Introduction to OpenCL

Sylvain Lefebvre
INRIA

Forword

- Contact:

  sylvain.lefebvre@inria.fr

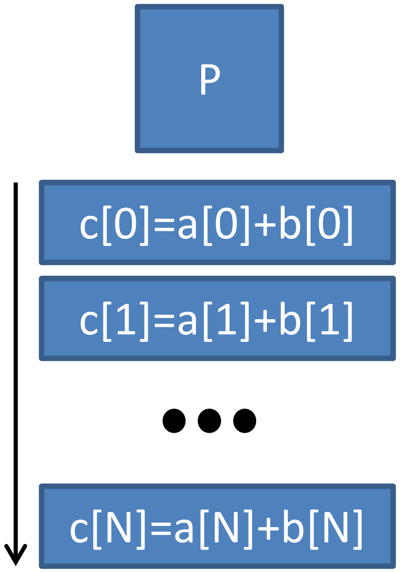
- Source code:

  http://webloria.loria.fr/~slefebvr/teaching/pcomp/

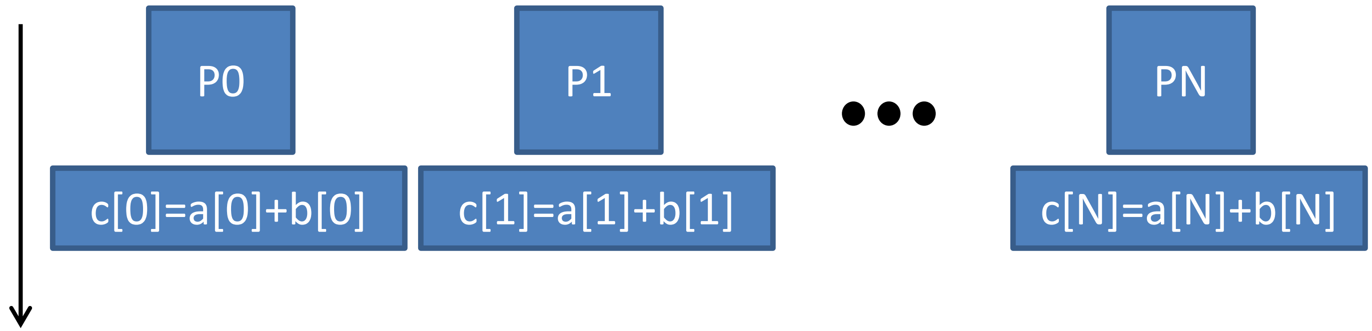
Data parallelism

- SIMD architectures
  - Single Instruction Multiple Data
- Example:
  - Large vector addition

    For i = 0 to N { c[i] = a[i] + b[i]; }



  - With SIMD



Why now?

- Could have been built before
  - And it has been built …
- Main reasons for recent success are:
  - Single core CPUs reach their limits
  - GPUs are widely available

GPU

- **Graphics** Processing Unit

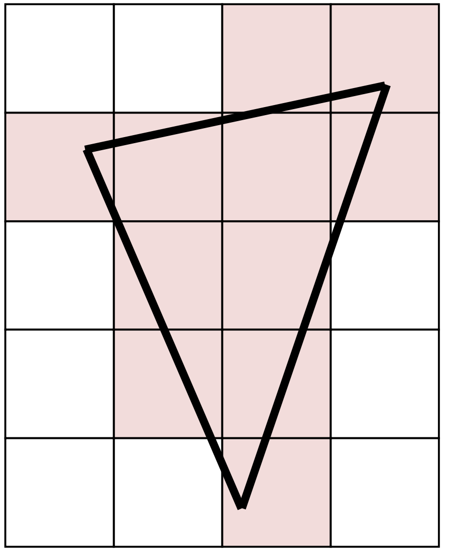
You have one in your mobile phone!

Thanks to video games

Why?

- 3D rendering:
  - Many pixels
  - Similar operations in each

  ➔SIMD is perfect for this



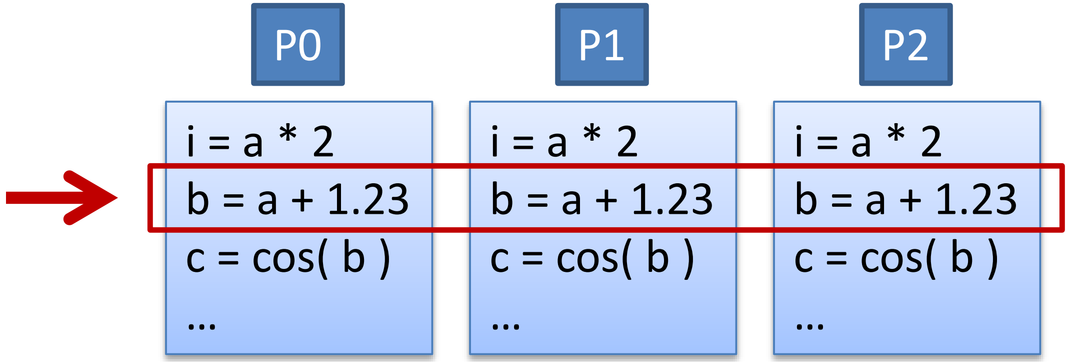
- Game quality increased
- Developers asked for programmable GPUs

