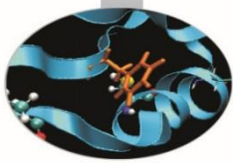
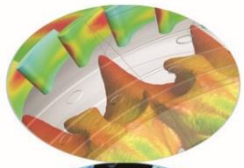
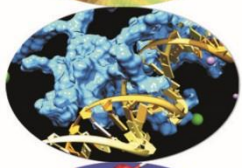
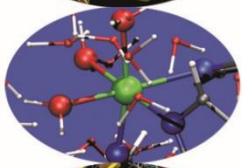
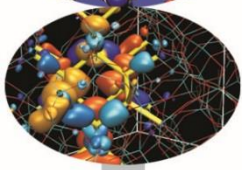
CINECA SCAI
SuperComputing Applications and Innovation

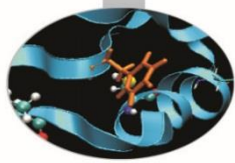


From CUDA to OpenCL

Piero Lanucara, *SCAI*
p.lanucara@cineca.it

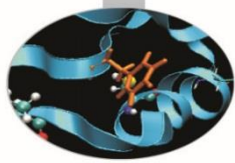
PRACE
CINECA



- Let's start from a simple CUDA code (matrixMul from NVIDIA CUDA samples).
  - Now, you perfectly know how to compile and run on NVIDIA hardware (Galileo, one K80 device)
  - You should probably see something like this:

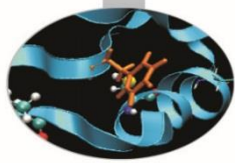
```
[planucar@node495 matrixMul]$ ./matrixMul -wA=2048 -wB=2048 -hA=2048 -hB=2048
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "Tesla K80" with compute capability 3.7

MatrixA(2048,2048), MatrixB(2048,2048)
Computing result using CUDA Kernel...
done
Performance= 377.17 GFlop/s, Time= 45.550 msec, Size= 17179869184 Ops,
WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may
vary when GPU Boost is enabled.
```

Houston, we have a problem!



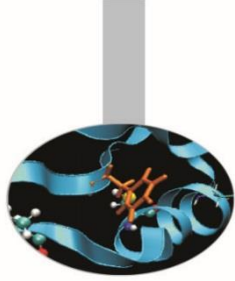
- If you have CUDA code and NVIDIA hardware you're already satisfied (apart from any performance consideration)
- On the other hand, if you have CUDA code and non NVIDIA hardware you could have a big problem…and your application will be the shortest way to produce a "seg-fault" ☺
- You need a more "portable" solution to this problem…
- ….OpenCL?

Don’t worry, be happy!



- If you have CUDA code, you’ve already done the hard work!
  - I.e. working out how to split up the problem to run effectively on a many-core device
- Switching between CUDA and OpenCL is mainly changing the host code syntax
  - Apart from indexing and naming conventions in the kernel code (simple to change!)

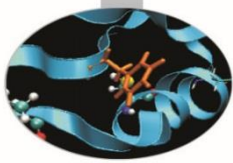
CUDA vector addition



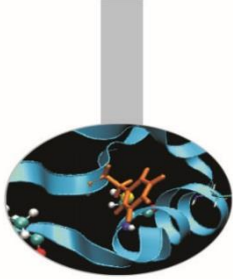
- The “hello world” program of data parallel programming is a program to add two vectors

C[i] = A[i] + B[i] for i=0 to N-1

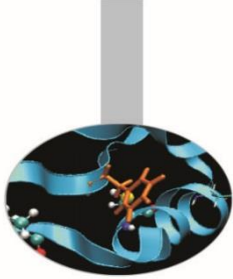
- CUDA simple NVIDIA Sample listing follows



```
/**
 * Copyright 1993-2015 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

/**
 * Vector addition: C = A + B.
 *
 * This sample is a very basic sample that implements element by element
 * vector addition. It is the same as the sample illustrating Chapter 2
 * of the programming guide with some additions like error checking.
 */

#include <stdio.h>

// For the CUDA runtime routines (prefixed with "cuda_")
#include <cuda_runtime.h>

/**
 * CUDA Kernel Device code
 *
 * Computes the vector addition of A and B into C. The 3 vectors have the same
 * number of elements numElements.
 */
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}

/**
 * Host main routine
 */
```

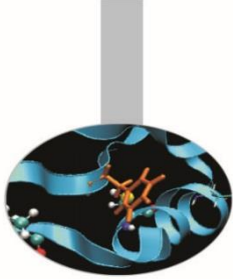


```
int
main(void)
{
    // Error code to check return values for CUDA calls
    cudaError_t err = cudaSuccess;

    // Print the vector length to be used, and compute its size
    int numElements = 50000;
    size_t size = numElements * sizeof(float);
    printf("[Vector addition of %d elements]\n", numElements);

    // Allocate the host input vector A
    float *h_A = (float *)malloc(size);

    // Allocate the host input vector B
    float *h_B = (float *)malloc(size);

    // Allocate the host output vector C
    float *h_C = (float *)malloc(size);

    // Verify that allocations succeeded
    if (h_A == NULL || h_B == NULL || h_C == NULL)
    {
        fprintf(stderr, "Failed to allocate host vectors!\n");
        exit(EXIT_FAILURE);
    }

    // Initialize the host input vectors
    for (int i = 0; i < numElements; ++i)
    {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }

    // Allocate the device input vector A
    float *d_A = NULL;
    err = cudaMalloc((void **)&d_A, size);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector A (error
code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
```

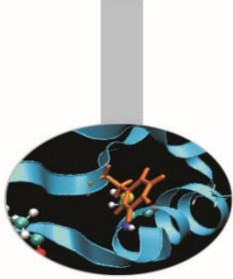


```
// Allocate the device input vector B
    float *d_B = NULL;
    err = cudaMalloc((void **)&d_B, size);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector B (error
code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Allocate the device output vector C
    float *d_C = NULL;
    err = cudaMalloc((void **)&d_C, size);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector C (error
code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
```

```
// Copy the host input vectors A and B in host memory to the
device input vectors in
    // device memory
    printf("Copy input data from the host memory to the CUDA
device\n");
    err = cudaMemcpy(d_A, h_A, size,
cudaMemcpyHostToDevice);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to copy vector A from host to device
(error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    err = cudaMemcpy(d_B, h_B, size,
cudaMemcpyHostToDevice);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to copy vector B from host to device
(error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
```



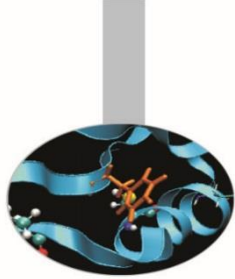
```
 // Launch the Vector Add CUDA Kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =(numElements + threadsPerBlock - 1) /
threadsPerBlock;
    printf("CUDA kernel launch with %d blocks of %d
threads\n", blocksPerGrid, threadsPerBlock);
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,
d_B, d_C, numElements);
    err = cudaGetLastError();

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to launch vectorAdd kernel (error
code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Copy the device result vector in device memory to the
host result vector
    // in host memory.
    printf("Copy output data from the CUDA device to the host
memory\n");
    err = cudaMemcpy(h_C, d_C, size,
cudaMemcpyDeviceToHost);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to copy vector C from device to host
(error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Verify that the result vector is correct
    for (int i = 0; i < numElements; ++i)
    {
        if (fabs(h_A[i] + h_B[i] - h_C[i]) > 1e-5)
        {
            fprintf(stderr, "Result verification failed at element
%d!\n", i);
            exit(EXIT_FAILURE);
        }
    }

    printf("Test PASSED\n");

    // Free device global memory
    err = cudaFree(d_A);
```



```
if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to free device vector A (error code
%s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    err = cudaFree(d_B);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to free device vector B (error code
%s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    err = cudaFree(d_C);

if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to free device vector C (error code
%s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
```

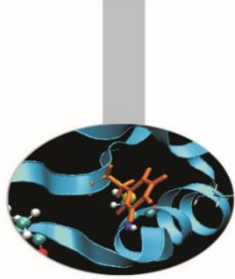
```
 // Free host memory
    free(h_A);
    free(h_B);
    free(h_C);

    // Reset the device and exit
    // cudaDeviceReset causes the driver to clean up all state.
While
    // not mandatory in normal operation, it is good practice.  It
is also
    // needed to ensure correct operation when the application
is being
    // profiled. Calling cudaDeviceReset causes all profile data
to be
    // flushed before the application exits
    err = cudaDeviceReset();

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to deinitialize the device!
error=%s\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    printf("Done\n");
    return 0;
}
```

OpenCL vector addition



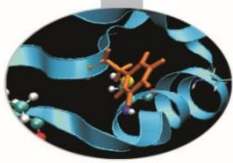
- For the OpenCL solution, there are two parts
  - Host code
  - Kernel code

Host code



- **By default, CUDA initializes the GPU automatically**
  - If you needed anything more complicated (multi-device etc.) you must do so manually
- **OpenCL always requires explicit device initialization**
  - **It runs not just on NVIDIA® GPUs and so you must tell it which device(s) to use**

CUDA to OpenCL terminology



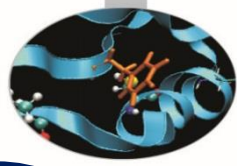
CUDA	OpenCL
GPU	Device (CPU, GPU etc)
Multiprocessor	Compute Unit, or CU
Scalar or CUDA core	Processing Element, or PE
Global or Device Memory	Global Memory
Shared Memory (per block)	Local Memory (per workgroup)
Local Memory (registers)	Private Memory
Thread Block	Work-group
Thread	Work-item
Warp	No equivalent term (yet)
Grid	NDRange

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 functions)
  5. Submit ***commands*** … transfer memory objects and execute kernels

Please, refer to he 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

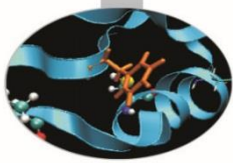


```
// Fill vectors a and b with random float values
   int i = 0;
   int count = LENGTH;
   for(i = 0; i < count; i++){
      h_a[i] = rand() / (float)RAND_MAX;
      h_b[i] = rand() / (float)RAND_MAX;
   }
```

```
// Set up platform and GPU device

cl_uint numPlatforms;

// Find number of platforms
err = clGetPlatformIDs(0, NULL, &numPlatforms);
checkError(err, "Finding platforms");
if (numPlatforms == 0)
{
    printf("Found 0 platforms!\n");
    return EXIT_FAILURE;
}
```

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

   // Secure a GPU
   for (i = 0; i < numPlatforms; i++)
   {
      err = clGetDeviceIDs(Platform[i], DEVICE, 1, &device_id,
NULL);
      if (err == CL_SUCCESS)
      {
         break;
      }
   }

   if (device_id == NULL)
      checkError(err, "Finding a device");

   err = output_device_info(device_id);
   checkError(err, "Printing device output");
```

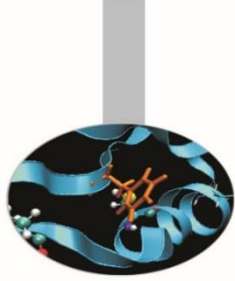
1. Define the platform



```
// Create a compute context
    context = clCreateContext(0, 1, &device_id, NULL, NULL,
&err);
    checkError(err, "Creating context");

    // Create a command queue
    commands = clCreateCommandQueue(context, device_id,
0, &err);
    checkError(err, "Creating command queue");

    // Create the compute program from the source buffer
    program = clCreateProgramWithSource(context, 1, (const
char **) & KernelSource, NULL, &err);
    checkError(err, "Creating program");

    // Build the program
    // Piero: added option

    char options[] = "-cl-mad-enable";
    err = clBuildProgram(program, 0, NULL, options, NULL,
NULL);
    if (err != CL_SUCCESS)
    {
        size_t len;
        char buffer[2048];
       printf("Error: Failed to build program executable!\n%s\n",
err_code(err));
        clGetProgramBuildInfo(program, device_id,
CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
        printf("%s\n", buffer);
        return EXIT_FAILURE;
    }
```

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

    // Create the input (a, b) and output (c) arrays in device
memory
    d_a  = clCreateBuffer(context,  CL_MEM_READ_ONLY,
sizeof(float) * count, NULL, &err);
    checkError(err, "Creating buffer d_a");

    d_b  = clCreateBuffer(context,  CL_MEM_READ_ONLY,
sizeof(float) * count, NULL, &err);
    checkError(err, "Creating buffer d_b");

    d_c  = clCreateBuffer(context,  CL_MEM_WRITE_ONLY,
sizeof(float) * count, NULL, &err);
    checkError(err, "Creating buffer d_c");

    // Write a and b vectors into compute device memory
    err = clEnqueueWriteBuffer(commands, d_a, CL_TRUE, 0,
sizeof(float) * count, h_a, 0, NULL, NULL);
    checkError(err, "Copying h_a to device at d_a");
```

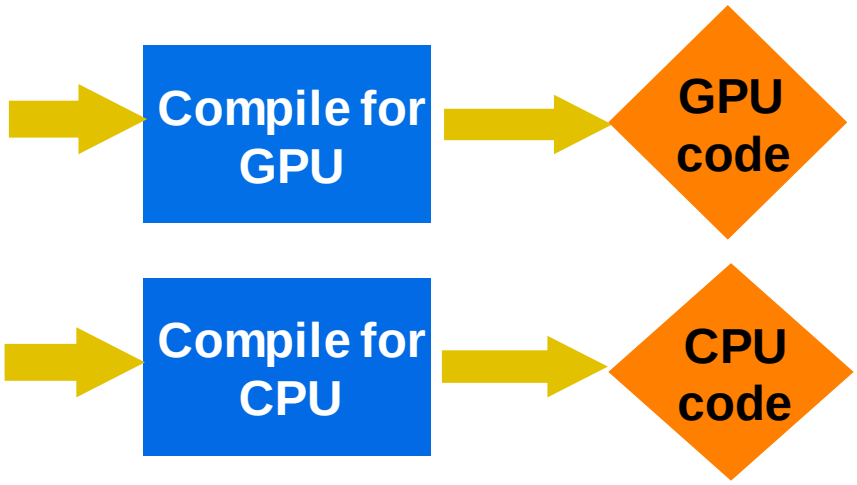
Building Program Objects

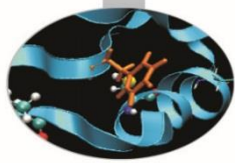


- The program object encapsulates:
  - A context
  - The program kernel source or binary
  - List of target devices and build options
- The C API build process to create a program object:
  - clCreateProgramWithSource()
  - clCreateProgramWithBinary()

OpenCL uses runtime compilation … because in general you don't know the details of the target device when you ship the program

