# GPU accelerated libraries for computational science and engineering problems

A. Strelchenko


CaSToRC

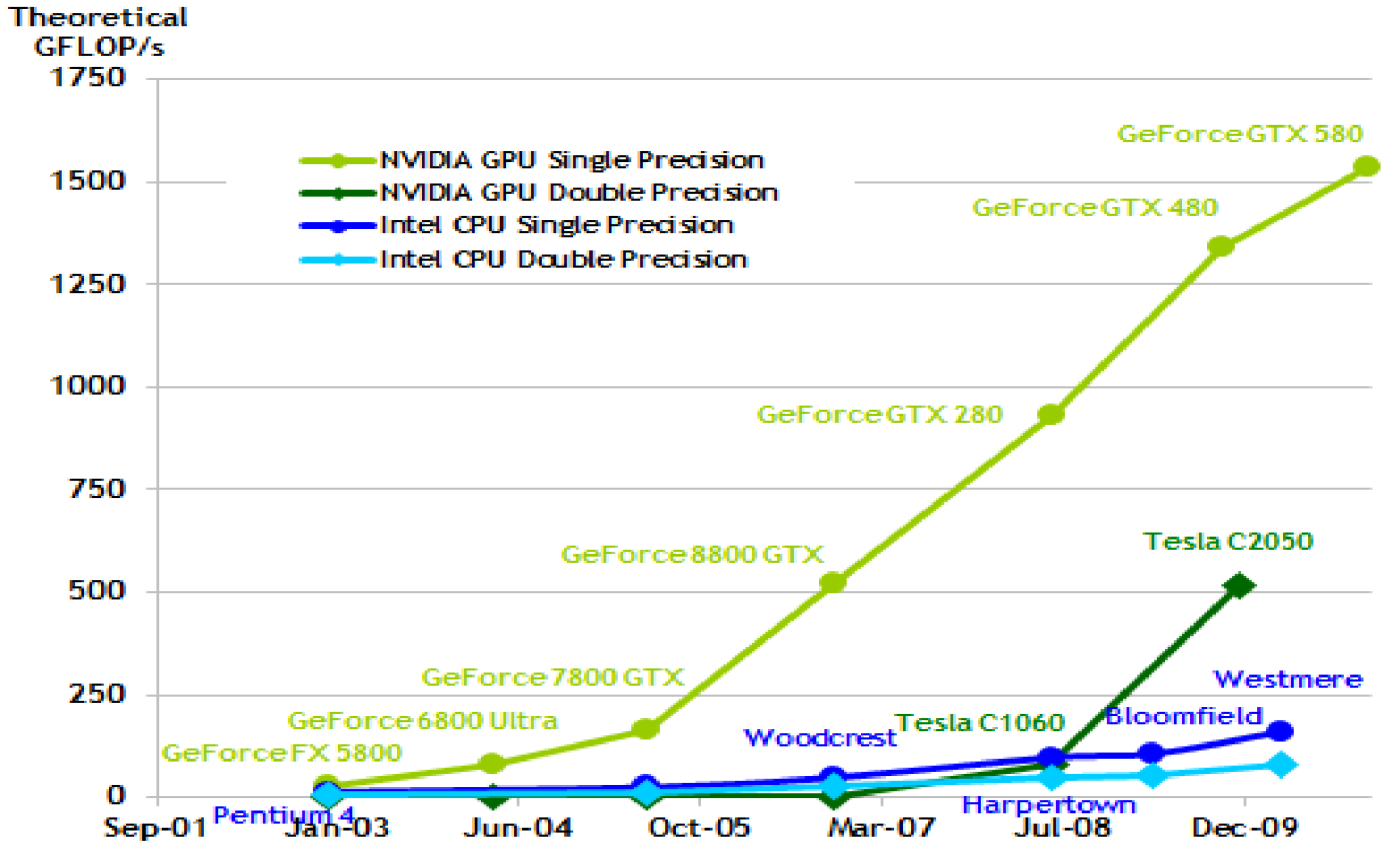
# Plan:

- **Challenges of Heterogeneous Architectures**
- **Introduction to GPGPU APIs**
  - CUDA API
  - OpenCL API
- **Overview of GPU accelerated libraries**
- **Links**

