

## **GPULab Library - a High-Performance Library for PDE Solvers**

Glimberg, Stefan Lemvig

Publication date: 2011

Document Version Publisher's PDF, also known as Version of record

Link back to DTU Orbit

*Citation (APA):* Glimberg, S. L. (Author). (2011). GPULab Library - a High-Performance Library for PDE Solvers. Sound/Visual production (digital)

#### **General rights**

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.

- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

## GPULab Library - a High-Performance Library for PDE Solvers

#### Stefan L. Glimberg

Section of Scientific Computing Department of Informatics and Mathematical Modelling Technical University of Denmark

# CFD and Free-Surface Motion Workshop June 9<sup>th</sup>





< 17 >

000	00000000	00000000

## Outline



Programmable GPUs





## Who am I?

## Stefan L. Glimberg

- Master degree in Computer Science 2009 University of Copenhagen
- Thesis: Smoke Simulation for Fire Engineering using CUDA
- Started as a PhD student in 2010, DTU Section of Scientific Computing
- Research Project: Desktop Scientific Computing on Consumer Graphics Card
  - Subproject: Scientific GPU Computing for PDE Solvers



#### GPULab - http://gpulab.imm.dtu.dk/



The GPUlab is a competence center and laboratory for the use of Graphics Processing Units (GPUs) for visualization, scientific computations, and high-performance computing. The purpose is to attract focal interests in the use of GPUs by both engineering students and researchers in projects.

#### Projects

- Auto-tuning of Dense Linear Algebra on GPUs
- Accelerating Economic Model Predictive Control using GPUs
- Fast simulation of unsteady Nonlinear water waves
- and more ...
- Your project?

#### **GPULab Hardware**

New gear! Two powerful machines, each less than 3,000. The price for one GPU and one CPU is comparable, from 100 - 1,000.





## Graphical Processing Units

#### GPUs are

- massively parallel processors, capable of executing thousands of threads in parallel
- $\bullet$  available for the average consumer, with prices ranging from  $\sim$  \$100 \$1,000
- highly programmable, and not only for graphical purposes (CUDA / OpenCL)



## Why Even Bother?



Figure: Floating point operations per second. http://www.nvidia.com



< ∃⇒

・ロト ・回ト ・目

Introduction Pro	grammable GPUs
000 00	0000000

#### Why Even Bother?



Figure: Memory bandwidth for the CPU and GPU. http://www.nvidia.com



3.0

・ロト ・回ト ・目

Programmable GPUs
00000000

## Why Even Bother?

The future looks promising. However, notice the unit change on the y-axis.



Figure: Road map for the next years. http://www.nvidia.com



< □ > < 同 >

## CUDA Implementation

Implementing a simple CUDA program is not very difficult.

- Read the CUDA Programming Guide
- Localize parts in the code that can be parallelized (for-loops usually)
- Execute lots of threads, each processing one element

However, converting an entire solver is difficult, and it is even more difficult to get the highest possible performance.

#### Bottle Necks

Memory bound - The bandwidth (GB/s) between memory and chip is the bottle neck, e.g. BLAS 1 operations.

Compute bound - The clock frequency (GFLOPS) is the bottle neck, e.g. BLAS 3 operations.



1

2

3

4

5

6 7

8

9

## A Simple Example - square a vector of size N

#### Host (CPU):

```
void
square_host(float* in, float* out, int N)
{
    for(int i=0; i<N; ++i)
    {
        float val = in[i];
        out[i] = val*val;
    }
}</pre>
```

#### Device (GPU):

```
1 void __global__
2 square_device(float* in, float* out, int N)
3 {
4 int i = blockDim.x*blockIdx.x*threadIdx.x;
5 float val = in[i];
6 out[i] = val*val;
7 }
```



(日) (同) (日) (日)

## A Finite Difference Example

Based on Taylor series expansion we can derive a set of coefficients for calculating the derivative of u:

$$\frac{\partial u(x_i)}{\partial x} \approx \sum_{n=-\alpha}^{\beta} c_n u(x_{i+n})$$