First programming GPUs

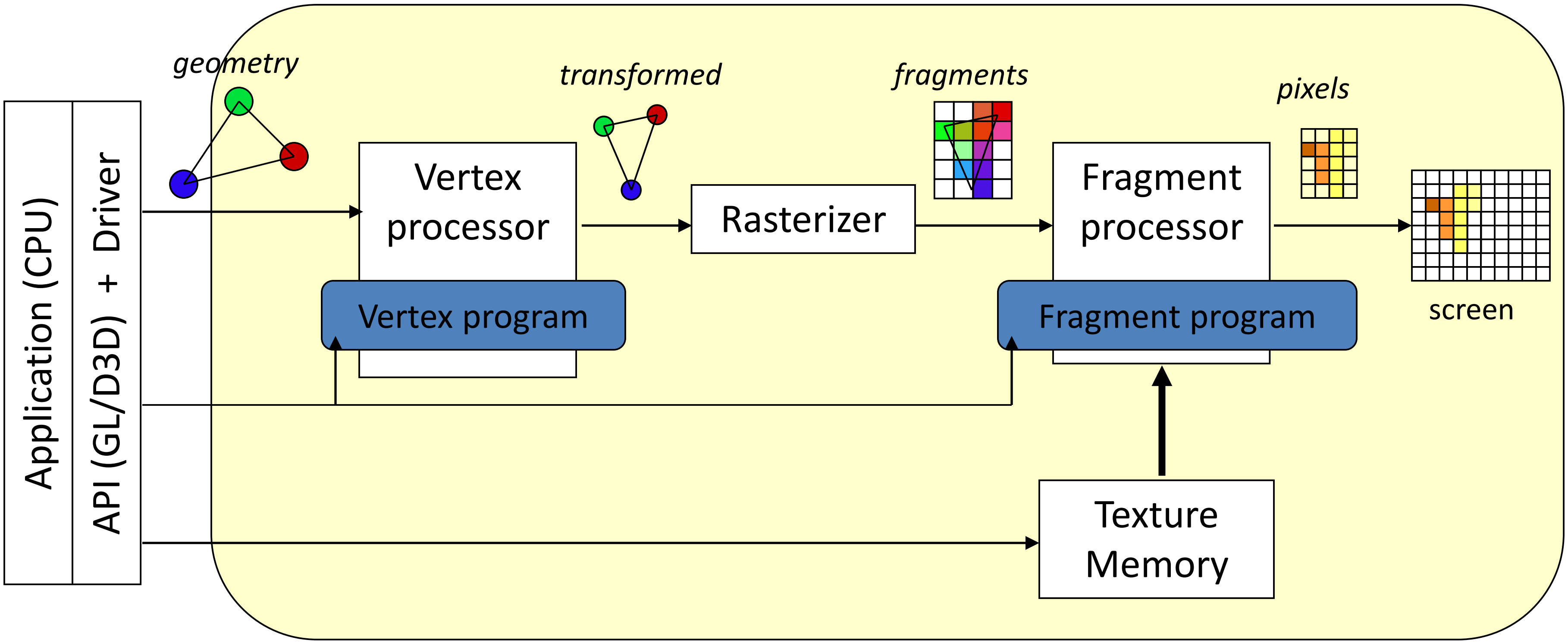
- Tipping point: Geforce3 in 2001
- First truly generic GPU: GeForce FX 5x
  - Programmed with Nvidia CG
- Generality kept increasing

Concept

- One processor per pixel
- Each processor executes the same program
- Processors are in ‘lockstep’
  - Same instruction executed at each step



Graphics pipeline (simplified)



Programmable graphics pipeline

- At first, all had to be done within this pipeline
  - No data-dependent loop instruction (while/for)
  - No true if/the/else
  - No generic writes (had to align with pixels)

- There was a reason to these limitations:

  <u>**Performance**</u>

Nowadays

- No longer limited to graphics pipeline
  - CUDA / OpenCL are generic languages
  - DirectX / OpenGL support ‘compute’
- Few limitations on what is possible
  - But everything has a cost!!
  - Optimizing **<u>is</u>** difficult

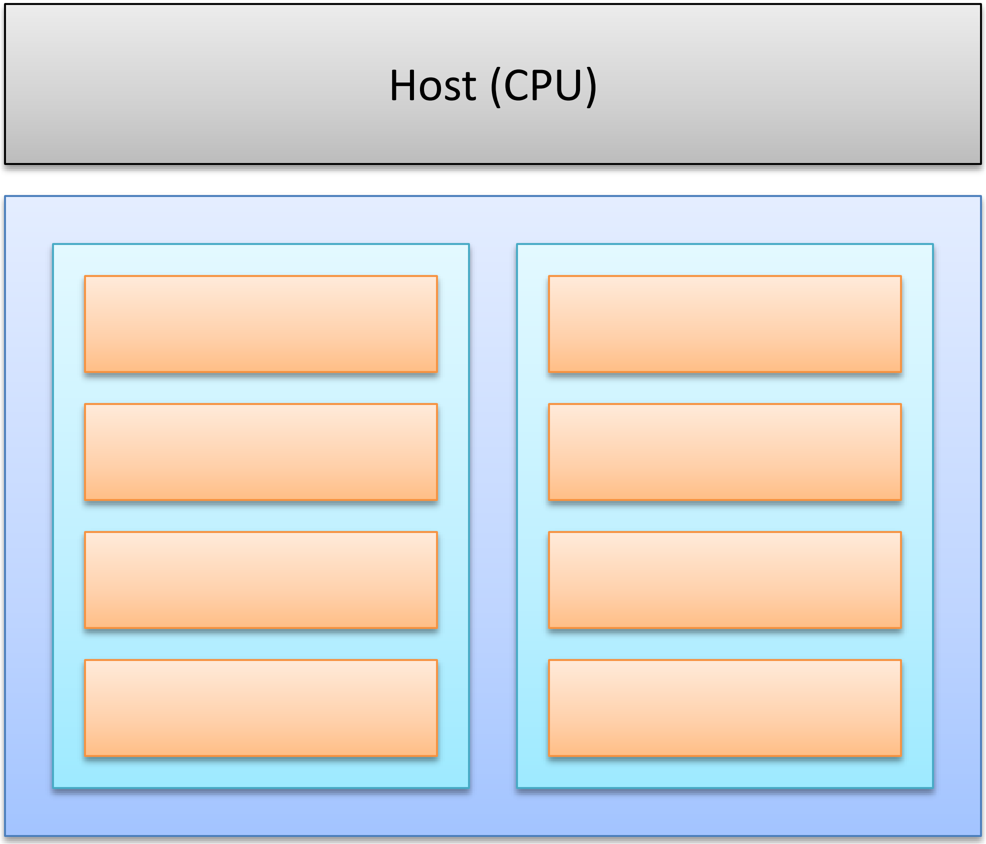
OpenCL

- The easiest to start with
  - Multi-platform, multi-hardware
- Concepts similar to CUDA
- Somewhat lower performance
  - But will probably catch-up
- Currently fewer libraries

