OpenCL
An Introduction for HPC programmers

(based on a tutorial presented at ISC'11 by Tim Mattson and Udeepta Bordoloi)

Tim Mattson, Intel

Acknowledgements: Ben Gaster (AMD), Udeepta Bordoloi (AMD)
and colleagues in the Khronos OpenCL Group.

Preliminaries:

- **Disclosures**
  - The views expressed in this tutorial are those of the people delivering the tutorial.
    - We are <u>not</u> speaking for our employers.
    - We are <u>not</u> speaking for Khronos

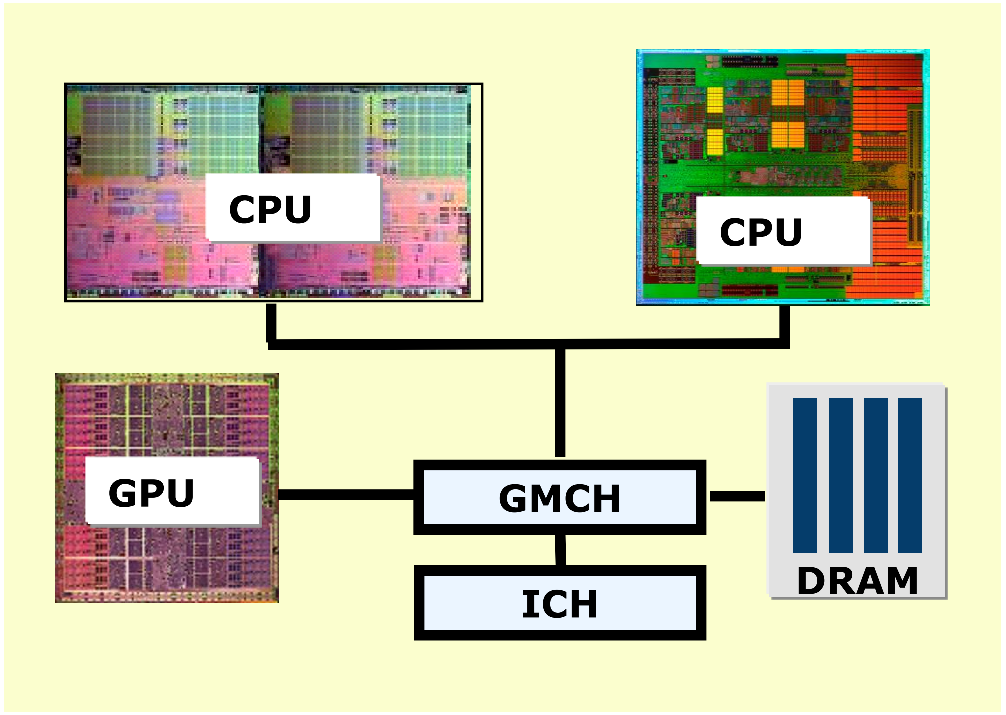
- **We take our tutorials VERY seriously:**
  - Help us improve … give us feedback and tell us how you would make this tutorial even better.

Agenda

- **Heterogeneous computing and the origins of OpenCL**
- **OpenCL overview**
- **Mapping OpenCL onto CPUs**
- **Exploring the spec with code: embarrassingly parallel**
  - Vector addition: the basic platform layer
  - Matrix multiplication: kernel programming language
- **Exploring the spec with code: dealing with dependencies**
  - Optimizing matrix mul.: work groups and the memory model
  - Reduction: synchronization
- **A survey of the rest of OpenCL**

It’s a Heterogeneous world

- A modern platform Includes:
  - One or more CPUs
  - One or more GPUs
  - DSP processors
  - … other?

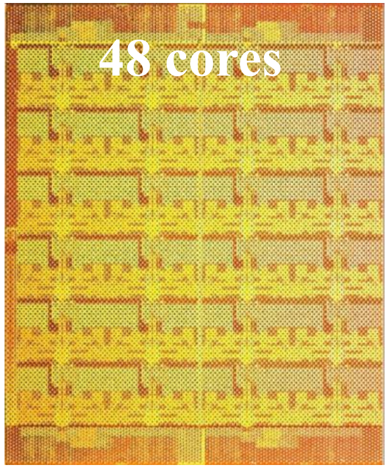


OpenCL lets Programmers write a single portable program that uses ALL resources in the heterogeneous platform

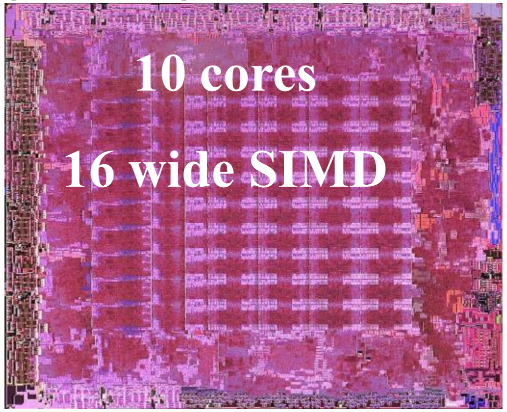
GMCH = graphics memory control hub, ICH = Input/output control hub

Microprocessor trends

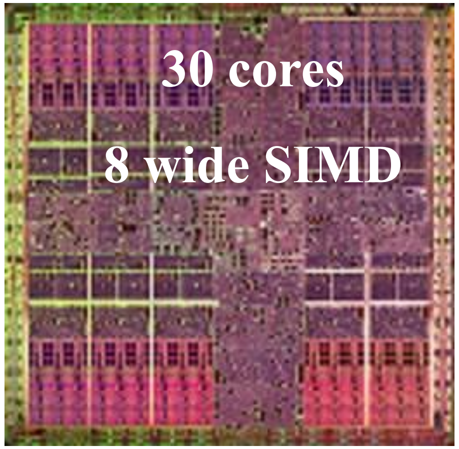
Individual processors are many core (and often heterogeneous) processors.



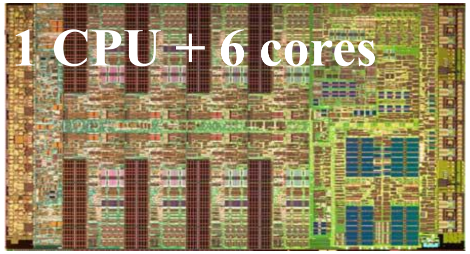
Intel SCC Processor



ATI RV770



NVIDIA Tesla C1060



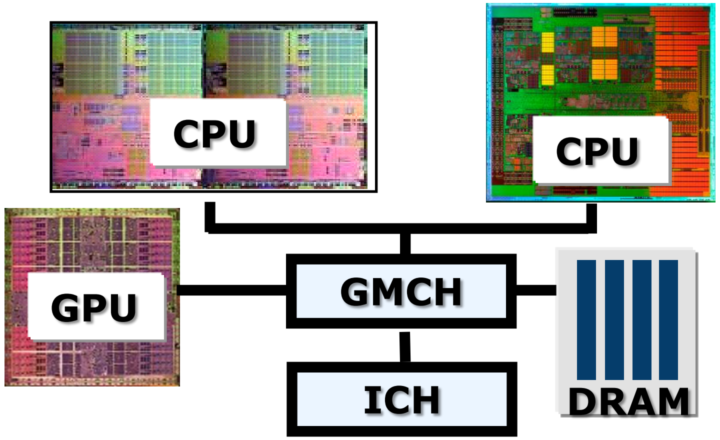
IBM Cell

The Heterogeneous many-core challenge: How are we to build a software ecosystem for the Heterogeneous many core platform?

3rd party names are the property of their owners.

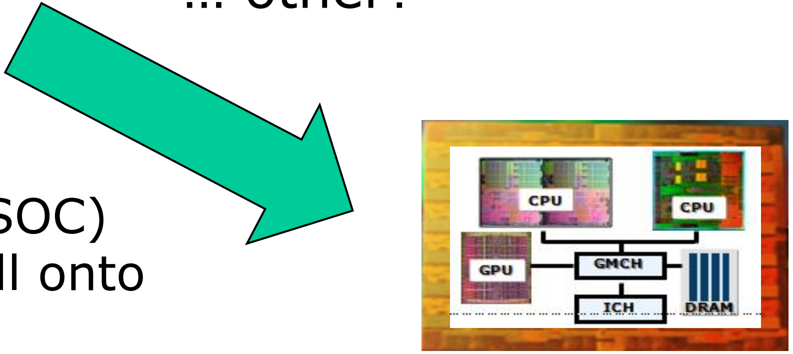
Thank you to Lee Howes (AMD) for help with the GPU core counts.

It’s a Heterogeneous world - part 2



- A modern platform includes:
  - CPU(s)
  - GPU(s)
  - DSP processors
  - … other?

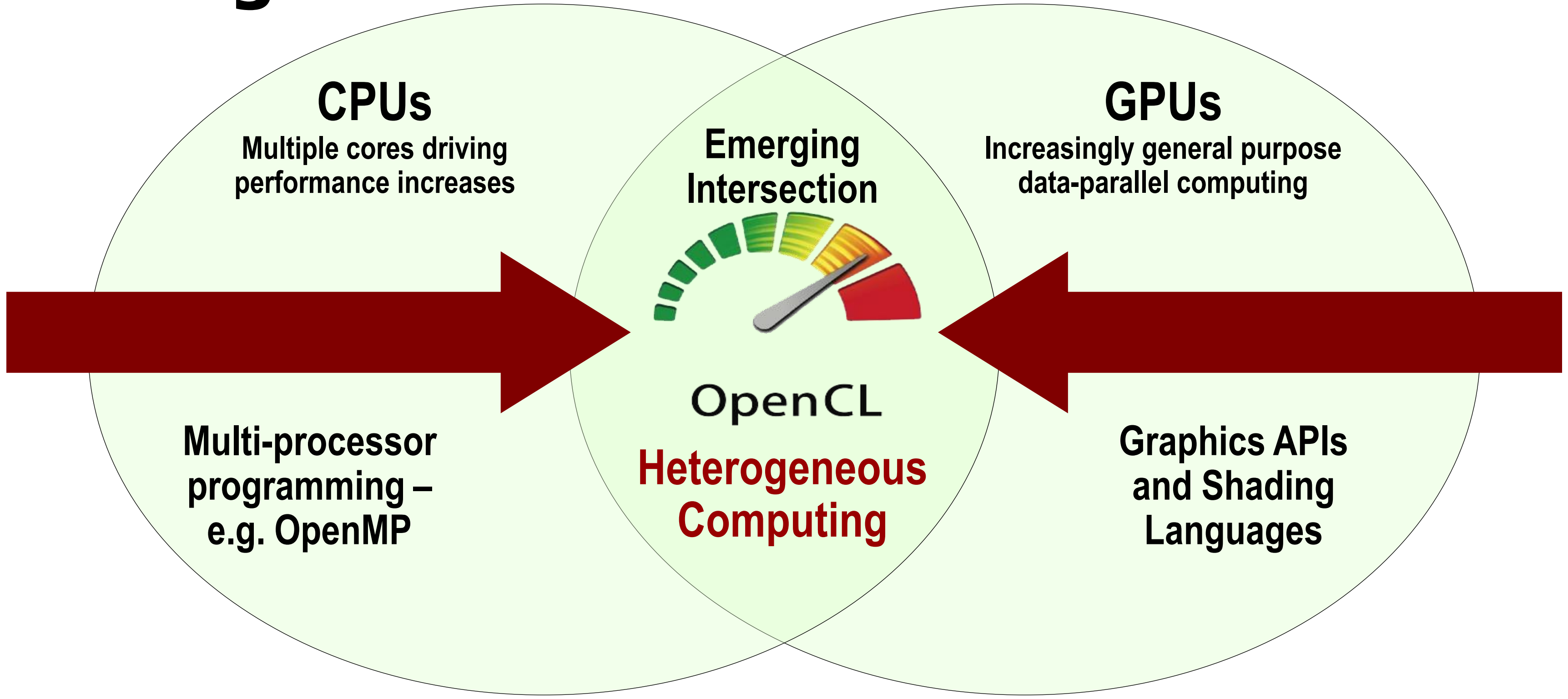
- And System on a Chip (SOC) trends are putting this all onto one chip



The future belongs to heterogeneous, many core SOC as the standard building block of computing

GMCH = graphics memory control hub, ICH = Input/output control hub SOC = system on a chip

Industry Standards for Programming Heterogeneous Platforms


CPUs
Multiple cores driving performance increases
Emerging Intersection
GPUs
Increasingly general purpose data-parallel computing
OpenCL
Multi-processor programming – e.g. OpenMP
Heterogeneous Computing
Graphics APIs and Shading Languages

OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

The BIG idea behind OpenCL

- **Replace loops with functions (a kernel) executing at each point in a problem domain.**
  - E.g., process a 1024 x 1024 image with one kernel invocation per pixel or 1024 x 1024 = 1,048,576 kernel executions

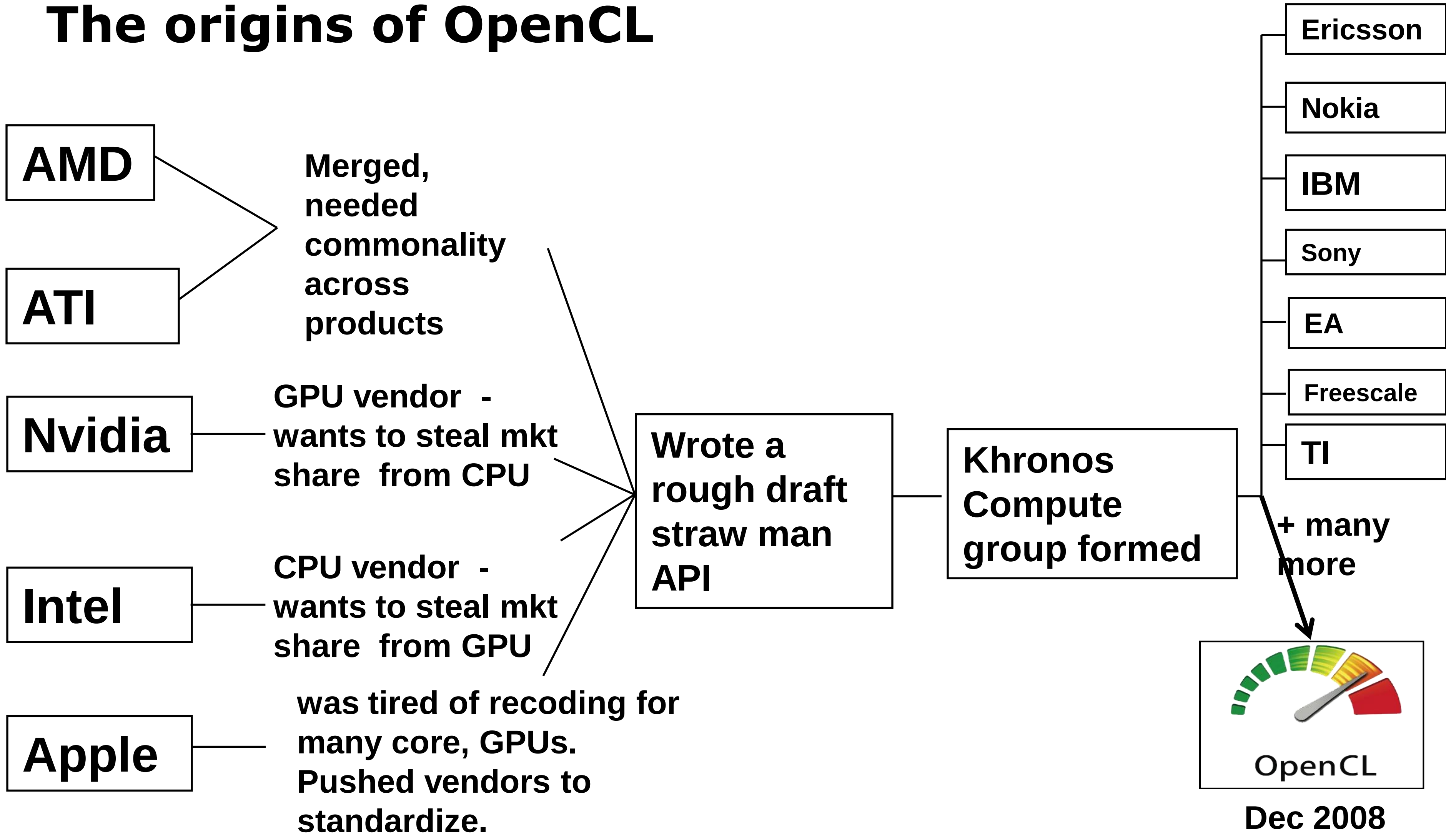
Traditional loops

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
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];