# Welcome to heterogeneous world!

| Rank | Site | Computer |
|---|---|---|
| 1 | RIKEN Advanced Institute for Computational Science (AICS)<br>Japan | K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect<br>Fujitsu |
| 2 | National Supercomputing Center in Tianjin<br>China | NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050<br>NUDT |
| 3 | DOE/SC/Oak Ridge National Laboratory<br>United States | Cray XT5-HE Opteron 6-core 2.6 GHz<br>Cray Inc. |
| 4 | National Supercomputing Centre in Shenzhen (NSCS)<br>China | Dawning TC3600 Blade System, Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050<br>Dawning |
| 5 | GSIC Center, Tokyo Institute of Technology<br>Japan | HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows<br>NEC/HP |

# GPUs vs CPU (theor. peak perf.)



# Challenges of using heterogeneous systems

- **High level of parallelism:**
  – [e.g., Tesla M2070 has 2x448 CUDA cores]

- **Hybrid architectures**
  – match algorithmic requirements to architectural strength
  – [e.g., well-parallelizable apps to run on GPUs]

- **Compute vs communication gap:**
  – [e.g. Tesla M2070 peak perf. ~ O(1000) Gflop/s,
  while PCIe BW is ~ O(1)GB/s, GPU memory BW O(100)GB/s]

# Challenges of using heterogeneous systems

- Optimized software packages and libraries are often an easy way to improve performance of an application
- When porting large legacy projects, they may be the only way to optimize for a new platform, since code changes would require extensive validation efforts
- Main concern is to find a project that provide general solution and design pattern that can be adapted to your project
- This tutorial is mostly about general purpose scientific libraries

# General purpose libraries

- **CUDA-based high performance libraries:**
  - cuBLAS : Complete BLAS library
  - cuSPARSE : Sparse Matrix library
  - cuRAND : RNG library
  - cuFFT : FFT library
  - MAGMA : BLAS (iter. solvers, eigensolvers)
  - CUSP : Sparse matrix library
- **OpenCL-based high performance libraries :**
  - APPML : AMD math libs (BLAS & FFT)
  - ViennaCL : LA routines (inc. solvers etc.)

# General purpose libraries

- **Mathematica 8 (CUDA & OpenCL support):**
  - multiple-GPU support
  - access to *Mathematica's* flexible programming language
  - access to *Mathematica's* computable data, visualization etc.
  - GPU accelerated functions (BLAS, FFT, image processing)

- **MATLAB (CUDA support):**
  - data manipulation on NVIDIA GPU
  - GPU accelerated MATLAB operations
  - integration of CUDA kernels into MATLAB apps.
  - use of multiple GPUs on the desktop and a computer cluster

# Specialized software packages (examples)

- **Monte-Carlo methods**
  - Monte-Carlo eXtreme (MCX) package for time-resolved photon transport in 3D turbid media,

    http://mcx.sourceforge.net
  - GPU stochastic simulation for data analysis (GPUSS),
    http://www.oxford-man.ox.ac.uk/gpuss

