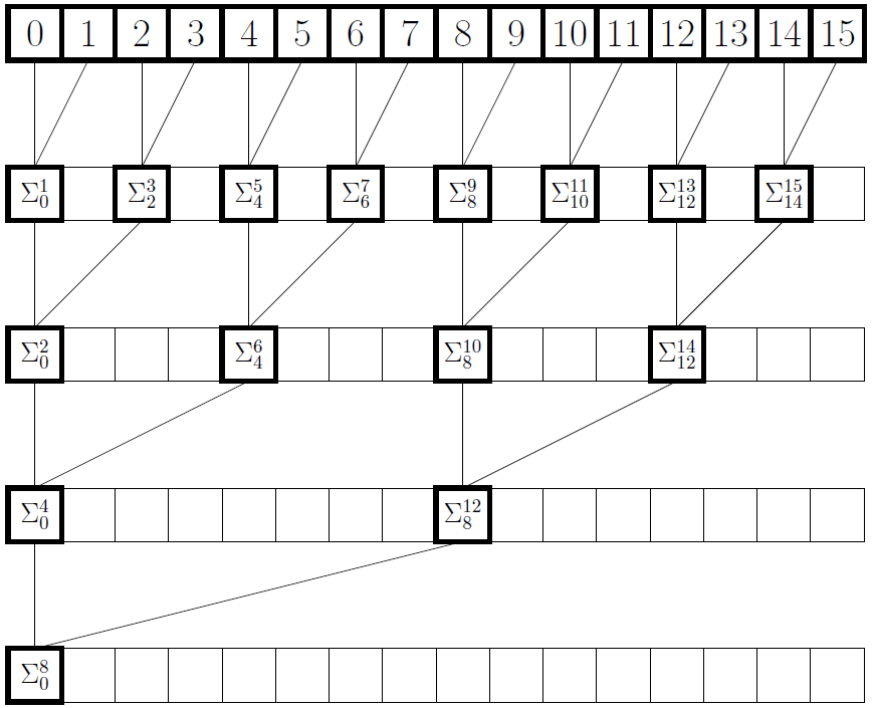
Minimize Idling
Local and global synchronization (2)

- Local synchronization:
  - Keep work groups small → less effect
  - No synchronization needed within warp/wavefront
- Global synchronization
  - Exchange computation for data access
  - E.g. Hotspot: calculate heat chip floorplan
    - $Heat_{cell} = f(heat_{neighbors})$
    - Calculate NxN tiles and synchronize each time?
    - No: calculate
      - Iteration 0: (N+k)x(N+k) tile
      - …
      - Iteration k-1: NxN tile



Parallel Sum

- Parallel Sum:
  - binary tree algorithm
- 6 different versions



Parallel Sum 1 and 2

From global to local memory

```
3  // naive global memory
4  __kernel void reduction1(__global int *src,
5                           __global int *dst,
6                           unsigned int size)
7  {
8      if (get_global_id(0) >= size)
9          return;
10
11     for (int s = 1; s < get_local_size(0); s *= 2) {
12         if (get_local_id(0) % (2*s) == 0) {
13             src[get_global_id(0)] += src[get_global_id(0) + s];
14         }
15         barrier(CLK_GLOBAL_MEM_FENCE);
16     }
17
18     if (get_local_id(0) == 0) {
19         dst[get_group_id(0)] = src[get_global_id(0)];
20     }
21
22     return;
23 }
```

