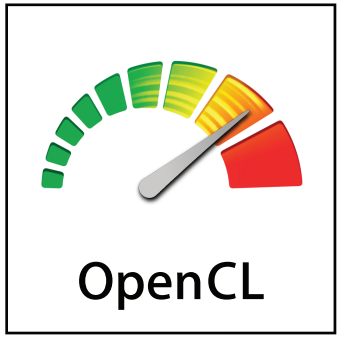
Portable Programs for Heterogeneous Computing: A Hands-on Introduction

Tim Mattson
Intel Corp.

Alice Koniges
Berkeley Lab/NERSC

Simon McIntosh-Smith
University of Bristol



Acknowledgements: James Price and Tom Deakin of the University of Bristol

Agenda

- Logistics
- Introduction to Heterogeneous computing
- OpenCL
  - Overview
  - Host Programs
  - Kernel Programs
  - Kernel code optimization
  - OpenCL ecosystem
- OpenMP
  - Overview
  - “Target” and “Target data” directives
  - Mapping onto GPUs: the distribute directive

IMPORTANT: Read This

- Disclosures
  - The views expressed in this tutorial are those of the people delivering the tutorial.
    - We are not speaking for our employers.
    - We are not speaking for the OpenMP ARB or the Khronos OpenCL language committee.
- We take these tutorials VERY seriously:
  - Help us improve … tell us how you would make this tutorial better.

OpenCL Learning progression (part 1)

Topic	Exercise	concepts
I. OCL intro		OpenCL overview, history and Core models.
II. Host programs	Vadd program	Understanding host programs
III. Kernel programs	Basic Jacobi solver	The OpenCL execution model and how it relates to kernel programs.
IV. Memory coalescence	Reorganizing the A matrix in the Jacobi solver program.	Memory layout effects on kernel performance
V. Divergent control flows	Divergent control flow in the Jacobi solver	Control flows and how they impact performance
VI. Occupancy	Work group size optimization for the Jacobi solver	Keeping all the resources busy
VII. Memory hierarchy in OpenCL	Demo: Matrix Multiplication	Working with private, local and global memory

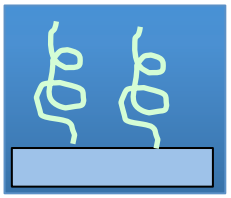
OpenMP Learning progression (part 2)

Topic	Exercise	concepts
I. OpenMP intro	Parallel Hello world	OpenMP overview and checking out our OpenMP environment
II. Core elements of traditional OpenMP	Pi program … parallel loops	Parallel loops and supporting constructs for shared memory systems
III. The target directive	Basic Jacobi solver	The host-device model
IV. Target data construct	Optimizing memory movement in the Jacobi solver program.	Data regions and optimizing the host-device model
V. OpenMP for GPUs	Jacobi optimizations	Working with leagues and the distribute clause

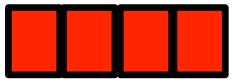
Agenda

- Logistics
- Introduction to Heterogeneous computing
- OpenCL
  - Overview
  - Host Programs
  - Kernel Programs
  - Kernel code optimization
  - OpenCL ecosystem
- OpenMP
  - Overview
  - “Target” and “Target data” directives
  - Mapping onto GPUs: the distribute directive

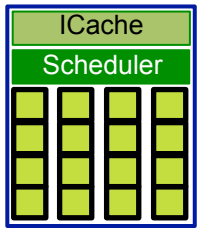
Hardware Diversity: Basic Building Blocks



CPU Core: one or more hardware threads sharing an address space. Optimized for low latencies.

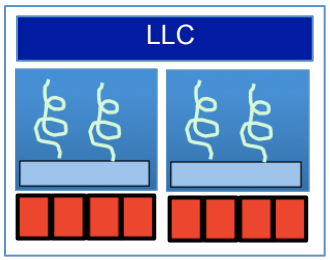


SIMD: Single Instruction Multiple Data.
Vector registers/instructions with 128 to 512 bits so a single stream of instructions drives multiple data elements.

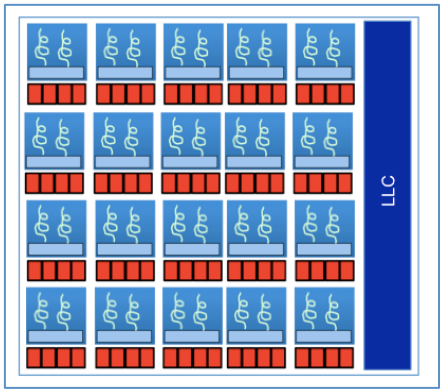


SIMT: Single Instruction Multiple Threads.
A single stream of instructions drives many threads. More threads than functional units. Over subscription to hide latencies. Optimized for throughput.

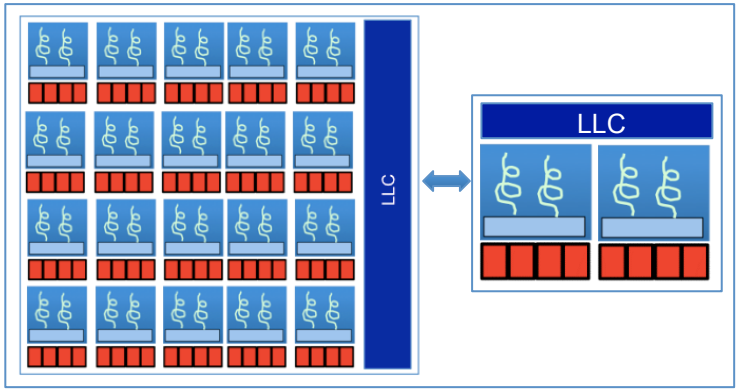
Hardware Diversity: Combining building blocks to construct nodes



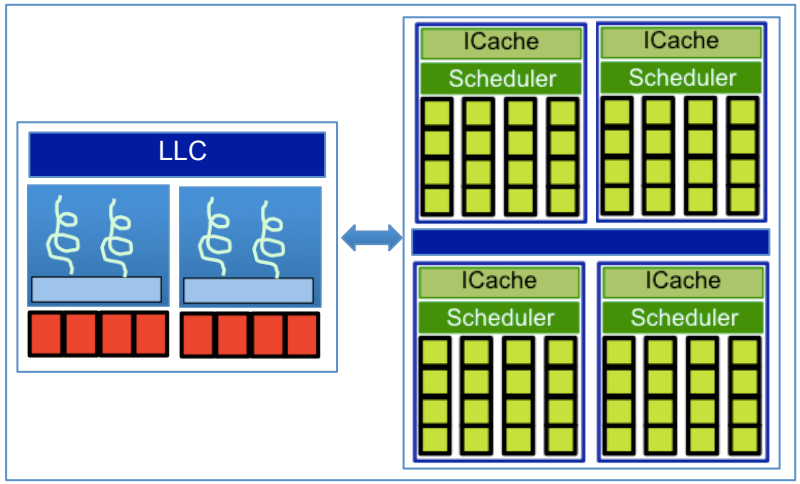
Multicore CPU



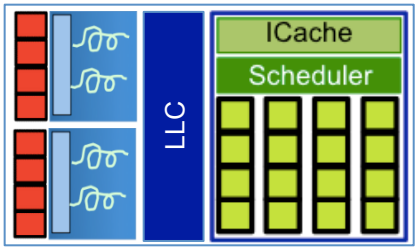
Manycore CPU



Heterogeneous:
CPU+manycore CPU

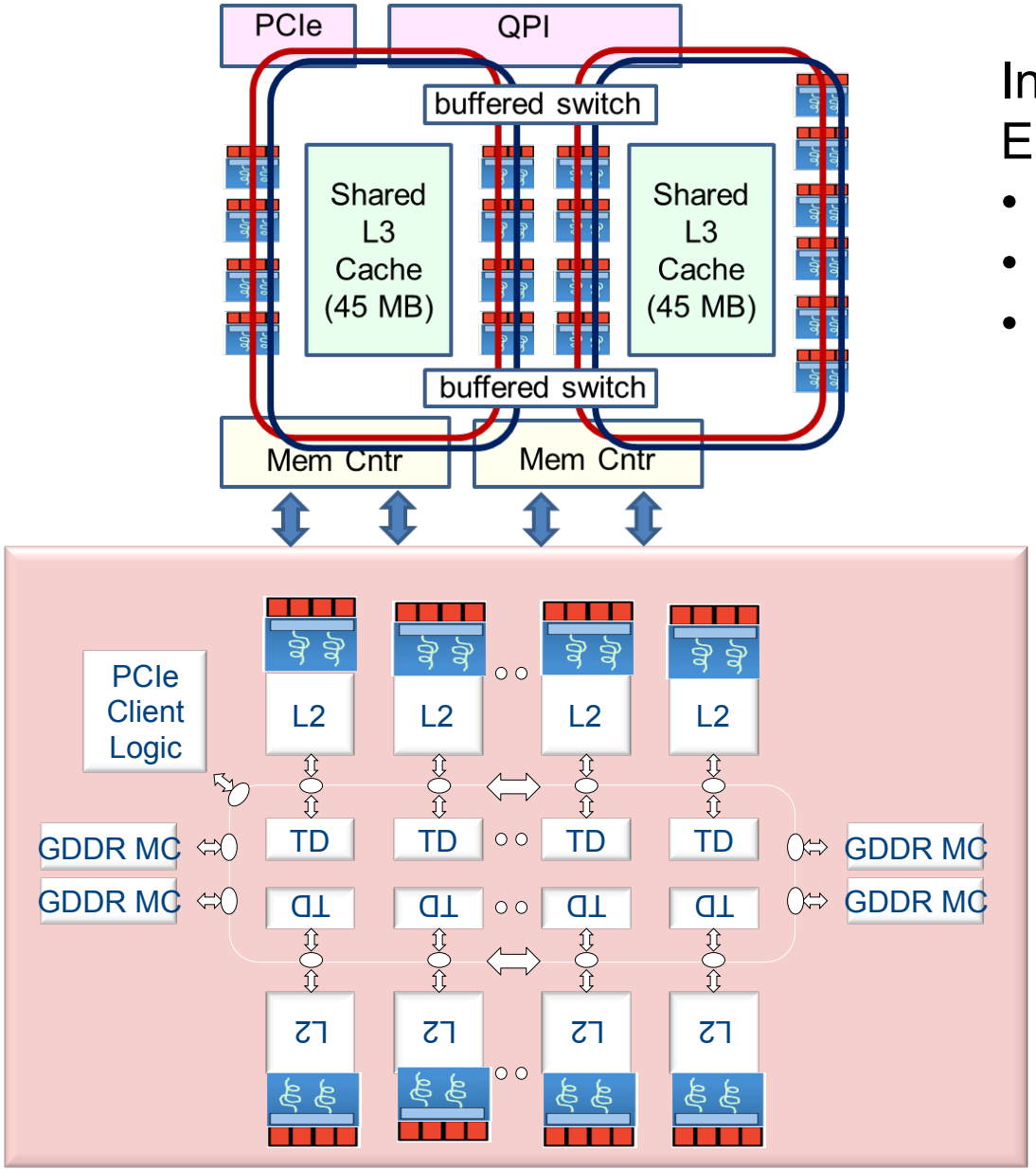


Heterogeneous: CPU+GPU



Heterogeneous:
Integrated CPU+GPU

Hardware diversity: CPUs



Intel® Xeon® processor
E7 v3 series (Haswell or HSW)

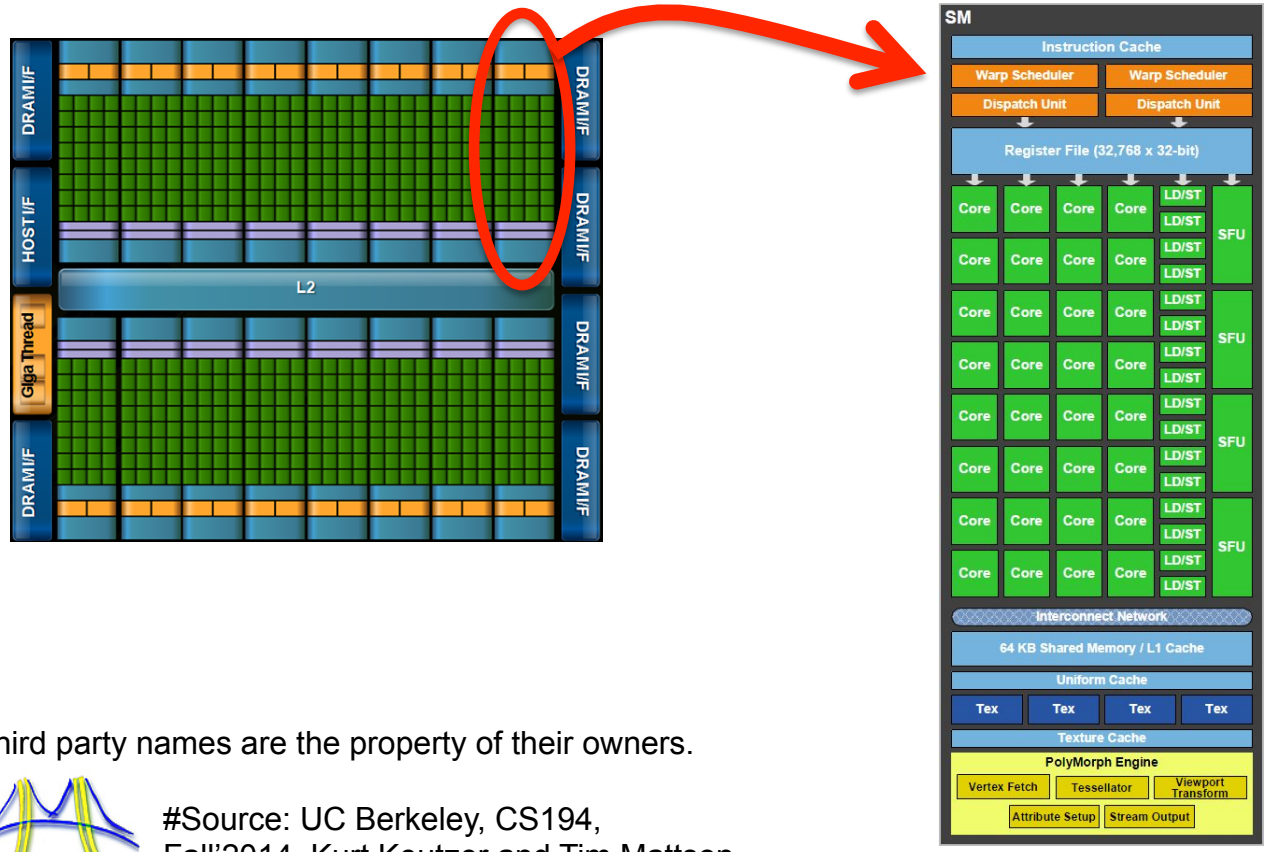
- 18 cores
- 36 Hardware threads
- 256 bit wide vector units

Intel® Xeon Phi™ coprocessor
(Knights Corner)

- 61 cores
- 244 Hardware threads
- 512 bit wide vector units

Hardware diversity: GPUs

- Nvidia® GPUs are a collection of “Streaming Multiprocessors” (SM)
  - Each SM is analogous to a core of a Multi-Core CPU
- Each SM is a collection of SIMD execution pipelines that share control logic, register file, and L1 Cache#

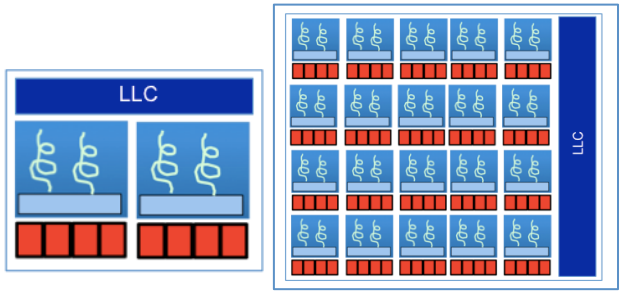


For example: an NVIDIA Tesla C2050 (Fermi) GPU with 3GB of memory and 14 streaming multiprocessor cores*.

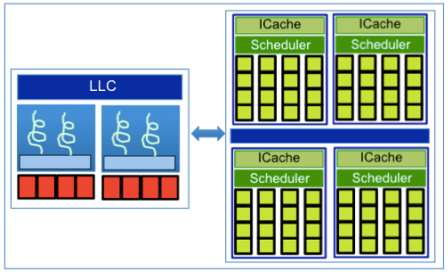
#Source: UC Berkeley, CS194, Fall’2014, Kurt Keutzer and Tim Mattson

*Source: http://www.nersc.gov/users/computational-systems/dirac/node-and-gpu-configuration/

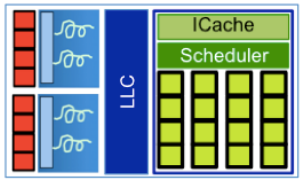
Hardware Diversity: programming models



OpenMP, OpenCL, pthreads, MPI, TBB, Cilk, C++'11…

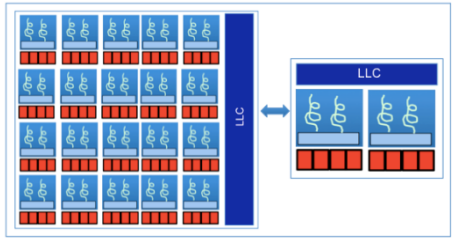


OpenMP, OpenCL, CUDA, OpenACC



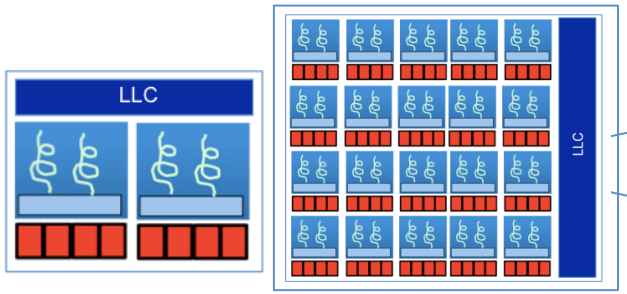
OpenMP, OpenCL,

Do you notice a trend?

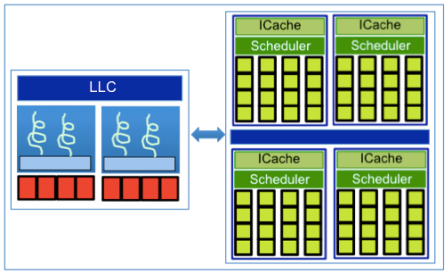


OpenMP, OpenCL, pthreads, TBB, Cilk, C++'11…

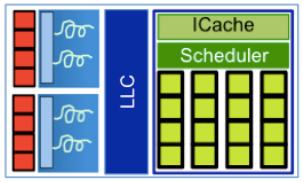
Hardware Diversity: programming models



OpenMP, OpenCL, pthreads, MPI, TBB, Cilk, C++'11…

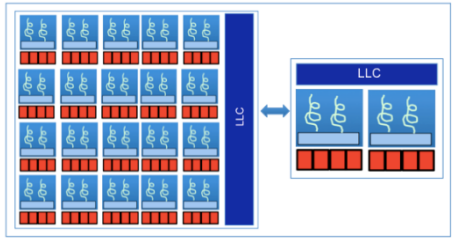


OpenMP, OpenCL, CUDA, OpenACC



OpenMP, OpenCL,

If you want to support the diversity of nodes in HPC from a single source-code base, you have only two choices: OpenMP and OpenCL

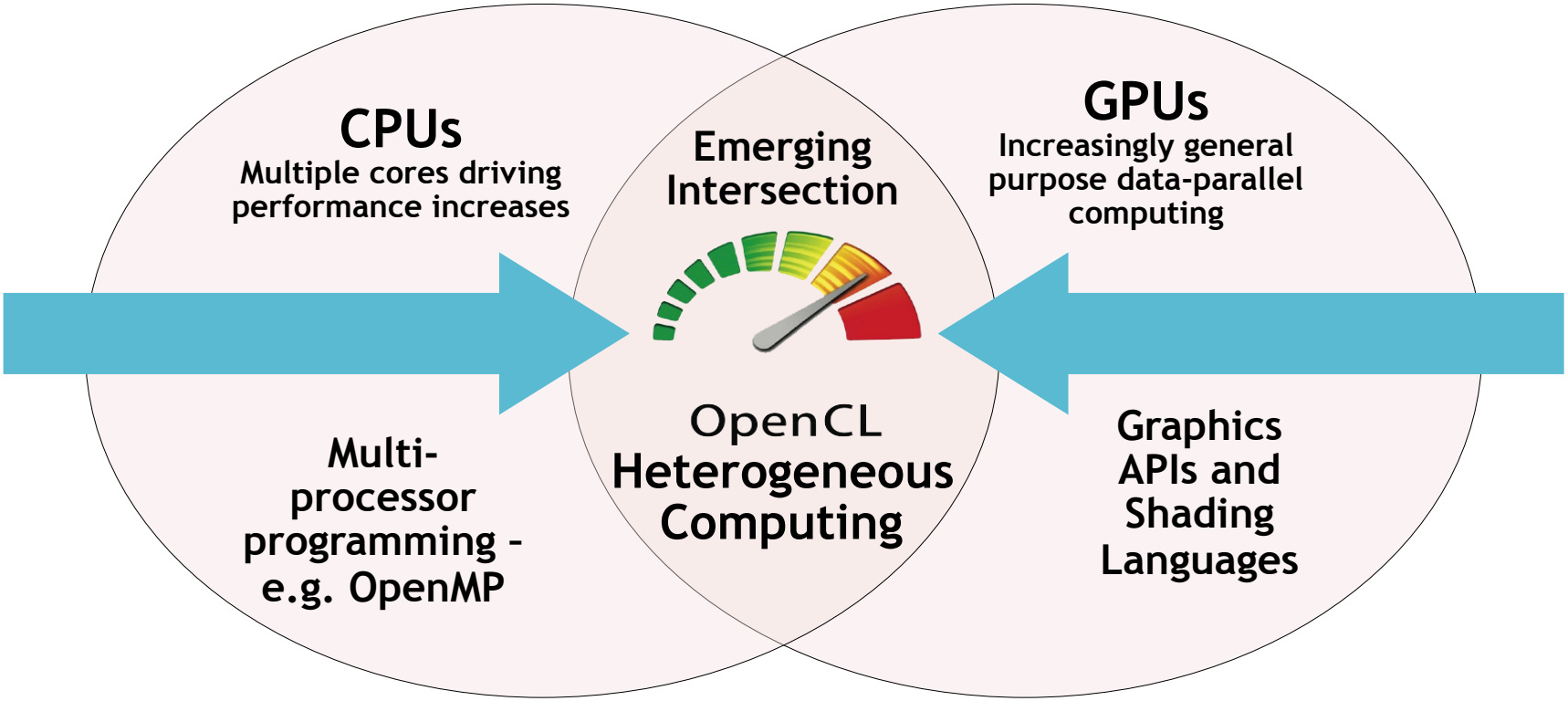


OpenMP, OpenCL, pthreads, TBB, Cilk, C++'11…

Agenda

- Logistics
- Introduction to Heterogeneous computing
- OpenCL
  - Overview
  - Host Programs
  - Kernel Programs
  - Kernel code optimization
  - OpenCL ecosystem
- OpenMP
  - Overview
  - “Target” and “Target data” directives
  - Mapping onto GPUs: the distribute directive

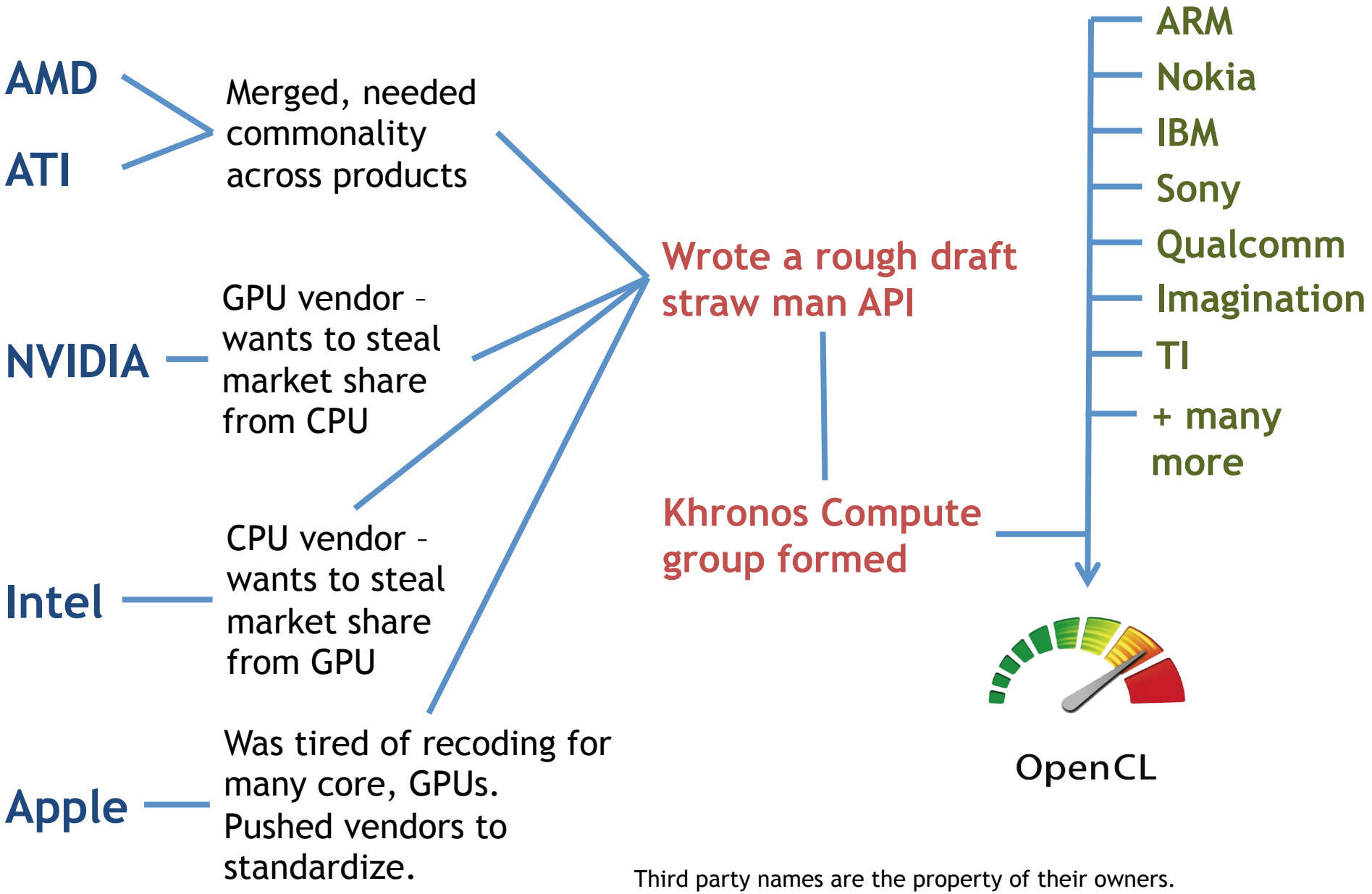
Industry Standards for Programming Heterogeneous Platforms



