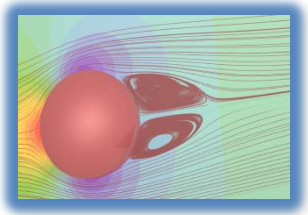
UNIVERSITY OF WYOMING



CU++ : An Object Oriented Tool for CFD Applications on GPUs

Dominic Chandar, Jay Sitaraman, and Dimitri Mavriplis
University of Wyoming
Laramie, WY-82070

2012 GPU Technology Conference,
San Jose, CA
17 May 2012

Motivation - Simplify Numerical Software Development

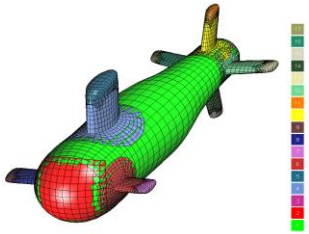
Overture – LLNL
A++P++ Library

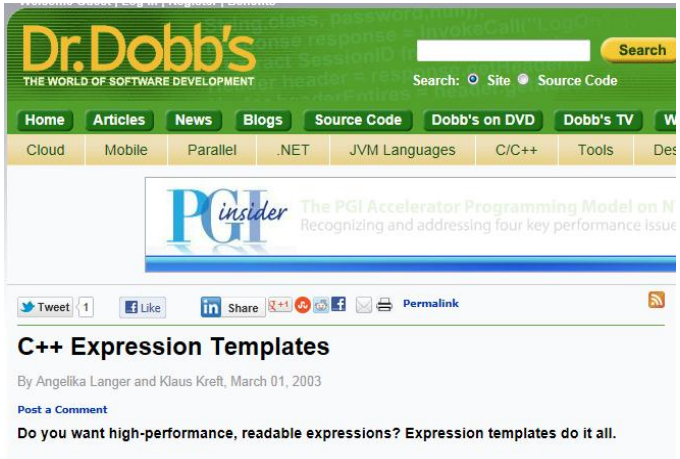
Vector addressing – Fortran like statements

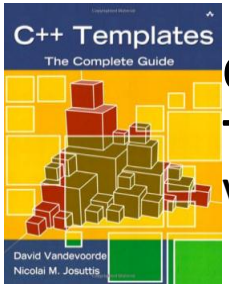
```
u(I) = u(I+1) + 0.5*u(I-1)
```

Serial or Parallel mode indices

CG Flow Codes







C++ Templates : The Complete Guide
Vandevoode and Josuttis

CUDA Based Expression Templates which were developed concurrently

CUDA Expression Templates

Paul Wiemann, Stephan Wenger, Marcus Magnor

Home | Team | Research | Teaching

Paul Wiemann, Stephan Wenger, and Marcus Magnor:
"CUDA Expression Templates",
in *WSCG Communication Papers Proceedings*, pp. 185–192, January 2011.
ISBN 978-80-86943-82-4
[pdf] [bib] [source]

CHAPTER

Processing Device Arrays with C++ Metaprogramming

32

GPU Computing Gems

Jonathan M. Cohen

Indexing is not straightforward as A++P++

Does not work with MPI

Does not work for unstructured data

A Simple Example - 2D Poisson Equation on a Rectangular Domain - $\nabla^2 u = 0$

Discretized form on a Cartesian grid reads :

$$u_{ij} = \frac{1}{4}\left(u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right)$$

C/C++ Serial Implementation

```
for ( int idy = 1 ; idy <= n-1 ; idy++ ){
  for ( int idx = 1 ; idx <= n-1 ; idx++ ){
      int id  = idx + n*idy;          // ( I, J )
      int idr = idx+1 + n*idy;        // ( I, J+1)
      int idl = idx-1 + n*idy;        // ( I, J-1)
      int idt = idx + n*(idy+1);      // ( I+1, J)
      int idb = idx + n*(idy-1);      // ( I-1, J)
unp1[id] = 0.25*( un[idr] + un[idl] + un[idt] + un[idb] )
    }}
```

Comparison of C++, CUDA, and CU++

C/C++ Serial Implementation

```
for ( int idy = 1 ; idy <= n-1 ; idy++ ){
  for ( int idx = 1 ; idx <= n-1 ; idx++ ){
      int id  = idx + n*idy;          // ( I, J )
      int idr = idx+1 + n*idy;        // ( I, J+1)
      int idl = idx-1 + n*idy;        // ( I, J-1)
      int idt = idx + n*(idy+1);      // ( I+1, J)
      int idb = idx + n*(idy-1);      // ( I-1, J)
unp1[id] = 0.25*( un[idr] + un[idl] + un[idt] + un[idb] )
    }}
```