If we set up a matrix based on finite difference coefficients we get

C00	o <i>c</i> <sub>01</sub>	<i>c</i> <sub>02</sub>	0	0	0	0	0	ר <i>u</i> 0 ר		$\begin{bmatrix} u'_0 \end{bmatrix}$
$c_{10}$	$c_{11}$	$c_{12}$	0	0	0	0	0	<i>u</i> <sub>1</sub>		$u_1^{}$
0	$c_{10}$	$c_{11}$	$c_{12}$	0	0	0	0	<i>u</i> <sub>2</sub>		$u_2^{\dagger}$
0	0	$c_{10}$	$c_{11}$	<i>c</i> <sub>12</sub>	0	0	0	<i>u</i> <sub>3</sub>	$\sim$	$u_3^T$
0	0	0	$c_{10}$	$c_{11}$	$c_{12}$	0	0	<i>u</i> <sub>4</sub>	$\sim$	$u_4^{\gamma}$
0	0	0	0	$c_{10}$	$c_{11}$	<i>c</i> <sub>12</sub>	0	u <sub>5</sub>		$u_5'$
0	0	0	0	0	$c_{10}$	$c_{11}$	<i>c</i> <sub>12</sub>	<i>u</i> <sub>6</sub>		$u_6^7$
LΟ	0	0	0	0	C <sub>20</sub>	C <sub>21</sub>	C22			$\begin{bmatrix} u_7^{i} \end{bmatrix}$

but there is a lot of repetitions in the matrix and it is very sparse.

DTII

## A Finite Difference Example (II)

So in compact form we only need

$$\mathbf{c} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}.$$
(1)

< □ > < 同 >

We call this the stencil.



It looks parallelizable !

GPULab Library - a High-Performance Library for PDE Solvers

Technical University of Denmark

1
2

3

4

5

6

7

8

1

2

3

4

5

6

7

8

## A Finite Difference Example (III)



#### Host version:

```
void finite_difference(float* out, float* in, float* stencil, int alpha, int N){
    for(int n=alpha; n<N-alpha; ++i){
      float sum = 0.f;
      for(int i=-alpha; i<=alpha; ++i)
           sum += stencil[alpha+i] * in[n+i];
           out[n] = sum;
    }
}</pre>
```

Device version:

```
__global__
void finite_difference(float* out, float* in, float* stencil, int alpha, int N){
    unsigned int idx = blockDim.x * blockIdx.x + threadIdx.x;
    float sum = 0.f;
    for(unsigned int i = -alpha; i<=alpha; ++i)
        sum += stencil[alpha+i] * in[idx+i];
    out[idx] = sum;
}
```

There are still some tweaking to do.

## A Finite Difference Example (IV)

#### Performance results for CPU and GPU implementations.



Figure: Timings for a vector with 1,000,000 elements. Using a Tesla C1070 GPU and an Intel Core i7 @ 1.73GHz CPU.



< □ > < 同 >

## A GPU-based Framework for PDE Solvers

So now I finally arrived at what I really want to talk about:

- The GPULab library



#### Objective

Remove all nonsense for the non-expert GPU programmers - put it into a highly generic framework.

There has been a tendency to wrap GPU techniques onto an existing CPU solver, instead of using an existing GPU framework to solve the same problem.

#### Key components for High-Performance PDE solvers

- Stencil based flexible order FD operations
- Iterative methods for solving large systems of eqs. (mixed precision)
- Domain decomposition techniques

#### Component-based Framework

We have decided to invest time now to develop a generic framework, in order to easily solve a broad range of PDE problems in the future. (Inspired by the PETSc framework).