} // execute over "n" work-items
```

The origins of OpenCL



AMD

ATI

Merged, needed commonality across products

Nvidia

GPU vendor - wants to steal mkt share from CPU

Intel

CPU vendor - wants to steal mkt share from GPU

Apple

was tired of recoding for many core, GPUs. Pushed vendors to standardize.

Wrote a rough draft straw man API

Khronos Compute group formed

Ericsson

Nokia

IBM

Sony

EA

Freescale

TI

+ many more

OpenCL

Dec 2008

Third party names are the property of their owners.

OpenCL Working Group within Khronos

- **Diverse industry participation …**
  - Processor vendors, system OEMs, middleware vendors, application developers.
- **OpenCL became an important standard "on release" by virtue of the market coverage of the companies behind it.**

3DLABS SEMICONDUCTOR · ACTIVISION BLIZZARD · AMD · ARM · BROADCOM · EA

codeplay · ERICSSON · freescale semiconductor · GE · HI CORP. · IBM · intel

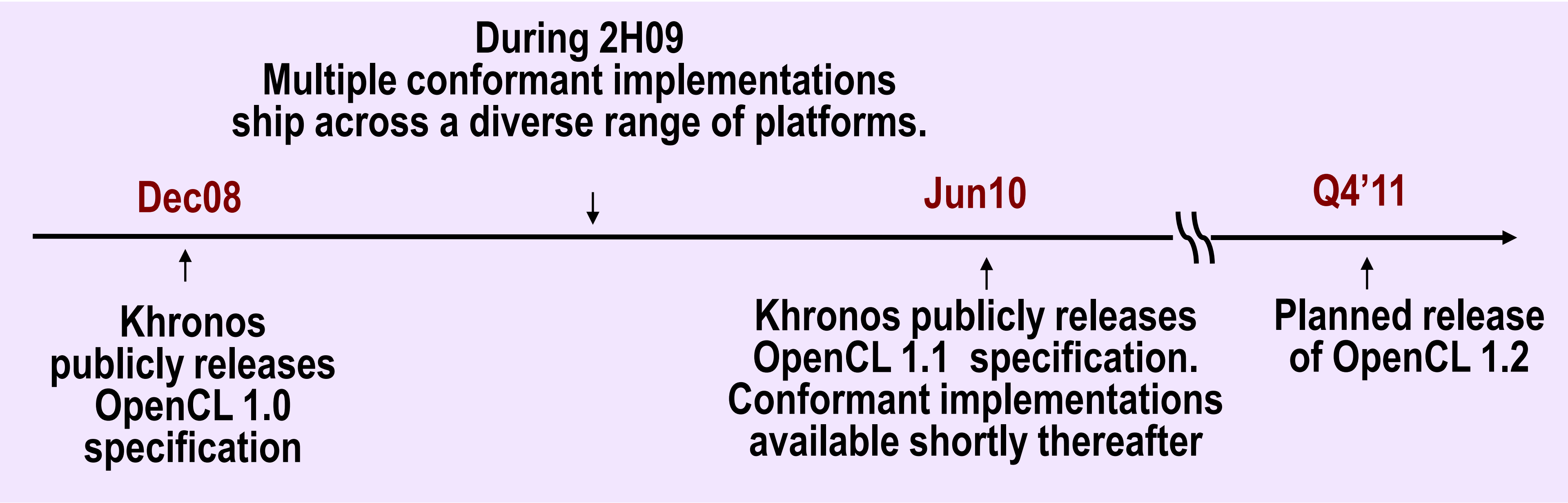
Imagination TECHNOLOGIES · Los Alamos NATIONAL LABORATORY · MOTOROLA · movidia · NOKIA · nVIDIA · QNX SOFTWARE SYSTEMS

RAPIDMIND · SAMSUNG · Seaweed SYSTEMS · TAKUMI · TEXAS INSTRUMENTS · UMEÅ UNIVERSITET

KHRONOS GROUP

OpenCL Timeline

- **Launched Jun’08 … 6 months from “strawman” to OpenCL 1.0.**
- **Rapid innovation to match pace of HW innovation**
  - 18 months from 1.0 to 1.1 … Goal: a new OpenCL every 18 months.
  - We are committed to backwards compatibility .. To protect your SW investments.


During 2H09
Multiple conformant implementations
ship across a diverse range of platforms.
Dec08
Jun10
Q4’11
Khronos publicly releases OpenCL 1.0 specification
Khronos publicly releases OpenCL 1.1 specification. Conformant implementations available shortly thereafter
Planned release of OpenCL 1.2

OpenCL: From cell phone to supercomputer

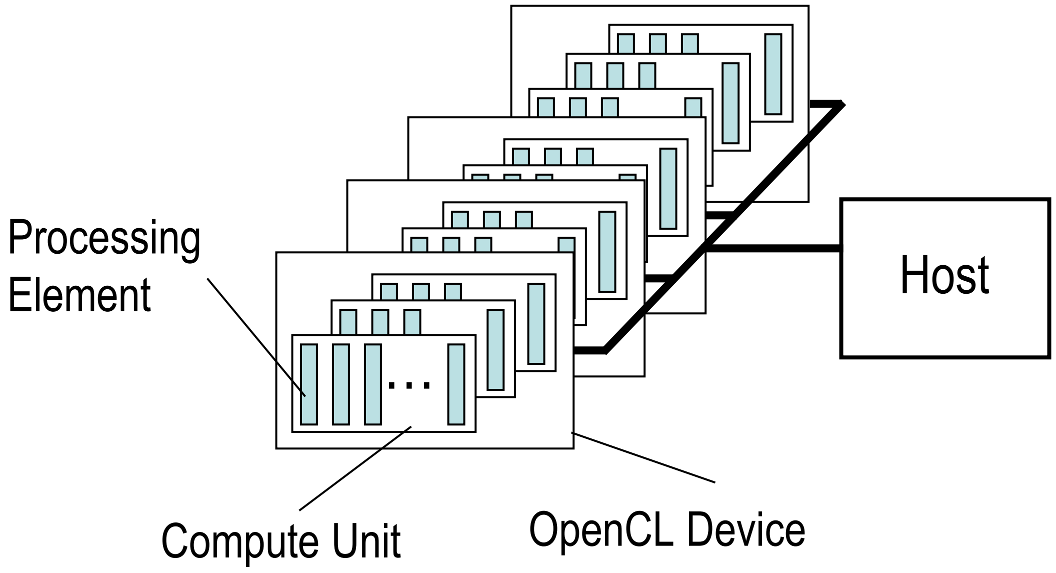
- **OpenCL Embedded profile for mobile and embedded silicon**
    - Relaxes some data type and precision requirements
    - Avoids the need for a separate "ES" specification
- **Khronos APIs provide computing support for imaging & graphics**
    - Enabling advanced applications in, e.g., Augmented Reality
- **OpenCL will enable parallel computing in new markets**
    - Mobile phones, cars, avionics

A camera phone with GPS processes images to recognize buildings and landmarks and provides relevant data from internet

Agenda

- Heterogeneous computing and the origins of OpenCL
- OpenCL overview
- Mapping OpenCL onto CPUs
- Exploring the spec with code: embarrassingly parallel
  - Vector addition: the basic platform layer
  - Matrix multiplication: kernel programming language
- Exploring the spec with code: dealing with dependencies
  - Optimizing matrix mul.: work groups and the memory model
  - Reduction: synchronization
- A survey of the rest of OpenCL

OpenCL Platform Model



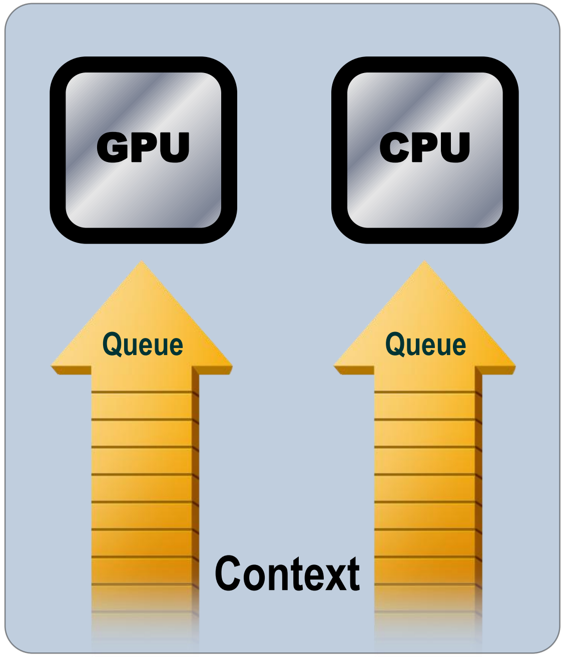
- **One <u>Host</u> + one or more <u>OpenCL Devices</u>**
  - Each OpenCL Device is composed of one or more <u>Compute Units</u>
    - Each Compute Unit is further divided into one or more <u>Processing Elements</u>

Context

- **What is an OpenCL Context:**
  - The environment within which the *kernels execute,*
  - *The domain in which* synchronization and memory management is defined.
- **The *context includes:***
  - *A set of devices,*
  - *The* memory accessible to those *devices*
  - *One or more command-queues used to schedule execution of a kernel(s) or operations on memory objects.*
- **Contexts are used to contain and manage the state of the “world” in OpenCL.**
  - Kernel execution commands
  - Memory commands - transfer or mapping of memory object data
  - Synchronization commands - constrains the order of commands

Command-Queues

- All commands to a device are submitted through a command-queue.
- Execution of the command-queue is guaranteed to be completed at sync points
- Each Command-queue points to a single device within a context.
- A single device can simultaneously be attached to multiple command queues.
- Multiple command-queues can be created to handle independent commands that don't require synchronization

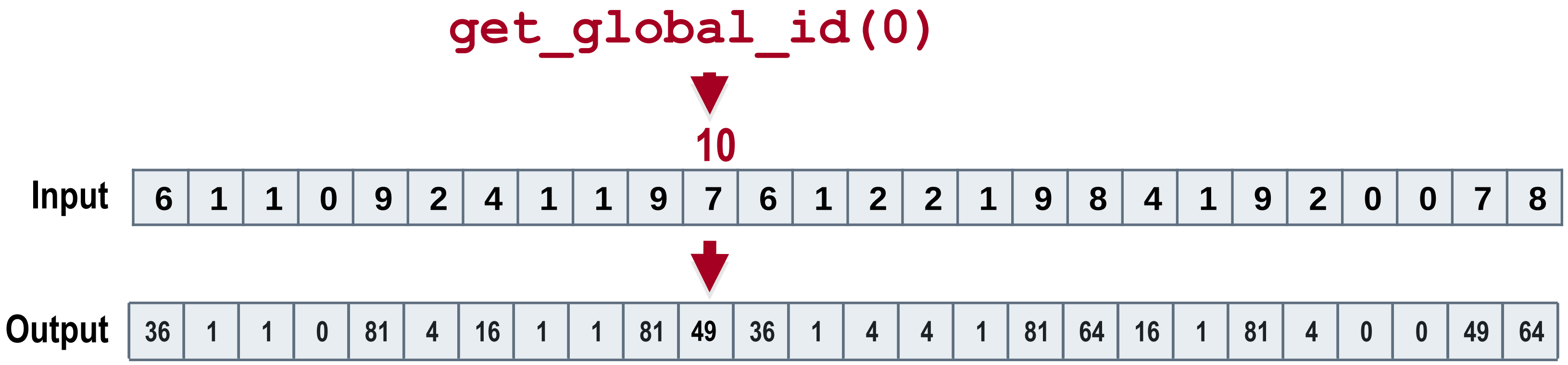


Commands Queued in-order. Can execute in-order or out-of order depending on the queue.

Execution model (kernels)

- OpenCL execution model … define a problem domain and execute an instance of a kernel for each point in the domain

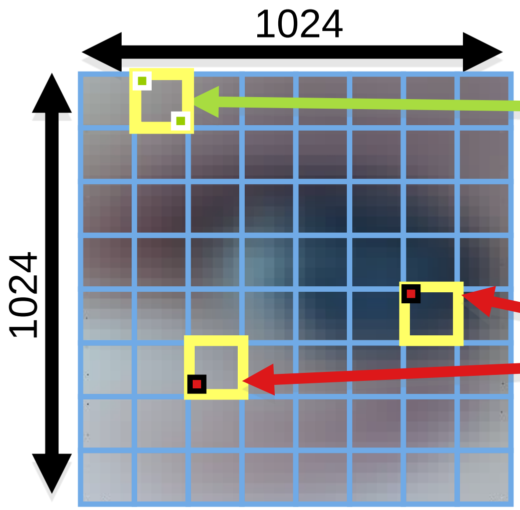
```
kernel void square(
        global float* input,
        global float* output)
{
   int i = get_global_id(0);
   output[i] = input[i] * input[i];
}
```



Input	6	1	1	0	9	2	4	1	1	9	7	6	1	2	2	1	9	8	4	1	9	2	0	0	7	8
Output	36	1	1	0	81	4	16	1	1	81	49	36	1	4	4	1	81	64	16	1	81	4	0	0	49	64

An N-dimension domain of work-items

- Global Dimensions: 1024 x 1024 (whole problem space)
- Local Dimensions: 128 x 128 (work group … executes together)

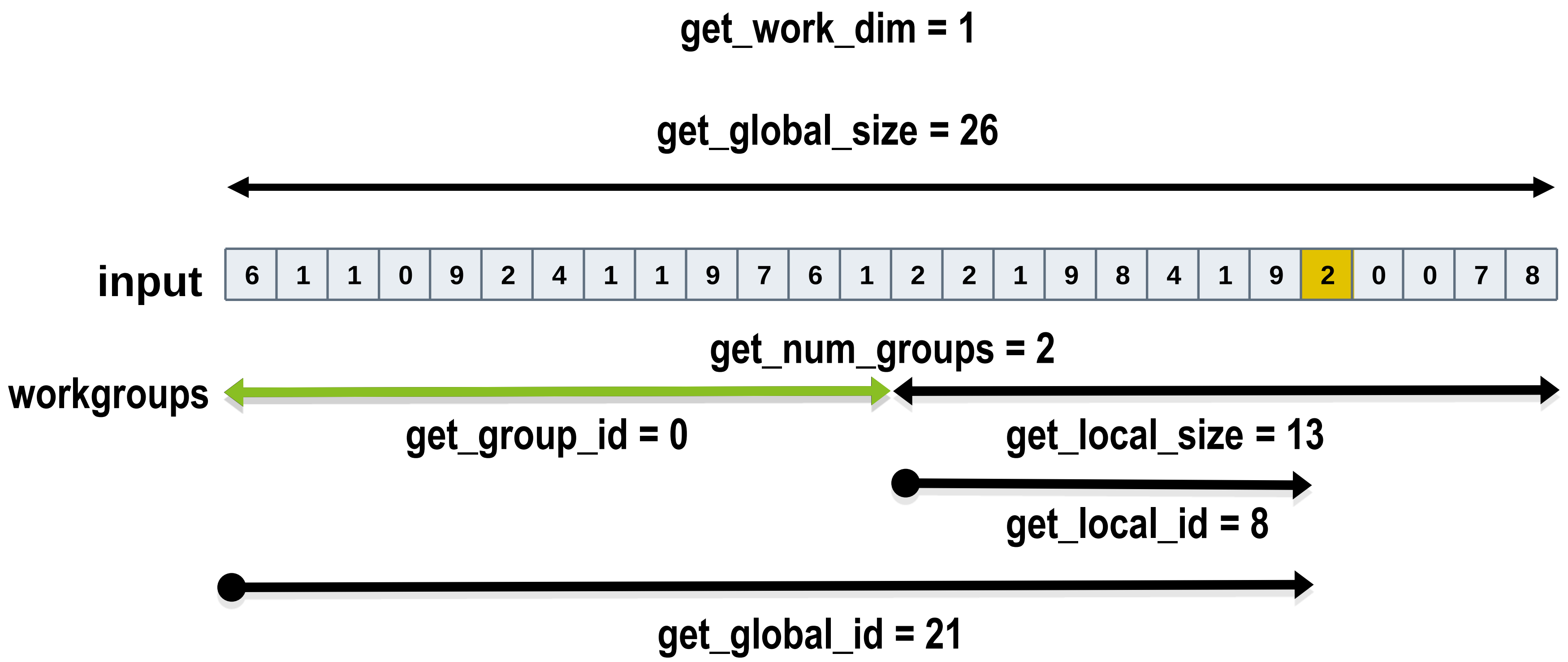


Synchronization between work-items possible only within workgroups: barriers and memory fences

Cannot synchronize outside of a workgroup

- Choose the dimensions that are “best” for your algorithm

Keeping track of work-items and work-groups

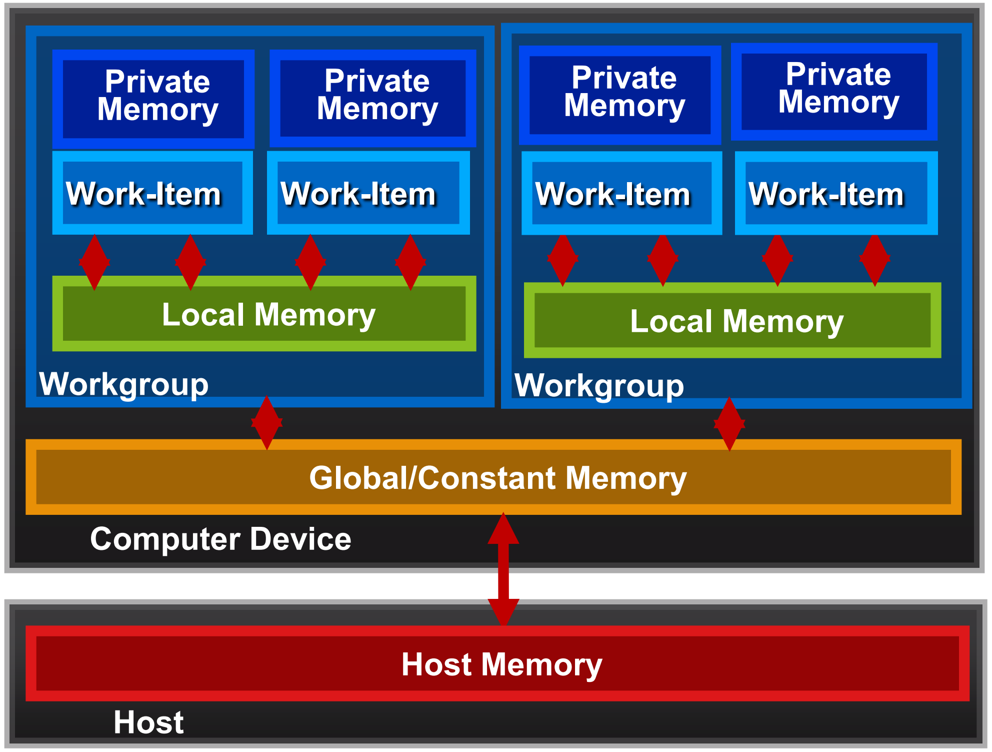


Kernel Execution

- **A command to execute a kernel must be enqueued to the command-queue**
  - *clEnqueueNDRangeKernel*()
    - Data-parallel execution model
    - Describes the ***index space*** for kernel execution
    - Requires information on NDRange dimensions and work-group size
  - *clEnqueueTask*()
    - Task-parallel execution model (multiple queued tasks)
    - Kernel is executed on a single work-item
  - *clEnqueueNativeKernel*()
    - Task-parallel execution model
    - Executes a native C/C++ function not compiled using the OpenCL compiler
    - This mode does not use a kernel object so arguments must be passed in

OpenCL Memory Model

- Private Memory
  - Per work-item
- Local Memory
  - Shared within a workgroup
- Global/Constant Memory
  - Visible to all workgroups
- Host Memory
  - On the CPU



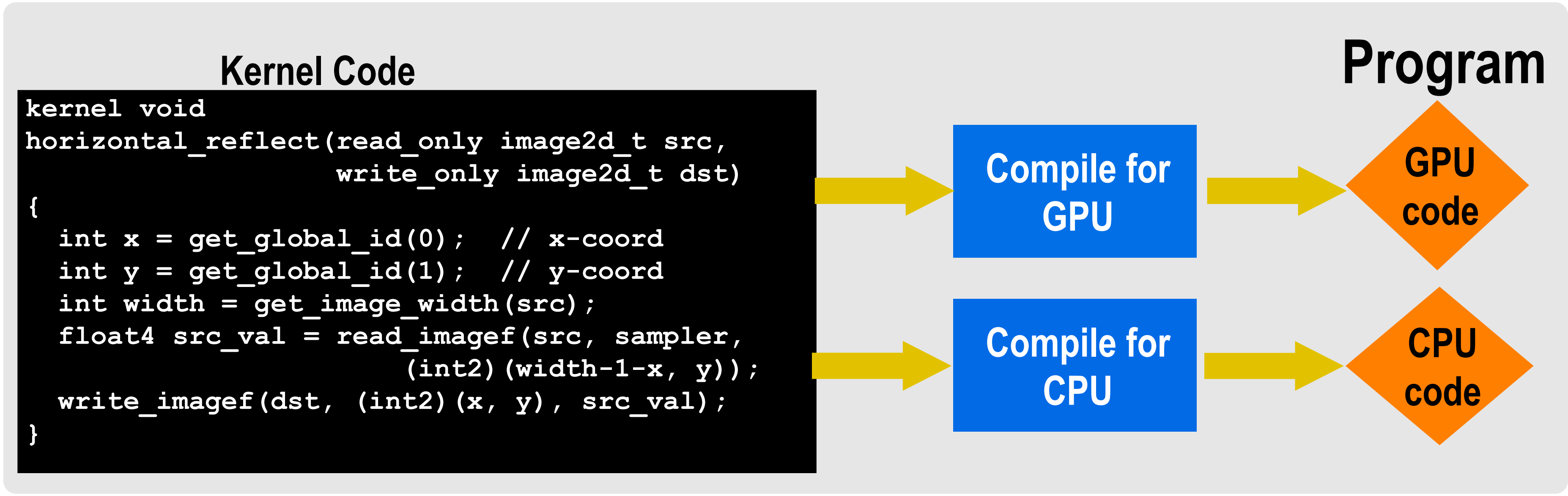
- **Memory management is explicit**
  You must move data from host -> global -> local *and* back

Memory Consistency

- **“OpenCL uses a relaxed consistency memory model; i.e.**
  - the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.”
- **Within a work-item:**
  - Memory has load/store consistency to its private view of memory
- **Within a work-group:**
  - Local memory is consistent between work-items at a barrier.
- **Global memory is consistent within a work-group, at a barrier, but not guaranteed across different work-groups**
- **Consistency of memory shared between commands (e.g. kernel invocations) are enforced through synchronization (barriers, events, in-order queue)**

Building Program objects

- **The program object encapsulates:**
  - A context
  - The program source/binary
  - List of target devices and build options
- **The Build process … to create a program object**
  - clCreateProgramWithSource()
  - clCreateProgramWithBinary()



```
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);
}
```

OpenCL C for Compute Kernels

- **Derived from ISO C99**
  - A few restrictions: recursion, function pointers, functions in C99 standard headers ...
  - Preprocessing directives defined by C99 are supported
- **Built-in Data Types**
  - Scalar and vector data types, Pointers
  - Data-type conversion functions: convert_type<_sat><_roundingmode>
  - Image types: image2d_t, image3d_t and sampler_t
- **Built-in Functions — Required**
  - work-item functions, math.h, read and write image
  - Relational, geometric functions, synchronization functions
- **Built-in Functions — Optional**
  - double precision, atomics to global and local memory
  - selection of rounding mode, writes to image3d_t surface

OpenCL C Language Highlights

- **Function qualifiers**
  - "__kernel" qualifier declares a function as a kernel
  - Kernels can call other kernel functions
- **Address space qualifiers**
  - __global, __local, __constant, __private
  - Pointer kernel arguments must be declared with an address space qualifier
- **Work-item functions**
  - Query work-item identifiers
    - get_work_dim(), get_global_id(), get_local_id(), get_group_id()
- **Synchronization functions**
  - Barriers - all work-items within a work-group must execute the barrier function before any work-item can continue
  - Memory fences - provides ordering between memory operations

OpenCL C Language Restrictions

- Pointers to functions are not allowed
- Pointers to pointers allowed within a kernel, but not as an argument
- Bit-fields are not supported
- Variable length arrays and structures are not supported
- Recursion is not supported
- Writes to a pointer of types less than 32-bit are not supported
- Double types are optional in OpenCL, but the key word is reserved

Vector Types

- Portable between different vector instruction sets
- Vector length of 2, 4, 8, and 16
- char2, ushort4, int8, float16, double2, …
- Endian safe
- Aligned at vector length
- Vector operations (elementwise) and built-in functions

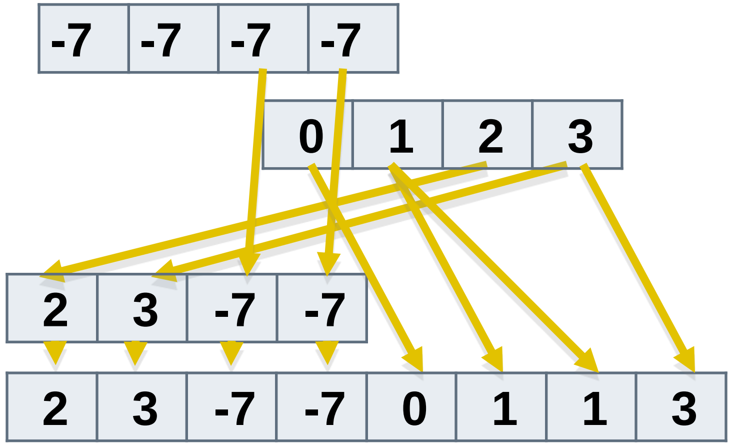
Vector Operations

- Vector literal

```
int4 vi0 = (int4) -7;

int4 vi1 = (int4)(0, 1, 2, 3);
```

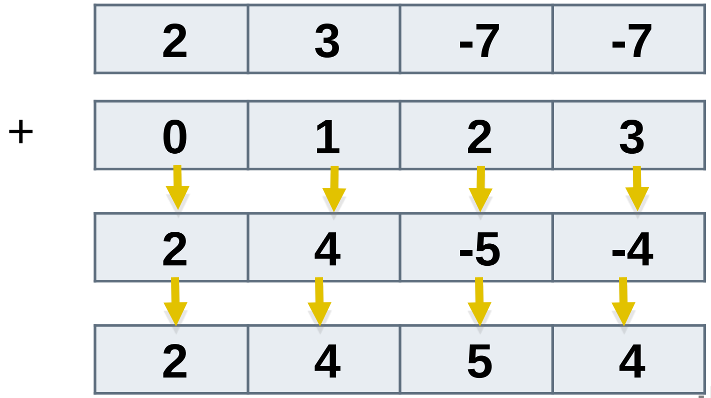
- Vector components