- **Molecular Modeling**
  - OpenMM library for molecular dynamics,

    https://simtk.org/home/openmm
  - NAMD/VMD molecular modeling system,

    http://www.ks.uiuc.edu/Research/vmd/

# Specialized software packages (cont.)

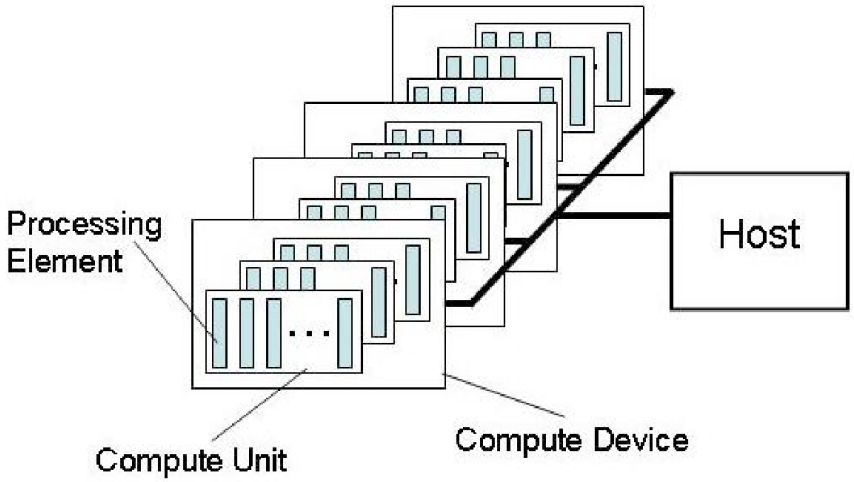
- **Computational fluid dynamics**
  - Symscape ofgpu library: GPU linear solvers for OpenFOAM,
    http://www.symscape.com/gpu-0-2-openfoam

- **Quantum chemistry**
  - Quantum ESPRESSO: electronic-structure calculations and materials modeling at the nano-scale,
    http://www.quantum-espresso.org/

- **Climate and atmospheric science**
  - GPU accelerated WSM5 microphysics
    http://www.mmm.ucar.edu/wrf/WG2/GPU/WSM5.htm

# Hybrid platform model

- **A typical hybrid platform consists of**
  - one or more CPUs
  - one or more accelerators (GPUs, DSPs etc.)



# Minimal programming approach on hybrid systems

- **What is required:**
  - device/context initialization
  - DeviceToHost & HostToDevice data transfer
  - host-side multi-thread execution and synchronization

- **CUDA API**
  - currently for NVIDIA GPU only (or x86 compiler from Portland)
- **OpenCL API**
  - close to CUDA driver API (e.g., explicit context management etc.)

- **OpenACC** (PRAGMA compiler directives )

# CUDA API overview

- **CUDA runtime API**
  - implemented in cudart dynamic library
  - minimal set of extensions to the C language
  - functions for memory management, system management etc.
  - build on top of a lower-level CUDA driver API

- **CUDA driver API**
  - additional of control over system
  - explicit initialization, context, module (analog. dll) management
  - semi-assembly language, *PTX*

# CUDA device management

- Device profiling:

  cudaGetDeviceCount(void* device)
  - retrun the number of compute-capable devices

  cudaGetDeviceProperties(struct cudaDeviceProp *prop, int device)
  - return information about the compute device

- Device setup:

  cudaSetDevice(int device)
  - set device to be used for GPU execution

  cudaChooseDevice(int *dev, struct cudaDeviceProp *prop)
  - return the device which has properties that best match *prop

# CUDA memory management

- 1D linear arrays (similar to C-arrays) :

  cudaMalloc(void**dev_ptr, size_t size);

- 2D/3D arrays, e.g.:

  cudaMallocPitched(void**dev_ptr, size_t *pitch, size_t width, size_t height);

- Moving data to/from device:

  cudaMemcpy(void* dst, const void* src, size, enum cudaMemcpyKind);

- Flag cudaMemcpyKind:
  - cudaMemcpyHostToDevice etc.
  - cudaMemcpyDefault : for Fermi architecture (due to UVA space)

- Memory deallocation: cudaFree();

# CUDA memory management

- Memory allocation and copy:

```
#define N 1024
int main(){
    int *hx;
    //Initialization of host arrays:
     ...
    //Allocation of the device arrays:
    int *dx;
    cudaMalloc(&dx, N*sizeof(int));
    //Copy data from host to device:
    cudaMemcpy(dx, hx, N*sizeof(int), cudaMemcpyHostToDevice);
    ...
}
```

# Page-locked memory

- Benefits :
  - concurrent memory copies (with kernel execution)
  - can be mapped into address space of the device (no need to copy!)
  - increase host-device PCIe bandwidth

- **WARNING:**
  consuming too much pinned memory may reduce system performance!