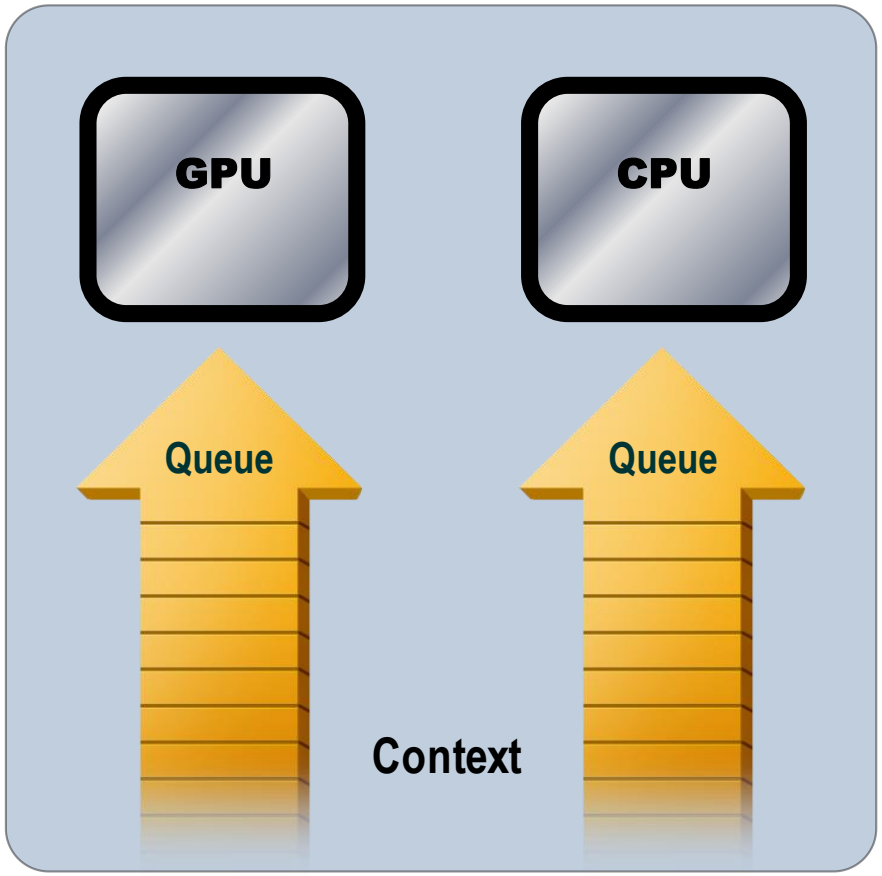
```
__kernel void
horizontal_reflect(read_only image2d_t src,
                   write_only image2d_t dst)
{
  int x = get_global_id(0);  // x-coord
  int y = get_global_id(1);  // y-coord
  int width = get_image_width(src);
  float4 src_val = read_imagef(src, sampler,
                   (int2)(width-1-x,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

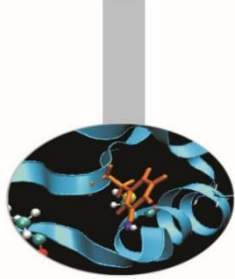




Command-Queues

- Commands include:
  - Kernel executions
  - Memory object management
  - Synchronization
- The only way to submit commands to a device is through a command-queue.
- Each command-queue points to a single device within a context.
- Multiple command-queues can feed a single device.
  - Used to define independent streams of commands that don't require synchronization

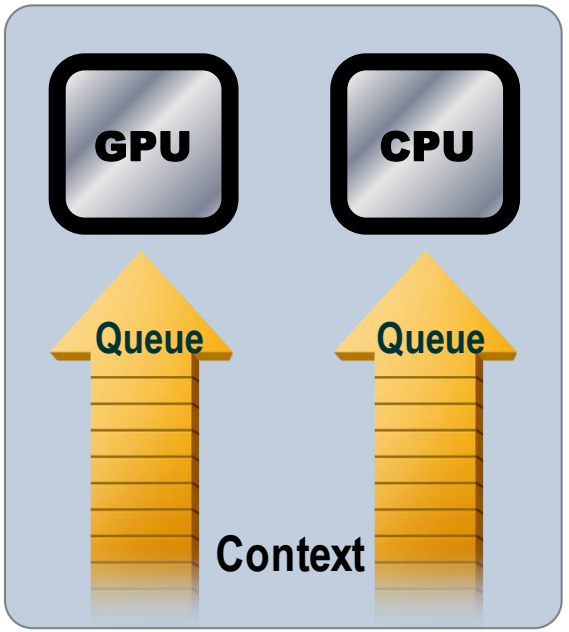




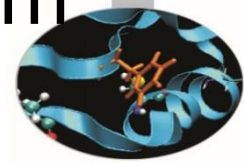
Command-Queue execution details

Command queues can be configured in different ways to control how commands execute

- ***In-order queues***:
  - Commands are enqueued and complete in the order they appear in the program (program-order)
- ***Out-of-order queues***:
  - Commands are enqueued in program-order but can execute (and hence complete) in any order.
- Execution of commands in the command-queue are guaranteed to be completed at synchronization points

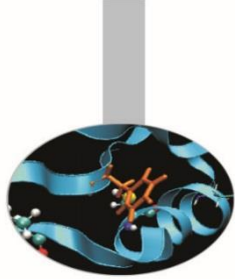


2. Create and Build the program



```
// Create a compute context
    context = clCreateContext(0, 1, &device_id, NULL, NULL,
&err);
    checkError(err, "Creating context");

    // Create a command queue
    commands = clCreateCommandQueue(context, device_id,
0, &err);
    checkError(err, "Creating command queue");

    // Create the compute program from the source buffer
    program = clCreateProgramWithSource(context, 1, (const
char **) & KernelSource, NULL, &err);
    checkError(err, "Creating program");

    // Build the program
    // Piero: added option

    char options[] = "-cl-mad-enable";
    err = clBuildProgram(program, 0, NULL, options, NULL,
NULL);
    if (err != CL_SUCCESS)
    {
        size_t len;
        char buffer[2048];
       printf("Error: Failed to build program executable!\n%s\n",
err_code(err));
        clGetProgramBuildInfo(program, device_id,
CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
        printf("%s\n", buffer);
        return EXIT_FAILURE;
    }
```

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

    // Create the input (a, b) and output (c) arrays in device
memory
    d_a  = clCreateBuffer(context,  CL_MEM_READ_ONLY,
sizeof(float) * count, NULL, &err);
    checkError(err, "Creating buffer d_a");

    d_b  = clCreateBuffer(context,  CL_MEM_READ_ONLY,
sizeof(float) * count, NULL, &err);
    checkError(err, "Creating buffer d_b");

    d_c  = clCreateBuffer(context,  CL_MEM_WRITE_ONLY,
sizeof(float) * count, NULL, &err);
    checkError(err, "Creating buffer d_c");

    // Write a and b vectors into compute device memory
    err = clEnqueueWriteBuffer(commands, d_a, CL_TRUE, 0,
sizeof(float) * count, h_a, 0, NULL, NULL);
    checkError(err, "Copying h_a to device at d_a");
```



Error messages

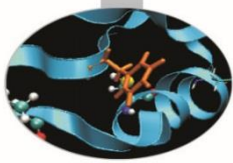
- Fetch and print error messages:

```
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);
}
```

- Important to do check all your OpenCL API error messages!

- Easier in C++ with try/catch

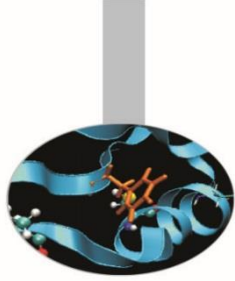
3. Setup Memory Objects



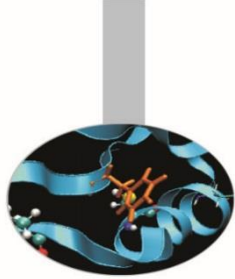
```
// Create a compute context
    context = clCreateContext(0, 1, &device_id, NULL, NULL,
&err);
    checkError(err, "Creating context");

    // Create a command queue
    commands = clCreateCommandQueue(context, device_id,
0, &err);
    checkError(err, "Creating command queue");

    // Create the compute program from the source buffer
    program = clCreateProgramWithSource(context, 1, (const
char **) & KernelSource, NULL, &err);
    checkError(err, "Creating program");

    // Build the program
    // Piero: added option

    char options[] = "-cl-mad-enable";
    err = clBuildProgram(program, 0, NULL, options, NULL,
NULL);
    if (err != CL_SUCCESS)
    {
        size_t len;
        char buffer[2048];
       printf("Error: Failed to build program executable!\n%s\n",
err_code(err));
        clGetProgramBuildInfo(program, device_id,
CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
        printf("%s\n", buffer);
        return EXIT_FAILURE;
    }
```

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

    // Create the input (a, b) and output (c) arrays in device
memory
    d_a  = clCreateBuffer(context,  CL_MEM_READ_ONLY,
sizeof(float) * count, NULL, &err);
    checkError(err, "Creating buffer d_a");

    d_b  = clCreateBuffer(context,  CL_MEM_READ_ONLY,
sizeof(float) * count, NULL, &err);
    checkError(err, "Creating buffer d_b");

    d_c  = clCreateBuffer(context,  CL_MEM_WRITE_ONLY,
sizeof(float) * count, NULL, &err);
    checkError(err, "Creating buffer d_c");

    // Write a and b vectors into compute device memory
    err = clEnqueueWriteBuffer(commands, d_a, CL_TRUE, 0,
sizeof(float) * count, h_a, 0, NULL, NULL);
    checkError(err, "Copying h_a to device at d_a");
```

Memory Objects



	CUDA C	OpenCL C
Allocate	float* d_x; cudaMalloc(&d_x, sizeof(float)*size);	cl_mem d_x = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float)*size, NULL, NULL);
Host to Device	cudaMemcpy(d_x, h_x, sizeof(float)*size, cudaMemcpyHostToDevice);	clEnqueueWriteBuffer(queue, d_x, CL_TRUE, 0, sizeof(float)*size, h_x, 0, NULL, NULL);
Device to Host	cudaMemcpy(h_x, d_x, sizeof(float)*size, cudaMemcpyDeviceToHost);	clEnqueueReadBuffer(queue, d_x, CL_TRUE, 0, sizeof(float)*size, h_x, 0, NULL, NULL);



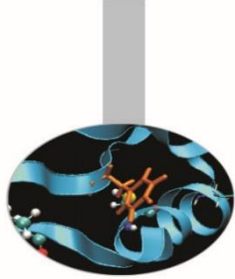
Memory Objects

- Buffers are declared on the host as type: cl_mem
- Arrays in host memory hold your original host-side data:

```
float h_a[LENGTH], h_b[LENGTH];
```

- Create the buffer (d_a), assign sizeof(float)*count bytes from "h_a" to the buffer and copy it into device memory:

```
cl_mem d_a = clCreateBuffer(context,
 CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  sizeof(float)*count, h_a, NULL);
```



Memory Objects

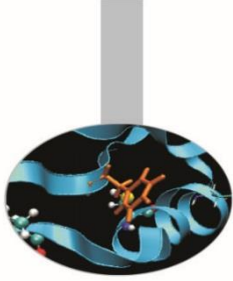
- Other common memory flags include:
  CL_MEM_WRITE_ONLY, CL_MEM_READ_WRITE

- These are from the point of view of the device

- Submit command to copy the buffer back to host memory at "h_c":
  - CL_TRUE = blocking, CL_FALSE = non-blocking

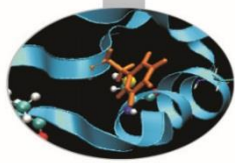
```
clEnqueueReadBuffer(queue, d_c, CL_TRUE,
                    sizeof(float)*count, h_c,
                    NULL, NULL, NULL);
```

4. Define the kernel



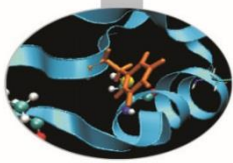
```
// Create the compute kernel from the program
    ko_vadd = clCreateKernel(program, "vadd", &err);
    checkError(err, "Creating kernel");

d_b  = clCreateBuffer(context,  CL_MEM_READ_ONLY,
sizeof(float) * count, NULL, &err);
    checkError(err, "Creating buffer d_b");

    d_c  = clCreateBuffer(context,  CL_MEM_WRITE_ONLY,
sizeof(float) * count, NULL, &err);
    checkError(err, "Creating buffer d_c");

    // Write a and b vectors into compute device memory
    err = clEnqueueWriteBuffer(commands, d_a, CL_TRUE, 0,
sizeof(float) * count, h_a, 0, NULL, NULL);
    checkError(err, "Copying h_a to device at d_a");

    err = clEnqueueWriteBuffer(commands, d_b, CL_TRUE, 0,
sizeof(float) * count, h_b, 0, NULL, NULL);
    checkError(err, "Copying h_b to device at d_b");

    // Set the arguments to our compute kernel
    err  = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_a);
    err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_b);
    err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_c);
    err |= clSetKernelArg(ko_vadd, 3, sizeof(unsigned int),
&count);
checkError(err, "Setting kernel arguments");

double rtime = wtime();

    // Execute the kernel over the entire range of our 1d input
data set
    // letting the OpenCL runtime choose the work-group size
    global = count;
    err = clEnqueueNDRangeKernel(commands, ko_vadd, 1,
NULL,  &global, NULL, 0, NULL,  NULL);
    checkError(err, "Enqueueing kernel");

    // Wait for the commands to complete before stopping the
timer
    err = clFinish(commands);
    checkError(err, "Waiting for kernel to finish");

    rtime = wtime() - rtime;
    printf("\nThe kernel ran in %lf seconds\n",rtime);

    // Read back the results from the compute device
    err = clEnqueueReadBuffer( commands, d_c, CL_TRUE, 0,
sizeof(float) * count, h_c, 0, NULL, NULL  );
    if (err != CL_SUCCESS)
    {
        printf("Error: Failed to read output array!\n%s\n",
err_code(err));
        exit(1);
}
```

Dividing up the work



- To enqueue the kernel
  - CUDA – specify the number of thread blocks and threads per block
  - OpenCL – specify the problem size and (optionally) number of work-items per work-group

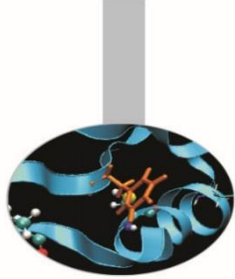
5. Enqueue Commands



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

d_b  = clCreateBuffer(context,  CL_MEM_READ_ONLY,
sizeof(float) * count, NULL, &err);
    checkError(err, "Creating buffer d_b");

    d_c  = clCreateBuffer(context,  CL_MEM_WRITE_ONLY,
sizeof(float) * count, NULL, &err);
    checkError(err, "Creating buffer d_c");

    // Write a and b vectors into compute device memory
    err = clEnqueueWriteBuffer(commands, d_a, CL_TRUE, 0,
sizeof(float) * count, h_a, 0, NULL, NULL);
    checkError(err, "Copying h_a to device at d_a");

    err = clEnqueueWriteBuffer(commands, d_b, CL_TRUE, 0,
sizeof(float) * count, h_b, 0, NULL, NULL);
    checkError(err, "Copying h_b to device at d_b");

    // Set the arguments to our compute kernel
    err  = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_a);
    err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_b);
    err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_c);
    err |= clSetKernelArg(ko_vadd, 3, sizeof(unsigned int),
&count);
checkError(err, "Setting kernel arguments");
```

```
double rtime = wtime();

    // Execute the kernel over the entire range of our 1d input
data set
    // letting the OpenCL runtime choose the work-group size
    global = count;
    err = clEnqueueNDRangeKernel(commands, ko_vadd, 1,
NULL,  &global, NULL, 0, NULL,  NULL);
    checkError(err, "Enqueueing kernel");

    // Wait for the commands to complete before stopping the
timer
    err = clFinish(commands);
    checkError(err, "Waiting for kernel to finish");

    rtime = wtime() - rtime;
    printf("\nThe kernel ran in %lf seconds\n",rtime);

    // Read back the results from the compute device
    err = clEnqueueReadBuffer( commands, d_c, CL_TRUE, 0,
sizeof(float) * count, h_c, 0, NULL, NULL  );
    if (err != CL_SUCCESS)
    {
        printf("Error: Failed to read output array!\n%s\n",
err_code(err));
        exit(1);
}
```

Enqueue a kernel (C)



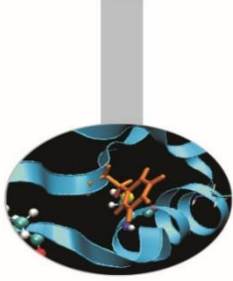
CUDA C

```
dim3 threads_per_block(30,20);

dim3 num_blocks(10,10);

kernel<<<num_blocks,
        threads_per_block>>>();
```

OpenCL C

```
const size_t global[2] =
        {300, 200};

const size_t local[2] =
        {30, 20};

clEnqueueNDRangeKernel(
    queue, &kernel,
    2, 0, &global, &local,
    0, NULL, NULL);
```

Vector Addition – Host Program



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

// get the list of GPU devices associated with context
clGet                                                  cb);
cl_device_id[] devices = malloc(cb);
clGetContextInfo(context,CL_CONTEXT_DEVICES,cb,devices,NULL);

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

// allocate the buffer memory objects
memobj
    CL_I                                              JLL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcb, NULL);

memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                  sizeof(cl_float)*n, NULL, NULL);

// create the
program = c
                  &program_source, NULL, NULL);
```

Define platform and queues

Define memory objects

Create the program

```
// build the prog
err = clBuildPro                                       LL,NULL);

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

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

// execute ker
err = clEnque                                          el, 1, NULL,
                  global_work_size, NULL,0,NULL,NULL);

// read output array
err = clEn
                  n*sizeof(cl_float), dst,
                  0, NULL, NULL);
```

Build the program

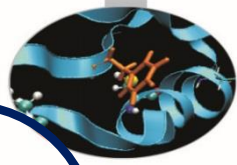
Create and setup kernel

Execute the kernel

Read results on the host

It's complicated, but most of this is "boilerplate" and not as bad as it looks.

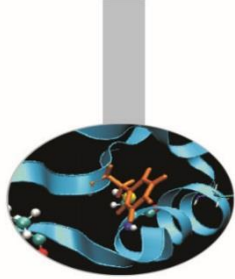
Vector Addition – CUDA Kernel



```
/**
 * Copyright 1993-2015 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

/**
 * Vector addition: C = A + B.
 *
 * This sample is a very basic sample that implements element by element
 * vector addition. It is the same as the sample illustrating Chapter 2
 * of the programming guide with some additions like error checking.
 */

#include <stdio.h>

// For the CUDA runtime routines (prefixed with "cuda_")
#include <cuda_runtime.h>
```

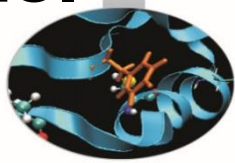
```
/**
 * CUDA Kernel Device code
 *
 * Computes the vector addition of A and B into C. The 3 vectors have the same
 * number of elements numElements.
 */
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}