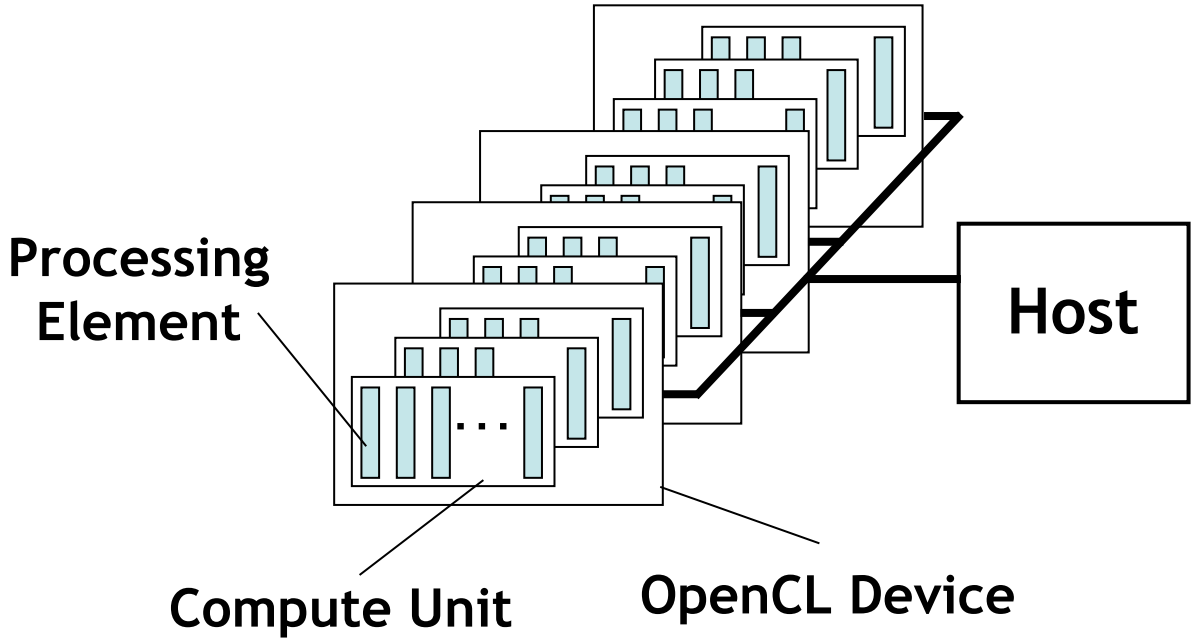
OpenCL – Open Computing Language

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

The origins of OpenCL



OpenCL Platform Model



- One *Host* and one or more *OpenCL Devices*
  - Each OpenCL Device is composed of one or more *Compute Units*
    - Each Compute Unit is divided into one or more *Processing Elements*
- Memory divided into *host memory* and *device memory*

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

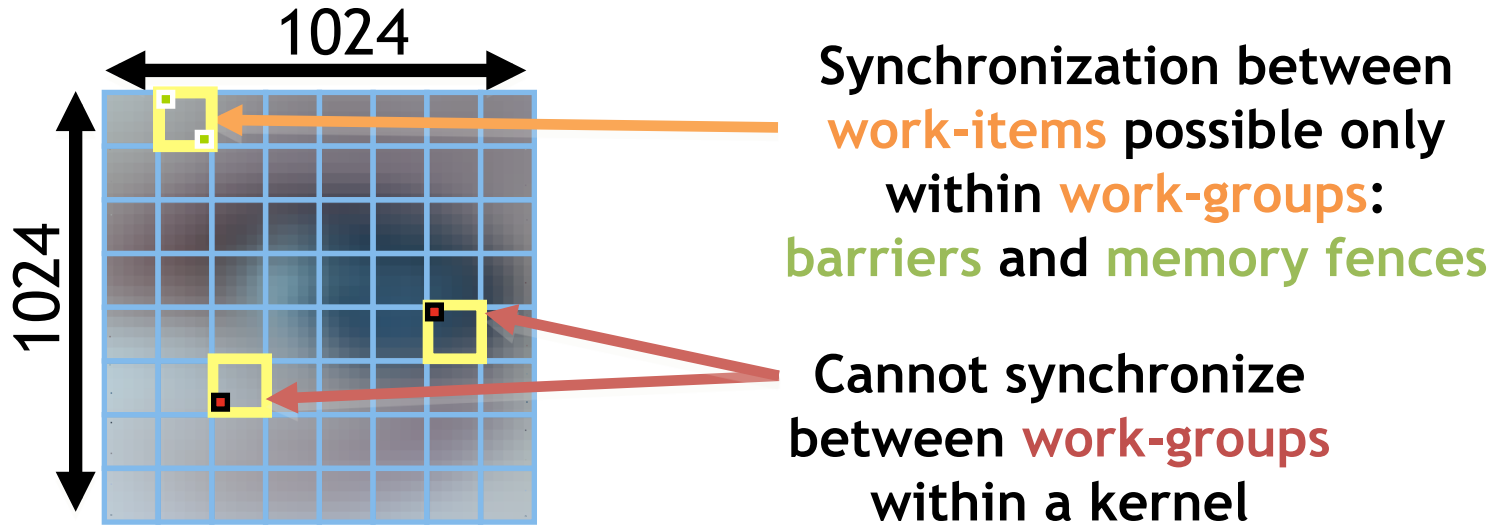
```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

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
```

An N-dimensional domain of work-items

- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 128x128 (**work-group**, executes together)

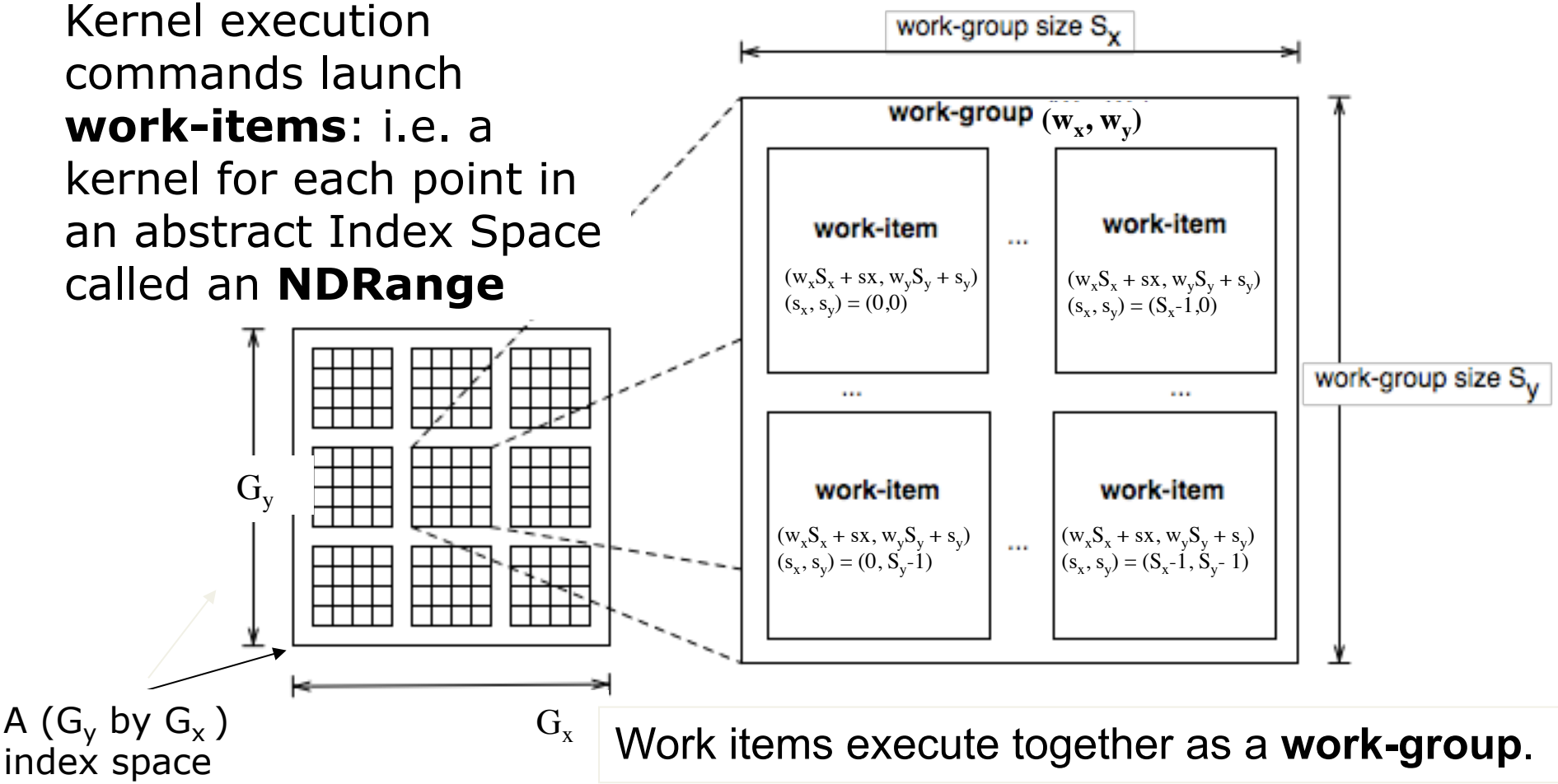


- Choose the dimensions that are “best” for your algorithm

Execution Model

- Host defines a **command queue** and associates it with a context (devices, kernels, memory, etc).
- Host enqueues commands to the command queue

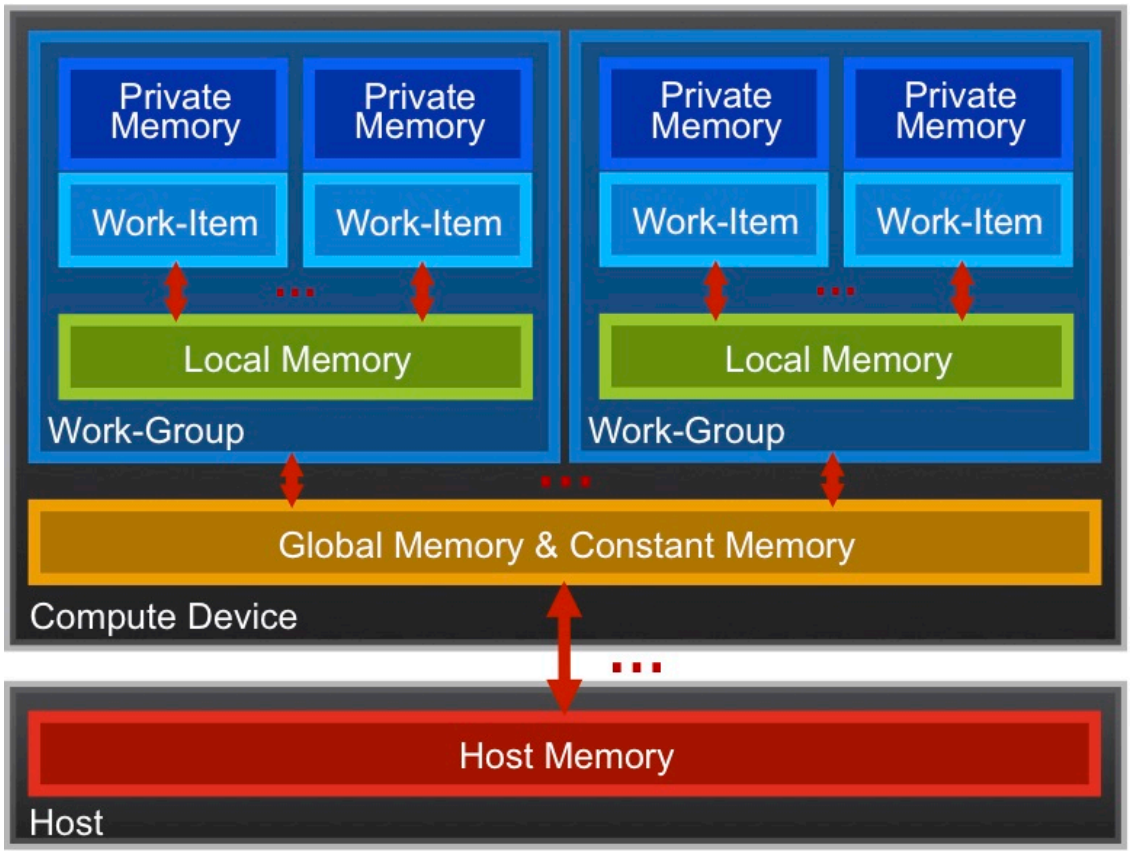
Kernel execution commands launch **work-items**: i.e. a kernel for each point in an abstract Index Space called an **NDRange**



Work items execute together as a **work-group**.

OpenCL Memory model

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



Memory management is **explicit**:
You are responsible for moving data from
host → global → local *and* back

Example: vector addition

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

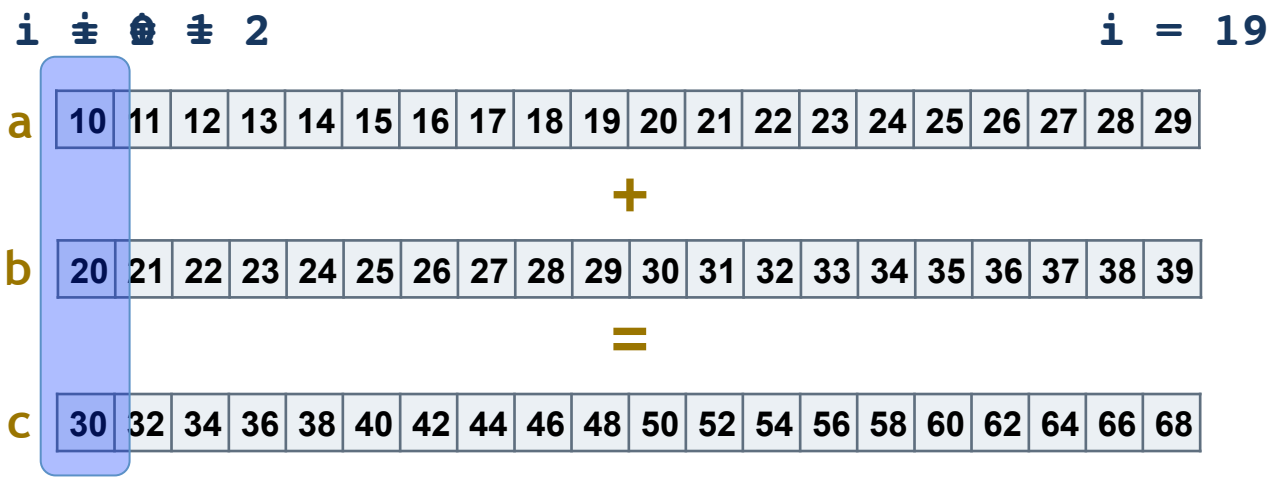
- C[i] = A[i] + B[i] for i=1 to N

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

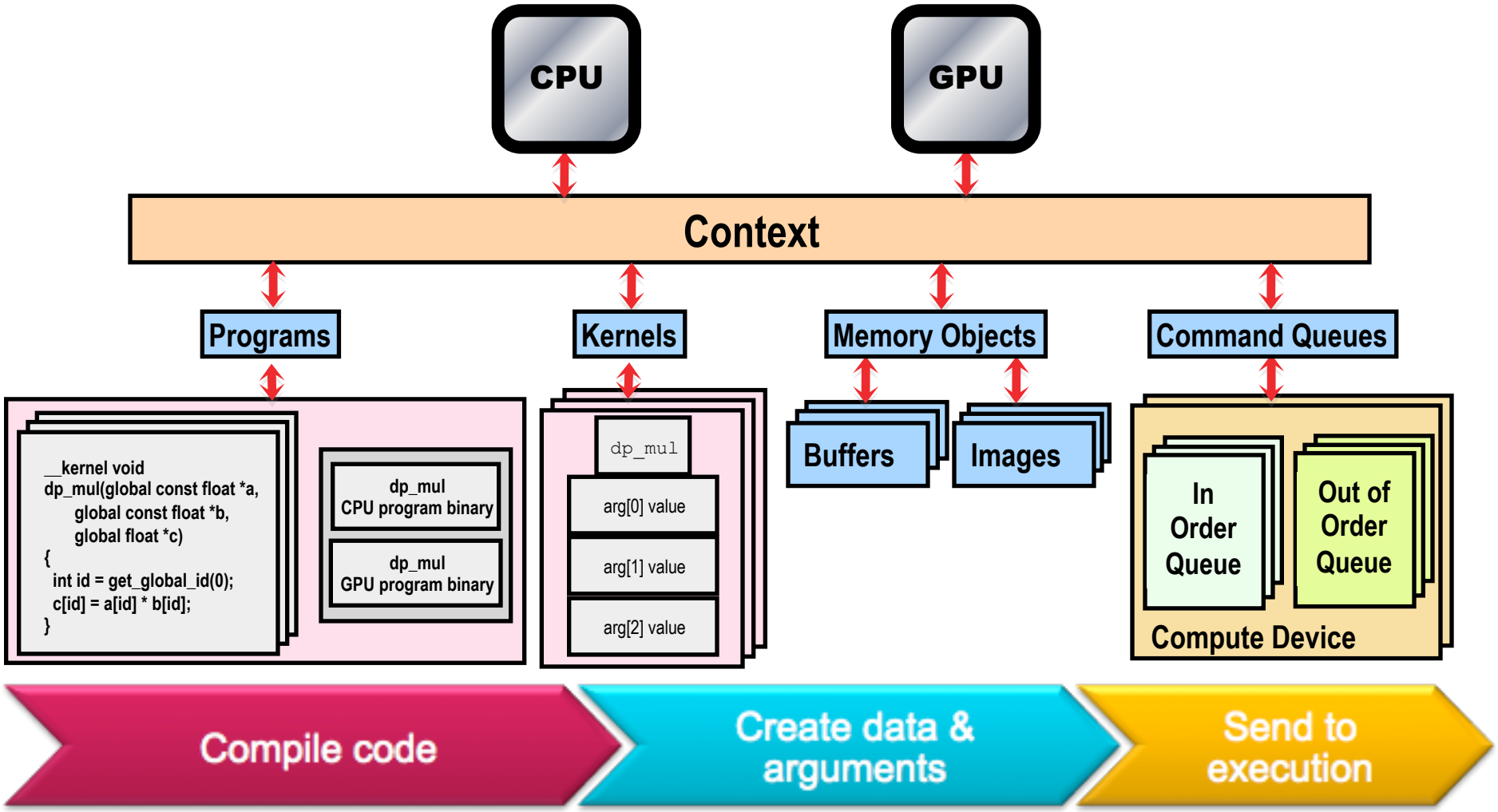
Execution model (kernels)

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

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



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



Agenda

- Logistics
- Introduction to Heterogeneous computing
- OpenCL
  - Overview
  - Host Programs
  - Kernel Programs
  - Kernel code optimization
  - OpenCL ecosystem
- OpenMP
  - Overview
  - “Target” and “Target data” directives
  - Mapping onto GPUs: the distribute directive

Vector Addition – Host

- The host program is the code that runs on the host to:
  - Setup the environment for the OpenCL program
  - Create and manage 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 the ***kernel*** (attach arguments to kernel function)
  5. Submit ***commands*** … transfer memory objects and execute kernels

As we go over the next set of slides, cross reference content on the slides to your reference card. This will help you get used to the reference card and how to pull information from the card and express it in code.

1. Define the platform

- **Grab the first available Platform:**

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

- **Use the first CPU device the platform provides:**

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

- **Create a simple context with a single device:**

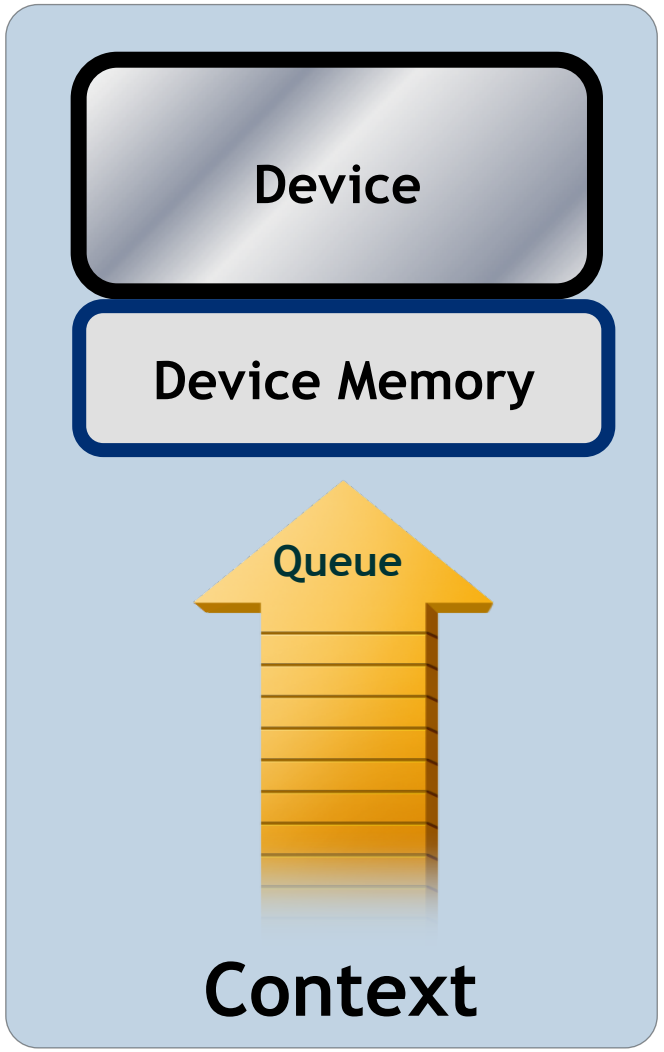
```
context = clCreateContext(firstPlatformId, 1, &device_id, NULL,
                                                NULL, &err);
```

- **Create a simple command queue to feed our compute device:**

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

Context and Command-Queues

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



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);
```

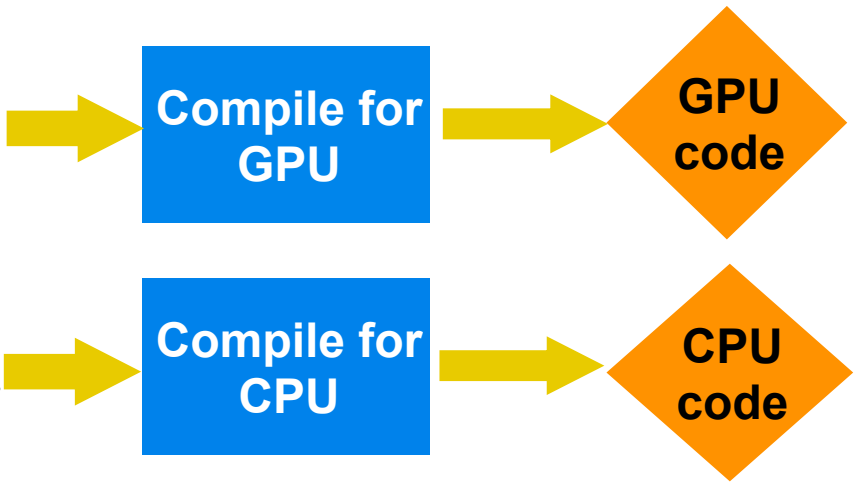
- **Fetch and print error messages (if (err != CL_SUCCESS)):**

```
size_t len;          char buffer[2048];

clGetProgramBuildInfo(program, device_id,
                              CL_PROGRAM_BUILD_LOG, sizeof(buffer),
                              buffer, &len);
printf("%s\n", buffer);
```

Run-time kernel compilation

- OpenCL uses **run-time compilation** … because in general you don't know what the target device will be when you ship the program

```
__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);
}
```



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 h_a[N], h_b[N], h_c[N];

for (i = 0; i < N; i++) {
        h_a[i] = rand() / (float)RAND_MAX;
        h_b[i] = rand() / (float)RAND_MAX;
}
```

- **Define OpenCL memory objects:**

```
d_a  = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(float) * N, NULL, NULL);
d_b  = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(float) * N, NULL, NULL);
d_c  = clCreateBuffer(context,  CL_MEM_WRITE_ONLY, sizeof(float) * N, NULL, NULL);
```

4. Define the kernel

- **Create kernel object from the kernel function “vadd”:**

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

- **Attach the “vadd” kernel's arguments to memory objects:**

```
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
```

5. Submit commands

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

```
err = clEnqueueWriteBuffer( commands, d_a, CL_FALSE, 0,
                            sizeof(float) * N, h_a, 0, NULL, NULL );
err = clEnqueueWriteBuffer( commands, d_b, CL_FALSE, 0,
                            sizeof(float) * N, h_b, 0, NULL, NULL );
```

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

```
size_t global[1] = {N};
err = clEnqueueNDRangeKernel( commands, kernel, 1, NULL,
                                        global, NULL, 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 finished before the read begins.**

```
err = clEnqueueReadBuffer( commands, d_c, CL_TRUE, 0,
                           sizeof(float) * N, h_c, 0, NULL, NULL );
```

Vector Addition – Host Program

```
// Find number of platforms
err = clGetPlatformIDs(0, NULL, &numPlatforms);

// Get all platforms
cl_platform_id Platform[numPlatforms];
err = clGetPlatformIDs(numPlatforms, Platform, NULL);

// Find a device
for
  e
  i
}

// Create a context for 1 device
context = clCreateContext(0, 1, &device_id, NULL,NULL,&err);

// Create a command queue for our device
commands = clCreateCommandQueue(context, device_id, 0, &err);
```

1) Define the platform:

Devices + Context + Queues

```
// Create the compute program from the source buffer
program = clCreateProgramWithSource(context, 1,

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

2) Create and build the program

```
// Create the input and output arrays in device memory
d_a = clCreateBuffer(context, CL_MEM_READ_ONLY,
                     sizeof(float) * N, NULL, NULL);
d_b
d_c
                     sizeof(float) * N, NULL, NULL);
```

3) Define memory objects

```
// Create the compute kernel from the program
ko_vadd = clCreateKernel(program, "vadd", &err);

// Set
err  =                                          d_a);
err |=                                          &d_b);
err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_c);
```

4) Create and setup kernel

```
// Write a and b vectors into device global memory
err = clEnqueueWriteBuffer(commands, d_a, CL_TRUE, 0,
                           sizeof(float) * N, h_a, 0, NULL, NULL);

err =                                                    0,

// Exec
err =                                                    ize
                                                         NULL,
                                                         ULL);

// Read back the results from device global memory
err = clEnqueueReadBuffer(commands, d_c, CL_TRUE, 0,
            sizeof(float) * N, h_c, 0, NULL, NULL );
```

5) Execute commands:

- **a) Write buffers to device**
- **b) Execute the kernel**
- **c) Read buffer back to host**

It's complicated, but most of this is "boilerplate" and the same for most codes.

