Tutorial

E. Ketelaer, D. Lukarski

Multi-platform Linear Algebra

Example for using GPUs

modified on July 4, 2014

Version 1.3

Contents

Using GPU in HiFlow^3 Simulations

1 Introduction

HiFlow^3 is a multi-purpose finite element software providing powerful tools for efficient and accurate solution of a wide range of problems modeled by partial differential equations (PDEs). Based on object-oriented concepts and the full capabilities of C++ the HiFlow^3 project follows a modular and generic approach for building efficient parallel numerical solvers. It provides highly capable modules dealing with the mesh setup, finite element spaces, degrees of freedom, linear algebra routines, numerical solvers, and output data for visualization. Parallelism - as the basis for high performance simulations on modern computing systems - is introduced on two levels: coarse-grained parallelism by means of distributed grids and distributed data structures, and fine-grained parallelism by means of platform-optimized linear algebra back-ends.

1.1 How to Use the Tutorial?

You find the example code (convdiff_benchmark.cc, convdiff_benchmark.h), a parameter file for the first gpu numerical example (convdiff_benchmark.xml).The geometry data (*.inp, *.vtu) is stored in the directory `/hiflow/examples/data`.

1.2 Using HiFlow^3

Compile HiFlow^3 with the help of cmake – follow the first steps in installation guide from the web site. To use the NVIDIA GPU [3], you need to enable the CUDA [2] backend (`WANT_CUDA` to `ON`) or to enable the OpenCL [1] backend (`WANT_OPENCL` to `ON`). For some versions of the CUDA and OpenCL you might need to adapt or to update the cmake check files which are located in directory `hiflow/cmake`, files `FindCUDA.cmake` and `FindOpenCL.cmake`. Make sure that CUDA and/or OpenCL is installed in order to use the NVIDIA GPU devices. Type **./convdiff_benchmark**, to execute the program in sequential mode. To execute in parallel mode with four processes, type **mpirun -np 4 ./convdiff_benchmark**. In both cases, you need to make sure that the default parameterfile `convdiff_benchmark.xml` is stored in the same directory as the binary, and that the geometry data specified in the parameter file is stored in `/hiflow/examples/data`. Alternatively, you can specify the path of your own xml-file with the name of your xml-file , i.e. to execute sequentially type **./convdiff_benchmark** `/"path_to"/convdiff_benchmark.xml`, to execute it in parallel mode with four processes type
mpirun -np 4 ./convdiff_benchmark `/"path_to"/convdiff_benchmark.xml`.

2 Linear Algebra Module

The HiFlow^3 finite element toolbox is based on a modular and generic framework in C++. The module LAtoolbox handles the basic linear algebra operations and offers linear solvers and

Algorithm 1 Distributed matrix vector multiplication $y = Ax$

function DISTR_MVMULT(A, x, y)
  Start asynchronous communication: exchange ghost values;
  $y_{\text{int}} = A_{\text{diag}}\, x_{\text{int}}$;
  Synchronize communication;
  $y_{\text{int}} = y_{\text{int}} + A_{\text{offdiag}}\, x_{\text{ghost}}$;
end function

preconditioners. It is implemented as a two-level library. The upper (or global) level is an MPI-layer which is responsible for the distribution of data among the nodes and performs cross-node computations, e.g. scalar products and sparse matrix-vector multiplications. The lower (or local) level takes care of the on-node routines offering an interface independent of the given platform.

2.1 The Global Inter-Node MPI-level

The MPI-layer of the LAtoolbox takes care of communication and computations in the context of finite element methods. Given a partitioning of the underlying domain (handed over by the mesh module) the DoF module distributes the degrees of freedom (DoF) according to the chosen finite element approach. This results in a row-wise distribution of the assembled matrices and vectors. Each local sub-matrix is divided into two blocks: a diagonal block representing all couplings and interactions within the subdomain, and an off-diagonal block representing the couplings across subdomain interfaces.
In Figure 1 we find a domain partitioning into four subdomains. In order to determine the structure of the global system matrices, first, each subdomain has to be associated with a single process on a node. Then, each process not only detects couplings within its own subdomain, but also couplings to the so-called *ghost DoF*, i.e. neighboring DoF which are owned by a different process. These identifications are performed simultaneously and independently on each node since the mesh module offers a layer of ghost DoF and hence no further communication is necessary. DoF i and j interact if the matrix has a non-zero element in row i and column j.