/**
 * Host main routine
 */
```

Indexing at work



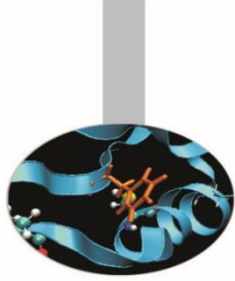
CUDA C	OpenCL
gridDim	get_num_groups()
blockIdx	get_group_id()
blockDim	get_local_size()
gridDim * blockDim	get_global_size()
threadIdx	get_local_id()
blockIdx * blockdim + threadIdx	get_global_id()

Vector Addition – OpenCL Kernel



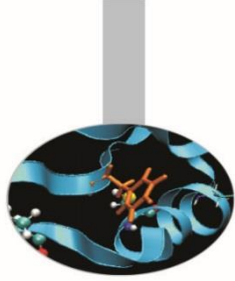
```
const char *KernelSource = "\n" \
"#pragma OPENCL EXTENSION  cl_nv_compiler_options :
enable          \n" \
"__kernel void vadd(                                    \n" \
"   __global float* a,                                  \n" \
"   __global float* b,                                  \n" \
"   __global float* c,                                  \n" \
"   const unsigned int count)                               \n" \
"{                                                  \n" \
"   int i = get_global_id(0);                               \n" \
"   if(i < count)                                   \n" \
"       c[i] = a[i] + b[i];                             \n" \
"}                                                  \n" \
"\n";

//------------------------------------------------------------------------
```

OpenCL C Language Highlights



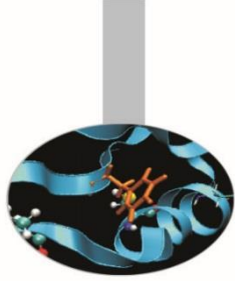
- Function qualifiers
  - **__kernel** qualifier declares a function as a kernel
    - I.e. makes it visible to host code so it can be enqueued
  - Kernels can call other kernel-side functions
- Address space qualifiers
  - **__global, __local, __constant, __private**
  - Pointer kernel arguments must be declared with an address space qualifier
- Work-item functions
  - get_work_dim(), get_global_id(), get_local_id(), get_group_id()
- Synchronization functions
  - **Barriers** - all work-items within a work-group must execute the barrier function before any work-item can continue
  - **Memory fences** - provides ordering between memory operations

Differences in kernels



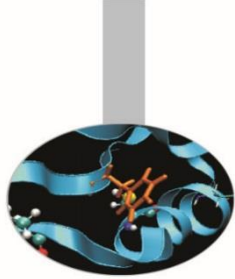
- Where do you find the kernel?
  - OpenCL - either a string (const char *), or read from a file
  - CUDA – a function in the host code
- Denoting a kernel
  - OpenCL - __kernel
  - CUDA - __global__
- When are my kernels compiled?
  - OpenCL – at runtime
  - CUDA – with compilation of host code

Run OpenCL



- Goal:
  - Use DeviceInfo and vectorAdd directory
- Procedure:
  - Enter in each of them
  - Run make
  - Run the executables
- Expected output:
  - A message to standard output for both executables

Run OpenCL-2



- DeviceInfo output

Number of devices: 2