Vector add kernel code

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

Exercise 1: Running the Vadd kernel

- Goal:
  - To inspect and verify that you can run an OpenCL kernel
- Procedure:
  - Use the vadd.c program we provide. It will run a simple kernel to add two vectors together.
  - Look at the host code and identify the API calls in the host code.
- Expected output:
  - A message verifying that the vector addition completed successfully
- Extra:
  - Try the DeviceInfo example, print out information about the OpenCL devices in the system.

Logging in to BlueCrystal

- University of Bristol supercomputer
- 341 16-core x86 nodes
- 76 NVIDIA K20 GPUs with OpenCL

Logging in to BlueCrystal

- Get inside the firewall:

```
ssh workshop@hpc.cs.bris.ac.uk
```

Password: `An3shmeokit`

- Then ssh into BlueCrystal with your username

```
ssh train01@bluecrystalp3.acrc.bris.ac.uk
```

Password: `workshop`

- Change directory: `cd OpenCL`

- Submit jobs to the queue

```
qsub submit_deviceinfo
```

Agenda

- Logistics
- Introduction to Heterogeneous computing
- OpenCL
  - Overview
  - Host Programs
  - Kernel Programs
  - Kernel code optimization
  - OpenCL ecosystem
- OpenMP
  - Overview
  - “Target” and “Target data” directives
  - Mapping onto GPUs: the distribute directive

Kernel programming

- Kernel programming is where all the action is at in OpenCL
- Writing simple OpenCL kernels is quite easy, so we'll cover that quickly
- Optimizing OpenCL kernels to run really fast is much harder, so that's where we're going to spend some time

OpenCL C kernel language

- Derived from **ISO C99**
  - A few ***restrictions***: no recursion, function pointers, functions in C99 standard headers ...
  - Preprocessing directives defined by C99 are supported (#include etc.)

- 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

OpenCL C Language Highlights

- Function qualifiers
  - **__kernel** qualifier declares a function as a kernel
    - I.e. makes it visible to host code so it can be enqueued
  - Kernels can call other device-side functions
- Address space qualifiers
  - **__global, __local, __constant, __private**
  - Pointer kernel arguments must be declared with an address space qualifier

OpenCL C Language Highlights

Work-item functions:

- uint **get_work_dim()** **number of dimensions (1, 2, or 3)**
- size_t **get_global_id(**uint **n)** **global work-item ID in dim. “n”**
- size_t **get_local_id(**uint **n)** **local work-item ID in dim. “n**
- size_t **get_group_id(**uint **n)** **ID of work-group in dim. “n”**
- size_t **get_global_size(**uint **n)** **num. of work-items in dim. “n”**
- size_t **get_local_size(**uint **n)** **work group size in dim. “n”**

OpenCL C Language Highlights

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

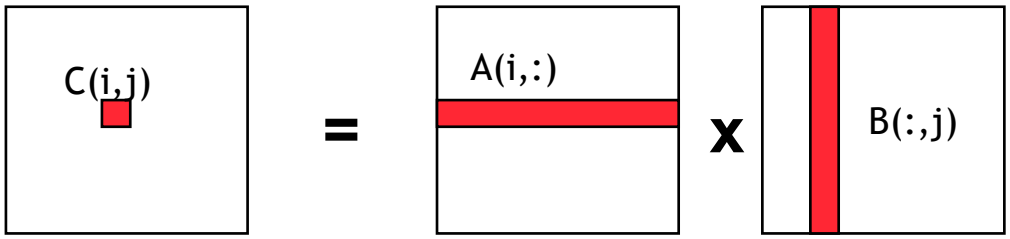
- Pointers to functions are ***not*** allowed
- Pointers to pointers allowed ***within*** a kernel, but not as an argument to a kernel invocation
- Bit-fields are not supported
- Variable length arrays and structures are not supported
- Recursion is not supported (yet!)
- Double types are optional in OpenCL v1.2, but the key word is reserved

  (note: most implementations support double)

Matrix multiplication: sequential code

We calculate C=AB, where all three matrices are NxN

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
                // C(i, j) = sum(over k) A(i,k) * B(k,j)
                C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```



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

Matrix multiplication performance

- Serial C code on CPU (single core).

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A

Device is Intel® Xeon® CPU, E5649 @ 2.53GHz using the gcc compiler.

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)

{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
              // C(i, j) = sum(over k) A(i,k) * B(k,j)
              C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

Matrix multiplication: OpenCL kernel (1/2)

```
__kernel void mat_mul(const int N, __global float *A,
                      __global float *B, __global float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
              // C(i, j) = sum(over k) A(i,k) * B(k,j)
              C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

Mark as a kernel function and specify memory qualifiers

Matrix multiplication: OpenCL kernel (2/2)

```
__kernel void mat_mul(const int N, __global float *A,
                      __global float *B, __global float *C)
{
    int i, j, k;
    i = get_global_id(0);
        j = get_global_id(1);
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
              // C(i, j) = sum(over k) A(i,k) * B(k,j)
              C[i*N+j] += A[i*N+k] * B[k*N+j];
            }

}
```

Replace loops with the work item's global id

Matrix multiplication: kernel cleaned-up

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

```
__kernel void mmul(
    const int N,
    __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 = 0.0f;
  for (k = 0; k < N; k++)
    tmp += A[i*N+k]*B[k*N+j];

  C[i*N+j] = tmp;
}
```

Jacobi solver serial code

```
conv = LARGE;  iters = 0;  xnew = x1;  xold = x2;
while ((conv > TOLERANCE) && (iters<MAX_ITERS)) {
  iters++;
  xtmp = xnew;  xnew = xold;  xold = xtmp;  // swap pointers

  for (i=0; i<Ndim; i++) {
      xnew[i] = (TYPE) 0.0;
      for (j=0; j<Ndim;j++) {
          if (i!=j)
            xnew[i]+= A[i*Ndim + j]*xold[j];
      }
      xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
  }
  // test convergence
  conv = 0.0;
  for (i=0; i<Ndim; i++) {
      tmp  = xnew[i]-xold[i];
      conv += tmp*tmp;
  }
  conv = sqrt((double)conv);
}
```

Exercise 2: Jacobi Solver Program

- Goal:
  - To write a non-trivial OpenCL kernel
- Procedure:
  - Look at the program Jac_solv_ocl_basic.c
  - We provide a C host program and the function prototype for the Jacobi_solver kernel program.
  - Write the body of the kernel program.
- Expected output:
  - A message verifying that the program ran correctly.

Jacobi solver kernel code (1/2)

```
#define TYPE double
#if (TYPE == double)
    #pragma OPENCL EXTENSION cl_khr_fp64 : enable
#endif

__kernel void jacobi(
          const unsigned Ndim,
          __global TYPE * A,  __global TYPE * b,
          __global TYPE * xold,  __global TYPE * xnew)
{
      size_t i = get_global_id(0);

      xnew[i] = (TYPE) 0.0;
      for (int j = 0; j < Ndim; j++)  {
            if (i != j)
                  xnew[i] += A[i*Ndim + j] * xold[j];
      }
      xnew[i] = (b[i] - xnew[i]) / A[i*Ndim + i];
}
```

Jacobi solver kernel code (2/2)

```
__kernel void convergence(
        __global TYPE * xold,  __global TYPE * xnew,
        __local TYPE * conv_loc,  __global TYPE * conv  )
{
   size_t i = get_global_id(0);
   TYPE tmp;
   tmp = xnew[i] - xold[i];
   conv_loc[get_local_id(0)] = tmp * tmp;
   barrier(CLK_LOCAL_MEM_FENCE);

   for (int offset = get_local_size(0) / 2; offset > 0; offset /= 2)  {
      if (get_local_id(0) < offset)    {
         conv_loc[get_local_id(0)] += conv_loc[get_local_id(0) + offset];
      }
      barrier(CLK_LOCAL_MEM_FENCE);
   }
   if (get_local_id(0) == 0)  {  conv[get_group_id(0)] = conv_loc[0];  }
}
```

A kernel enqueued on the host to compute convergence. This implements a reduction with the last stage of the reduction occurring on the host.

Jacobi Solver Results

- Serial code running on the Intel® Xeon® CPU and icc took 83 seconds.
- With OpenMP for multithreading
  - 25.3 seconds with 32 threads (hyperthreading enabled)
  - 19.0 seconds with 16 threads (hyperthreading disabled)
- Running the OpenMP version natively on the Intel® Xeon® Phi Processor took 4.8 seconds.

Different versions of the Jacobi Solver with OpenCL. Runtimes in seconds				
TYPE = double NDIM = 4096	**Nvidia K40 GPU**	**AMD 290X GPU**	**Intel Xeon PHI processor**	**Intel Xeon processor**
Basic	35.0	198.2	245.2	23.6

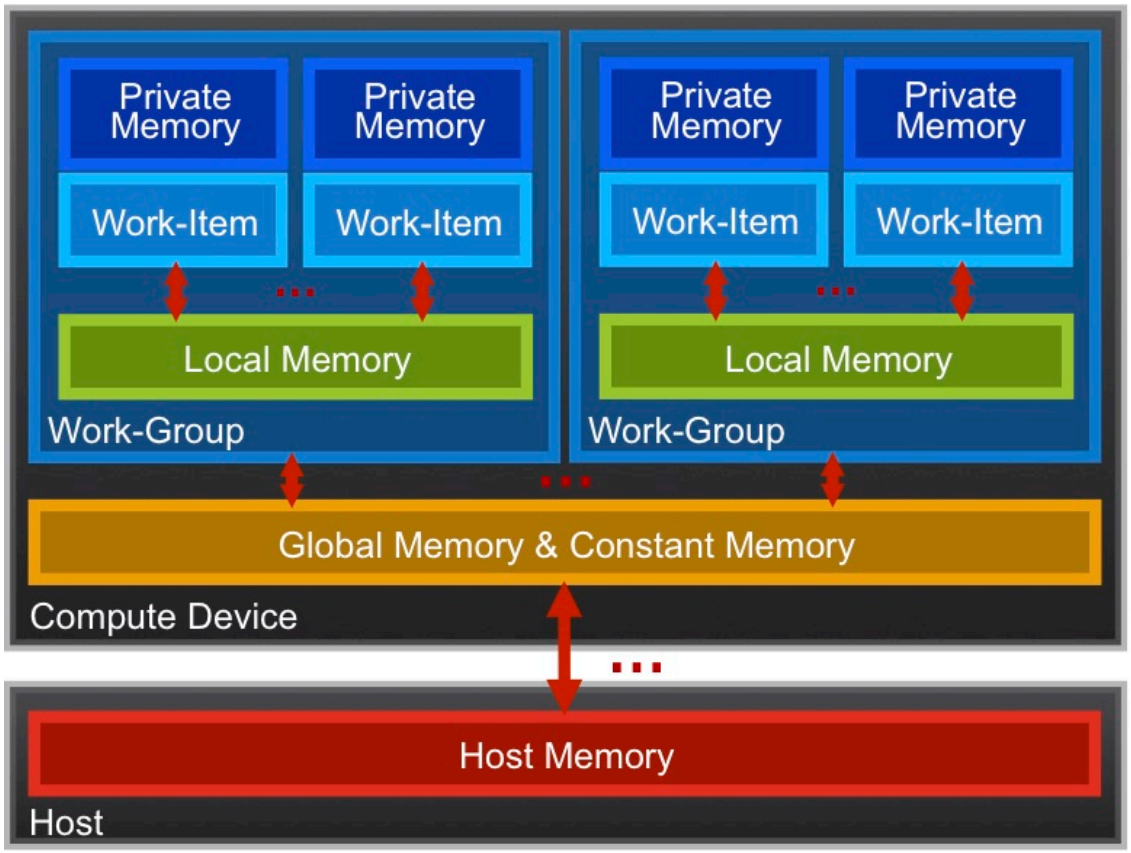
Dual-socket Intel® Xeon® CPU E5-2687W (16 cores total, hyper-threading enabled) and the Intel® icc compiler.

Agenda

- Logistics
- Introduction to Heterogeneous computing
- OpenCL
  - Overview
  - Host Programs
  - Kernel Programs
  - Kernel code optimization and compilation
    - Memory coalescence
    - Divergent control flows
    - Occupancy
    - Other Optimizations
    - Working with the OpenCL Memory Hierarchy
  - OpenCL ecosystem
- OpenMP

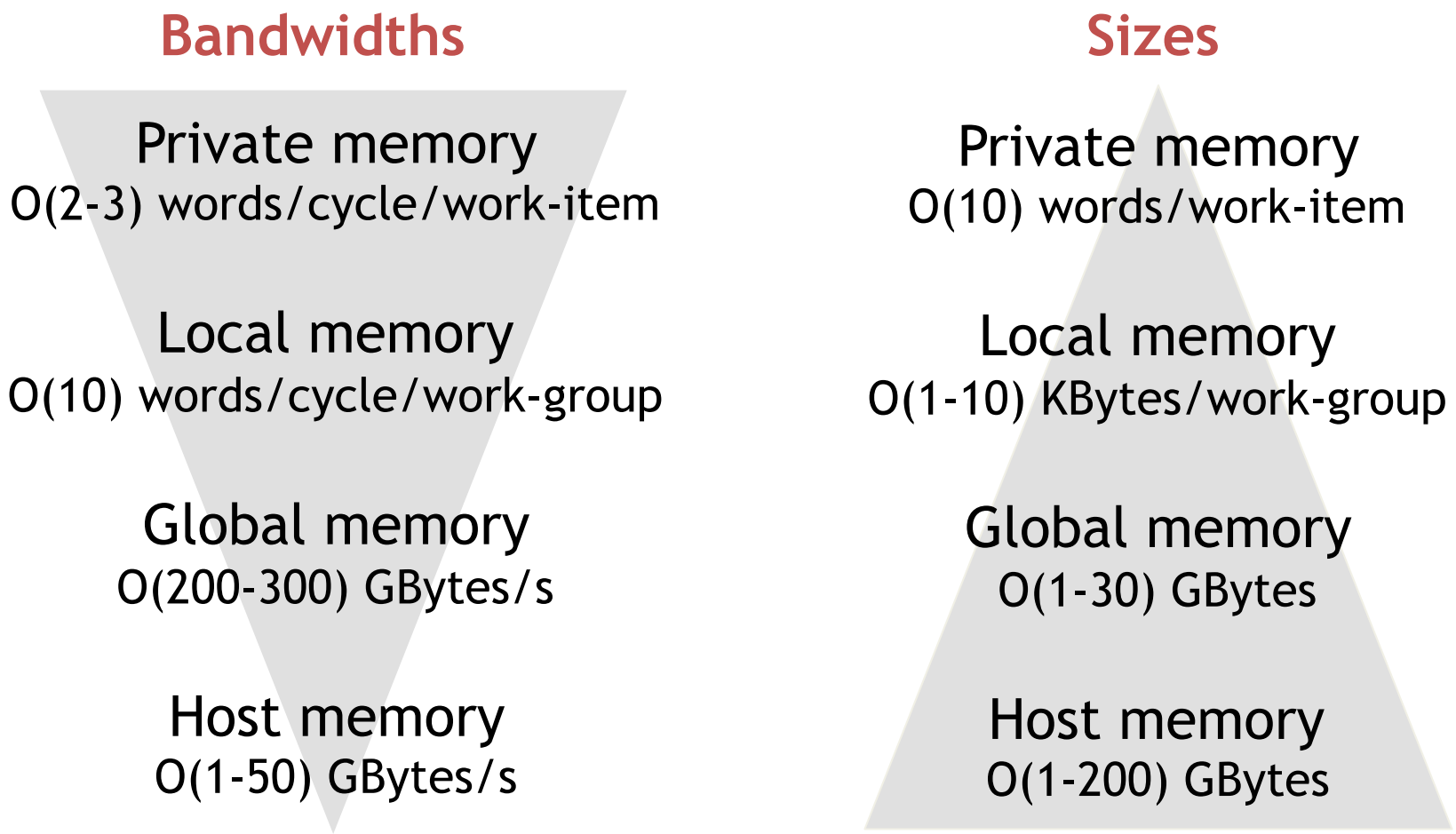
OpenCL Memory model

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



Memory management is **explicit**:
You are responsible for moving data from
host → global → local *and* back

The Memory Hierarchy



Managing the memory hierarchy is one of ***the*** most important things to get right to achieve good performance

*Size and performance numbers are approximate and for a high-end discrete GPU, circa 2015

Coalesced Access

- ***Coalesced memory accesses*** are key for high performance code
- In principle, it's very simple, but frequently requires transposing/ transforming data on the host before sending it to the GPU
- Sometimes this is an issue of AoS vs. SoA

Memory layout is critical to performance

- "Structure of Arrays vs. Array of Structures"
- Array of Structures (AoS) more natural to code

```
struct Point{ float x, y, z, a; };
Point *Points;
```

- Structure of Arrays (SoA) suits memory coalescence in vector units

```
struct { float *x, *y, *z, *a; } Points;
```

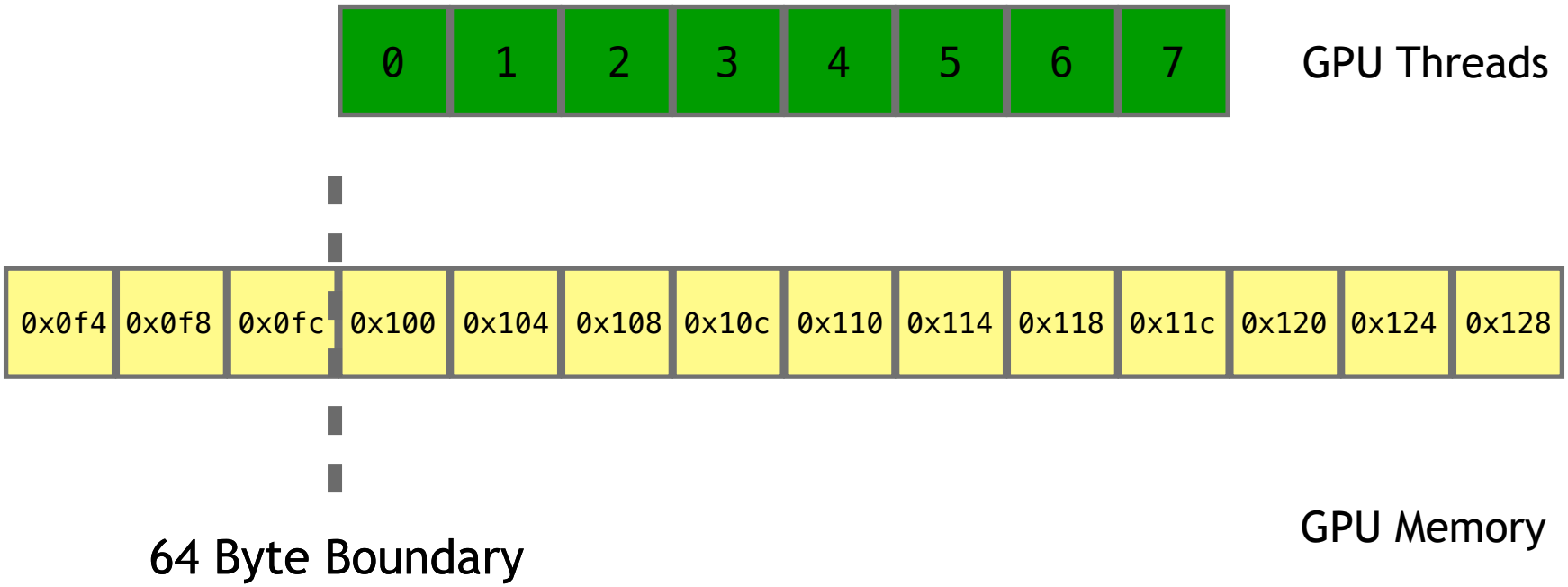
Adjacent work-items/ vector-lanes like to access adjacent memory locations

Coalescence

- Coalesce - to combine into one
- Coalesced memory accesses are key for high bandwidth
- Simply, it means, if thread *i* accesses memory location *n* then thread *i+1* accesses memory location *n+1*
- In practice, it's not quite as strict…

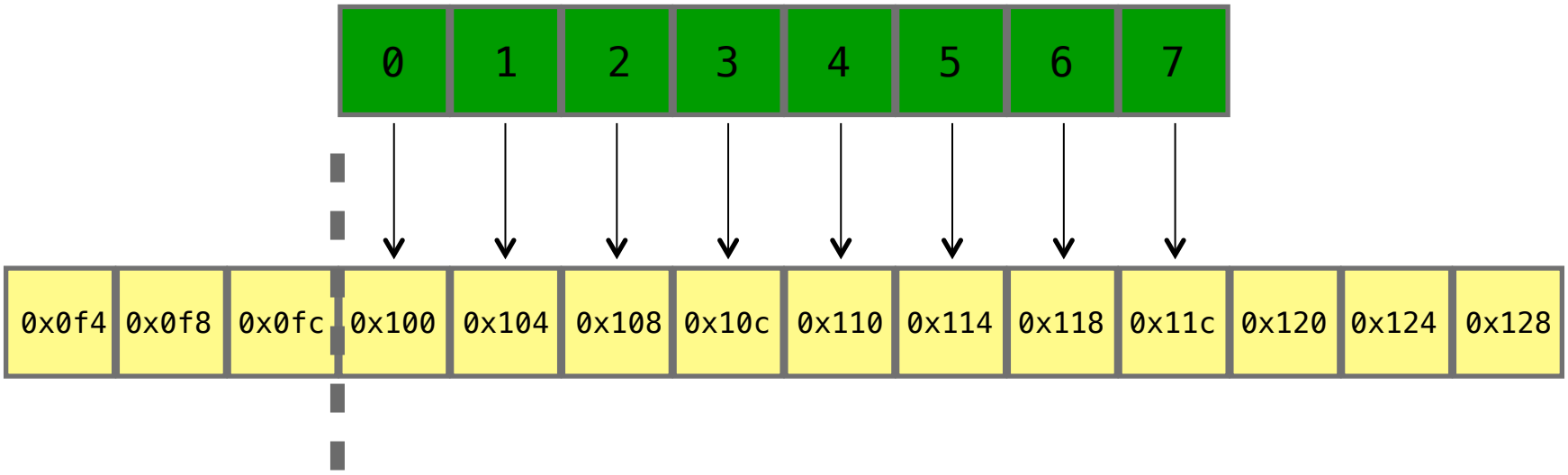
```
__kernel func( __global float *memA,
               __global float *memB)
{
        int g_id = get_global_id(0);

        // ideal
        float val1 = memA[g_id];

        // still pretty good
        const int c = 3;
        float val2 = memA[g_id + c];

        // stride size is not so good
        float val3 = memA[c*g_id];

        const int loc =
          some_strange_func(g_id);

        // terrible!
        float val4 = memA[loc];
}
```

Memory access patterns



Memory access patterns

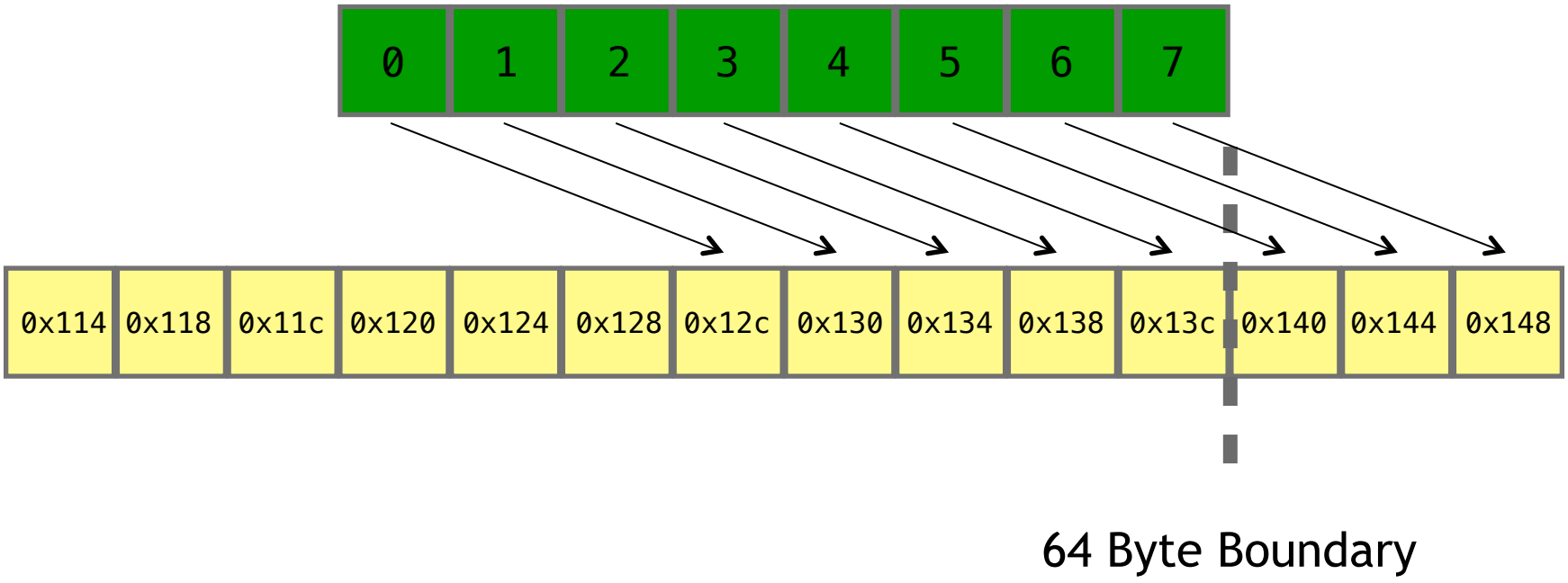
```
float val1 = memA[g_id];
```



64 Byte Boundary

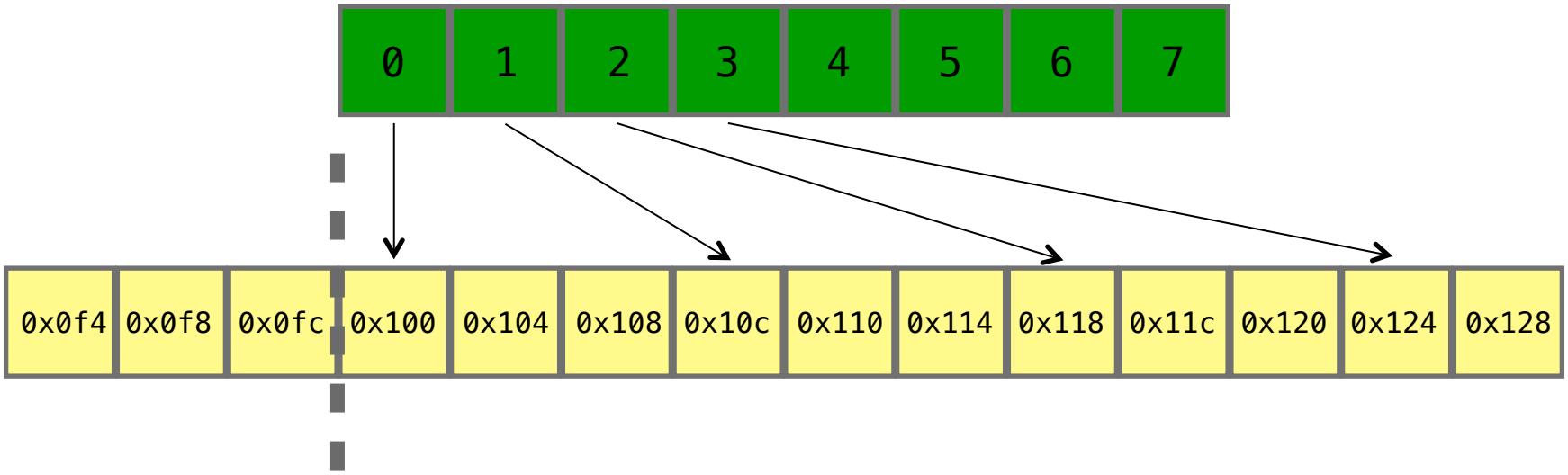
Memory access patterns

