



A High Performance GPU-based Framework for PDE Prototyping

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). A High Performance GPU-based Framework for PDE Prototyping. Sound/Visual production (digital) <http://gpulab.imm.dtu.dk/courses.html>

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.

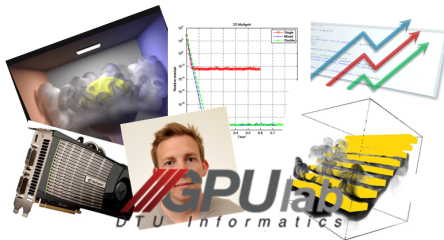
- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in 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.

Who am I?

Stefan L. Glimberg - Part of GPUlab

- Master degree in Computer Science 2009 - University of Copenhagen
 - Thesis: *Smoke Simulation for Fire Engineering using CUDA*
- PhD student, started 2010, DTU - Section of Scientific Computing
 - Project: *Scientific GPU Computing for PDE Solvers*



CUDA Implementation

Implementing a simple CUDA program is not very difficult.

- 1 Read the CUDA Programming Guide
- 2 Localize parts in the code that can be parallelized
- 3 Execute a lot of threads, each processing one element

CUDA Implementation

Implementing a simple CUDA program is not very difficult.

- 1 Read the CUDA Programming Guide
- 2 Localize parts in the code that can be parallelized
- 3 Execute a lot of threads, each processing one element

BLAS1 Example: $y = ax + y$

Host (CPU):

```
1 void
2 axpy_host(float a, float* x, float
   * y, int N)
3 {
4     for(int i=0; i<N; ++i)
5     {
6         y[i] = a*x[i] + y[i];
7     }
8 }
```

CUDA Implementation

Implementing a simple CUDA program is not very difficult.

- 1 Read the CUDA Programming Guide
- 2 Localize parts in the code that can be parallelized
- 3 Execute a lot of threads, each processing one element

BLAS1 Example: $y = ax + y$

Host (CPU):

```
1 void
2 axpy_host(float a, float* x, float
   * y, int N)
3 {
4     for(int i=0; i<N; ++i)
5     {
6         y[i] = a*x[i] + y[i];
7     }
8 }
```

Device (GPU):

```
1 __global__ void
2 axpy_device(float a, float* x,
   float* y, int N)
3 {
4     int i = blockDim.x*blockIdx.x+
       threadIdx.x;
5     y[i] = a*x[i] + y[i];
6 }
```

CUDA Implementation

Implementing a simple CUDA program is not very difficult.

- 1 Read the CUDA Programming Guide
- 2 Localize parts in the code that can be parallelized
- 3 Execute a lot of threads, each processing one element

BLAS1 Example: $y = ax + y$

Host (CPU):

```
1 void
2 axpy_host(float a, float* x, float
   * y, int N)
3 {
4     for(int i=0; i<N; ++i)
5     {
6         y[i] = a*x[i] + y[i];
7     }
8 }
```

Better one:

```
1 template <typename T>
2 __global__ void
3 axpy_device(T a, T* x, T* y, int N
   )
4 {
5     int i = blockDim.x*blockIdx.x+
        threadIdx.x;
6     y[i] = a*x[i] + y[i];
7 }
```


CUDA Implementation

Implementing a simple CUDA program is not very difficult.

- 1 Read the CUDA Programming Guide
- 2 Localize parts in the code that can be parallelized
- 3 Execute a lot of threads, each processing one element

BLAS1 Example: $y = ax + y$

Host (CPU):

```
1 void
2 axpy_host(float a, float* x, float
   * y, int N)
3 {
4     for(int i=0; i<N; ++i)
5     {
6         y[i] = a*x[i] + y[i];
7     }
8 }
```

Better one:

```
1 template <typename T>
2 __global__ void
3 axpy_device(T a, T* x, T* y, int N
   )
4 {
5     int i = blockDim.x*blockIdx.x+
        threadIdx.x;
6     y[i] = a*x[i] + y[i];
7 }
```

However, converting entire solvers for engineering applications is difficult, and it is even more difficult to get the best possible performance.