- Allocation:
  cudaHostAlloc(void** hostPtr, size_t size, unsigned int FLAGS);

  FLAGS : cudaHostAlloc{Portable, WriteCombined, Mapped}

- Deallocation: cudaHostFree();

# Page-locked memory

- Page-locked memory example:

```
int main(){
    //Allocate page-locked host arrays:
    cudaHostAlloc(&hx, N*sizeof(int), cudaHostAllocWriteCombined);
    ...
    //Allocate device arrays:
    cudaMalloc(&dx, N*sizeof(int));
    ...
    //Copy data from host to device:
    ...
    cudaFreeHost(hx);
}
```

# Page-locked memory

- Zero copy (no UVA space) example:

```
int main(){
    ...
    cudaSetDeviceFlags(cudaDeviceMapHost);
     //Allocate page-locked host arrays:
    cudaHostAlloc(&hx, N*sizeof(int), cudaHostAllocWriteCombined);
    ...
     //Map the host pointer to device address space:
    int *dx;
    cudaHostGetDevicePointer((void**)&dx, (void*)hx, 0);
     ...
}
```

# Asynchronous concurrent execution

- Asynchronous function calls:
  - kernel launches
  - device to device memory copies
  - host to device memory copies (< 64KB)
  - memory copies with Async suffix
- Concurrent behavior:
  - overlap of data transfer and kernel execution
  - concurrent kernel execution (within a single context, GF100 only)
  - concurrent data transfer (device-to-host with host-to-devcie,GF100 only)
- Warning:
  - host memory must be page-locked!

# CUDA stream API: command streams and host synchronization

- CUDA streams:
  - sequence of commands that execute in order
  - different streams may execute out-of-order

- Stream creation:

```
cudaStream_t stream;
cudaStreamCreate(&stream);
```

- Synchronization of commands in (a) stream(s):

```
cudaDeviceSynchronize(); //synchronization for all command streams
cudaStreamSynchronize(stream); // ...for a particular command stream
```

# CUDA stream API: examples

- Example1 (concurrent data transfer):

```
cudaStream_t stream;
...
cudaMemcpyAsync(dPtr0, hPtr0, size, cudaMemcpyHostToDevice, stream);
cudaMemcpyAsync(hPtr1, dPtr1, size, cudaMemcpyDeviceToHost, stream);
```

- Example2 (overlap of data transfer with kernel execution):

```
cudaStream_t stream0, stream1;
...
cudaMemcpyAsync(dPtr0, hPtr0, size, cudaMemcpyHostToDevice, stream0);
/*  Set stream 1 that will be used to execute some CUDA functions */
/*  Launch CUDA function e.g. CUBLAS, CUSPARSE etc...              */
```

# OpenCL API highlights

- **Platform layer API**
  - Provides management for computational resources
  - Query, select and initialize computational device
  - Create context(s) and command queue(s)

- **Runtime API**
  - Submit device code for execution
  - Define execution domain for compute kernels
  - Manage scheduling, compute and memory resources

# OpenCL objects

- **Setup objects:**
  - devices, contexts, queues
- **Memory objects:**
  - buffers, images, samplers
- **Execution objects:**
  - programs, kernels
- **Synchronization/profiling objects:**
  - events

# Typical OpenCL application

- **Host code**
  - query compute devices
  - create contexts
  - manage memory objects associated to contexts
  - *compile program objects*
  - issue commands to command-queues
  - synchronization of commands
- **Device code (the kernel):**
  - written in OpenCL C
  - executes (in parallel) on the device

# OpenCL platform layer

- Query OpenCL platform information:
  - clGetPlatformIDs()
    - – list of available platforms
  - clGetPlatformInfo()
    - – profile, version, vendor, extensions
  - clGetDeviceIDs()
    - – list of available devices
  - clGetDeviceInfo()
    - – type, capabilities

- OpenCL platforms: 'Advanced Micro Devices, Inc.', 'NVIDIA Corporation'

- OpenCL devices: CL_DEVICE_TYPE_CPU, CL_DEVICE_TYPE_GPU , etc.

