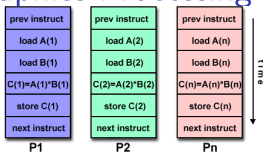
Scientific Computing WS 2017/2018

Lecture 28

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

SIMD Hardware: Graphics Processing Units (GPU)

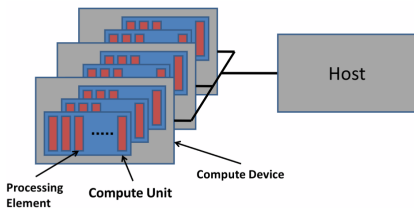


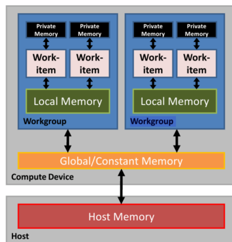
[Source: computing.llnl.gov/tutorials]

- 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 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="-O3`
- ...... but to do this one has to compile gcc with a special configuration...

Small Recap: iterative methods

Elements of iterative methods (Saad Ch.4)

Let $V = \mathbb{R}^n$ be equipped with the inner product $(\cdot, \cdot)$, let A be an $n \times n$ nonsingular matrix.

Solve $Au = b$ iteratively

- Preconditioner: a matrix $M \approx A$ "approximating" the matrix A but with the property that the system $Mv = f$ is easy to solve
- Iteration scheme: algorithmic sequence using M and A which updates the solution step by step

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.

The Jacobi method

- Let $A = D - E - F$, where D: main diagonal, E: negative lower triangular part F: negative upper triangular part
- Preconditioner: $M = D$, where D is the main diagonal of $A \Rightarrow$

$$u_{k+1,i} = u_{k,i} - \frac{1}{a_{ii}}\left(\sum_{j=1\dots n} a_{ij}u_{k,j} - b_i\right) \quad (i = 1\dots n)$$

- Equivalent to the succesive (row by row) solution of

$$a_{ii}u_{k+1,i} + \sum_{j=1\dots n, j\neq i} a_{ij}u_{k,j} = b_i \quad (i = 1\dots n)$$

- Already calculated results not taken into account
- Alternative formulation with $A = M - N$:

$$\begin{aligned} u_{k+1} &= D^{-1}(E+F)u_k + D^{-1}b \\ &= M^{-1}Nu_k + M^{-1}b \end{aligned}$$

- Variable ordering does not matter

Incomplete LU factorizations (ILU)

Idea (Varga, Buleev, 1960):

- fix a predefined zero pattern
- apply the standard LU factorization method, but calculate only those elements, which do not correspond to the given zero pattern
- Result: incomplete LU factors L, U, remainder R:

$$A = LU - R$$

- Problem: with complete LU factorization procedure, for any nonsingular matrix, the method is stable, i.e. zero pivots never occur. Is this true for the incomplete LU Factorization as well ?

ILU(0)

- Generally better convergence properties than Jacobi, Gauss-Seidel
- One can develop block variants
- Alternatives:
  - ILUM: ("modified"): add ignored off-diagonal entries to $\tilde{D}$
  - ILUT: zero pattern calculated dynamically based on drop tolerance
- Dependence on ordering
- Can be parallelized using graph coloring
- Not much theory: experiment for particular systems
- I recommend it as the default initial guess for a sensible preconditioner
- Incomplete Cholesky: symmetric variant of ILU

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 $k \to \infty$?

Spectral radius and convergence

Definition The spectral radius $\rho(A)$ is the largest absolute value of any eigenvalue of A: $\rho(A) = \max_{\lambda \in \sigma(A)} |\lambda|$.

Theorem (Saad, Th. 1.10) $\lim_{k\to\infty} A^k = 0 \Leftrightarrow \rho(A) < 1$.

Proof, $\Rightarrow$: Let u_i be a unit eigenvector associated with an eigenvalue λ_i. Then

$$
\begin{aligned}
Au_i &= \lambda_i u_i \\
A^2 u_i &= \lambda_i A_i u_i = \lambda^2 u_i \\
&\vdots \\
A^k u_i &= \lambda^k u_i \\
\text{therefore} \quad ||A^k u_i||_2 &= |\lambda^k| \\
\text{and} \quad \lim_{k\to\infty} |\lambda^k| &= 0
\end{aligned}
$$

so we must have $\rho(A) < 1$

Corollary from proof

Theorem (Saad, Th. 1.12)

$$\lim_{k\to\infty} ||A^k||^{\frac{1}{k}} = \rho(A)$$

□

Back to iterative methods

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

Matrix preconditioned Richardson iteration

M, A spd.