```
vi0.lo = vi1.hi;

int8 v8 = (int8)(vi0, vi1.s01, vi1.odd);
```

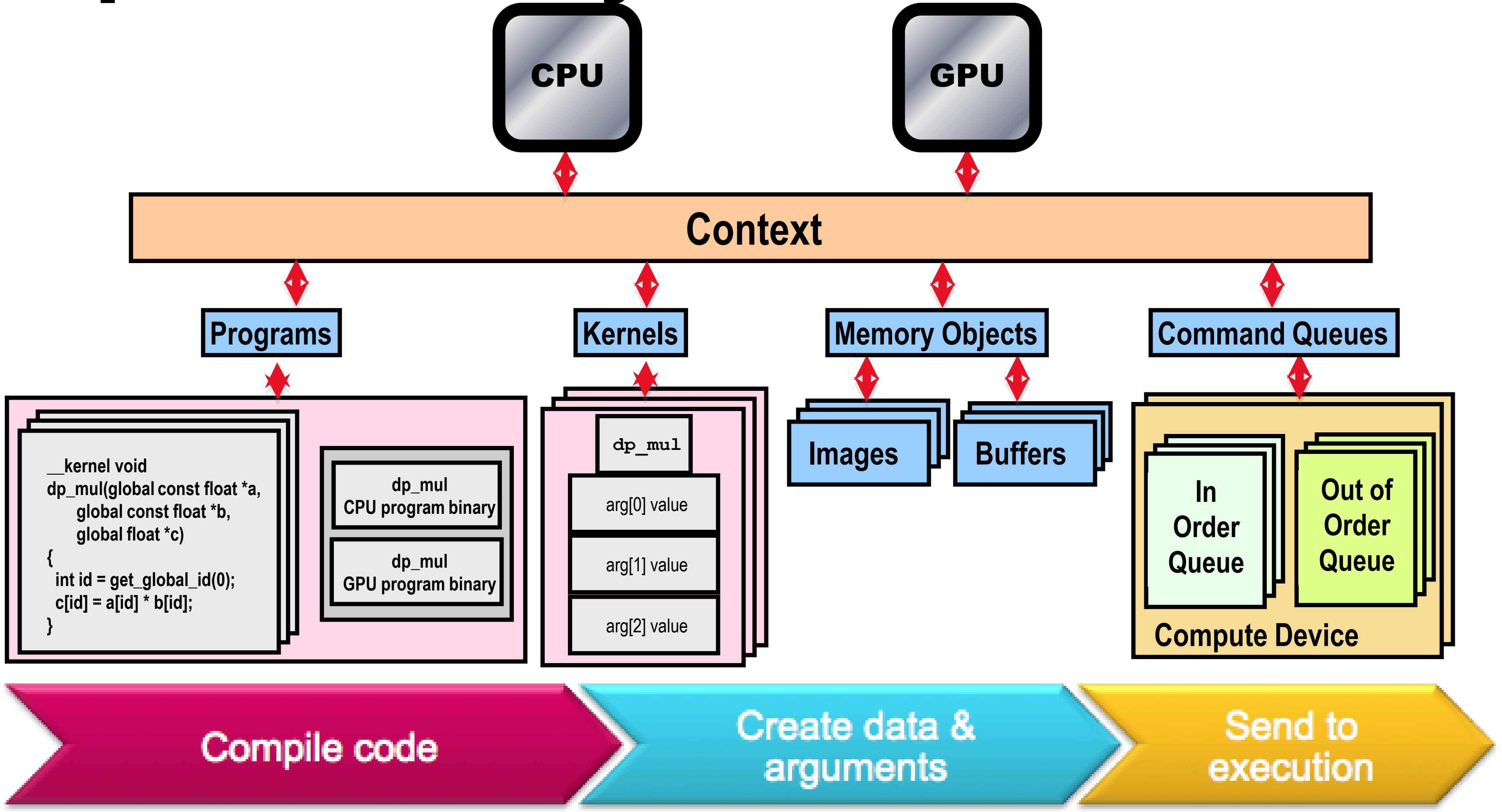


- Vector ops

```
vi0 += vi1;


vi0 = abs(vi0);
```



OpenCL summary



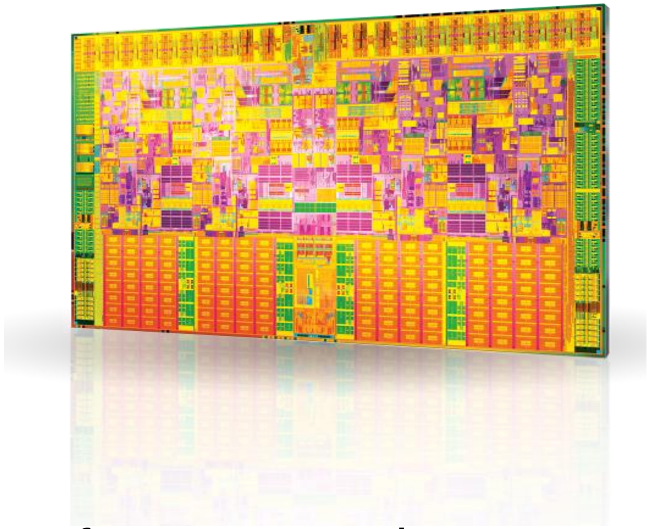
```
__kernel void
dp_mul(global const float *a,
      global const float *b,
      global float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
```

Agenda

- Heterogeneous computing and the origins of OpenCL
- OpenCL overview
- ➡ Mapping OpenCL onto CPUs
- Exploring the spec with code: embarrassingly parallel
  - Vector addition: the basic platform layer
  - Matrix multiplication: kernel programming language
- Exploring the spec with code: dealing with dependencies
  - Optimizing matrix mul.: work groups and the memory model
  - Reduction: synchronization
- A survey of the rest of OpenCL

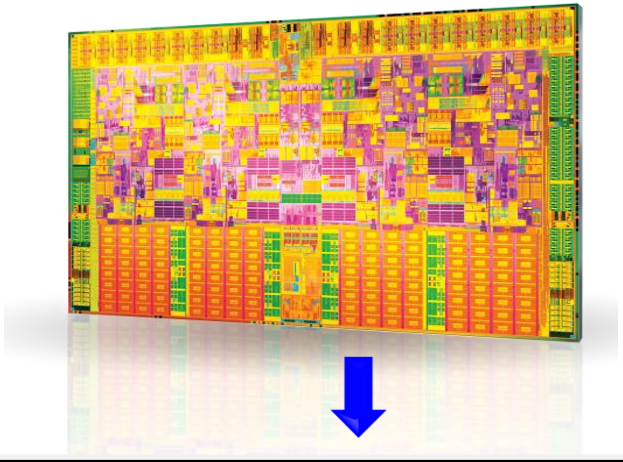
Heterogeneous computing and the CPU

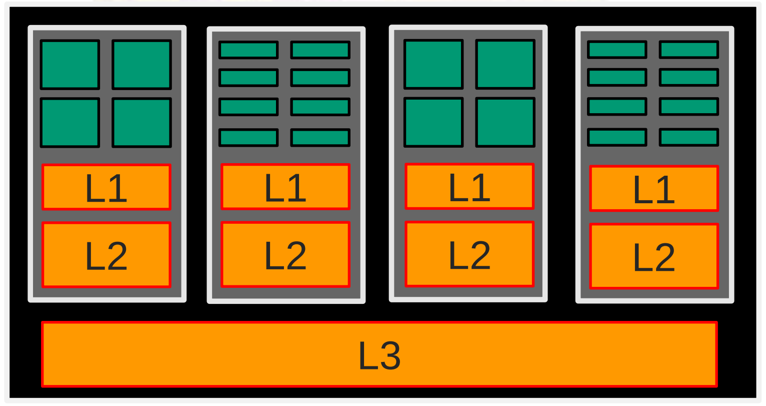
- **Challenge … how do you exploit the performance of modern CPU's**
  - Multi-Core / SMT
  - Vector Units

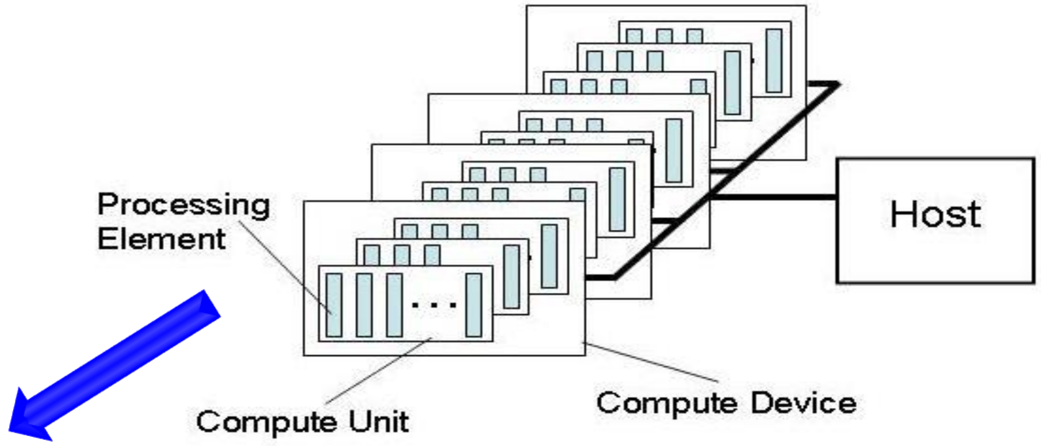


- Hypothesis: OpenCL is an effective platform for programming a CPU
  - OpenCL can handle multiple cores, multiple CPUs, and vector units.
  - Uses a single programming model which simplifies programming.
  - OpenCL provides a portable interface to vector instructions (SSE, AVX, etc).
  - The long term hope … Performance portable across CPU product lines and eventually between CPU and GPUs.

OpenCL view of Core™ i7







Core™ i7 975

- **8 Compute Units**
  - **Quad Core + SMT**
- **4/8/16 Processing Elements per Unit**
  - **128bit XMM registers**
  - **Data type determines # of elements…**
- **(32K L1 + 256K L2) Per Core, 8M L3 Shared**

OpenCL’s Two Styles of Data-Parallelism

- **Explicit SIMD data parallelism:**
  - The kernel defines one stream of instructions
  - Parallelism from using wide vector types
  - Size vector types to match native HW width
  - Combine with task parallelism to exploit multiple cores.

- **Implicit SIMD data parallelism (i.e. shader-style):**
  - Write the kernel as a “scalar program”
  - Kernel automatically mapped onto the lanes of the SIMD-compute-resources and cores by the compiler/runtime/hardware.

Both approaches are viable CPU options

Explicit SIMD data parallelism

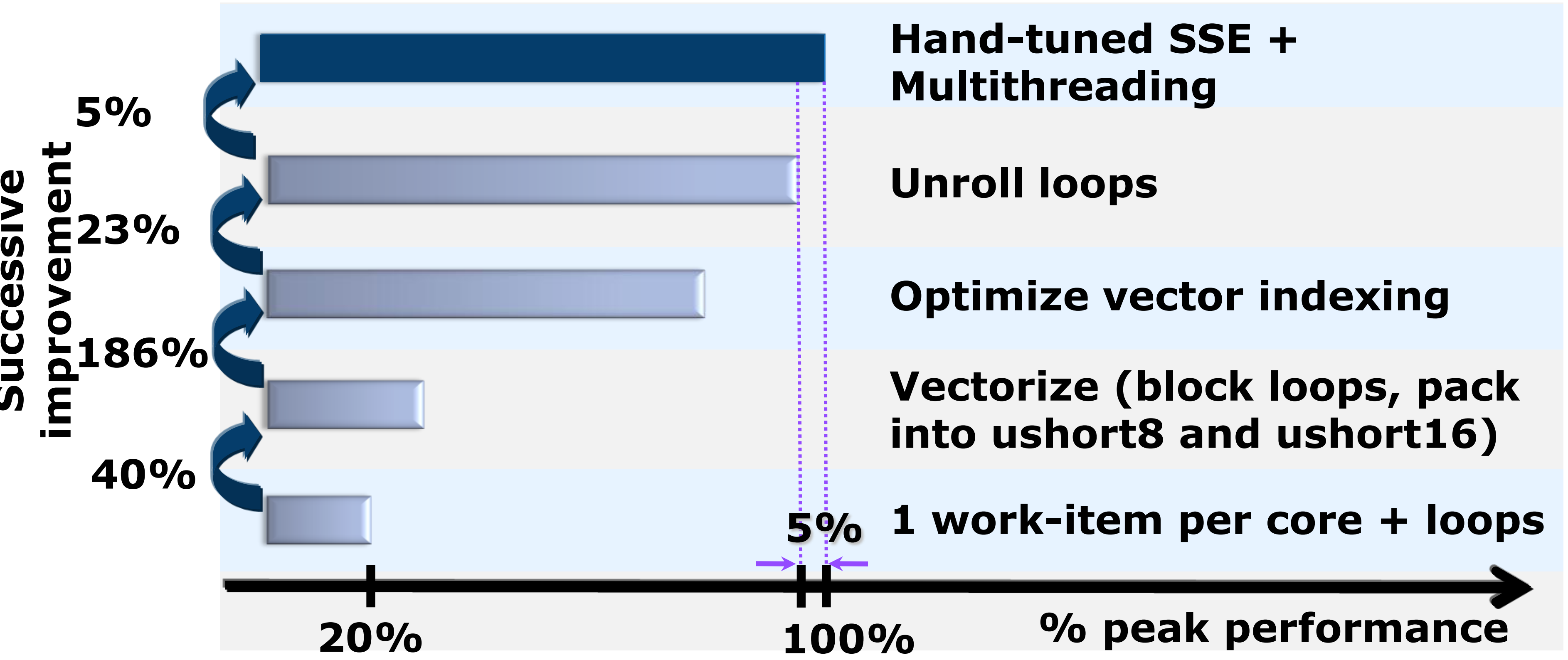
- **OpenCL as a portable interface to vector instruction sets.**
  - Block loops and pack data into vector types (float4, ushort16, etc).
  - Replace scalar ops in loops with blocked loops and vector ops.
  - Unroll loops, optimize indexing to match machine vector width

```
float a[N], b[N], c[N];
for (i=0; i<N; i++)
    c[i] = a[i]*b[i];

<<< the above becomes >>>>

float4 a[N/4], b[N/4], c[N/4];
for (i=0; i<N/4; i++)
     c[i] = a[i]*b[i];
```

Explicit SIMD data parallelism means you tune your code to the vector width and other properties of the compute device

Explicit SIMD data parallelism: Case Study

- Video contrast/color optimization kernel on a dual core CPU.

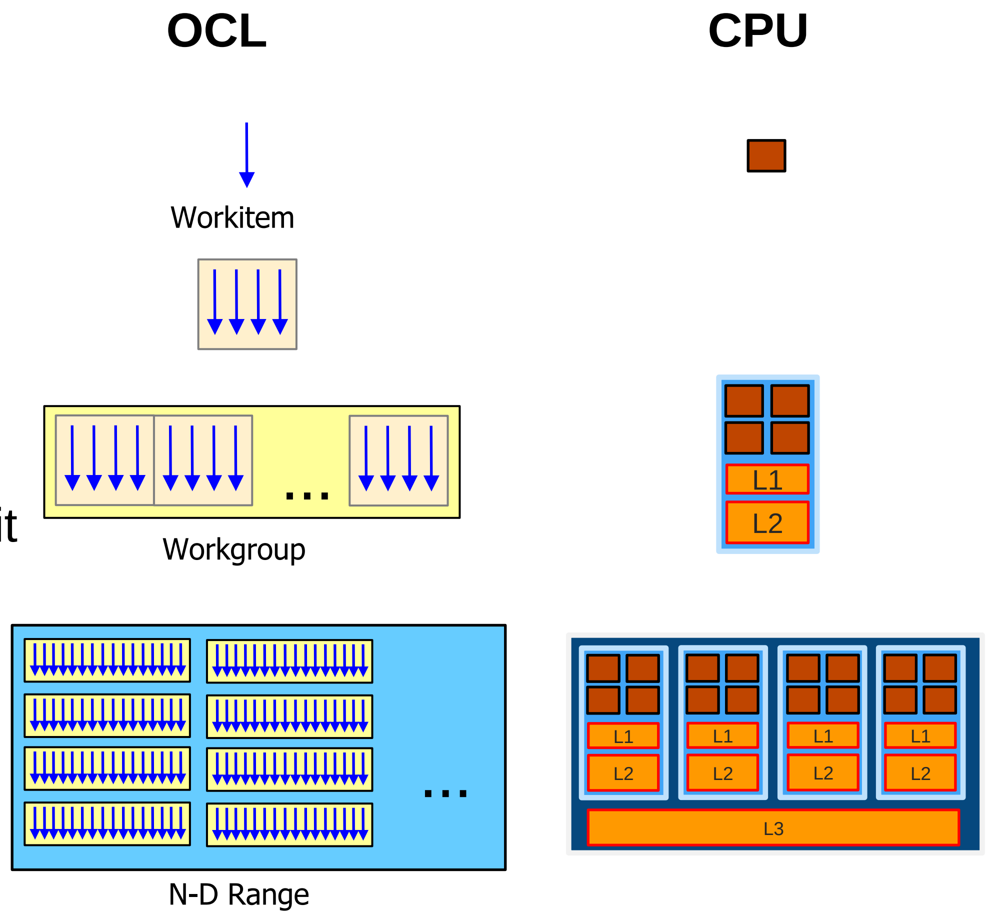


Good news: OpenCL code 95% of hand-tuned SSE/MT perf.

Bad news: New platform, redo all those optimizations.

Implicit Data Parallelism on the CPU

- One workitem runs on a single SSE lane
- Workitems are packed to SSE registers as part of the OpenCL Compilation process
- Workgroup is executed on a compute unit (HW Thread)
- Kernel is executed over an N-D Range, which is divided to workgroups
- Several Workgroups run concurrently on all compute unit (HW threads)



Implicit vs. explicit SIMD: N-body Simulation

Given *N* bodies with an initial position x_i and velocity v_i for, the force f_{ij} on body *i* caused by body *j* is given by following (G is gravity):

$$\mathbf{f}_{ij} = G\frac{m_i m_j}{\left\|\mathbf{r}_{ij}\right\|^2} \cdot \frac{\mathbf{r}_{ij}}{\left\|\mathbf{r}_{ij}\right\|}, \qquad \mathbf{F}_i = \sum_{\substack{1\le j\le N \\ j\ne i}} \mathbf{f}_{ij}$$

where m_i and m_j are the masses of bodies *i* and *j*, respectively; $r_{ij} = x_j - x_i$

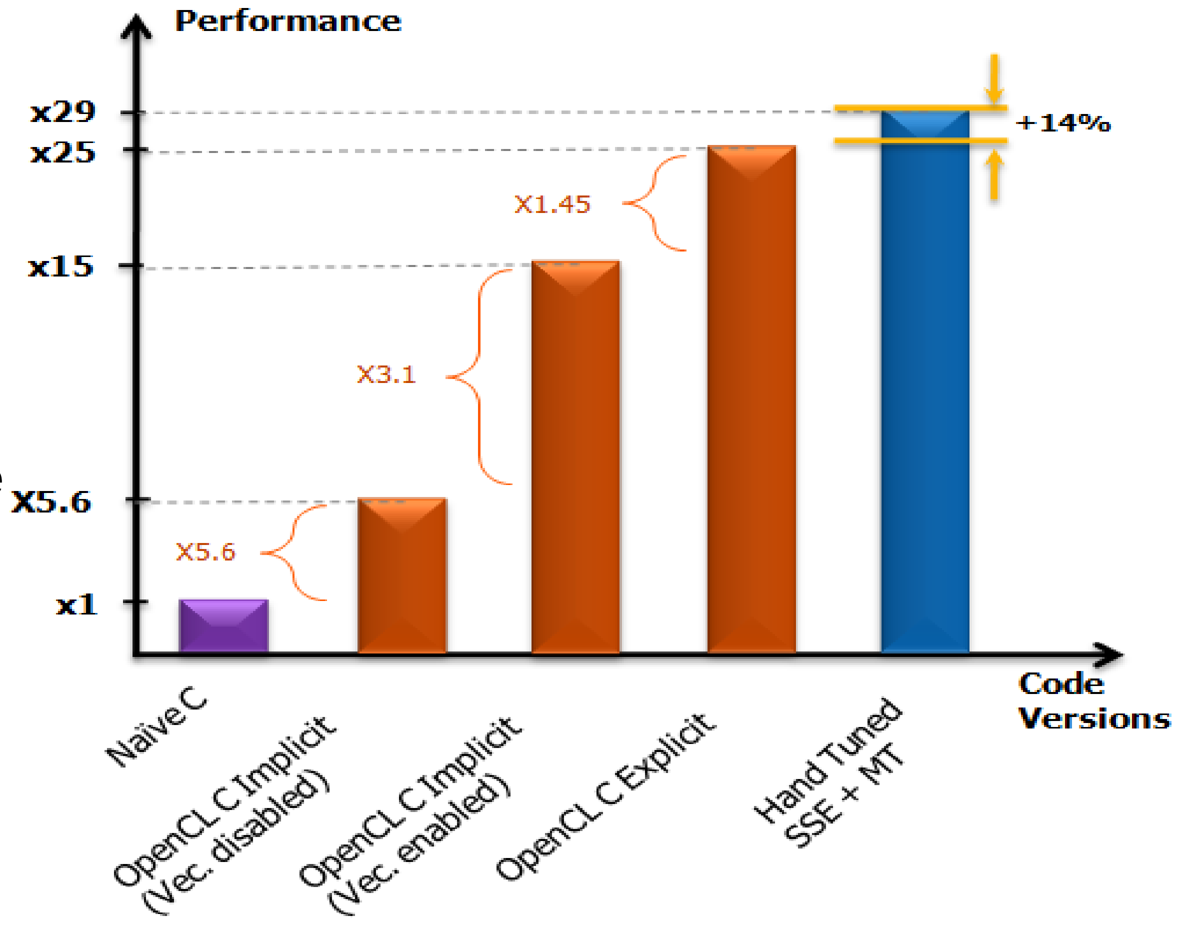
The acceleration is computed as $a_i = F_i/m_i$

Note: we are well aware that for values of N found in realistic systems, this direct sum algorithm (order N squared) is inferior to well known order N or order N $\log_2$ N algorithms. As with our matrix multiplication examples, this simple algorithm was selected for its pedagogical value.

NBody Performance

Results from Intel's internal OpenCL implementation:*

- **Implicit Data Parallelism**
  - "shader-style" code
  - Benefit from multi-core/SMT
- **Explicit Data-Parallelism**
  - Hand tuned OpenCL C code
  - OpenCL Explicit version is x25 faster than Naïve C *
  - Explicit version is only 14% slower than highly optimized code *



* Results measured on Core^{TM} i7 975, 3.3 GHz, 6GB DDR3
Results depends on the algorithm/code running

Agenda

- **Heterogeneous computing and the origins of OpenCL**
- **OpenCL overview**
- **Mapping OpenCL onto CPUs**
- **Exploring the spec with code: embarrassingly parallel**
  - Vector addition: the basic platform layer
  - Matrix multiplication: kernel programming language
- **Exploring the spec with code: dealing with dependencies**
  - Optimizing matrix mul.: work groups and the memory model
  - Reduction: synchronization
- **A survey of the rest of OpenCL**

Example: vector addition

- **The “hello world” program of data parallel programming is a program to add two vectors**

    C[i] = A[i] + B[i]   for i=1 to N

- **For the OpenCl solution, there are two parts**
    - Kernel code
    - Host code

Vector Addition - Kernel

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

Vector Addition - Host

- **The host program … the code that runs on the host to:**
  - Setup the environment for the OpenCL program
  - Create and mange kernels
- **5 simple steps in a basic Host program**
  1. Define the platform … platform = devices+context+queues
  2. Create and Build the program (dynamic library for kernels)
  3. Setup memory objects
  4. Define kernel (attach kernel function to arguments)
  5. Submit commands … move memory objects and execute kernels

Our goal is extreme portability so we expose everything (i.e. we are a bit verbose). But most of a host code is the same from one application to the next … the re-use makes the verbosity a non-issue

1. Define the platform

- Grab the first available Platform:

```
err = clGetPlatformIDs(1, &firstPlatformId, &numPlatforms);
```

- Use the first GPU device the platform provides

```
    err = clGetDeviceIDs(firstPlatformId, CL_DEVICE_TYPE_GPU, 1,
                                                    &device_id, NULL);
```

- Create a simple context with a single device

```
context = clCreateContext(firstPlatformId, 1, &device_id, NULL,
                                          NULL, &err);
```

- Create a simple command queue to feed our compute device

```
 commands = clCreateCommandQueue(context, device_id, 0, &err);
```

2. Create and Build the program

- Define source code for the kernel-program as a string literal (great for toy programs) or read from a file (common in real apps).
- Build the program object:

```
program = clCreateProgramWithSource(context, 1,
            (const char **) & KernelSource, NULL, &err);
```

- Compile the program to create a “dynamic library” from which specific kernels can be pulled:

```
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

- Fetch and print error messages (if(err != CL_SUCCESS))

```
size_t len;          char buffer[2048];
clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer),
                                                                  buffer, &len);
printf("%s\n", buffer);
```

3. Setup Memory Objects

- **For vector addition, 3 memory objects … one for each input vector (A and B) and one for the output vector (C).**
- **Create input vectors and assign values on the host:**

```
float     a_data[LENGTH], b_data[LENGTH], c_res [LENGTH];
for(i = 0; i < count; i++){
      a_data[i] = rand() / (float)RAND_MAX;
      b_data[i] = rand() / (float)RAND_MAX;
}
```

- **Define OpenCL memory objects**

```
a_in   = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(float) * count, NULL, NULL);
b_in   = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(float) * count, NULL, NULL);
c_out  = clCreateBuffer(context,  CL_MEM_WRITE_ONLY, sizeof(float) * count, NULL, NULL);
```

4. Define the kernel

- Create kernel object from the kernel function “vadd”

```
kernel = clCreateKernel(program, "vadd", &err);
```

- Attach arguments to the kernel function “vadd” to memory objects

```
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_in);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_in);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &c_out);
err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &count);
```

5. Submit commands

- Write Buffers from host into global memory (as non-blocking operations)

```
err = clEnqueueWriteBuffer(commands, a_in, CL_FALSE, 0,
                    sizeof(float) * count, a_data, 0, NULL, NULL);
err = clEnqueueWriteBuffer(commands, b_in, CL_FALSE, 0,
                    sizeof(float) * count, b_data, 0, NULL, NULL);
```

- Enqueue the kernel for execution (note: in-order queue so this is OK)

```
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL,
                              &global, &local, 0, NULL, NULL);
```

- Read back the result (as a blocking operation). Use the fact that we have an in-order queue which assures the previous commands are done before the read begins.

```
err = clEnqueueReadBuffer( commands, c_out, CL_TRUE, 0,
                    sizeof(float) * count, c_res, 0, NULL, NULL );
```

Vector Addition - Host Program

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
   CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
                                        NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
   devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0],
   0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA,
                                            NULL);}
memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
                                            NULL);
memobjs[2] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                             sizeof(cl_float)*n, NULL,
                                            NULL);
// create the program
program = clCreateProgramWithSource(context, 1,
   &program_source, NULL, NULL);
```

```
// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL,
                                             NULL);

// create the kernel
kernel = clCreateKernel(program, “vec_add”, NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                  sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
   NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
   CL_TRUE, 0, n*sizeof(cl_float), dst, 0, NULL, NULL);
```

Vector Addition - Host Program

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
```

Define platform and queues

```
// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
                                        NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
   devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0],
   0, NULL);
```

Define Memory objects

```
                                                 READ_ONLY |
   CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA,
   NULL);}
memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
   NULL);
memobjs[2] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                            sizeof(cl_float)*n, NULL,
   NULL);
// create the program
program =
   &progra
```

Create the program

```
//
er                                           , NULL, NULL,
   NULL);
```

Build the program

```
/
k                                                  LL);
```

Create and setup kernel

```
// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                  sizeof(cl_mem));
// set work-item dimensions
global_w

// execu
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
   NULL, global_work_size, NULL, 0, NULL, NULL);
```

Execute the kernel

```
// rea
err =                                              TRUE,
   0,
```

Read results on the host

It’s complicated, but most of this is “boilerplate” and not as bad as it looks.

Agenda

- Heterogeneous computing and the origins of OpenCL
- OpenCL overview
- Mapping OpenCL onto CPUs
- Exploring the spec with code: embarrassingly parallel
  - Vector addition: the basic platform layer
  - → Matrix multiplication: kernel programming language
- Exploring the spec with code: dealing with dependencies
  - Optimizing matrix mul.: work groups and the memory model
  - Reduction: synchronization
- A survey of the rest of OpenCL

Linear Algebra

- **Definition:**
  - The branch of mathematics concerned with the study of vectors, vector spaces, linear transformations, and systems of linear equations.
- **Example: Consider the following system of linear equations**

$$\begin{aligned} x + 2y + z &= 1 \\ x + 3y + 3z &= 2 \\ x + y + 4z &= 6 \end{aligned}$$

  - This system can be represented in terms of vectors and a matrix as the classic "Ax = b" problem.

$$\begin{pmatrix} 1 & 2 & 1 \\ 1 & 3 & 3 \\ 1 & 1 & 4 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 6 \end{pmatrix}$$

Solving Ax=b

- **LU Decomposition:**
  - transform a matrix into the product of a lower triangular and upper triangular matrix. It is used to solve a linear system of equations.

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 1 \\ 1 & 3 & 3 \\ 1 & 2 & 4 \end{pmatrix}$$

$$L \cdot U = A$$

- Solving for x — Given a problem Ax=b
  - Ax=b — LUx=b
  - $Ux=(L^{-1})b$ — $x = (U^{-1})L^{-1}b$

Linear transformations

- **A matrix, A ∊ R^{MxP} multiplies a vector, x ∊ R^{P} to define a linear transformation of that vector into R^{m}.**
- **Matrix multiplication defines the composition of two linear transformations on that vector space**

Compute C = A*B where

$C \in R^{NxM}$

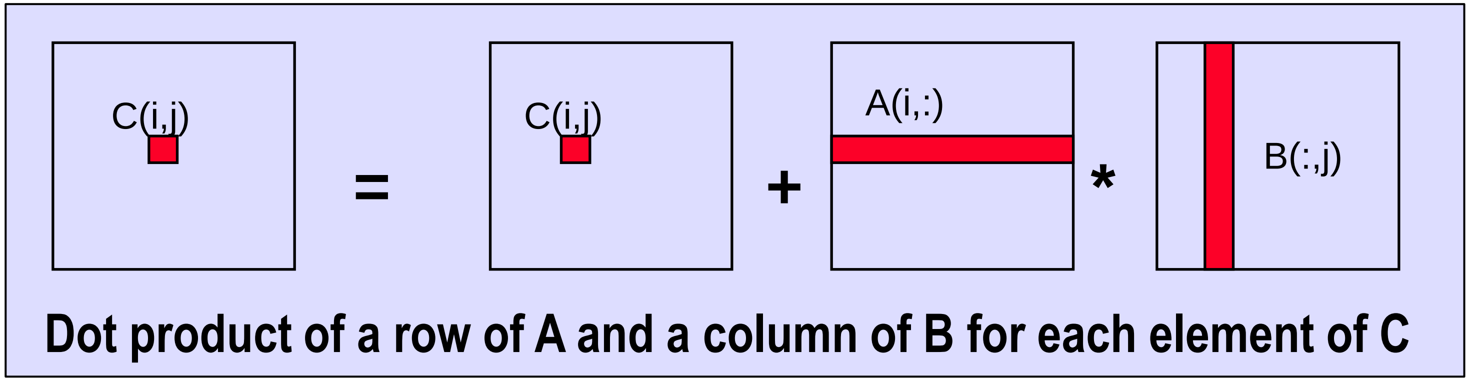
$A \in R^{NxP}$

$B \in R^{PxM}$

- Matrix multiplication is the core building block of Linear Algebra

Matrix Multiplication: Sequential code

```
void mat_mul(int Mdim, int Ndim, int Pdim, float *A, float *B, float *C)
{
  int i, j, k;
  for (i=0; i<Ndim; i++){
    for (j=0; j<Mdim; j++){
      for(k=0;k<Pdim;k++){      //C(i,j) = sum(over k) A(i,k) * B(k,j)
        C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
      }
    }
  }
}
```



Dot product of a row of A and a column of B for each element of C

Matrix Multiplications Performance

- **Results on an Apple laptop (OS X Snow Leopard) with an Intel CPU. Gcc compiler … no optimization.**

Case	MFLOPS
CPU: Sequential C (not OpenCL)	167

Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

Note: The goal of this study is to explore features of the OpenCL API. If we want high performance matrix multiplication we'd use the OpenCL native function interface to call MKL.

Matrix Multiplication: OpenCL kernel (1/4)

```
void mat_mul(int Mdim, int Ndim, int Pdim, float *A, float *B, float *C)
{
   int i, j, k;
   for (i=0; i<Ndim; i++){
      for (j=0; j<Mdim; j++){
         for(k=0;k<Pdim;k++){      //C(i,j) = sum(over k) A(i,k) * B(k,j)
            C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
         }
      }
   }
}
```

Matrix Multiplication: OpenCL kernel (2/4)

```
~~void mat_mul(~~
  ~~int Mdim, int Ndim, int Pdim,~~
  ~~float *A, float *B, float *C)~~
{
  int i, j, k;
  for (i=0; i<Ndim; i++){
    for (j=0; j<Mdim; j++){
      for(k=0;k<Pdim;k++){      //C(i,j) = sum(over k) A(i,k) * B(k,j)
        C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
      }
    }
  }
}
```

```
__kernel mat_mul(
  const int Mdim, const int Ndim, const int Pdim,
  __global float *A, __global float *B, __global float *C)
```

Mark as a kernel function and specify memory qualifiers

Matrix Multiplication: OpenCL kernel (3/4)

```
__kernel mat_mul(
   const int Mdim, const int Ndim, const int Pdim,
 __global float *A, __global float *B, __global float *C)
{
   int i, j, k;
   ~~for (i=0; i<Ndim; i++){~~
      ~~for (j=0; j<Mdim; j++){~~
         for(k=0;k<Pdim;k++){      //C(i,j) = sum(over k) A(i,k) * B(k,j)
            C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
         }
      ~~}~~
   ~~}~~
}
```

```
i = get_global_id(0);
j = get_global_id(1);
```

Remove outer loops and set work item coordinates

Matrix Multiplication: OpenCL kernel (4/4)

```
__kernel mat_mul(
   const int Mdim, const int Ndim, const int Pdim,
 __global float *A, __global float *B, __global float *C)
{
   int i, j, k;
  i = get_global_id(0);
  j = get_global_id(1);
        for(k=0;k<Pdim;k++){      //C(i,j) = sum(over k) A(i,k) * B(k,j)
           C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
        }


}
```

Matrix Multiplication: OpenCL kernel

Rearrange a bit and use a local scalar for intermediate C element values (a common optimization in Matrix Multiplication functions)

```
__kernel mmul(
    const int Mdim,
    const int Ndim,
    const int Pdim,
    __global float* A,
    __global float* B,
    __global float* C)
{
    int k;
    int i = get_global_id(0);
    int j = get_global_id(1);
    float tmp;
    tmp = 0.0;
      for(k=0;k<Pdim;k++)
        tmp += A[i*Ndim+k] * B[k*Pdim+j];
     C[i*Ndim+j] = tmp;
}
```

Matrix Multiplication host program

```
#include "mult.h"
int main(int argc, char **argv)
{
   float *A, *B, *C;
   int Mdim, Ndim, Pdim;
   int err, szA, szB, szC;
   size_t global[DIM];
   size_t local[DIM];
   cl_device_id device_id;
   cl_context context;
   cl_command_queue commands;
   cl_program program;
   cl_kernel kernel;
   cl_uint nd;
   cl_mem a_in, b_in, c_out;
   Ndim = ORDER;
   Pdim = ORDER;
   Mdim = ORDER;
   szA = Ndim*Pdim;
   szB = Pdim*Mdim;
   szC = Ndim*Mdim;
   A = (float *)malloc(szA*sizeof(float));
   B = (float *)malloc(szB*sizeof(float));
   C = (float *)malloc(szC*sizeof(float));
   initmat(Mdim, Ndim, Pdim, A, B, C);
```

```
   err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
   context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
   commands = clCreateCommandQueue(context, device_id, 0, &err);

   a_in  = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * szA, NULL, NULL);
   b_in  = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * szB, NULL, NULL);
   c_out = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * szC, NULL, NULL);

   err = clEnqueueWriteBuffer(commands, a_in, CL_TRUE, 0, sizeof(float) * szA, A, 0, NULL, NULL);
   err = clEnqueueWriteBuffer(commands, b_in, CL_TRUE, 0, sizeof(float) * szB, B, 0, NULL, NULL);

   *program = clCreateProgramWithSource(context, 1, (const char **) & C_elem_KernelSource,  NULL, &err);
   err = clBuildProgram(*program, 0, NULL, NULL, NULL, NULL);

   *kernel = clCreateKernel(*program, "mmul", &err);
   err = clSetKernelArg(*kernel, 0, sizeof(int), &Mdim);
   err |= clSetKernelArg(*kernel, 1, sizeof(int), &Ndim);
   err |= clSetKernelArg(*kernel, 2, sizeof(int), &Pdim);
   err != clSetKernelArg(*kernel, 3, sizeof(cl_mem), &a_in);
   err |= clSetKernelArg(*kernel, 4, sizeof(cl_mem), &b_in);
   err |= clSetKernelArg(*kernel, 5, sizeof(cl_mem), &c_out);

   global[0] = (size_t) Ndim; global[1] = (size_t) Mdim; *nd = 2;
   err = clEnqueueNDRangeKernel(commands, kernel, nd, NULL, global, NULL, 0, NULL, NULL);
   clFinish(commands);
   err = clEnqueueReadBuffer( commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL );
   test_results(A, B, c_out);
}
```

Matrix Multiplication host program

```
#include "mult.h"
int main(int argc, char **argv)
{
  float *A, *B, *C;
  int Mdim, Ndim, Pdim;
  int err, szA, szB, szC;
  size_t global[DIM];
  size_t lo
  cl_devic
  cl_conte
  cl_comm
  cl_progr
  cl_kerne
  cl_uint nd;
  cl_mem a_in, b_in, c_out;
  Ndim = ORDER;
  Pdim = ORDER;
  Mdim = ORDER;
  szA = Ndim*Pdim;
  szB = Pdim*Mdim;
  szC = Ndim*Mdim;
  A = (float *)malloc(szA*sizeof(float));
  B = (float *)malloc(szB*sizeof(float));
  C = (float *)malloc(szC*sizeof(float));
  Initmat(Mdim, Ndim, Pdim, A, B, C);
```

```
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
commands = clCreateCommandQueue(context, device_id, 0, &err);

a_in  = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * szA, NULL, NULL);
b_in  = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * szB, NULL, NULL);
c_out = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * szC, NULL, NULL);

err = clEnqueueWriteBuffer(commands, a_in, CL_TRUE, 0, sizeof(float) * szA, A, 0, NULL, NULL);

&err);

err = clSetKernelArg(*kernel, 0, sizeof(int), &Mdim);
err |= clSetKernelArg(*kernel, 1, sizeof(int), &Ndim);
err |= clSetKernelArg(*kernel, 2, sizeof(int), &Pdim);
err != clSetKernelArg(*kernel, 3, sizeof(cl_mem), &a_in);
err |= clSetKernelArg(*kernel, 4, sizeof(cl_mem), &b_in);
err |= clSetKernelArg(*kernel, 5, sizeof(cl_mem), &c_out);

global[0] = (size_t) Ndim; global[1] = (size_t) Mdim; *nd = 2;
err = clEnqueueNDRangeKernel(commands, kernel, nd, NULL, global, NULL, 0, NULL, NULL);
clFinish(commands);
err = clEnqueueReadBuffer( commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL );
test_results(A, B, c_out);
}
```

Note: This isn't as bad as you might think.
This is almost the same as the host code we wrote for vector add.
It's "boilerplate" … you get it right once and just re-use it.

Matrix Multiplication host program

```
#include "mult.h"
int main(int argc, char **argv)
{
   float *A, *B, *C;
   int Mdim, Ndim, Pdim;
   int err, szA, szB, szC;
   size_t global[DIM];
   size_t local[DIM];
   cl_device_id device_id;
   cl_context context;
   cl_command_queue commands;
   cl_pro
   cl_ker
   cl_uin
   cl_me
   Ndim = ORDER;
   Pdim = ORDER;
   Mdim = ORDER;
   szA = Ndim*Pdim;
   szB = Pdim*Mdim;
   szC = Ndim*Mdim;
   A = (float *)malloc(szA*sizeof(float));
   B = (float *)malloc(szB*sizeof(float));
   C = (float *)malloc(szC*sizeof(float));
   initmat(Mdim, Ndim, Pdim, A, B, C);
```

Declare and initialize data

```
   err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
   context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
   commands = clCreateCommandQueue(context, device_id, 0, &err);
```

Setup the platform

```
   a_in  = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * szA, NULL, NULL);
   b_in  = clCreateBuffer(context, CL_MEM_R
   c_out = clCreateBuffer(context, CL_MEM_W

   err = clEnqueueWriteBuffer(commands, a_
   err = clEnqueueWriteBuffer(commands, b_in, CL_TRUE, 0, sizeof(float) * szB, B, 0, NULL, NULL);
```

Setup buffers and write A and B matrices to the device memory

```
   *program = clCreateProgramWithSource(context, 1, (const char **) & C_elem_KernelSource,  NULL, &err);
   err = clBuildProgram(*program, 0, NULL, NULL, NULL, NULL);

   *kernel = clCreateKernel(*program, "mmul", &err);
   err = clSetKernelArg(*kernel, 0, sizeof(int), &Mdim);
   err |= clSetKernelArg(*kernel, 1, sizeof(int), &Ndim);
   err |= clSetKernelArg(*kernel, 2, sizeof(int), &Pdim);
   err != clSetKernelArg(*kernel, 3, sizeof(cl_mem), &a_in);
   err |= clSetKernelArg(*kernel, 4, sizeof(cl_mem), &b_in);
   err |= clSetKernelArg(*kernel, 5, sizeof(cl_mem), &c_out);
```

Build the program, define the Kernel,and setup arguments

```
   global[0] = (size_t) Ndim; global[1] = (size_t) Mdim; *nd = 2;
   err = clEnqueueNDRangeKernel(commands, ker
   clFinish(commands);
   err = clEnqueueReadBuffer( commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL );
   test_results(A, B, c_out);
}
```

Run the kernel and collect results

Matrix Multiplication host program

```
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

```
#include
int main(
{
   float *A
   int Mdi
   int err,
   size_t g
   size_t local[DIM];
   cl_device_id device_id;
   cl_context context;
```

The only parts new to us …

1. 2D ND Range set to dimensions of C matrix
2. Local sizes set to NULL in clEnqueueNDRangeKernel() to tell system to pick local dimensions for us.