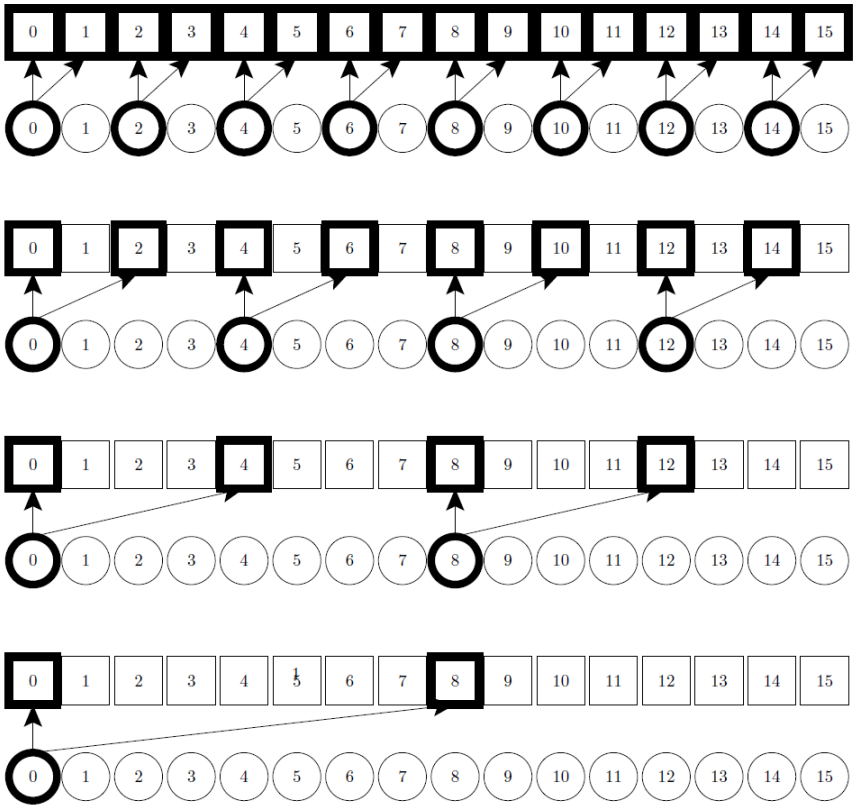
```
25 // naive local memory
26 __kernel void reduction2(__global int *src,
27                           __global int *dst,
28                           unsigned int size)
29 {
30     __local int shared[WGSZ];
31     unsigned int gx = get_global_id(0);
32     unsigned int lx = get_local_id(0);
33
34     shared[get_local_id(0)] = gx < size ? src[get_global_id(0)] : 0;
35     barrier(CLK_LOCAL_MEM_FENCE);
36
37     for (int s = 1; s < get_local_size(0); s *= 2) {
38         if (get_local_id(0) % (2*s) == 0) {
39             shared[get_local_id(0)] += shared[get_local_id(0) + s];
40         }
41         barrier(CLK_LOCAL_MEM_FENCE);
42     }
43
44     if (get_local_id(0) == 0) {
45         dst[get_group_id(0)] = shared[0];
46     }
47 }
```

Parallel Sum 3

Reduce idling threads

Divergence!

