∼

# Parallelization using the GPU

Scientific Computing Winter 2016/2017

Lecture 29

Jürgen Fuhrmann

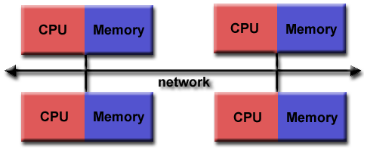
juergen.fuhrmann@wias-berlin.de

made wit pandoc

∼

## **R**ecap

## MIMD Hardware: Distributed memory



[Source: computing.llnl.gov/tutorials]

- “Linux Cluster”
- “Commodity Hardware”
- Memory scales with number of CPUs interconneted
- High latency for communication
- Mostly programmed using MPI (Message passing interface)
- Explicit programming of communications:
  gather data, pack, send, receive, unpack, scatter

## MPI - Message passing interface

- library, can be used from C,C++, Fortran, python
- de facto standard for programming on distributet memory system (since $\approx$ 1995)
- highly portable
- support by hardware vendors: optimized communication speed
- based on sending/receiving messages over network
  - instead, shared memory can be used as well
- very elementary programming model, need to hand-craft communications

## How to install

- OpenMP/C++11 threads come along with compiler
- MPI needs to be installed in addition
- Can run on multiple systems
- openmpi available for Linux/Mac (homebrew)/ Windows (cygwin)
  - `https://www.open-mpi.org/faq/?category=mpi-apps`
  - Compiler wrapper `mpic++` - wrapper around (configurable) system compiler - proper flags + libraries to be linked
  - Process launcher `mpirun`
- launcher starts a number of processes which execute statements independently, ocassionally waiting for each other

## Threads vs processes

- Threads are easier to create than processes since they don't require a separate address space.
- Multithreading requires careful programming since threads share data strucures that should only be modified by one thread at a time. Unlike threads, processes don't share the same address space.
- Threads are considered lightweight because they use far less resources than processes.
- Processes are independent of each other. Threads, since they share the same address space are interdependent, so caution must be taken so that different threads don't step on each other.
  This is really another way of stating #2 above.
- A process can consist of multiple threads.
- MPI is based on processes, C++11 threads and OpenMP are based on threads.

## MPI Hello world

```
//  Initialize MPI.
MPI_Init ( &argc, &argv );

//  Get the number of processes.
MPI_Comm_size ( MPI_COMM_WORLD, &nproc );

// Create index vector for processes
std::vector<unsigned long> idx(nproc+1);

//  Determine the rank (number) of this process.
MPI_Comm_rank ( MPI_COMM_WORLD, &iproc );

if ( iproc == 0 )    cout << "The number of processes available is " << nproc << "\n";

cout << "Hello from proc  " << iproc << endl;

MPI_Finalize ( );
```

- Compile with `mpic++ mpi-hello.cpp -o mpi-hello`
- All MPI programs begin with `MPI_Init()` and end with `MPI_Finalize()`
- the *communicator* `MPI_COMM_WORLD` designates all processes in the current process group, there may be other process groups etc.
- The whole program is started $N$ times as system process, not as thread: `mpirun -n N mpi-hello`

## MPI hostfile

```
host1  slots=n1
host2  slots=n2

...
```

- Distribute code execution over several hosts
- Need ssh public key access and common file system acces for proper execution

## MPI Send

```
MPI_Send (start, count, datatype, dest, tag, comm)
```

- The message buffer is described by (start, count, datatype)
- The target process is specified by dest, which is the rank of the target process in the communicator specified by comm
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.
- The tag codes some type of message

## MPI Receive

```
MPI_Recv(start, count, datatype, source, tag, comm, status)
```

- Waits until a matching (on source and tag) message is received from the system, and the buffer can be used.
- source is rank in communicator specified by comm, or MPI_ANY_SOURCE
- status contains further information
- Receiving fewer than count occurrences of datatype is OK, but receiving more is an error.

## MPI Broadcast

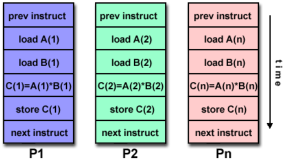
```
MPI_Bcast(start, count, datatype, root, comm )
```

- Broadcasts a message from the process with rank “root” to all other processes of the communicator
- Root sends, all others receive.