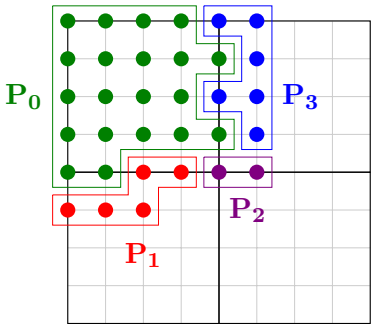

Figure 1: Domain partitioning: DoF of process $\mathbf{P_0}$ are marked in green (interior DoF in the diagonal block); the remaining DoF represent inter-process couplings for process $\mathbf{P_0}$ (ghost DoF in the off-diagonal block)

The distributed (sparse) matrix vector multiplication is given in Algorithm 1. While every process is computing its local contribution of the matrix vector multiplication an asynchronous communication for exchanging the ghost values is initiated. After this communication phase has been completed, the local contributions from coupled ghost DoF are added accordingly.

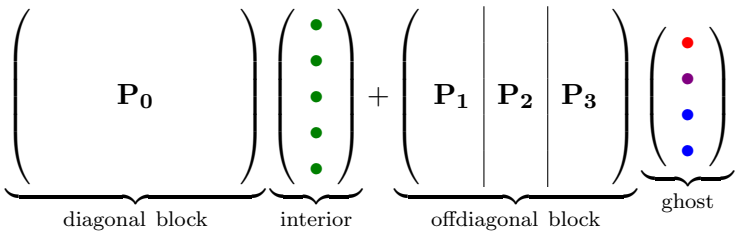

Figure 2: Distributed matrix vector multiplication

2.2 Local On-Node Level

The highly optimized BLAS 1 and 2 routines are implemented in the local multi-platform linear algebra toolbox (lmpLAtoolbox) which is acting on each of the subdomains. After the discretization of the PDE by means of finite element or finite volume methods typically a large sparse linear system with very low number of non-zeros is obtained. Therefore, the Compressed Sparse Row (CSR) data structure [4] is the favorable sparse matrix data format. On the NVIDIA GPU platforms we use a CUDA implementation with CSR matrix-vector multiplication based on a scalar version as it is described in [5, 6]. Our library supports several back-ends including multi-core CPUs, GPUs (CUDA/NVIDIA) and OpenCL (NVIDIA).

The LAtoolbox provides unified interfaces as an abstraction of the hardware and gives easy-to-use access to the underlying heterogeneous platforms. In this respect the application developer can utilize the LAtoolbox without any knowledge of the hardware and the system configuration. The final decision on which specific platform the application will be executed can be taken at run time. A data parallel model is used for the parallelization of the basic BLAS 1 and 2 routines which by their nature provide fine-grained parallelism. From a theoretical point of view the data parallel model results in good load balancing and scalability across several computing units. These expectations have been confirmed by our practical experiments across a variety of platforms and systems.

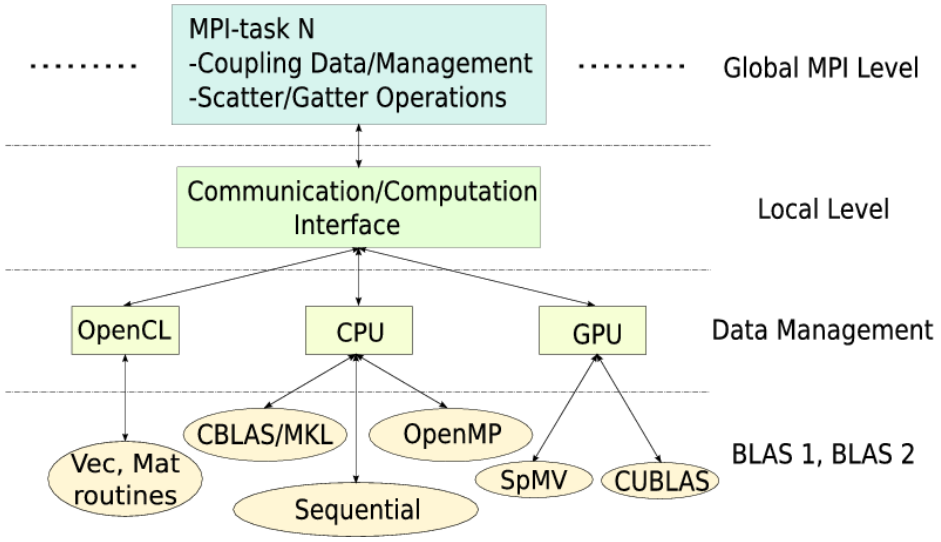

Figure 3: Structure of the lmpLAtoolbox and LAtoolbox for distributed computation and node-level computation across several devices in a homogeneous and heterogeneous environment.

