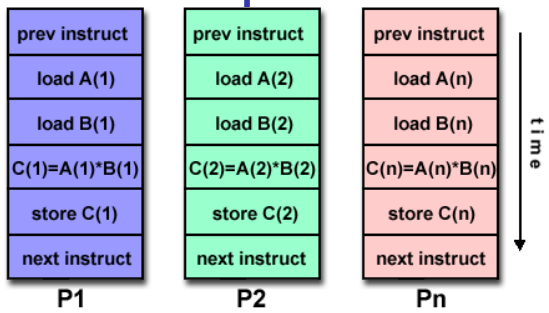
Scientific Computing WS 2018/2019

Lecture 25

Jürgen Fuhrmann

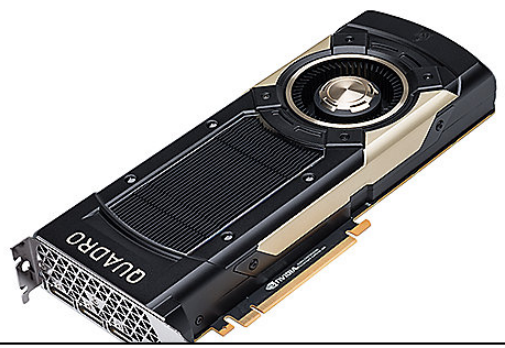
juergen.fuhrmann@wias-berlin.de

SIMD Hardware: Graphics Processing Units (GPU)



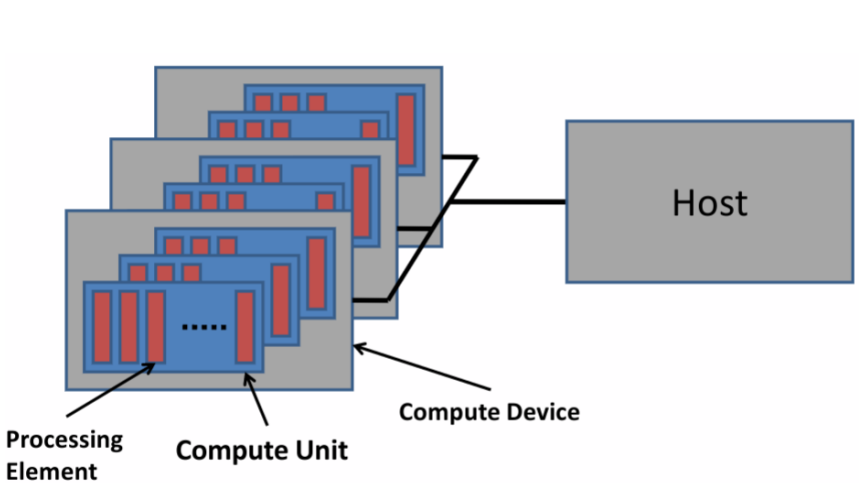
[Source: computing.llnl.gov/tutorials]

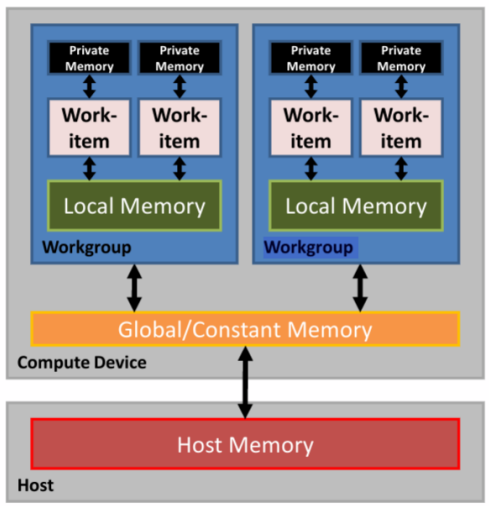
- Principle useful for highly structured data
- Example: textures, triangles for 3D graphis rendering
- During the 90's, *Graphics Processing Units* (GPUs) started to contain special purpose SIMD hardware for graphics rendering
- 3D Graphic APIs (DirectX, OpenGL) became transparent to programmers: rendering could be influenced 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 based on compiler directives
- GPGPUs are *accelerator cards* added to a computer with own memory, many vector processing pipelines and special bus interconnect (NVidia Quadro GV100: 32GB +5120 units, NVLink; Tensor cores)
- CPU-GPU connection via mainbord bus / special link





[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
  - Run the heavy duty loops in local memory
  - Send data back to host CPU
- For high performance one needs explicit management of these steps
- Bottleneck: Data transfer CPU $\leftrightarrow$ GPU
- 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.