- Scaled Richardson iteration with preconditoner M

$$u_{k+1} = u_k - \alpha M^{-1}(Au_k - b)$$

- Spectral equivalence estimate

$$0 < \gamma_{min}(Mu, u) \leq (Au, u) \leq \gamma_{max}(Mu, u)$$

- $\Rightarrow \gamma_{min} \leq \lambda_i \leq \gamma_{max}$
- $\Rightarrow$ optimal parameter $\alpha = \frac{2}{\gamma_{max}+\gamma_{min}}$
- Convergence rate with optimal parameter: $\rho \leq \frac{\kappa(M^{-1}A)-1}{\kappa(M^{-1}A)+1}$
- This is one possible way for convergence analysis which at once gives convergence rates
- But ... how to obtain a good spectral estimate for a particular problem ?

Richardson for 1D heat conduction: Convergence factor

- Condition number + spectral radius

$$\kappa(M^{-1}A) = \kappa(A) = \frac{4(1+2h)^2}{\pi^2 h^2} - 1$$

$$\rho(I - M^{-1}A) = \frac{\kappa - 1}{\kappa + 1} = 1 - \frac{\pi^2 h^2}{2(1+2h)^2}$$

- Bad news: $\rho \to 1 \quad (h \to 0)$
- Typical situation with second order PDEs:

$$\kappa(A) = O(h^{-2}) \quad (h \to 0)$$

$$\rho(I - D^{-1}A) = 1 - O(h^2) \quad (h \to 0)$$

Iterative solver complexity I

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

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

- Assume $\rho < \rho_0 < 1$ independent of h resp. N, A sparse and 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 $\rho = 1 - h^\delta \Rightarrow \ln \rho \approx -h^\delta$
- $k = O(h^{-\delta})$
- d: space dimension, then $h \approx N^{-\frac{1}{d}} \Rightarrow k = O(N^{\frac{\delta}{d}})$
- Assume $O(N)$ complexity of one iteration step
  $\Rightarrow$ Overall complexity $O(N^{\frac{d+\delta}{d}})$
- Jacobi: $\delta = 2$, something better with at least $\delta = 1$?

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	$O(N^{\frac{5}{3}})$	$O(N^{\frac{4}{3}})$	$O(N^2)$	$O(N^{\frac{4}{3}})$

- In 1D, iteration makes not much sense
- In 2D, we can hope for parity
- In 3D, beat sparse matrix solvers with $\rho = 1 - O(h)$?

Conjugate gradients IV - The algorithm

Given initial value u_0, spd matrix A, right hand side b.

$$
\begin{aligned}
d_0 &= r_0 = b - Au_0 \\
\alpha_i &= \frac{(r_i, r_i)}{(Ad_i, d_i)} \\
u_{i+1} &= u_i + \alpha_i d_i \\
r_{i+1} &= r_i - \alpha_i A d_i \\
\beta_{i+1} &= \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)} \\
d_{i+1} &= r_{i+1} + \beta_{i+1} d_i
\end{aligned}
$$

At the i-th step, the algorithm yields the element from $e_0 + \mathcal{K}_i$ with the minimum energy error.

Theorem The convergence rate of the method is

$$
||e_i||_A \leq 2 \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^i ||e_0||_A
$$

where $\kappa = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}$ is the spectral condition number.

Preconditioning

Let M be spd, and spectrally equivalent to A, and assume that $\kappa(M^{-1}A) << \kappa(A)$.

Let E be such that $M = EE^T$, e.g. its Cholesky factorization. Then, $\sigma(M^{-1}A) = \sigma(E^{-1}AE^{-T})$:

Assume $M^{-1}Au = \lambda u$. We have

$$(E^{-1}AE^{-T})(E^T u) = (E^T E^{-T})E^{-1}Au = E^T M^{-1}Au = \lambda E^T u$$

$\Leftrightarrow E^T u$ is an eigenvector of $E^{-1}AE^{-T}$ with eigenvalue λ.