A closer look at the architecture

(Following OpenCL naming conventions)

- Host
- Compute device
- Compute units
- Processing elements



Execution model

- A compute device executes a *kernel* ...

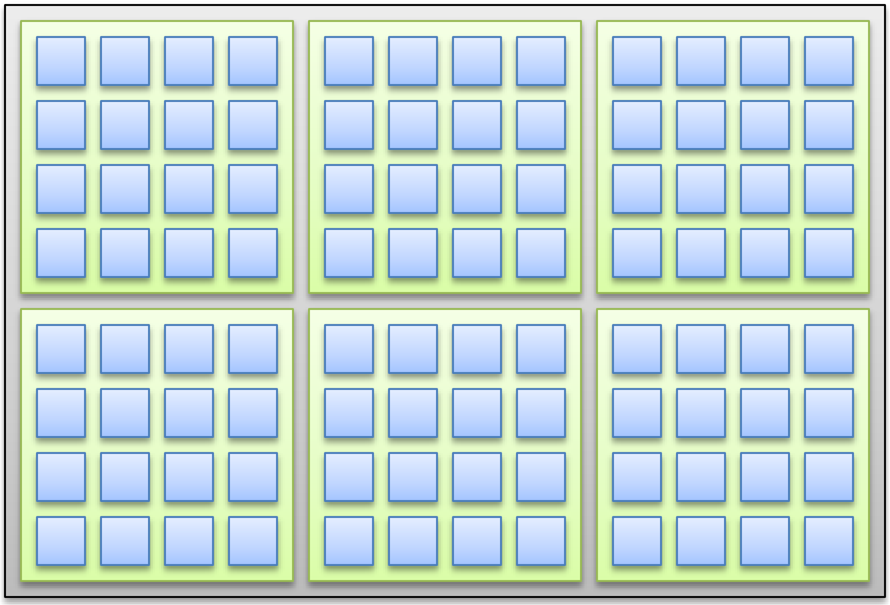
  ... in parallel on the processing elements

```
__kernel void mainKernel(
    __global const int *a,
    __global const int *b,
    __global int *c)
{
    int id = get_global_id( 0 );
    c[ id ] = a[ id ] + b[ id ];
}
```

Execution model

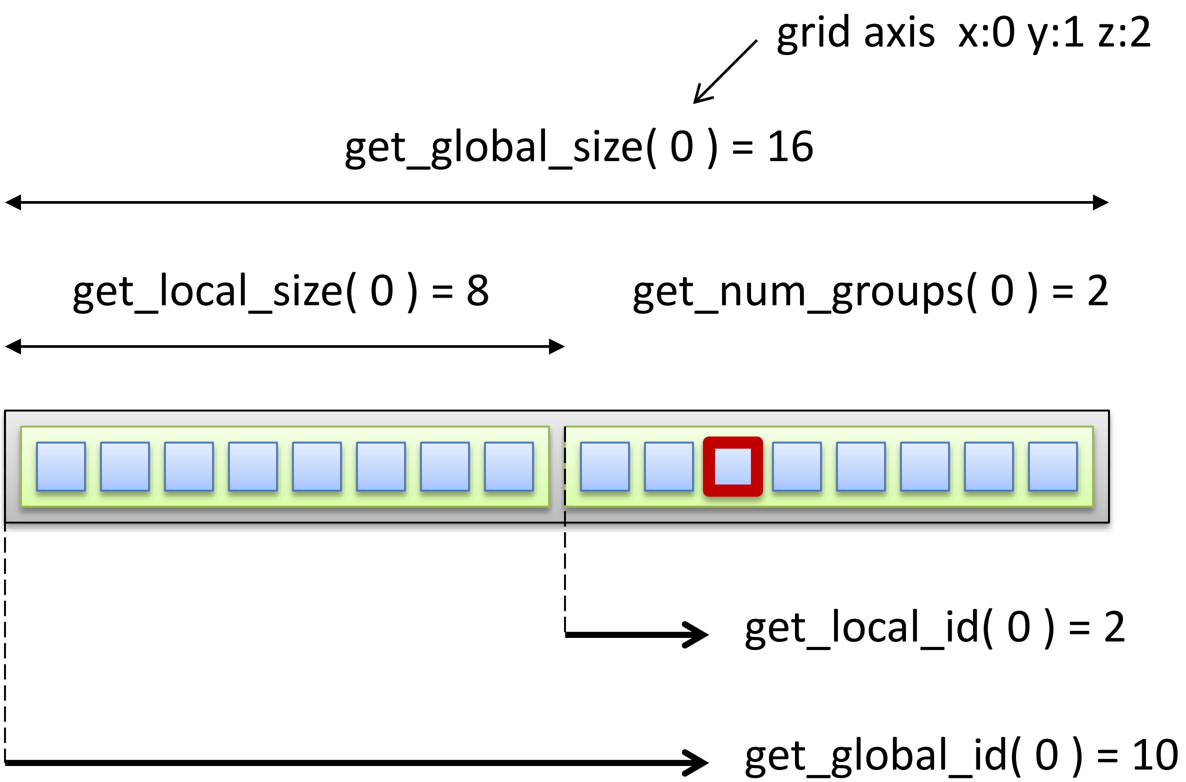
- Computations organized on a grid
  - 1D, 2D or 3D
- Work-items, work-groups
  - 12 x 8 work-items
  - 3 x 2 work-groups
  - Each group is 4x4 items