```
-----------------------
    Name: Tesla K40m
    Version: OpenCL C 1.2
    Max. Compute Units: 15
    Local Memory Size: 48 KB
    Global Memory Size: 11519 MB
    Max Alloc Size: 2879 MB
    Max Work-group Total Size: 1024
    Max Work-group Dims: ( 1024 1024 64 )
-----------------------
-----------------------
    Name: Tesla K40m
    Version: OpenCL C 1.2
    Max. Compute Units: 15
    Local Memory Size: 48 KB
    Global Memory Size: 11519 MB
    Max Alloc Size: 2879 MB
    Max Work-group Total Size: 1024
    Max Work-group Dims: ( 1024 1024 64 )
```

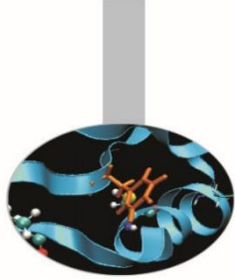
- vectorAdd output

[planucar@node166 C]$ ./vadd

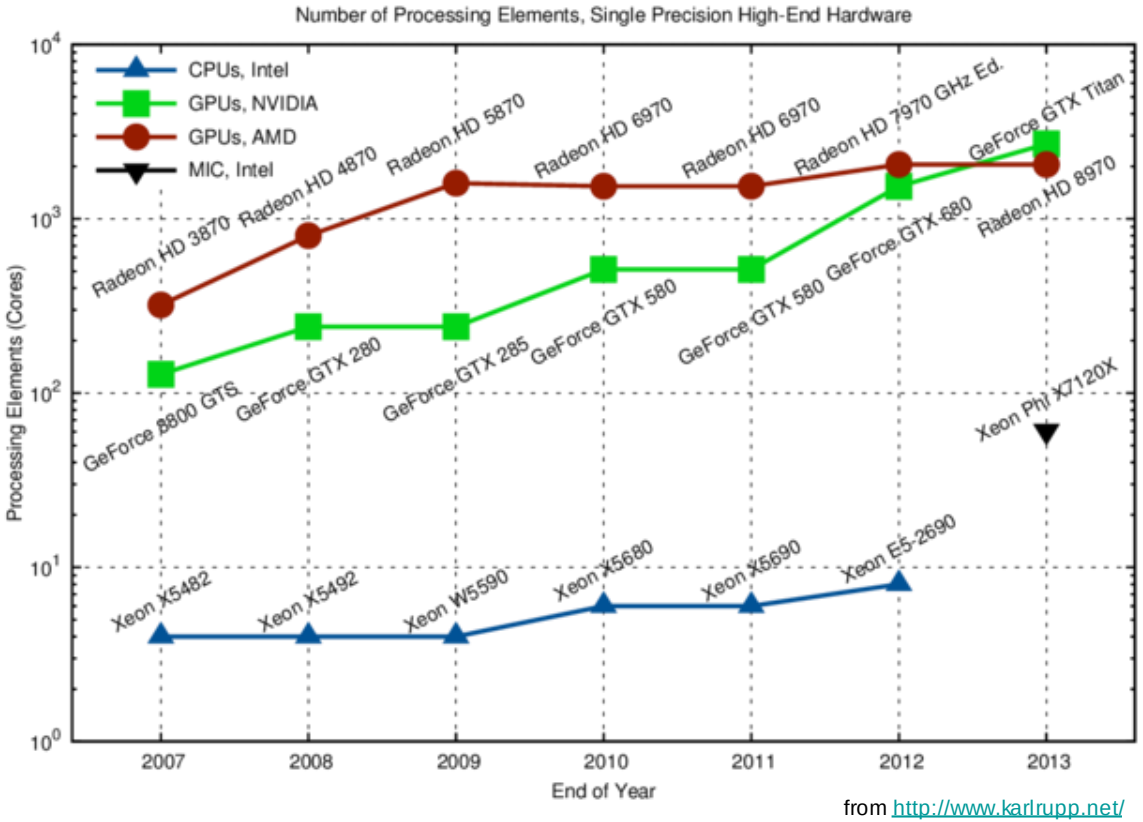
Device is Tesla K40m GPU from NVIDIA Corporation with a max of 15 compute units

The kernel ran in 0.000061 seconds

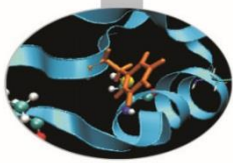
C = A+B: 8192 out of 8192 results were correct.

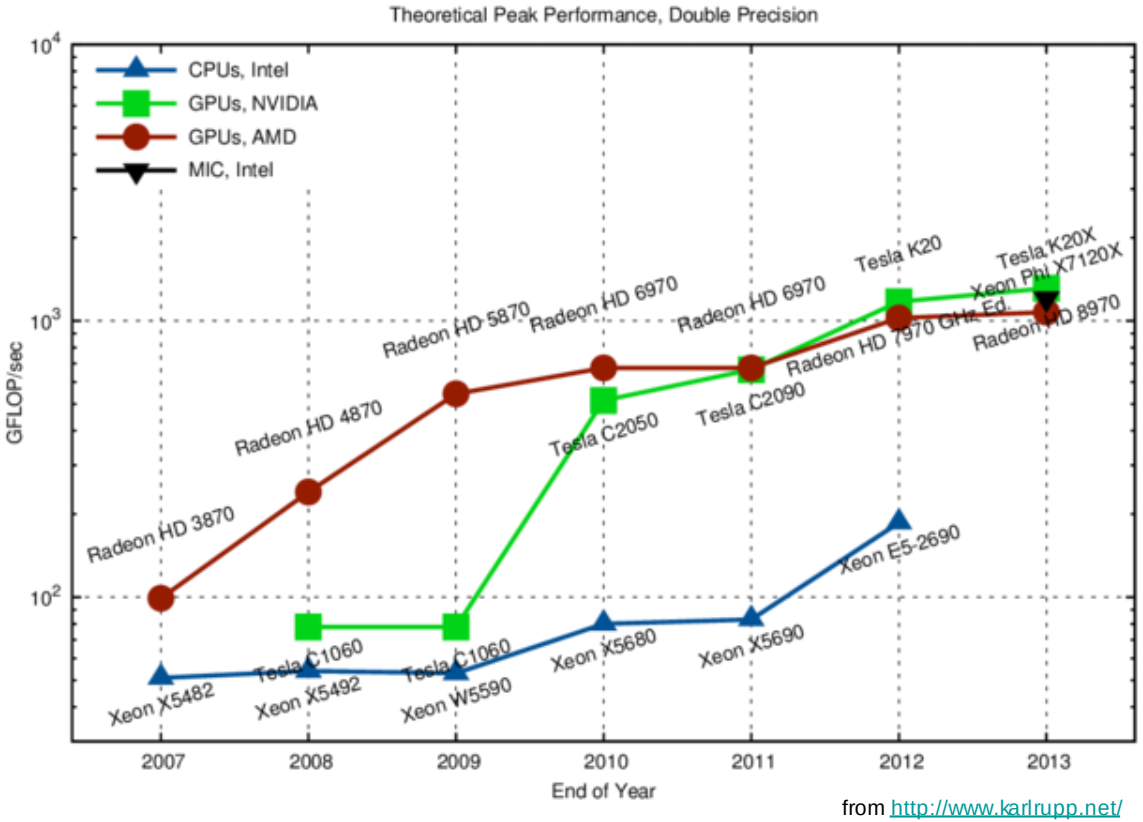


OpenCL and portability

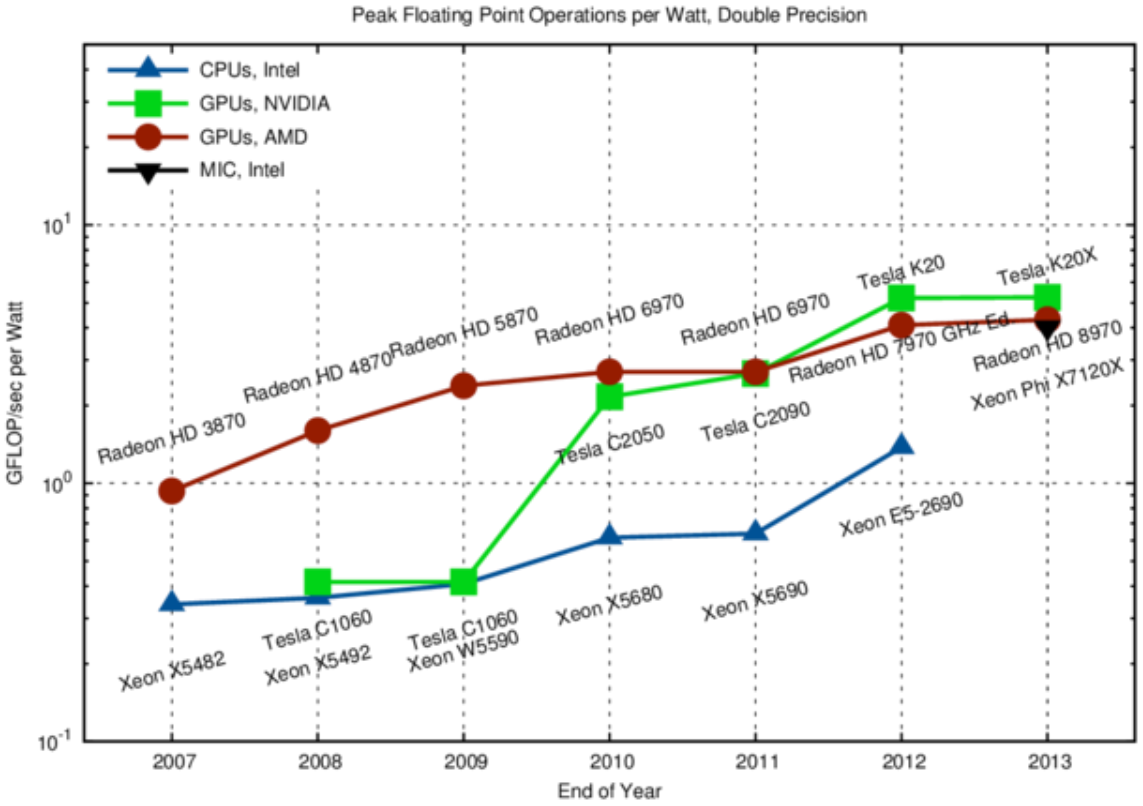


from http://www.karlrupp.net/





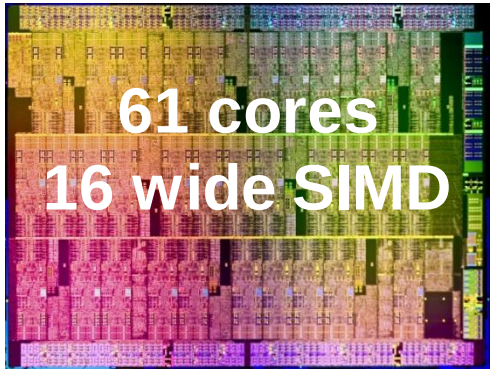
from http://www.karlrupp.net/



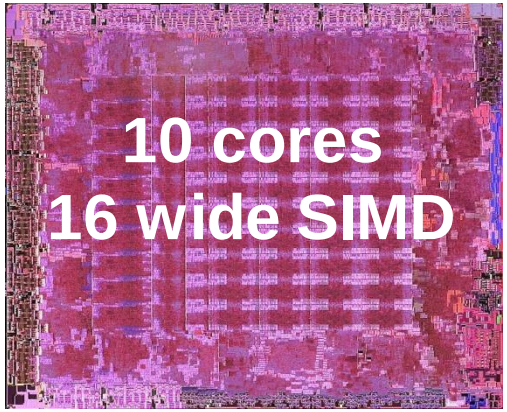
from http://www.karlrupp.net/

Microprocessor trends

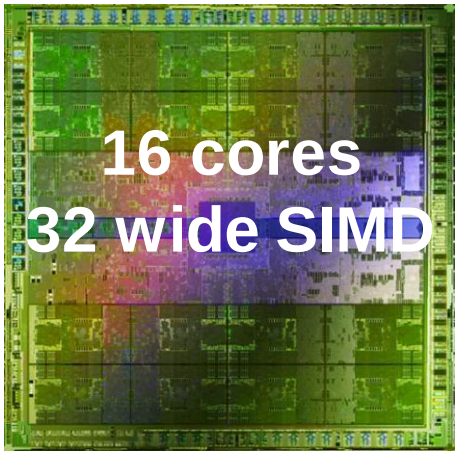
Individual processors have many (possibly heterogeneous) cores.



Intel® Xeon Phi™ coprocessor



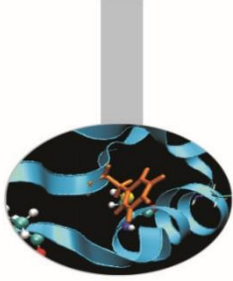
ATI™ RV770



NVIDIA® Tesla® C2090

The (Heterogeneous) many-core challenge:

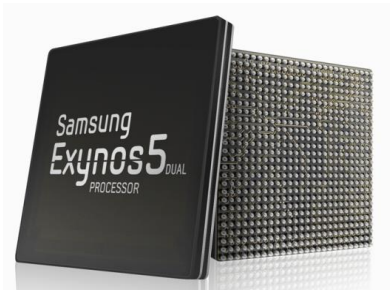
How are we to build a software ecosystem for the Heterogeneous many core platform?



Heterogeneous High Performance Programming framework

A modern computing platform includes:

- One or more CPUs
- One of more GPUs
- DSP processors
- Accelerators
- … other?

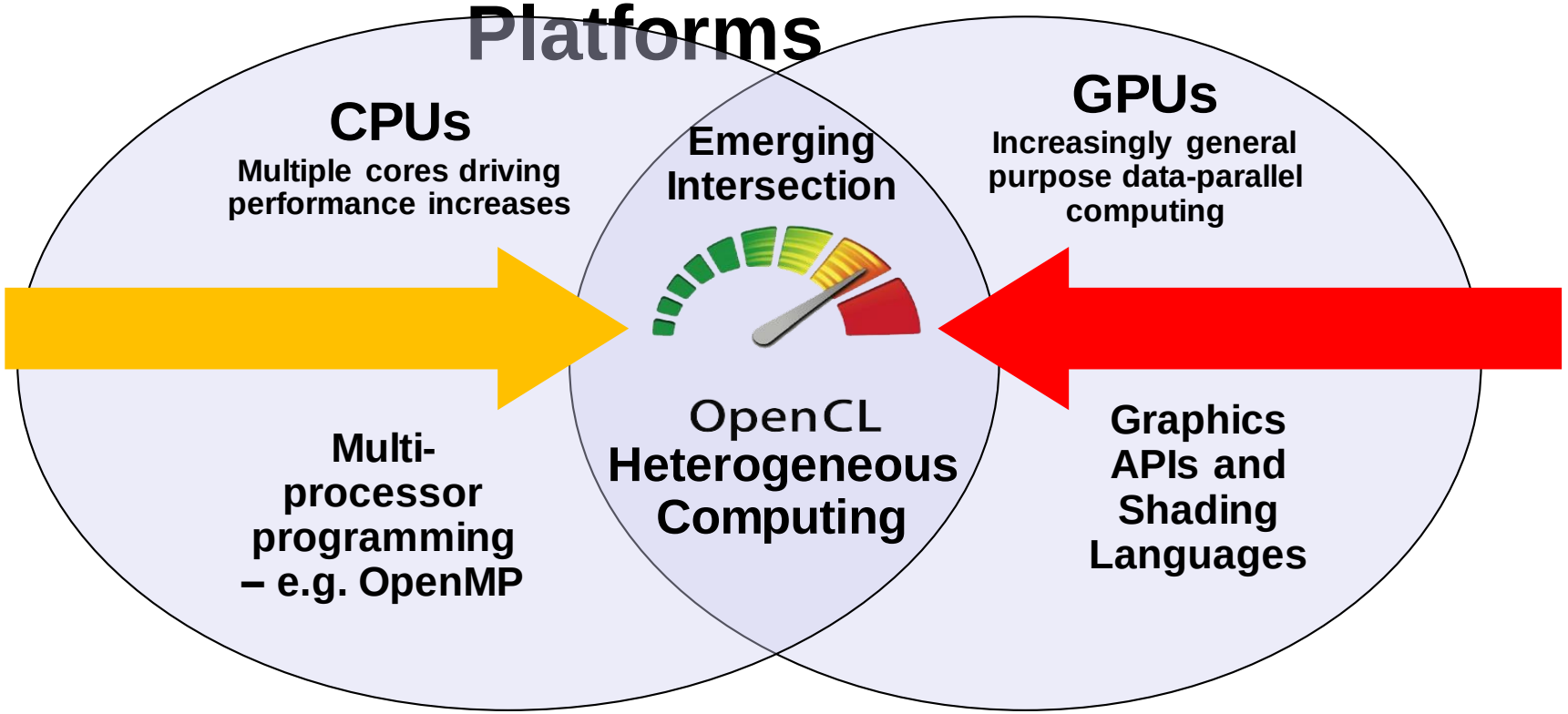


E.g. Samsung® Exynos 5:

- Dual core ARM A15 1.7GHz, Mali T604 GPU

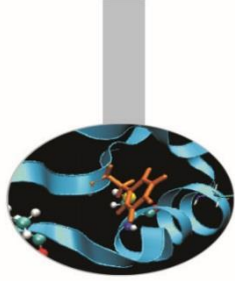
OpenCL lets Programmers write a single <u>portable</u> program that uses <u>ALL</u> resources in the heterogeneous platform

Industry Standards for Programming Heterogeneous Platforms



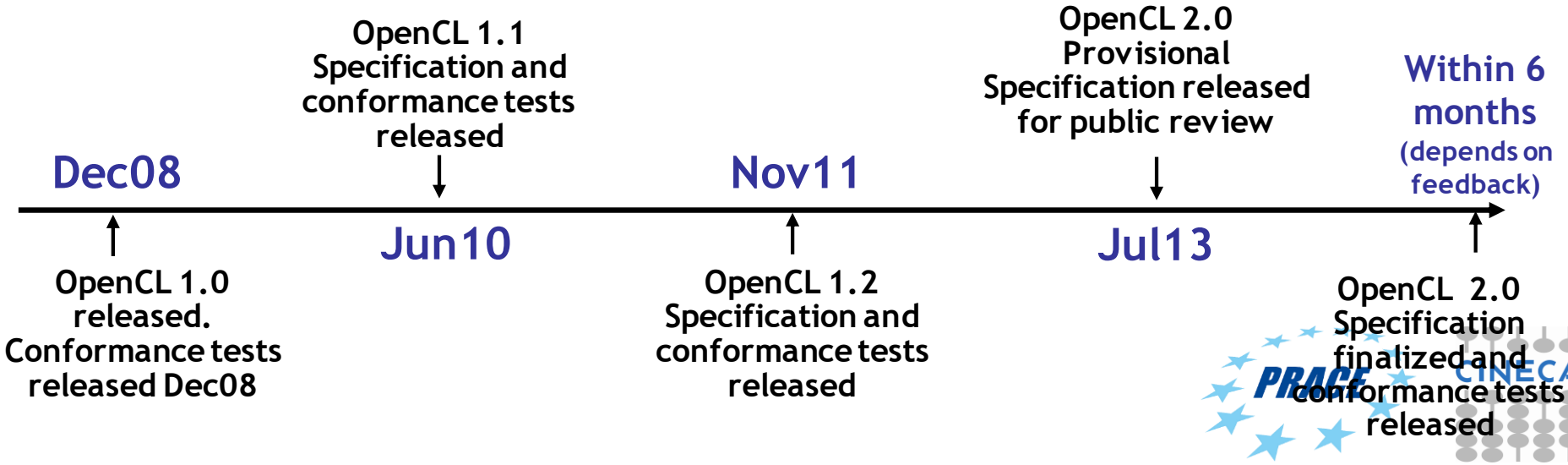
OpenCL – Open Computing Language

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



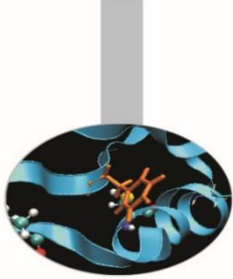
OpenCL Timeline

- Launched Jun'08 … 6 months from "strawman" to OpenCL 1.0
- Rapid innovation to match pace of hardware innovation
  - 18 months from 1.0 to 1.1 and from 1.1 to 1.2
  - Goal: a new OpenCL every 18-24 months
  - Committed to backwards compatibility to protect software investments



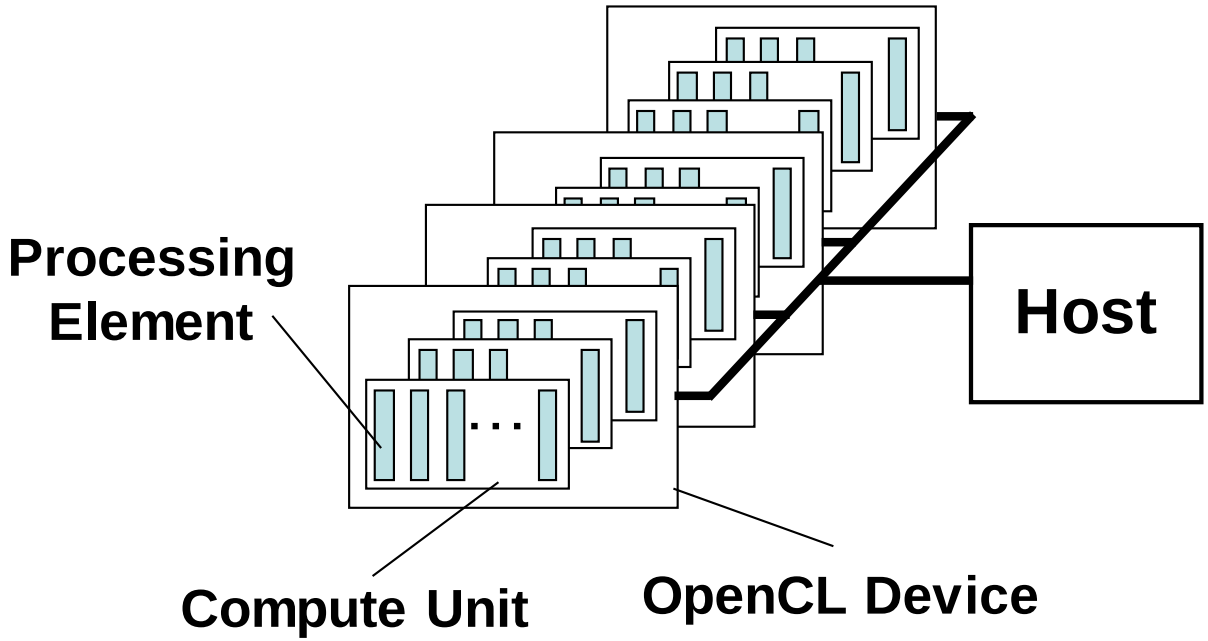
OpenCL Working Group within Khronos

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



3DLABS SEMICONDUCTOR · ACTIVISION BLIZZARD · AMD · ARM · BROADCOM · EA · codeplay · ERICSSON · freescale semiconductor · HI CORP. · IBM · intel · Imagination TECHNOLOGIES · Los Alamos NATIONAL LABORATORY · MOTOROLA · movidia · NOKIA · NVIDIA · QNX SOFTWARE SYSTEMS · RAPIDMIND · SAMSUNG · Seaweed SYSTEMS · TAKUMI · TEXAS INSTRUMENTS · PRACE · CINECA · KHRONOS GROUP

OpenCL abstract Platform Model

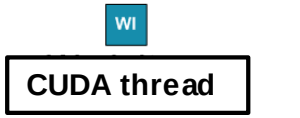


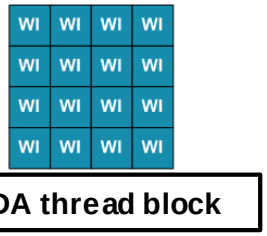
- One ***Host*** and one or more ***OpenCL Devices***
  - Each OpenCL Device is composed of one or more ***Compute Units***
    - Each Compute Unit is divided into one or more ***Processing Elements***
- Memory divided into ***host memory*** and ***device memory***

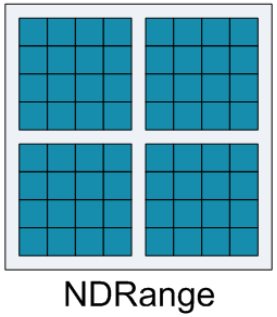
OpenCL NVIDIA Platform Model

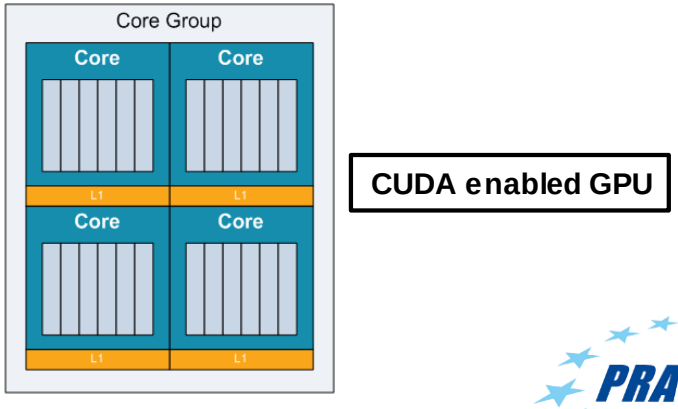
E.g. NVIDIA® K80:

- Dual NVIDIA K40 GPU

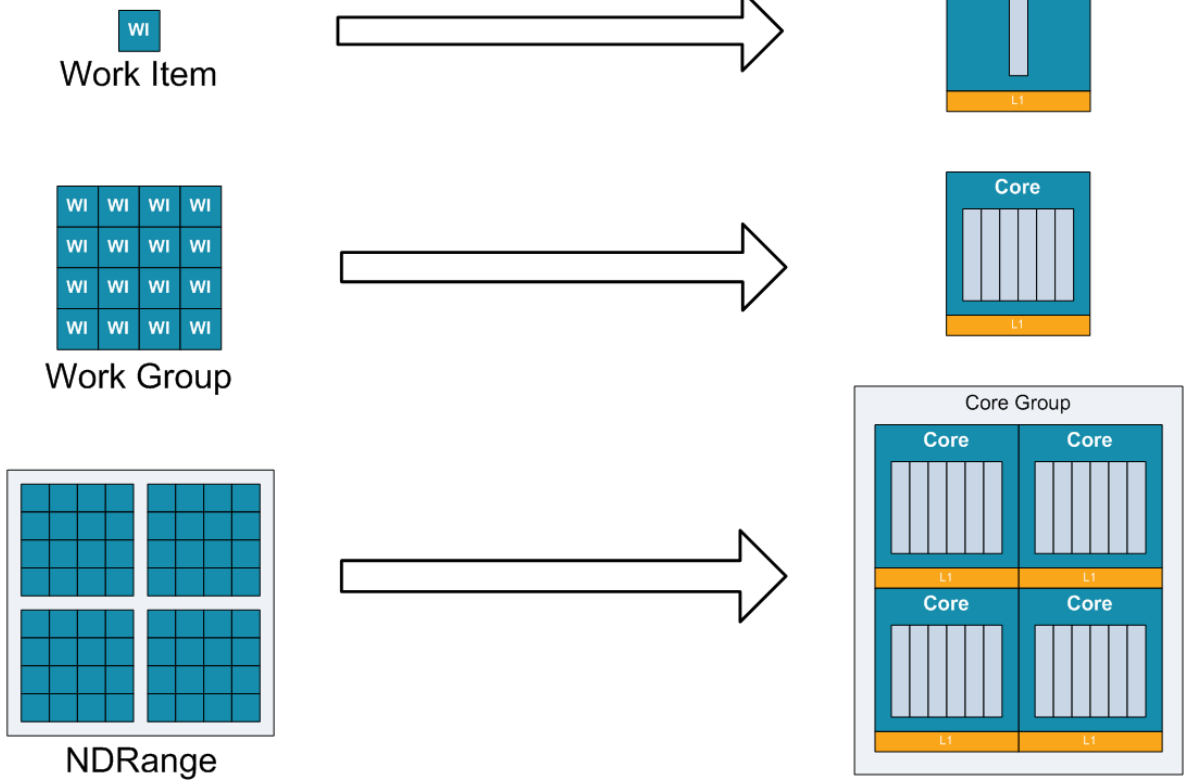






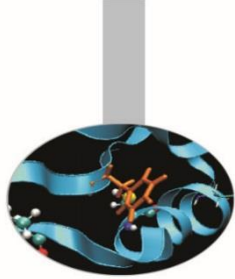


OpenCL NVIDIA Platform Model



- Each WI runs as one of the thread within a CUDA SP
- A whole WG executes on a single SMP
- Several WG can reside on a single SMP (depending on WG memory and SMP resources
- Each kernel is executed on a CUDA device

Run OpenCL on NVIDIA (K80)



- DeviceInfo output

Number of OpenCL platforms: 1

Platform: NVIDIA CUDA
Vendor: NVIDIA Corporation
Version: OpenCL 1.2 CUDA 7.5.23
Number of devices: 1

Name: Tesla K80
Version: OpenCL C 1.2
Max. Compute Units: 13
Local Memory Size: 48 KB
Global Memory Size: 11519 MB
Max Alloc Size: 2879 MB
Max Work-group Total Size: 1024
Max Work-group Dims: (1024 1024 64)

- vectorAdd output

[planucar@node495 C]$ ./vadd

Device is Tesla K80 GPU from NVIDIA Corporation with a max of 13 compute units

The kernel ran in 0.000052 seconds
C = A+B: 8192 out of 8192 results were correct.

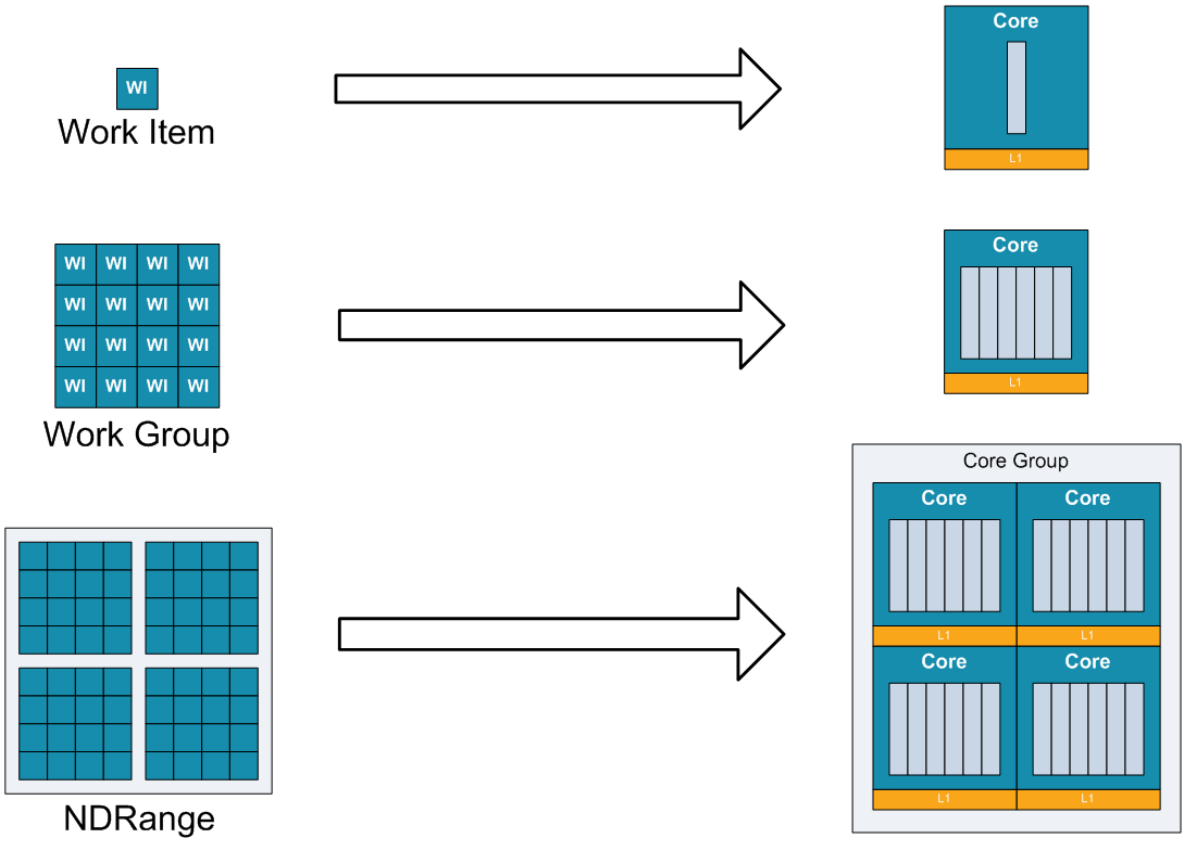
The MontBlanc proto

- 2 racks, 8 standard BullX chassis, 72 compute blades fitting 1080 compute cards, for a total of 2160 CPUs and 1080 GPUs.
- SoC Samsung Exynos 5 Dual CPU Cortex-A15@1.7Ghz dual core.
- GPU ARM Mali T604 (OpenCL 1.1 capable).

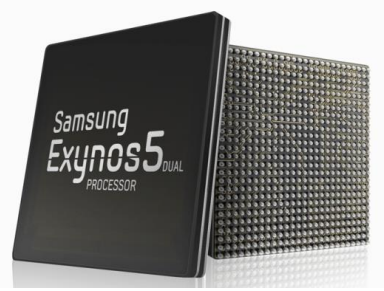
OpenCL Mali Platform Model

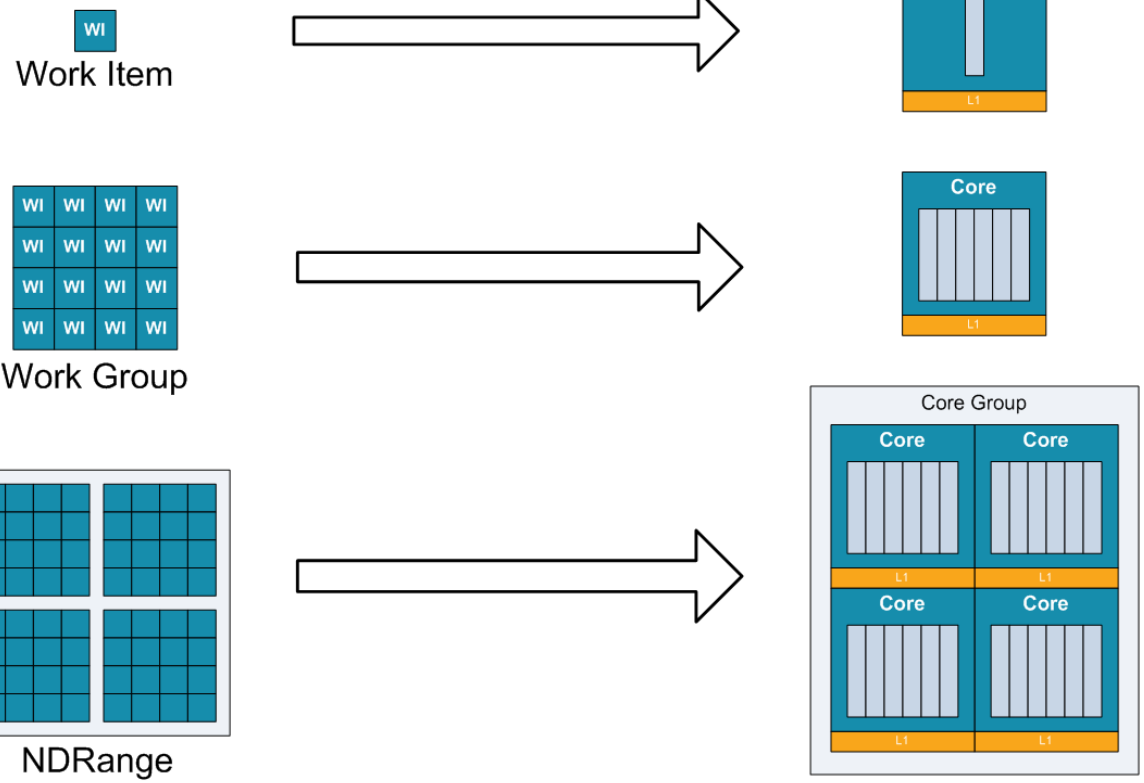
E.g. Samsung® Exynos 5:

- Dual core ARM A15 1.7GHz, Mali T604 GPU



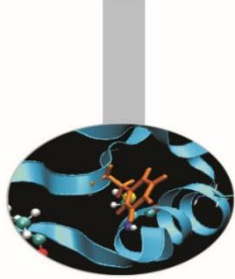
OpenCL Mali Platform Model





- Each WI runs as one of the thread within a core
- Up to 256 threads(WI) per core
- A whole WG executes on a single core
- “Adiacent” WG are scheduled onto core in a round-robin fashion

Run OpenCL on Mali GPU



- DeviceInfo output

Number of OpenCL platforms: 1

Platform: ARM Platform
Vendor: ARM
Version: OpenCL 1.1
Number of devices: 1

Name: Mali-T604
Version: OpenCL C 1.1
Max. Compute Units: 4
Local Memory Size: 32 KB
Global Memory Size: 3527 MB
Max Alloc Size: 881 MB
Max Work-group Total Size: 256
Max Work-group Dims: (256 256 256)

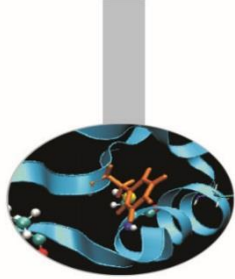
- vectorAdd output

planucar@mb-login-1:~/OpenCL2016/Exercises-Solutions/Exercises/Exercise02/C$ ./vadd

Device is Mali-T604 GPU from ARM with a max of 4 compute units

The kernel ran in 0.000346 seconds
C = A+B: 1024 out of 1024 results were correct.

OpenCL runtime: how it works?



- A simple mechanism is used to address multiple separate vendor drivers: **ICD loader** (ICD stands for Installable Client Drivers)
- At every OpenCL function call the ICD loader infers the vendor ICD function to call from the arguments to the function.
- The structure *_cl_icd_dispatch* is a **function pointer dispatch table** to direct calls to a particular vendor implementation (**ICD library**).

- ICD compatible object has the following structure:

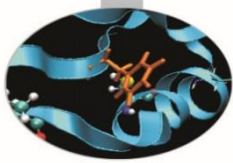
```
Struct _cl_<object>
{
  struct _cl_icd_dispatch *dispatch:
// ….remainder of internal data
};
```

<object> can be platform_id, devide_id, context, etc

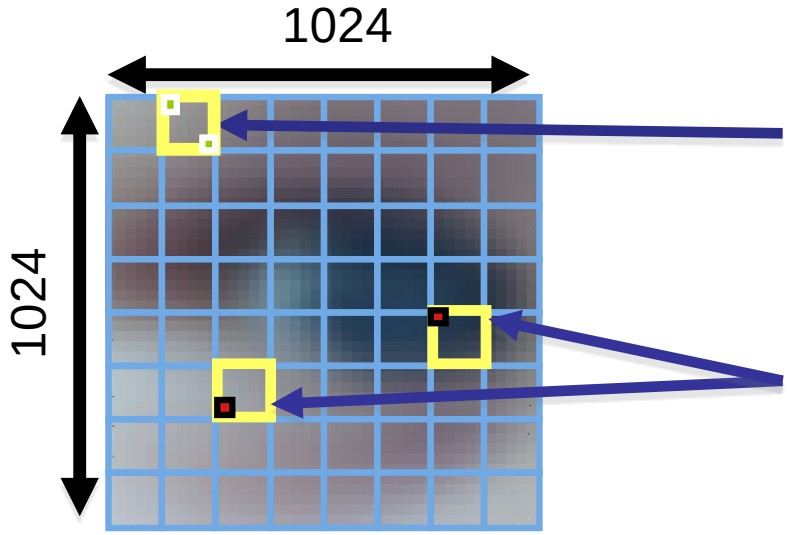
Example: **NVIDIA ICD loader**.

- Search for ***/etc/OpenCL/vendors***
- **ICD loader** opens the file containing the vendor ICD library (shared object). In this case the file **nvidia.icd** text line: **libnvidia-opencl.so.1**

An N-dimensional domain of work-items



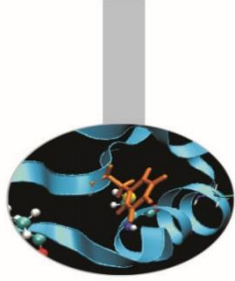
- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 128x128 (**work-group**, executes together)



Synchronization between work-items possible only within work-groups: and

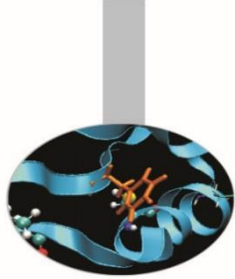
Cannot synchronize between work-groups within a kernel

- Choose the dimensions that are “best” for your algorithm (and hardware)



OpenCL N Dimensional Range (NDRange)

- The problem we want to compute should have some **dimensionality**;
  - For example, compute a kernel on all points in a cube
- When we execute the kernel we specify **up to 3 dimensions**
- We also **specify the total problem size** in each dimension – this is called the **global** size
- We associate each point in the iteration space with a **work-item**



OpenCL N Dimensional Range (NDRange)

- Work-items are grouped into **work-groups**; work-items within a work-group can share **local memory** and can **synchronize**
- We can specify the number of work-items in a work-group – this is called the **local** (work-group) size
- Or the OpenCL run-time can choose the work-group size for you (usually not optimally)

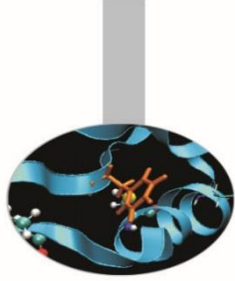
CINECA SCAI
SuperComputing Applications and Innovation

OpenCL Matrix Multiply live@CINECA

PRACE

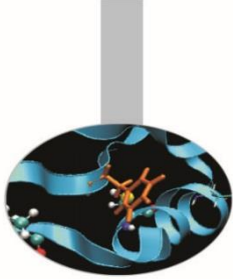
CINECA

How to extract working dirs



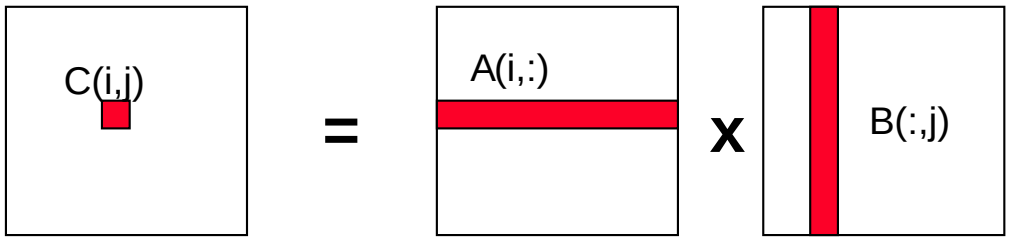
- wget from standard **hpcforge.cineca.it** site the file: ***OpenCL_Exercise_Solutions.tgz***
- *tar xvfz* ***OpenCL_Exercise_Solutions.tgz***
- ***OpenCL2017*** *directory is created*
- ***….that's all !***

Matrix multiplication: sequential code



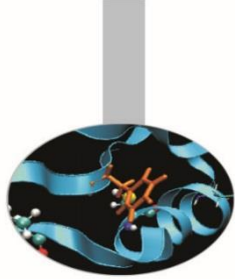
We compute C=AB, where all three matrices are NxN

