Scientific Computing WS 2017/2018

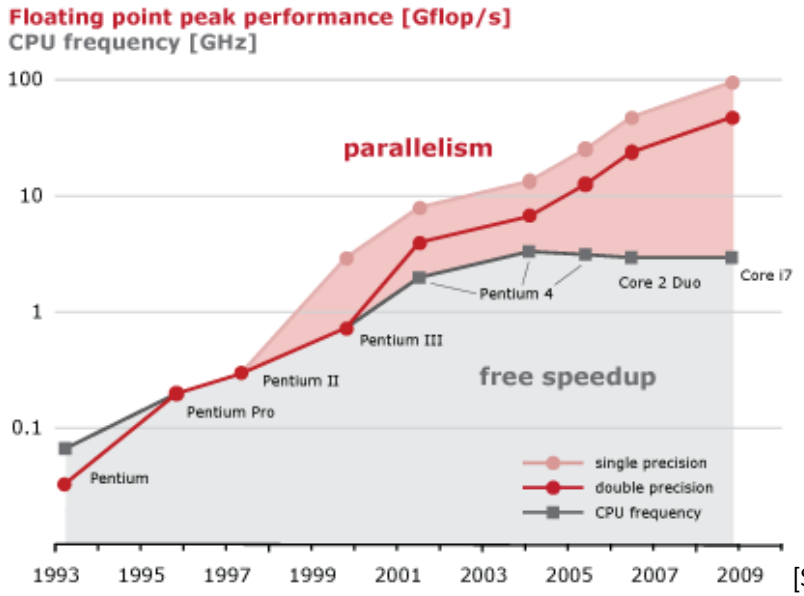
Lecture 27

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

# Why parallelization ?

- Computers became faster and faster without that...

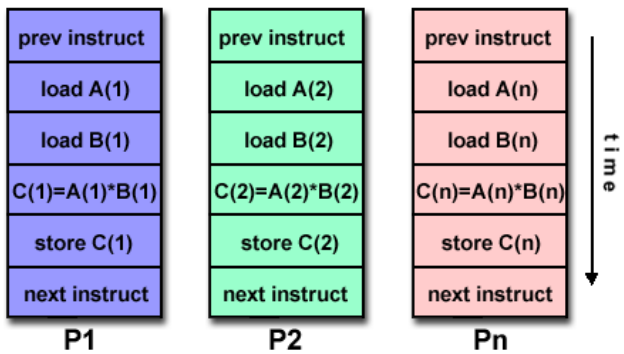



[Source: spiralgen.com]

- But: clock rate of processors limited due to physical limits
- $\Rightarrow$ parallelization is the main road to increase the amount of data processed
- Parallel systems nowadays ubiquitous: even laptops and smartphones have multicore processors
- Amount of accessible memory per processor is limited $\Rightarrow$ systems with large memory can be created based on parallel processors

# Parallel paradigms

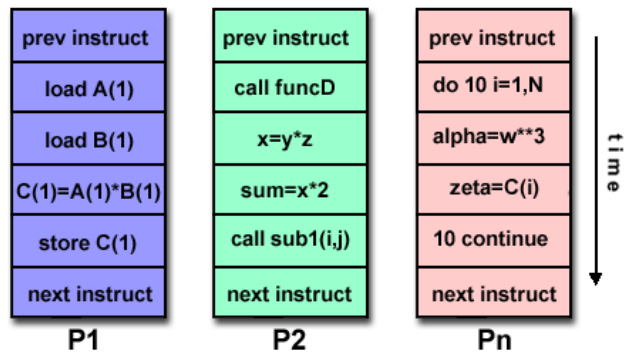
## SIMD
Single Instruction Multiple Data




[Source: computing.llnl.gov/tutorials]

- "classical" vector systems: Cray, Convex ...
- Graphics processing units (GPU)

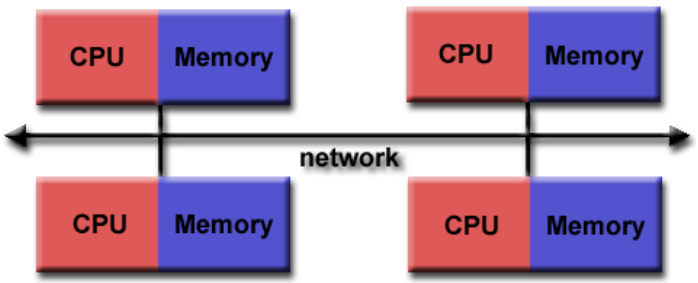
## MIMD
Multiple Instruction Multiple Data




[Source: computing.llnl.gov/tutorials]

- Shared memory systems
  - IBM Power, Intel Xeon, AMD Opteron ...
  - Smartphones ...
  - Xeon Phi R.I.P.
- Distributed memory systems
  - interconnected CPUs