2D execution grid



Execution kernel

- The kernel can request its location



Kernel

```
__kernel void mainKernel(
    __global const int *a,
    __global const int *b,
    __global int *c)
{
    int id = get_global_id( 0 );
    c[ id ] = a[ id ] + b[ id ];
}
```

Next

OpenCL API and language

OpenCL

NVidia: included in CUDA Toolkit

http://developer.nvidia.com/object/cuda_download.html

AMD: developer package

http://developer.amd.com/sdks/AMDAPPSDK/downloads/Pages/default.aspx

➔ Can also be used with multi-core CPU

OpenCL

- Two parts:
  - C / C++ API running on the host
  - The kernels running on the device

- Typical program:
  1. Setup OpenCL
  2. Load program from file and send to device
  3. Send data from host to device
  4. Execute kernel on device
  5. Read back result from device to host

Processing queue

- Host and device run in parallel
- Host sends commands
  - Waiting would waste time!
- Commands are added to a queue
  - Device consumes commands
  - Host can do something else
  - Host checks whether results are available

OpenCL Setup

- Get information about platform

```
std::vector<cl::Platform> platformList;
cl::Platform::get(&platformList);
```

- Create context

```
clu_Context=new cl::Context( … )
```

- Get devices

```
clu_Devices=clu_Context->getInfo<CL_CONTEXT_DEVICES>();
```

- Create processing queue

```
clu_Queue=new cl::CommandQueue( … )
```

This is “program.cl”

```
__kernel void mainKernel(
    __global const int *a,
    __global const int *b,
    __global int *c)
{
    int id = get_global_id( 0 );
    c[ id ] = a[ id ] + b[ id ];
}
```

Loading a program

```
string prog = loadFileIntoString("program.cl");
cl::Program::Sources source( 1,
             make_pair(prog.c_str(), prog.length()+1));
```

- A program may have multiple kernels

```
g_Kernel = new cl::Kernel(
                        *g_Program, "mainKernel",&err);
```

Data buffers

```
int inBuffer_host[1024];
 /// write something in inBuffer_host
cl::Buffer inBuffer(
       *g_Context,
        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
        1024 * sizeof(int),
        inBuffer_host);

cl::Buffer outBuffer(
       *g_Context,
        CL_MEM_WRITE_ONLY,
        1024 * sizeof(int),
        NULL);
```

Arguments

```
err = g_Kernel->setArg(0, inBuffer);
```

```
__kernel void mainKernel(
    __global const int *a,
    __global const int *b,
    __global int *c)
{
    int id = get_global_id( 0 );
    c[ id ] = a[ id ] + b[ id ];
}
```

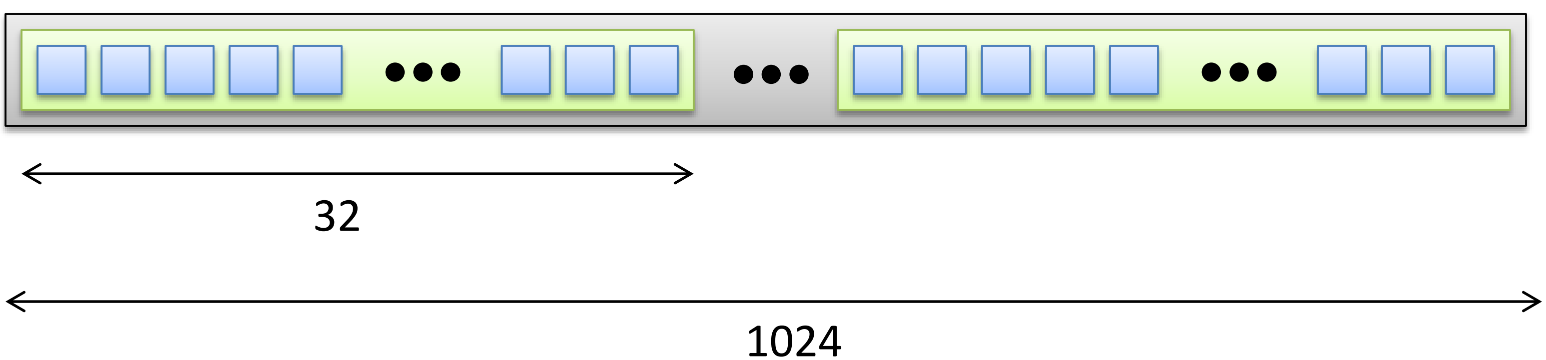
arg 0
arg 1
arg 2

Executing the kernel

```
err = g_Queue->enqueueNDRangeKernel(
        *g_Kernel,
         cl::NullRange,
         cl::NDRange(1024),
         cl::NDRange(32)
      );
```

Global work size (can be large)

Group size (limited, e.g. 128 max)



Reading back data

```
int result[1024];
err = g_Queue->enqueueReadBuffer(
    outBuffer,
    false, // do not wait
    0,
    1024 * sizeof(int),
    result);

// Finish all operations
g_Queue->finish();

// Display the result
for (int i = 0; i < 1024 ; i++ ) {
  cerr << result[i] << ' ';
}
```