# OpenCL context creation

- Get the platform ID:

```
cl_platform_id    platform;
clGetPlatformIDs(1, &platform, NULL);
```

- Get the first GPU device associated with the platform:

```
cl_device_id    device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
```

- Create an OpenCL context for the GPU device:

```
cl_context  context
                    = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
```

# OpenCL command queues

- All work is done through command queue(s) (a)synchronously:
  - Memory operations (e.g., copy to/from device etc.)
  - Kernel execution
- Command queue properties:

  CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE

  CL_PROFILING_ENABLE
- Create a command queue on a specific device:

```
cl_command_queue   queue
        = clCreateCommandQueue(context, device,
                                        CL_QUEUE_PROFILING_ENABLE, NULL);
```

# OpenCL memory objects

- **Buffers objects:**
  - 1D collection of objects (like C-arrays)
  - scalar, vector types and user-defined structures
  - accessed via pointers in compute kernels
- **Image objects:**
  - 2D and 3D, frame buffer or images
  - addressed via build-in functions
- **Sampler objects:**
  - describe how to sample an image in the kernel
  - addressing, filtering modes

# Buffers

- Create a buffer object (example):

```
int   *hx = (int*)malloc(N*sizeof(int));
cl_mem   dx
            = clCreateBuffer(context, flags, N*sizeof(cl_int),  hx, NULL);
```

- Parameter *flags* specifies allocation and usage information:

| | |
|---|---|
| CL_MEM_READ_WRITE | : read & written by a *kernel* |
| CL_MEM_{READ/WRITE}_ONLY | : read or written by a *kernel* |
| CL_MEM_USE_HOST_PTR | : use *host* memory |
| CL_MEM_ALLOC_HOST_PTR | : allocate from *host* accessible memory |
| CL_MEM_COPY_HOST_PTR | : the same as above + copy from *host* memory |

# Operations on memory objects

- Write to a buffer from a host memory:

```
cl_int   err = clEnqueueWriteBuffer(queue, dx, blocking_write,
                                    0, N*sizeof(int),  hx, 0, NULL, NULL);
```

- Parameter *blocking_write* specifies :

  CL_TRUE : does not return until operation will complete
  CL_FALSE : non-blocking operation

```
cl_int err = clEnqueueReadBuffer(queue, dy, blocking_read,
                                 0, N*sizeof(int), hy, 0 NULL, NULL)
```

# Event objects and synchronization

- Status of each command can be traced by an event object:
  To enable command profiling set CL_QUEUE_PROFILING_ENABLE flag!

- Capture command profiling information:
  clGetProfilingInfo(clEvent, *param_info*, 0, NULL, NULL);

  – *param_info* : CL_PROFILING_COMMAND_QUEUED etc.

- Command synchronization through event:
  clWaitForEvents(1, &clEvent);

- Synchronization of all commands in a command queue:
  clEnqueueBarrier(queue);

# CURAND overview

- **PRNGs:**
  - xor-shift (XORWOW)
  - combined multiple recursive (MRG32K3A)
  - Mersenne Twister (MTGP32)
- **Sobol quasi-random number generators**
- **Host API for generating random numbers in bulk**
- **Inline implementation allows use inside GPU kernels**
- **Single-/double- precision, uniform, normal, log-normal distributions**

# CURAND sequence of operations

- **Create a new generator :** curandCreateGenerator()
- **Set the generator options:** curandSetPseudorandomGeneratorSeed()
- **Allocate device memory**
- **Set command stream (if necessary):** curandSetStream()
- **Generate random numbers: e.g.** curandGenerateUniform()
- **Use the result (e.g., copy back to host)**
- **Clean up:** curandDestroyGenerator()

# CURAND exercise: compute Pi

- **Create a new generator :**
  1) curandGenerator_t gen;
  2) curandCreateGenerator( &gen, TYPE_OF_GENERATOR);

- **Set the generator seed:**
  3) curandSetPseudoRandomGeneratorSeed(gen, seed);

- **Generate random numbers:**
  4) curandGenerateUniform(gen, hostPtr, dataSize);

- **Use the result (e.g., copy back to host, and estimate the number)**

- **Clean up:** curandDestroyGenerator()

# CUFFT overview

- Algorithm based on Cooley-Tuley
- Interface similar to FFTW
- 1D, 2D, 3D transforms with real and complex data
- Batch execution (multiple transforms in parallel)
- In-place and out-of-place transforms
- Thread-safe API (can be called from multiple host threads)