```
const int c = 3;
float val2 = memA[g_id + c];
```



Memory access patterns

```
float val3 = memA[3*g_id];
```

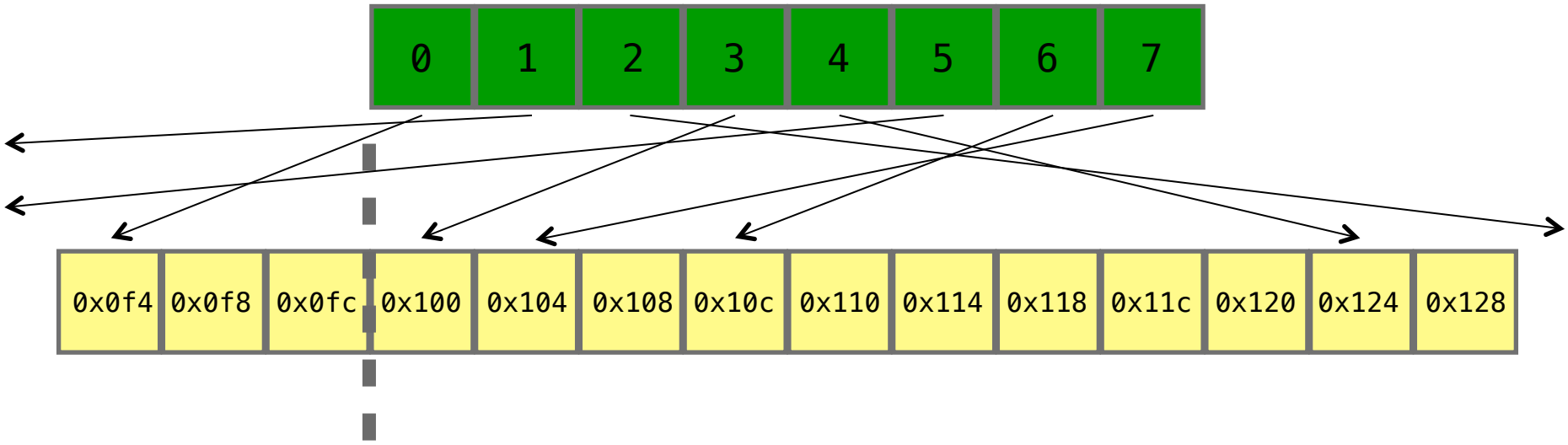


64 Byte Boundary

Strided access results in multiple memory transactions (and kills throughput)

Memory access patterns

```
const int loc =
  some_strange_func(g_id);

float val4 = memA[loc];
```



64 Byte Boundary

Thought exercise

- Consider the memory access patterns in your Jacobi solver kernel.
- There is a memory alignment problem…
- If you want to generate the transpose of the A matrix (a column major order), we provide a function inside mm_utils.c that you can call inside the host code to do this.

```
void init_colmaj_diag_dom_near_identity_matrix(int Ndim,  TYPE *A);
```

Jacobi solver kernel code (1/2)

```
#define TYPE double
#if (TYPE == double)
    #pragma OPENCL EXTENSION cl_khr_fp64 : enable
#endif

kernel void jacobi(
        const unsigned Ndim,
        global TYPE * A,  global TYPE * b,
        global TYPE * xold,  global TYPE * xnew)

{
    size_t i = get_global_id(0);

    xnew[i] = (TYPE) 0.0;
    for (int j = 0; j < Ndim; j++)  {
        if (i != j)
            xnew[i] += A[j*Ndim + i] * xold[j];
    }
    xnew[i] = (b[i] - xnew[i]) / A[i*Ndim + i];}
```

Swap indices on A to match column major layout – was A[i*Ndim+j]

Jacobi Solver Results

- Serial code running on the Intel® Xeon® CPU and icc took 83 seconds.
- With OpenMP for multithreading
  - 25.3 seconds with 32 threads (hyperthreading enabled)
  - 19.0 seconds with 16 threads (hyperthreading disabled)
- Running the OpenMP version natively on the Intel® Xeon® Phi Processor took 4.8 seconds.

Different versions of the Jacobi Solver with OpenCL. Runtimes in seconds				
TYPE = double NDIM = 4096	**Nvidia K40 GPU**	**AMD 290X GPU**	**Intel Xeon PHI processor**	**Intel Xeon processor**
Basic	35.0	198.2	245.2	23.6
Colmaj	14.1	15.3	35.8	71.5

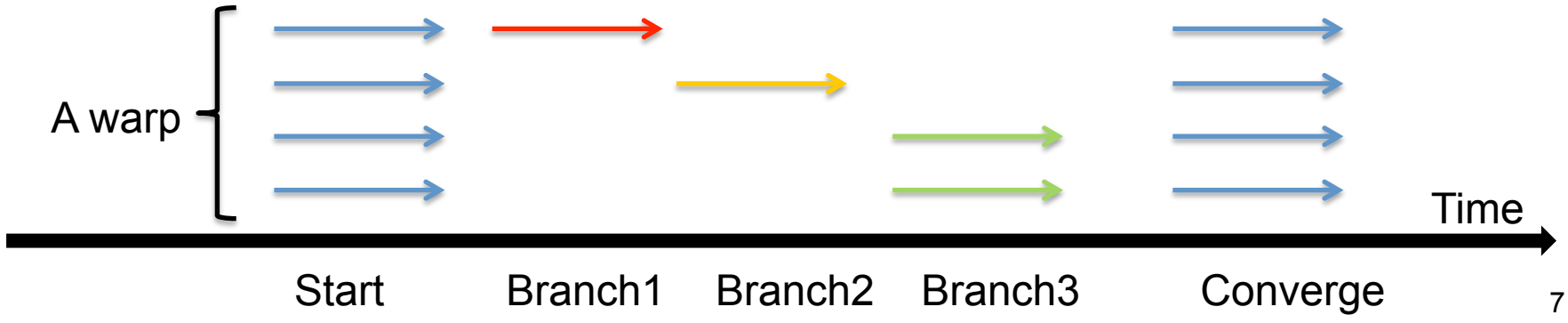
Dual-socket Intel® Xeon® CPU E5-2687W (16 cores total, hyper-threading enabled) and the Intel® icc compiler.

Agenda

- Logistics
- Introduction to Heterogeneous computing
- OpenCL
  - Overview
  - Host Programs
  - Kernel Programs
  - Kernel code optimization and compilation
    - Memory coalescence
    - Divergent control flows
    - Occupancy
    - Other Optimizations
    - Working with the OpenCL Memory Hierarchy
  - OpenCL ecosystem
- OpenMP

Single Instruction Multiple Data

- Individual threads of a warp start together at the same program address
- Each thread has its own instruction address counter and register state
  - Each thread is free to branch and execute independently
  - Provide the MIMD abstraction
- Branch behavior
  - Each branch will be executed serially
  - Threads not following the current branch will be disabled



Branching

- GPUs tend not to support speculative execution, which means that branch instructions have high latency
- This latency can be hidden by switching to alternative work-items/work-groups, but avoiding branches where possible is still a good idea to improve performance
- When different work-items executing within the same SIMD ALU array take different paths through conditional control flow, we have *divergent branches (vs. uniform branches)*
- These are even worse: work-items will stall while waiting for the others to complete
- We can use predication, selection and masking to convert conditional control flow into straight line code and significantly improve the performance of code that has lots of conditional branches

Branching

Conditional execution

```
// Only evaluate expression
// if condition is met
if (a > b)
{
  acc += (a - b*c);
}
```

Selection and masking

```
// Always evaluate expression
// and mask result
temp = (a - b*c);
mask = (a > b ? 1.f : 0.f);
acc += (mask * temp);
```

Exercise 3

- Eliminate the branch in your Jacobi solver kernel.
- Hint: comparisons return 1 for true and 0 for false…
- Measure the impact – how much faster does the code go?

Jacobi solver kernel code (1/2)

```
#define TYPE double
#if (TYPE == double)
    #pragma OPENCL EXTENSION cl_khr_fp64 : enable
#endif

kernel void jacobi(
        const unsigned Ndim,
        global TYPE * A,  global TYPE * b,
        global TYPE * xold,  global TYPE * xnew)
{
    size_t i = get_global_id(0);

    xnew[i] = (TYPE) 0.0;
    for (int j = 0; j < Ndim; j++)  {

        xnew[i] += A[j*Ndim + i] * xold[j] * (TYPE)(i != j);
    }
    xnew[i] = (b[i] - xnew[i]) / A[i*Ndim + i];
}
```

Jacobi Solver Results

- Serial code running on the Intel® Xeon® CPU and icc took 83 seconds.
- With OpenMP for multithreading
  - 25.3 seconds with 32 threads (hyperthreading enabled)
  - 19.0 seconds with 16 threads (hyperthreading disabled)
- Running the OpenMP version natively on the Intel® Xeon® Phi Processor took 4.8 seconds.

Different versions of the Jacobi Solver with OpenCL. Runtimes in seconds				
TYPE = double NDIM = 4096	**Nvidia K40 GPU**	**AMD 290X GPU**	**Intel Xeon PHI processor**	**Intel Xeon processor**
Basic	35.0	198.2	245.2	23.6
Colmaj	14.1	15.3	35.8	71.5
No Branch	13.3	15.6	16.6	38.8

Note: optimizations in the table are cumulative

Dual-socket Intel® Xeon® CPU E5-2687W (16 cores total, hyper-threading enabled) and the Intel® icc compiler.

Agenda

- Logistics
- Introduction to Heterogeneous computing
- OpenCL
  - Overview
  - Host Programs
  - Kernel Programs
  - Kernel code optimization and compilation
    - Memory coalescence
    - Divergent control flows
    - Occupancy
    - Other Optimizations
    - Working with the OpenCL Memory Hierarchy
  - OpenCL ecosystem
- OpenMP

Keep the processing elements (PE) busy

- **Occupancy**: a measure of the fraction of time during a computation when the PE's are busy. Goal is to keep this number high (well over 50%).
- Pay attention to the number of work-items and work-group sizes
  - Rule of thumb: On a modern GPU you want at least 4 work-items per PE in a Compute Unit
  - More work-items are better, but diminishing returns, and there is an upper limit
    - Each work item consumes PE finite resources (registers etc)

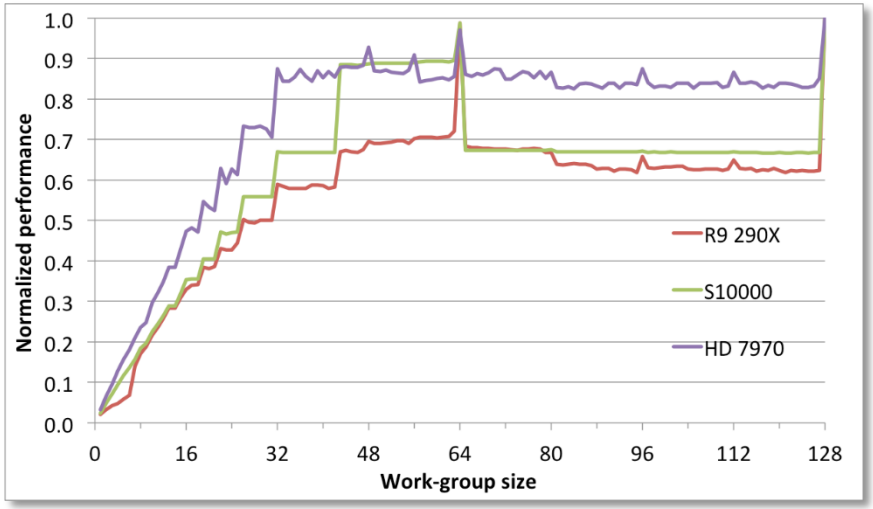
Occupancy

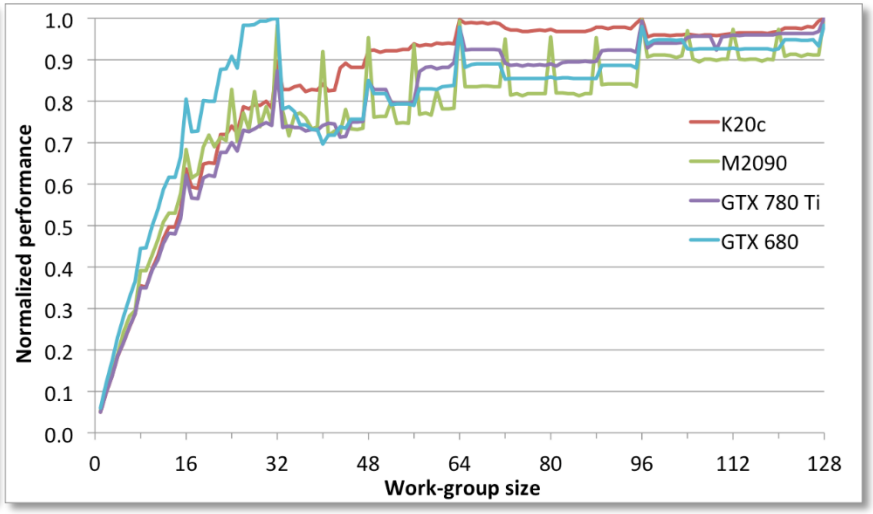
- Number of work-groups per compute unit (CU) depends on registers and local memory size per work-group
- E.g. NVIDIA's K40 has 128 words of memory per processor element (PE), i.e. 128 registers per core; and 48KB of local memory per CU
- But, multiple work-items (threads) will be scheduled on a single PE (similar to hyperthreading)
- In fact, global memory latency is so high that multiple work-items per PE are a *requirement* for achieving a good proportion of peak performance!

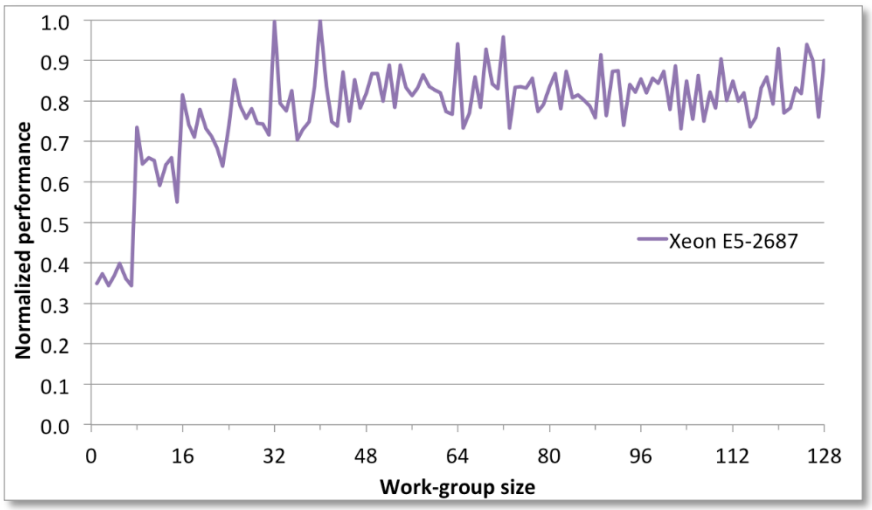
Work-group sizes

- Work-group sizes being a power of 2 helps on most architectures. At a minimum use multiples of:
  - 8 for Intel® AVX CPUs
  - 16 for Intel® Xeon Phi™ processors
  - 32 for Nvidia® GPUs
  - 64 for AMD®
  - May be different on different hardware
- On most systems aim to run lots of work-groups. For example, on Xeon Phi, multiples of the number of threads available (e.g. 240 on a 5110P) is optimal, but as many as possible is good (1000+)

Effect of work-group sizes







Optional Exercise

If time allows:

- Experiment with different work group sizes (can make a ***big*** difference!)
- Run the host program with the flag –h to see the command line options. One of them (--wg) will vary the workgroup size.

Jacobi Solver Results

- Serial code running on the Intel® Xeon® CPU and icc took 83 seconds.
- With OpenMP for multithreading
  - 25.3 seconds with 32 threads (hyperthreading enabled)
  - 19.0 seconds with 16 threads (hyperthreading disabled)
- Running the OpenMP version natively on the Intel® Xeon® Phi Processor took 4.8 seconds.

Different versions of the Jacobi Solver with OpenCL. Runtimes in seconds				
TYPE = double NDIM = 4096	**Nvidia K40 GPU**	**AMD 290X GPU**	**Intel Xeon PHI processor**	**Intel Xeon processor**
Basic	35.0	198.2	245.2	23.6
Colmaj	14.1	15.3	35.8	71.5
No Branch	13.3	15.6	16.6	38.8
Opt WG size	13.2	15.1	15.0	32.1

Note: optimizations in the table are cumulative

Dual-socket Intel® Xeon® CPU E5-2687W (16 cores total, hyper-threading enabled) and the Intel® icc compiler.

Agenda

- Logistics
- Introduction to Heterogeneous computing
- OpenCL
  - Overview
  - Host Programs
  - Kernel Programs
  - Kernel code optimization and compilation
    - Memory coalescence
    - Divergent control flows
    - Occupancy
    - Other Optimizations
    - Working with the OpenCL Memory Hierarchy
  - OpenCL ecosystem
- OpenMP

Constant Memory

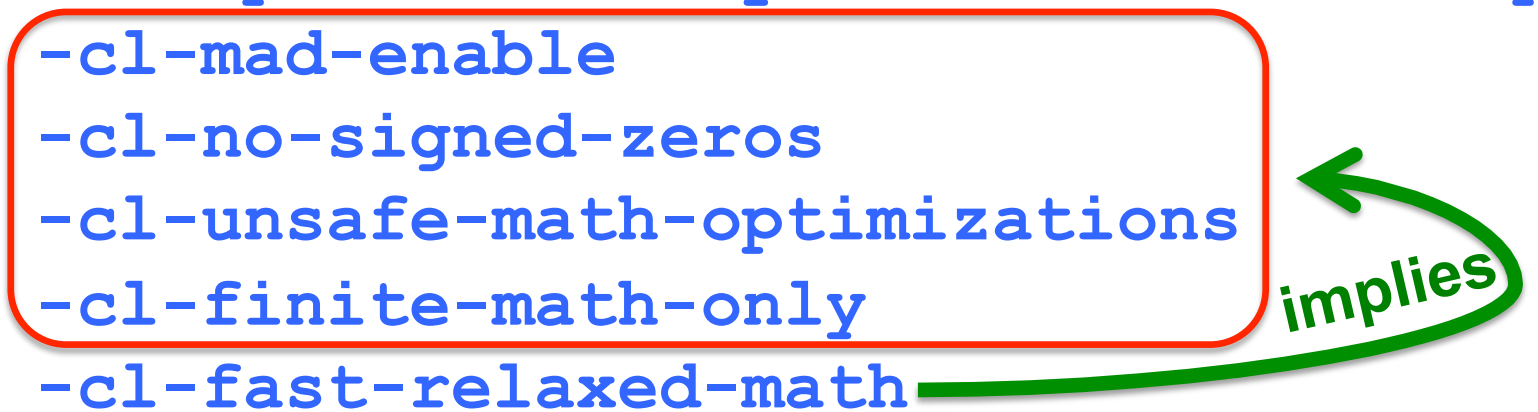
- Constant memory can be considered a store for data that never changes
- Setting and updating constants in memory uses the same interface as global memory, with enqueueRead/ enqueueWrite commands
- The difference is how it is declared in the kernel
- Some devices may have dedicated on-chip caches or data-paths for constant memory
- Devices are guaranteed to support constant memory allocations of at least 64kB
- Can also declare OpenCL program scope constant data, but this has to be initialized at OpenCL program compile time

```
kernel void
calc_something
(
  global float *a,
  global float *b,
  global float *c,

  //constant memory is
  //set on the host
  constant float *params
)
{
  //code here
}
```

Compiler Options

- OpenCL compilers accept a number of flags that affect how kernels are compiled:

```
-cl-opt-disable
-cl-single-precision-constant
-cl-denorms-are-zero
-cl-fp32-correctly-rounded-divide-sqrt
-cl-mad-enable
-cl-no-signed-zeros
-cl-unsafe-math-optimizations
-cl-finite-math-only
-cl-fast-relaxed-math
```



Other compilation hints

- Can use an attribute to inform the compiler of the work-group size that you intend to launch kernels with:

```
__attribute__((reqd_work_group_size(x, y, z)))
```

- As with C/C++, use the **`const`**/**`restrict`** keywords for kernel arguments where appropriate to make sure the compiler can optimise memory accesses

Optional Exercise

- Experiment with different optimizations to get the best runtime you can.

Jacobi Solver Results

- Serial code running on the Intel® Xeon® CPU and icc took 83 seconds.
- With OpenMP for multithreading
  - 25.3 seconds with 32 threads (hyperthreading enabled)
  - 19.0 seconds with 16 threads (hyperthreading disabled)
- Running the OpenMP version natively on the Intel® Xeon® Phi Processor took 4.8 seconds.

Different versions of the Jacobi Solver with OpenCL. Runtimes in seconds				
TYPE = double NDIM = 4096	**Nvidia K40 GPU**	**AMD 290X GPU**	**Intel Xeon PHI processor**	**Intel Xeon processor**
Basic	35.0	198.2	245.2	23.6
Colmaj	14.1	15.3	35.8	71.5
No Branch	13.3	15.6	16.6	38.8
Opt WG size	13.2	15.1	15.0	32.1
Unroll by 4	6.2	6.7	13.3	32.1

Note: optimizations in the table are cumulative

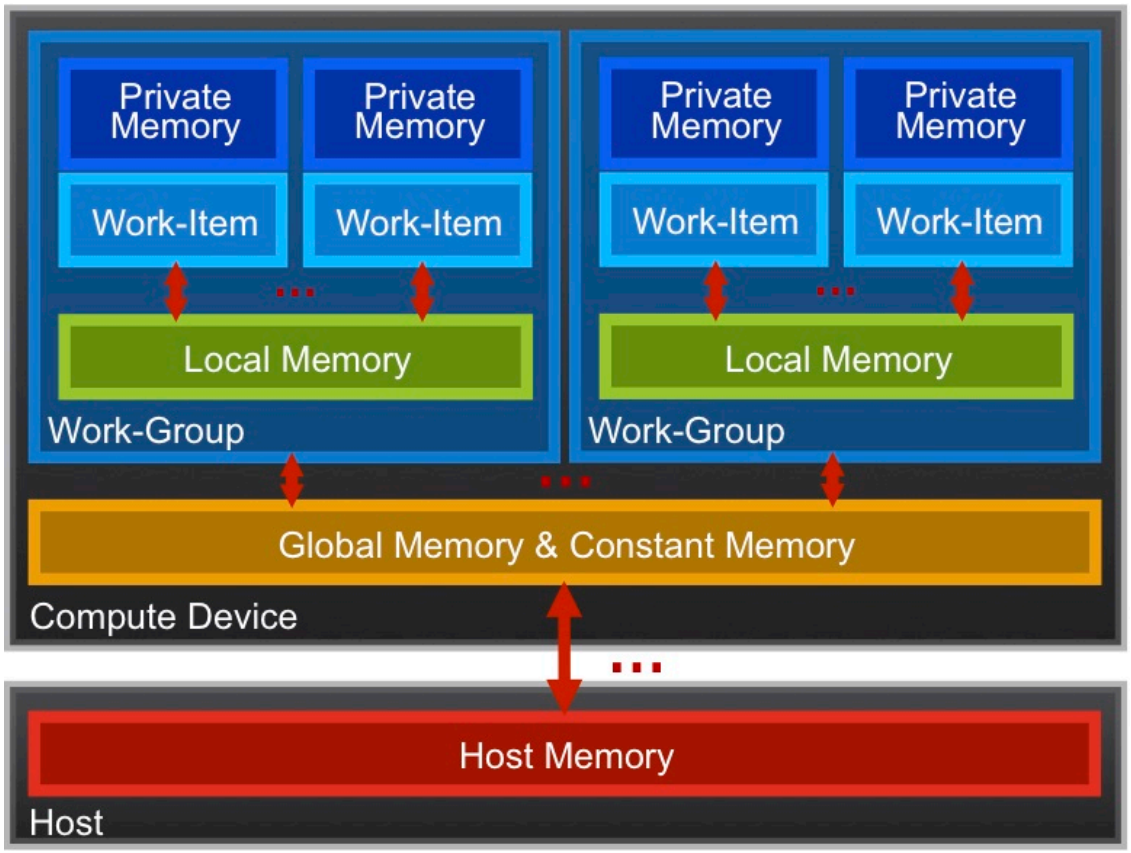
Dual-socket Intel® Xeon® CPU E5-2687W (16 cores total, hyper-threading enabled) and the Intel® icc compiler.

Agenda

- Logistics
- Introduction to Heterogeneous computing
- OpenCL
  - Overview
  - Host Programs
  - Kernel Programs
  - Kernel code optimization and compilation
    - Memory coalescence
    - Divergent control flows
    - Occupancy
    - Other Optimizations
    - Working with the OpenCL Memory Hierarchy
  - OpenCL ecosystem
- OpenMP

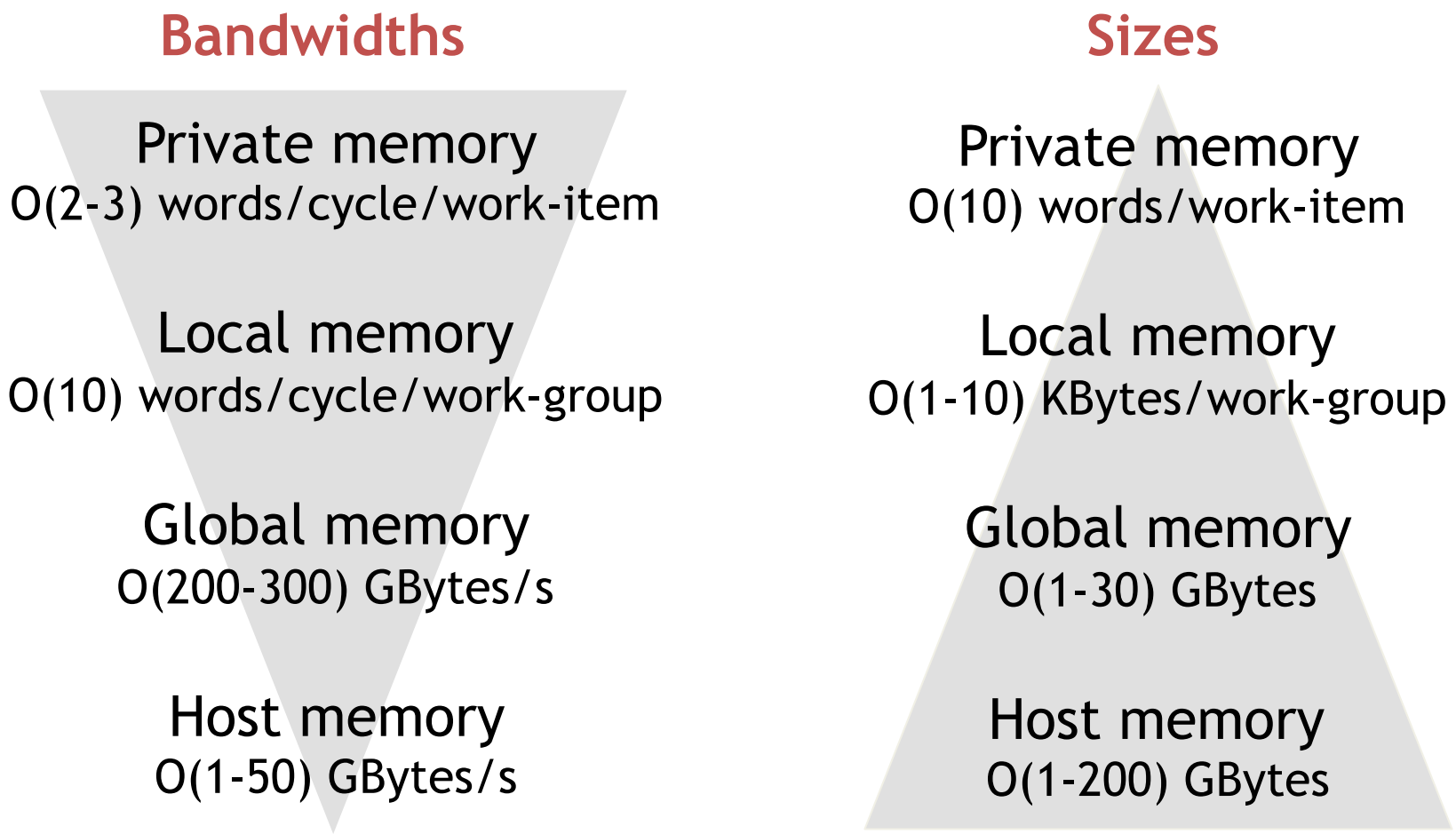
OpenCL Memory model

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



Memory management is **explicit**:
You are responsible for moving data from
host → global → local *and* back

The Memory Hierarchy

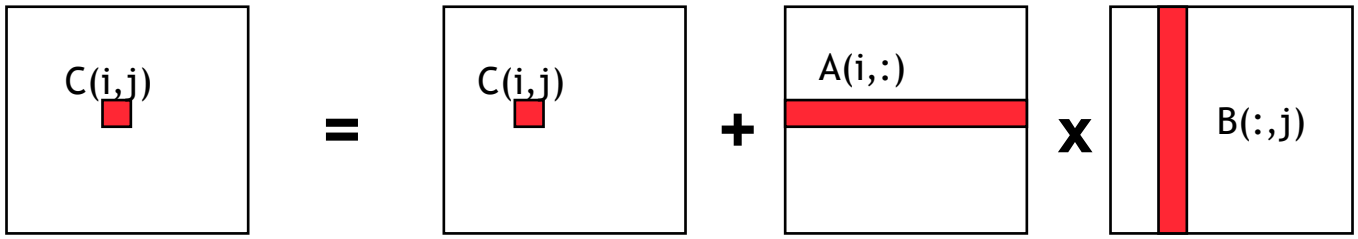


Managing the memory hierarchy is one of ***the*** most important things to get right to achieve good performance

*Size and performance numbers are approximate and for a high-end discrete GPU, circa 2015

Optimizing matrix multiplication

- MM cost determined by FLOPS and memory movement:
  - $2*n^3 = O(n^3)$ FLOPS
  - Operates on $3*n^2 = O(n^2)$ numbers
- To optimize matrix multiplication, we must ensure that for every memory access we execute as many FLOPS as possible.
- Outer product algorithms are faster, but for pedagogical reasons, let's stick to the simple dot-product algorithm.

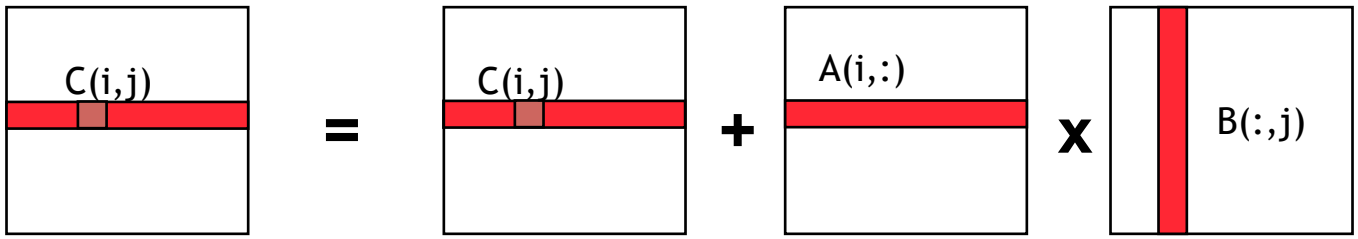


Dot product of a row of A and a column of B for each element of C

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

Optimizing matrix multiplication

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

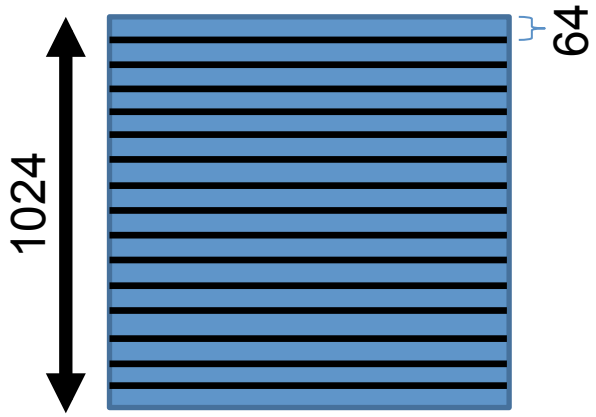


Dot product of a row of A and a column of B for each element of C

- And with an eye towards future optimizations, let's collect work-items into work-groups with 64 work-items per work-group

An N-dimension domain of work-items

- Global Dimensions: 1024 (1D)
  Whole problem space (index space)
- Local Dimensions: 64 (work-items per work-group)
  Only 1024/64 = 16 work-groups in total



- Important implication: we will have a lot fewer work-items per work-group (64) and work-groups (16). Why might this matter?

Matrix multiplication: One work item per row of C

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

Mat. Mul. host program (1 row per work-item)

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT
int main(void)
{  // declarations (not shown)
  sz = N * N;
  std::vector<float> h_A(sz);
  std::vector<float> h_B(sz);
  std::vector<float> h_C(sz);

 cl::Buffer d_A, d_B, d_C;

// initialize matrices  and setup
// the problem (not shown)

 cl::Context context(DEVICE);
 cl::Program program(context,
    util::loadProgram("mmulCrow.cl“,
        true));
cl::CommandQueue queue(context);

auto mmul = cl::make_kernel
        <int, cl::Buffer, cl::Buffer, cl::Buffer>
                    (program, "mmul");

  d_A   = cl::Buffer(context, begin(h_A),
                                  end(h_A), true);
  d_B   = cl::Buffer(context, begin(h_B),
                                  end(h_B), true);
  d_C   = cl::Buffer(context,
                    CL_MEM_WRITE_ONLY,
                     sizeof(float) * sz);

  mmul(cl::EnqueueArgs( queue,
                     cl::NDRange(N),
                     cl::NdRange(64)),
                     N, d_A,  d_B,  d_C);

  cl::copy(queue, d_C, begin(h_C),
                                  end(h_C));

   // Timing and check results (not shown)
}
```

Mat. Mul. host program (1 row per work-item)

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT
int main(void)
{  // declarations (not shown)
  sz = N * N;
  std::vector<float> h_A(sz);
  std::vector<float> h_B(sz);
  std::vector<float> h_C(sz);
```

Changes to host program:

1. 1D ND Range set to number of rows in the C matrix
2. Local Dimension set to 64 (which gives us 16 work-groups which matches the GPU's number of compute units).

```
    true));
```

```
cl::CommandQueue queue(context);

auto mmul = cl::make_kernel
        <int, cl::Buffer, cl::Buffer, cl::Buffer>
                    (program, "mmul");

  d_A   = cl::Buffer(context, begin(h_A),
                                  end(h_A), true);
  d_B   = cl::Buffer(context, begin(h_B),
                                  end(h_B), true);
  d_C   = cl::Buffer(context,
                    CL_MEM_WRITE_ONLY,
                     sizeof(float) * sz);

  mmul(cl::EnqueueArgs( queue,
                    cl::NDRange(N),
                    cl::NdRange(64)),
                    N, d_A,  d_B,  d_C);

  cl::copy(queue, d_C, begin(h_C),
                                  end(h_C));

   // Timing and check results (not shown)
}
```

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8

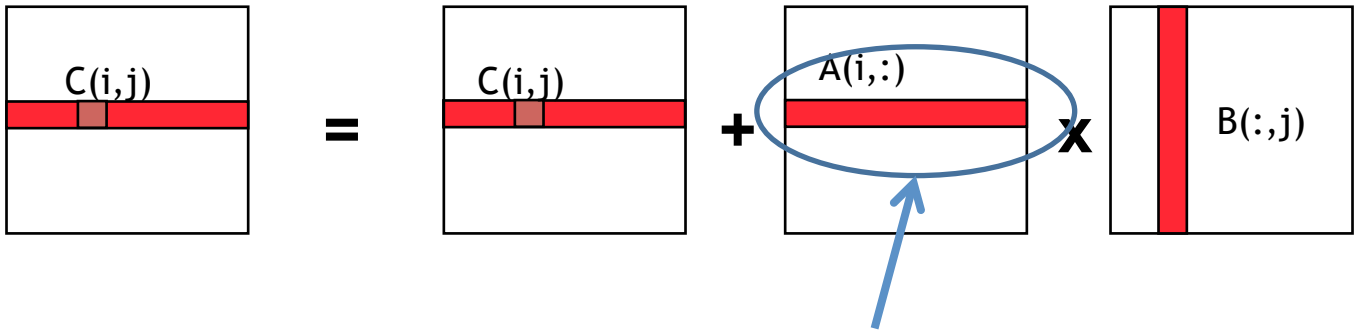
This has started to help.

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

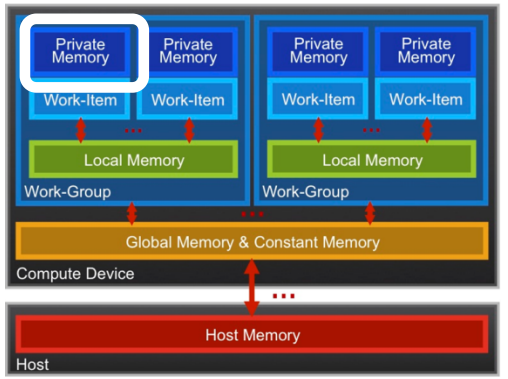
These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Optimizing matrix multiplication

- Notice that, in one row of C, each element reuses the same row of A.
- Let's copy that row of A into private memory of the work-item that's (exclusively) using it to avoid the overhead of loading it from global memory for each C(i,j) computation.



Private Memory



- A work-items private memory:
  - A **very scarce** resource, only a few tens of 32-bit words per Work-Item at most (on a GPU)
  - If you use too much it spills to global memory or reduces the number of Work-Items that can be run at the same time, potentially harming performance*
  - Think of these like registers on the CPU
- How do you create and manage private memory?
  - Declare statically inside your kernel

* Occupancy on a GPU

Matrix multiplication: (Row of A in private memory)

```
__kernel void mmul(
    const int N,
    __global float *A,
    __global float *B,
    __global float *C)
{
  int j, k;
  int i = get_global_id(0);
  float tmp;
  float Awrk[1024];

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

  for (j = 0; j < N; j++) {
    tmp = 0.0f;
    for (k = 0; k < N; k++)
      tmp += Awrk[k]*B[k*N+j];

    C[i*N+j] = tmp;
  }
}
```

Matrix multiplication: (Row of A in private memory)

```
__kernel void mmul(
    const int N,
    __global float *A,
    __global float *B,
    __global float *C)
{
  int j, k;
  int i = get_global_id(0);
  float tmp;
  float Awrk[1024];

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

  for (j = 0; j <
    tmp = 0.0f;
    for (k = 0; k
      tmp += Awrk[

    C[i*N+j] = tmp;
  }
}
```

Setup a work array for A in private memory*

Copy a row of A into private memory from global memory before we start with the matrix multiplications.

(*Actually, this is using ***far*** more private memory than we'll have and so Awrk[] will be spilled to global memory)

Mat. Mul. host program (Row of A in private memory)

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT
int main(void)
{  // declarations (not shown)
  sz = N * N;
  std::vector<float> h_A(sz);
  std::vector<float> h_B(sz);
  std::vector<float> h_C(sz);

 cl::Buffer d_A, d_B, d_C;

// initialize matrices  and setup
// the problem (not shown)

 cl::Context context(DEVICE);
 cl::Program program(context,
    util::loadProgram("mmulCrow.cl“,
        true));
```

```
cl::CommandQueue queue(context);

auto mmul = cl::make_kernel
        <int, cl::Buffer, cl::Buffer, cl::Buffer>
                    (program, "mmul");

  d_A   = cl::Buffer(context, begin(h_A),
                                  end(h_A), true);
  d_B   = cl::Buffer(context, begin(h_B),
                                  end(h_B), true);
  d_C   = cl::Buffer(context,
                    CL_MEM_WRITE_ONLY,
                     sizeof(float) * sz);

  mmul(cl::EnqueueArgs( queue,
                     cl::NDRange(N),
                     cl::NDRange(64)),
                     N, d_A,  d_B,  d_C);

  cl::copy(queue, d_C, begin(h_C),
                                  end(h_C));

  // Timing and check results (not shown)
}
```

Host program unchanged from last exercise

Matrix multiplication performance

- Matrices are stored in global memory.

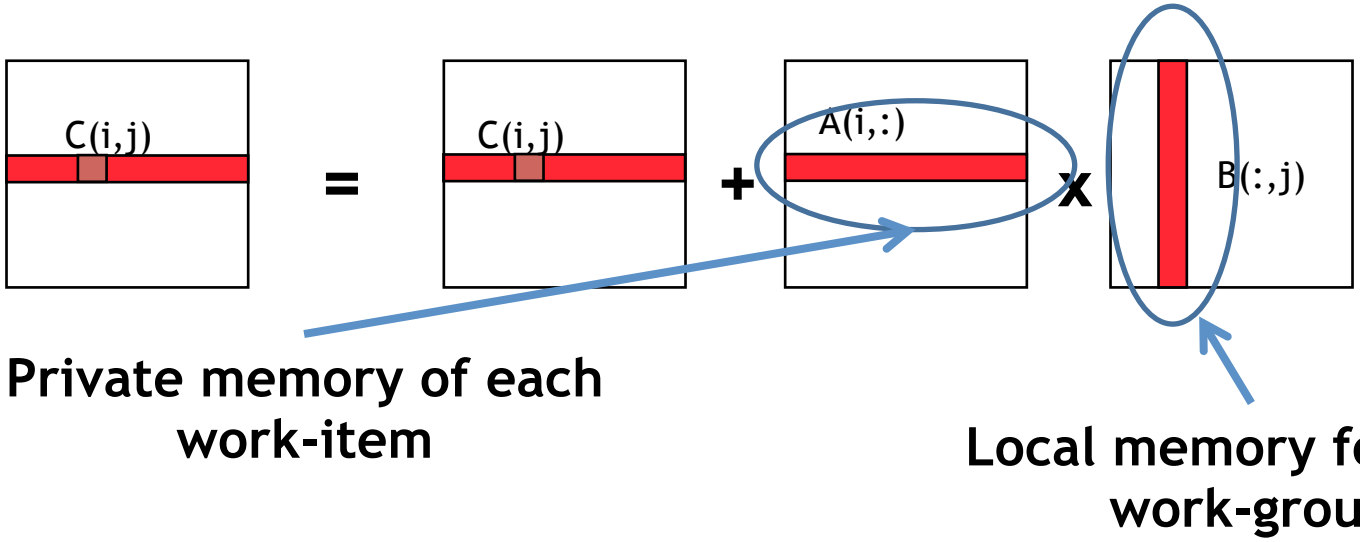
Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8
C row per work-item, A row private	3,385.8	8,584.3

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

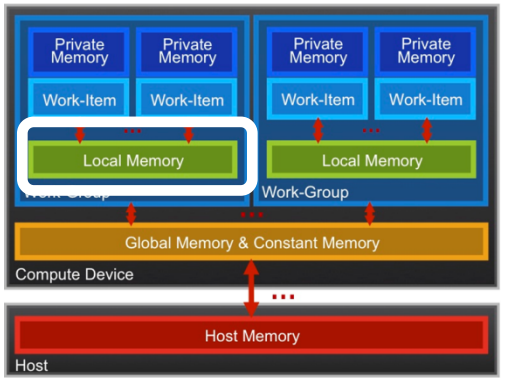
These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Optimizing matrix multiplication

- We already noticed that, in one row of C, each element uses the same row of A
- Each work-item in a work-group also uses the same columns of B
- So let's store the B columns in local memory (which is shared by the work-items in the work-group)



Local Memory

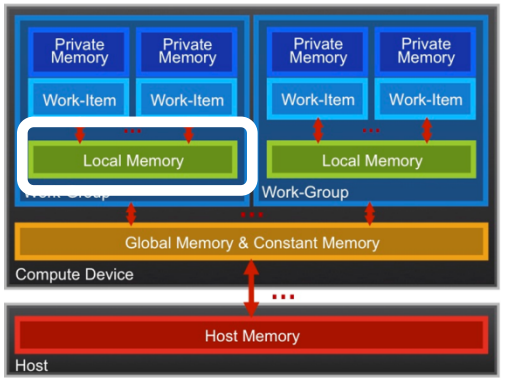


- A work-group's shared memory
  - Typically 10's of KBytes per Compute Unit*
  - Use Local Memory to hold data that can be **reused by all the work-items** in a work-group
  - As multiple Work-Groups may be running on each Compute Unit (CU), only a fraction of the total Local Memory size may be available to each Work-Group

- How do you create and manage local memory?
  - Create local memory kernel argument on the host
    - `err |= clSetKernelArg(mmul, 3, sizeof(float)*N, NULL);`
    - This command makes the 4th argument to kernel mmul a pointer to a newly allocated local memory buffer of size 4N bytes
  - Mark kernel arguments that are from local memory as __local
  - Your kernels are responsible for transferring data between Local and Global/Constant memories … there are built-in functions to help (async_work_group_copy(), async_workgroup_strided_copy(), etc)

*Size and performance numbers are approximate and for a high-end discrete GPU, circa 2015

Local Memory performance hints



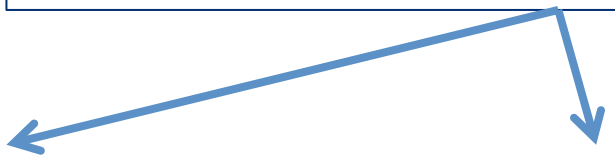
- Local Memory doesn't always help…
  - CPUs don't have special hardware for it
  - This can mean excessive use of Local Memory might slow down kernels on CPUs
  - GPUs now have effective on-chip caches which can provide much of the benefit of Local Memory but without programmer intervention
  - Access patterns to Local Memory affect performance in a similar way to accessing Global Memory
    - Have to think about things like coalescence & bank conflicts
  - So, your mileage may vary!

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 the work-item's private view of memory, i.e. it sees its own reads and writes correctly
- 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!!**
  - This is a common source of bugs!
- Consistency of memory shared between commands (e.g. kernel invocations) is enforced by synchronization (barriers, events, in-order queue)

Work-Item Synchronization

Ensure correct order of memory operations to local or global memory (with flushes or queuing a memory fence)



- 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 its 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! (Critical issue!)**

  – **Only solution: finish the kernel and start another**

Matrix multiplication: B column shared between work-items

```
__kernel void mmul(
        const int N,
    __global float *A,
    __global float *B,
    __global float *C,
    __local  float *Bwrk)
{
 int j, k;
 int i = get_global_id(0);

 int iloc=get_local_id(0);
 int nloc=get_local_size(0);

 float tmp;
 float Awrk[1024];

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

 for (j = 0; j < N; j++) {

   for (k=iloc; k< N; k+=nloc)
     Bwrk[k] = B[k* N+j];
   barrier(CLK_LOCAL_MEM_FENCE);

   tmp = 0.0f;
   for (k = 0; k < N; k++)
     tmp += Awrk[k]*Bwrk[k];

   C[i*N+j] = tmp;
   barrier(CLK_LOCAL_MEM_FENCE);
 }
}
```

Matrix multiplication: B column shared between work-items

```
__kernel void mmul(
        const int N,
    __global float *A,
    __global float *B,
    __global float *C,
    __local  float *Bwrk)
{
 int j, k;
 int i = get_global_id(0);

 int iloc=get_local_id(0);
 int nloc=get_local_size(0);

 float tmp;
 float Awrk[1024];

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

 for (j = 0; j < N; j++) {

   for (k=iloc; k< N; k+=nloc)
     Bwrk[k] = B[k* N+j];
   barrier(CLK_LOCAL_MEM_FENCE);

   tmp = 0.0f;
   for (k = 0; k < N; k++)
     tmp += Awrk[k]*Bwrk[k];

   C[i*N+j] = tmp;
   barrier(CLK_LOCAL_MEM_FENCE);
 }
}
```

Pass a work array in local memory to hold a column of B. All the work-items do the copy "in parallel" using a cyclic loop distribution (hence why we need iloc and nloc)

Mat. Mul. host program (Share a column of B within a work-group)

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT
int main(void)
{  // declarations (not shown)
  sz = N * N;
  std::vector<float> h_A(sz);
  std::vector<float> h_B(sz);
  std::vector<float> h_C(sz);

 cl::Buffer d_A, d_B, d_C;

// initialize matrices  and setup
// the problem (not shown)

 cl::Context context(DEVICE);
 cl::Program program(context,
    util::loadProgram("mmulCrow.cl“,
        true));
```

```
cl::CommandQueue queue(context);

auto mmul = cl::make_kernel
       <int, cl::Buffer, cl::Buffer, cl::Buffer,
            cl::LocalSpaceArg > (program, "mmul");

  d_A   = cl::Buffer(context, begin(h_A), end(h_A),true);
  d_B   = cl::Buffer(context, begin(h_B), end(h_B),true);
  d_C   = cl::Buffer(context,
            CL_MEM_WRITE_ONLY, sizeof(float) * sz);

  cl::LocalSpaceArg Bwrk =
                  cl::Local(sizeof(float) * Pdim);

  mmul(cl::EnqueueArgs( queue,
                cl::NDRange(N),  cl::NDRange(64)),
                 N, d_A,  d_B,  d_C, Bwrk);

  cl::copy(queue, d_C, begin(h_C), end(h_C));

   // Timing and check results (not shown)
}
```

Mat. Mul. host program (Share a column of B within a work-group)

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT
int main(void)
{  // declarations (not shown)
  sz = N * N;
  std::vector<float> h_A(sz);
c
/
/
  cl::Context context(DEVICE);
  cl::Program program(context,
     util::loadProgram("mmulCrow.cl“,
        true));
```

Change host program to pass local memory to kernels.