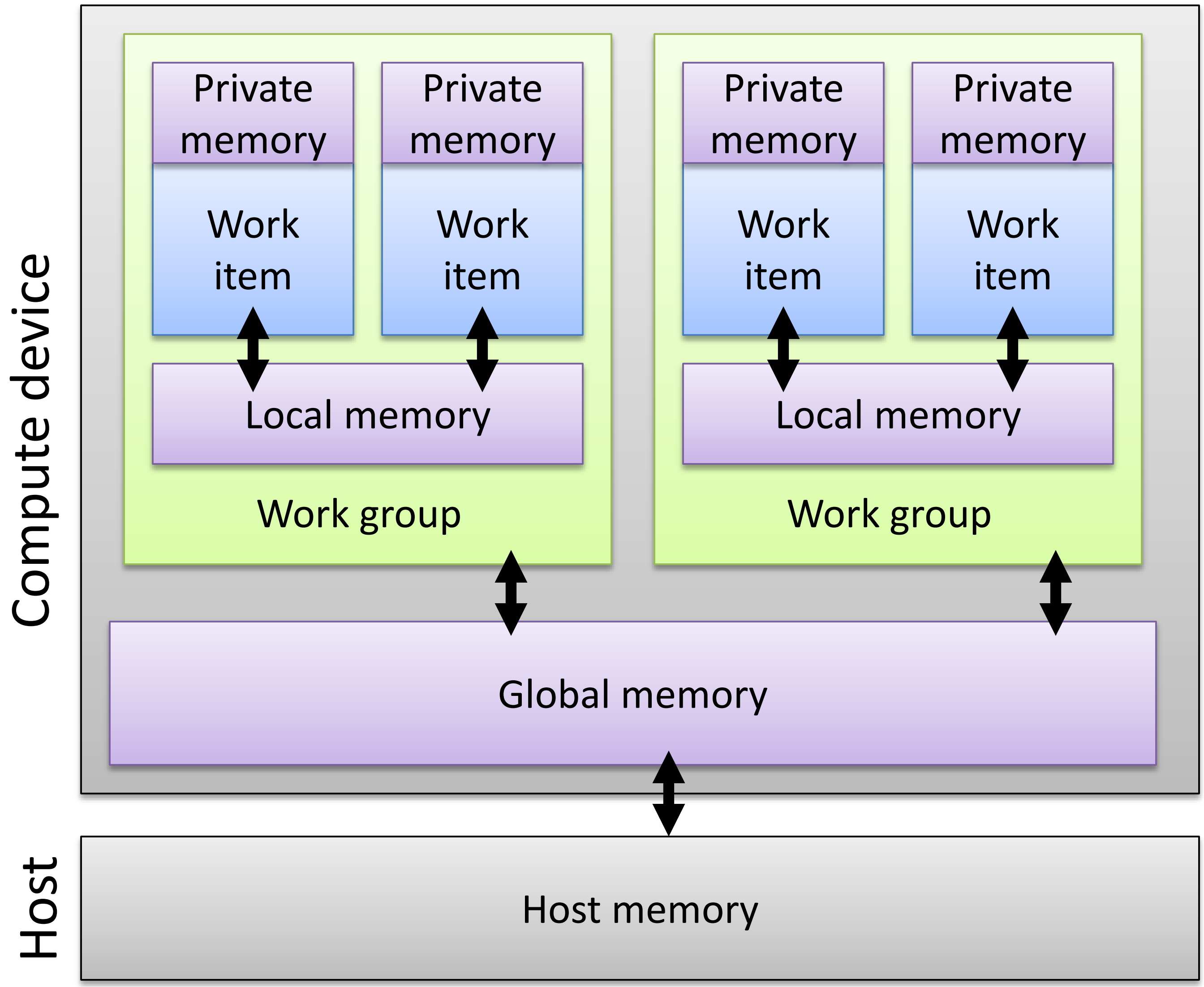
Demo

- Intro_basics

Next

Memory model

Memory model



Allocating global memory

- Use cl::Buffer

Host C++

```
Cl::Buffer mem(context, CL_MEM_READ_WRITE, byte_sz);

k->setArg(0 , mem )
```

OpenCL

```
__kernel void mainKernel ( __global int *data )
{
   ...
}
```

Allocating local memory

- Within CL:

```
__kernel void mainKernel()
{
    __local int shared[1024];
}
```

- From host:

```
k->setArg(0 , cl::__local(
                    1024*sizeof(int)) )
```

```
__kernel void mainKernel (
        __local int *shared )
{
   ...
}
```

Local vs. global

- Local memory is <u>very</u> fast

- Visible by all threads within a work group
  - Never across work groups

- Very limited in size
  - 16KB – 32KB per compute unit
  - Split between work groups

Memory types

```
__kernel void mainKernel( __global const int *a )
{
    __local int tmp;

    int id = get_global_id( 0 );


}
```