The layered structure and organization of both the LAtoolbox and the ImpLAtoolbox is depicted in Figure 3. It shows a high-level view of distributed communication and computation across nodes and the node-level computation across several devices in a heterogeneous environment.
A cross-platform makefile generator (cmake) identifies all software libraries (i.e. OpenMP, CUDA, MKL, and etc) available on the underlying computing platform. After compilation the decision of final platform can be taken at run-time e.g. by providing information in a configuration file or from user input. All data structures (matrices and vectors) are distributed or offloaded accordingly. On a lean platform like a netbook or even a smartphone the project and all modules can be used in their sequential version. Within this setting there is further room for autotuning mechanisms, e.g. when several devices are available.
The code fragment in Listing 1 exemplifies handling of the ImpLAtoolbox. It details declaration of two vectors and a matrix. The CPU performs all input and output operations. To this end, the CPU matrix type is declared. Later conversion between two different platforms is based on a copy-like function. For avoiding unnecessary transfers to the same device a simple platform check is typically used.
The PCIe bus between host and device is a known bottleneck for attached accelerators. In particular, bandwidth limitations occur for updates of inter-process couplings between subdomains kept on different devices. Advanced data packaging and transfer techniques are utilized for mitigation of delays due to these inevitable cross-device data transfers. The underlying idea for maximized throughput is to manage and reorganize irregular data structures on the host CPUs and to transfer repackaged data in huge and continuous buffers to the accelerators.
Due to the full abstraction within the libraries there is limited or no access to the platform-specific data buffers. In certain cases – e.g. for nested loop iterations or irregular data access patterns – there is the option for defining device-specific data structures (vectors and matrices) with direct access to all data buffers.

Listing 1: Example code for ImpLAtoolbox

```
CPU_IMatrix<double> mat_cpu;
// Read matrix from a file
mat_cpu.ReadFile('matrix.mtx');

IVector<double> *x, *y ;
// init a vector for a specific platform and implementation
x = init_vector<double>(size,"vec_x", platform, impl);
// clone y as x
y = x->CloneWithoutContent();

IMatrix<double> *mat;
// init empty matrix on a specific platform
// (nnz,nrow,ncol,name,platform,implementation,format)
mat = init_matrix<double>(0,0,0,"A",platform,impl,CSR);
// Copy the sparse structure of the matrix
mat->CopyStructureFrom(mat_cpu);
// Copy only the values of the matrix
mat->CopyFrom(mat_cpu);

...
```

```
  // Usage of BLAS 1 routines
  y->CopyFrom(*x);          // y = x
  y->Axpy(*x, 2.3 );        // y = y + 2.3*x
  x->Scale(6.0);            // x = x * 6.0
  // print the scalar product of x and y
  cout << y->dot(*x);

  // Usage of BLAS 2 routines
  mat->VectorMult(*y, x); // x = mat*y

...

  delete x, y, mat;
```

Due to the modular setup and the consistent structure, the ImpLAtoolbox can be used as a standalone library independently of HiFlow[3]. It offers a complete and unified interface for many hardware platforms. Hence, the LAtoolbox not only offers fined-grained parallelism but also flexible utilization and cross-platform portability.
Theoretical details as well as benchmarks on different platforms of the preconditioners with impact on the solution procedure can be found in [8].

2.3 Different Backends on the Global-level

The global platform abstracted vectors and matrices have to be declared as pointers to the base class and initiated via the `init_platform_vec<>()` and `init_platform_mat<>()` init functions. Always the platform vectors/matrices have to be allocated together with the corresponding CPU vectors/matrices. If the vectors/matrices are set to be a CPU type – the pointers (`dev_` device-pointers) refer to the CPU objects. An example for the allocation can be found in Listing 2.