```
                                                                              );
err = clEnqueueWriteBuffer(commands, b_in, CL_TRUE, 0, sizeof(float) * szB, B, 0, NULL, NULL);
```

```
*program = clCreateProgramWithSource(context, 1, (const char **) & C_elem_KernelSource, NULL, &err);
```

```
global[0] = (size_t) Ndim;       global[1] = (size_t) Mdim;       *nd  = 2;
err = clEnqueueNDRangeKernel(commands, kernel, ndim, NULL, global, NULL, 0, NULL, NULL);
clFinish(commands);
err = clEnqueueReadBuffer( commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL );
test_results(A, B, c_out);
```

```
   szA = Ndim*Pdim;
   szB = Pdim*Mdim;
   szC = Ndim*Mdim;
   A = (float *)malloc(szA*sizeof(float));
   B = (float *)malloc(szB*sizeof(float));
   C = (float *)malloc(szC*sizeof(float));
   initmat(Mdim, Ndim, Pdim, A, B, C);
```

```
err |= clSetKernelArg(*kernel, 5, sizeof(cl_mem), &c_out);
```

```
global[0] = (size_t) Ndim; global[1] = (size_t) Mdim; *nd = 2;
err = clEnqueueNDRangeKernel(commands, kernel, nd, NULL, global, NULL, 0, NULL, NULL);
clFinish(commands);
err = clEnqueueReadBuffer( commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL );
test_results(A, B, c_out);
}
```

Matrix Multiplications Performance

- **Results on an Apple laptop with an NVIDIA GPU and an Intel CPU. Matrices are stored in global memory.**

Case	MFLOPS
CPU: Sequential C (not OpenCL)	167
GPU: C(i,j) per work item, all global	511
CPU: C(i,j) per work item, all global	744

Note: The goal of this study is to explore features of the OpenCL API. If we want high performance matrix multiplication we'd use the OpenCL native function interface to call MKL.

Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units
Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

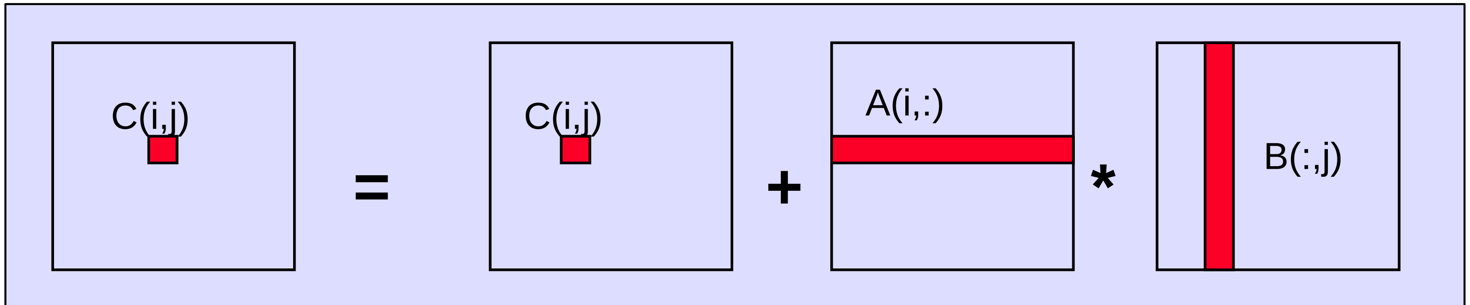
3rd party names are the property of their owners.

Agenda

- **Heterogeneous computing and the origins of OpenCL**
- **OpenCL overview**
- **Mapping OpenCL onto CPUs**
- **Exploring the spec with code: embarrassingly parallel**
  - Vector addition: the basic platform layer
  - Matrix multiplication: kernel programming language
- **Exploring the spec with code: dealing with dependencies**
  - ➡ Optimizing matrix mul.: work groups and the memory model
  - Reduction: synchronization
- **A survey of the rest of OpenCL**

Optimizing Matrix Multiplication

- Cost determined by flops and memory movement:
  - $2*n^3 = O(n^3)$ Flops
  - operates on $3*n^2$ numbers
- To optimize matrix multiplication, we must assure that for every memory movement, we execute as many flops as possible.
- Outer product algorithms are faster, but for pedagogical reasons, lets stick to the simple dot-product algorithm.

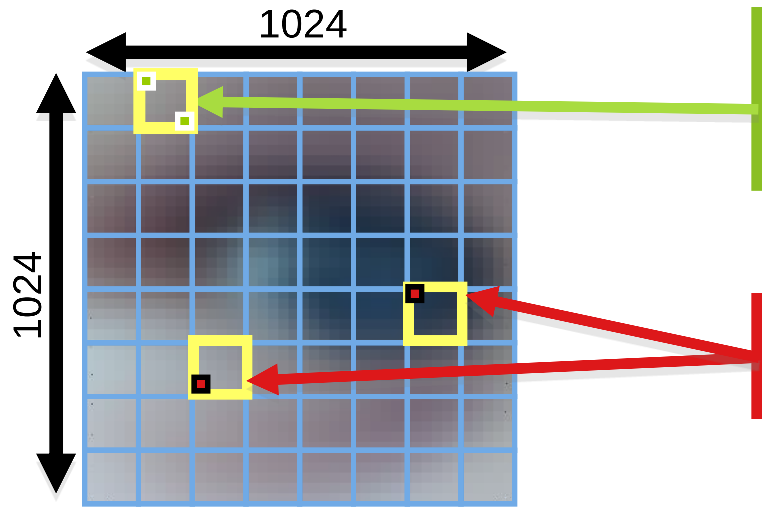


Dot product of a row of A and a column of B for each element of C

- We will work with work-item/work-group sizes and the memory model to optimize matrix multiplication

An N-dimension domain of work-items

- Global Dimensions: 1024 x 1024 (whole problem space)
- Local Dimensions: 128 x 128 (work group … executes together)



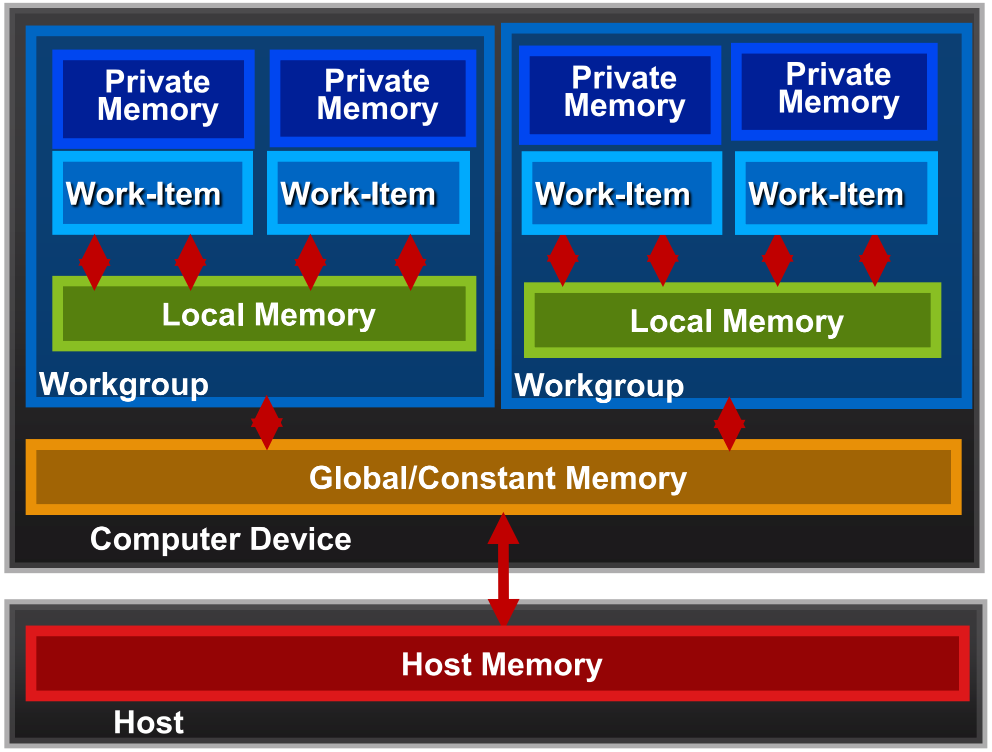
Synchronization between work-items possible only within workgroups: barriers and memory fences

Cannot synchronize outside of a workgroup

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

OpenCL Memory Model

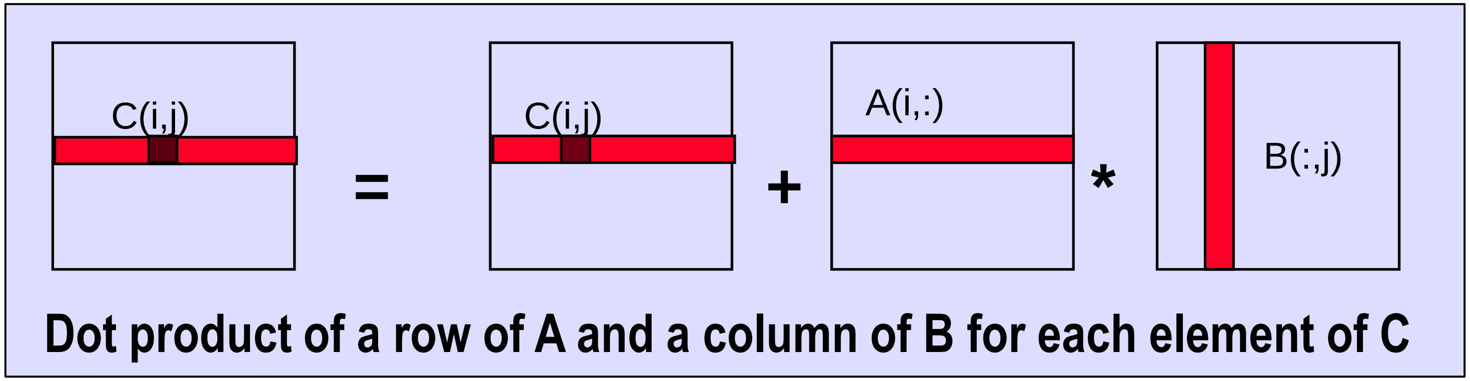
- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a workgroup
- **Global/Constant Memory**
  - Visible to all workgroups
- **Host Memory**
  - On the CPU



- **Memory management is explicit**
  You must move data from host -> global -> local *and* back

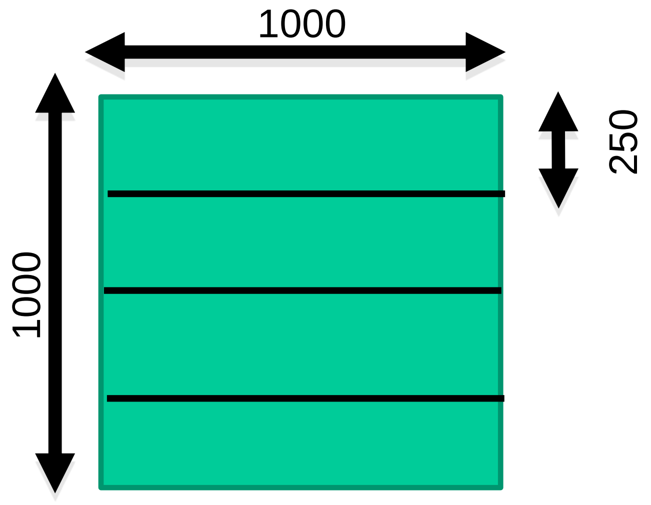
Optimizing Matrix Multiplication

- There may be significant overhead to manage work items and work groups.
- Let's have each work item compute a full row of C


C(i,j)
=
C(i,j)
+
A(i,:)
*
B(:,j)
Dot product of a row of A and a column of B for each element of C

An N-dimension domain of work-items

- Whole problem space: 1000 x 1000
- Global Dimensions: 1000 (A 1D NDRange, one row per work-item)
- Local Dimensions: 250 (One work group for each compute unit*)



*Remember … the device we are optimizing for is a GeForce® 8600M GT GPU from NVIDIA with 4 compute units

Reduce work-item overhead …

do one row of C per work-item

```
__kernel mmul(
     const int Mdim,
     const int Ndim,
     const int Pdim,
     __global float* A,
     __global float* B,
     __global float* C)
{
    int k,j;
    int i = get_global_id(0);
    float tmp;
   for(j=0;j<Mdim;j++){
         tmp = 0.0;
         for(k=0;k<Pdim;k++)
 tmp += A[i*Ndim+k] * B[k*Pdim+j];
         C[i*Ndim+j] = tmp;
    }
}
```

MatMult host program: one row of C per work-item

```
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

Changes to host program:

1. 1 D ND Range set to number of rows in the C matrix
2. Local Dim set to 250 so number of work-groups match number of compute units (4 in this case) for our order 1000 matrices

```
#include
int main(
{
   float *A
   int Mdi
   int err,
   size_t g
   size_t local[DIM];
   cl_device_id device_id;
   cl_context context;
   cl_command_queue commands;
   cl_program program;
   cl_kernel kernel;
   cl_uint nd;
   szA = Ndim*Pdim;
   szB = Pdim*Mdim;
   szC = Ndim*Mdim;
   A = (float *)malloc(szA*sizeof(float));
   B = (float *)malloc(szB*sizeof(float));
   C = (float *)malloc(szC*sizeof(float));
   initmat(Mdim, Ndim, Pdim, A, B, C);
```

```
);
err = clEnqueueWriteBuffer(commands, b_in, CL_TRUE, 0, sizeof(float) * szB, B, 0, NULL, NULL);
```

```
*program = clCreateProgramWithSource(context, 1, (const char **) & C_elem_KernelSource,  NULL, &err);
err = clBuildProgram(*program, 0, NULL, NULL, NULL, NULL);

*kernel = clCreateKernel(*program, "mmul", &err);
err = clSetKernelArg(*kernel, 0, sizeof(int), &Mdim);
err |= clSetKernelArg(*kernel, 5, sizeof(cl_mem), &c_out);
```

```
global[0] = (size_t) Ndim; local[0] = (size_t) 250;   *nd  = 1;
err = clEnqueueNDRangeKernel(commands, kernel, nd, NULL, global, local, 0, NULL, NULL);
```

```
global[0] = (size_t) Ndim; local[0] = (size_t) 250; *nd = 1;
err = clEnqueueNDRangeKernel(commands, kernel, nd, NULL, global, local, 0, NULL, NULL);
clFinish(commands);
err = clEnqueueReadBuffer( commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL );
test_results(A, B, c_out);
}
```

Results: MFLOPS

- Results on an Apple laptop with an NVIDIA GPU and an Intel CPU.

Case	MFLOPS
CPU: Sequential C (not OpenCL)	167
GPU: C(i,j) per work item, all global	511
GPU: C row per work item, all global	258
CPU: C(i,j) per work item	744

This on its own didn't help. Note, we made no attempt to take NVIDIA's 32-warp size into account

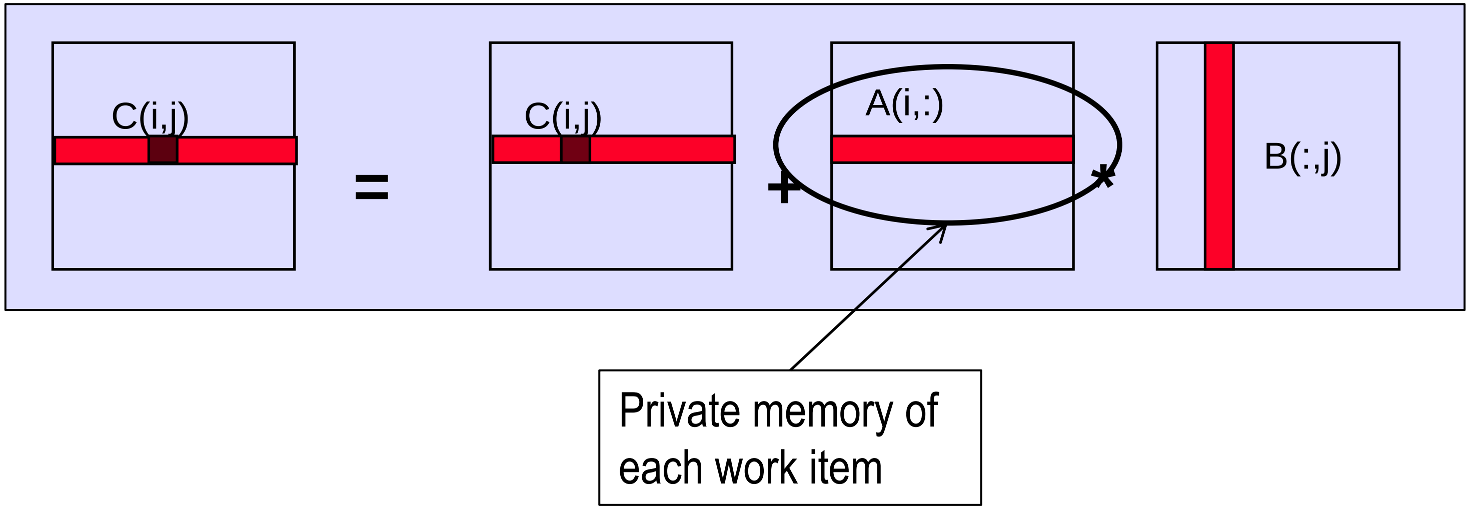
Note: The goal of this study is to explore features of the OpenCL API. If we want high performance matrix multiplication we'd use the OpenCL native function interface to call MKL.

Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units
Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

3rd party names are the property of their owners.

Optimizing Matrix Multiplication

- Notice that each element of C in a row uses the same row of A.
- Let’s copy A into private memory so we don’t incur the overhead of pulling it from global memory for each C(i,j) computation.



Row of C per work item, A row private

```
__kernel mmul(
     const int Mdim,
     const int Ndim,
     const int Pdim,
     __global float* A,
     __global float* B,
     __global float* C)
{
    int k,j;
    int i = get_global_id(0);
    float Awrk[1000];
    float tmp;
    for(k=0;k<Pdim;k++)
          Awrk[k] = A[i*Ndim+k];
    for(j=0;j<Mdim;j++){
         tmp = 0.0;
         for(k=0;k<Pdim;k++)
             tmp += Awrk[k] * B[k*Pdim+j];
         C[i*Ndim+j] = tmp;
     }
}
```

Setup a work array for A in private memory and copy into from global memory before we start with the matrix multiplications.

MatMult host program: one row of C per work-item

```
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

Changes to host program:

1. 1 D ND Range set to number of rows in the C matrix
2. Local Dim set to 250 so number of work-groups match number of compute units (4 in this case) for our order 1000 matrices

```
#include
int main(
{
   float *A
   int Mdi
   int err,
   size_t g
   size_t local[DIM];
   cl_device_id device_id;
   cl_context context;
   cl_command_queue commands;
   cl_program program;
   cl_kernel kernel;
   cl_uint nd;
```

```
);
err = clEnqueueWriteBuffer(commands, b_in, CL_TRUE, 0, sizeof(float) * szB, B, 0, NULL, NULL);
```

```
*program = clCreateProgramWithSource(context, 1, (const char **) & C_elem_KernelSource,  NULL, &err);
err = clBuildProgram(*program, 0, NULL, NULL, NULL, NULL);

*kernel = clCreateKernel(*program, "mmul", &err);
err = clSetKernelArg(*kernel, 0, sizeof(int), &Mdim);
```

```
global[0] = (size_t) Ndim; local[0] = (size_t) 250;   *nd = 1;
err = clEnqueueNDRangeKernel(commands, kernel, nd, NULL, global, local, 0, NULL, NULL);
```

```
   szA = Ndim*Pdim;
   szB = Pdim*Mdim;
   szC = Ndim*Mdim;
   A = (float *)malloc(szA*sizeof(float));
   B = (float *)malloc(szB*sizeof(float));
   C = (float *)malloc(szC*sizeof(float));
   initmat(Mdim, Ndim, Pdim, A, B, C);
```

```
err |= clSetKernelArg(*kernel, 5, sizeof(cl_mem), &c_out);
```

```
global[0] = (size_t) Ndim; local[0] = (size_t) 250; *nd = 1;
err = clEnqueueNDRangeKernel(commands, kernel, nd, NULL, global, local, 0, NULL, NULL);
clFinish(commands);
err = clEnqueueReadBuffer( commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL );
test_results(A, B, c_out);
}
```

Matrix Multiplications Performance

- **Results on an Apple laptop with an NVIDIA GPU and an Intel CPU.**

Case	MFLOPS
CPU: Sequential C (not OpenCL)	167
GPU: C(i,j) per work item, all global	511
GPU: C row per work item, all global	258
GPU: C row per work item, A row private	873
CPU: C(i,j) per work item	744

Big impact

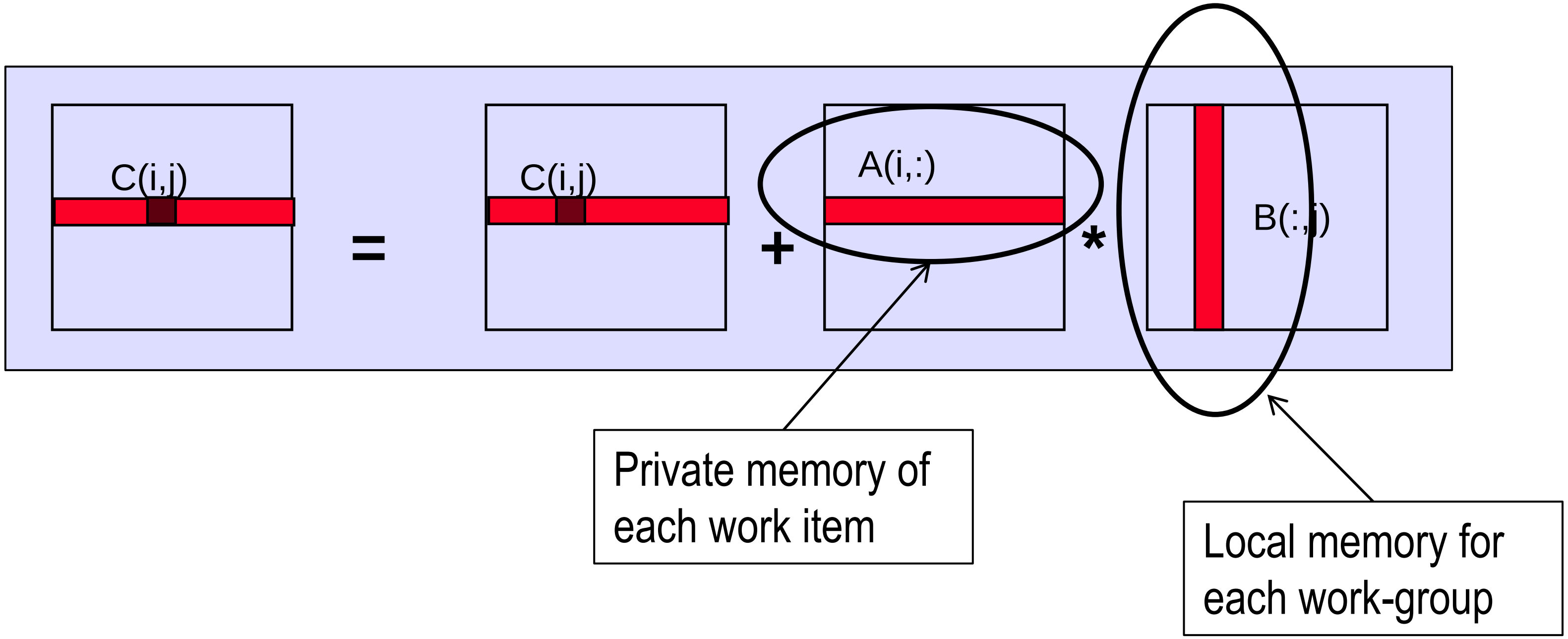
Note: The goal of this study is to explore features of the OpenCL API. If we want high performance matrix multiplication we'd use the OpenCL native function interface to call MKL.

Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units
Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

3rd party names are the property of their owners.

Optimizing Matrix Multiplication

- Notice that each element of C uses the same row of A.
- Each work-item in a work-group uses the same columns of B
- Let’s store the B columns in local memory


C(i,j)
=
C(i,j)
+
A(i,:)
*
B(:,j)
Private memory of each work item
Local memory for each work-group

Row of C per work item, A row private, B columns local

```
__kernel mmul(
     const int Mdim,
     const int Ndim,
     const int Pdim,
     __global float* A,
     __global float* B,
     __global float* C,
     __local float* Bwrk)
{
    int k,j;
    int i = get_global_id(0);
    int iloc = get_local_id(0);
    int nloc = get_local_size(0);
    float Awrk[1000];
    float tmp;
    for(k=0;k<Pdim;k++)
              Awrk[k] = A[i*Ndim+k];
       for(j=0;j<Mdim;j++){
       for(k=iloc;k<Pdim;k=k+nloc)
              Bwrk[k] = B[k*Pdim+j];
    barrier(CLK_LOCAL_MEM_FENCE);
              tmp = 0.0;
              for(k=0;k<Pdim;k++)
    tmp += Awrk[k] * Bwrk[k];
              C[i*Ndim+j] = tmp;
       }
}
```

Pass in a pointer to local memory. Work-items in a group start by copying the columns of B they need into the local memory.

MatMult host program: No change from before

```
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

Changes to host program:

1. Modify so we pass local memory to kernels. This requires a change to the kernel argument list … a new call to clSetKernelArg is needed … it sets aside memory (sizeof(float)*Pdim) but doesn't assign it a value

```
#include
int main(
{
   float *A
   int Mdi
   int err,
   size_t g
   size_t local[DIM];
   cl_device_id device_id;
   cl_context context;
   cl_command_queue commands;
   cl_program program;
   cl_kernel kernel;
   cl_uint nd;
   cl_mem a_in, b_in, c_out
   Ndim = ORDER;
   Pdim = ORDER;
   Mdim = ORDER;
   szA = Ndim*Pdim;
   szB = Pdim*Mdim;
   szC = Ndim*Mdim;
   A = (float *)malloc(szA*sizeof(float));
   B = (float *)malloc(szB*sizeof(float));
   C = (float *)malloc(szC*sizeof(float));
   initmat(Mdim, Ndim, Pdim, A, B, C);
```

```
err = clEnqueueWriteBuffer(commands, b_in, CL_TRUE, 0, sizeof(float) * szB, B, 0, NULL, NULL);
```

```
*program = clCreateProgramWithSource(context, 1, (const char **) & C_elem_KernelSource,  NULL, &err);
err = clBuildProgram(*program, 0, NULL, NULL, NULL, NULL);

*kernel = clCreateKernel(*program, "mmul", &err);
```

```
err |= clSetKernelArg(*kernel, 6, sizeof(float)*Pdim, NULL);
```

```
err |= clSetKernelArg(*kernel, 2, sizeof(int), &Pdim);
err != clSetKernelArg(*kernel, 3, sizeof(cl_mem), &a_in);
err |= clSetKernelArg(*kernel, 4, sizeof(cl_mem), &b_in);
err |= clSetKernelArg(*kernel, 5, sizeof(cl_mem), &c_out);
err |= clSetKernelArg(*kernel, 6, sizeof(float)*Pdim, NULL);
```

```
global[0] = (size_t) Ndim; global[1] = (size_t) Mdim; *nd = 2;
err = clEnqueueNDRangeKernel(commands, kernel, nd, NULL, global, local, 0, NULL, NULL);
clFinish(commands);
err = clEnqueueReadBuffer( commands, c_out, CL_TRUE, 0, sizeof(float) * szC, C, 0, NULL, NULL );
test_results(A, B, c_out);
}
```

Matrix Multiplications Performance

- **Results on an Apple laptop with an NVIDIA GPU and an Intel CPU.**

Case	MFLOPS
CPU: Sequential C (not OpenCL)	167
GPU: C(i,j) per work item, all global	511
GPU: C row per work item, all global	258
GPU: C row per work item, A row private	873
GPU: C row per work item, A private, B local	2472
CPU: C(i,j) per work item	744

Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units
Device is Intel® Core™2 Duo CPU   T8300 @ 2.40GHz

3rd party names are the property of their owners.