A GPU-based Framework for PDE Solvers

Why not put all this into a framework

→ The GPULab library



Objective

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

Avoid wrapping GPU codes onto an existing CPU solver! Instead, use an existing GPU framework to solve the same problem.

Key components for our High-Performance PDE library

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

Framework Outline

Generic vector and matrix classes is the backbone for most of our algorithms. Template specializations take care of dispatching.

Examples:

```

1  const int I = 100;
2  gpulab::vector<float,host_memory>  x_h(I,3.f);  // Create host vector x
3  gpulab::vector<float,host_memory>  y_h(I,2.f);  // Create host vector y
4  y_h.axy(4.f,x_h);                               // Do y = a*x+y on the host
5
6  gpulab::vector<float,device_memory> x_d(x_h);   // Create device vector x (from host)
7  gpulab::vector<float,device_memory> y_d(y_h);   // Create device vector y (from host)
8  y_d.axy(4.f,x_d);                               // Do y = a*x+y on the device
9
10 gpulab::matrix<float,device_memory> A_d(I,I);   // Create a dense matrix
11 A_d.diag(2.f);                                   // Set diagonal elements
12 A_d(2,3) = 3.f;                                  // Set specific element
13
14 gpulab::solvers::cg(A_d,x_d,b_d);                // Solve Ax = y using Conjugate Gradient
15 gpulab::solvers::gmres(A_d,x_d,b_d);             // Solve Ax = y using GMRES
16
17 gpulab::io::print(x_d,gpulab::io::T0_TEXT_FILE);// Print result

```

Ideas are based on the C++ standard library, Thrust, and CUSP that exists for GPUs.

Framework Outline (II)

Assembling a linear equation solver from a textbook recipe:

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  $k > k_{max}$ 
```

The Defect Correction method with a multigrid preconditioner is the backbone of our free surface solver.

Framework Outline (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& precondition, monitor<typename V::value_type>
   & m)
3  {
4      m.reset_iteration_count();
5      // Allocate space for residual and delta
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         precondition(A,d,r);
17         // Update solution
18         x.axpy(1,d);
19         // Next iteration
20         ++m;
21     }
22 }
```

Framework Outline (IV)

Defect correction results for 100 iterations with a Jacobi preconditioner. It is easy to compare host/device code.

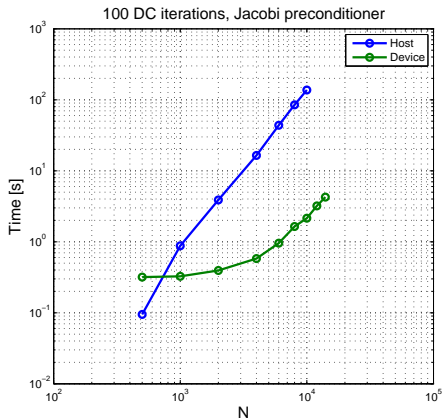


Figure: $N = 10,000$: 2 sec vs 2 min. NVIDIA Quadro FX 880M vs Intel i7 @1.73GHz.

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

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

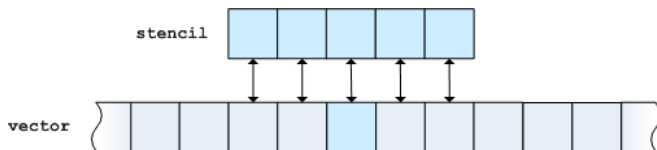
but there is a lot of repetitions in the matrix and it is very sparse.

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.

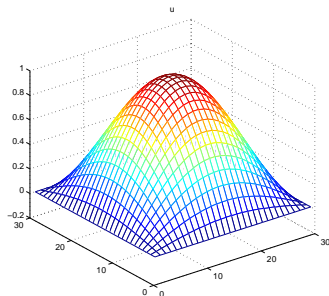


It is parallelizable!