CUDA Implementation

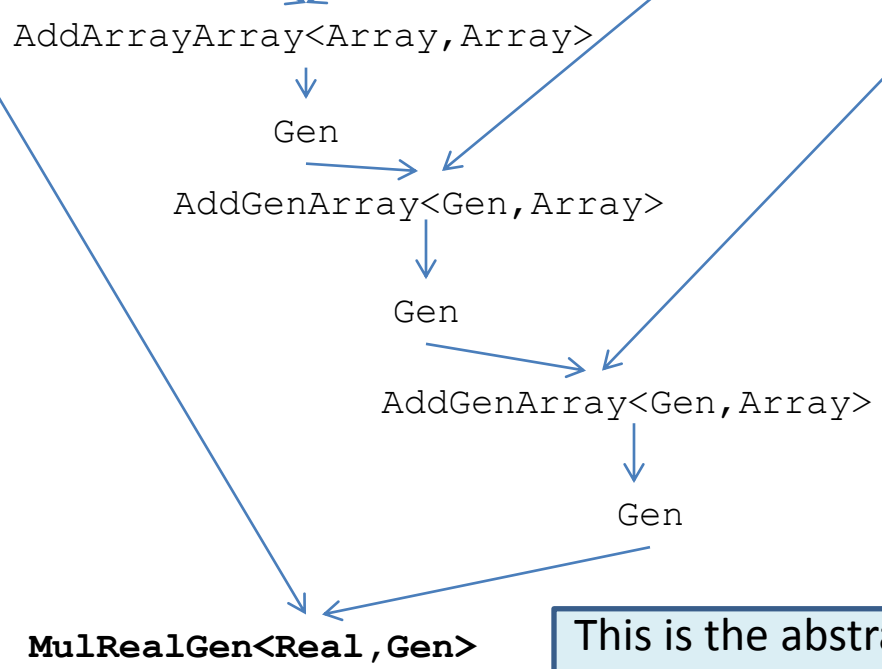
```
__global__ void point_jacobi( float* unp1, float* un,. . .)
{
  int idx = threadIdx.x + blockIdx.x*blockDim.x;
  int idy = threadIdy.y + blockIdy.y*blockDim.y;
  int id  = idx + n*idy;          // ( I, J )
  int idr = idx+1 + n*idy;        // ( I, J+1)
  int idl = idx-1 + n*idy;        // ( I, J-1)
  int idt = idx + n*(idy+1);      // ( I+1, J)
  int idb = idx + n*(idy-1);      // ( I-1, J)
  if ( idx >=1 && idx <=n-1 && idy >=1 && idy <= n-1 )
    unp1[id] = 0.25*( un[idr] + un[idl] + un[idt] + un[idb] )
}
```

CU++ Implementation

```
// Index objects are used to represent the base and bound of the array
Index i(1,N-2), j(1,N-2);
// u is a distributed array object defined as follows:
distArray u(N,N);
   for ( step = 0 ; step < maxNumberofSteps ; step++ )
     {
        u(i,j) = 0.25*( u(i,j+1) + u(i,j-1) + u(i+1,j) + u(i-1,j) ;
     }
```

Encoding The Jacobi Expression – Compile Time

```
u(i,j) = 0.25*( u(i+1,j) + u(i-1,j) + u(i,j+1) + u(i,j-1) )
```



This is the abstract object that the generic kernel will see

Decoding The Jacobi Expression - Runtime

```
Template < typename ComplexType >
__global__ void computeKernel( ComplexType ctype, real* result )
{
  int TID = threadIDx.x + …

  result[TID] = ctype[TID];
}
```

MulRealGen<Real,Gen>

[ ] operators are overloaded

```
__device__ real operator [] ( int i )
{
  return  constant_val * v[i];
}
```