Matrix Multiplications Performance

- **Results on an Apple laptop with an NVIDIA GPU and an Intel CPU.**

Case	Speedup
CPU: Sequential C (not OpenCL)	1
GPU: C(i,j) per work item, all global	3
GPU: C row per work item, all global	1.5
GPU: C row per work item, A row private	5.2
GPU: C row per work item, A private, B local	15
CPU: C(i,j) per work item	4.5

Wow!!! OpenCL on a GPU is radically faster that C on a CPU, right?

Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units
Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

3rd party names are the property of their owners.

CPU vs GPU: Let’s be fair

- **We made no attempt to optimize the CPU/C code but we worked hard to optimize OpenCL/GPU code.**
- **Lets optimize the CPU code**
  - Use compiler optimization (level O3).
  - Replace float with double (CPU ALU’s like double)
  - Reorder loops:

- Float, no opt 167 mflops
- Double, O3 272 mflops

```
void mat_mul_ijk(int Mdim, int Ndim, int Pdim,
                    double *A, double *B, double *C)
{
   int i, j, k;
   for (i=0; i<Ndim; i++)
      for (j=0; j<Mdim; j++)
         for(k=0;k<Pdim;k++)
C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
}
```

```
void mat_mul_ikj(int Mdim, int Ndim, int Pdim,
                    double *A, double *B, double *C)
{
   int i, j, k;
   for (i=0; i<Ndim; i++)
      for(k=0;k<Pdim;k++)
         for (j=0; j<Mdim; j++)
C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
}
```

- ijk: 272 mflops
- ikj: 1130 mflops
- kij: 481 mflops

Matrix Multiplications Performance

- **Results on an Apple laptop with an NVIDIA GPU and an Intel CPU.**

Case	Speedup
CPU: Sequential C (not OpenCL)	1
GPU: C(i,j) per work item, all global	0.45
GPU: C row per work item, all global	0.23
GPU: C row per work item, A row private	0.77
GPU: C row per work item, A private, B local	2.2
CPU: C(i,j) per work item	0.66

And we still are only using one core … and we are not using SSE so there is lots of room to further optimize the CPU code.

Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units
Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

3rd party names are the property of their owners.

Agenda

- **Heterogeneous computing and the origins of OpenCL**
- **OpenCL overview**
- **Mapping OpenCL onto CPUs**
- **Exploring the spec with code: embarrassingly parallel**
  - Vector addition: the basic platform layer
  - Matrix multiplication: kernel programming language
- **Exploring the spec with code: dealing with dependencies**
  - Optimizing matrix mul.: work groups and the memory model
  - ➡ Reduction: synchronization
- **A survey of the rest of OpenCL**

Work-Item Synchronization

- **Within a work-group**
  - void barrier()
  - Takes optional flags CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE
  - A work-item that encounters a barrier() will wait until ALL work-items in the work-group reach the barrier()
  - Corollary: If a barrier() is inside a branch, then the branch must be taken by either
    - ALL work-items in the work-group, OR
    - NO work-item in the work-group

- **Across work-groups**
  - No guarantees as to where and when a particular work-group will be executed relative to another work-group
  - Cannot exchange data, or have barrier-like synchronization between two different work-groups

Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC'11 OpenCL tutorial

Reduction

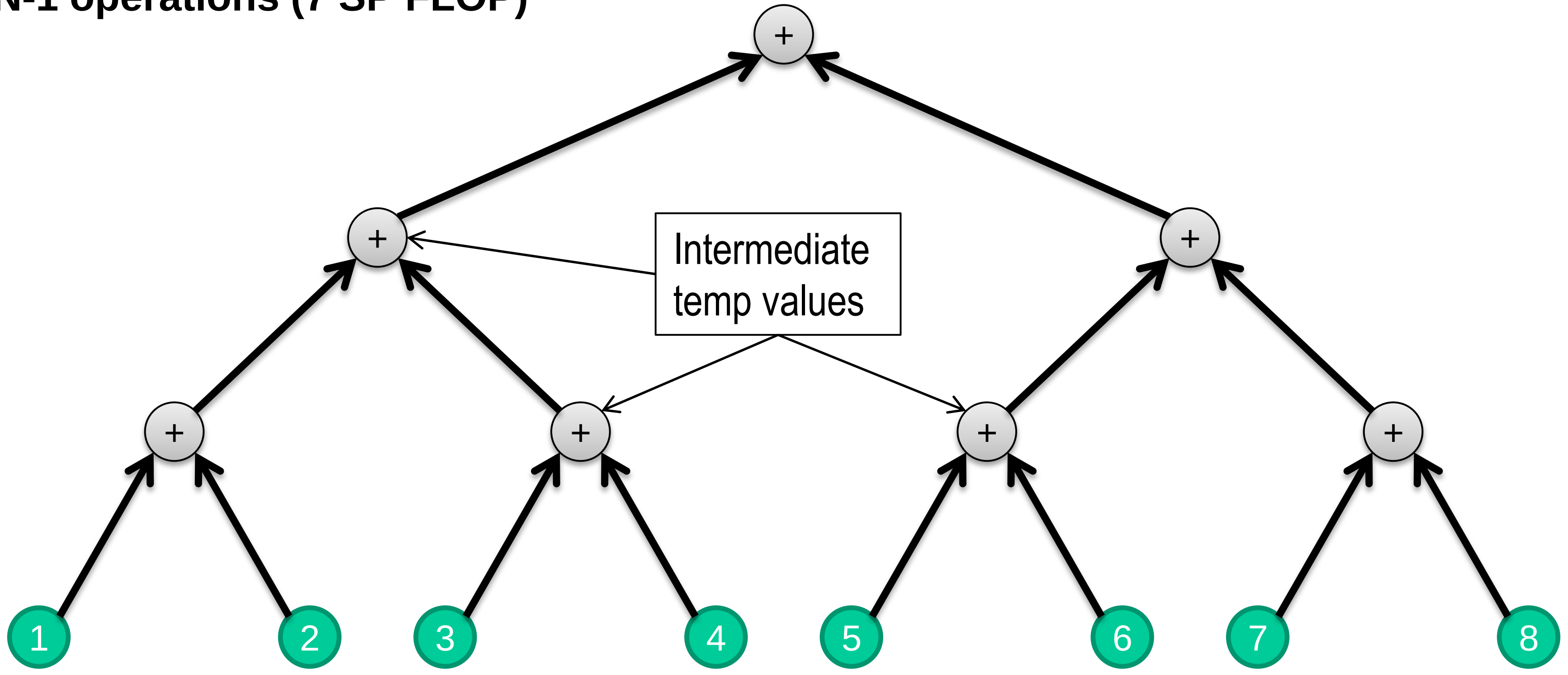
- **Goal: Reduce a set of numbers to a single value**
  - E.g., find sum of all elements in an array
- **Sequential code**

```
int reduce(int Ndim, int *A)
{
        sum = 0;
        for (int i=0; i<Ndim; i++)
        sum += A[i];
}
```

Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC'11 OpenCL tutorial

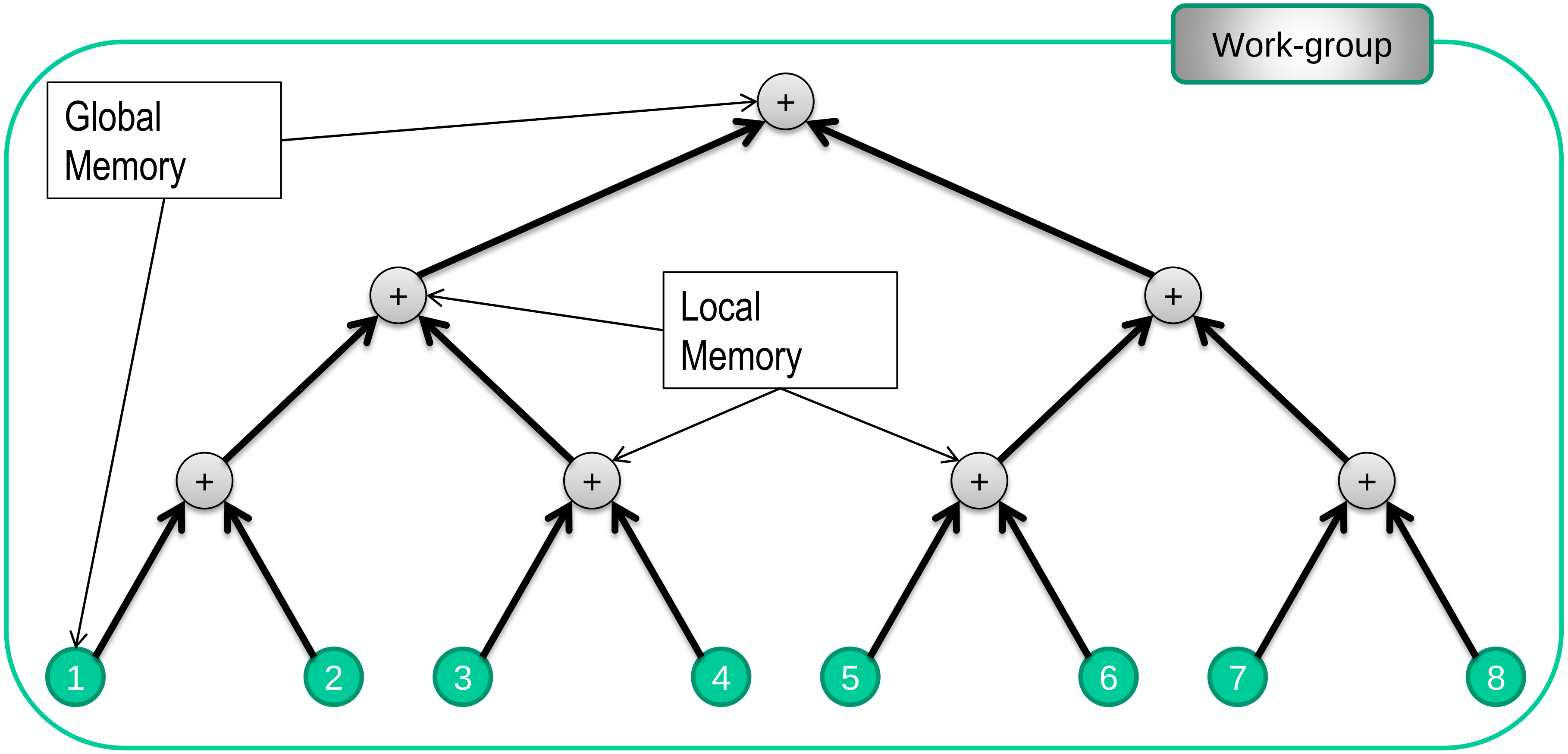
Parallel Reduction (Basic Idea)

- N values read (32 bytes)
- N-1 operations (7 SP FLOP)



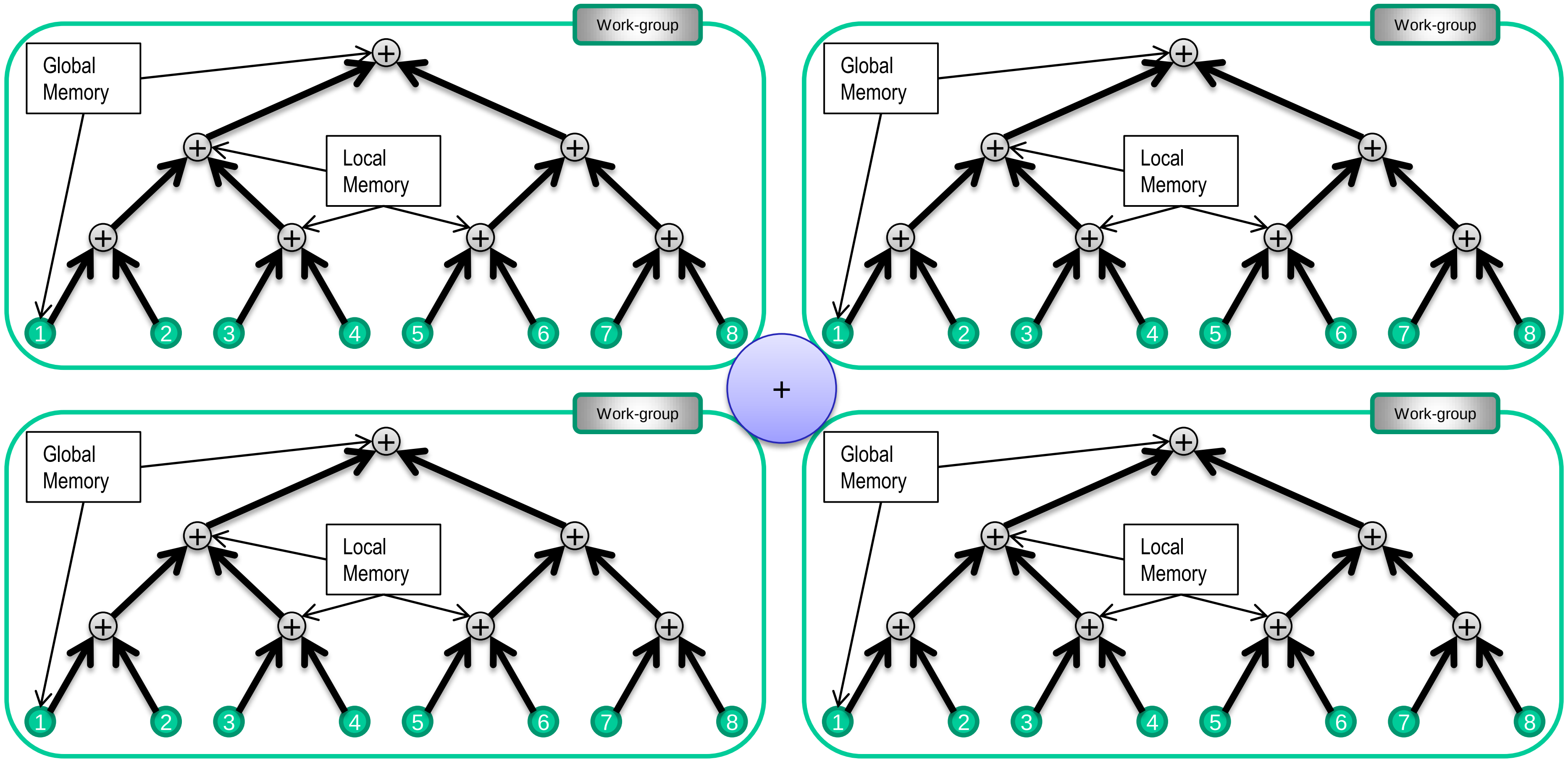
Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC'11 OpenCL tutorial

Parallel Reduction (OpenCL)



Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC'11 OpenCL tutorial

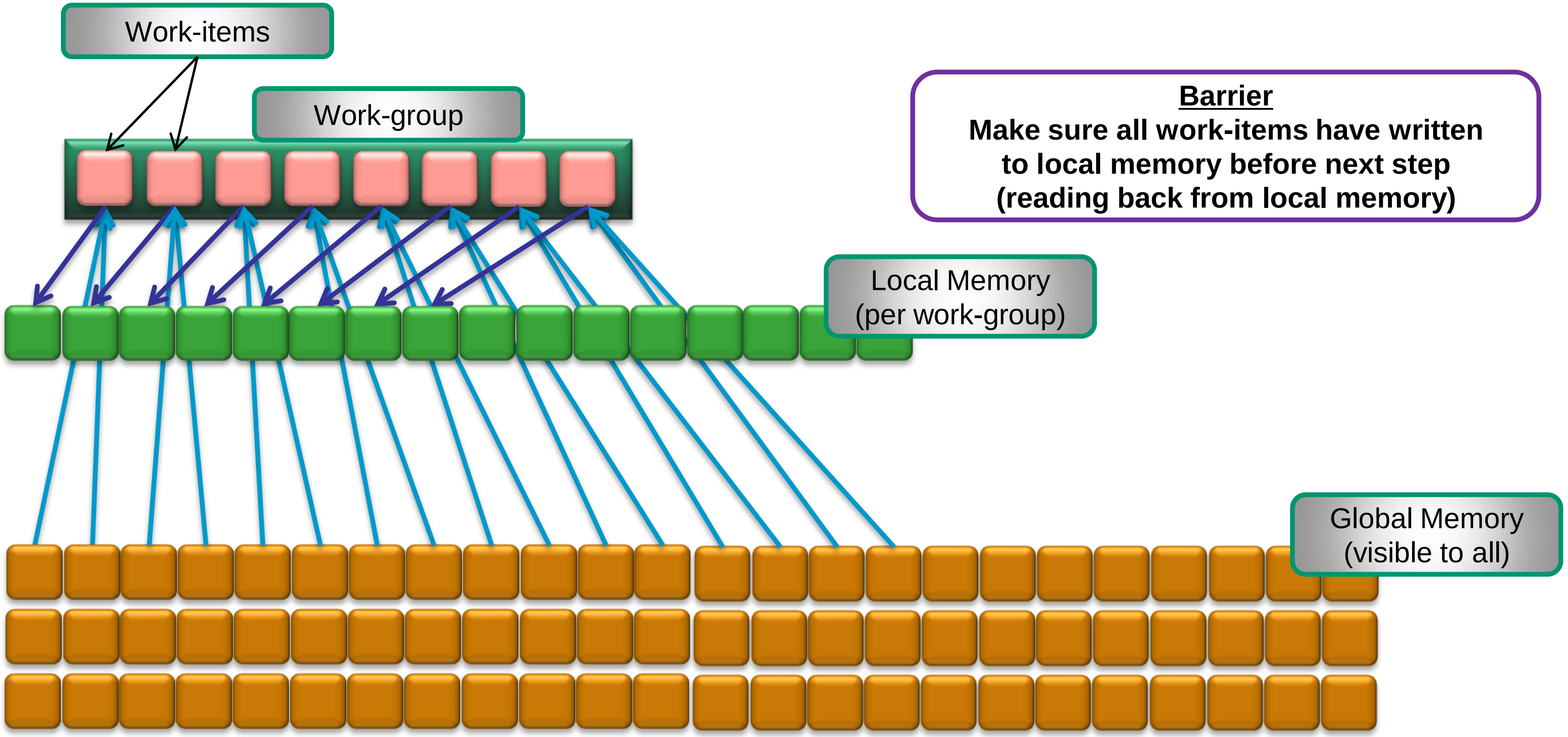
Parallel Reduction (OpenCL)



Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC’11 OpenCL tutorial

OpenCL Reduction Kernel (1/5)

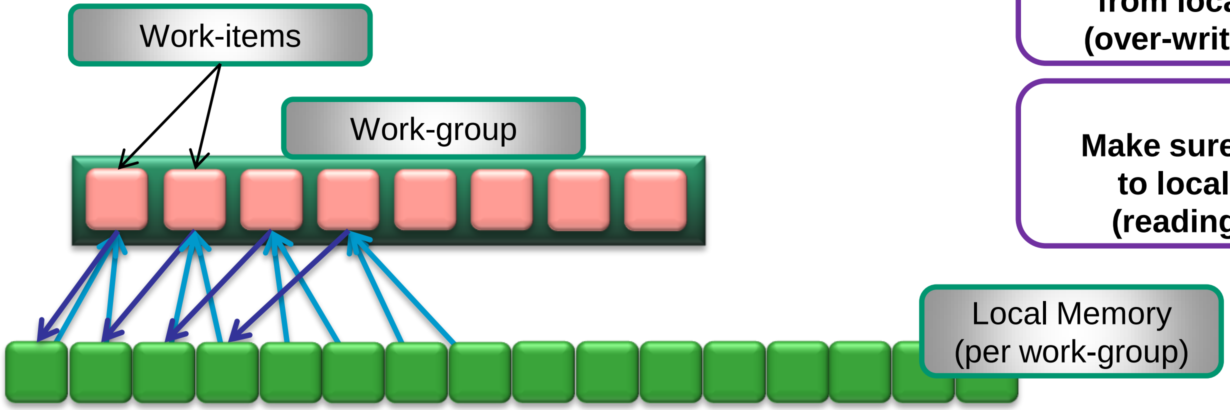
- **For each work-group:**


Work-items
Work-group
Barrier
Make sure all work-items have written to local memory before next step (reading back from local memory)
Local Memory (per work-group)
Global Memory (visible to all)

Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC'11 OpenCL tutorial

OpenCL Reduction Kernel (2/5)

- **For each work-group:**



<u>Barrier</u>
Make sure all work-items have read from local memory before next step (over-writing local memory locations)

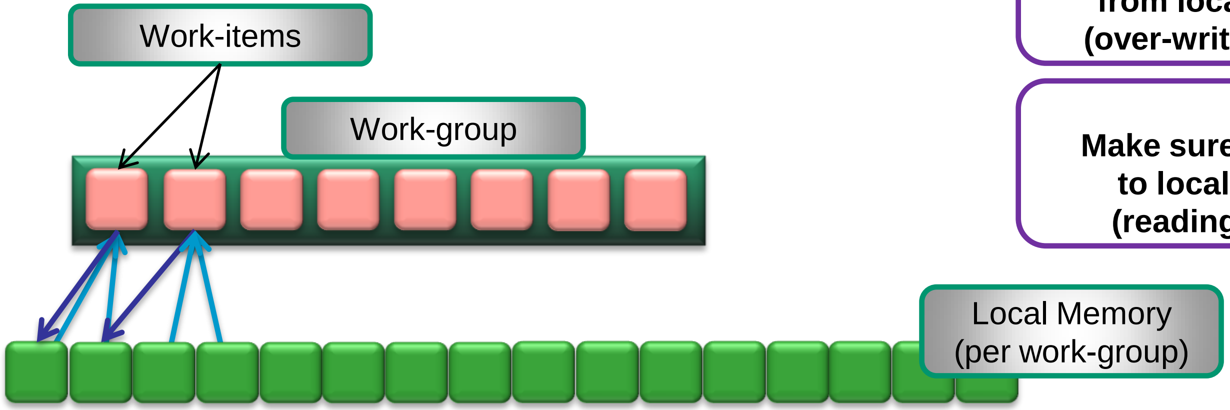
<u>Barrier</u>
Make sure all work-items have written to local memory before next step (reading back from local memory)

- **Why do we need a barrier at the end of the previous step?**
  - **To ensure that all work-items in previous step finish writing to their respective local memory locations, before a potentially different work-item reads from the same memory location.**

Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC'11 OpenCL tutorial

OpenCL Reduction Kernel (3/5)

- **For each work-group:**



<u>Barrier</u>
Make sure all work-items have read from local memory before next step (over-writing local memory locations)

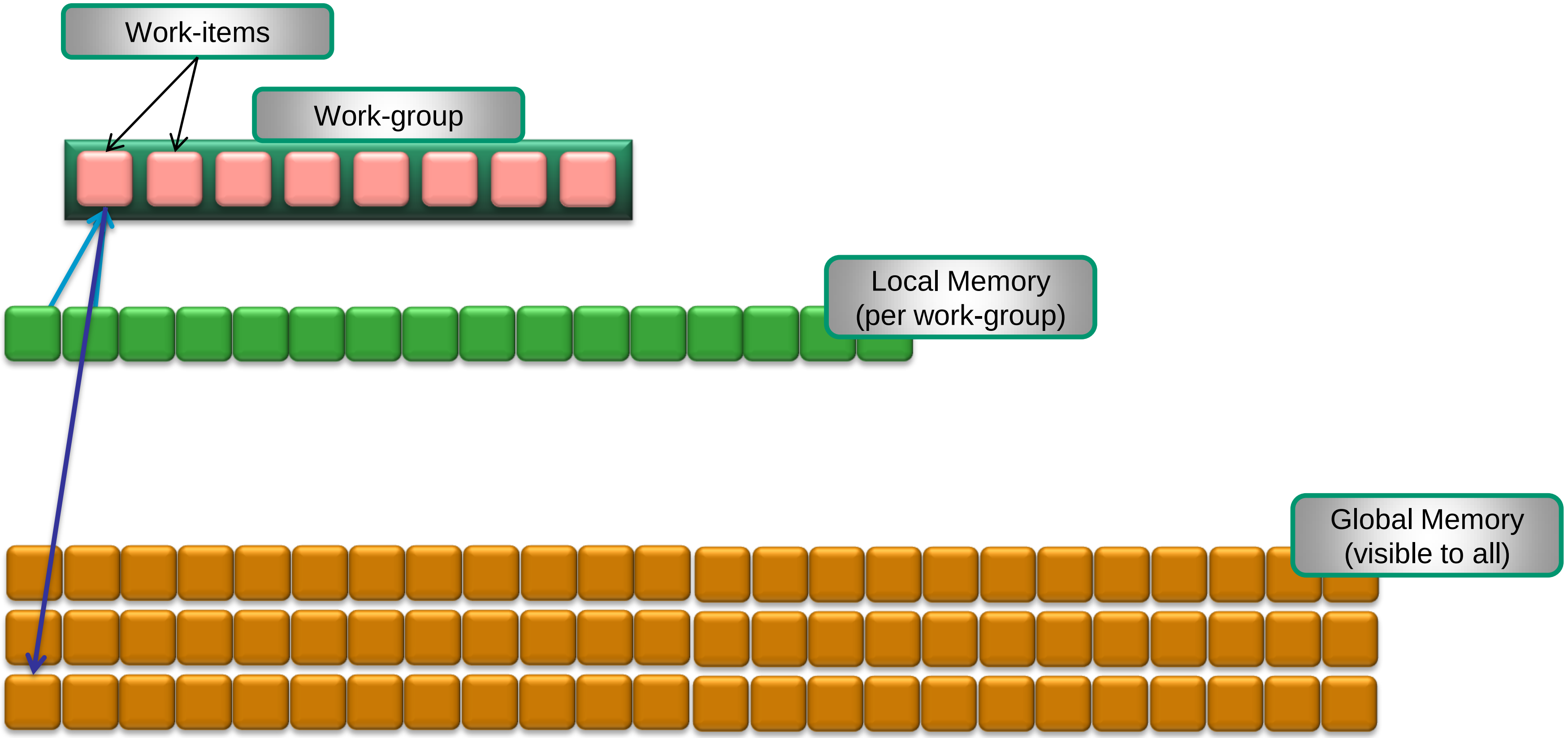
<u>Barrier</u>
Make sure all work-items have written to local memory before next step (reading back from local memory)

- **Put dummy barriers in the idle work-items**

Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC'11 OpenCL tutorial

OpenCL Reduction Kernel (4/5)

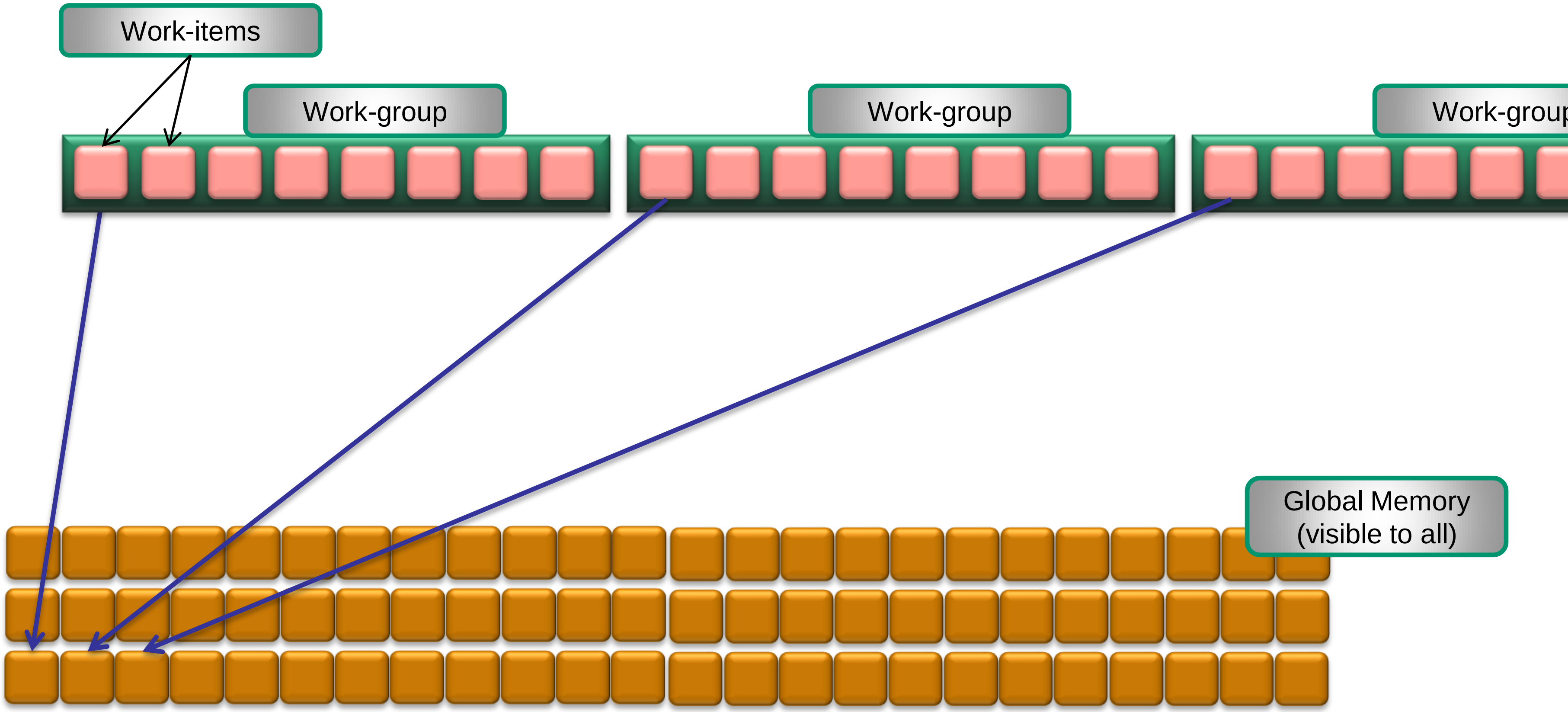
- **Each work-group writes a partial sum to global memory**



Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC'11 OpenCL tutorial

OpenCL Reduction Kernel (5/5)

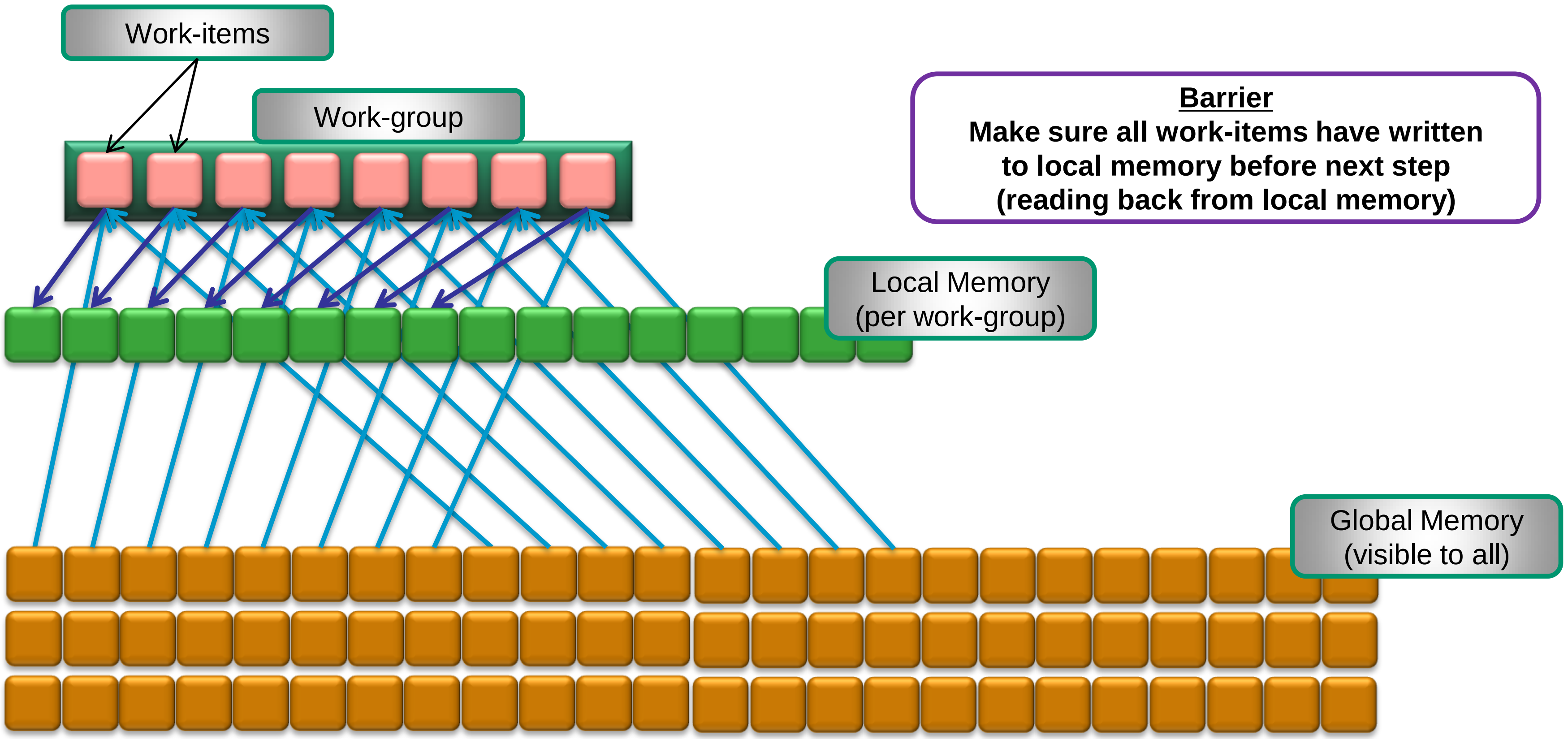
- The partial sums are then be added either on the host, or using another OpenCL reduction kernel (with a single work-group)



Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC'11 OpenCL tutorial

Alternative Reduction Kernel (1/5)

- **Better read pattern for GPU device**



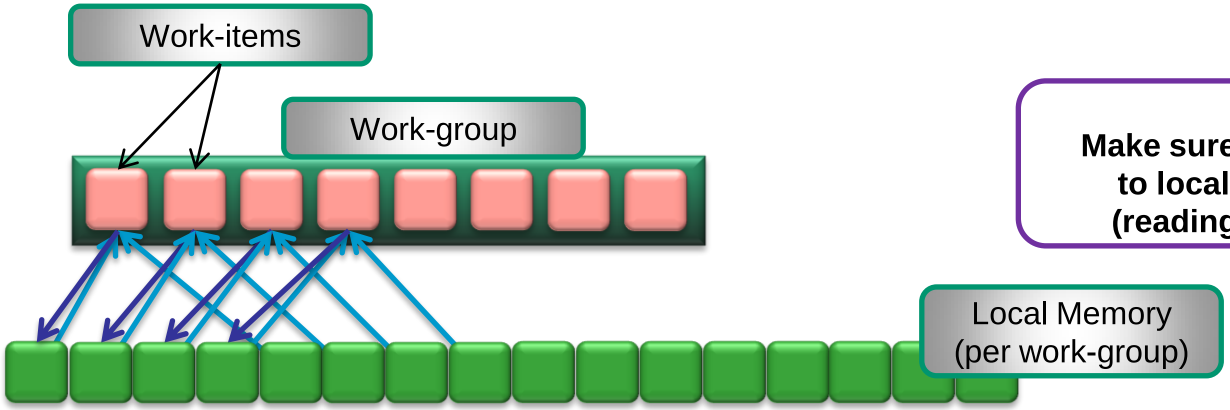
Barrier
Make sure all work-items have written to local memory before next step (reading back from local memory)

Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC'11 OpenCL tutorial

Alternative Reduction Kernel (2/5)

- Remove need for barrier before write to local memory

B



Barrier
Make sure all work-items have written to local memory before next step (reading back from local memory)

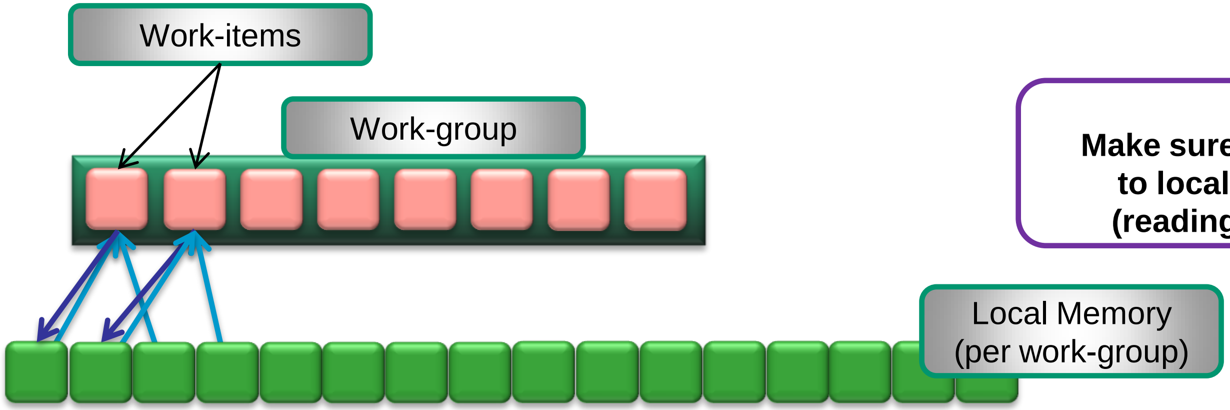
- Why do we not need a barrier before writing to local memory?
  - In this kernel, a work-item is writing to its own read location (and not to a location from where a potentially different work-item reads)
  - But we still need a barrier after writing to local memory

Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC'11 OpenCL tutorial

Alternative Reduction Kernel (3/5)

- Remove need for barrier before write to local memory

B



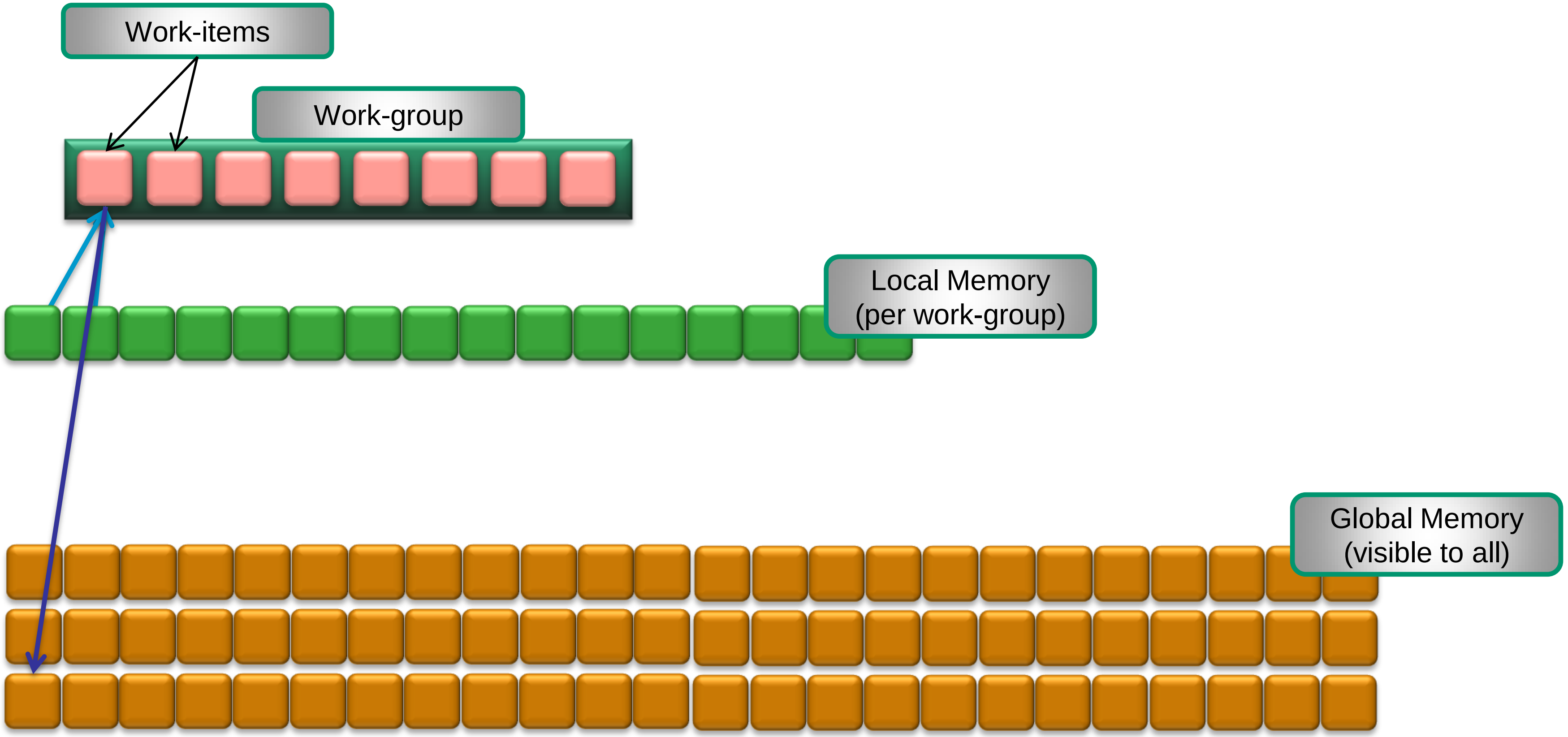
Barrier
Make sure all work-items have written to local memory before next step (reading back from local memory)

Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC'11 OpenCL tutorial

Alternative Reduction Kernel (4/5)