These stencil operations are implemented into matrix-like classes.

2D Poisson Problem

We implemented these stencil operations into matrix-like classes. Here is an example of a 2D Poisson equation and a possible solution:



$$\begin{aligned}\partial_{xx}u + \partial_{yy}u &= f, & (x,y) \in \Omega([0,1]^2) \\ u &= 0, & (x,y) \in \partial\Omega\end{aligned}$$

2D Poisson Problem (II)

```

1  typedef gpulab::device_memory memory_space;      // Use host/device memory
2
3  // Setup grid and domain
4  gpulab::grid_dim<int> dim(100,100);              // 100x100 grid
5  gpulab::grid_dim<double> phys0(0.,0.);            // Domain starts in x=0, y=0
6  gpulab::grid_dim<double> phys1(1.,1.);            // Domain end in x=1, y=1
7  gpulab::grid_properties<int,double> props(dim, phys0, phys1);
8
9  gpulab::grid<double,memory_space> u(props);       // Create u
10 gpulab::grid<double,memory_space> f(props);       // Create f
11
12 // Create the stencil operator (implicit matrix)
13 gpulab::FD::stencil_2d<double> A(2,4);           // Second order derivative, fourth order accuracy
14
15 A.mult(u,f);                                       // Calculate f = du/dxx + du/dyy
16
17 gpulab::monitor m(iter,rtol,atol);                // Stopping criteria
18 gpulab::solvers::cg(A,u,f,m);                     // Solve Au = f using Conjugate Gradient
19
20 // Test for convergence
21 if(m.converged())
22     printf("Converged in %d iterations\n", m.iteration_count());

```

Stencil Performance

Performance results for computing $\partial_x u$ on the CPU and GPU.

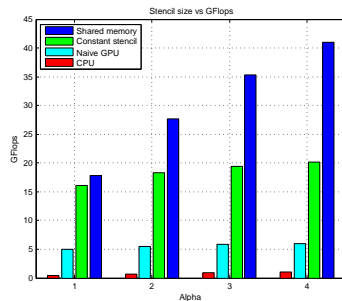
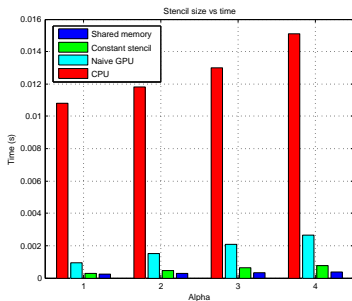


Figure: Timings for 1,000,000 elements. Tesla C1070 GPU and an Intel Core i7 @ 1.73GHz CPU.

PDE Assembling - Work in Progress

We want to assemble the PDE solvers from building blocks (components), such that it is easy to change parts, employ mixed precision etc.

```

1  typedef gpulab::grid<float,device_memory>    vector_type;
2  typedef gpulab::FD::stencil_2d<float>        matrix_type;
3
4  typedef gpulab::solvers::multigrid_types<
5      , vector_type                                // Vector type
6      , matrix_type                               // Matrix type
7      , gpulab::solvers::jacobi_2d                // Preconditioner
8      , gpulab::solvers::grid_handler_3d         // Grid handler
9  > mg_types;
10
11 typedef gpulab::solvers::dc_types<
12     , vector_type                                // Vector type
13     , matrix_type                               // Matrix type
14     , gpulab::solvers::multigrid<mg_types>      // Preconditioner
15 > dc_types;
16
17 typedef gpulab::solvers::free_surface_solver_types<
18     , vector_type                                // Vector type
19     , matrix_type                               // Matrix type
20     , gpulab::solvers::dc<dc_types>             // Solver
21     , gpulab::integration::ERK4                // Time integrator
22 > solver_types;
23
24 // In our program we write
25 gpulab::solvers::free_surface_solver<solver_types> s(...); // Init solver
26 s.take_step(dt);                                           // Take time step

```

That's it ...

Thank you !