- Add an arg of type LocalSpaceArg is needed.
- Allocate the size of local memory
- Update argument list in kernel functor

```
cl::CommandQueue queue(context);

auto mmul = cl::make_kernel
      <int, cl::Buffer, cl::Buffer, cl::Buffer,
           cl::LocalSpaceArg > (program, "mmul");

  d_A   = cl::Buffer(context, begin(h_A), end(h_A),true);
  d_B   = cl::Buffer(context, begin(h_B), end(h_B),true);
  d_C   = cl::Buffer(context,
            CL_MEM_WRITE_ONLY, sizeof(float) * sz);

  cl::LocalSpaceArg Bwrk =
                  cl::Local(sizeof(float) * Pdim);

  mmul(cl::EnqueueArgs( queue,
               cl::NDRange(N),  cl::NDRange(64)),
                N, d_A,  d_B,  d_C, Bwrk);

  cl::copy(queue, d_C, begin(h_C), end(h_C));

   // Timing and check results (not shown)
}
```

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8
C row per work-item, A row private	3,385.8	8,584.3
C row per work-item, A private, B local	10,047.5	8,181.9

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

The CuBLAS SGEMM provides an effective measure of peak achievable performance on the GPU. CuBLAS performance = 283366.4 MFLOPS

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
        for (k = 0; k < N; k++) {
          C[i*N+j] += A[i*N+k] * B[k*N+j];
        }
      }
    }
}
```

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
          C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Let's get rid of all those ugly brackets

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (i = ib*NB; i < (ib+1)*NB; i++)
     for (jb = 0; jb < NB; jb++)
       for (j = jb*NB; j < (jb+1)*NB; j++)
         for (kb = 0; kb < NB; kb++)
           for (k = kb*NB; k < (kb+1)*NB; k++)
             C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Break each loop into chunks with a size chosen to match the size of your fast memory

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Rearrange loop nest to move loops over blocks "out" and leave loops over a single block together

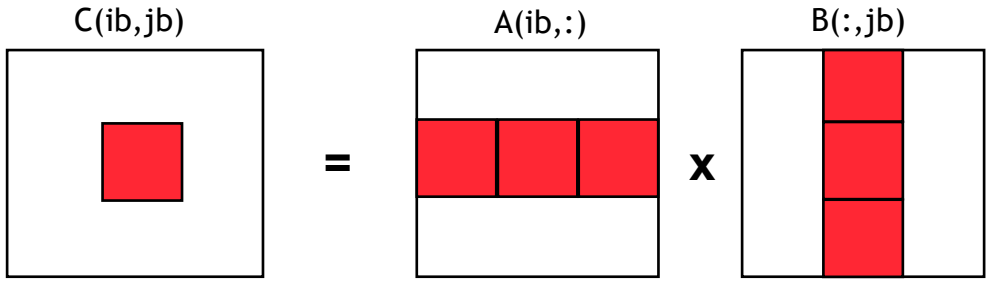
Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

This is just a local matrix multiplication of a single block

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)
       sgemm(C, A, B, …)    // C_ib,jb = A_ib,kb * B_kb,jb
}
```



Note: sgemm is the name of the level three BLAS routine to multiply two matrices

Blocked matrix multiply: kernel

```
#define blksz 16
__kernel void mmul(
              const unsigned int N,
              __global float* A,
              __global float* B,
              __global float* C,
              __local  float* Awrk,
              __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  //  compute element C(i,j)
  int i = get_global_id(0);
  int j = get_global_id(1);

  // Element C(i,j) is in block C(Iblk,Jblk)
  int Iblk = get_group_id(0);
  int Jblk = get_group_id(1);

  // C(i,j) is element C(iloc, jloc)
  //  of block C(Iblk, Jblk)
  int iloc = get_local_id(0);
  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;

  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;   int Ainc  = blksz;
  int Bbase = Jblk*blksz;     int Binc  = blksz*N;

 // C(Iblk,Jblk) = (sum over Kblk)
A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {   //Load A(Iblk,Kblk) and B(Kblk,Jblk).
      //Each work-item loads a single element of the two
      //blocks which are shared with the entire work-group

      Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
      Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];


      barrier(CLK_LOCAL_MEM_FENCE);

      #pragma unroll
      for(kloc=0; kloc<blksz; kloc++)
  Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];


      barrier(CLK_LOCAL_MEM_FENCE);

      Abase += Ainc;    Bbase += Binc;
   }
   C[j*N+i] = Ctmp;
}
```

Blocked matrix multiply: kernel

```
#define blksz 16
__kernel void mmul(
              const unsigned int N,
              __global float* A,
              __global float* B,
              __global float* C,
              __local  float* Awrk,
              __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  //  compute element C(i,j)
  int i = get_global_id(0);
  int j = get_global_id(1);

  // Element C(i,j) is in block C(Iblk,Jblk)
  int Iblk = get_group_id(0);
  int Jblk = get_group_id(1);

  // C(i,j) is element C(iloc, jloc)
  //  of block C(Iblk, Jblk)
  int iloc = get_local_id(0);
  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;

  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;   int Ainc  = blksz;
  int Bbase = Jblk*blksz;     int Binc  = blksz*N;

 // C(Iblk,Jblk) = (sum over Kblk)
A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {   //Load A(Iblk,Kblk) and B(Kblk,Jblk).
     //Each work-item loads a single element of the two
     //blocks which are shared with the entire work-group
     Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
     Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

     barrier(CLK_LOCAL_MEM_FENCE);

     #pragma unroll
     for(kloc=0; kloc<blksz; kloc++)
  Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

     barrier(CLK_LOCAL_MEM_FENCE);
     Abase += Ainc;    Bbase += Binc;
  }
  C[j*N+i] = Ctmp;
}
```

Load A and B blocks, wait for all work-items to finish

Wait for everyone to finish before going to next iteration of Kblk loop.

Matrix multiplication … Portable Performance

- Single Precision matrix multiplication (order 1000 matrices)

	CPU	Xeon Phi	Core i7, HD Graphics	NVIDIA Tesla
Sequential C (compiled /O3)	224.4		1221.5	
C(i,j) per work-item, all global	841.5	13591		3721
C row per work-item, all global	869.1	4418		4196
C row per work-item, A row private	1038.4	24403		8584
C row per work-item, A private, B local	3984.2	5041		8182
Block oriented approach using local (blksz=16)	12271.3	74051 (126322*)	38348 (53687*)	119305
Block oriented approach using local (blksz=32)	16268.8			

Xeon Phi SE10P, CL_CONFIG_MIC_DEVICE_2MB_POOL_INIT_SIZE_MB = 4 MB
* The comp was run twice and only the second time is reported (hides cost of memory movement.

Intel® Core™ i5-2520M CPU @2.5 GHz (dual core) Windows 7 64 bit OS, Intel compiler 64 bit version 13.1.1.171, OpenCL SDK 2013, MKL 11.0 update 3.

Intel Core i7-4850HQ @ 2.3 GHz which has an Intel HD Graphics 5200 w/ high speed memory. ICC 2013 sp1 update 2.
Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8
C row per work-item, A row private	3,385.8	8,584.3
C row per work-item, A private, B local	10,047.5	8,181.9

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Matrix multiplication performance (CPU)

- Matrices are stored in global memory.

Case	MFLOPS CPU
Sequential C (not OpenCL, compiled /O3)	224.4
C(i,j) per work-item, all global	841.5
C row per work-item, all global	869.1
C row per work-item, A row private	1038.4
C row per work-item, A private, B local	3984.2
Block oriented approach using local (blksz=8)	7482.5
Block oriented approach using local (blksz=16)	12271.3
Block oriented approach using local (blksz=32)	16268.8
Intel MKL SGEMM	63780.6

Device is Intel® Core™ i5-2520M CPU @2.5 GHz (dual core) Windows 7 64 bit OS, Intel compiler 64 bit version 13.1.1.171, OpenCL SDK 2013, MKL 11.0 update 3.

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Agenda

- Logistics
- Introduction to Heterogeneous computing
- OpenCL
  - Overview
  - Host Programs
  - Kernel Programs
  - Kernel code optimization
  - OpenCL ecosystem
- OpenMP
  - Overview
  - “Target” and “Target data” directives
  - Mapping onto GPUs: the distribute directive

OpenCL 2.0

- OpenCL 2.0 was ratified in Nov'13
- Brings several important new features:
  - Shared Virtual Memory
  - Nested parallelism
  - Built-in work-group reductions
  - Generic address space
  - Pipes
  - C11 atomics
- Specification and headers available here
- Production drivers now available from Intel and AMD, with more expected to follow

- Standard Portable Intermediate Representation
- Defines an IR for OpenCL programs
- Means that developers can ship portable binaries instead of their OpenCL source
- Also intended to be a target for other languages/ programming models (C++ AMP, SYCL, OpenACC, DSLs)
- SPIR 1.2 & SPIR 2.0 ratified, SPIR-V provisional available now
- Implementations available from Intel and AMD, with more on the way

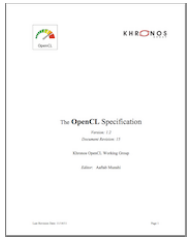
- Single source C++ abstraction layer for OpenCL
- Goal is to enable the creation of C++ libraries and frameworks that utilize OpenCL
- Can utilize SPIR to target OpenCL platform
- Supports ‘host-fallback’ (CPU) when no OpenCL devices available
- Provisional specification released Mar’14
- Codeplay and AMD working on implementations

Libraries

- clFFT / clBLAS / clRNG (all on github)
- Arrayfire (open source soon)
- Boost compute with VexCL
- ViennaCL (PETSc), PARALUTION
- Lots more - see the Khronos OpenCL pages:

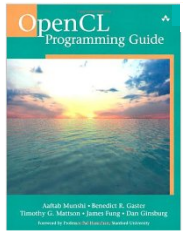
https://www.khronos.org/opencl/resources

Resources: https://www.khronos.org/opencl/

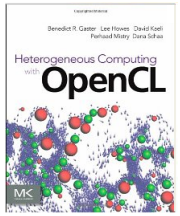


The OpenCL specification
Surprisingly approachable for a spec!
https://www.khronos.org/registry/cl/

OpenCL reference card
Useful to have on your desk(top)
Available on the same page as the spec.



OpenCL Programming Guide:
Aaftab Munshi, Benedict Gaster, Timothy G. Mattson and James Fung, 2011



Heterogeneous Computing with OpenCL
Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry and Dana Schaa, 2011

OpenCL Tutorials

http://handsonopencl.github.io

- One of the most popular OpenCL training courses on the web
- Completely open source (creative commons attribution CC BY license)
- Downloaded over 4,200 times so far!
- Lots of training material, examples and solutions, source code etc
- Works on Linux, Windows, OSX etc.

Other useful resources

- Lots of OpenCL examples in the SDKs from the vendors:
  - AMD, Intel, Nvidia, …
- The SHOC OpenCL/CUDA benchmark suite (available as source code):
  - https://github.com/vetter/shoc/wiki
- The GPU-STREAM memory bandwidth benchmark:
  - https://github.com/UoB-HPC/GPU-STREAM

Other useful resources

- IWOCL webpage & newsletter:
  - http://www.iwocl.org
  - http://www.iwocl.org/signup-for-updates/

- IWOCL annual conference
  - Spring each year
  - In Vienna, April 19-21 2016!

Conclusion

- OpenCL
  - Widespread industrial support
  - Defines a platform-API/framework for heterogeneous parallel computing, not just GPGPU or CPU-offload programming
  - Has the potential to deliver portably performant code; but it has to be used correctly

Agenda

- Logistics
- Introduction to Heterogeneous computing
- OpenCL
  - Overview
  - Host Programs
  - Kernel Programs
  - Kernel code optimization
  - OpenCL ecosystem
- OpenMP
  - Overview
  - “Target” and “Target data” directives
  - Mapping onto GPUs: the distribute directive

OpenMP* overview:

C$OMP FLUSH

#pragma omp critical

C$OMP THREADPRIVATE(/ABC/)

CALL OMP_SET_NUM_THREADS(10)

C$OMP PARALLEL COPYIN(/blk/)

C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

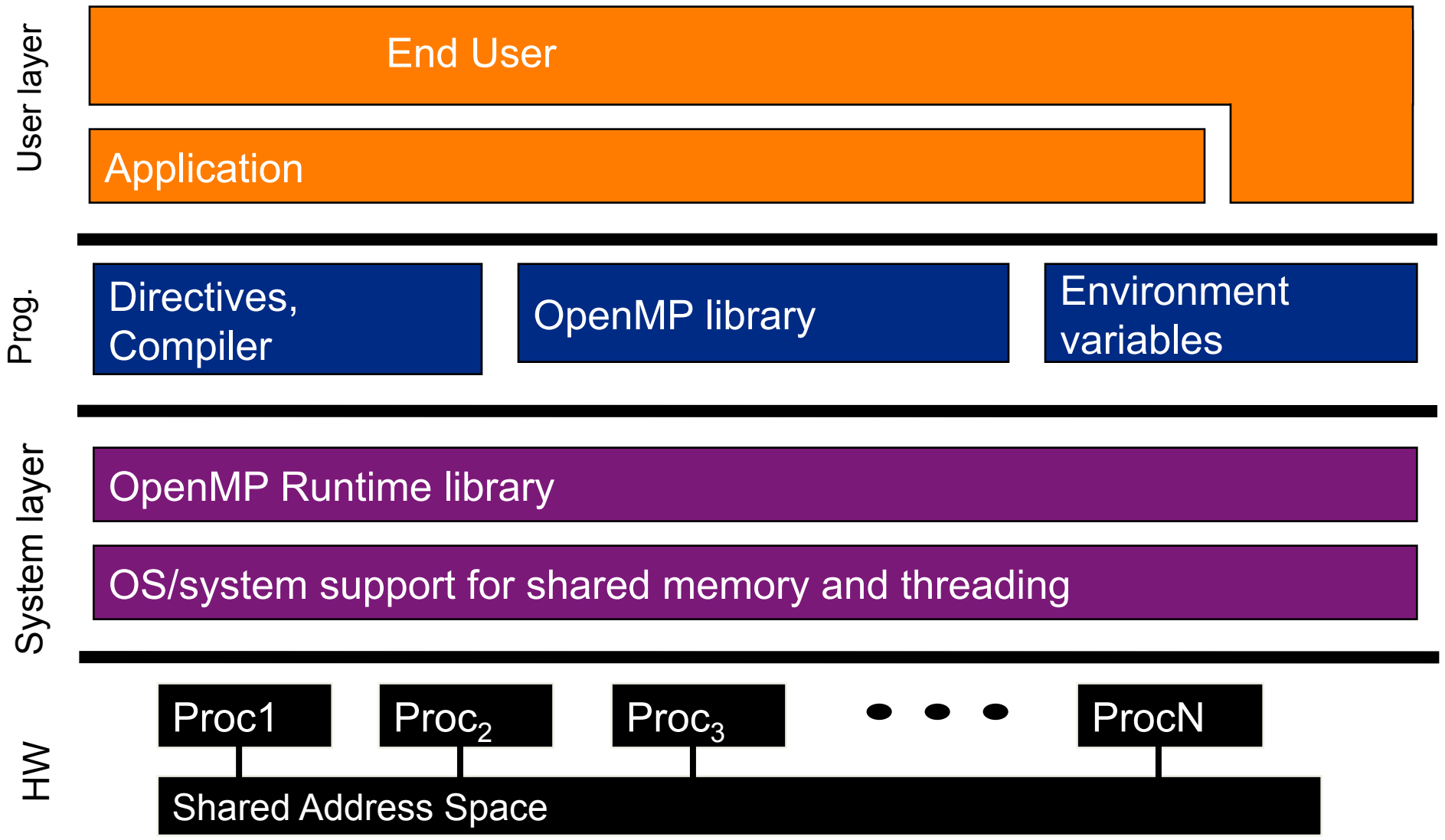
omp_set_lock(lck)

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes established SMP practice + vectorization and heterogeneous device programming

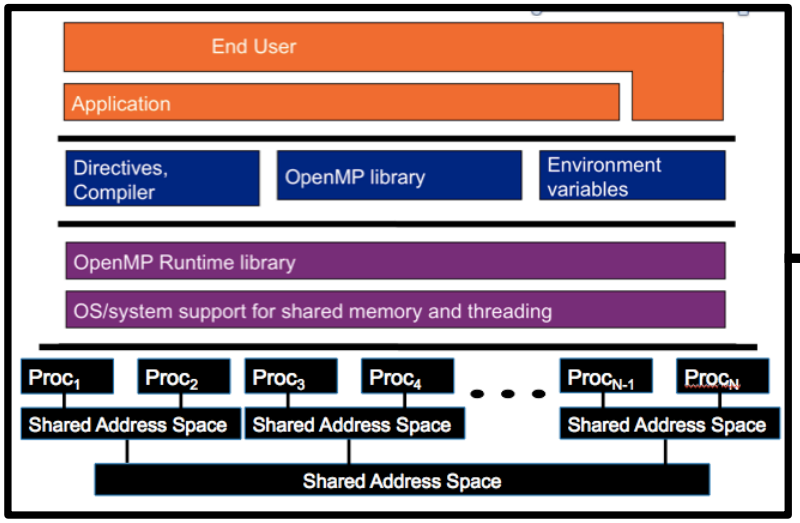
* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

OpenMP basic definitions: Basic Solution stack

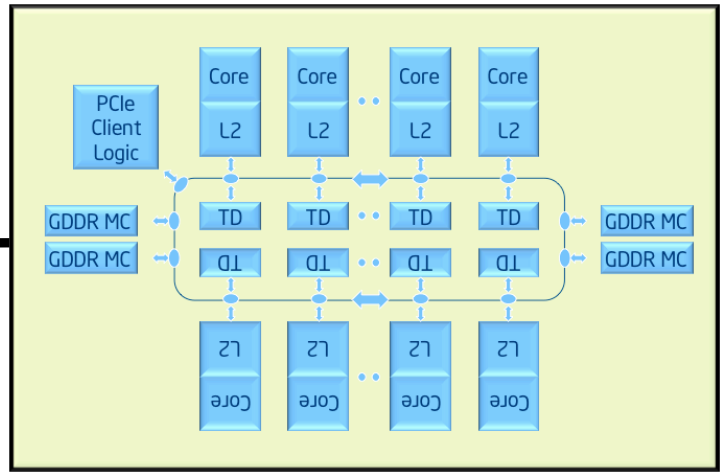


OpenMP basic definitions: Target solution stack

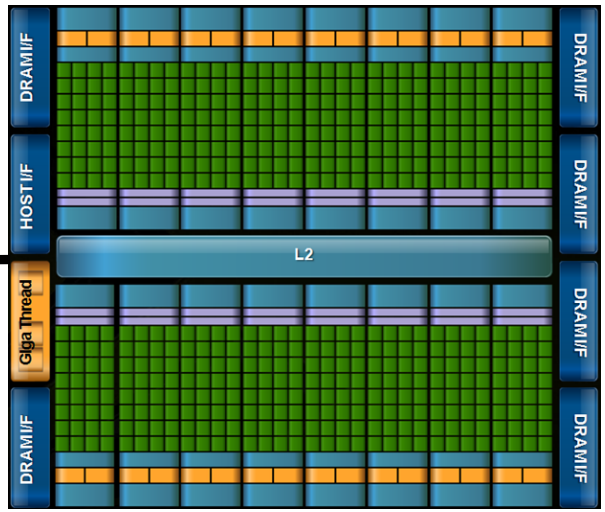
Supported (since OpenMP 4.0) with target, teams, distribute, and other constructs



Host



Target Device: Xeon Phi™ processor



Target Device: GPU

OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.

  #pragma omp construct [clause [clause]…]

  – Example

  #pragma omp parallel num_threads(4)

- Function prototypes and types in the file:

  #include <omp.h>
  use omp_lib

- Most OpenMP* constructs apply to a "structured block".
  – Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  – It's OK to have an exit() within the structured block.

Exercise, Part A: Hello world

Verify that your environment works

- Write a program that prints “hello world”.

```
#include<stdio.h>
int main()
{




    int ID = 0;

    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);

}
```

Exercise, Part B: Hello world

Verify that your OpenMP environment works

- Write a multithreaded program that prints “hello world”.

```
#include <omp.h>
#include <stdio.h>
int main()
{
 #pragma omp parallel
 {
    int ID = 0;
    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);
 }
}
```

Switches for compiling and linking

gcc -fopenmp Linux, OSX

pgcc -mp pgi

icl /Qopenmp intel (windows)

icc –openmp intel (linux)

Exercise: Solution

A multi-threaded “Hello world” program

- Write a multithreaded program where each thread prints “hello world”.

```
#include <omp.h>
#include <stdio.h>
int  main()
{

#pragma omp parallel
 {

    int ID = omp_get_thread_num();
    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);
  }
}
```

OpenMP include file

Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

Sample Output:

hello(1) hello(0) world(1)

world(0)

hello (3) hello(2) world(3)

world(2)

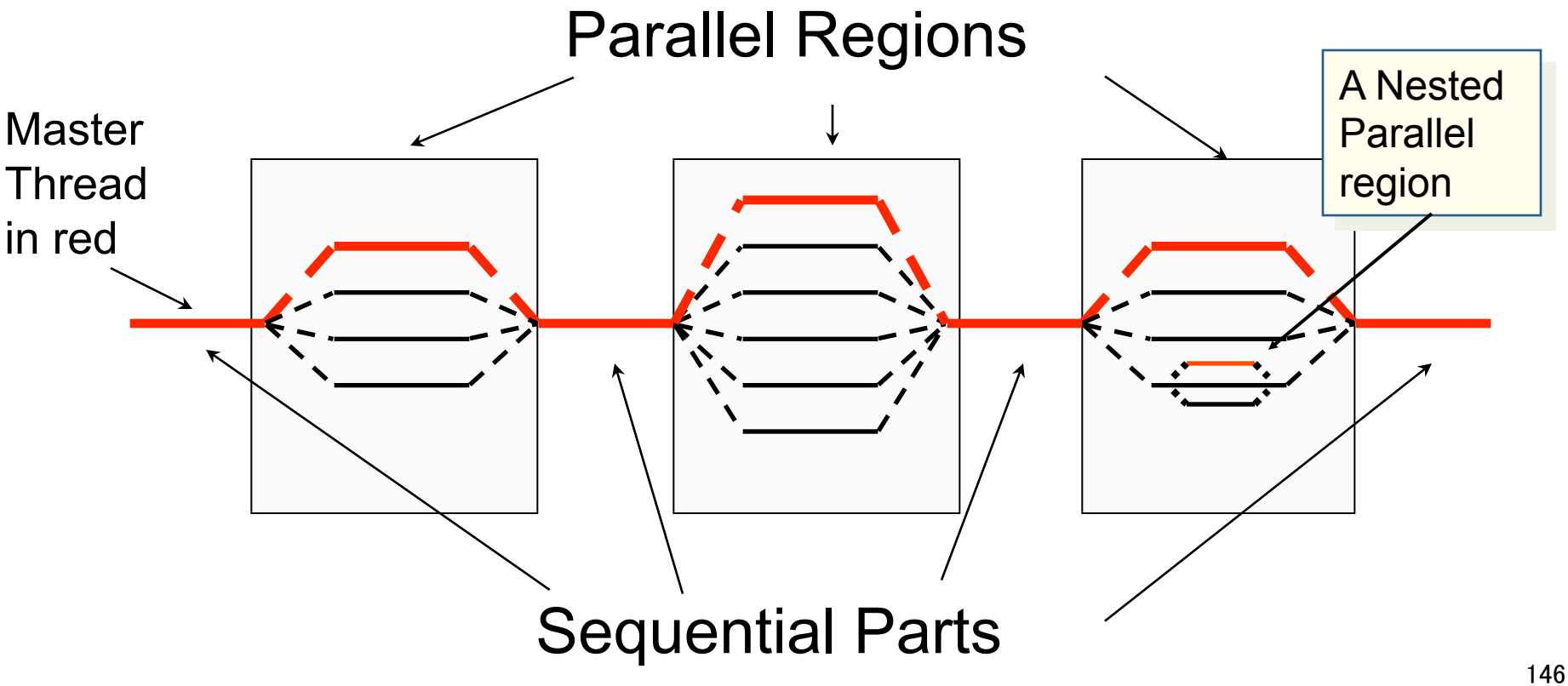
OpenMP overview:
How do threads interact?

- OpenMP is a multi-threading, shared address model
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - Race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

OpenMP programming model:

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



Thread creation: Parallel regions

- You create threads in OpenMP* with the parallel construct.
- For example, To create a 4 thread Parallel region:

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

Each thread executes a copy of the code within the structured block

Runtime function to request a certain number of threads

Runtime function returning a thread ID

- Each thread calls pooh(ID,A) for ID = 0 to 3

Thread creation: Parallel regions

- You create threads in OpenMP* with the parallel construct.
- For example, To create a 4 thread Parallel region:

```
double A[1000];

#pragma omp parallel num_threads(4)
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

clause to request a certain number of threads

Each thread executes a copy of the code within the structured block

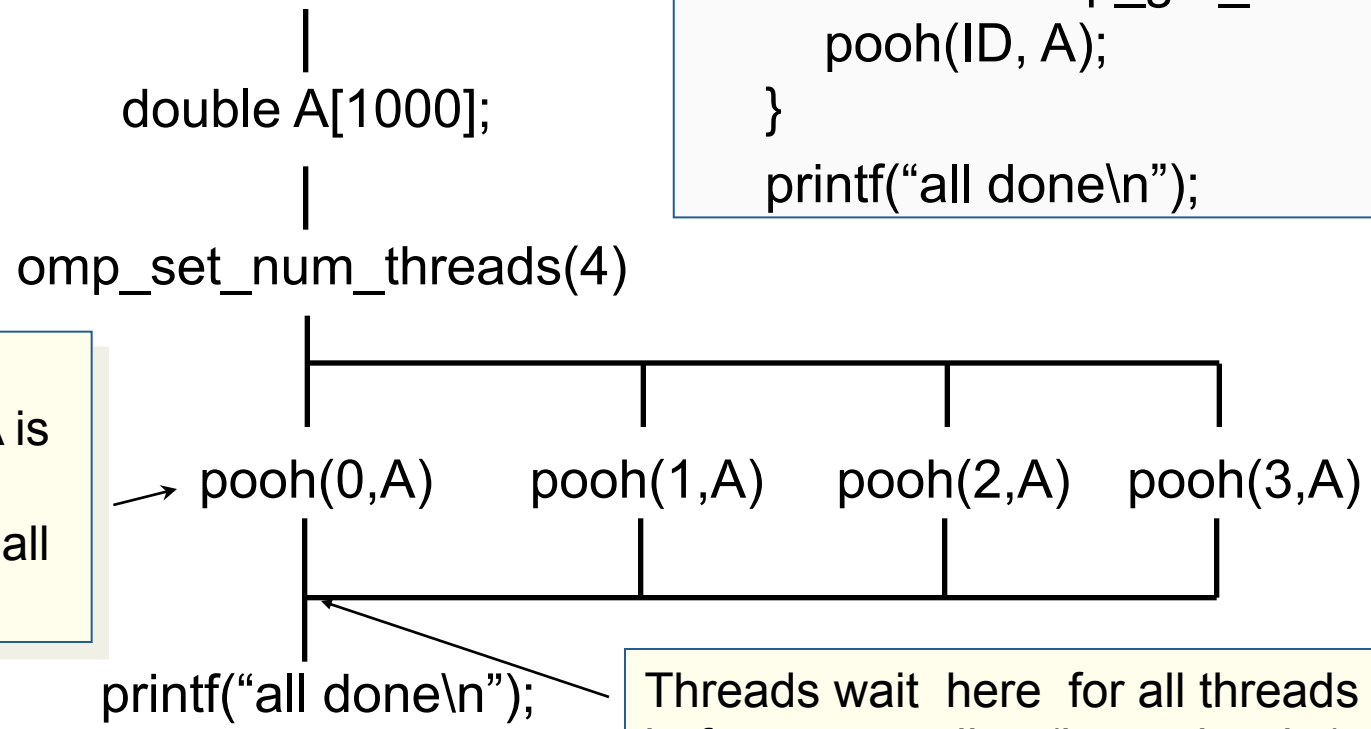
Runtime function returning a thread ID

- Each thread calls pooh(ID,A) for ID = 0 to 3

Thread creation: Parallel regions example

- Each thread executes the same code redundantly.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e., a *barrier*)

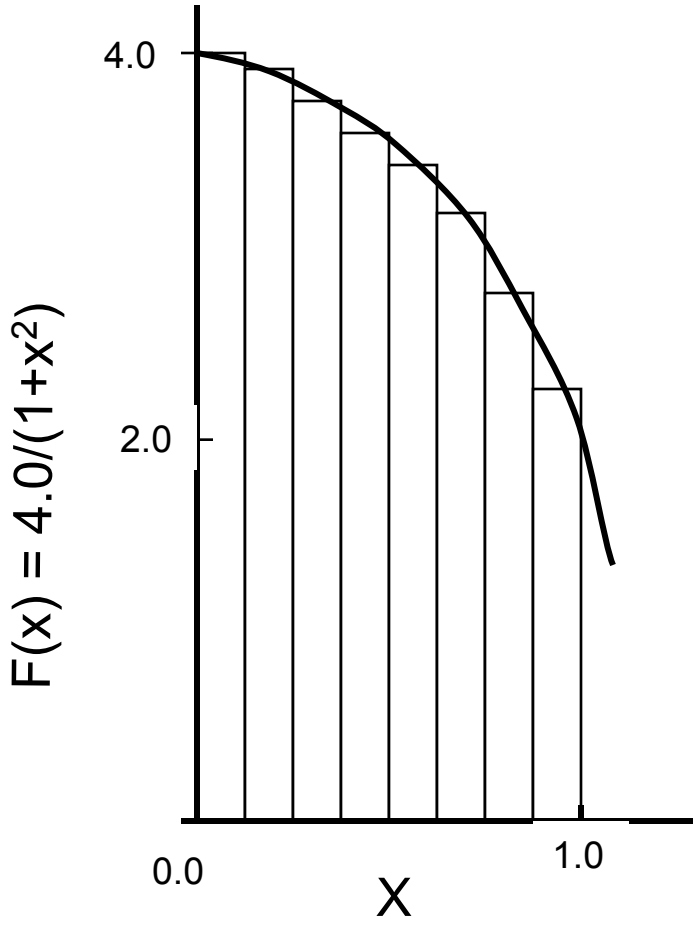
Synchronization: critical

- Mutual exclusion: Only one thread at a time can enter a **critical** region.

Threads wait their turn – only one at a time calls consume()

```
float  res;
#pragma omp parallel
{     float B;   int i, id, nthrds;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
       for(i=id;i<niters;i+=nthrds){
            B =  big_job(i);
#pragma omp critical
            res += consume (B);
      }
}
```

Exercises: Numerical integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Example: An SPMD solution to the PI program

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{              int nthreads; double  pi=0.0;              step = 1.0/(double) num_steps;
               omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
               int i, id, nthrds;    double x, sum;
               id = omp_get_thread_num();
               nthrds = omp_get_num_threads();
               if (id == 0)   nthreads = nthrds;
                 for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
                              x = (i+0.5)*step;
                              sum += 4.0/(1.0+x*x);
                 }
             #pragma omp critical
                        pi += sum * step;
}
}
```

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes “out of scope” beyond the parallel region … so you must sum it in here. Must protect summation into pi in a critical region so updates don’t conflict

Example: An SPMD solution to the PI program

```
#include <omp.h>
static long num_steps = 100000000;        double step;
#define NUM_THREADS 2
void main ()
{             int nthreads; double  pi=0.0;             step = 1.0/(double) num_steps;
              omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
             int i, id, nthrds;    double x, sum;
             id = omp_get_thread_num();
             nthrds = omp_get_num_threads();
             if (id == 0)   nthreads = nthrds;
               for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
                           x = (i+0.5)*step;
                           sum += 4.0/(1.0+x*x);
               }
            #pragma omp critical
                    pi += sum * step;
}
}
```

threads	SPMD critical
1	1.87
2	1.00
3	0.68
4	0.53

Original Serial program ran in 1.83 seconds.