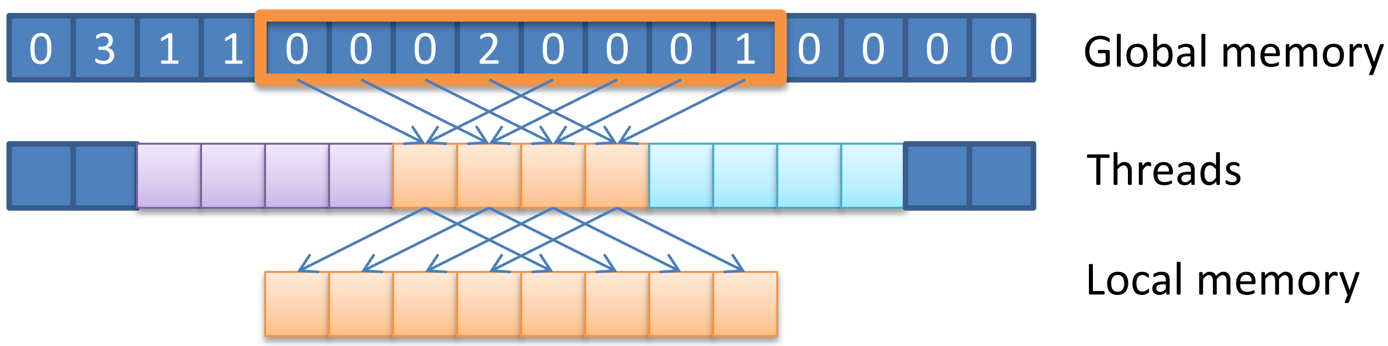
Global, unique

Shared, one per work group

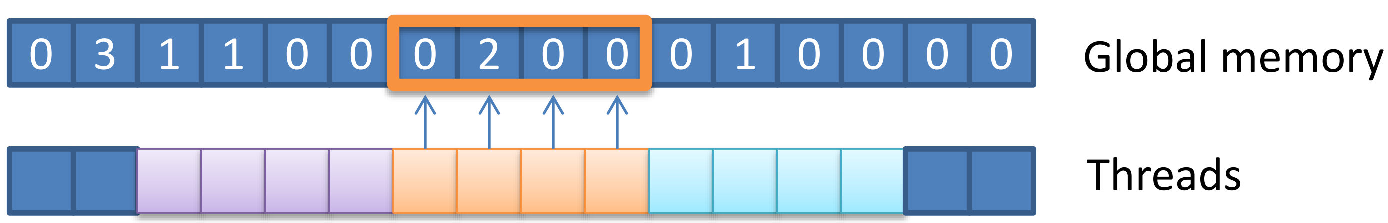
Private memory, one per thread

Pre-fetch in local

1. Each thread in group reads some *global* data

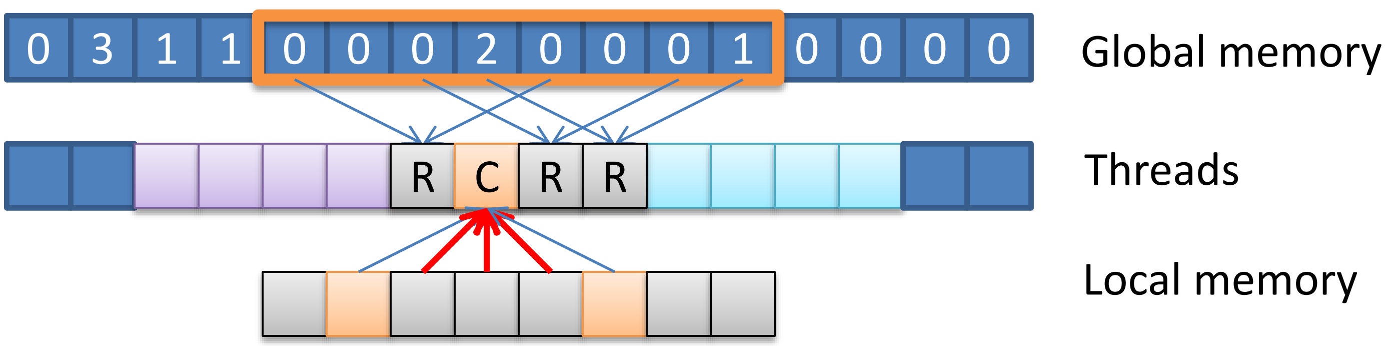


2. Each thread computes from *local* data
3. Threads dump result in *global*



Pre-fetch in local

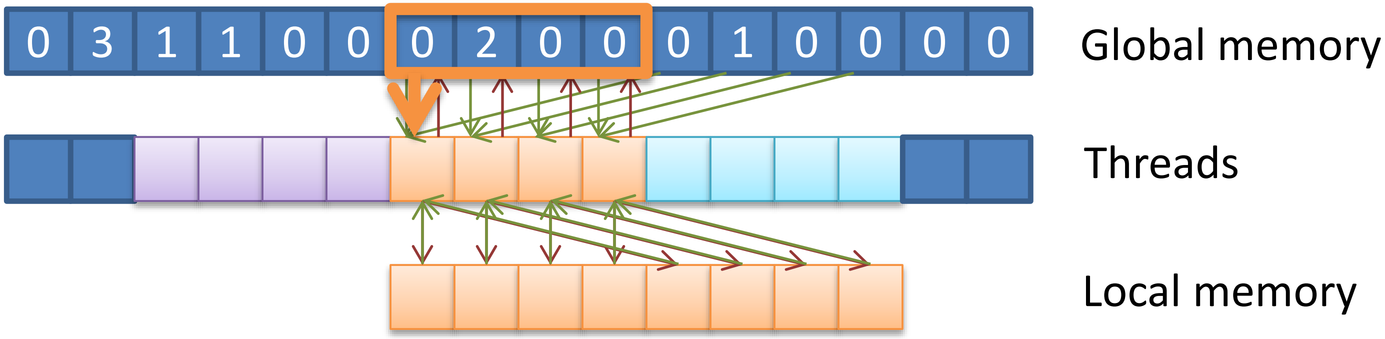
2. Each thread computes from *local* data
   - Could start while others are not done reading
   - Requires a synchronization!