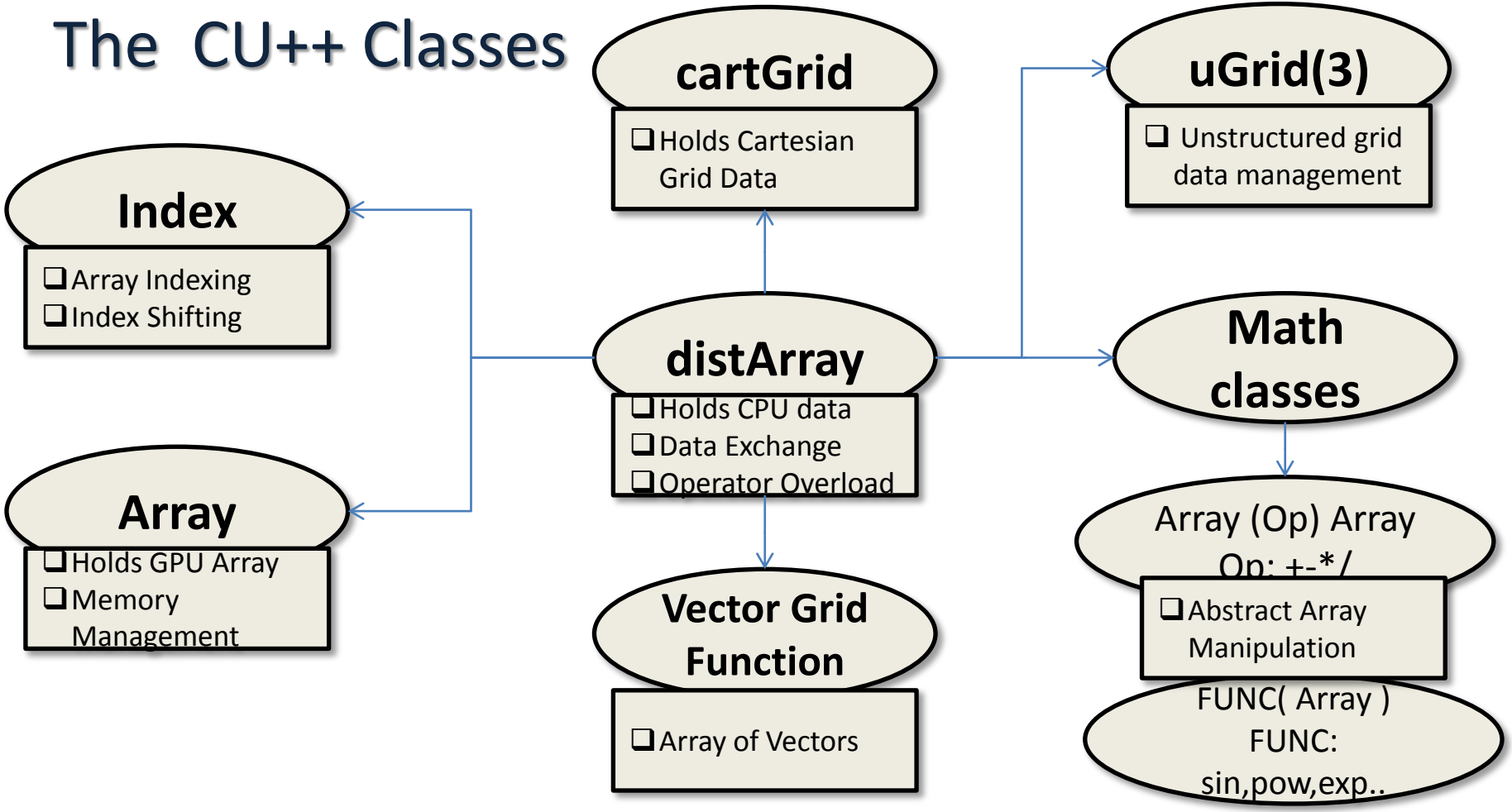
AddGenArray<Gen,Array>

AddGenArray<Gen,Array>

AddArrayArray<Array,Array>

The Tree expands during Run time

The CU++ Classes



The CU++ Classes

cartGrid

- Holds Cartesian Grid Data

uGrid

- Holds Unstructured Grid Data

Index

- Array Indexing
- Index Shifting

distArray

- Holds CPU data
- Data Exchange
- Operator Overload

Math classes

Array

- Holds GPU Array
- Memory Management

Array (Op) Array

Op: +-*/

- Abstract Array Manipulation

Vector Grid Function

- Array of Vectors

FUNC(Array)

FUNC: sin,pow,exp..