# MIMD Hardware: Distributed memory



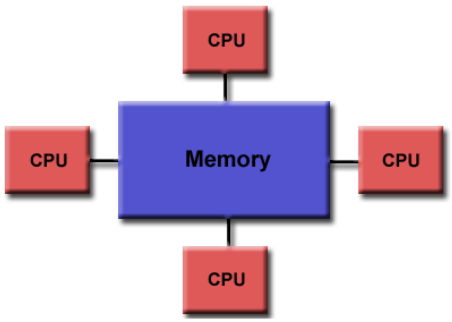

[Source: computing.llnl.gov/tutorials]

- "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_Send(buf,count,type,dest,tag,comm)
MPI_Recv(buf,count,type,src,tag,comm,stat)
```

# MIMD Hardware: Shared Memory

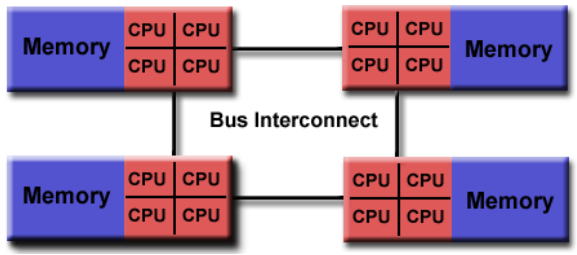
Symmetric Multiprocessing (SMP)/Uniform memory acces (UMA)




[Source: computing.llnl.gov/tutorials]

- Similar processors
- Similar memory access times

Nonuniform Memory Access (NUMA)



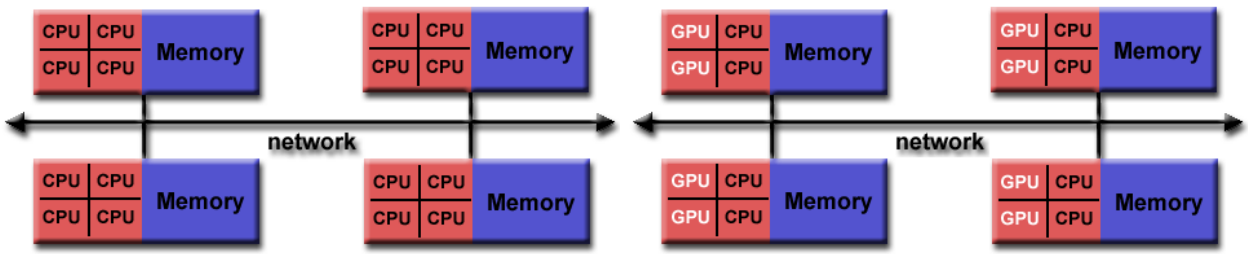

[Source: computing.llnl.gov/tutorials]

- Possibly varying memory access latencies
- Combination of SMP systems
- ccNUMA: Cache coherent NUMA

- Shared memory: one (virtual) address space for all processors involved
- Communication hidden behind memory acces
- Not easy to scale large numbers of CPUS
- MPI works on these systems as well

# Hybrid distributed/shared memory

- Combination of shared and distributed memory approach
- Top 500 computers




[Source: computing.llnl.gov/tutorials]

- Shared memory nodes can be mixed CPU-GPU
- Need to master both kinds of programming paradigms

# MPI Programming

- Typically, one writes *one program* which is started in multiple incarnations on different hosts in a network.
- MPI library calls are used to determine the identiy of a running program
- Communication + barriers have to be programmed explicitely.

# MPI Hello world

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

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

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

if ( iproc == 0 )
cout << "Number of available processes: " << 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 -np N mpi-hello`

# MPI hostfile