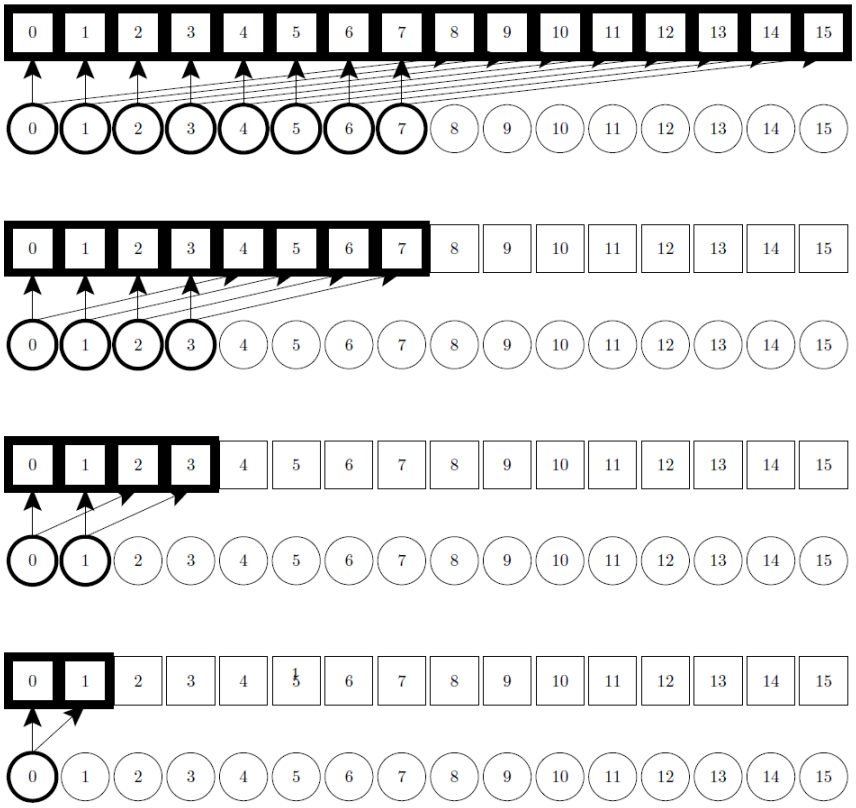
```
49 // local memory 1:2
50 __kernel void reduction3(__global int *src,
51                            __global int *dst,
52                            unsigned int size)
53 {
54     __local int shared[WGSZ];
55     unsigned int gx1 = get_global_id(0);
56     unsigned int gx2 = gx1 + get_global_size(0);
57     unsigned int lx = get_local_id(0);
58
59
60     shared[get_local_id(0)] = src[gx1];
61     if (gx2 < size) {
62         shared[get_local_id(0)] += src[gx2];
63     }
64
65     barrier(CLK_LOCAL_MEM_FENCE);
66
67     for (int s = 1; s < get_local_size(0); s *= 2) {
68         if (get_local_id(0) % (2*s) == 0) {
69             shared[get_local_id(0)] += shared[get_local_id(0) + s];
70         }
71         barrier(CLK_LOCAL_MEM_FENCE);
72     }
73     if (get_local_id(0) == 0) {
74         dst[get_group_id(0)] = shared[0];
75     }
76 }
```



Parallel Sum 4

Thread reordering

```
78 // static thread reordered reduction
79 __kernel void reduction4(__global int *src,
80                           __global int *dst,
81                           unsigned int size)
82 {
83     __local int shared[WGSZ];
84     unsigned int gx1 = get_global_id(0);
85     unsigned int gx2 = gx1 + get_global_size(0);
86     unsigned int lx = get_local_id(0);
87
88     shared[get_local_id(0)] = src[gx1];
89     if (gx2 < size) {
90         shared[get_local_id(0)] += src[gx2];
91     }
92
93     barrier(CLK_LOCAL_MEM_FENCE);
94
95     for (int s = get_local_size(0)/2; s >= 1; s /= 2) {
96         if (get_local_id(0) < s) {
97             shared[get_local_id(0)] += shared[get_local_id(0) + s];
98         }
99         barrier(CLK_LOCAL_MEM_FENCE);
100    }
101
102    if (get_local_id(0) == 0) {
103        dst[get_group_id(0)] = shared[0];
104    }
105 }
```