The CU++ Classes

cartGrid

- Holds Cartesian Grid Data

uGrid

- Holds Unstructured Grid Data

Index

- Array Indexing
- Index Shifting

distArray

- Holds CPU data
- Data Exchange
- Operator Overload

Math classes

Array

- Holds GPU Array
- Memory Management

Array (Op) Array

Op: +-*/

- Abstract Array Manipulation

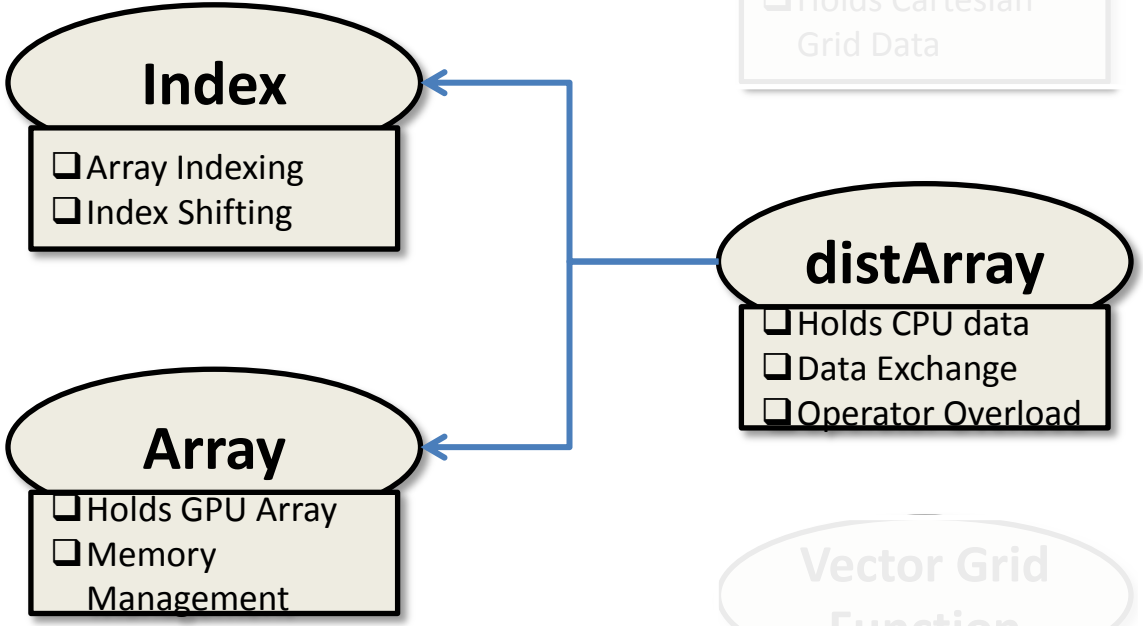
Vector Grid Function

- Array of Vectors

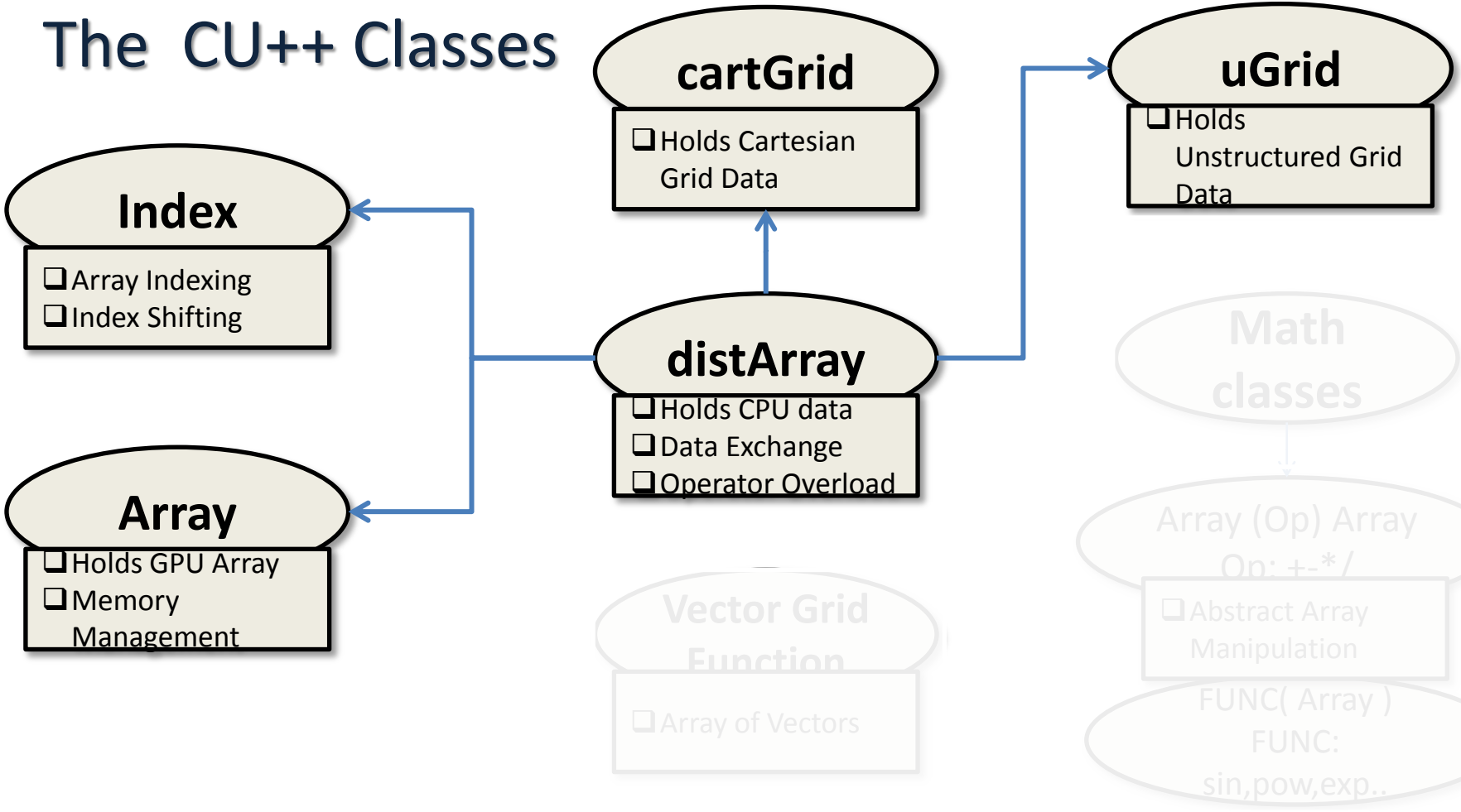
FUNC(Array)

FUNC: sin,pow,exp..