## Differences with OpenMP

- Programmer has to care about all aspects of communication and data distribution, even in simple situations
- In simple situations (regularly structured data) OpenMP provides reasonable defaults. For MPI these are not available
- For PDE solvers (FEM/FVM assembly) on unstructured meshes, in both cases we have to care about data distribution
- We need explicit handling of data at interfaces

## SIMD Hardware: Graphics Processing Units ( GPU)



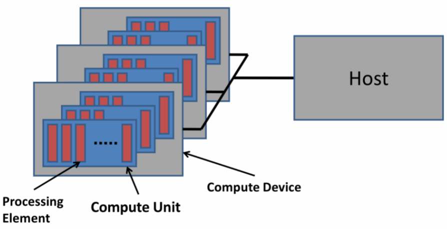
[Source: computing.llnl.gov/tutorials]

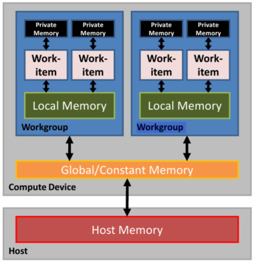
- Principle useful for highly structured data
- Example: textures, triangles for 3D graphis rendering
- During the 90's, *Graphics Processing Units* (GPUs) started to contain special purpose SIMD hardware for graphics rendering
- 3D Graphic APIs (DirectX, OpenGL) became transparent to programmers: rendering could be influences by "shaders" which essentially are programs which are compiled on the host and run on the GPU

[Source:HardwareZone.com.ph]

## General Purpose Graphics Processing Units (GPGPU)

- Graphics companies like NVIDIA saw an opportunity to market GPUs for computational purposes
- Emerging APIs which allow to describe general purpose computing tasks for GPUs: CUDA (Nvidia specific), OpenCL (ATI/AMD designed, general purpose), OpenACC(future ?)
- GPGPUs are *accelerator cards* added to a computer with own memory and many vector processing pipelines
  (NVidia Tesla K40: 12GB + 2880 units)
- CPU-GPU connection generally via mainbord bus



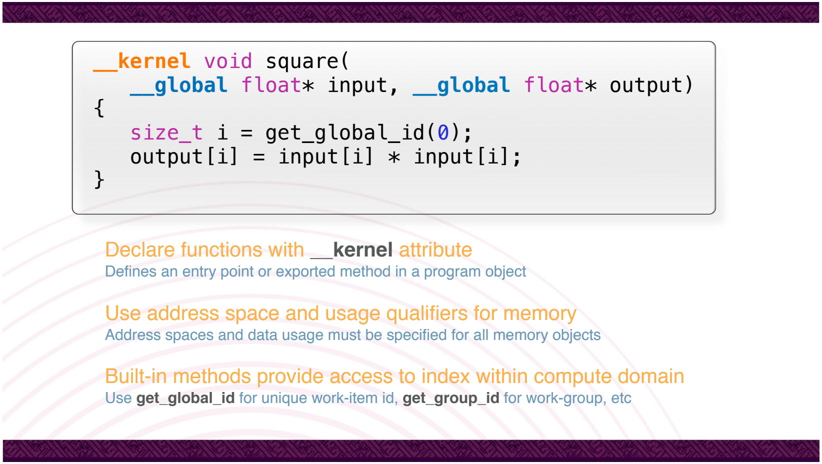


[Source: amd-dev.wpengine.netdna-cdn.com]

## GPU Programming paradigm

- CPU:
  - sets up data
  - triggers compilation of “kernels”: the heavy duty loops to be executed on GPU
  - sends compiled kernels (“shaders”) to GPU
  - sendse data to GPU, initializes computation
  - receives data back from GPU
- GPU:
  - receive data from host CPU
  - just run the heavy duty loops im local memory
  - send data back to host CPU
- CUDA and OpenCL allow explicit management of these steps
- High effiency only with good match between data structure and layout of GPU memory (2D rectangular grid)

## Example: OpenCL: computational kernel



```
__kernel void square(
   __global float* input, __global float* output)
{
   size_t i = get_global_id(0);
   output[i] = input[i] * input[i];
}
```