R: read
C: compute

Demo

- intro_shared



Next

Memory efficiency

Efficient memory: Global

- Coalesced accesses
  - Threads in a group share memory access.
  - Bandwidth is large if access is *coalesced*.
  - If not, accesses are *serialized.*

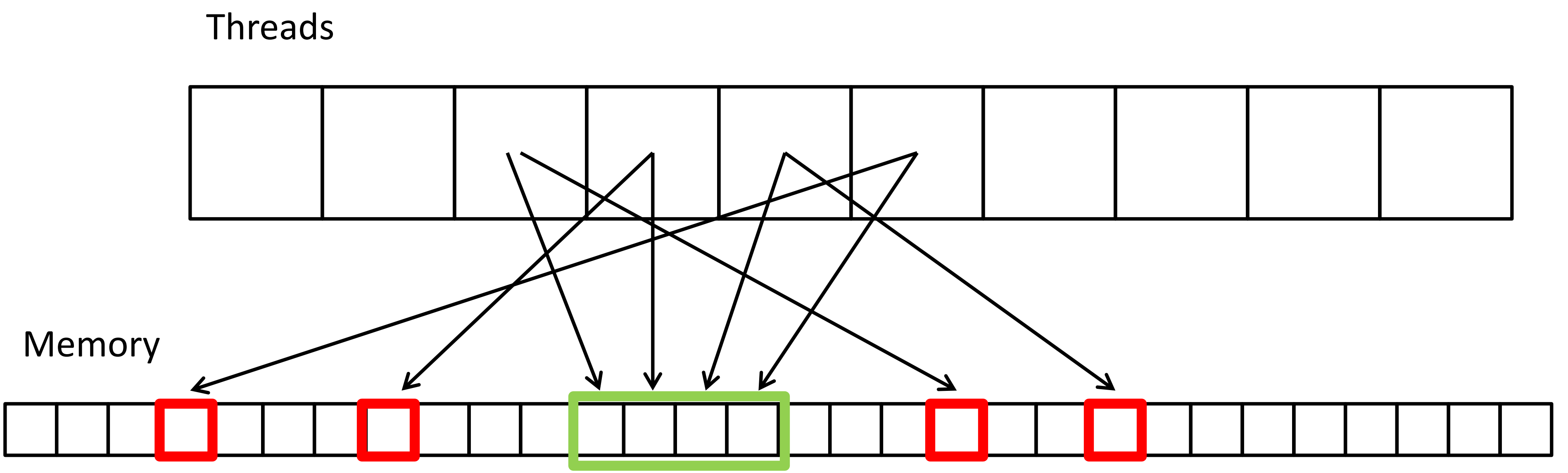
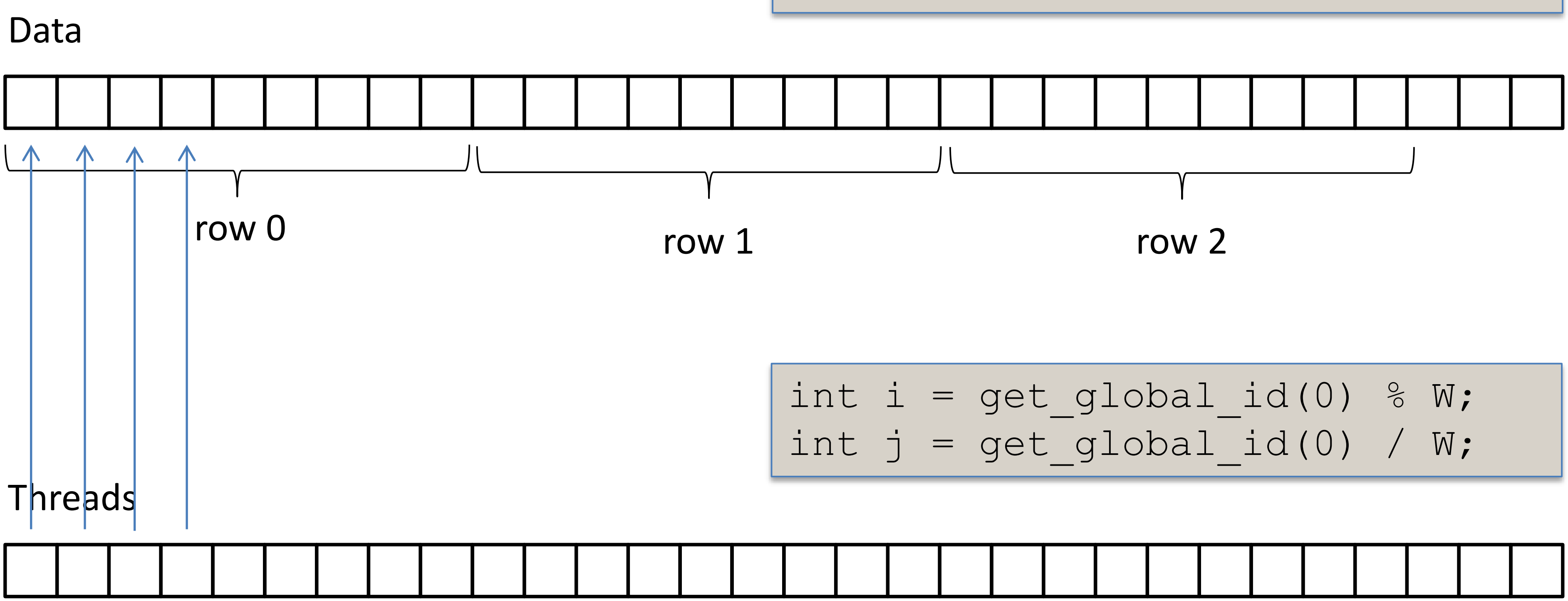


Image horizontal ‘flip’