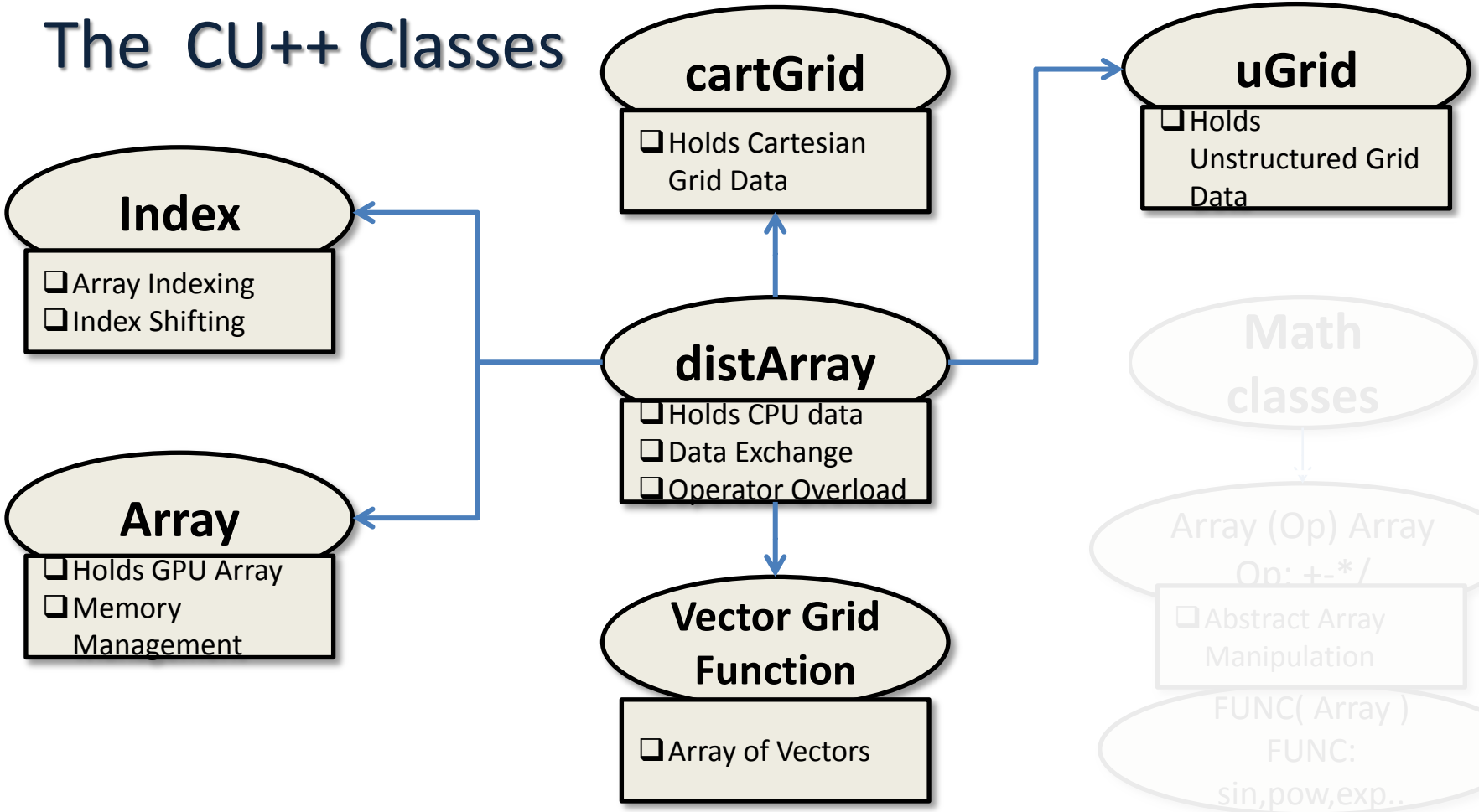
The CU++ Classes



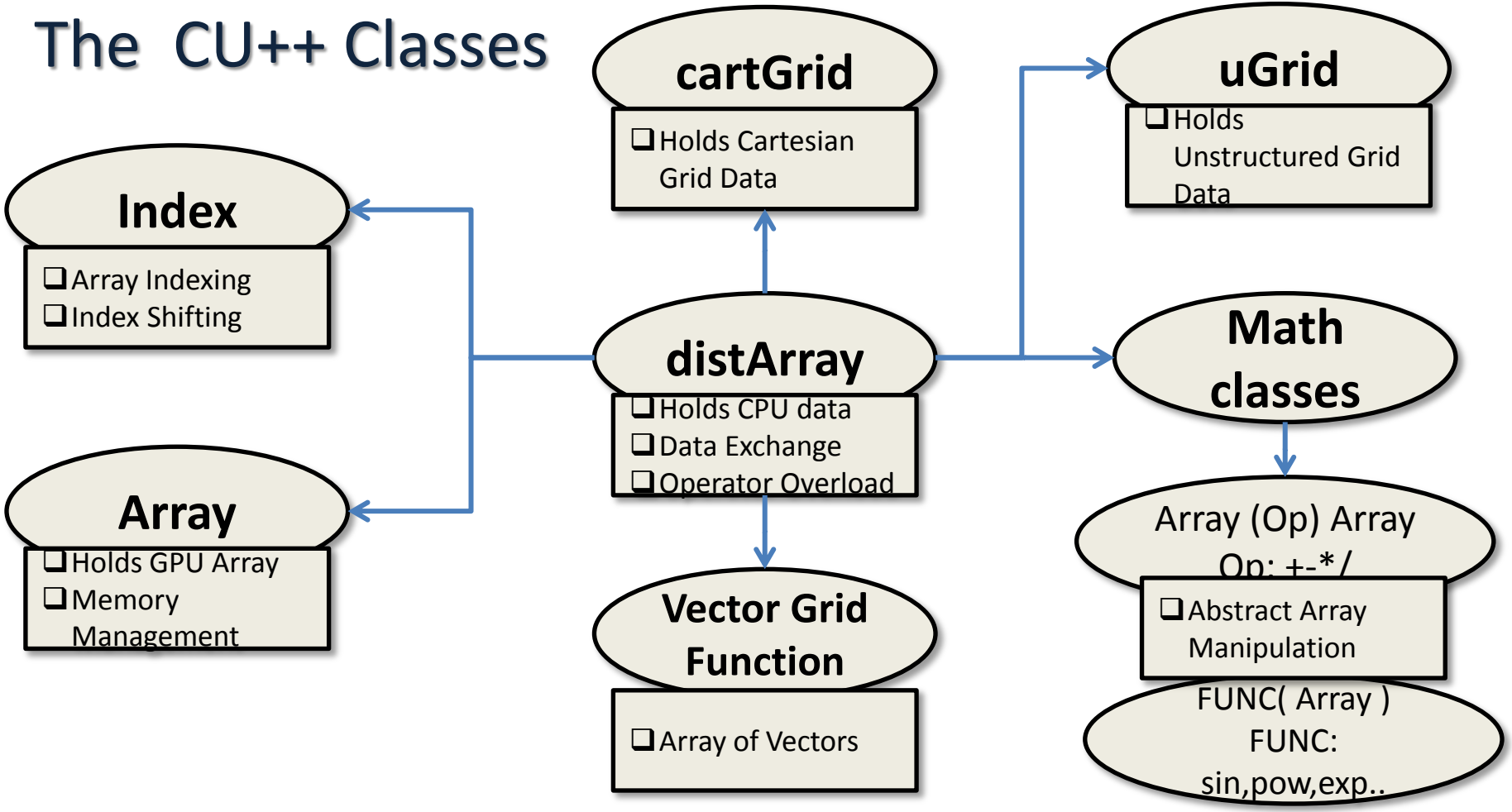
The CU++ Classes



The CU++ Classes



The CU++ Classes



CU++ Features : Array Assignment

We have a structured grid of size $Nx * Ny$, and we would like to fill the internal nodes with a constant value

C++ serial version

```
real *U = new real [Nx*Ny]  ;

for ( int i = 1; i < Ny-1 ; i++)
  for ( int j = 1 ; j < Nx-1 ; j++)
     U[i+j*Nx]=10.0;
```

CU++ET v on

```
distArray U(Nx,Ny);
Index I1(1,Nx-2), I2(1,Ny-2);
U(I1,I2)=10.0;
```

Parallel assignment

CU++ Features : Handling Unstructured Data

```
// Declare an array to hold the solution
distArray Q( number_of_nodes );

// Declare an array to hold the boundary node indices
distArray bNodeIndex( number_of_boundary_nodes) ;