Declare functions with **__kernel** attribute

Defines an entry point or exported method in a program object

Use address space and usage qualifiers for memory

Address spaces and data usage must be specified for all memory objects

Built-in methods provide access to index within compute domain

Use **get_global_id** for unique work-item id, **get_group_id** for work-group, etc

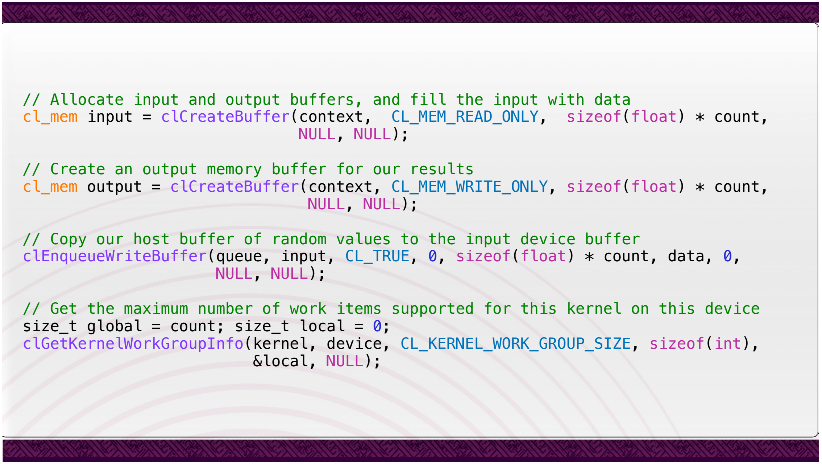
[Source: http://sa10.idav.ucdavis.edu/docs/sa10-dg-opencl-overview.pdf]

## OpenCL: Resource build up, kernel creation

```
// Fill our data set with random float values
int count = 1024 * 1024;
for(i = 0; i < count; i++)
    data[i] = rand() / (float)RAND_MAX;

// Connect to a compute device, create a context and a command queue
cl_device_id device;
clGetDeviceIDs(CL_DEVICE_TYPE_GPU, 1, &device, NULL);
cl_context context = clCreateContext(0, 1, & device, NULL, NULL, NULL);
cl_command_queue queue = clCreateCommandQueue(context, device, 0, NULL);

// Create and build a program from our OpenCL-C source code
cl_program program = clCreateProgramWithSource(context, 1, (const char **) &src,
                                               NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// Create a kernel from our program
cl_kernel kernel = clCreateKernel(program, "square", NULL);
```

[Source: http://sa10.idav.ucdavis.edu/docs/sa10-dg-opencl-overview.pdf]

## OpenCL: Data copy to GPU



```c
// Allocate input and output buffers, and fill the input with data
cl_mem input = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(float) * count,
                              NULL, NULL);

// Create an output memory buffer for our results
cl_mem output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * count,
                               NULL, NULL);

// Copy our host buffer of random values to the input device buffer
clEnqueueWriteBuffer(queue, input, CL_TRUE, 0, sizeof(float) * count, data, 0,
                     NULL, NULL);

// Get the maximum number of work items supported for this kernel on this device
size_t global = count; size_t local = 0;
clGetKernelWorkGroupInfo(kernel, device, CL_KERNEL_WORK_GROUP_SIZE, sizeof(int),
                         &local, NULL);
```

[Source: http://sa10.idav.ucdavis.edu/docs/sa10-dg-opencl-overview.pdf]

## OpenCL: Kernel execution, result retrieval from GPU

```c
// Set the arguments to our kernel, and enqueue it for execution
clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
clSetKernelArg(kernel, 2, sizeof(unsigned int), &count);
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global, &local, 0, NULL, NULL);

// Force the command queue to get processed, wait until all commands are complete
clFinish(queue);

// Read back the results
clEnqueueReadBuffer( queue, output, CL_TRUE, 0, sizeof(float) * count, results, 0,
                     NULL, NULL );

// Validate our results
int correct = 0;
for(i = 0; i < count; i++)
    correct += (results[i] == data[i] * data[i]) ? 1 : 0;

// Print a brief summary detailing the results
printf("Computed '%d/%d' correct values!\n", correct, count);
```

[Source: http://sa10.idav.ucdavis.edu/docs/sa10-dg-opencl-overview.pdf]

## OpenCL Summary

- Need good programming experience and system management skills in order to set up tool chains with properly matching versions, vendor libraries etc.
  - (I was not able to get this running on my laptop in finite time. . . )
- Very cumbersome programming, at least as explicit as MPI
- Data structure restrictions limit class of tasks which can run efficiently on GPUs.

