GPULab Library - a High-Performance Library for PDE Solvers

Glimberg, Stefan Lemvig

Publication date:
2011

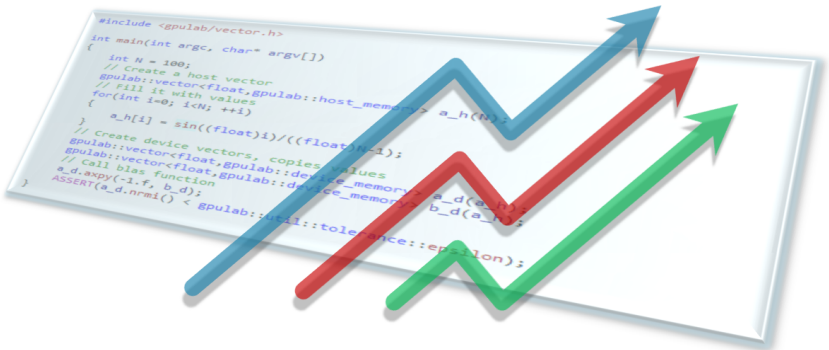
Document Version
Publisher's PDF, also known as Version of record

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^{th}



Outline

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/

GPU-Lab
DTU Informatics

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.

(a) Our new workstation

(b) One Tesla C1060 GPU

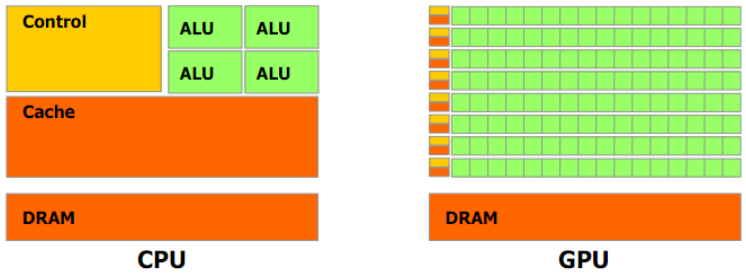
Graphical Processing Units

GPUs are

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

(c) A Fermi GPU

(d) The GPU chip



(e) Transistors on CPU/GPU

Why Even Bother?