Index I(0, number_of_boundary_nodes-1);

// Short Notation
#define BI bNodeIndex(I)

//Get the boundary  node indices
getBoundaryNodeIndex ( bNodeIndex )

// Do a small computation on the boundary nodes
Q(BI) = Q(BI) + SIN(x(BI))*COS(Y(BI));
```

```
                0  1   2   3   4  5 . . . .
bNodeIndex = [ 2  8  23  24  15 19 . . . .        ]
```

CU++ Features : Misc. Features

```
cartGrid cg(0,1,Nx,0,1,Ny,0,1,Nz
```

Creates a Cartesian Grid in Parallel

```
vectorGridFunction u(cg,2)
```

Array of vectors (2 components), each vector has dimension Nx*Ny*Nz

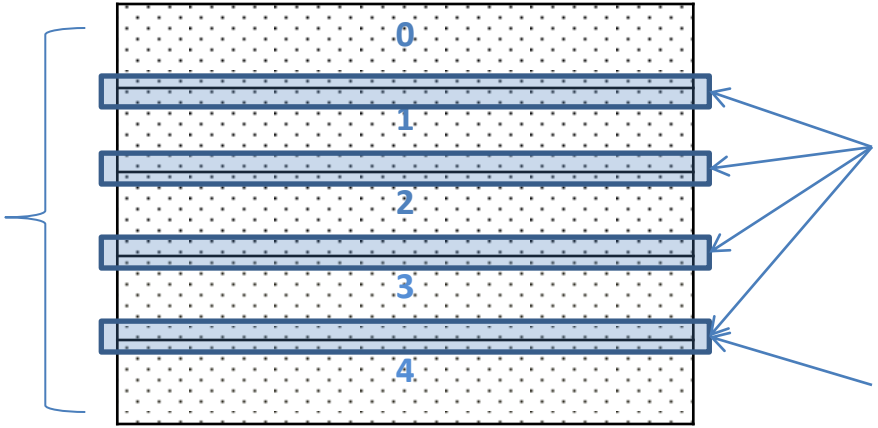
```
u[0](I1,I2,I3) = func()
```

Assigns an arbitrary function to the *first* component of *u*

```
u[1](I1+1,I2,I3) = POW(u[0](I1,I2,I3),2.1)
```