Compiler directive based GPU programming

- OpenMP
  - OpenMP4.0
  - Implentation in commercial compilers
  - GCC, Clang implementations under development
- OpenACC
  - Idea similar to OpenMP: use compiler directives
  - Future merge with OpenMP initially intended, now they seem to be competitors
  - Intended for different accelerator types (Nvidia GPU ...)
  - Commercial compiler vendors, e.g. PGI (with free academic license valid one year)
  - GCC, Clang implementations under development

OpenACC code

- "Shader":

```
void vecaddgpu( float *restrict r, float *a, float *b, int n, int nrepeat)
{
  int irepeat;
  #pragma acc kernels loop present(r,a,b)
  for (irepeat=0;irepeat<nrepeat; irepeat++)
  for( int i = 0; i < n; ++i ) r[i] = a[i] + b[i] + irepeat;
}
```

- Invocation from CPU

```
a = (float*)malloc( n*sizeof(float) );
b = (float*)malloc( n*sizeof(float) );
r = (float*)malloc( n*sizeof(float) );
e = (float*)malloc( n*sizeof(float) );
#pragma acc data copyin(a[0:n],b[0:n]) copyout(r[0:n])
{
  vecaddgpu( r, a, b, n, nrepeat );
}
```

- Compile with PGI compiler (https://www.pgroup.com/)

```
pgcc -ta=tesla -fast -o add2 add2.c
```

Other ways to program GPU

- Directly use graphics library
- Modern 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
- No communication with CPU in the first place
- 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)

Simple iteration with preconditioning

Idea: $A\hat{u} = b \Rightarrow$

$$\hat{u} = \hat{u} - M^{-1}(A\hat{u} - b)$$

$\Rightarrow$ iterative scheme

$$u_{k+1} = u_k - M^{-1}(Au_k - b) \quad (k = 0, 1 \dots)$$

1. Choose initial value u_0, tolerance ε, set $k = 0$
2. Calculate *residuum* $r_k = Au_k - b$
3. Test convergence: if $||r_k|| < \varepsilon$ set $u = u_k$, finish
4. Calculate *update*: solve $Mv_k = r_k$
5. Update solution: $u_{k+1} = u_k - v_k$, set $k = i + 1$, repeat with step 2.

Convergence

- Let $\hat{u}$ be the solution of $Au = b$.
- Let $e_k = u_j - \hat{u}$ be the error of the k-th iteration step

$$\begin{aligned} u_{k+1} &= u_k - M^{-1}(Au_k - b) \\ &= (I - M^{-1}A)u_k + M^{-1}b \\ u_{k+1} - \hat{u} &= u_k - \hat{u} - M^{-1}(Au_k - A\hat{u}) \\ &= (I - M^{-1}A)(u_k - \hat{u}) \\ &= (I - M^{-1}A)^k(u_0 - \hat{u}) \end{aligned}$$

  resulting in

$$e_{k+1} = (I - M^{-1}A)^k e_0$$

- So when does $(I - M^{-1}A)^k$ converge to zero for $k \to \infty$?

Back to iterative methods

Sufficient condition for convergence: $\rho(I - M^{-1}A) < 1$.

Iterative solver complexity I

- Solve linear system iteratively until $||e_k|| = ||(I - M^{-1}A)^k e_0|| \leq \epsilon$

$$\rho^k e_0 \leq \epsilon$$
$$k \ln \rho < \ln \epsilon - \ln e_0$$
$$k \geq k_\rho = \left\lceil \frac{\ln e_0 - \ln \epsilon}{\ln \rho} \right\rceil$$

- $\Rightarrow$ we need at least k_ρ iteration steps to reach accuracy ϵ
- Optimal iterative solver complexity - assume:
  - $\rho < \rho_0 < 1$ independent of h resp. N
  - A sparse ($A \cdot u$ has complexity $O(N)$)
  - Solution of $Mv = r$ has complexity $O(N)$.