# CUFFT overview

- **CUFFT preliminary setup :** #include <cufft.h>
- **Internal types:** cufftReal, cufftDoubleReal etc.
- **Transformation type:** CUFFT_{R2C, C2R, C2C, D2Z, Z2D, Z2Z}
- **FFTW compatibility:** CUFFT_COMPATIBILITY_FFTW_{PADDING etc.}
- **CUFFT transform directions:** CUFFT_{FORWARD, INVERSE}

# CUFFT sequence of operations

- **Create a CUFFT plan:** cufftPlanXd(cufftHandle *plan)
- **Allocate memory on GPU and copy data from Host**
- **Perform transformation:** cufftExec{C2C, Z2Z etc.}
- **Copy data back to host**
- **Release CUFFT resources ( e.g.,** cufftDestroy(plan)**)**

# CUFFT exercise: 1d Z2Z transform

- **Create a cufft plan :**
  1) specify transform size, type (CUFFT_Z2Z), and direction (CUFFT_FORWARD);
  2) cufftHandle plan;
  3) cufftPlan1d(plan, size, type, batch);

- **Copy data to device:**

- **Transform the signal:**
  4) cufftExecZ2Z(plan, in, out, dir);

- **Use the result (e.g., copy back to host, and compare with FFTW)**

- **Clean up:** cufftDestroy(plan)

# CUBLAS overview

- **Level 1 (vector, vector):**
  - AXPY : y[i] = a * x[i] + y[i]
  - Dot products, norms etc.
- **Level 2 (matrix, vector):**
  - vector multiplication by a general matrix : GEMV
  - triangular solver: TRSV
- **Level 3 (matrix, matrix):**
  - general matrix matrix multiplication : GEMM
  - triangular solver: TRSM

# CUBLAS overview

- **CUBLAS uses column-major storage (**following BLAS convention**)**
- **Supports 4 types:**
  - float, double, complex, double complex
  - prefixes: S, D, C, Z
- **Function naming convention cublas+function name:**
  - example : cublasSgemm = cublas+S+ge+mm
- **Helper functions: memory allocation, data transfer**

# CUBLAS sequence of operations

- **Create CUBLAS context (if necessary):** cublasCreate()
- **Set CUDA stream (if necessary):** cublasSetStream()
- **Allocate CUBLAS resources on GPU:**
  – either using CUDA API (e.g. , cudaMalloc() , cudaMemcpy())
  – or using CUBLAS routines (e.g., cublasAlloc(), cublasSetVector(), etc.)
- **Perform computations with CUBLAS functions:**
  – e.g., cublasDgemm(), cublasDdot() etc.
- **Copy data back to host and shutdown the library:** cublasDestroy()

# CUSPARSE overview

- **Sparse matrix formats: COO, CSR, CSC, ELL, HYB**
- **Level 1 (sparse vector, dense vector):**
  – axpy, dot, gthr, sctr etc.
- **Level 2 (sparse matrix, dense vector):**
  – csrmv, csrmv_solve, csrmv_analisys (sparse triangular linear solver)
- **Level 3 (sparse matrix, set of dense vectors):**
  – csrmm (sparse matrix - 'tall' dense matrix product)
- **Conversion routines (between different matrix formats)**

# CUSPARSE sequence of operations

- **Create CUSPARSE context (if necessary):** cusparseCreate()
- **Set CUDA stream (if necessary):** cusparseSetStream()
- **Allocate CUSPARSE resources on GPU:** (e.g. , cudaMalloc() , cudaMemcpy())
- **Create sparse matrix descriptor:** cusparseCreateMatDescriptor()
- **Setup sparse matrix descriptor:** cusparseSetMatIndexBase(), etc.
- **Perform computations with CUSPARSE functions:** e.g., cusparseScsrmv()
- **Copy data back to host and shutdown the library:**
  - cusparseDestroyMatDescr(), cusparseDestroy()

# CUBLAS/CUSPARSE exercise: CG solver

- **Solving sparse linear system** Ax=b **with CG algorithm:**

1) r0 = b – A x0; p0 = r0;
2) FOR i = 0, 1, 2, … WHILE (r, r) < tolerance DO:
3) alpha = (r, r) / (p, A p);
4) x = x + alpha*p;
5) new_r = r – alpha*A p;
6) beta = (new_r, new_r) / (r, r);
7) p = new_r + beta*p; r = new_r;
8) END