## OpenACC (Open Accelerators)

- Idea similar to OpenMP: use compiler directives
- Future merge with OpenMP intended
- Intended for different accelerator types (GPU, Xeon Phi . . .)
- GCC, Clang implementations on the way (but not yet in the usual repositories)

## OpenACC Sample program

```
#define N 2000000000

#define vl 1024

int main(void) {

  double pi = 0.0f;
  long long i;

  #pragma acc parallel vector_length(vl)
  #pragma acc loop reduction(+:pi)
  for (i=0; i<N; i++) {
    double t= (double)((i+0.5)/N);
    pi +=4.0/(1.0+t*t);
  }

  printf("pi=%11.10f\n",pi/N);

  return 0;

}
```

- compile with `gcc-5 openacc.c -fopenacc -foffload=nvptx-none -foffload="-O3" -O3 -o openacc-gpu`
- . . . . . . but to do this one has to compile gcc with a special configuration. . .

## Other ways to program GPU

- WebGL: directly use capabilities of graphics hardware via html, Javascript in the browser
- Example: Gray-Scott model for Reaction-Diffusion: two chemical species.
  - $U$ is created with rate $f$ and decays with rate $f$
  - $U$ reacts wit $V$ to more $V$
  - $V$ deacays with rate $f+k$.
  - $U, V$ move by diffusion

$$1 \xrightarrow{f} U$$

$$U + 2V \xrightarrow{1} 3V$$

$$V \xrightarrow{f+k} 0$$

$$F \xrightarrow{f} 0$$

- Stable states:
  - No $V$
  - "Much of $V$", then it feeds on $U$ an re-creates itself
- Reaction-Diffusion equation from mass action law:

$$\partial_t u - D_u \Delta u + uv^2 - f(1-u) = 0$$
$$\partial_t v - D_v \Delta v - uv^2 + (f+k)v = 0$$

## Discretization

- ... GPUs are fast so we choose the explicit Euler method:

$$\frac{1}{\tau}(u_{n+1} - u_n) - D_u \Delta u_n + u_n v_n^2 - f(1 - u_n) = 0$$
$$\frac{1}{\tau}(v_{n+1} - u_v) - D_v \Delta v_n - u_n v_n^2 + (f + k) v_n = 0$$

- Finite volume discretization on grid of size *h*

## The shader

```
<script type="x-webgl/x-fragment-shader" id="timestep-shader">
precision mediump float;
uniform sampler2D u_image;
uniform vec2 u_size;
const float F = 0.05, K = 0.062, D_a = 0.2, D_b = 0.1;
const float TIMESTEP = 1.0;
void main() {
    vec2 p = gl_FragCoord.xy,
         n = p + vec2(0.0, 1.0),
         e = p + vec2(1.0, 0.0),
         s = p + vec2(0.0, -1.0),
         w = p + vec2(-1.0, 0.0);

    vec2 val = texture2D(u_image, p / u_size).xy,
         laplacian = texture2D(u_image, n / u_size).xy
        + texture2D(u_image, e / u_size).xy
        + texture2D(u_image, s / u_size).xy
        + texture2D(u_image, w / u_size).xy
        - 4.0 * val;

    vec2 delta = vec2(D_a * laplacian.x - val.x*val.y*val.y + F * (1.0-val.x),
        D_b * laplacian.y + val.x*val.y*val.y - (K+F) * val.y);

    gl_FragColor = vec4(val + delta * TIMESTEP, 0, 0);
}
</script>
```

- Embedded as script into html page

## Why does this work so well here ?

- Data structure fits very well to topology of GPU
  - rectangular grid
  - 2 unknowns to be stored in x,y components of vec2
- GPU speed allows to “break” time step limitation of explicit Euler
- Data stay within the graphics card: once we loaded the initial value, all computations, and rendering use data which are in the memory of the graphics card.
- Depending on the application, choose the best way to proceed
- e.g. deep learning (especially training speed)