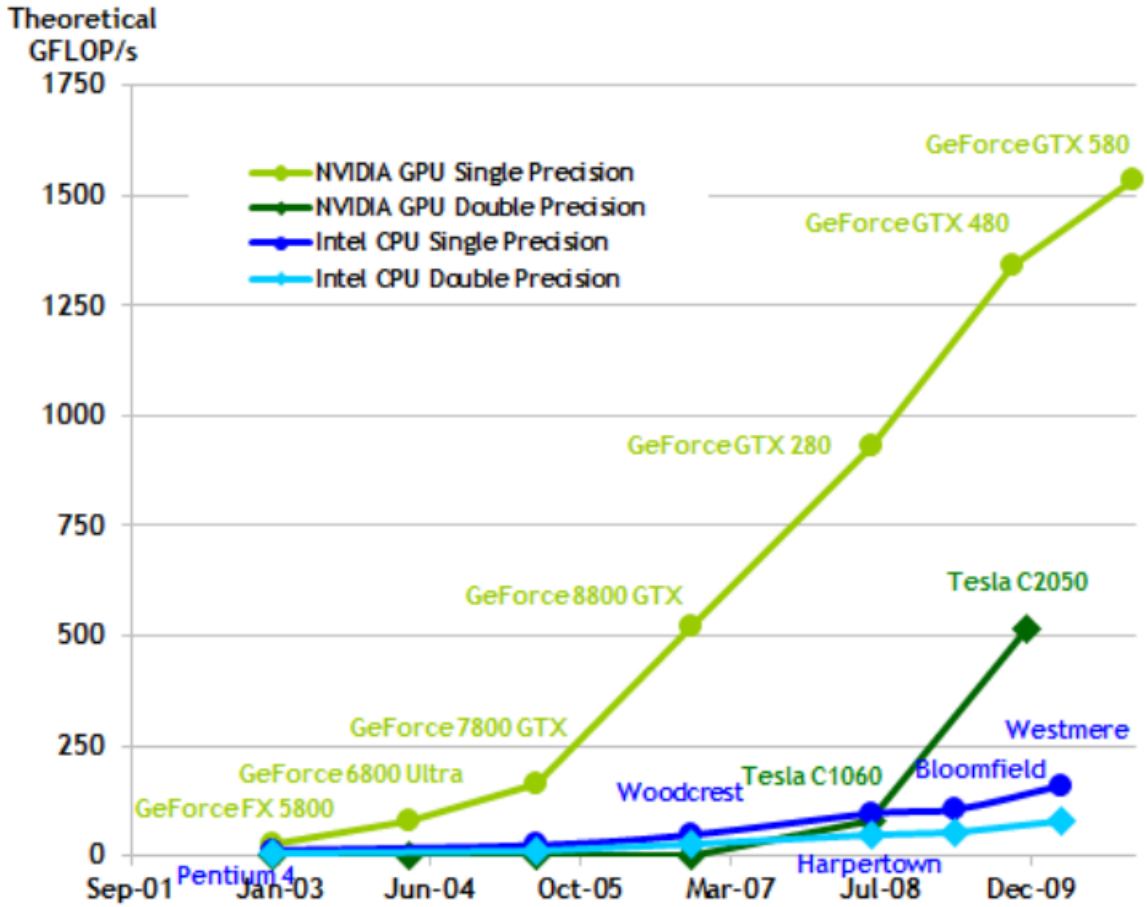


Figure: Floating point operations per second. http://www.nvidia.com

Why Even Bother?