# CUBLAS/CUSPARSE exercise: CG solver

- **CUBLAS/CUSPARSE routines:**

1) r0 = b – A x0; p0 = r0;
2) FOR i = 0, 1, 2, … WHILE (r, r) < tolerance DO:
3) alpha = (r, r) / (p, A p); // A p done by CUSPARSE
4) x = x + alpha*p;
5) new_r = r – alpha*A p;
6) beta = (new_r, new_r) / (r, r);
7) p = new_r + beta*p; r = new_r;
8) END

# CUBLAS/CUSPARSE exercise:CG solver

- **Create CUSPARSE context:**
  1) cusparseHandle_t handle;
  2) cusparseCreate(&handle);

- **Create sparse matrix descriptor:**
  3) cusparseMatDescr_t descr;
  4) cusparseCreateMatDescriptor(&descr);

- **Setup sparse matrix descriptor:**
  5) cusparseSetMatType(..., CUSPARSE_MATRIX_TYPE_GENERAL);
  6) cusparseSetMatIndexBase(..., CUSPARSE_INDEX_BASE_ZERO);
  7) cusparseSetMatDiagType(..., CUSPARSE_DIAG_TYPE_NON_UNIT);

# CUBLAS/CUSPARSE exercise:CG solver

- **CUSPARSE mv operation** y = a*op(A)*x + b*y**:**

  cusparseScsrmv(handle, op_type, m, n, descr, Val, Rows, Cols, a, x, b, y) ;

  op_type = CUSPARSE_OPERATION_NON_TRANSPOSE

- **CUBLAS operations:**

  1) <x, y> = cublasDdot(n, x , inc_x, y, inc_y);

  2) y=y+a*x : cublasDaxpy(n, a, x, inc_x, y, inc_y);

  inc_{x, y} – stride between consequent elements of x or y

- **Shutdown CUSPARSE library:**

  cusparseDestroyMatDescr(descr); cusparseDestroy(sparseHandle);

# MAGMA overview

- **Fundamental linear algebra algorithms:**
  - one- and two-sided factorizations
  - linear solvers (based on LU, QR and Cholesky  preconditioning)
  - eigen-solvers
- **CPU and GPU interfaces**
  - e.g., magma_sgetrf_gpu() for GPU, magma_sgetrf() for CPU
- **Mixed precision solvers (using iterative refinement procedure)**
- **BLAS routines (optimized for MAGMA algorithms)**

# MAGMA overview

- **Magma interface:** #include “magma.h”
- **One-sided factorizations {LU, QR, Cholesky}:**
  - magma_X{getrf, geqrf, potrf} : CPU interface
  - magma_X{getrf, geqrf, potrf}_gpu : GPU interface
- **Linear solvers (work precision):**
  - magma_X{getrs, geqrs, potrs} : CPU interface
  - magma_X{getrs, geqrs, potrs}_gpu : GPU interface
- **Linear solvers (mixed precision):**
  - magma_ds{gesv, geqr, posv}_gpu : GPU interface

# MAGMA exercise: iterative solver with LU preconditioning

- **Solving** AX = B **through LU factorization, where** A **is** nxn **matrix:**

magma_dgesv_gpu(n, nrhs, A, lda, ipiv, B, ldb, info) ;

nrhs : number of right hand sides

A : matrix of dimension (lda, n), lda >= max(1, n)

on entry – source vector B (or tall matrix) , on exit – solution vector B

ipiv : integer array used to keep pivot indices of dim = min(m, n)

ldb : leading dimension of array b ldb >= max(1, n)

info : output integer 0 (successful exit), -i (ith arg is illigal)

# ViennaCL overview

- **Provides C++ interface to OpenCL implemented LA routines**
- **Direct/iterative solvers and preconditioners:**
  - direct solvers for upper/lower triang. systems
  - iterative solvers: CG, BiCG, GMRES
  - ILU, Jacobi, row-scaling
- **Interoperability with other libs (e.g., ublas, eigen, mtl4)**
- **Allows to use custom OpenCL kernels to reach full functionality**
- **Support for multiple devices/contexts**