```
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: sequential code



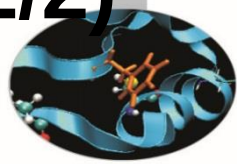
```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
              // C(i, j) = sum(over k) A(i,k) * B(k,j)
              C[i*N+j] += A[i*N+k] * B[k*N+j];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

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, j) = sum(over k) A(i,k) * B(k,j)
        for (k = 0; k < N; k++) {
          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)



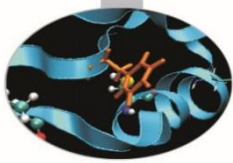
```
__kernel void mat_mul(
  const int N,
  __global float *A, __global float *B, __global float *C)
{
    int i, j, k;
    i = get_global_id(0);
    j = get_global_id(1);
        for (k = 0; k < N; k++) {
            // C(i, j) = sum(over k) A(i,k) * B(k,j)
            C[i*N+j] += A[i*N+k] * B[k*N+j];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

    }
}
```

Remove outer loops and set work-item co-ordinates

Matrix multiplication: OpenCL kernel

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

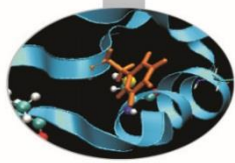
Matrix multiplication: OpenCL kernel improved



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

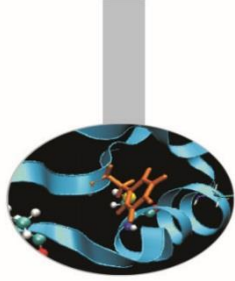
```
__kernel void mmul(
    const int N,
    __global float *A,
    __global float *B,
    __global float *C)
{
  int k;
  int i = get_global_id(0);
  int j = get_global_id(1);
 if ( (i < N) && (j < N) )
{
  float tmp = 0.0f;
  for (k = 0; k < N; k++)
   tmp += A[i*N+k]*B[k*N+j];
  }
  C[i*N+j] += tmp;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

Exercise 2: run serial and first matMul OpenCL



- Goal:
  - Use ***basic*** directory
- Procedure:
  - Enter
  - Run make
  - Run the executable
- Expected output:
  - A message to standard output for serial and first matMul execution

Matrix multiplication performance (Galileo compute node, N=2048)

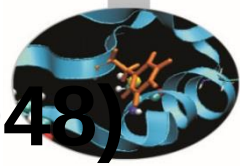


- **Matrices are stored in global memory.**

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)		N/A
C(i,j) per work-item, all global	N/A	

GPU Device is Kepler® K80 GPU from NVIDIA® with a max of 13 compute units, 512 PEs
CPU Device is Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz

Matrix multiplication performance (MontBlanc proto compute node, N=2048)

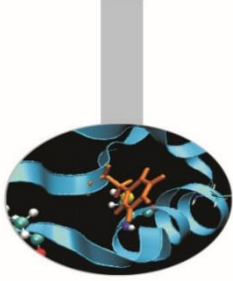


- **Matrices are stored in global memory.**

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	37.9	N/A
C(i,j) per work-item, all global	N/A	220.7

GPU Device is Mali® T604 GPU with a max of 4 compute units
CPU Device is ARM(R) A15 @ 1.7GHz

Matrix multiplication performance (Galileo compute node, N=2048)

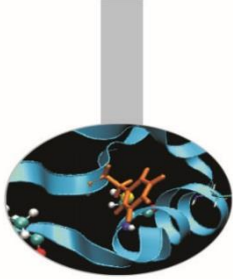


- **Matrices are stored in global memory.**

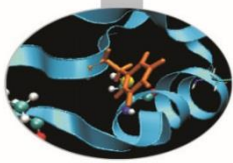
Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	725	N/A
C(i,j) per work-item, all global	N/A	8835

Far from optimal(CUDA)

GPU Device is Kepler® K80 GPU from NVIDIA® with a max of 13 compute units, 2496 PEs
CPU Device is Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz



OpenCL and (performance?) Portability



Portable performance in OpenCL

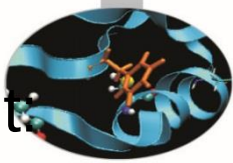
- Portable performance is always a challenge, more so when OpenCL devices can be so varied (CPUs, GPUs, …)
- But OpenCL provides a powerful framework for writing performance portable code
- The target is writing code that should work well on most OpenCL devices