Listing 2: Platform abstracted vectors/matrices declaration in LAtoolbox

```
CMatrix matrix_, *dev_matrix_;
CVector sol_, rhs_, *dev_sol_, *dev_rhs_;

void ConvDiff::initialize_platform() {
  // init platform for matrix
  init_platform_mat<CMatrix>(la_sys_.Platform, matrix_impl_,
                             la_matrix_format_, matrix_precond_,
                             comm_, couplings_, &matrix_,
                             &dev_matrix_, la_sys_);
  // init platform for solution and right hand side vectors
  init_platform_vec<CVector>(la_sys_.Platform, vector_impl_,
                             comm_, couplings_, &sol_,
                             &dev_sol_, la_sys_);
  init_platform_vec<CVector>(la_sys_.Platform, vector_impl_,
                             comm_, couplings_, &rhs_,
                             &dev_rhs_, la_sys_);
  init_platform_ = false;
}
```

```
ConvDiff::~ConvDiff() {
  // Delete Platform Mat/Vec
  if (la_sys_.Platform != CPU) {
    // matrix
    delete dev_matrix_;

    // vector
    delete dev_sol_;
    delete dev_rhs_;
  }
}
```

The pair of vectors/matrices (i.e. for CPU and platform-specific) are needed mainly for the assembling procedure – it is available only for CPU-based objects, see Listing 3. After the assembling procedure the data are copied to the platform-specific objects – in case of CPU vectors/matrices the copy functions do not call itself.

Listing 3: Assembling for pPlatform abstracted vectors/matrices

```
void ConvDiff::assemble_system() {
  TimingScope tscope("Assemble system.");
  ConvDiffAssembler local_asm(beta_, nu_);
  StandardGlobalAssembler<double> global_asm;

  global_asm.assemble_matrix(space_, local_asm, matrix_);
  global_asm.assemble_vector(space_, local_asm, rhs_);

  // treatment of dirichlet boudary dofs
  if (!dirichlet_dofs_.empty()) {
    matrix_.diagonalize_rows(vec2ptr(dirichlet_dofs_),
                             dirichlet_dofs_.size(),
    static_cast<Scalar>(1.));
    rhs_.SetValues(vec2ptr(dirichlet_dofs_),
                   dirichlet_dofs_.size(),
                   vec2ptr(dirichlet_values_));
    sol_.SetValues(vec2ptr(dirichlet_dofs_),
                   dirichlet_dofs_.size(),
                   vec2ptr(dirichlet_values_));
  }

  rhs_.UpdateCouplings();
  sol_.UpdateCouplings();

  dev_matrix_->CopyStructureFrom(matrix_);
  dev_sol_->CopyStructureFrom(sol_);
  dev_rhs_->CopyStructureFrom(rhs_);

  dev_sol_->CopyFrom(sol_);
  dev_rhs_->CopyFrom(rhs_);
  dev_matrix_->CopyFrom(matrix_);
```

}

On the other side, the preconditioners and the solver can work only with the abstracted objects (i.e. pointers to the base class). However, there is one more trick for deploying some of the preconditioners – it might need to permute the stiffness matrix. Due to the couplings, this step is performed on the global level with the permutation function in the DoF module. An example for building different preconditioners is shown in Listing 4. The linear solver works directly with the base class of the vectors/matrices objects – for post-processing the data needs to be copy back from the device.

Listing 4: Block-preconditioners and solver with platform abstracted vectors/matrices

```
void ConvDiff::set_up_preconditioner() {
  TimingScope tscope("Set up preconditioner");
  if (matrix_precond_ == JACOBI) {
    PreconditionerBlockJacobiStand<LAD> *precond;
    precond = new PreconditionerBlockJacobiStand<LAD>;

    precond->Init_Jacobi(*dev_sol_);

    precond_ = precond;
  } else {
    PreconditionerMultiColoring<LAD> *precond;
    precond = new PreconditionerMultiColoring<LAD>;
    if (matrix_precond_ == GAUSS_SEIDEL) {
      precond->Init_GaussSeidel();
    } else if (matrix_precond_ == SGAUSS_SEIDEL) {
      precond->Init_SymmetricGaussSeidel();
    } else if (matrix_precond_ == ILU) {
      precond->Init_ILU(0);
    } else if (matrix_precond_ == ILU2) {
      int param1 = param_["LinearAlgebra"]["ILU2Param1"].get<int>();
      int param2 = param_["LinearAlgebra"]["ILU2Param2"].get<int>();
      precond->Init_ILU(param1, param2);
    }
    precond->Preprocess(*dev_matrix_, *dev_sol_, &(space_.dof()));
    // sync
    MPI_Barrier(comm_);

    prepare_lin_alg_structures();

    prepare_bc();

    assemble_system();

    precond_ = precond;
  }
  precond_->SetupOperator(*dev_matrix_);
  precond_->Build();
  precond_->Print();
```

```
  linear_solver_->SetupPreconditioner(*precond_);
}

void ConvDiff::solve_system() {
  linear_solver_->SetupOperator(*dev_matrix_);
  // sync
  MPI_Barrier(comm_);
  {
    TimingScope tscope("Solve system.");

    linear_solver_->Solve(*dev_rhs_, dev_sol_);
  }
  // sync
  MPI_Barrier(comm_);

  sol_.CopyFrom(*dev_sol_);
}
```