  $\Rightarrow$ Number of iteration steps k_ρ independent of N
  $\Rightarrow$ Overall complexity $O(N)$

Iterative solver complexity II

- Assume
  - $\rho = 1 - h^\delta \Rightarrow \ln \rho \approx -h^\delta \rightarrow k_\rho = O(h^{-\delta})$
  - d: space dimension $\Rightarrow h \approx N^{-\frac{1}{d}} \Rightarrow k_\rho = O(N^{\frac{\delta}{d}})$
  - $O(N)$ complexity of one iteration step (e.g. Jacobi, Gauss-Seidel)

  $\Rightarrow$ Overall complexity $O(N^{1+\frac{\delta}{d}}) = O(N^{\frac{d+\delta}{d}})$
- Jacobi: $\delta = 2$
- Hypothetical "Improved iterative solver" with $\delta = 1$?
- Overview on complexity estimates

dim	$\rho = 1 - O(h^2)$	$\rho = 1 - O(h)$	LU fact.	LU solve
1	$O(N^3)$	$O(N^2)$	$O(N)$	$O(N)$
2	$O(N^2)$	$O(N^{\frac{3}{2}})$	$O(N^{\frac{3}{2}})$	$O(N \log N)$
3	$O(N^{\frac{5}{3}})$	$O(N^{\frac{4}{3}})$	$O(N^2)$	$O(N^{\frac{4}{3}})$

Multigrid: Iterative solver with O(N) complexity

Idea: combine classical preconditioners with coarse grid correction

- Assume embedded finite element spaces $V_0 \dots V_l$ such tha $V_0 \subset V_1 \subset \dots V_l$
- V_k is produced from V_{k-1} by subdividing each triangle into four. Alternative: finite difference refinement
- $\Rightarrow$ interpolation operator $I_{k-1}^k : V_{k-1} \to V_k$
- $\Rightarrow$ restriction operator $R_{k-1}^k = (I_{k-1}^k)^T : V_k \to V_{k-1}$
- Discretization matrix A_k on each level $k = 0 \dots l$
- "Smoother" (Jacobi, ILU, ...) M_k on each level $k = 1 \dots l$
- Number of smoothig steps n_s
- Coarse grid solver
- Number of coarse grid correction steps γ

Multigrid Algorithm

Procedure `Multigrid`(l, u_l, f_l)

if $l = 0$ **then**

$u_0 = A_0^{-1} f_0$ `// coarse grid solution`

else

for $i = 1, n_s$ **do**

$u_l = u_l - M_l^{-1}(A_l u_l - f_l)$ `// pre-smoothing`

end

$f_{l-1} = R_{l-1}^{l}(A_l u_l - f_l)$ `// restriction`

$u_{l-1} = 0$

for $i = 1, \gamma$ **do**

`Multigrid`$(l-1, u_{l-1}, f_{l-1})$ `// coarse grid corr.`

end

$u_l = u_l - I_{l-1}^{l} u_{l-1}$ `// interpolation`

for $i = 1, n_s$ **do**

$u_l = u_l - M_l^{-1}(A_l u_l - f_l)$ `// post-smoothing`

end

end

end

Multigrid remarks

- $\gamma = 1 \Rightarrow$ V-Cycle, $\gamma = 2 \Rightarrow$ W-Cycle
- Use as a preconditioner in CG methods
- First development in early 60ies by Bakhvalov, Fedorenko
- Works well for hierarchically embedded grid systems and smooth problem coefficients: $O(N)$ solution complexity
- Other variant can use embedding of FEM spaces of growing polynomial degree
- "Algebraic multigrid": define coarse grid, interpolations in an algebraic way by choosing coarse grid points and an interpolation from matrix entries
- Hybrid variant: structured grid, matrix dependent transfer operators for problems with strongly varying coefficients (my PhD. thesis)

Examinations

Tue Feb 26.
Wed Feb 27.
Wed Mar 14.
Thu Mar 15.
Tue Mar 26.
Wed Mar 27.
Thu Mar 28.
Wed May 8 14:00-17:00

- 13:00 times do **not** work! Please reschedule (sorry).