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

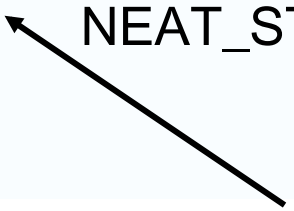
SPMD vs. worksharing

- A parallel construct by itself creates an SPMD or "Single Program Multiple Data" program … i.e., each thread redundantly executes the same code.
- How do you split up pathways through the code between threads within a team?
  - Worksharing constructs
    - Loop construct
    - Sections/section constructs
    - Single construct
    - Distribute construct
  - Task constructs

The loop worksharing constructs

- The loop worksharing construct splits up loop iterations among the threads in a

```
#pragma omp parallel
{
#pragma omp for
        for (I=0;I<N;I++){
                NEAT_STUFF(I);
        }
}
```



The variable I is made “private” to each thread by default. You could do this explicitly with a “private(I)” clause

Loop construct name:

- C/C++: for
- Fortran: do

Reduction

- How do we handle this case?

```
double  ave=0.0, A[MAX];    int i;
  for (i=0;i< MAX; i++) {
        ave + = A[i];
  }
  ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can't be trivially removed
- This is a very common situation … it is called a "reduction".
- Support for reduction operations is included in most parallel programming environments.

Reduction

- OpenMP reduction clause:

  reduction (op : list)

- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in “list” must be shared in the enclosing parallel region.

```
double  ave=0.0, A[MAX];    int i;
#pragma omp parallel for reduction (+:ave)
 for (i=0;i< MAX; i++) {
       ave + = A[i];
 }
 ave = ave/MAX;
```

OpenMP: Reduction operands/initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.

Exercise: Pi with loops

- Go back to the serial pi program and parallelize it with a loop construct
- Your goal is to minimize the number of changes made to the serial program.

Example: Pi with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{    int i;            double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
      #pragma omp parallel
      {
          double x;
         #pragma omp for reduction(+:sum)
             for (i=0;i< num_steps; i++){
                    x = (i+0.5)*step;
                    sum = sum + 4.0/(1.0+x*x);
             }
       }
            pi = step * sum;
}
```

Create a team of threads … without a parallel construct, you'll never have more than one thread

Create a scalar local to each thread to hold value of x at the center of each interval

Break up loop iterations and assign them to threads … setting up a reduction into sum. Note … the loop index is local to a thread by default.

Results*: pi with a loop and a reduction

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: Pi with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{   int i;          double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel
    {
        double x;
      #pragma omp for reduction(+:sum)
          for (i=0;i< num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
          }
    }
        pi = step * sum;
}
```

threads	SPMD critical	PI Loop
1	1.87	1.91
2	1.00	1.02
3	0.68	0.80
4	0.53	0.68

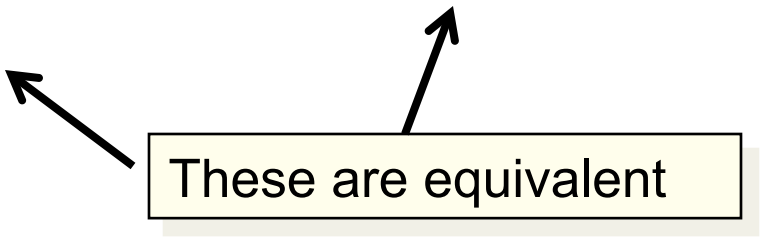
*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Combined parallel/worksharing construct

- OpenMP shortcut: Put the “parallel” and the worksharing directive on the same line

```
 double  res[MAX];  int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

```
 double  res[MAX];  int i;
#pragma omp parallel for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
```



Synchronization: Barrier

- Barrier: Each thread waits until all threads arrive.

```
double A[big], B[big], C[big];

#pragma omp parallel
{
        int id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
        for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
#pragma omp for nowait
        for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }
        A[id] = big_calc4(id);
}
```

implicit barrier at the end of a for worksharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

Data environment: Default storage attributes

- Shared memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
  - Both: dynamically allocated memory (ALLOCATE, malloc, new)
- But not everything is shared...
  - Stack variables in subprograms(Fortran) or functions(C) called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

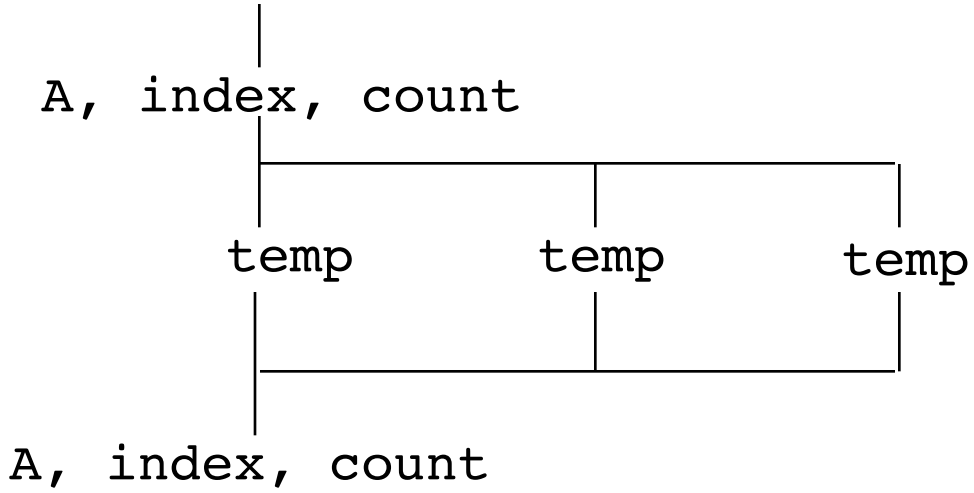
Data sharing: Examples

```
double A[10];
int main() {
int index[10];
#pragma omp parallel
      work(index);
printf("%d\n", index[0]);
}
```

```
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
  ...
}
```

A, index and count are shared by all threads.

temp is local to each thread



Data sharing: Private clause

- private(var) creates a new local copy of var for each thread.
  - The value of the private copies is uninitialized
  - The value of the original variable is unchanged after the region

```
void wrong() {
      int tmp = 0;
#pragma omp parallel for private(tmp)
      for (int j = 0; j < 1000; ++j)
            tmp += j;
      printf("%d\n", tmp);
}
```

tmp was not initialized

tmp is 0 here

Firstprivate clause

- Variables initialized from a shared variable
- C++ objects are copy-constructed

```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
        if ((i%2)==0) incr++;
        A[i] = incr;
}
```

Each thread gets its own copy of incr with an initial value of 0

Exercise

- Start from the jacobi_solv.c file.
- Parallelize the program using openMP loop constructs.
  - #pragma omp parallel for reduction(+:list)
  - Common clauses
    - private(list)
    - firstprivate(list)
    - num_threads(integer-expression)
  - Timing
    - double omp_get_wtime()

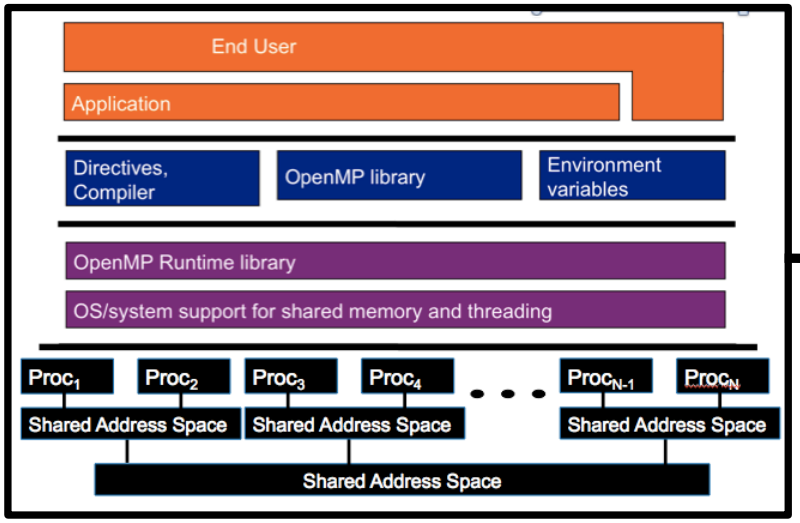
Call before and after …
difference is the elapsed time.

Agenda

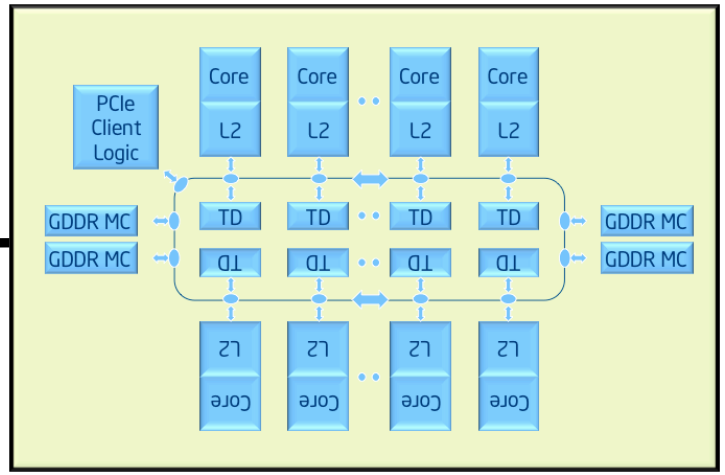
- Logistics
- Introduction to Heterogeneous computing
- OpenCL
  - Overview
  - Host Programs
  - Kernel Programs
  - Kernel code optimization
  - OpenCL ecosystem
- OpenMP
  - Overview
  - “Target” and “Target data” directives
  - Mapping onto GPUs: the distribute directive

OpenMP basic definitions: Target solution stack

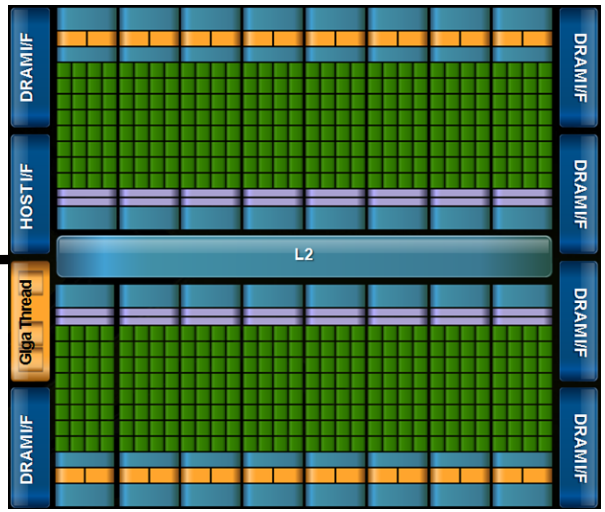
Supported (since OpenMP 4.0) with target, teams, distribute, and other constructs



Host



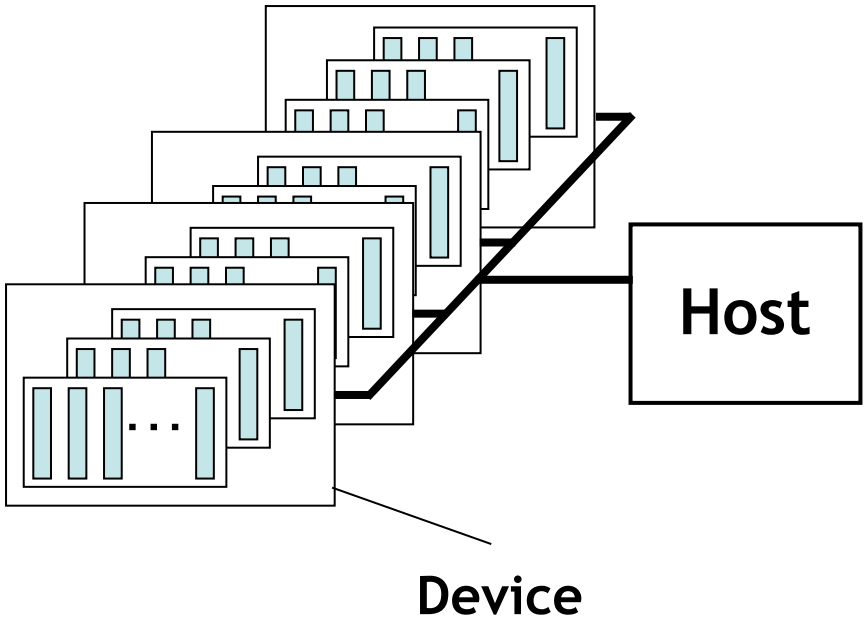
Target Device: Xeon Phi™ processor



Target Device: GPU

The OpenMP device programming model

- OpenMP uses a host/device model
  - The host is where the initial thread of the program begins execution
  - Zero or more devices are connected to the host




```
#include <omp.h>
#include <stdio.h>
Int main()
{
    printf("There are %d devices\n",
            omp_get_num_devices());
}
```

Target directive

- The target construct offloads a code region to a device.

```
#pragma omp target
{….}  // a structured block of code
```

- An initial thread running on the device executes the code in the code block.

```
#pragma omp target
{
    #pragma omp parallel for
        {do lots of stuf}
}
```

Target directive

- The target construct offloads a code region to a device.

```
#pragma omp target device(1)
{….}  // a structured block of code
```

Optional clause to select some device other than the default device.

- An initial thread running on the device executes the code in the code block.

```
#pragma omp target
{
    #pragma omp parallel for
        {do lots of stuf}
}
```

The target data environment

- The target clause creates a data environment on the device:

```
int i, a[N], b[N], c[N];
#pragma omp target
```

Original variables on the host: N, i, a, b, c …

```
#pragma omp parallel for private(i)
        for(i=0;i<N;i++){
            c[i]+=a[i]+b[i];
        }
```

Are mapped onto the corresponding variables on the device: N, i, a, b, c …

- Originals variables copied into corresponding variables before the initial thread begins execution on the device.
- Corresponding variables copied into original variables when the target code region completes

Controlling data movement

```
int i, a[N], b[N], c[N];
#pragma omp target map(to:a,b) map(tofrom:c)
```

Data movement can be explicitly controlled with the map clause

- The various forms of the map clause
  - **map(to:list)**: *read-only* data on the device. Variables in the list are initialized on the device using the original values from the host.
  - **map(from:list)**: *write-only* data on the device: initial value of the variable is not initialized. At the end of the target region, the values from variables in the list are copied into the original variables.
  - **map(tofrom:list)**: the effect of both a map-to and a map-from
  - **map(alloc:list)**: data is allocated and uninitialized on the device.
  - **map(list)**: equivalent to map(tofrom:list).
- For pointers you must use array notation ..
  - Map(to:a[0:N])

Exercise

- Start with the parallel jacobi_solver from the last exercise.
- Use the target clause to offload the execution of this solver on the Xeon-phi.
  - #pragma omp target
  - #pragma omp target map(to:list) map(from:list) map(tofrom:list)
  - int omp_get_num_devices();
  - #pragma omp parallel for reduction(+:var) private(list)

Jacobi Solver (serial 1/2)

```
while((conv > TOL) && (iters<MAX_ITERS))
  {
    iters++;
    xtmp  = xnew;    // don't copy arrays.
    xnew  = xold;    // just swap pointers.
    xold  = xtmp;

    for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
            if(i!=j)
              xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

Jacobi Solver (serial 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
    for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

} \\ end while loop
```

Jacobi Solver (Par Targ, 1/2)

```
while((conv > TOL) && (iters<MAX_ITERS))
  {
    iters++;
    xtmp  = xnew;    // don't copy arrays.
    xnew  = xold;    // just swap pointers.
    xold  = xtmp;
#pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                   map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
     #pragma omp parallel for private(i,j)
    for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
            if(i!=j)
              xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

Jacobi Solver (Par Targ, 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
#pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                        map(to:Ndim) map(tofrom:conv)
  #pragma omp parallel for private(i,tmp) reduction(+:conv)
    for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

} \\ end while loop
```

Jacobi Solver (Par Targ, 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
#pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                        map(to:Ndim) map(tofrom:conv)
  #pragma omp parallel for private(i,tmp) reduction(+:conv)
    for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

} \\ end while loop
```

This worked but the performance was awful. Why?

System	Implementation	Ndim = 1000	Ndim = 4096
Intel® Xeon Phi™ co-processor (knights corner)	Target dir per loop	134 seconds	Did not finish (> 40 minutes)
	Native OMP	3.2 seconds	5.3 seconds

Data movement dominates!!!

```
while((conv > TOLERANCE) && (iters<MAX_ITERS))
  { iters++;
    xtmp  = xnew;    // don't copy arrays.
    xnew  = xold;    // just swap pointers.
    xold  = xtmp;

    #pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                       map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
     #pragma omp parallel for private(i,j)
     for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
           if(i!=j)
             xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
     }
// test convergence
    conv = 0.0;
    #pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                           map(to:Ndim) map(tofrom:conv)
      #pragma omp parallel for private(i,tmp) reduction(+:conv)
      for (i=0; i<Ndim; i++){
         tmp  = xnew[i]-xold[i];
         conv += tmp*tmp;
      }
    conv = sqrt((double)conv);
  }
```

Typically over 4000 iterations!

For each iteration, copy to device $(3*Ndim+Ndim^2)*sizeof(TYPE)$ bytes

For each iteration, copy from device $2*Ndim*sizeof(TYPE)$ bytes

For each iteration, copy to device $2*Ndim*sizeof(TYPE)$ bytes

Target data directive

- The **target data** construct creates a target data region.
- You use the map clauses for explicit data management

```
#pragma omp target data map(to: A,B) map(from: C)
{….}  // a structured block of code
```

- Data copied into the device data environment at the beginning of the directive and at the end
- Inside the **target data** region, multiple **target** regions can work with the single data region

```
#pragma omp target data map(to: A,B) map(from: C)
{
    #pragma omp target
        {do lots of stuff with A, B and C}
    {do something on the host}
    #pragma omp target
        {do lots of stuff with A, B, and C}
}
```

Target update directive

- You can update data between target regions with the target update directive.

```
#pragma omp target data map(to: A,B) map(from: C)
{
    #pragma omp target
        {do lots of stuf with A, B and C}

    #pragma omp  update from(A)

    host_do_something_with(A)

    #pragma omp update to(A)

    #pragma omp target
        {do lots of stuff with A, B, and C}
}
```

Copy A from the device onto the host.

Copy A on the host to A on the device. t

Exercise

- Modify your parallel jacobi_solver from the last exercise.
- Use the target data construct to create a data region. Manage data movement with map clauses to minimize data movement.
  - #pragma omp target
  - #pragma omp target data
  - #pragma omp target map(to:list) map(from:list) map(tofrom:list)
  - int omp_get_num_devices();
  - #pragma omp parallel for reduction(+:var) private(list)

Jacobi Solver (Par Targ Data, 1/2)

```
#pragma omp target data map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                        map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;
     xtmp  = xnew;    // don't copy arrays.
     xnew  = xold;    // just swap pointers.
     xold  = xtmp;
 #pragma update to(xnew[0:Ndim], xold[0:Ndim])
 #pragma omp target
       #pragma omp parallel for private(i,j)
     for (i=0; i<Ndim; i++){
         xnew[i] = (TYPE) 0.0;
         for (j=0; j<Ndim;j++){
             if(i!=j)
               xnew[i]+= A[i*Ndim + j]*xold[j];
         }
         xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
     }
```

Jacobi Solver (Par Targ Data, 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
#pragma omp update to(conv)
#pragma omp target
    #pragma omp parallel for private(i,tmp) reduction(+:conv)
    for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
#pragma omp update from (conv)
    conv = sqrt((double)conv);

} \\ end while loop
```

Jacobi Solver Results: summary

System	Implementation	Ndim = 1000	Ndim = 4096
Intel® Xeon™ processor	parfor	0.55 seconds	21 seconds
	par_for	0.36 seconds	21 seconds
Intel® Xeon Phi™ co-processor (knights corner)	Target dir per loop	134 seconds	Did not finish (> 40 minutes)
	Data region + target per loop	3.4 seconds	12.2 seconds
	Native par_for	3.2 seconds	5.3 seconds
	OpenCL Best	2.9 seconds	32.5 seconds

Source: Tom Deakin and James Prices, University of Bristol, UK. All results with the Intel icc compiler. Compiler options -03.