- Each work-group writes a partial sum to global memory

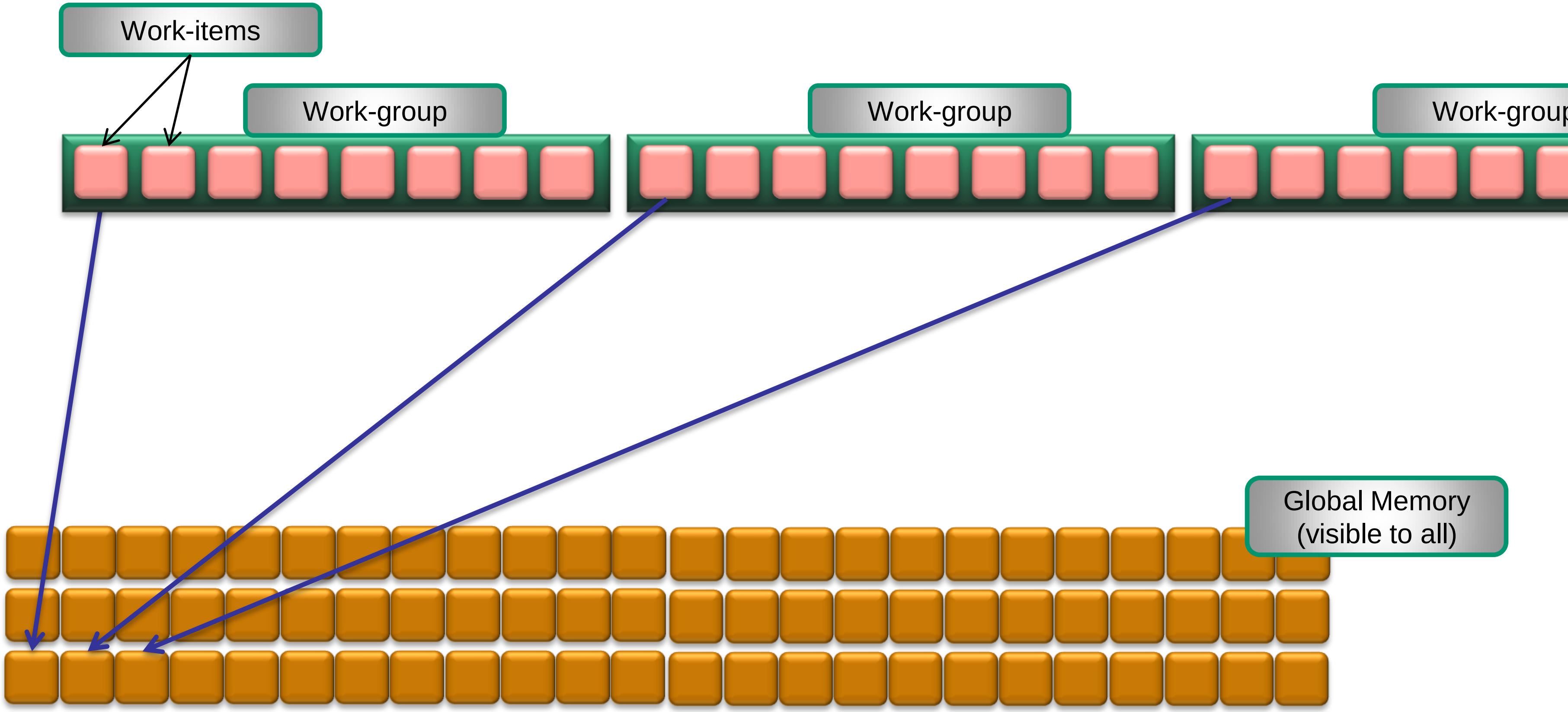
c



Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC'11 OpenCL tutorial

Alternative Reduction Kernel (5/5)

- The partial sums are then be added either on the host, or using another OpenCL reduction kernel (with a single work-group)



Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC'11 OpenCL tutorial

Alternative Reduction Kernel (source)

```
__kernel void
reduce( __global float4* input,
        __global float4* output,
        __local  float4* localmem)
{
  int tid = get_local_id(0);
  int bid = get_group_id(0);
  int gid = get_global_id(0);

  int localSize  = get_local_size(0);
  int globalSize = get_global_size(0);

  ...
  ...
```

```
  ...
  ...
A // load data to local memory
  localmem[tid]
     = input[gid] + input[gid + globalSize];
  barrier(CLK_LOCAL_MEM_FENCE);

B // repeat reduction in local memory
  for(int s = localSize/2; s > 1; s >>= 1)
  {
    if(tid < s)
      localmem[tid] += localmem[tid + s];

    // keep barrier outside conditional
    barrier(CLK_LOCAL_MEM_FENCE);
  }

C // write result to global memory
  if (tid == 0)
    output[bid] = localmem[0] + localmem[1];
}
```

Source: Udeepta Bordoloi of AMD provided these slides on Reduction for our ISC'11 OpenCL tutorial

Agenda

- **Heterogeneous computing and the origins of OpenCL**
- **OpenCL overview**
- **Mapping OpenCL onto CPUs**
- **Exploring the spec with code: embarrassingly parallel**
  - Vector addition: the basic platform layer
  - Matrix multiplication: kernel programming language
- **Exploring the spec with code: dealing with dependencies**
  - Optimizing matrix mul.: work groups and the memory model
  - Reduction: synchronization
- **A survey of the rest of OpenCL**

Events

- **An event is an object that communicates the status of commands in OpenCL … legal values for an event:**
  - **CL_QUEUED**: command has been enqueued.
  - **CL_SUBMITED**: command has been submitted to the compute device
  - **CL_RUNNING**: compute device is executing the command
  - **CL_COMPLETE**: command has completed
  - **ERROR_CODE**: a negative value, indicates an error condition occurred.
- **Can query the value of an event from the host … for example to track the progress of a command.**

```
cl_int clGetEventInfo (
        cl_event event,    cl_event_info param_name,
        size_t param_value_size,      void *param_value,
        size_t *param_value_size_ret)
```

- **Examples:**
  - **CL_EVENT_CONTEXT**
  - **CL_EVENT_COMMAND_EXECUTION_STATUS**
  - **CL_EVENT_COMMAND_TYPE**

Generating and consuming events

- Consider the command to enqueue a kernel. The last three arguments optionally expose events (NULL otherwise).

```
cl_int clEnqueueNDRangeKernel (
    cl_command_queue command_queue,
    cl_kernel kernel,    cl_uint work_dim,
    const size_t *global_work_offset,
    const size_t *global_work_size,
    const size_t *local_work_size,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list,
    cl_event *event)
```

- Number of events this command is waiting to complete before executing

- Array of pointers to the events being waited upon … Command queue and events must share a context.

- Pointer to an event object generated by this command.

Event: basic event usage

- Events can be used to impose order constraints on kernel execution.
- Very useful with out of order queues.

```
cl_event    k_events[2];

err = clEnqueueNDRangeKernel(commands, kernel1, 1,
        NULL, &global, &local, 0, NULL, &k_events[0]);

err = clEnqueueNDRangeKernel(commands, kernel2, 1,
        NULL, &global, &local, 0, NULL, &k_events[1]);

err = clEnqueueNDRangeKernel(commands, kernel3, 1,
        NULL, &global, &local, 2, k_events, NULL);
```

- Enqueue two kernels that expose events

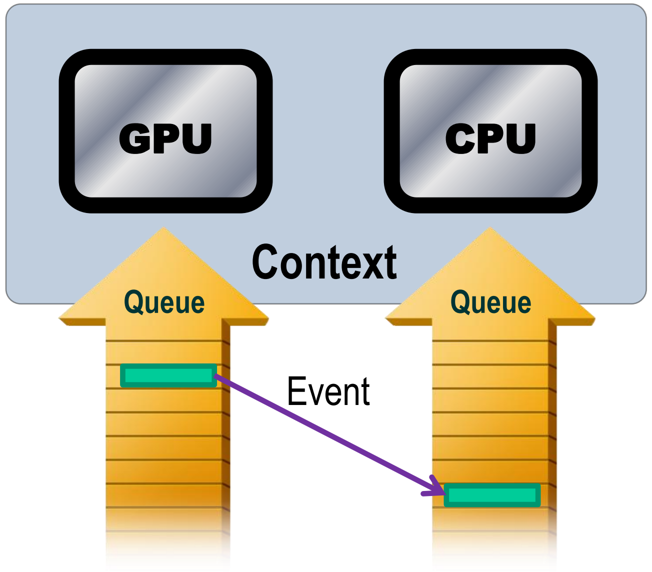
- Wait to execute until two previous events complete.

Why Events? Won't a barrier do?

- A barrier defines a synchronization point … commands following a barrier wait to execute until all prior enqueued commands complete

```
cl_int clEnqueueBarrier (
        cl_command_queue command_queue)
```

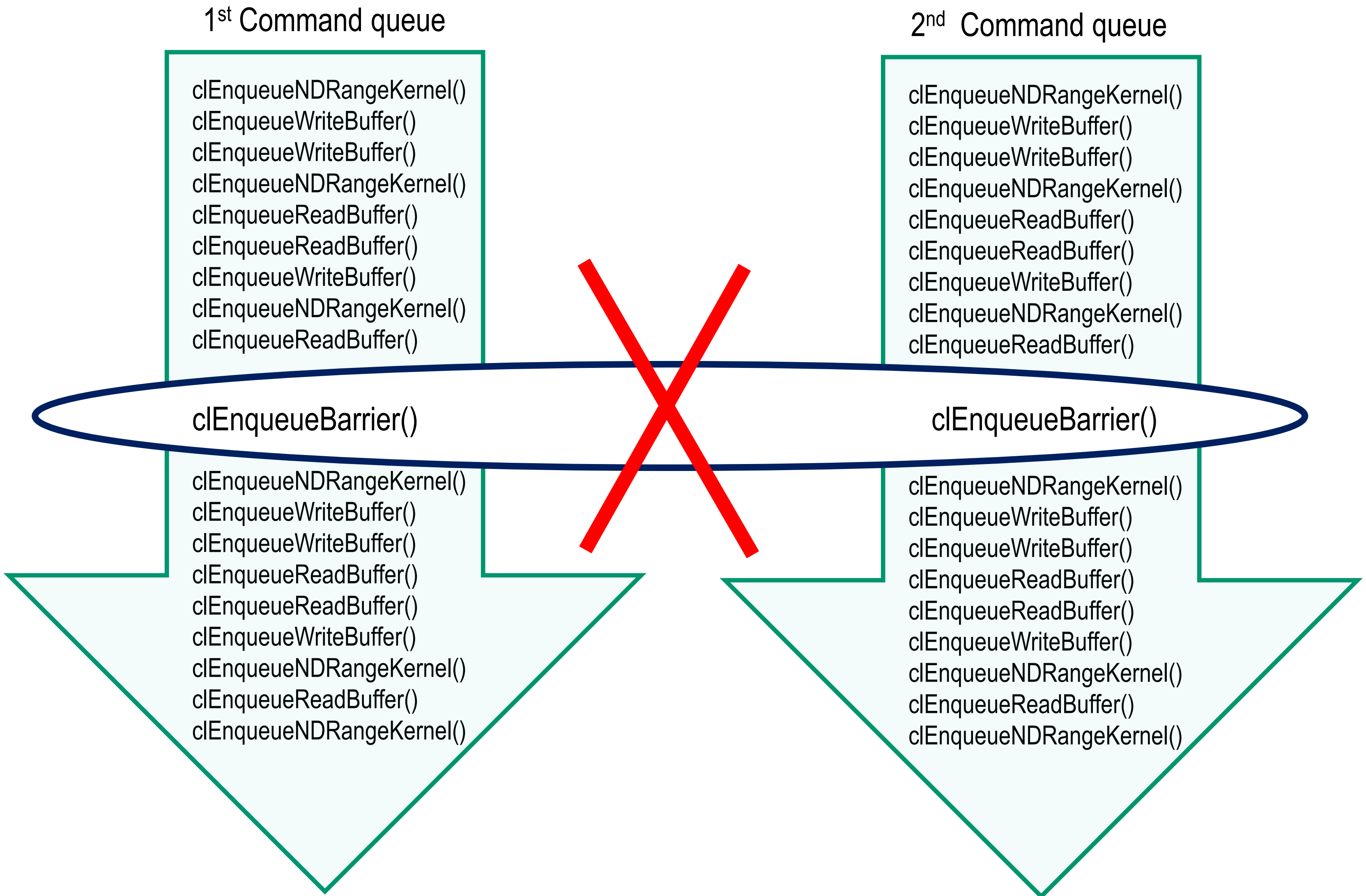
- Events provide fine grained control … this can really matter with an out of order queue.



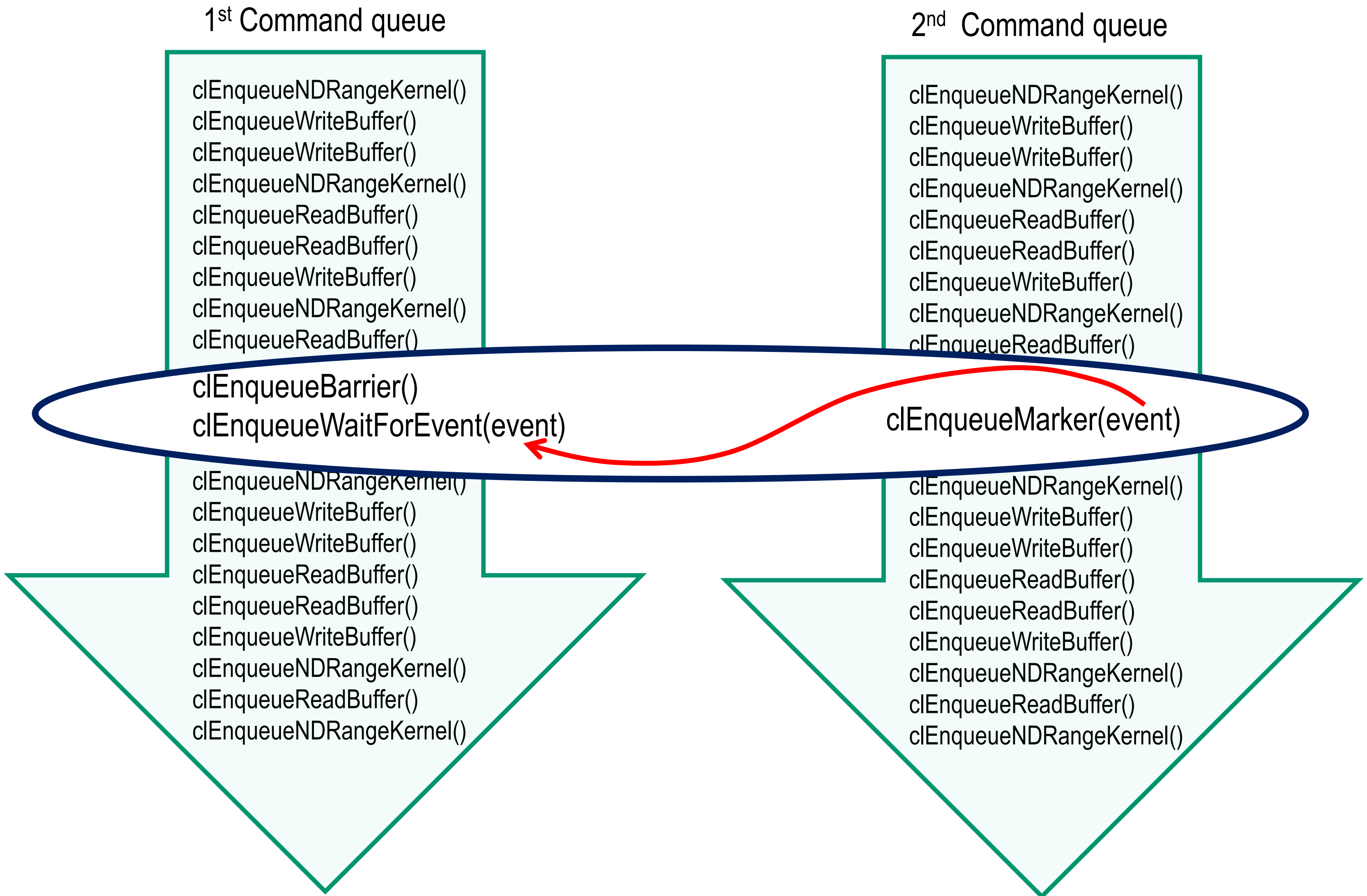
- Events work between commands in different queues … as long as they share a context!

- Events convey more information than a barrier ... Provide info on state of a command, not just weather its complete or not.

Barriers between queues: clEnqueueBarrier doesn't work



Barriers between queues: this works!



Host code influencing commands: User events

- “user code” running on a host thread can generate event objects

```
cl_event clCreateUserEvent (
        cl_context context,
        cl_int *errcode_ret)
```

- Created with value CL_SUBMITTED.
- It’s just another event to enqueued commands.
- Can set the event to one of the legal event values

```
cl_int clSetUserEventStatus (
      cl_event event,
      cl_int execution_status)
```

- Example use case: Queue up block of commands that wait on user input to finalize state of memory objects before proceeding.

Commands Influencing host code

- A thread running on the host can pause waiting on a list of events to complete. This is done with the function:

```
cl_int clWaitForEvents (
    cl_uint num_events,
    const cl_event *event_list)
```

- Number of events to wait on

- An array of pointers to event objects.

- Example use case: Host code waiting for an event to complete before extracting information from the event.

Profiling with Events

- **OpenCL is a performance oriented language … Hence performance analysis is an essential part of OpenCL programming.**
- **The OpenCL specification defines a portable way to collect profiling data.**
- **Can be used with most commands placed on the command queue … includes:**
  - Commands to read, write, map or copy memory objects
  - Commands to enqueue kernels, tasks, and native kernels
  - Commands to Acquire or Release OpenGL objects
- **Profiling works by turning an event into an opaque object to hold timing data.**

Using the Profiling interface

- Profiling is enabled when a queue is created with the CL_QUEUE_PROFILING_ENABLE flag is set.
- When profiling is enabled, the following function is used to extract the timing data

```
cl_int clGetEventProfilingInfo (
    cl_event event,
    cl_profiling_info param_name,
    size_t param_value_size,
    void *param_value,
    size_t *param_value_size_ret)
```

- Profiling data to query (see next slide)

- Expected and actual sizes of profiling data.

- Pointer to memory to hold results

cl_profiling_info values

- **CL_PROFILING_COMMAND_QUEUED**
  - the device time in nanoseconds when the command is enqueued in a command-queue by the host. (cl_ulong)
- **CL_PROFILING_COMMAND_SUBMIT**
  - the device time in nanoseconds when the command is submitted to compute device. (cl_ulong)
- **CL_PROFILING_COMMAND_START**
  - the device time in nanoseconds when the command starts execution on the device. (cl_ulong)
- **CL_PROFILING_COMMAND_END**
  - the device time in nanoseconds when the command has finished execution on the device. (cl_ulong)

Profiling Example

```
cl_event prof_event;
cl_command_queue comm;

comm = clCreateCommandQueue(
       context, device_id,
      CL_QUEUE_PROFILING_ENABLE,
       &err);

err = clEnqueueNDRangeKernel(
          comm, kernel,
          nd, NULL, global, NULL,
          0, NULL, prof_event);

clFinish(comm);
err = clWaitForEvents(1, &prof_event );
```

```
cl_ulong start_time, end_time;
size_t return_bytes;

err = clGetEventProfilingInfo(
    prof_event,
     CL_PROFILING_COMMAND_QUEUED,
     sizeof(cl_ulong),
     &start_time,
     &return_bytes);

err = clGetEventProfilingInfo(
     prof_event,
     CL_PROFILING_COMMAND_END,
     sizeof(cl_ulong),
     &end_time,
     &return_bytes);

run_time =(double)(end_time - start_time);
```

Events inside Kernels … Async. copy

```
// A, B, C kernel args … global  buffers.
// Bwrk is a local buffer

for(k=0;k<Pdim;k++)
        Awrk[k] = A[i*Ndim+k];

for(j=0;j<Mdim;j++){
   event_t ev_cp  = async_work_group_copy(
      (__local float*) Bwrk, (__global float*) B,
      (size_t) Pdim, (event_t) 0);

   wait_group_events(1, &ev_cp);

   for(k=0, tmp= 0.0;k<Pdim;k++)
         tmp  += Awrk[k] *  Bwrk[k];
   C[i*Ndim+j] = tmp;
}
```

- Compute a row of C = A * B:
  - 1 A col. per work-item
  - Work group shares rows of B

- Start an async. copy for row of B returning an event to track progress.

- Wait for async. copy to complete before proceeding.

- Compute element of C using A from private memory and B from local memory.

OpenCL 1.1 - API

- Thread-safety
  - All API calls, except **clSetKernelArg**, are thread safe
- Sub-buffer objects
  - Create an object that represents a specific region in a buffer object
  - Easy and efficient mechanism to distribute regions of a buffer object across multiple devices
  - OpenCL™ synchronization mechanism ensures modifications to sub-buffer object reflected in appropriate region of parent buffer object

OpenCL 1.1 - API

- User Events
  - **clEnqueue***** commands can wait on event
  - In OpenCL™ 1.0, events can only refer to OpenCL™ commands
  - Need ability to enqueue commands that wait on an external, user defined, event
- Event CallBacks
  - **clSetEventCallbackFn** to register a user callback function
  - called when command identified by event has completed
  - Allows applications to enqueue new OpenCL™ commands based on event state changes in a non-blocking manner
- Lot’s more API stuff too

OpenCL 1.1 - Language

- **Implicit Conversions**
  - OpenCL™ 1.0 requires widening for arithmetic operators

```
float4 a, b;
float c;
b = a + c; // c is widened to a float4 vector
           // first and then the add is performed
```

  - OpenCL™ 1.1 extends this feature for all operators
    - relational, equality, bitwise, logical, ternary

OpenCL 1.1 - Language

- **New built-in functions**
  - **get_global_offset**
  - **clamp** for integer data types (scalar and vector)
    - Constrains integers to fall between an input min,max range.
  - **async_work_group_strided_copy**
    - strided async copy of data from global <---> local memory
  - **shuffle** - construct a permutation of elements from 1 or 2 input vectors and a mask

OpenCL 1.1 – OpenCL/OpenGL Sharing

- **Improve performance of OpenCL/ OpenGL interoperability**
  - Portable OpenCL/ OpenGL sharing requires
    - a **glFinish** before **clEnqueueAcquireGLObjects**
    - a **clFinish** after **clEnqueueReleaseGLObjects**
  - **glFinish** / **clFinish** are heavyweight APIs

OpenCL 1.1 – OpenCL/OpenGL Sharing

- **Improve performance of OpenCL/ OpenGL interoperability**
  - Create a OpenCL event from an OpenGL sync object
  - Create a OpenGL sync object from a OpenCL event
  - Allows for a finer grained waiting mechanism
    - Use **event_wait_list** argument for events that refer to OpenGL commands to complete
    - Use OpenGL sync APIs to wait for specific OpenCL™ commands to complete

Conclusion

- **OpenCL defines a platform-API/framework for heterogeneous computing … not just GPGPU or CPU-offload programming.**

- **OpenCL has the potential to deliver portably performant code; but only if its used correctly:**
  - Implicit SIMD data parallel code has the best chance of mapping onto a diverse range of hardware … once OpenCL implementation quality catches up with mature shader languages.

- **The future is clear:**
  - Mixing task parallel and data parallel code in a single program … balancing the load among <u>ALL OF</u> the platform's available resources.