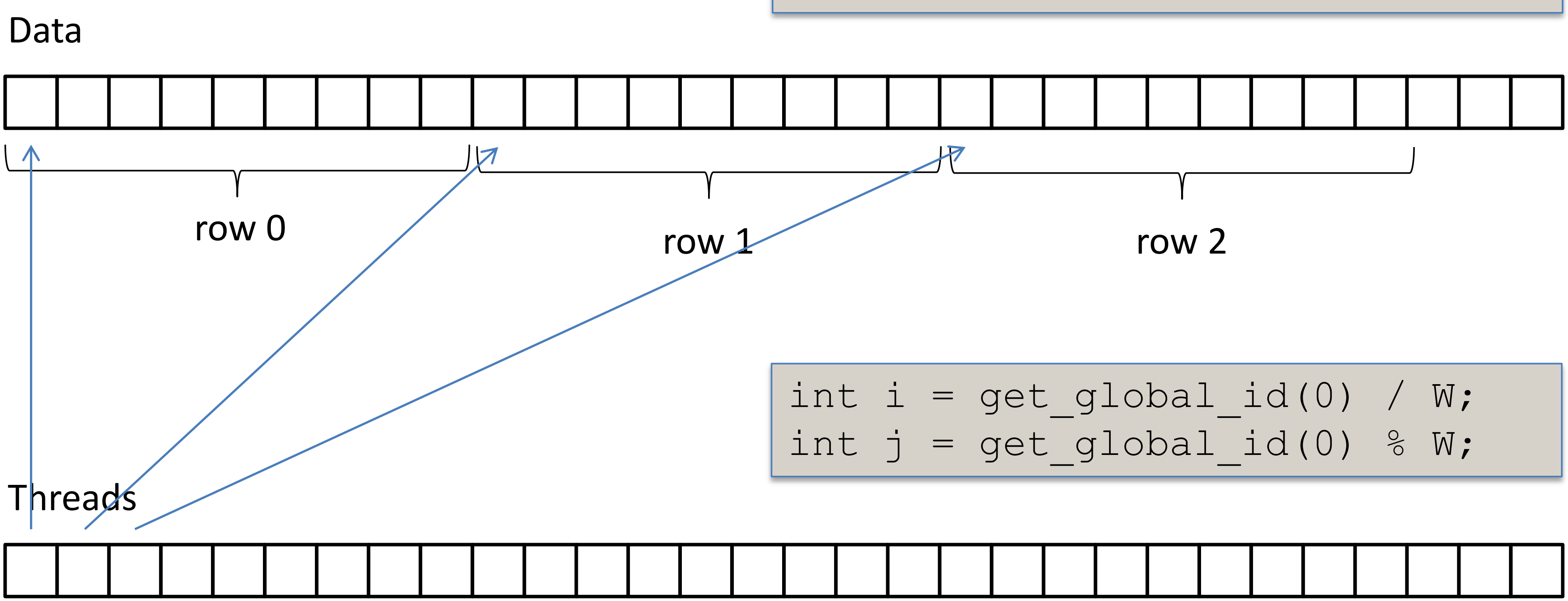
```
if ( i < W/2 ) {
   int f              = W-1-i;
   int tmp            = img[ f + j * W ];
   img[ f + j * W ] = img[ i + j * W ];
   img[ i + j * W ] = tmp;
 }
```




```
int i = get_global_id(0) % W;
int j = get_global_id(0) / W;
```

Image horizontal ‘flip’

```
if ( i < W/2 ) {
    int f              = W-1-i;
    int tmp            = img[ f + j * W ];
    img[ f + j * W ] = img[ i + j * W ];
    img[ i + j * W ] = tmp;
}
```



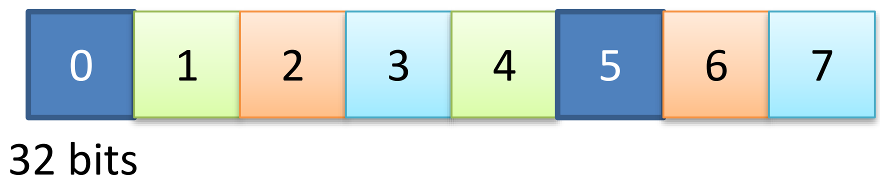

```
int i = get_global_id(0) / W;
int j = get_global_id(0) % W;
```

Demo

- Intro_coalesced

Efficient memory: Local

- Interleaved banks (Nvidia prog. guide)
  - Continuous addresses map to different banks



  - Threads within group should avoid conflicts: Different thread, different bank
  - Exception: broadcast (all access same)

Bank conflicts

- Depends on hardware
  - ATI / Nvidia / card generation

- Global memory:
  - Also suffers from bank conflicts (➔ __constant)
  - But generally less important than coalescence

Demo

- Intro_banks

Thank you

Questions?

Next

Synchronization

Synchronization mechanisms

- Memory barriers
  - OpenCL instruction `barrier`
  - `barrier( CLK_LOCAL_MEM_FENCE );`
  - `barrier( CLK_GLOBAL_MEM_FENCE );`

- Atomic operations
  - OpenCL instructions `atomic_*`

Barriers

- Guarantees that all memory operations ***within work group*** are completed

- Two flavors:
  - CLK_**LOCAL**_MEM_FENCE
  - CLK_**GLOBAL**_MEM_FENCE
  - Local synchronization is faster

Warnings

- Execution stops until **all** thread in group reach it

- If (x) { barrier() } else { ... }
  - ➔ Will likely hang the device

- for (int i = 0 ; i < N ; i ++) { ... barrier() ... }
  - ➔ Ok iff N is a constant (no data dependency)

Demo

- Intro_synchro (barrier)

Atomic operations

- Synchronization at the hardware level
  - atomic_inc / dec
  - atomic_add / sub
  - atomic_and / or / xor
  - atomic_min / max
  - atomic_cmpxchg (compare and swap)
  - atomic_xchg

Atomic operations

- Avoids complex synchronization mechanisms
  - To be preferred whenever possible.

- No free lunch:
  - If two threads conflicts, a slow down results.
  - However, result is correct!

Atomics in OpenCL

- Requires an extension

```
#pragma OPENCL EXTENSION cl_khr_global_int32_base_atomics : enable
#pragma OPENCL EXTENSION cl_khr_local_int32_base_atomics : enable
#pragma OPENCL EXTENSION cl_khr_global_int32_extended_atomics : enable
#pragma OPENCL EXTENSION cl_khr_local_int32_extended_atomics : enable
```

Demo

- Intro_synchro (atomic)

Exercise

- Basic programming with OpenCL