Generic vector and matrix classes will be the backbone for all algorithms.

```
1
     const int I = 100:
2
     // Create and allocate some host vectors
3
     gpulab::vector<float.host memorv> x h(I.3.f);
4
     gpulab::vector<float.host memory> v h(I.2.f);
5
    // Do v = a*x+v on the host
6
    v h.axpv(4.f.x h):
7
8
     // Create and allocate some device vectors
q
     gpulab::vector<float.device memorv> x d(I.3.f);
     gpulab::vector<float,device_memory> y_d(I,2.f);
10
    // Do v = a * x + v on the device
11
12
    v d.axpv(4.f.x d):
```

Ideas are based on the Thrust and CUSP libraries, in fact we inherit from the Thrust vector classes.



## Component-based Framework (II)

**An example:** The Defect Correction method with a multigrid preconditioner is the backbone of our nonlinear water wave solver.

## Defect Correction algorithm

**Algorithm:** Defect Correction Method for approximate solution of Ax = bChoose  $x^{[0]}$ 1 /\* initial guess \*/ 2 k = 03 Repeat  $r^{[k]} = b - A x^{[k]}$ 4 /\* high order defect \*/ 5 Solve  $M\delta^{[k]} = r^{[k]}$ /\* preconditioner \*/  $x^{[k+1]} = x^{[k]} + \delta^{[k]}$ 6 /\* defect correction \*/ 7 k = k + 18 **Until** convergence or  $k > k_{max}$ 



## Component-based Framework (III)

The implementation is generic and simple!

```
1
     template <typename V, typename M, typename P>
 2
     void defect correction (M const& A, V& x, V const& b, P& precond, monitor <typename V::value type>
             & m)
 3
 4
       m.reset iteration count():
 5
       // Allocate space for residual and delta x
       V r(x.size());
 6
 7
       V d(x.size()):
 8
       while(1)
 q
10
         A.mult(x.r):
         r.axpby(1, -1, b);
11
12
         // Close enough to stop
13
         if(m.finished(r))
14
           break:
15
         // Solve using pre-conditioner
         precond(A,d,r);
16
17
         // Update solution
         x.axpy(1,d);
18
19
         // Next iteration
20
         ++m:
21
       7
22
     3
```



## Component-based Framework (IV)

Defect correction results for 100 iterations with a dense Jacobi preconditioner.





A B > A B > A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A

#### Ideas for the Future

In time we want to assemble our PDE solvers from building blocks (components), such that it is easy to change parts.

```
typedef gpulab::vector<float,device_memory> vector_type;
1
2
     typedef gpulab::FD::stencil<float>
                                                   matrix_type;
3
4
     typedef gpulab::solvers::multigrid_types<
5
       . vector type
                                                  // Vector type
       , matrix_type
                                                  // Matrix type
6
7
       , gpulab::solvers::jacobi
                                                  // Preconditioner
8
       . gpulab::solvers::grid handler 3d
                                                  // Grid handler
q
    > mg_types;
10
11
     typedef gpulab::solvers::dc_types<
12
       , vector_type
                                                  // Vector type
13
       , matrix_type
                                                  // Matrix type
                                                  // Preconditioner
14
       . gpulab::solvers::multigrid<mg types>
15
    > dc types:
16
     typedef gpulab::solvers::my_solver_types <
18
       , vector_type
                                                   // Vector type
19
       , matrix_type
                                                   // Matrix type
20
       , gpulab::solvers::dc<dc_types>
                                                   // Solver
       . gpulab::integration::ERK4<vector type> // Time integrator
21
22
    > mv solver types:
23
24
    // In our program we write
25
     gpulab::solvers::my_solver<my_solver_types> s(...); // Init solver
26
     s.take_step(dt);
                                                           // Take time step
```



<ロ> <問> <問> < 思> < 思>

#### Future Work

We are working with the OceanWave3D model for coastal and offshore engineering.

We want to solve large problems - fast!

- $\bullet\,$  Currently we are limited by the GPU memory  $\sim\,6GB$   $\rightarrow\,75M$  dof.
- We want to be limited by the total number of GPUs
- Solving on multiple GPUs on multiple workstations
- Linear scaling of time vs GPUs





## That's it ...

# Thank you !



Image: A math a math



GPULab Library - a High-Performance Library for PDE Solvers

Technical University of Denmark

-