```
host1  slots=n1
host2  slots=n2

...
```

- Distribute code execution over several hosts
- MPI gets informed how many independent processes can be run on which node and distributes the required processes accordingly
- MPI would run more processes than slots available. Avoid this situation !
- Need ssh public key access and common file system access for proper execution
- Telling mpi to use host file:
  `mpirun --hostfile hostfile -np N mpi-hello`

# MPI Send

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

- Send data to other process(es)
- The message buffer is described by (start, count, datatype):
  - start: Start address
  - count: number of items
  - datatype: data type of one item
- 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)`

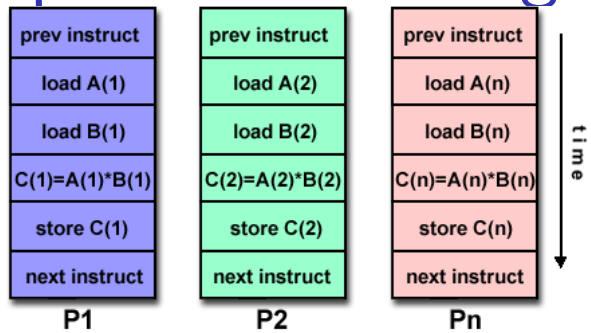
- Waits until a matching (on source and tag) message is received from the system, and the buffer can be used.
- source is rank in communicator specified by comm, or `MPI_ANY_SOURCE`
- status contains further information
- Receiving fewer than count occurrences of datatype is OK, but receiving more is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

# SIMD Hardware: Graphics Processing Units ( GPU)

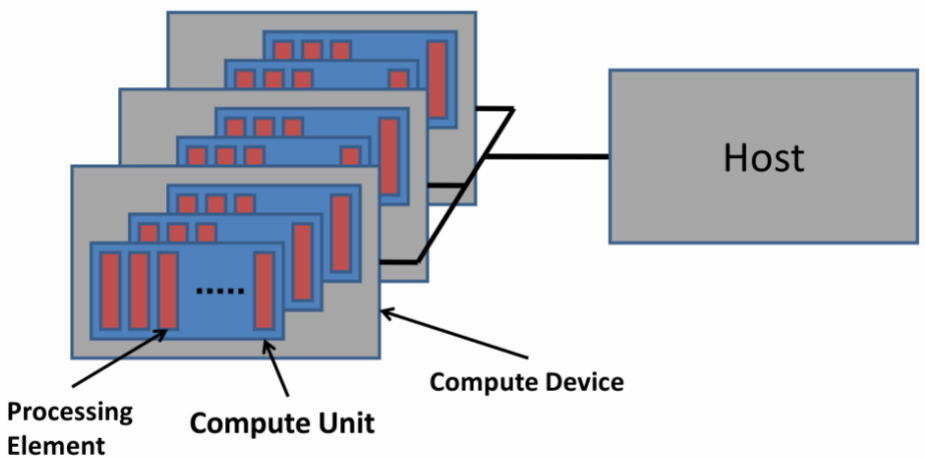


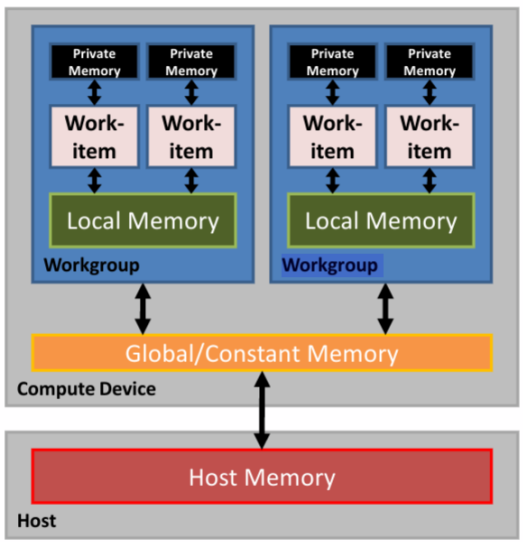

[Source: computing.llnl.gov/tutorials]

- 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

# 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
  - sends 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)

# NVIDIA Cuda

- Established by NVIDIA GPU vendor
- Works only on NVIDIA cards
- Claimed to provide optimal performance

# CUDA Kernel code

- The kernel code is the code to be executed on the GPU aka "Device"
- It needs to be compiled using special CUDA compiler

```
#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];
    }
}
```

# CUDA Host code I

```
int main(void)
{
    int numElements = 50000;
    size_t size = numElements * sizeof(float);

    // Allocate host vectors
    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);

    // 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 device vectors
    float *d_A = NULL;
    float *d_B = NULL;
    float *d_C = NULL;

    assert(cudaMalloc((void **)&d_A, size)==cudaSuccess);
    assert(cudaMalloc((void **)&d_B, size)==cudaSuccess);
    assert(cudaMalloc((void **)&d_C, size)==cudaSuccess);
    ...
```

# CUDA Host code II

```
    ...

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Launch the Vector Add CUDA Kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =(numElements + threadsPerBlock - 1)
                       / threadsPerBlock;

    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);

    assert(cudaGetLastError()==cudaSuccess);
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    free(h_A);
    free(h_B);
    free(h_C);
    cudaDeviceReset();
}
```

# OpenCL

- "Open Computing Language"
- Vendor independent
- More cumbersome to code

# 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

```
// 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

```
// 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 (Nvidia GPU ...)
- 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="-O`
- ..... but to do this one has to compile gcc with a special configuration...

# Other ways to program GPU

- Directly use graphics library
- OpenGL with shaders
- WebGL: OpenGL in the browser. Uses html and javascript.

# WebGL Example

- Gray-Scott model for Reaction-Diffusion: two 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 $U$ and 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 difference/finite volume discretization on grid of size $h$

$$-\Delta u \approx \frac{1}{h^2}(4u_{ij} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1})$$

# 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>
```

# 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)