Tremendous amount of computing power available

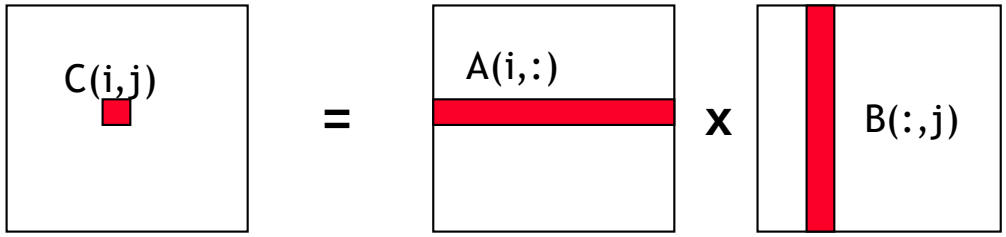
1170 GFLOPs peak

1070 GFLOPs peak

Optimizing matrix multiplication



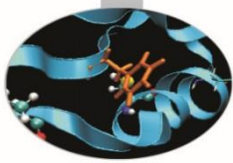
- MM cost determined by FLOPS and memory movement:
  - $2*n^3 = O(n^3)$ FLOPS
  - Operates on $3*n^2 = O(n^2)$ numbers
- To optimize matrix multiplication, we must ensure that for every memory access we execute as many FLOPS as possible.
- Outer product algorithms are faster, but for pedagogical reasons,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

Optimization issues: memory coalescing



- Efficient access to memory
  - Memory coalescing

Aligned (coalesced) access during load/store operations reduce the number of segments moved across the bus

  - Ideally get work-item i to access data[i] and work-item j to access data[j] at the same time etc.
  - Infact, to maximize global memory throughput access is important to maximize coalescing

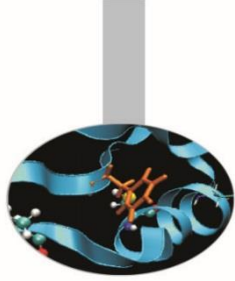
Non coalesced

```
{
  int k;
  int i = get_global_id(0);
  int j = get_global_id(1);
 if ( (i < N) && (j < N) )
{
  float tmp = 0.0f;
  for (k = 0; k < N; k++)
   tmp += A[i*N+k]*B[k*N+j];
  }
  C[i*N+j] += tmp;
}
```

Coalesced

```
{
  int k;
  int i = get_global_id(1);
  int j = get_global_id(0);
 if ( (i < N) && (j < N) )
{
  float tmp = 0.0f;
  for (k = 0; k < N; k++)
   tmp += A[i*N+k]*B[k*N+j];
  }
  C[i*N+j] += tmp;
}
```

Matrix multiplication performance (Galileo compute node, N=2048)



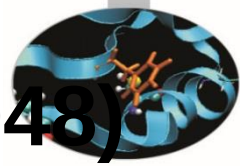
- **Matrices are stored in global memory.**

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	725	N/A
C(i,j) per work-item, all global coalesced	N/A	72680

Big Impact (still not optimal)

GPU Device is Kepler® K80 GPU from NVIDIA® with a max of 13 compute units, 2496 PEs
CPU Device is Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz

Matrix multiplication performance (MontBlanc proto compute node, N=2048)



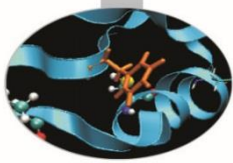
- **Matrices are stored in global memory.**

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	37.9	N/A
C(i,j) per work-item, all global coalesced	N/A	1308

Big Impact

GPU Device is Mali® T604 GPU with a max of 4 compute units
CPU Device is ARM(R) A15 @ 1.7GHz

Optimization issues: occupancy



- Efficient use of resources
  - Occupancy
    - Ideally, try to maximize the number of blocks and warps (on NVIDIA hardware) residing on each SMP for a given kernel
    - Infact, it depends on the NDRange of the call, the memory resources of the multiprocessor, and the resource requirements of the kernel

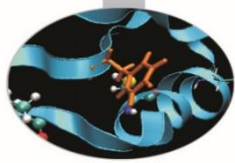
```
const size_t global[2] = {N, N};
err = clEnqueueNDRangeKernel(
    commands,
    kernel,
    2, NULL,
    global, NULL,
    0, NULL, NULL);
```

Original host code

```
const size_t global[2] = {N, N};
    const size_t local[2] = {8, 8};
    err = clEnqueueNDRangeKernel(
        commands,
        kernel,
        2, NULL,
        global, local,
        0, NULL, NULL);
```

Modified host code

Optimization issues: occupancy



Which is the best thread block size/work-group size to select (i.e. `LOCAL`)?
On **Kepler** architectures: each SM can handle up to ***2048*** total threads

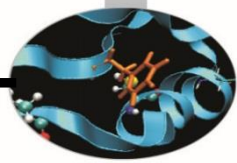
8x8 = 64 threads >>> 2048/64 = 32 blocks needed to fully load a SM **LOCAL = 8**
… yet there is a limit of maximum 16 resident blocks per SM for cc 3.x
so we end up with just 64x16 = 1024 threads per SM on a maximum of 2048 (only **50%** occupancy)

16x16 = 256 threads >>> 2048/256 = 8 blocks to fully load a SM **LOCAL = 16**
8x256 = 2048 threads per SM … reaching **full occupancy** per SM!

32x32 = 1024 threads >>> 2048/1024 = 2 blocks fully load a SM **LOCAL = 32**
2x1024 = 2048 threads per SM … reaching **full occupancy** per SM!

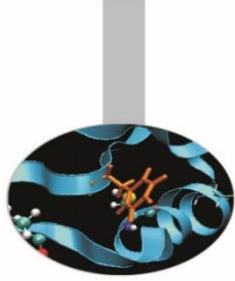
LOCAL = 16 or **32**

Exercise 3: run serial and coalesced occupancy findings matMul OpenCL



- Goal:
  - Use **occupancy** directory
- Procedure:
  - Enter in C directory. Modify host source in order to exploit different occupancy parameters.
  - Run make
  - Run the executable
- Expected output:
  - A message to standard output for serial and OpenCL matMul executions (according to different local size)

Matrix multiplication performance (Galileo compute node, N=2048)

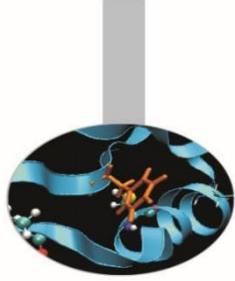


- **Matrices are stored in global memory.**

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	725	N/A
C(i,j) per work-item, all global coalesced Occupancy: local(8,8)	N/A	42950

GPU Device is Kepler® K80 GPU from NVIDIA® with a max of 13 compute units, 2496 PEs
CPU Device is Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz

Matrix multiplication performance (Galileo compute node, N=2048)

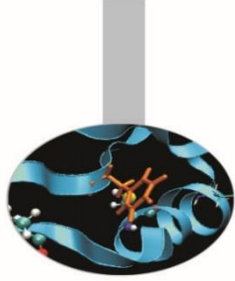


- **Matrices are stored in global memory.**

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	725	N/A
C(i,j) per work-item, all global coalesced Occupancy: local(16,16)	N/A	62593

GPU Device is Kepler® K80 GPU from NVIDIA® with a max of 13 compute units, 2496 PEs
CPU Device is Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz

Matrix multiplication performance (Galileo compute node, N=2048)



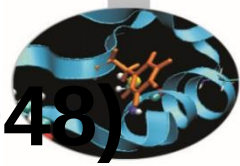
- **Matrices are stored in global memory.**

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	725	N/A
C(i,j) per work-item, all global coalesced Occupancy: local(32,32)	N/A	80287

Best result

GPU Device is Kepler® K80 GPU from NVIDIA® with a max of 13 compute units, 2496 PEs
CPU Device is Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz

Matrix multiplication performance (MontBlanc proto compute node, N=2048)



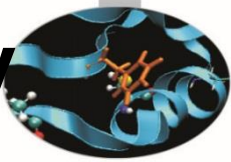
- **Matrices are stored in global memory.**

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	37.9	N/A
C(i,j) per work-item, all global coalesced Occupancy: local(16,16)	N/A	1952

Best result

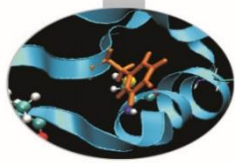
GPU Device is Mali® T604 GPU with a max of 4 compute units
CPU Device is ARM(R) A15 @ 1.7GHz

Optimization issues: Pinned Memory



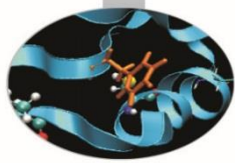
- In general, the fewer transfers you can do between host and device, the better
- But some are unavoidable
- It is possible to speed up these transfers, by using ***pinned memory*** (also called **page-locked** memory)
- If supported, can enable much faster host <-> device communications

Pinned Memory



- A regular **clEnqueueReadBuffer/clEnqueueWriteBuffer** command might manage ~6GB/s
- But PCI-E Gen 3.0 can sustain transfer rates of up to 16GB/s
- So, where has our bandwidth gone?
- **The operating system**
- In fact, an allocation may not even be **contiguous**
- So, **clEnqueueReadBuffer/clEnqueueWriteBuffer** ***must*** incur an additional host memory to host memory copy, wasting bandwidth and costing performance

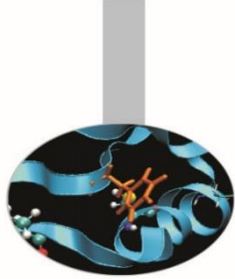
Pinned Memory



- Pinned memory side-steps this issue by giving the host process *direct* access to the portions of host memory that the DMA engines read and write to.
- This results in much less time spent waiting for transfers!

- Disclaimer: Not all drivers support it, and it makes allocations much more expensive (so it would be slow to continually allocate and free pinned memory!)

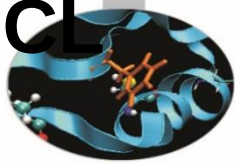
Using Pinned Memory



- OpenCL has no official support for pinned memory
- But e.g. NVIDIA supports pinned memory allocations (**CL_MEM_ALLOC_HOST_PTR** flag)
- In this way, when you allocate a **cl_mem object**, you also allocate **page-locked host memory** of the same size
- But this does not return the host pointer
- Reading and writing data is handled by **clEnqueueMapBuffer**, which *does* return the host pointer
- Eventually call **clEnqueueUnmapMemObject** when you're done

```
//create device buffer
cl_mem devPtrA = clCreateBuffer(
  context,
  CL_MEM_ALLOC_HOST_PTR, //pinned memory flag
  len,
  NULL, //host pointer must be NULL
  NULL
);

float *hostPtrA =
(float *) clEnqueueMapBuffer(
  queue,
  devPtrA,
  CL_TRUE, //blocking map
  CL_MAP_WRITE, //write data
  0,       //offset of region
  len,     //amount of data to be mapped
  0, NULL, NULL, //event information
  NULL     //error code pointer
);
```

Exercise: measuring bandwidth with OpenCL

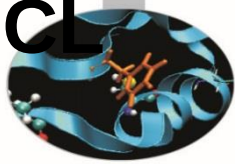


- Measure memory bandwidth versus increasing data size, for Host to Device, Device to Host and Device to Device transfers
- Rely on the oclBandwidthTest provided into the **nvidia-opencl-examples-cuda.4.2.9.sdk** directory:
  `./oclBandwidthTest --mode=range --start=<B> --end=<B> --increment=<B>`

Galileo compute node

Size (MB)	HtoD	DtoH	DtoD
1	3569	3718	71483
10	5198	5588	145985
100	7755	10336	166850

Exercise: measuring bandwidth with OpenCL

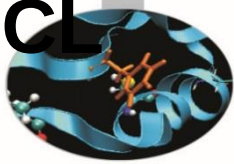


- Measure memory bandwidth versus increasing data size, for Host to Device, Device to Host and Device to Device transfers
- Rely on the oclBandwidthTest provided into the **nvidia-opencl-examples-cuda.4.2.9.sdk** directory:
  `./oclBandwidthTest --mode=range --start=<B> --end=<B> --increment=<B>`

Galileo compute node Pinned Memory

Size (MB)	HtoD	DtoH	DtoD
1	6814	6970	71999
10	7120	7996	145582
100	9137	11099	166794

Exercise: measuring bandwidth with OpenCL

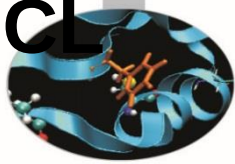


- Measure memory bandwidth versus increasing data size, for Host to Device, Device to Host and Device to Device transfers
- Rely on the oclBandwidthTest provided into the **nvidia-opencl-examples-cuda.4.2.9.sdk** directory:
  `./oclBandwidthTest --mode=range --start=<B> --end=<B> --increment=<B>`

MontBlanc proto

Size (MB)	HtoD	DtoH	DtoD
1	2481	2537	5090
10	1846	2164	3737
100	3055	3085	6112

Exercise: measuring bandwidth with OpenCL

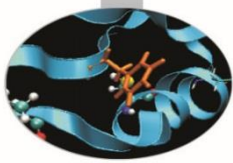


- Measure memory bandwidth versus increasing data size, for Host to Device, Device to Host and Device to Device transfers
- Rely on the oclBandwidthTest provided into the **nvidia-opencl-examples-cuda.4.2.9.sdk** directory:
  `./oclBandwidthTest --mode=range --start=<B> --end=<B> --increment=<B>`

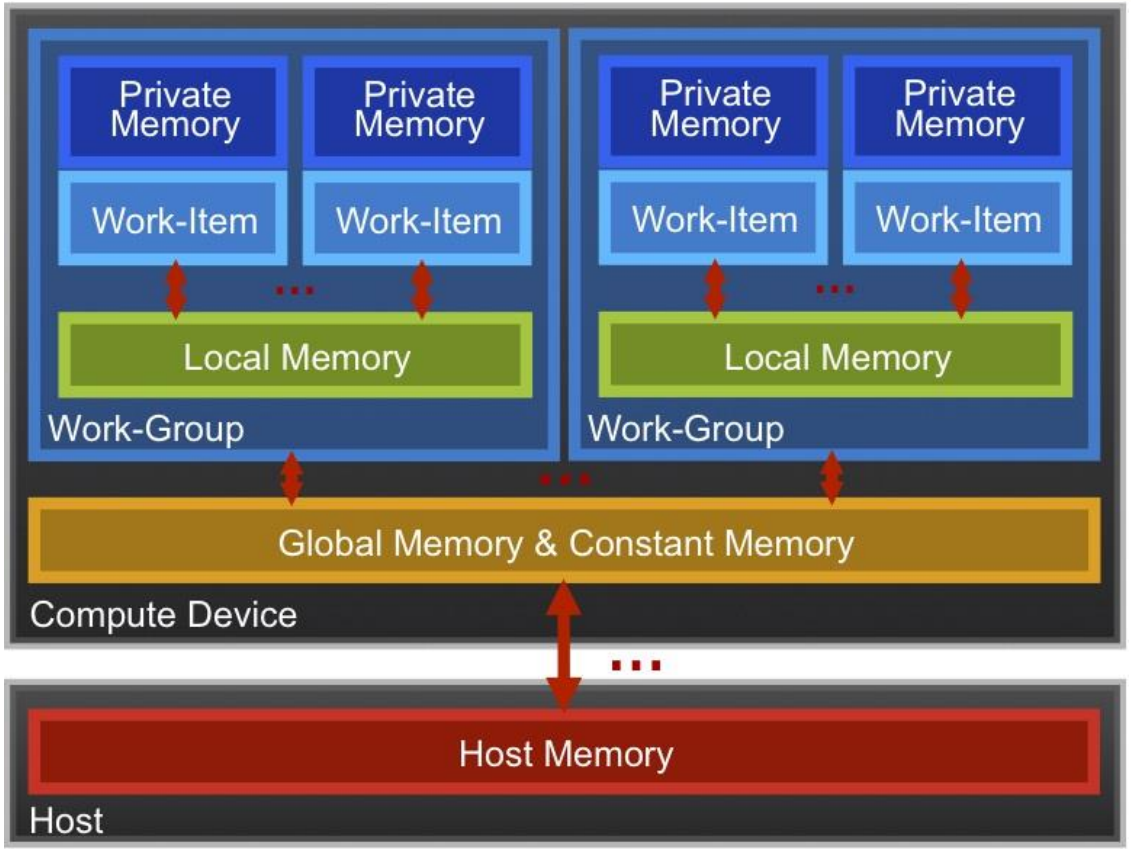
MontBlanc proto Pinned Memory

Size (MB)	HtoD	DtoH	DtoD
1	2552	2585	5141
10	2998	3033	5985
100	3067	3098	5831

OpenCL Memory model

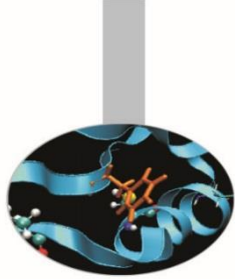


- Private Memory
  - Per work-item
- Local Memory
  - Shared within a work-group
- Global/Constant Memory
  - Visible to all work-groups
- Host memory
  - On the CPU