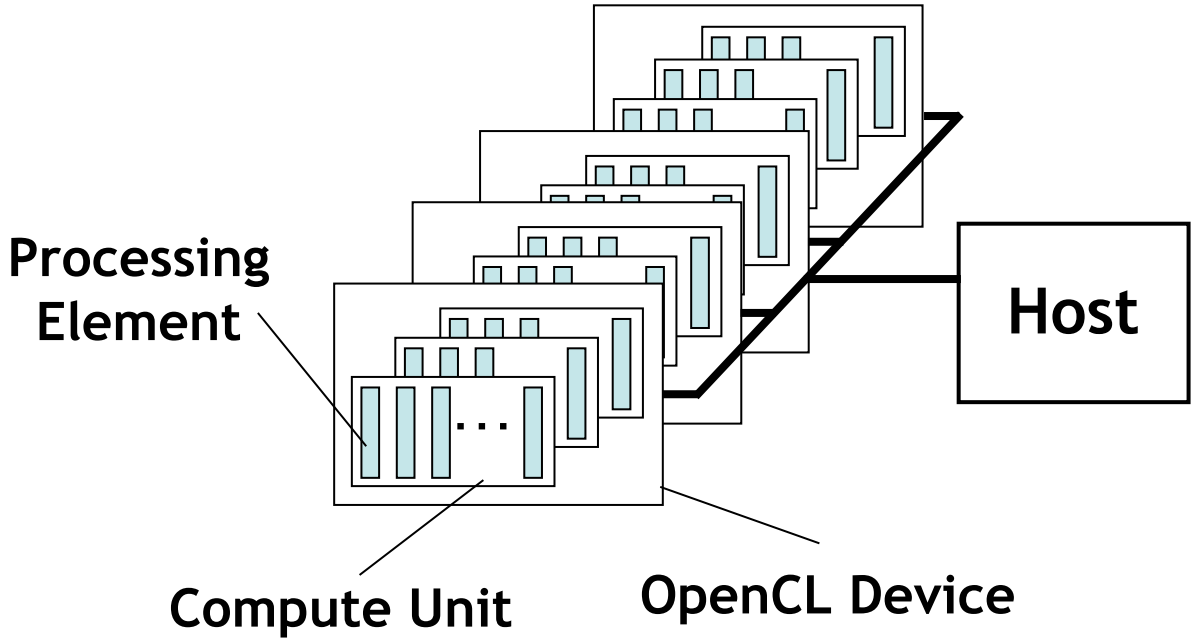
Agenda

- Logistics
- Introduction to Heterogeneous computing
- OpenCL
  - Overview
  - Host Programs
  - Kernel Programs
  - Kernel code optimization
  - OpenCL ecosystem
- OpenMP
  - Overview
  - “Target” and “Target data” directives
  - Mapping onto GPUs: the distribute directive

Mapping onto more complex devices

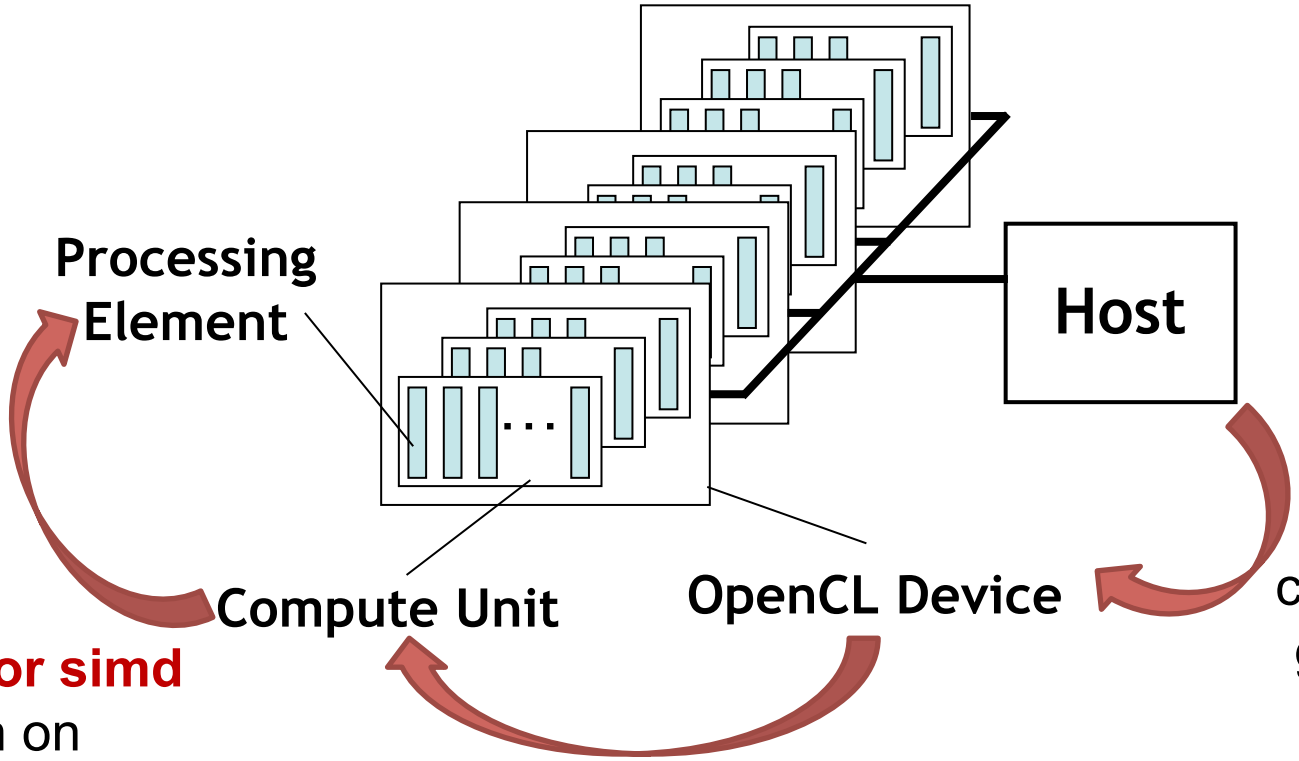
- So far, we have just “off-loaded” OpenMP code onto a general purpose CPU device that supports OpenMP multithreaded parallelism.
- How would we map OpenMP 4.0 onto a more specialized, throughput oriented device such as a GPU?

OpenCL Platform Model



- One ***Host*** and one or more ***OpenCL Devices***
  - Each OpenCL Device is composed of one or more ***Compute Units***
    - Each Compute Unit is divided into one or more ***Processing Elements***
- Memory divided into ***host memory*** and ***device memory***

OpenCL Platform Model and OpenMP



Target construct to get onto a device

Parallel for simd to run on processing elements + vector units

Teams construct to create a league of teams with one team of threads on each compute unit.

Distribute clause to assign work-groups to teams.

Consider our familiar VADD example

```
#include<omp.h>
#include<stdio.h>
#define N  1024
int main()
{
   float a[N], b[N], c[N];
   int i;

// initialize a, b and c ….

   for(i=0;i<N;i++)
         c[i] += a[i] + b[i];

// Test results, report results …

}
```

We will explore how to map this code onto Many-core processors (GPU and CPU) using the OpenMP constructs:

- target
- teams
- distribute

2 Constructs to control devices

- **teams** construct creates a league of thread teams:

  #pragma omp teams
- Supports the clauses:
  - num_teams(int) … the number of teams in the league
  - thread_limit(int) … max number of threads per team
  - Plus private(), firstprivate() and reduction()

- **distribute** construct distributes iterations of following loops to the master thread of each team in a league:

  #pragma omp distribute

  //immediately following for loop(s)
- Supports the clauses:
  - dist_schedule(static [, chunk] … the number of teams in the league.
  - collapse(int) … combine n closely nested loop into one before distributing.
  - Plus private(), firstprivate() and reduction()

Vadd: OpenMP to OpenCL connection

```
#pragma omp target map(to:a,b) map(tofrom:c)

#pragma omp teams num_teams(NCU) thread_limit(NPE)

#pragma omp distribute
 for (ib=0;ib<N; ib=ib+wrk_grp_sz)

   #pragma omp parallel for simd
   for (i=ib;  i<ib+wrk_grp_sz; i++)
     c[i] += a[i] + b[i];
```

Offload to a device.

Describe a device … NCU compute units & NPE proc. elements per compute unit

Distribute work-groups to compute units

The body of this loop are the Individual work-items in a work-group

Vadd: OpenMP to OpenCL connection

```
 int blksz=32, ib, Nblk;
 Nblk = N/blksz;
#pragma omp target map(to:a,b) map(tofrom:c)
   #pragma omp teams num_teams(NCU) thread_limit(NPE)

   #pragma omp distribute
    for (ib=0;ib<Nblk;ib++){
      int ibeg=ib*blksz;
      int iend=(ib+1)*blksz;
      if(ib==(Nblk-1))iend=N;

      #pragma omp parallel for simd
      for (i=ibeg; i<iend; i++)
        c[i] += a[i] + b[i];
   }
```

You can include any work-group wide code you want .. For example to explicitly control how iterations map onto work items in a work-group.

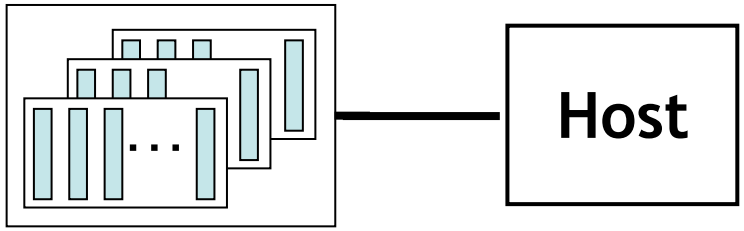
Vadd: OpenMP to OpenCL connection

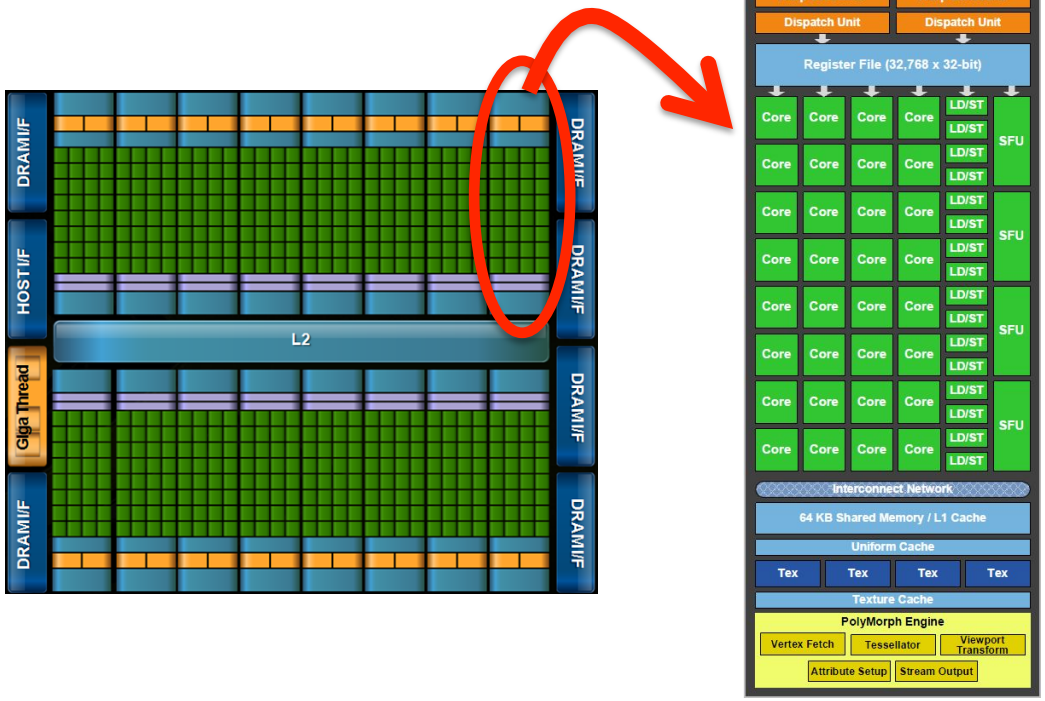
```
// A more compact way to write the VADD code, letting the runtime
//  worry about work-group details

#pragma omp target map(to:a,b) map(tofrom:c)
 #pragma omp teams distribute parallel for
      for (i=0; i<N; i++)
              c[i] += a[i] + b[i];
```

In many cases, you might be better off to just distribute the parallel loops to the league of teams and leave it to the runtime system to manage the details. This would be more portable code as well.

OpenMP Platform Model: GPU

- Let's consider one host and one Device.





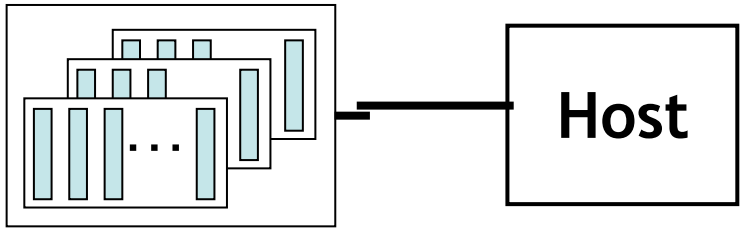
NVIDIA Tesla C2050 (Fermi) GPU with 14 streaming multiprocessor cores*.

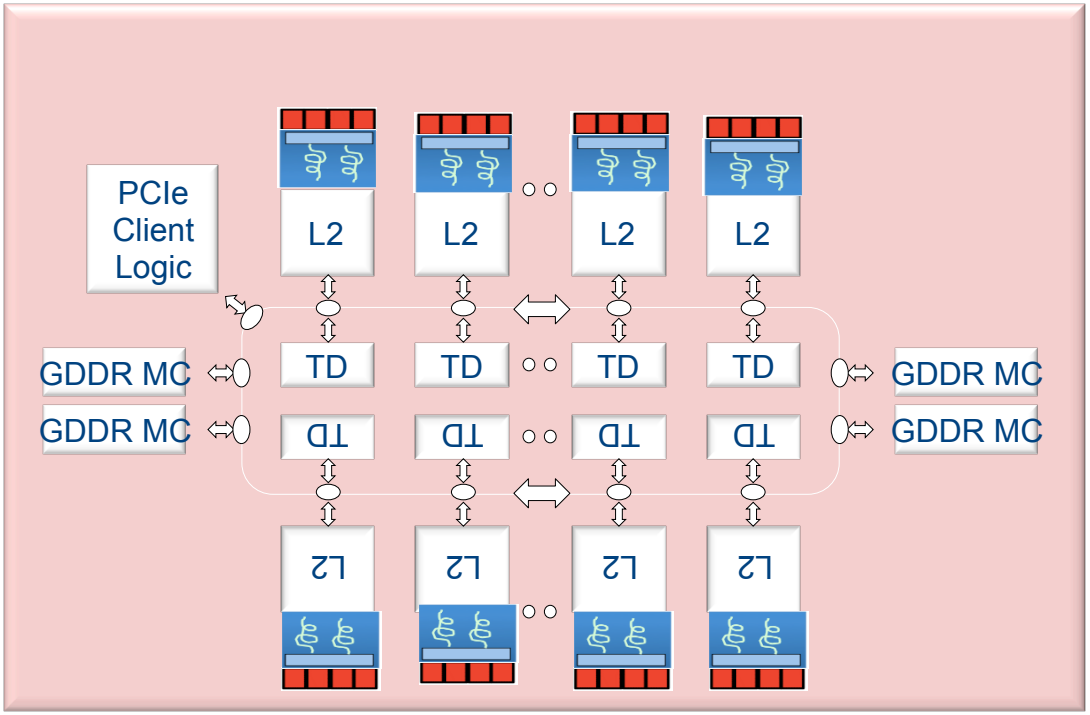
- Number of compute units: 14
- Number of PEs: 32
- Ideal work-group size: multiple of 32

*Source: http://www.nersc.gov/users/computational-systems/dirac/node-and-gpu-configuration/

OpenMP Platform Model: Intel® Xeon Phi™ processor

- Let's consider one host and one Device.





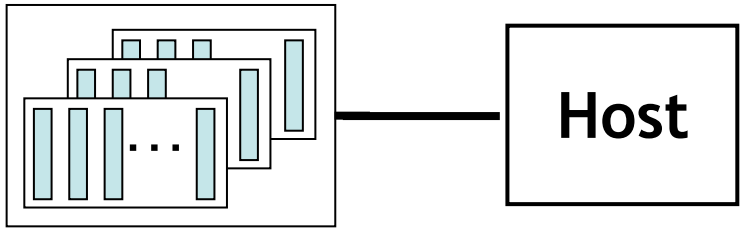
Intel® Xeon Phi™ processor: 60 cores, with 2 HW threads per core and a 512 bit wide vector unit.

- Number of compute units: 60
- Number of PEs: 2*vector width
- Ideal work-group size: multiple of vector width

Where "vector width" depends floating point type: 512/4*8 for float, 512/8*8 for double.

OpenMP Platform Model: summary

- Let's consider one host and one Device.



Device: GPU

NVIDIA Tesla C2050 (Fermi) GPU with 14 streaming multiprocessor cores*.

- Number of compute units: 14
- Number of PEs: 32
- Ideal work-group size: multiple of 32

Device: Many Core CPU

Intel® Xeon Phi™ processor: 60 cores, with 2 HW threads per core and a 512 bit wide vector unit.

- Number of compute units: 60
- Number of PEs: 2*vector width
- Ideal work-group size: multiple of vector width

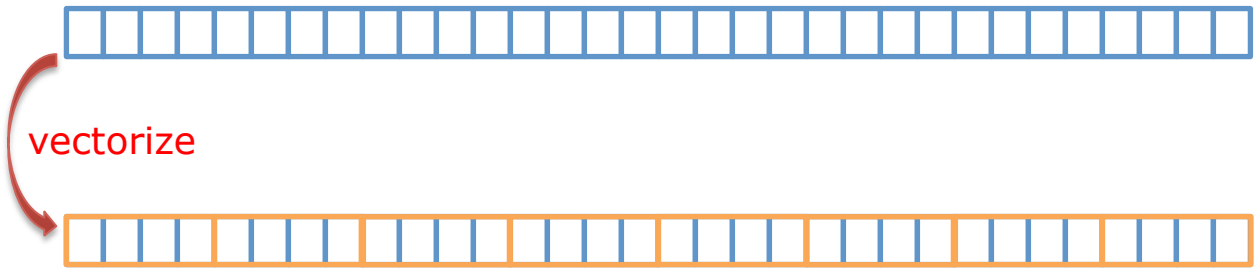
Where "vector width" depends floating point type: 512/4*8 for float, 512/8*8 for double.

OpenMP SIMD Loop Construct

- Vectorize a loop nest
  - Cut loop into chunks that fit a SIMD vector register

```
#pragma omp simd [clause[[,] clause],…]
for-loops
```

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;

  #pragma omp simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



Data Sharing Clauses

- **`private(var-list)`:**
  Uninitialized vectors for variables in *var-list*

- **`firstprivate(var-list)`:**
  Initialized vectors for variables in *var-list*

- **`reduction(op:var-list)`:**
  Create private variables for *var-list* and apply reduction operator *op* at the end of the construct

SIMD Loop Clauses

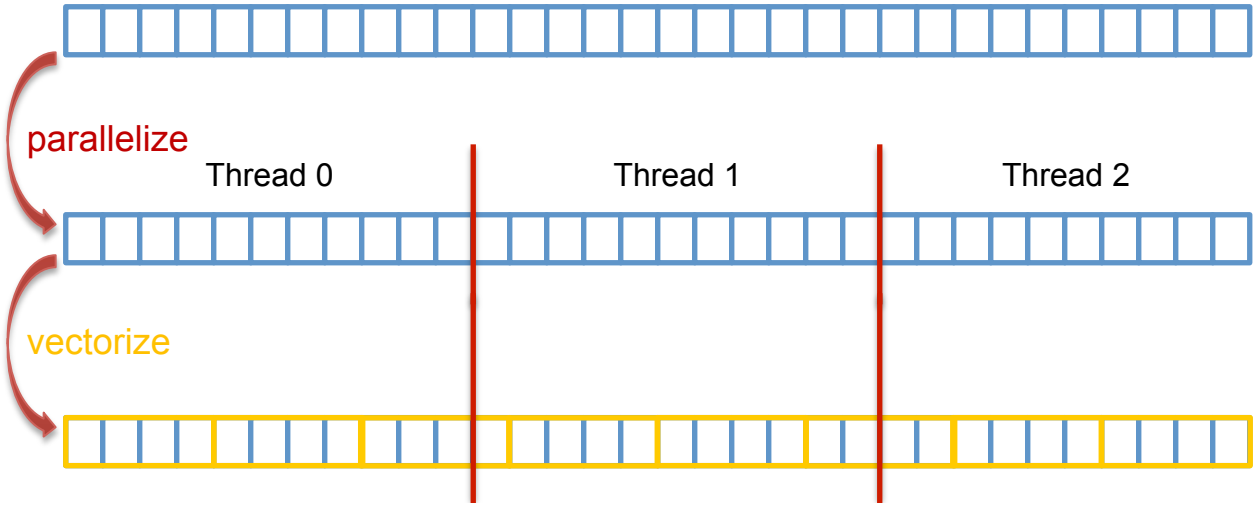
- **`safelen (`*`length`*`)`**
  - Maximum number of iterations that can run concurrently without breaking a dependence
  - in practice, maximum vector length
- **`linear (`*`list`*`[:`*`linear-step`*`])`**
  - The variable's value is in relationship with the iteration number

    $x_i = x_{orig} + i * \text{linear-step}$
- **`aligned (`*`list`*`[:`*`alignment`*`])`**
  - Specifies that the list items have a given alignment
  - Default is alignment for the architecture
- **`collapse (`*`n`*`)`**

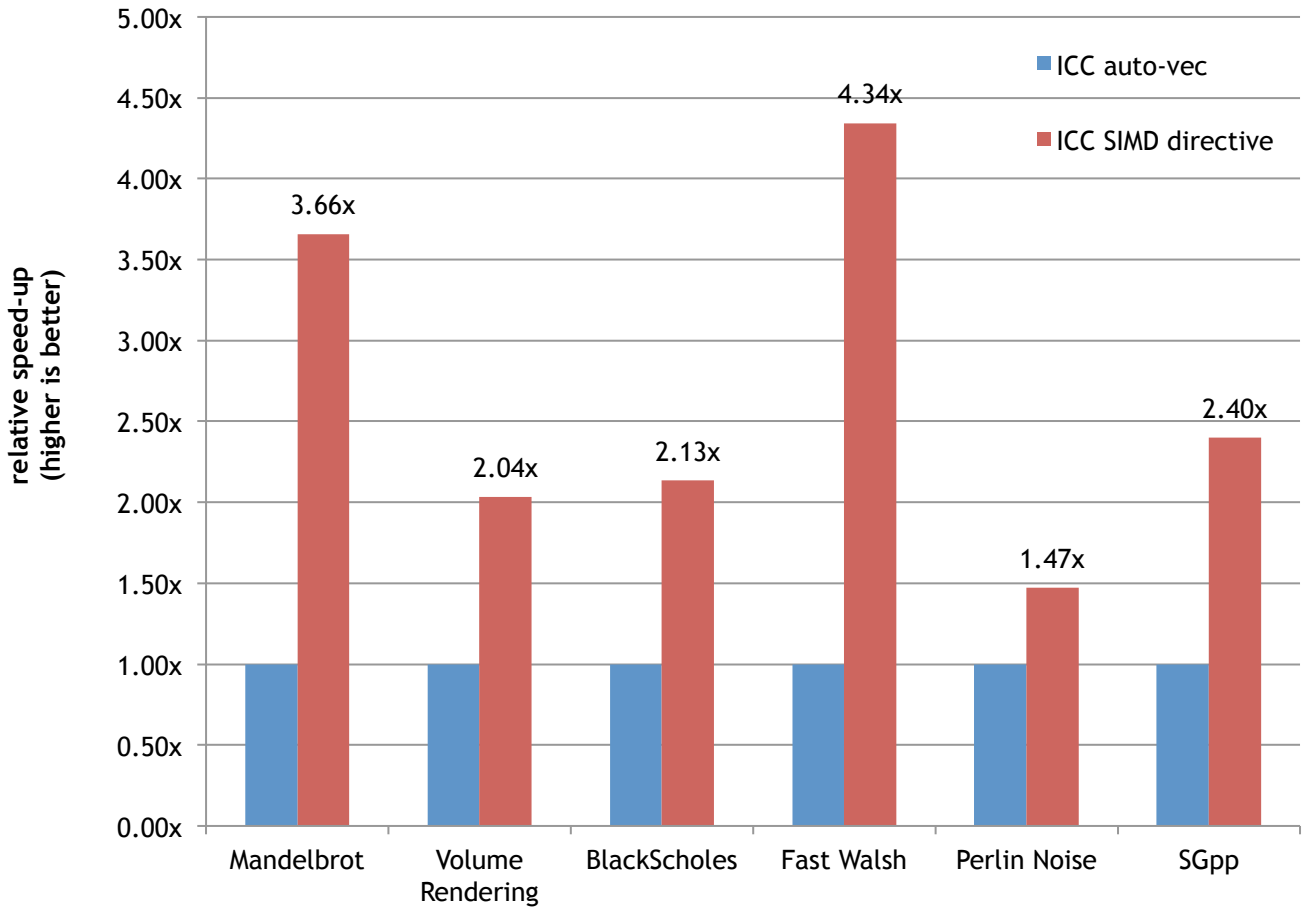
SIMD Worksharing Construct

- Parallelize and vectorize a loop nest
  - Distribute a loop's iteration space across a thread team
  - Subdivide loop chunks to fit a SIMD vector register
- `#pragma omp for simd` *`[clause[[,] clause],…]`*
  `for-loops`

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp parallel for simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



Performance of the SIMD Constructs



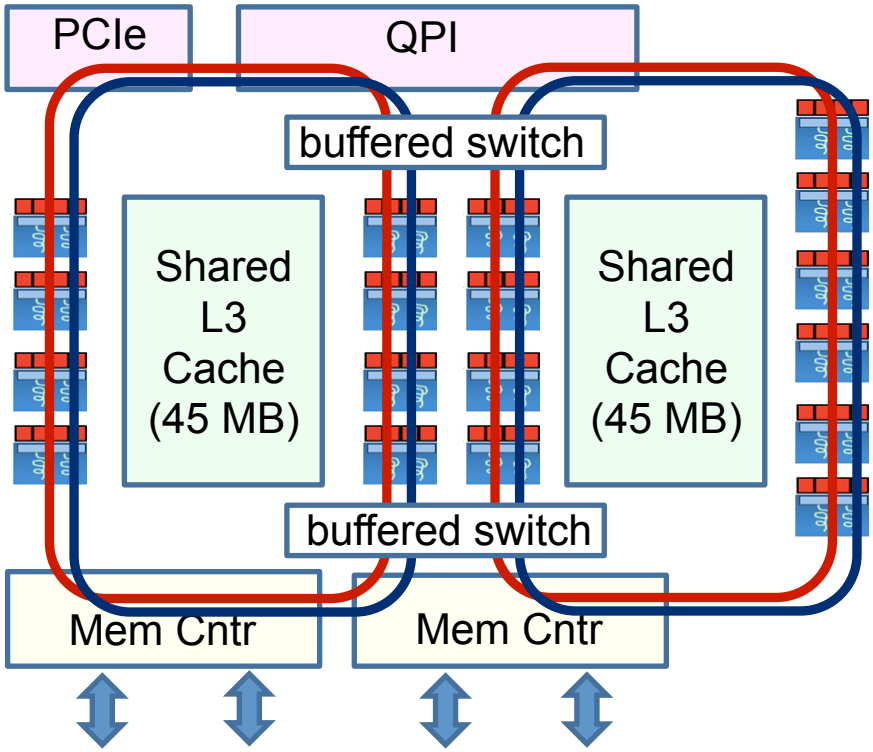
- M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell. Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP, pages 59-72, Rome, Italy, June 2012. LNCS 7312.

Exercise

- Two options:
  1. Modify your parallel jacobi_solver from the last exercise. Use the teams and distribute constructs to see if you can improve the performance of your code.
  2. Return to the pi program and see if you can achieve reasonable performance if you use the target, distribute and teams constructs. Hint: the SIMD clauses are critical in this case.
- target data construct to create a data region. Manage data movement with map clauses to minimize data movement.
  - #pragma omp target
  - #pragma omp target data
  - #pragma omp target teams num_teams(int) thread_limit(int)
  - #pragma omp distribute dist_schedule(static[, chunk])
  - #pragma omp target map(to:list) map(from:list) map(tofrom:list)
  - Int omp_get_num_devices();
  - #pragma omp parallel for reduction(+:var) private(list)

Conclusion

- OpenCL
  - Widespread industrial support
  - Defines a platform-API/framework for heterogeneous parallel computing, not just GPGPU or CPU-offload programming
  - Has the potential to deliver portably performant code; but it has to be used correctly
- OpenMP
  - Established technology for programming shared memory systems.
  - Growing and expanding over time to add NUMA, explicit vectorization, and programming heterogenous platforms.
- Between these two options, a wide range of programming styles are supported … there is no good excuse to use a non-portable/proprietary API.


PCIe
QPI
buffered switch
Shared
L3
Cache
(45 MB)
Shared
L3
Cache
(45 MB)
buffered switch
Mem Cntr
Mem Cntr