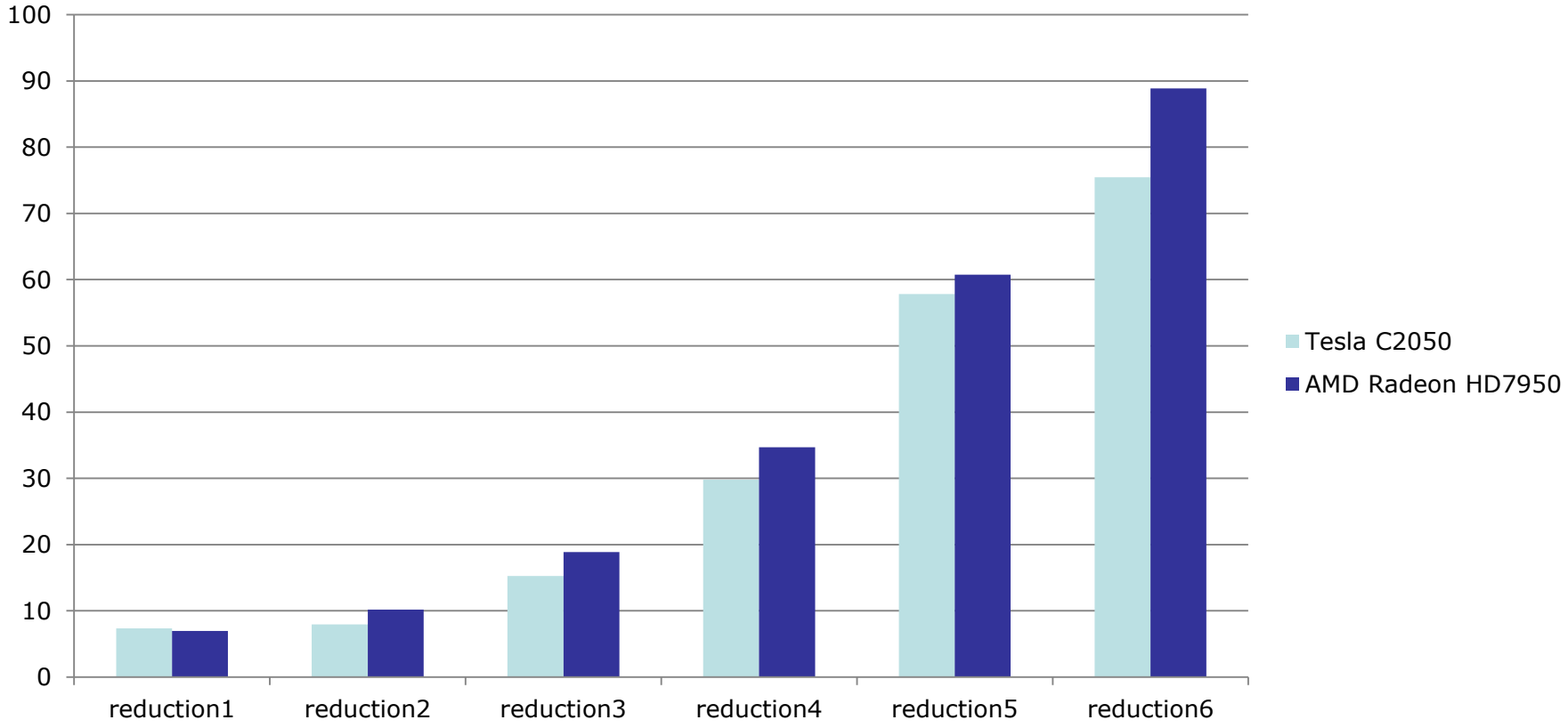
Parallel Sum 5 and 6

Multiple elements per work item and loop unrolling

```
107 // multiple elements per work item
108 // could have tried int4 values ...
109 __kernel void reduction5(__global int *src,
110                           __global int *dst,
111                           unsigned int size)
112 {
113     __local int shared[WGSZ];
114     unsigned int x0 = get_global_id(0);
115     unsigned int x1 = get_global_id(0) + get_global_size(0);
116     unsigned int x2 = get_global_id(0) + 2*get_global_size(0);
117     unsigned int x3 = get_global_id(0) + 3*get_global_size(0);
118
119     shared[get_local_id(0)] = src[x0] + src[x1] + src[x2];
120     if (x3 < size) {
121         shared[get_local_id(0)] += src[x3];
122     }
123     barrier(CLK_LOCAL_MEM_FENCE);
124
125     for (int s = get_local_size(0)/2; s >= 1; s /= 2) {
126         if (get_local_id(0) < s) {
127             shared[get_local_id(0)] += shared[get_local_id(0) + s];
128         }
129         barrier(CLK_LOCAL_MEM_FENCE);
130     }
131
132     if (get_local_id(0) == 0) {
133         dst[get_group_id(0)] = shared[0];
134     }
135 }
```

```
137 // add unrolling of last warp
138 void unrolled(volatile __local int *shared, int tid)
139 {
140     shared[tid] += shared[tid + 32];
141     shared[tid] += shared[tid + 16];
142     shared[tid] += shared[tid + 8];
143     shared[tid] += shared[tid + 4];
144     shared[tid] += shared[tid + 2];
145     shared[tid] += shared[tid + 1];
146
147 }
148
149 __kernel void reduction6(__global int *src,
150                          __global int *dst,
151                          unsigned int size)
152 {
153     __local int shared[WGSZ];
154     unsigned int x0 = get_global_id(0);
155     unsigned int x1 = get_global_id(0) + get_global_size(0);
156     unsigned int x2 = get_global_id(0) + 2*get_global_size(0);
157     unsigned int x3 = get_global_id(0) + 3*get_global_size(0);
158
159     shared[get_local_id(0)] = src[x0] + src[x1] + src[x2];
160     if (x3 < size) {
161         shared[get_local_id(0)] += src[x3];
162     }
163     barrier(CLK_LOCAL_MEM_FENCE);
164
165     for (int s = get_local_size(0)/2; s > 32; s /= 2) {
166         if (get_local_id(0) < s) {
167             shared[get_local_id(0)] += shared[get_local_id(0) + s];