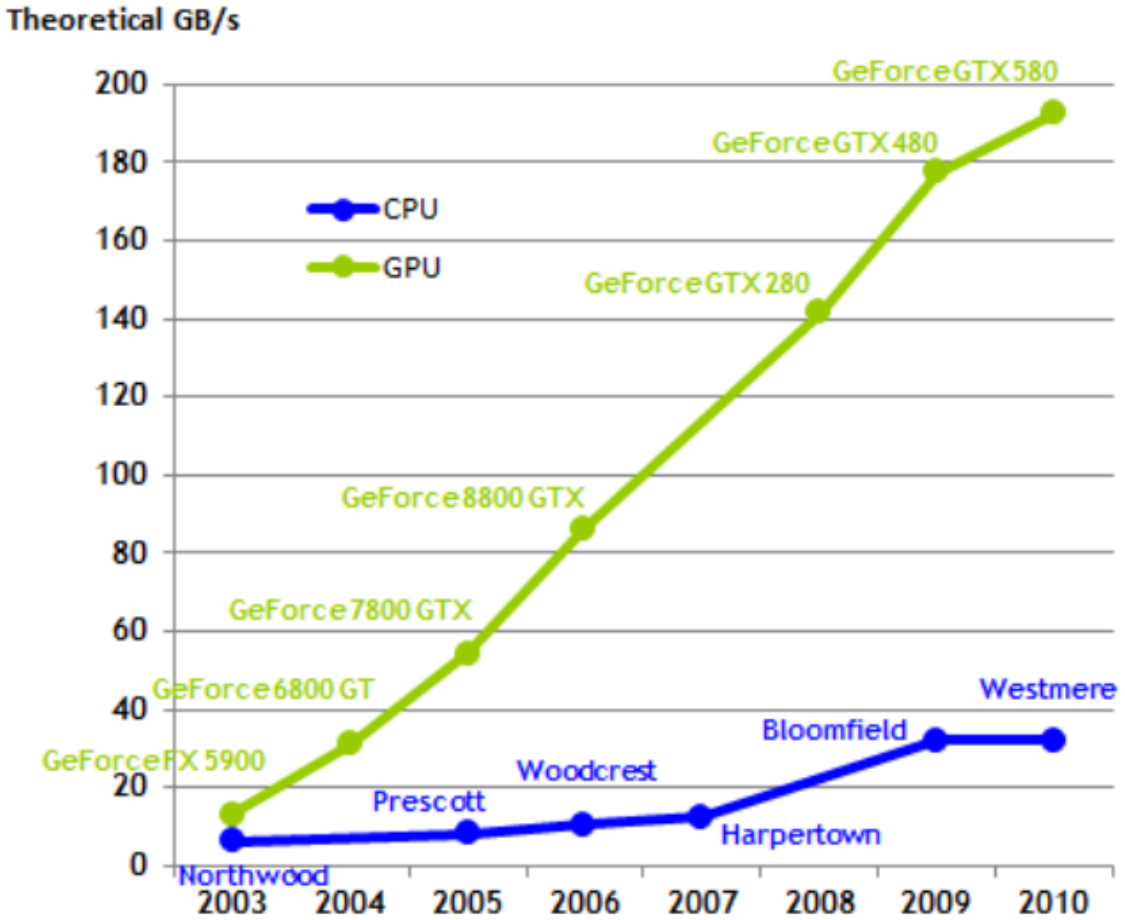


Figure: Memory bandwidth for the CPU and GPU. http://www.nvidia.com

Why Even Bother?

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

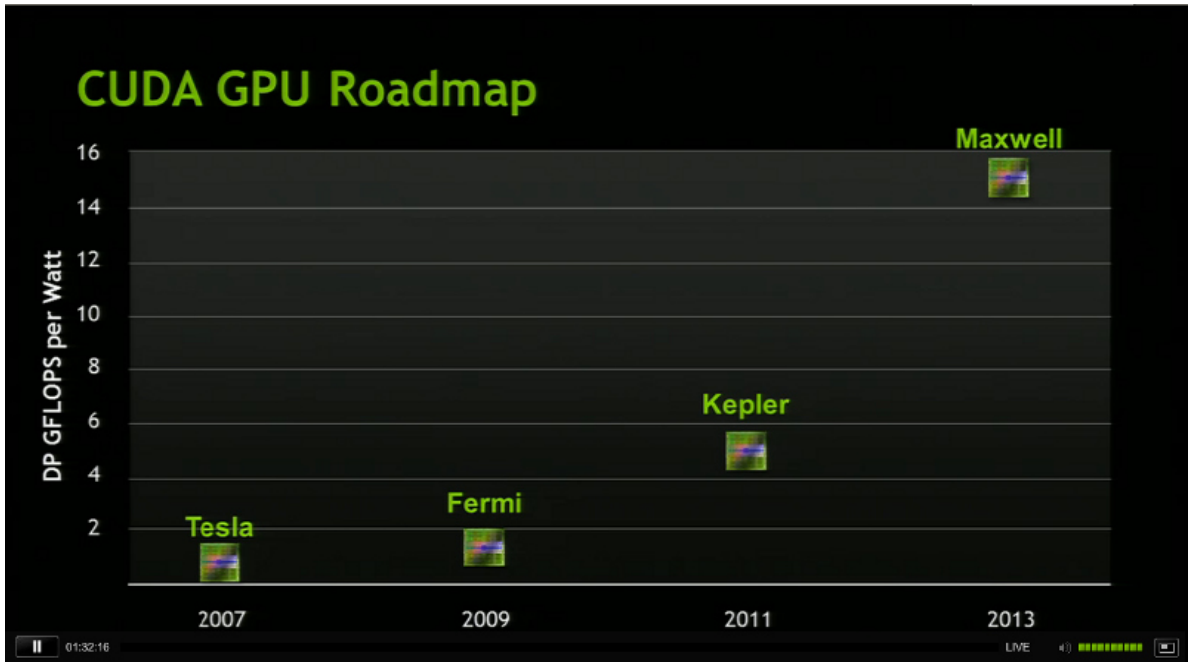


Figure: Road map for the next years. http://www.nvidia.com

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.