2.4 Parameter File

The needed parameters are initialized in the paramter file convdiff_benchmark.xml.
The `Platform` can be `CPU`, `GPU` or `OpenCL`. The possible matrix implementations are `MKL` and `OPENMP` for the CPU platform; `SCALAR` and `SCALAR_TEX` for the GPU (NVIDIA CUDA); and `OpenCL` for the `OpenCL` GPU NVIDIA devices. The possible vector implementations are `BLAS`, `MKL` and `OPENMP` for the CPU platform; `BLAS` for the GPU (NVIDIA CUDA); and `OpenCL` for the `OpenCL` backend. The parameters for the preconditioners and solvers can be found in the Listing 4 – see file `convdiff_benchmark.xml`.

```
,
<Param>
  <Model>
    <nu>1./37.</nu>
    <beta1>1.</beta1>
    <beta2>1.</beta2>
    <beta3>1.</beta3>
  </Model>
  <Mesh>
    <Filename>unit_square.inp</Filename>
    <RefinementLevel>4</RefinementLevel>
  </Mesh>
  <LinearAlgebra>
    <Platform>CPU</Platform>
    <MatrixImplementation>Naive</MatrixImplementation>
    <VectorImplementation>Naive</VectorImplementation>
    <MatrixFormat>CSR</MatrixFormat>
    <MatrixFreePrecond>ILU</MatrixFreePrecond>
    <ILU2Param1>1</ILU2Param1>
    <ILU2Param2>2</ILU2Param2>
  </LinearAlgebra>
  <FiniteElements>
    <Degree>3</Degree>
  </FiniteElements>
```

```
  <LinearSolver>
    <Name>GMRES</Name>
    <MaxIterations>10000</MaxIterations>
    <AbsTolerance>1.e-14</AbsTolerance>
    <RelTolerance>1.e-8</RelTolerance>
    <DivTolerance>1.e6</DivTolerance>
    <SizeBasis>50</SizeBasis>
    <Method>RightPreconditioning</Method>
    <UseILUPP>1</UseILUPP>
  </LinearSolver>
  <ILUPP>
    <PreprocessingType>0</PreprocessingType>
    <PreconditionerNumber>1010</PreconditionerNumber>
    <MaxMultilevels>20</MaxMultilevels>
    <MemFactor>0.8</MemFactor>
    <PivotThreshold>1.75</PivotThreshold>
    <MinPivot>0.01</MinPivot>
  </ILUPP>
  <Output>
    <LogFilename>info_conv_diff.log</LogFilename>
  </Output>
</Param>
```

3 GPU Clusters

The LAtoolbox can run on GPU cluster using MPI. Details can be found in the Poisson Tutorial. You can change the number of GPU devices per node in the file `platform_management.cc` (in `src/linear_algebra/lmp/`). The variable `num_gpu_per_node` describes the number of GPU devices that are availbe per node.
To obtain good scalability the mesh partitioning should provide minimum communication pattern – the ghost nodes should be as small as possible in order to minimize the communication over the PCI-E and network. Please note that the local preconditioners are applied in a block-Jacobi way and this will lead to different performance behavior of the linear solver for different number of MPI processes. To measure the pure scalability of cluster, you should use either no preconditioner or Jacobi type of preconditioner. Details can be found in [7, 8].

References

[1] http://www.khronos.org/opencl/.

[2] http://www.nvidia.com/object/cuda_home_new.html.

[3] http://www.nvidia.com/object/GPU_Computing.html.

[4] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[5] M. M. Baskaran and R. Bordawekar. Optimizing Sparse Matrix-Vector Multiplication on GPUs. Technical report, IBM, 2009.

[6] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[7] V. Heuveline, D. Lukarski, C. Subramanian, and J. P. Weiss. A Multi-platform Linear Algebra Toolbox for Finite Element Solvers on Heterogeneous Clusters. In *PPAAC'10, IEEE Cluster 2010 Workshops*, 2010.

[8] D. Lukarski. *Parallel Sparse Linear Algebra for Multi-core and Many-core Platforms – Parallel Solvers and Preconditioners*. PhD thesis, Karlsruhe Institute of Technology, 2012.