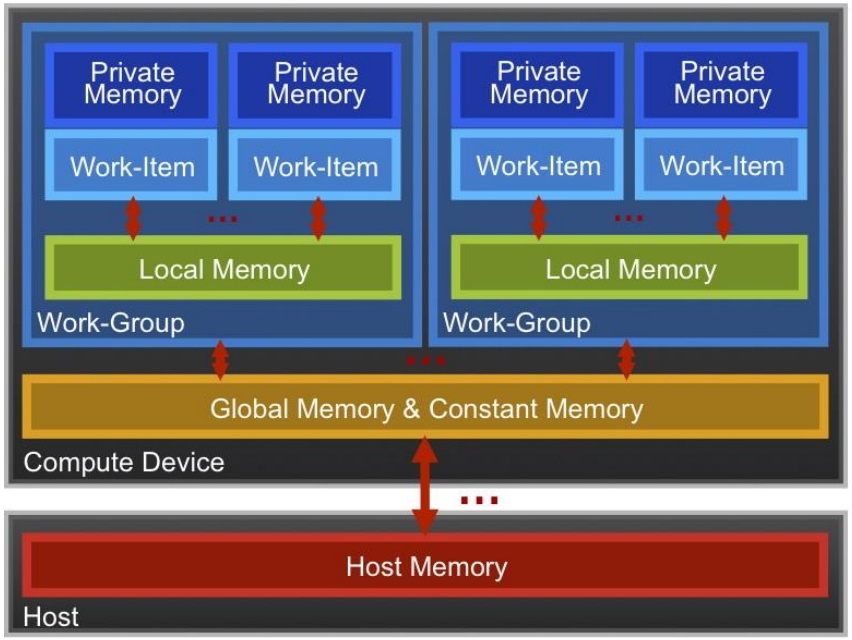


Memory management is explicit:
You are responsible for moving data from host → global → local *and* back

OpenCL Memory model

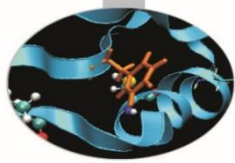


- Private Memory
  - Fastest & smallest: O(10) words/WI
- Local Memory
  - Shared by all WI's in a work-group
  - But not shared between work-groups!
  - O(1-10) Kbytes per work-group
- Global/Constant Memory
  - O(1-10) Gbytes of Global memory
  - O(10-100) Kbytes of Constant memory
- Host memory
  - On the CPU - GBytes

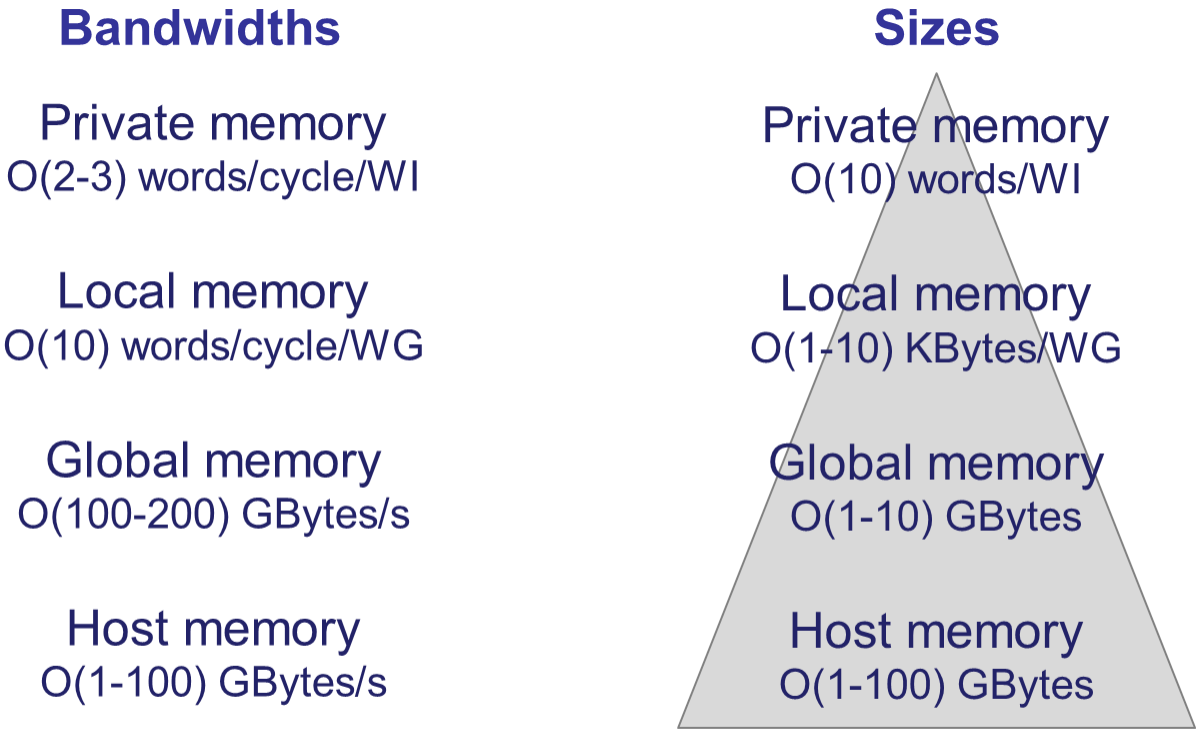


Memory management is explicit:
O(1-10) Gbytes/s bandwidth to discrete GPUs for
Host <-> Global transfers

Optimization issues: exploit memory hierarchy

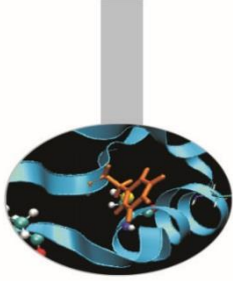


- Efficient use of resources
  - Memory hierarchy
    - Managing the memory hierarchy is one of *the* most important things to get right to achieve good performance

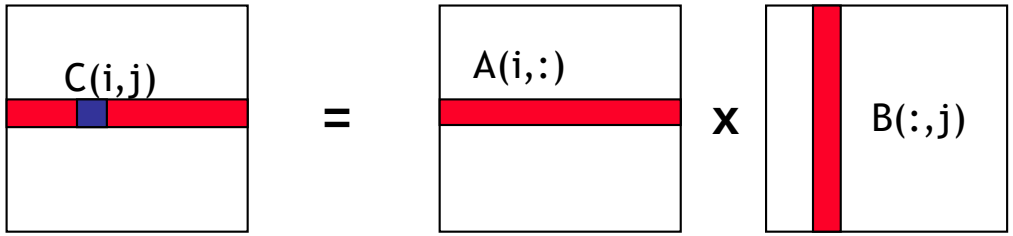


Speeds and feeds approx. for a high-end discrete GPU, circa 2011

Optimization issues: 1D NDRange

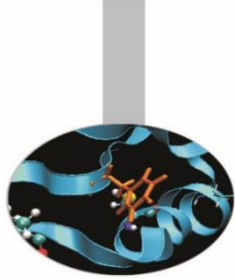


- Efficient use of resources
  - 1D NDRange
    - There may be significant overhead to manage work-items and work-groups.
    - So let's have each work-item compute a full row of C.



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

Optimization issues: 1D NDRange

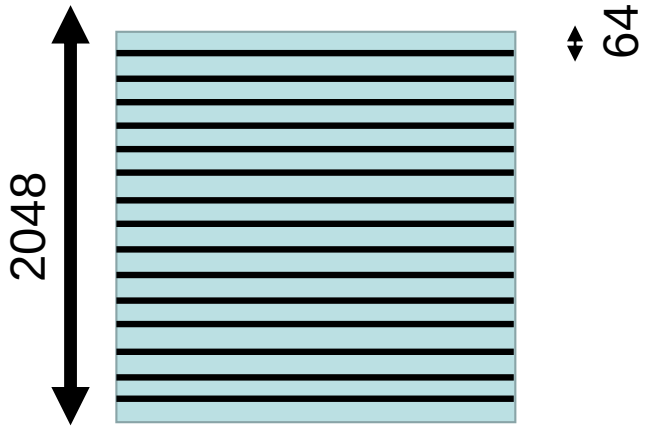


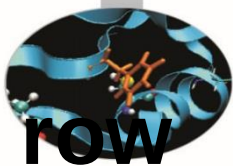
- Global Dimensions: 2048 (1D)

  Whole problem space (index space)

- Local Dimensions: 64 (work-items per work-group)

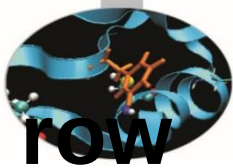
  Only 2048/64 = 32 work-groups in total





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;
  }
}
```



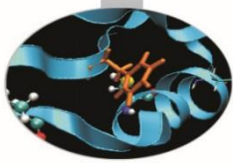
Matrix multiplication: One work item per row of C

```
__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;
  }
}
```

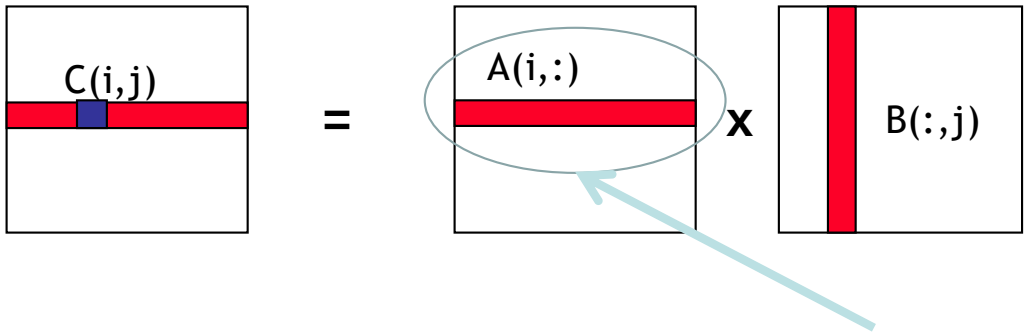
Changes to host program:

1. 1D ND Range set to number of rows in the C matrix
2. Local Dimension set to 64 so number of work-groups should be 32

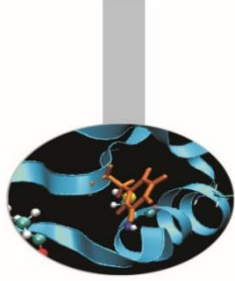
Optimizing matrix multiplication



- 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



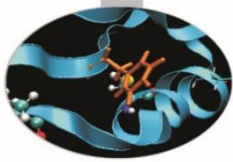
- Private Memory:
  - A very scarce resource, only a few tens of 32-bit words per Work-Item at most
  - If you use too much it spills to global memory or reduces the number of Work-Items that can be run at the same time, potentially harming performance*
  - Think of these like registers on the CPU

* Occupancy on a GPU

Why using too much private memory can be a good thing

- In reality private memory is just hardware registers, so only dozens of these are available per work-item
- Many kernels will allocate too many variables to private memory
- So the compiler already has to be able to deal with this
- It does so by *spilling* excess private variables to (global) memory
- You still told the compiler something useful – that the data will only be accessed by a single work-item
- This lets the compiler allocate the data in such as way as to enable more efficient memory access

Exercise 4: run serial and 1D NDRange and private matMul OpenCL



- Goal:
  - Use **private** directory
- Procedure:
  - Enter in C directory. Modify host source in order to exploit 1D NDRange decomposition. Try to set local dimension to 64.
  - Modify the kernel so that each work-item copies its own row of A into private memory
  - Run make
  - Run the executable
- Expected output:
  - A message to standard output for serial and OpenCL matMul executions (according to different local size)

Matrix multiplication: (Row of A in private memory)

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

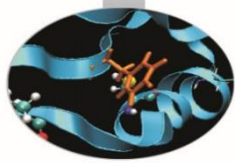
```
__kernel void mmul(
  const int N,
  __global float *A,
  __global float *B,
  __global float *C)
{
  int j, k;
  int i =
    get_global_id(0);
  float tmp;
  float Awrk[2048];

    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;
  }
}
```

Setup a work array for A in private memory*

CINECA

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

Matrix multiplication performance (Galileo compute node, N=2048)



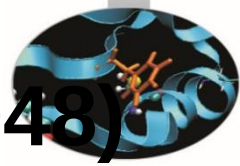
- **Matrices are stored in global memory. 1D NDRange. Each row of A in private memory**

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	725	N/A
C(i,j) per work-item, all global. 1D NDRange. Rows of A in private memory	N/A	16999

GPU Device is Kepler® K80 GPU from NVIDIA® with a max of 13 compute units, 2496 PEs
CPU Device is Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz

Altough not optimal 4 times faster than simple 1D NDRange (which is not shown)

Matrix multiplication performance (MontBlanc proto compute node, N=2048)



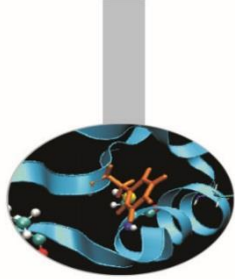
- **Matrices are stored in global memory. 1D NDRange. Each row of A in private memory**

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	37.9	N/A
C(i,j) per work-item, all global 1D NDRange. Rows of A in private memory	N/A	173

GPU Device is Mali® T604 GPU with a max of 4 compute units
CPU Device is ARM(R) A15 @ 1.7GHz

Altough not optimal 2 times faster than simple 1D NDRange (which is not shown)

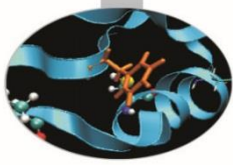
Local Memory*



- **T**ens of KBytes per Compute Unit
  - As multiple Work-Groups will be running on each CU, this means only a fraction of the total Local Memory size is available to each Work-Group
- Assume O(1-10) KBytes of Local Memory per Work-Group
  - Your kernels are responsible for transferring data between Local and Global/Constant memories
  - Use Local Memory to hold data that can be reused by all the work-items in a work-group
- Access patterns to Local Memory affect performance in a similar way to accessing Global Memory
  - Have to think about things like coalescence & bank conflicts

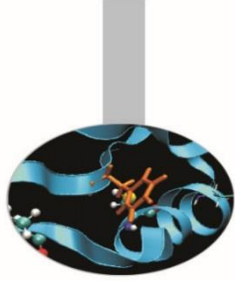
* Typical figures for a 2013 GPU

Memory Consistency



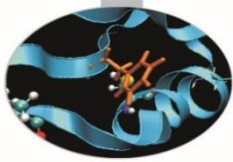
- OpenCL uses a relaxed consistency memory model; i.e.
  - The state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.
- Within a work-item:
  - Memory has load/store consistency to the work-item's private view of memory, i.e. it sees 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)

Thread Synchronization



CUDA	OpenCL
__syncthreads()	barrier()
__threadfenceblock()	mem_fence(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE)
No equivalent	read_mem_fence()
No equivalent	write_mem_fence()
__threadfence()	Finish one kernel and start another

Work-Item Synchronization



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

- 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

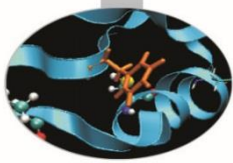
- Across work-groups

  – No guarantees as to where and when a particular work-group will be executed relative to another work-group

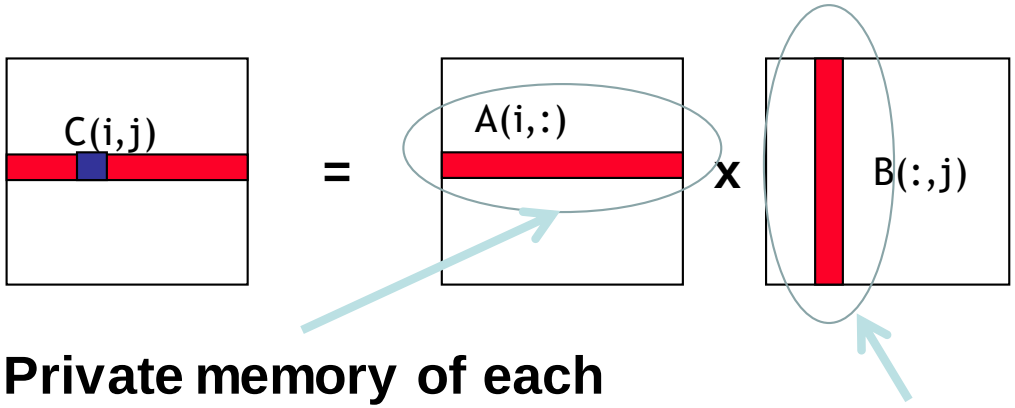
  – Cannot exchange data, or have barrier-like synchronization between two different work-groups! (Critical issue!)

  – Only solution: finish the kernel and start another

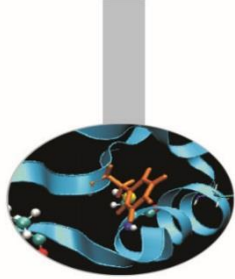
Optimizing matrix multiplication



- 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)



Declaring dynamic local/shared memory



CUDA C

1. Define an array in the kernel source as extern
   __shared__ int array[];

2. When executing the kernel, specify the third parameter as size in bytes of shared memory