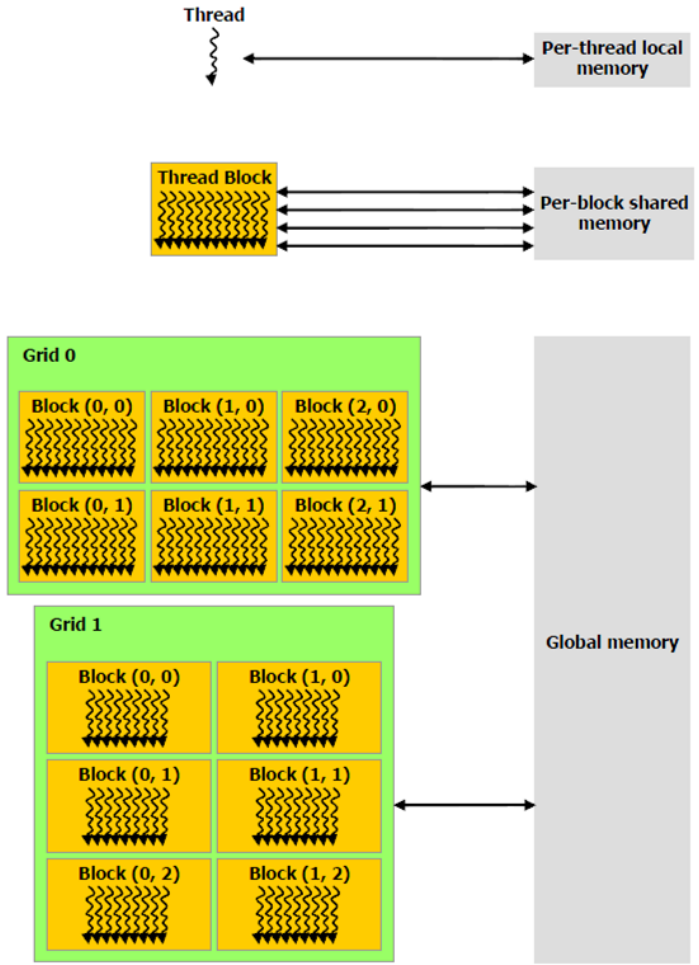
A Simple Example - square a vector of size N

Host (CPU):

```
1  void
2  square_host(float* in, float* out, int N)
3  {
4    for(int i=0; i<N; ++i)
5    {
6      float val = in[i];
7      out[i] = val*val;
8    }
9  }
```

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  }
```



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

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} & 0 & 0 & 0 & 0 & 0 \\ c_{10} & c_{11} & c_{12} & 0 & 0 & 0 & 0 & 0 \\ 0 & c_{10} & c_{11} & c_{12} & 0 & 0 & 0 & 0 \\ 0 & 0 & c_{10} & c_{11} & c_{12} & 0 & 0 & 0 \\ 0 & 0 & 0 & c_{10} & c_{11} & c_{12} & 0 & 0 \\ 0 & 0 & 0 & 0 & c_{10} & c_{11} & c_{12} & 0 \\ 0 & 0 & 0 & 0 & 0 & c_{10} & c_{11} & c_{12} \\ 0 & 0 & 0 & 0 & 0 & c_{20} & c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{bmatrix} \approx \begin{bmatrix} u'_0 \\ u'_1 \\ u'_2 \\ u'_3 \\ u'_4 \\ u'_5 \\ u'_6 \\ u'_7 \end{bmatrix}$$

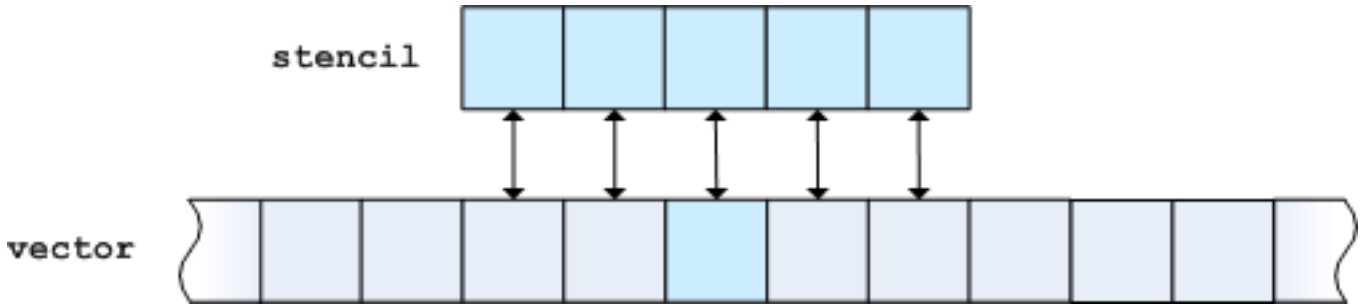
but there is a lot of repetitions in the matrix and it is very sparse.

A Finite Difference Example (II)

So in compact form we only need

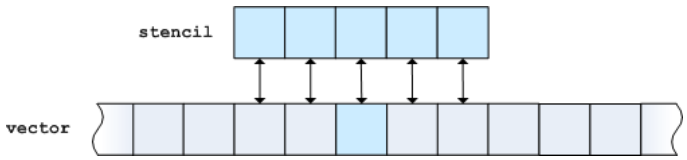
$$\mathbf{c} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}. \quad (1)$$

We call this the stencil.



It looks parallelizable !

A Finite Difference Example (III)



Host version:

```
1  void finite_difference(float* out, float* in, float* stencil, int alpha, int N){
2      for(int n=alpha; n<N-alpha; ++i){
3          float sum = 0.f;
4          for(int i=-alpha; i<=alpha; ++i)
5              sum += stencil[alpha+i] * in[n+i];
6          out[n] = sum;
7      }
8  }
```

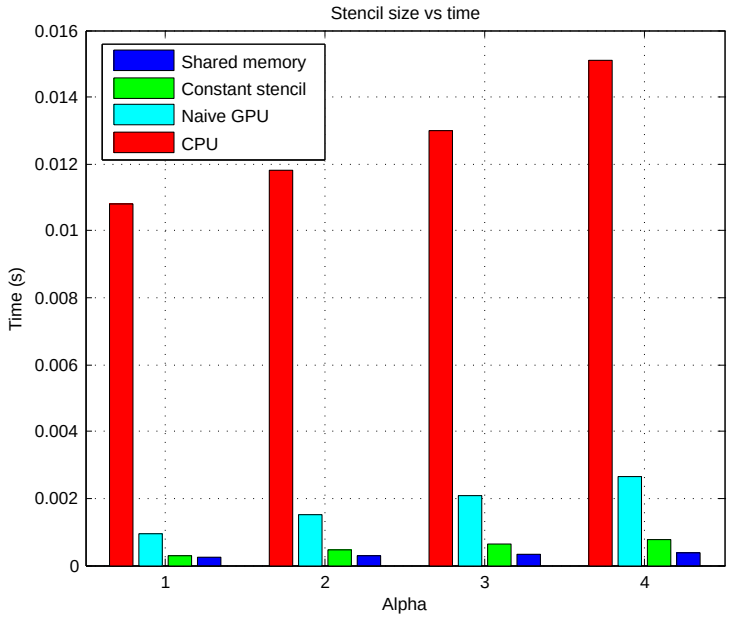
Device version:

```
1  __global__
2  void finite_difference(float* out, float* in, float* stencil, int alpha, int N){
3      unsigned int idx = blockDim.x * blockIdx.x + threadIdx.x;
4      float sum = 0.f;
5      for(unsigned int i = -alpha; i<=alpha; ++i)
6          sum += stencil[alpha+i] * in[idx+i];
7      out[idx] = sum;
8  }
```

There are still some tweaking to do.

A Finite Difference Example (IV)

Performance results for CPU and GPU implementations.



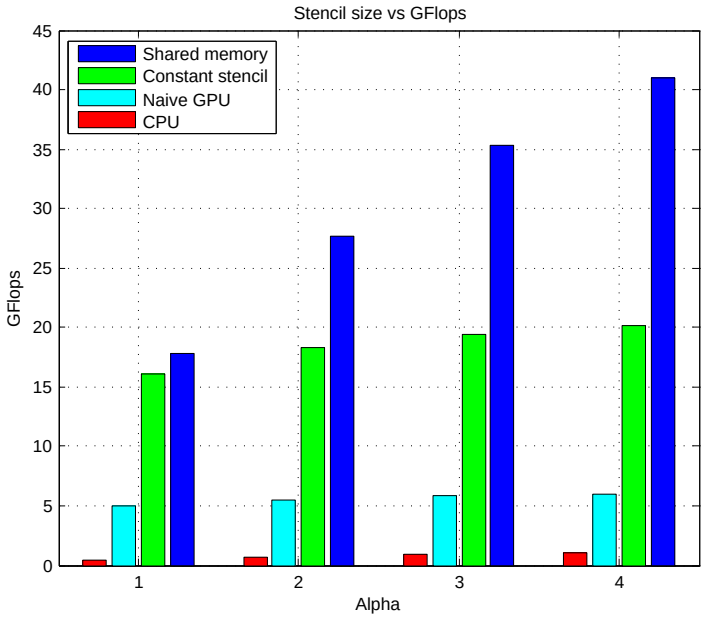
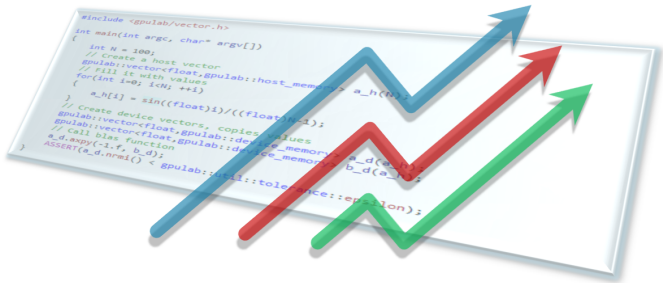


Figure: Timings for a vector with 1,000,000 elements. Using a Tesla C1070 GPU and an Intel Core i7 @ 1.73GHz CPU.

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_memory>  x_h(I,3.f);
4  gpulab::vector<float,host_memory>  y_h(I,2.f);
5  // Do y = a*x+y on the host
6  y_h.axpy(4.f,x_h);
7
8  // Create and allocate some device vectors
9  gpulab::vector<float,device_memory>  x_d(I,3.f);
10 gpulab::vector<float,device_memory>  y_d(I,2.f);
11 // Do y = a*x+y on the device
12 y_d.axpy(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 $Ax = b$

1	Choose $x^{[0]}$	/* initial guess */
2	$k = 0$	
3	**Repeat**	
4	$r^{[k]} = b - Ax^{[k]}$	/* high order defect */
5	Solve $M\delta^{[k]} = r^{[k]}$	/* preconditioner */
6	$x^{[k+1]} = x^{[k]} + \delta^{[k]}$	/* defect correction */
7	$k = k + 1$	
8	**Until** convergence or $k > k_{max}$	

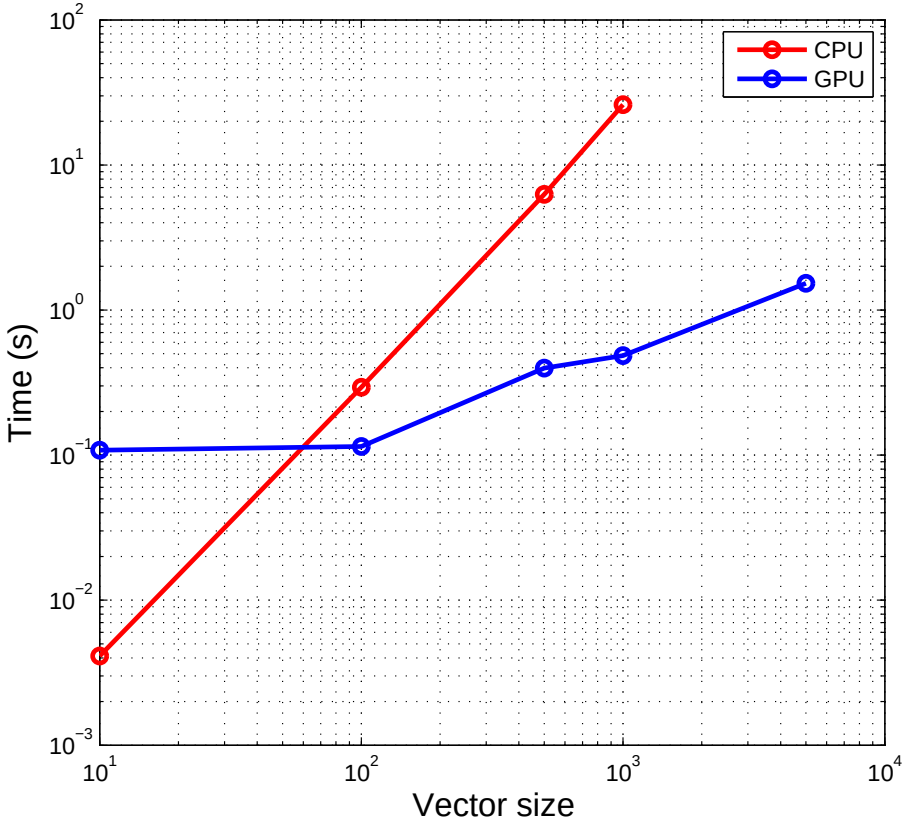
Component-based Framework (III)

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
6    V r(x.size());
7    V d(x.size());
8    while(1)
9    {
10     A.mult(x,r);
11     r.axpby(1, -1, b);
12     // Close enough to stop
13     if(m.finished(r))
14       break;
15     // Solve using pre-conditioner
16     precond(A,d,r);
17     // Update solution
18     x.axpy(1,d);
19     // Next iteration
20     ++m;
21   }
22 }
```