168         }
169         barrier(CLK_LOCAL_MEM_FENCE);
170     }
171
172     if (get_local_id(0) < 32) {
173         unrolled(shared, get_local_id(0));
174     }
175
176     if (get_local_id(0) == 0) {
177         dst[get_group_id(0)] = shared[0];
178     }
179 }
```

Resulting Performance [GB/s]



Images

OpenCL Images
Background

- GPUs have texture memory
  - Special hardware to deal with images
  - Take advantage of:
    - 2D- caching
    - Hardware interpolation of pixel values
    - Automatic handling of out-of-bounds access
- To work with images you need to create:
  - Image buffers
    - Cfr regular buffers
  - Image samplers
    - To access your image

OpenCL Images
image buffers

Host Code

```
cl_mem clCreateImage(
  cl_context context,
  cl_mem_flags flags,
  const cl_image_format *format,
  const cl_image_desc *image_desc,
  void *host_ptr,
  cl_int *errcode_ret)
```

- Image description:
  - Image dimensions
- Image format:
  - Channel order
  - Channel data type
- OpenCL <= 1.1:
  - clCreateImage1D, clCreateImage2D and clCreateImage3D

Device Code

```
__kernel void manipulateImage(
          __read_only image2d_t src_image,
          __write_only image2d_t dst_image,
          __global sampler_t sampler)
```

- Image:
  - read_only XOR write_only
- Sampler:
  - Necessary to access the image
  - See next

OpenCL Images
image samplers

Host Code

```
cl_sampler clCreateSampler (
    cl_context context,
    cl_bool normalized_coords,
    cl_addressing_mode addressing_mode,
    cl_filter_mode filter_mode,
    cl_int *errcode_ret)
```

- Normalized coordinates:
  - If true: coordinates in [0, 1.0]
- Addressing mode:
  - Behaviour for out of bounds access
- Filter mode:
  - Interpolation behaviour

Device Code

```
__kernel void darkenImage(
            __read_only image2d_t src_image,
            __write_only image2d_t dst_image,
            __global sampler_t sampler)
{
    int2 coord = (int2)(get_global_id(0),
                        get_global_id(1));
    uint offset = get_global_id(1)*0x4000 +
                  get_global_id(0)*0x1000;
    uint4 pixel = read_imageui(src_image,
                               sampler,
                               coord);
    pixel.x -= offset;
    write_imageui(dst_image, coord, pixel);
}
```