Preconditioned CG I

Now we can use the CG algorithm for the preconditioned system

$$E^{-1}AE^{-T}\tilde{x} = E^{-1}b$$

with $\tilde{u} = E^T u$

$$\begin{aligned}
\tilde{d}_0 = \tilde{r}_0 &= E^{-1}b - E^{-1}AE^{-T}u_0 \\
\alpha_i &= \frac{(\tilde{r}_i, \tilde{r}_i)}{(E^{-1}AE^{-T}\tilde{d}_i, \tilde{d}_i)} \\
\tilde{u}_{i+1} &= \tilde{u}_i + \alpha_i \tilde{d}_i \\
\tilde{r}_{i+1} &= \tilde{r}_i - \alpha_i E^{-1}AE^{-T}\tilde{d}_i \\
\beta_{i+1} &= \frac{(\tilde{r}_{i+1}, \tilde{r}_{i+1})}{(\tilde{r}_i, \tilde{r}_i)} \\
\tilde{d}_{i+1} &= \tilde{r}_{i+1} + \beta_{i+1}\tilde{d}_i
\end{aligned}$$

Not very practical as we need E

Preconditioned CG II

Assume $\tilde{r}_i = E^{-1} r_i$, $\tilde{d}_i = E^T d_i$, we get the equivalent algorithm

$$
\begin{aligned}
r_0 &= b - Au_0 \\
d_0 &= M^{-1} r_0 \\
\alpha_i &= \frac{(M^{-1} r_i, r_i)}{(Ad_i, d_i)} \\
u_{i+1} &= u_i + \alpha_i d_i \\
r_{i+1} &= r_i - \alpha_i A d_i \\
\beta_{i+1} &= \frac{(M^{-1} r_{i+1}, r_{i+1})}{(r_i, r_i)} \\
d_{i+1} &= M^{-1} r_{i+1} + \beta_{i+1} d_i
\end{aligned}
$$

It relies on the solution of the preconditioning system, the calculation of the matrix vector product and the calculation of the scalar product.

C++ implementation

```
template < class Matrix, class Vector, class Preconditioner, class Real >
int  CG(const Matrix &A, Vector &x, const Vector &b,
   const Preconditioner &M, int &max_iter, Real &tol)
{ Real resid;
  Vector p, z, q;
  Vector alpha(1), beta(1), rho(1), rho_1(1);
  Real normb = norm(b);
  Vector r = b - A*x;
  if (normb == 0.0)   normb = 1;
  if ((resid = norm(r) / normb) <= tol) {
    tol = resid;
    max_iter = 0;
    return 0;
  }
  for (int i = 1; i <= max_iter; i++) {
    z = M.solve(r);
    rho(0) = dot(r, z);
    if (i == 1)
      p = z;
    else {
      beta(0) = rho(0) / rho_1(0);
      p = z + beta(0) * p;
    }
    q = A*p;
    alpha(0) = rho(0) / dot(p, q);
    x += alpha(0) * p;
    r -= alpha(0) * q;
    if ((resid = norm(r) / normb) <= tol) {
      tol = resid;
      max_iter = i;
      return 0;
    }
    rho_1(0) = rho(0);
  }
  tol = resid;   return 1;
}
```

Iterative solution of 2D FEM Problems

- So far we used only sparse direct solver to solve 2D FEM problems
- Complexity estimates have been confirmed
- Try out Jacobi, ILU, CG
  - Estimate condition number from asymptotic convergence rate of iterative method for K large enugh

$$\rho(I - M^{-1}A) = \lim_{k\to\infty} ||(I - M^{-1}A)^k||^{\frac{1}{k}} \approx ||(I - M^{-1}A)^K||^{\frac{1}{K}}$$

$$\rho = \frac{\kappa(M^{-1}A) - 1}{\kappa(M^{-1}A) + 1}$$

$$\kappa(M^{-1}A) \approx \frac{1+\rho}{1-\rho}$$

  - For CG:

$$\kappa(M^{-1}A) \approx \left(\frac{1+\rho}{1-\rho}\right)^2$$