Component-based Framework (IV)

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



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.

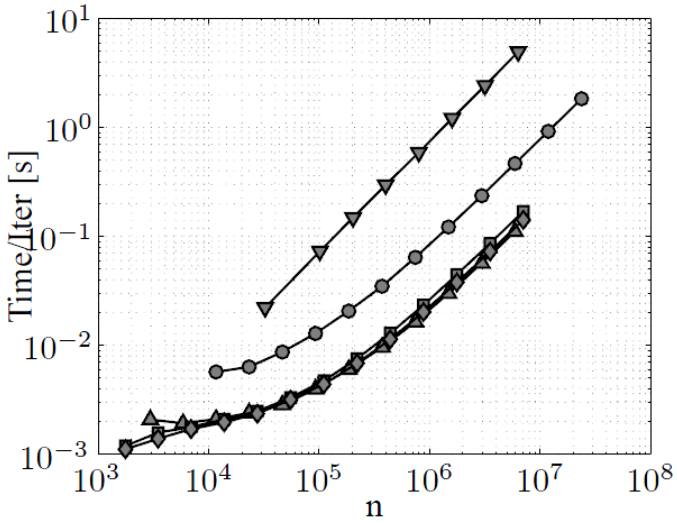
```
typedef gpulab::vector<float,device_memory> vector_type;
typedef gpulab::FD::stencil<float>          matrix_type;

typedef gpulab::solvers::multigrid_types<
  , vector_type                             // Vector type
  , matrix_type                             // Matrix type
  , gpulab::solvers::jacobi                 // Preconditioner
  , gpulab::solvers::grid_handler_3d        // Grid handler
> mg_types;

typedef gpulab::solvers::dc_types<
  , vector_type                             // Vector type
  , matrix_type                             // Matrix type
  , gpulab::solvers::multigrid<mg_types>    // Preconditioner
> dc_types;

typedef gpulab::solvers::my_solver_types<
  , vector_type                              // Vector type
  , matrix_type                              // Matrix type
  , gpulab::solvers::dc<dc_types>            // Solver
  , gpulab::integration::ERK4<vector_type>   // Time integrator
> my_solver_types;

// In our program we write
gpulab::solvers::my_solver<my_solver_types> s(...); // Init solver
s.take_step(dt);                                    // Take time step
```

Future Work

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

We want to solve large problems - **fast!**

- 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



(a) Absolute timings.

That's it ...

Thank you !