Assigns a function of the *first* component to the *second* component of *u* (shifted by 1)

CU++ Features : MPI Support



GPU Blocks
1-D Row-wise
Decomposition

Pull from GPU, and Send
Receive and Push to GPU

Halos for communication

Each GPU mapped by one CPU core

CU++ Features : Example Code, Poisson Solver

```
#include "CU++Runtime.h"
int main(int argc, char* argv[])
{
  // Problem size
  int Nx = 1000, Ny=1000, niter = 1e6;
  distArray::Init(argc,argv,Nx,Ny);

  // Create the partition type and declare the array 'u'
  ArrayPartition apobject(1,2,1);
  distArray u(Nx,Ny,apobject);

  // Indices of internal points
  Index i,j;
  u.getIndexOfInternalPoints(i,j);
```

CU++ Features : Example Code, Poisson Solver

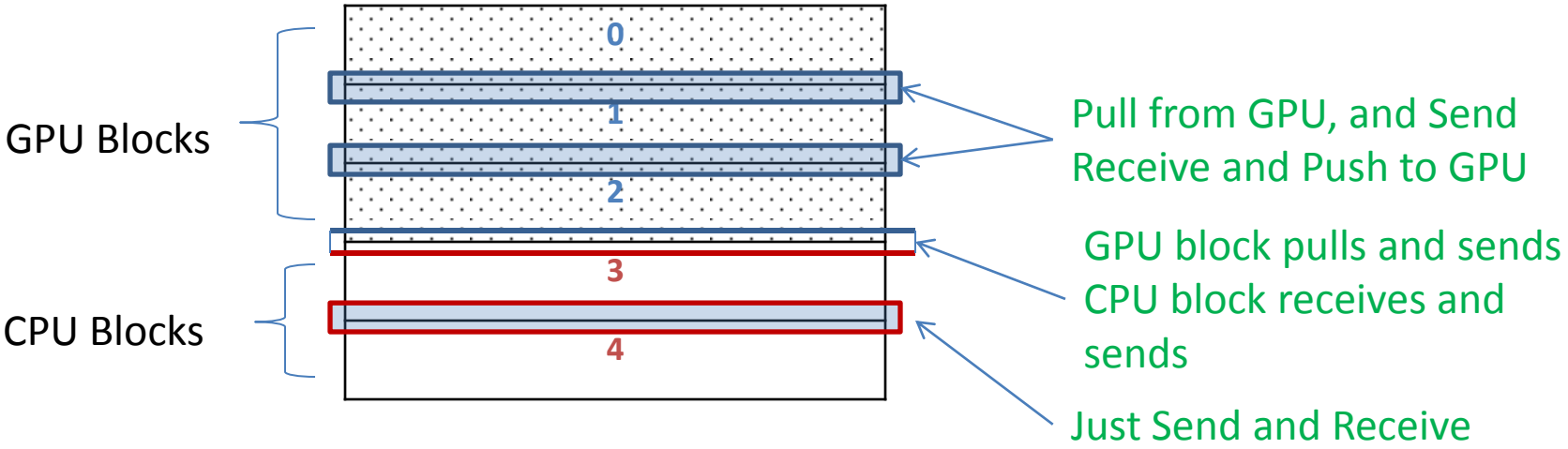
```
// Some constants
real dx = 1.0/(Nx+1);
real dy = 1.0/(Ny+1);
real lm = pow( (dx/dy),2.0);
real cont = 0.5/(1+lm);
real F = -2.0;

//  The main loop
for ( int step = 0 ; step < niter ; step++ )
   {
      u(i,j) = cont*( u(i+1,j) + u(i-1,j) +
                  lm*( u(i,j+1) + u(i,j-1)) -F*dx*dx );
      apobject.FixFringePoints(u);
   }
```

Idle CPUs, Make Them Work !

Each partitioned block knows whether it is a GPU block or a CPU block –

```
bool distArray::iamAGPUBlock = true/ false
```



Needs to be load balanced for achieving speed-up

Load Balancing math – GPU + CPU cores

Assume the following variables:

T: *Total Problem size*

s: *1 GPU/1CPU speed up*

n_g: *Number of GPUs*

n_c: *Number of CPUs*

N_1: *Problem size on GPU*

N_2: *Problem size on a CPU core*

The total problem size can be computed as:

$$T = n_g N_1 + n_c N_2$$

For the load to be balanced between a CPU core and GPU:

$$N_1 = sN_2$$

Using the above relations, we obtain

$$N_1 = \frac{sT}{n_g s + n_c} \quad , N_2 = \frac{T}{n_g s + n_c}$$

If only GPUs are used, then the time spent by **each GPU is**

$$t_{GPU1} \sim \frac{T}{n_g}$$

If the load is shared between GPUs and CPUs, the time spent by **each GPU** is

$$t_{GPU2} \sim N_1$$

Thus Speed-up of the GPU gained by sharing the load with the CPU is

$$\frac{t_{GPU1}}{t_{GPU2}} \sim \frac{T}{n_g N_1} \sim 1 + \left(\frac{n_c}{n_g}\right)\frac{1}{s}$$

Load Balancing math – GPU + CPU cores

$$\text{So Speed-up} \sim 1 + \left(\frac{n_c}{n_g}\right)\frac{1}{s}$$

- Let us assume $s \sim 10$ (common for unstructured grid solvers on GPU), and you have a 16 core CPU, with 4 GPUs.
- You can push for an additional 40% speed-up if you use the CPU cores also.