```
func<<<num_blocks,
 num_threads_per_block,
 shared_mem_size>>>(args);
```

OpenCL C

1. Have the kernel accept a local array as an argument

```
__kernel void func(
    __local int *array) {}
```

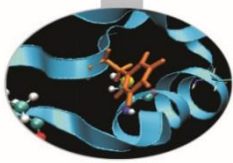
2. Specify the size by setting the kernel argument

```
clSetKernelArg(kernel, 0,
 sizeof(int)*num_elements,
 NULL);
```

Changes to host program:

1. Pass local memory to kernels.
   1. This requires a change to the kernel argument lists … an arg of type float is needed
   2. Allocate the size of local memory
   3. Update argument list in kernel functor

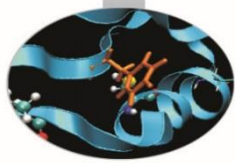
Matrix multiplication: B column shared between work-items



```
__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[2048];

  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)

Matrix multiplication performance (Galileo compute node, N=2048)



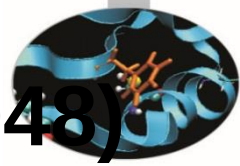
- **Matrices are stored in global memory. 1D NDRange. Each row of A in private memory, B col in shared**

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	725	N/A
C(i,j) per work-item, all global. 1D NDRange. Rows of A in private memory, B col in shared	N/A	14939

GPU Device is Kepler® K80 GPU from NVIDIA® with a max of 13 compute units, 2496 PEs
CPU Device is Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz

Not really exciting performances. More or less near the previous one

Matrix multiplication performance (MontBlanc proto compute node, N=2048)



- **Matrices are stored in global memory. 1D NDRange. Each row of A in private memory, B col in shared**

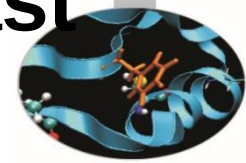
Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	37.9	N/A
C(i,j) per work-item, all global 1D NDRange. Rows of A in private memory, B col in shared	N/A	1654

Good result

GPU Device is Mali® T604 GPU with a max of 4 compute units
CPU Device is ARM(R) A15 @ 1.7GHz

Altough not optimal 10 times faster than the previous one

Making matrix multiplication *really* fast

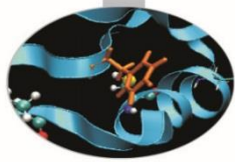


- Our goal has been to describe how to work with private, local and global memory. We've ignored many well-known techniques for making matrix multiplication fast
  - The number of work items must be a multiple of the fundamental machine "vector width". This is the **wavefront** on AMD, **warp** on NVIDIA, and the number of **SIMD lanes** exposed by **vector units** on a CPU
  - To optimize reuse of data, you need to use **blocking** techniques
    - Decompose matrices into **tiles**
    - **Copy** tiles into **local memory**
    - Do the **multiplication over the tiles**
    - **Update** global matrix

Changes to host program:

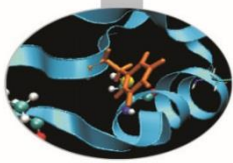
1. Back to the 2D decomposition
   1. This requires a change to the kernel argument lists … two args of type float are needed
   2. Allocate the size of local memory
   3. Update argument list in kernel functor
   4. Set the local worksize to the "warp size"

Exercise 5: run serial and 2D NDRange and shared memory matMul OpenCL



- Goal:
  - Use **private** directory
- Procedure:
  - Enter in C directory. Modify host source in order to set local dimension to “warp size”.
  - Modify the kernel filling each “dummy” assignment with correct syntax
  - Run make
  - Run the executable
- Expected output:
  - A message to standard output for serial and OpenCL matMul executions

Matrix multiplication: 2D tiles



```
#define blksz 32
```

"Warp Size"

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

  //  This work-item will compute element C(i,j)
  const int i = get_global_id(0);
  const int j = get_global_id(1);

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

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

  // The number of blocks are the same in each dimension
  const int Num_BLK = N/blksz;

  // Setup the upper-left-corner (base address) for the A and
  // B blocks plus the increments to advance base addresses as
  // we loop over blocks
      int Abase = Iblk*N*blksz;
  const int Ainc  = blksz;

      int Bbase = Jblk*blksz;
  const int Binc  = blksz*N;
```

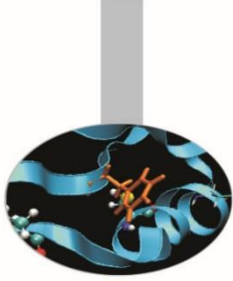
```
// C(Iblk,Jblk) = (sum over Kblk) A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk= 0;  Kblk<Num_BLK;  Kblk++)
  {
    // Load A(Iblk,Kblk) and B(Kblk,Jblk) into local memory.
    // Each work-item loads a single element of the two blocks
    // which are shared with the entire work-group.

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

    barrier(CLK_LOCAL_MEM_FENCE);

    // Compute dot products over local blocks to find
    // the contribution to C(i,j) from this block
    #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;
  }

  // update global C matrix
  C[j*N+i] = Ctmp;
}
```

Copy tiles to local memory

Do the multiplication over the tiles

Update global matrix

Matrix multiplication performance (Galileo compute node, N=2048)



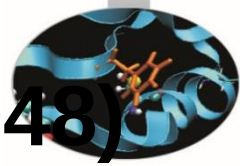
- **Matrices are stored in global memory. 2D NDRange. Shared memory tiles**

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	725	N/A
C(i,j) per work-item, all global. 2D NDRange. Shared memory tiles	N/A	227680

Exciting results

GPU Device is Kepler® K80 GPU from NVIDIA® with a max of 13 compute units, 2496 PEs
CPU Device is Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz

Matrix multiplication performance (MontBlanc proto compute node, N=2048)



- **Matrices are stored in global memory. 2D NDRange. Shared memory tiles**

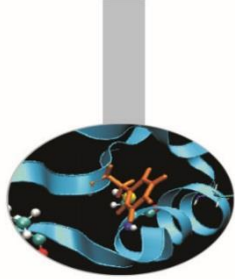
Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	37.9	N/A
C(i,j) per work-item, all global 2D NDRange. Shared memory tiles	N/A	980

Not so good result

GPU Device is Mali® T604 GPU with a max of 4 compute units
CPU Device is ARM(R) A15 @ 1.7GHz

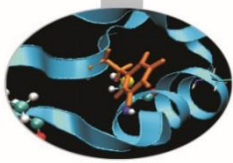
`Blksize setted to 8. What happens?`

Local Memory



- Local Memory doesn't always help…
  - CPUs (Mali GPUs….) don't have special hardware for it
  - For example, Mali use global memory instead of local
  - This can mean excessive use of Local Memory might slow down kernels
  - So, your mileage may vary!

Vector operations



- Modern microprocessors include vector units:

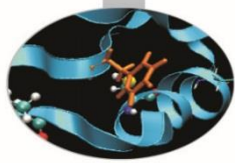
  Functional units that carry out operations on blocks of numbers
- For example, x86 CPUs have over the years introduced MMX, SSE, and AVX instruction sets …

  characterized in part by their widths (e.g. SSE operates on 128 bits at a time, AVX 256 bits etc)
- To gain full performance from these processors it is important to exploit these vector units
- Compilers can sometimes automatically exploit vector units.

  Experience over the years has shown, however, that you all too often have to code vector operations by hand.
- Example using 128 bit wide SSE:

```
#include "xmmintrin.h"    // vector intrinsics from gcc for SSE (128 bit wide)

__m128 ramp = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);  // pack 4 floats into vector register
__m128 vstep = _mm_load1_ps(&step);              // pack step into a vector register
__m128 xvec; = _mm_mul_ps(ramp,vstep);           // multiple corresponding 32 bit
                                                 // floats and assign to xvec
```

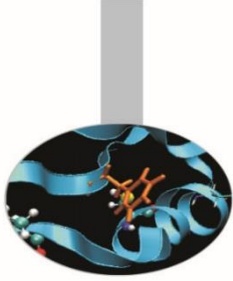
OpenCL Vector Types



- The OpenCL C kernel programming language provides a set of vector instructions:
  - These are portable between different vector instruction sets
- These instructions support vector lengths of 2, 4, 8, and 16 … for example:
  - **char2, ushort4, int8, float16, double2**, …
- Properties of these types include:
  - Endian safe
  - Aligned at vector length
  - Vector operations (elementwise) and built-in functions

Remember, double (and hence vectors of double) are optional in OpenCL v1.1

Vector Operations



- Vector literal

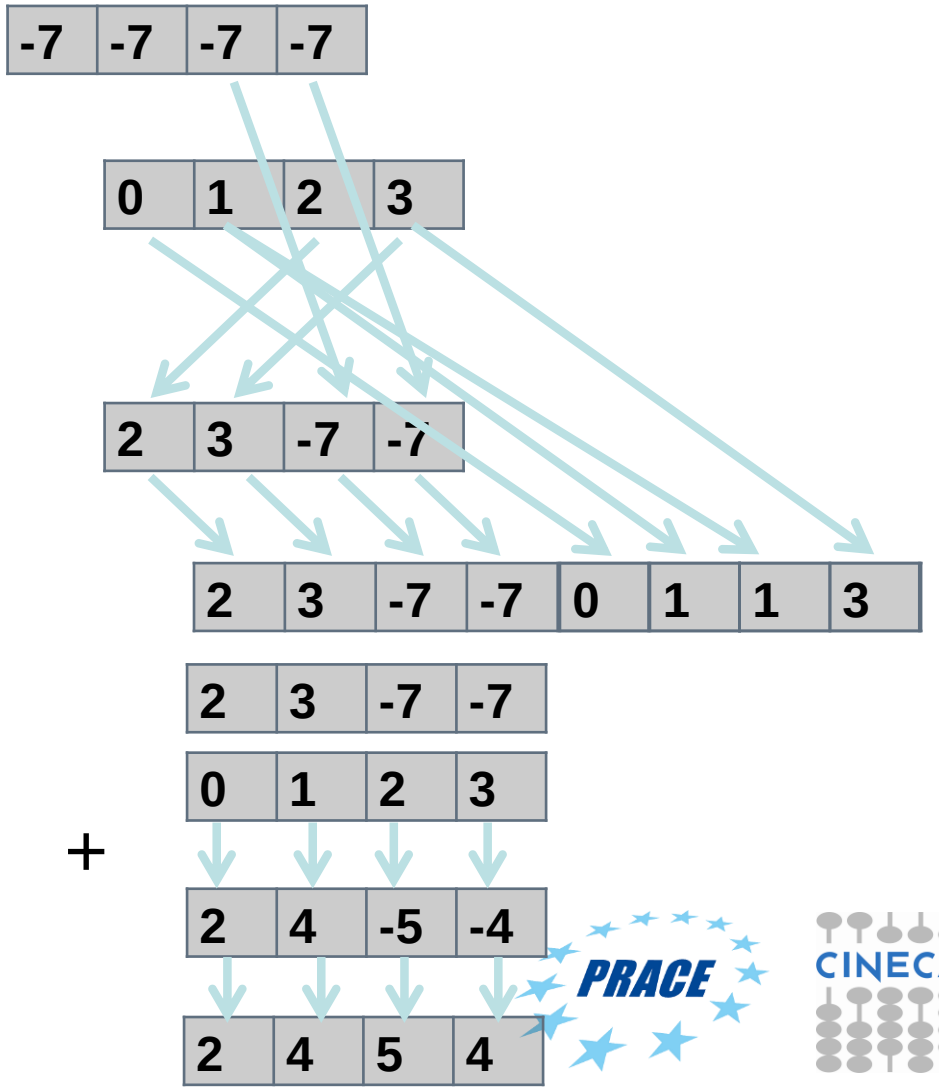
```
vi0 = (      ) -7;

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

- Vector components

```
vi0.lo = vi1.hi;

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

- Vector ops

```
vi0 += vi1;

vi0 = abs(vi0);
```



Making matrix multiplication vectorized

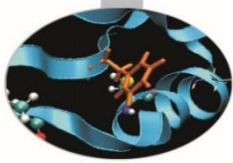
- Our goal has been to describe how to work with vectorization
  - The vectorised matrix multiplication sketch
    - Matrix A consists of $\left(\mathrm{N}, \frac{\mathrm{N}}{4}\right)$ submatrix of (1×4) *elements*
    - Matrix B consists of $\left(\frac{\mathrm{N}}{4}, \frac{\mathrm{N}}{4}\right)$ submatrix of of (4×4) *elements*
    - Matrix C consists of $\left(\mathrm{N}, \frac{\mathrm{N}}{4}\right)$ submatrix of of (1×4) *elements*

Changes to host program:

1. 2D decomposition
2. Set:
   1. const int NV4=N>>2
   2. const size_t global[2] = {NV4, N};

Before clEnqueueNDRangeKernel call

Matrix multiplication: 2D vector



```
__kernel void mmul(
    const int N,
    __global float4* A,
    __global float4* B,
    __global float4* C)
{
uint j = get_global_id (0);
uint i = get_global_id (1);
uint nv4 = N >> 2;
if ((i < N) && (j < nv4))
    {
float4 accum = ( float4 ) 0.0;
for ( uint k = 0; k < nv4 ; ++k)
        {
        float4 a = A[i*nv4 + k];
        float4 b0 = B[(4*k +0)* nv4 + j];
        float4 b1 = B[(4*k +1)* nv4 + j];
        float4 b2 = B[(4*k +2)* nv4 + j];
        float4 b3 = B[(4*k +3)* nv4 + j];
        accum +=
a.s0*b0+a.s1*b1+a.s2*b2+a.s3*b3;
        }
        C[i*nv4 + j] = accum;
}
}
```

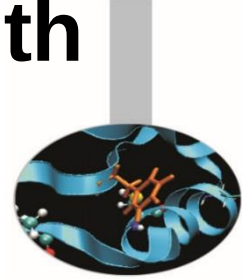
"Packed-4" input and output arrays

"Packed-4" consecutive elements of A starting at address i*nv4+k

"Packed-4x4" consecutive elements of B starting at addresses (4*k+0/1/2/3)*nv4+j

"Vector" operations

Matrix multiplication: 2D vector with vload4/vstore4 intrinsic



```
__kernel void mmul(
    const int N,
    __global float* A,
    __global float* B,
    __global float* C)
{
uint j = get_global_id (0);
uint i = get_global_id (1);
uint nv4 = N >> 2;
if ((i < N)  && (j < nv4))
   {
 {
float4 accum = ( float4 ) 0.0;
for ( uint k = 0; k < nv4 ; ++k)
                {
                float4 a = vload4(0,&A[i*nv4 + k]);
                float4 b0 = vload4(0,&B[(4*k +0)* nv4 + j]);
                float4 b1 = vload4(0,&B[(4*k +1)* nv4 + j]);
                float4 b2 = vload4(0,&B[(4*k +2)* nv4 + j]);
                float4 b3 = vload4(0,&B[(4*k +3)* nv4 + j]);
                accum +=
a.s0*b0+a.s1*b1+a.s2*b2+a.s3*b3;
                }
                vstore4(accum,i*nv4 + j,C);
}
}
```

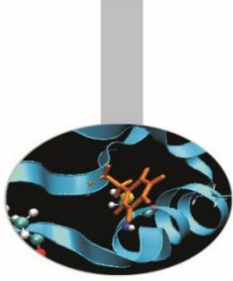
Input and output arrays

“Packed-4” consecutive elements of A starting at address i*nv4+k

“Packed-4x4” consecutive elements of B starting at addresses (4*k+0/1/2/3)*nv4+j

“Vector” operations

Matrix multiplication performance (Galileo compute node, N=2048)



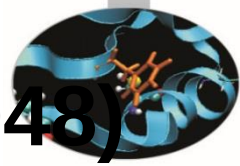
- **Matrices are stored in global memory. 2D NDRange Vectorised.**

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	725	N/A
C(i,j) per work-item, all global. 2D NDRange. Vectorised	N/A	97910

Good result

GPU Device is Kepler® K80 GPU from NVIDIA® with a max of 13 compute units, 2496 PEs
CPU Device is Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz

Matrix multiplication performance (MontBlanc proto compute node, N=2048)



- **Matrices are stored in global memory. 2D NDRange. Vectorised**

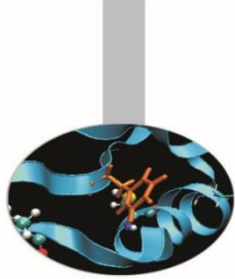
Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	37.9	N/A
C(i,j) per work-item, all global 2D NDRange. Vectorised	N/A	3136

Best result

GPU Device is Mali® T604 GPU with a max of 4 compute units
CPU Device is ARM(R) A15 @ 1.7GHz

What happens? Vectorisation is the key of success for the Mali GPU

Credits



Among the others:

- Simon McIntosh Smith
- MontBlanc/MontBlanc2 project (EU FP7)