# ViennaCL overview

- **Basic types:**
  - scalar/vector type: viennacl::scalar<T>, viennacl::vector<T, alignment>
  - dense matrices: viennacl::matrix<T, F, alignment>
  - sparse matrices: viennacl::compressed_matrix<T, alignment>
- **Basic LA operations: level 1, level 2 & level 3**
- **Sparse matrix vector multiplication**
- **By default, silently creates its own context and add all available devices with a single queue per device to the context**
- **Allows to use custom OpenCL contexts as well as devices**

# ViennaCL : examples

- Basic operations (scalars):

```
float cpu_float = 42.0f;
double cpu_double = 13.7603;
viennacl::scalar<float>  gpu_float(3.1415f);
viennacl::scalar<double> gpu_double = 2.71828;

//conversions and t
cpu_float = gpu_float;
gpu_float = cpu_double;  //automatic transfer and conversion

cpu_float = gpu_float * 2.0f;
cpu_double = gpu_float - cpu_float;
```

# ViennaCL : examples

- Direct solver for dense linear system :

```
using namespace viennacl::linalg;  //to keep solver calls short
viennacl::matrix<float>  vcl_matrix;
viennacl::vector<float>  vcl_rhs;
viennacl::vector<float>  vcl_result;

/* Set up matrix and vectors here */

//solution of an upper triangular system:
vcl_result = solve(vcl_matrix, vcl_rhs, upper_tag());
//solution of a lower triangular system:
vcl_result = solve(vcl_matrix, vcl_rhs, lower_tag());

//solution of a full system right into the load vector vcl_rhs:
lu_factorize(vcl_matrix);
lu_substitute(vcl_matrix, vcl_rhs);
```

# ViennaCL : examples

- Iterative solvers for sparse linear systems:

```
viennacl::compressed_matrix<float>  vcl_matrix;
viennacl::vector<float>  vcl_rhs;
viennacl::vector<float>  vcl_result;

/* Set up matrix and vectors here */

//solution using conjugate gradient solver:
vcl_result = viennacl::linalg::solve(vcl_matrix,
                                     vcl_rhs,
                                     viennacl::linalg::cg_tag());

//solution using BiCGStab solver:
vcl_result = viennacl::linalg::solve(vcl_matrix,
                                     vcl_rhs,
                                     viennacl::linalg::bicgstab_tag());

//solution using GMRES solver:
vcl_result = viennacl::linalg::solve(vcl_matrix,
                                     vcl_rhs,
                                     viennacl::linalg::gmres_tag());
```

# ViennaCL : examples

- Custom context:

```
cl_context my_context = ...;      //a context
cl_device_id my_device = ...;     //a device in my_context
cl_command_queue my_queue = ...; //a queue for my_device

//supply existing context 'my_context'
// with one device and one queue to ViennaCL using id '0':
viennacl::ocl::setup_context(0, my_context, my_device, my_queue);

viennacl::ocl::switch_context(id);
```

# ViennaCL : examples

- Wrapping OpenCL buffers with ViennaCL types:

```
cl_mem my_memory1 = ...;
cl_mem my_memory2 = ...;
cl_mem my_memory3 = ...;
cl_mem my_memory4 = ...;
cl_mem my_memory5 = ...;

//wrap my_memory1 into a vector of size 10
viennacl::vector<float> my_vec(my_memory1, 10);


//wrap my_memory3 into a CSR sparse matrix with 10 rows and 20 nonzeros
viennacl::compressed_matrix<float> my_sparse(my_memory3,
                                             my_memory4,
                                             my_memory5, 10, 10, 20);
```

# Links:

- http://developer.amd.com/gpu/AMDAPPSDK
  - AMD APP SDK,  incl. OpenCL for x86 architecture
- http://developer.nvidia.com/object/opencl.html
  - NVIDIA CUDA, documentations, forum

# Thank you

www.cyi.ac.cy

The Cyprus Institute

CaSToRC