Speed-up figures for the Jacobi Problem, 8 core CPU, with 7 GPUs

GPU Kernel purposely slowed down to achieve 10x speed-up wr.t a single core (s=10)

nGPUs	N Cores	Theoretical	Actual
1	7	1.7	1.51
2	6	1.3	1.25
4	4	1.1	1.1
6	2	1.03	1.02
7	1	1.01	1.02

Compiling and Executing CU++ Codes – mpiugc compiler tool

Compile for GPU compute capability 2.0

```
$ mpiugc -arch=sm_20 <program.cu> -o exe
```

Run just on 1 CPU (serial) – Same source code, no gpus

```
$ mpirun -np 1 <exe> -ngpu 0
```

Run on 1 GPU using 1 CPU core: 1 CPU core manages the GPU

```
$ mpirun -np 1 <exe> -ngpu 1
```

Compiling and Executing CU++ Codes – mpiugc compiler tool

Run on 6 GPUs using 6 CPU processes : 6 CPUS Just manage the GPUs

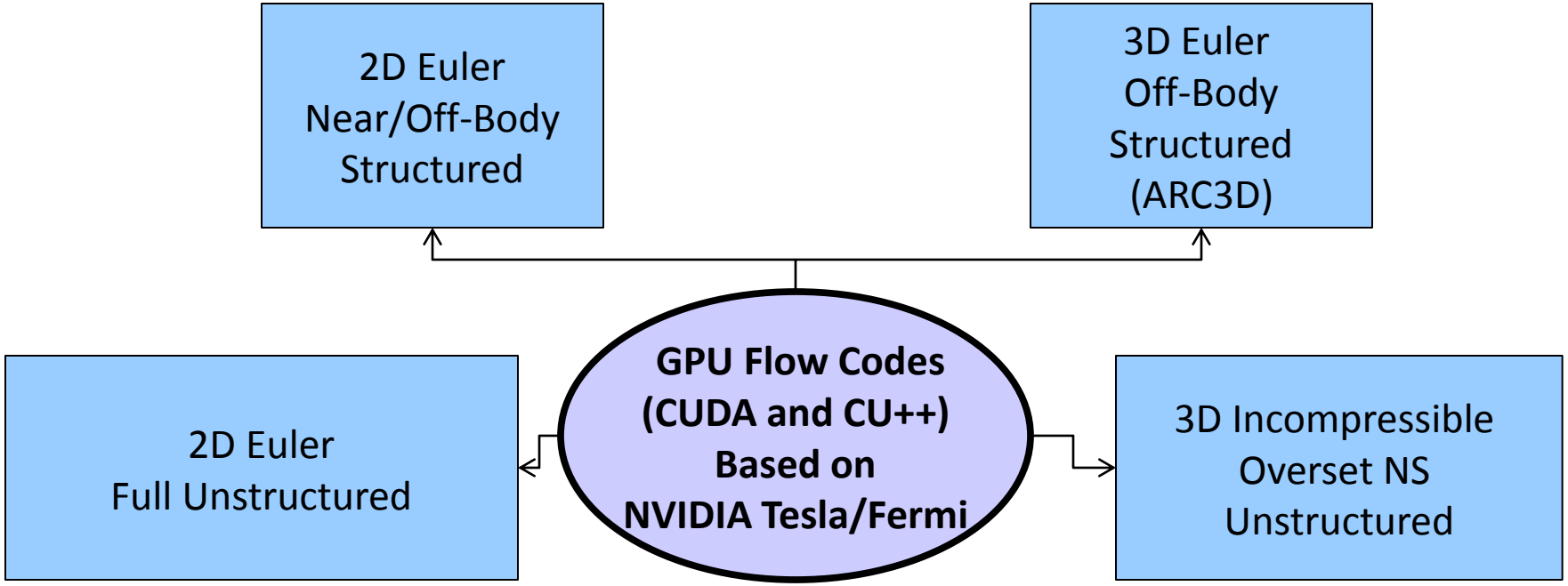
```
$ mpirun -np 6 <exe> -ngpu 6
```

What if you have a 16 core CPU and 4 GPUs and you want to
Use all cores and all GPUs (except the cores that are managing the GPU ?)

```
$ mpirun -np 16 <exe> -ngpu 4
```

- Processes 0-3 runs on the GPU
- Processes 4-15 run on the CPU

CU++ Applications



Acknowledgments

- *Support from the office of Naval Research under ONR grant N00014-09-1-1060 is gratefully acknowledged.*

- *NVIDIA for providing hardware*