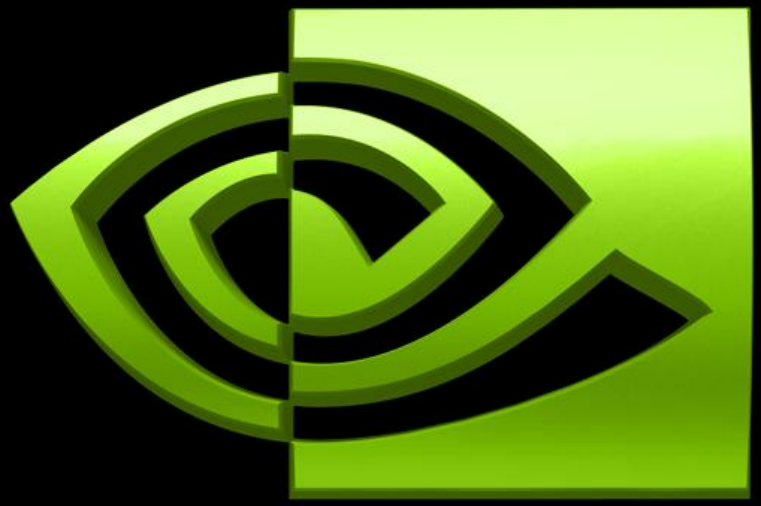


# OpenCurrent: Solving PDEs on Structured Grids with CUDA

**Jonathan Cohen**

NVIDIA Research


# OpenCurrent

# Why OpenCurrent?

- Teaching tool for community
  - 1300 downloads
  - Several projects “based on” the code w/o using it directly
- Benchmark whole application, not micro benchs
  - 22k lines of code, 57 CUDA kernels
  - Built-in unit tests and validation
- Research Vehicle: Software development paradigms
  - How to structure a complex API for CUDA? Multi-GPU?
  - Coding practices to balance perf / productivity
- Research Vehicle: Algorithmic development
  - E.g. sparse linear solvers in CUDA

# Research Question

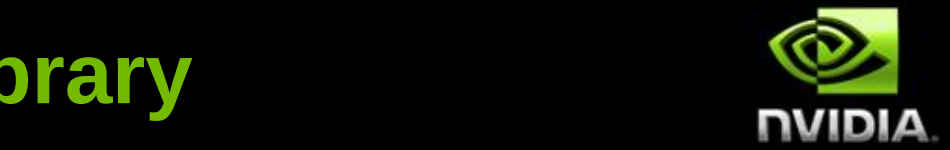
- If you were to rewrite code from scratch:
  - What is a good overall design?
  - What global optimizations are possible?
  - Is it worth it?
- Major Take-away: Avoid All Serial Bottlenecks
- In particular: Avoid All PCIE Transfers
  - => Move Everything to GPU
  - => Don’t want to sacrifice productivity

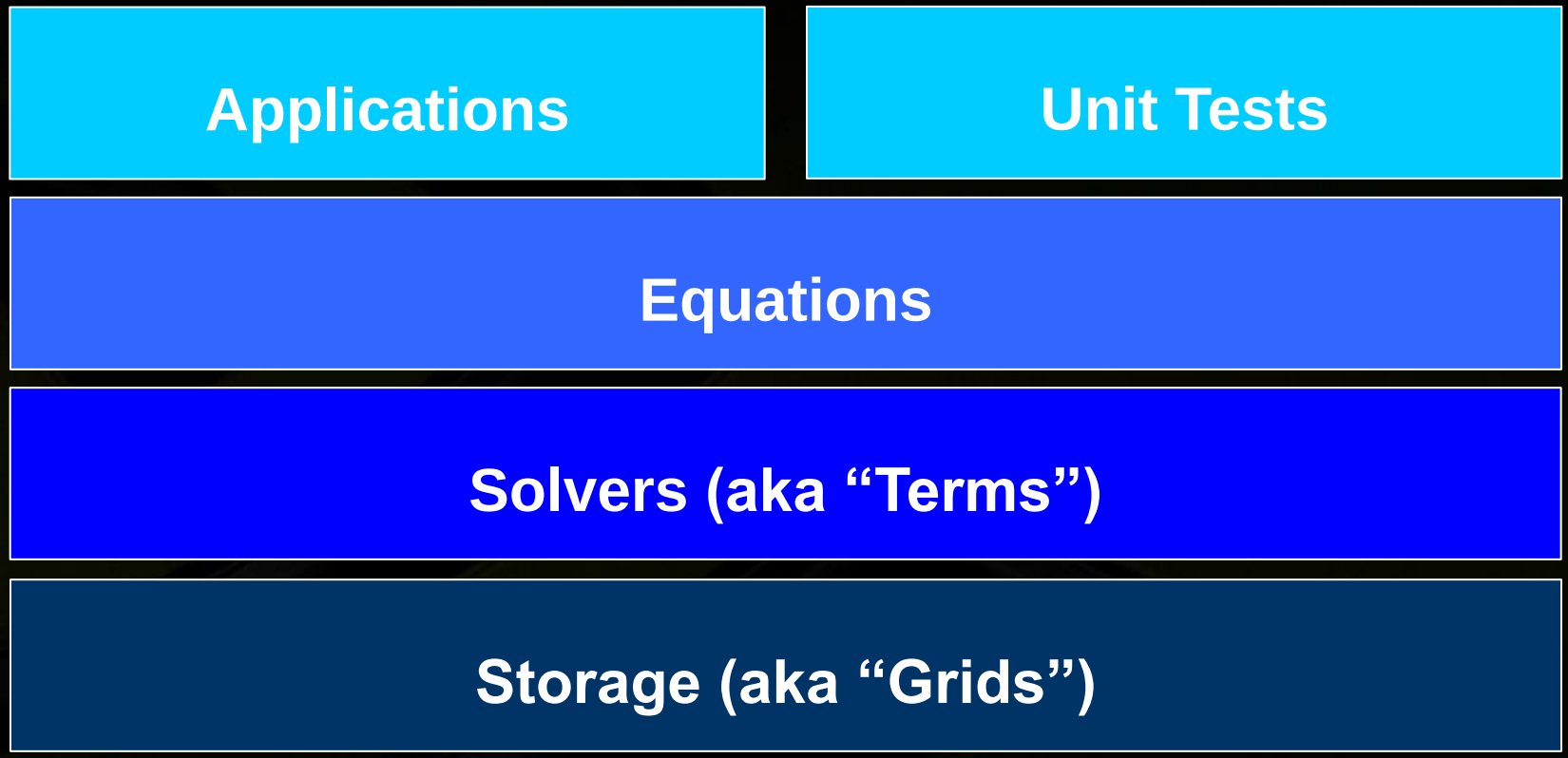
# Getting started

- Download v1.1 from googlecode:
  - http://code.google.com/p/opencurrent/downloads/list
  - Apache 2.0 OS license
- Requires CMake to build (cross-platform make)
  - http://www.cmake.org/cmake/resources/software.html
- CUDA 3.0 or greater
  - www.nvidia.com/object/cuda_get.html
- Optional: NetCDF 3.6.1 or 4.0

- Configure build, then go.
- Instructions: http://code.google.com/p/opencurrent/wiki/OpenCurrent

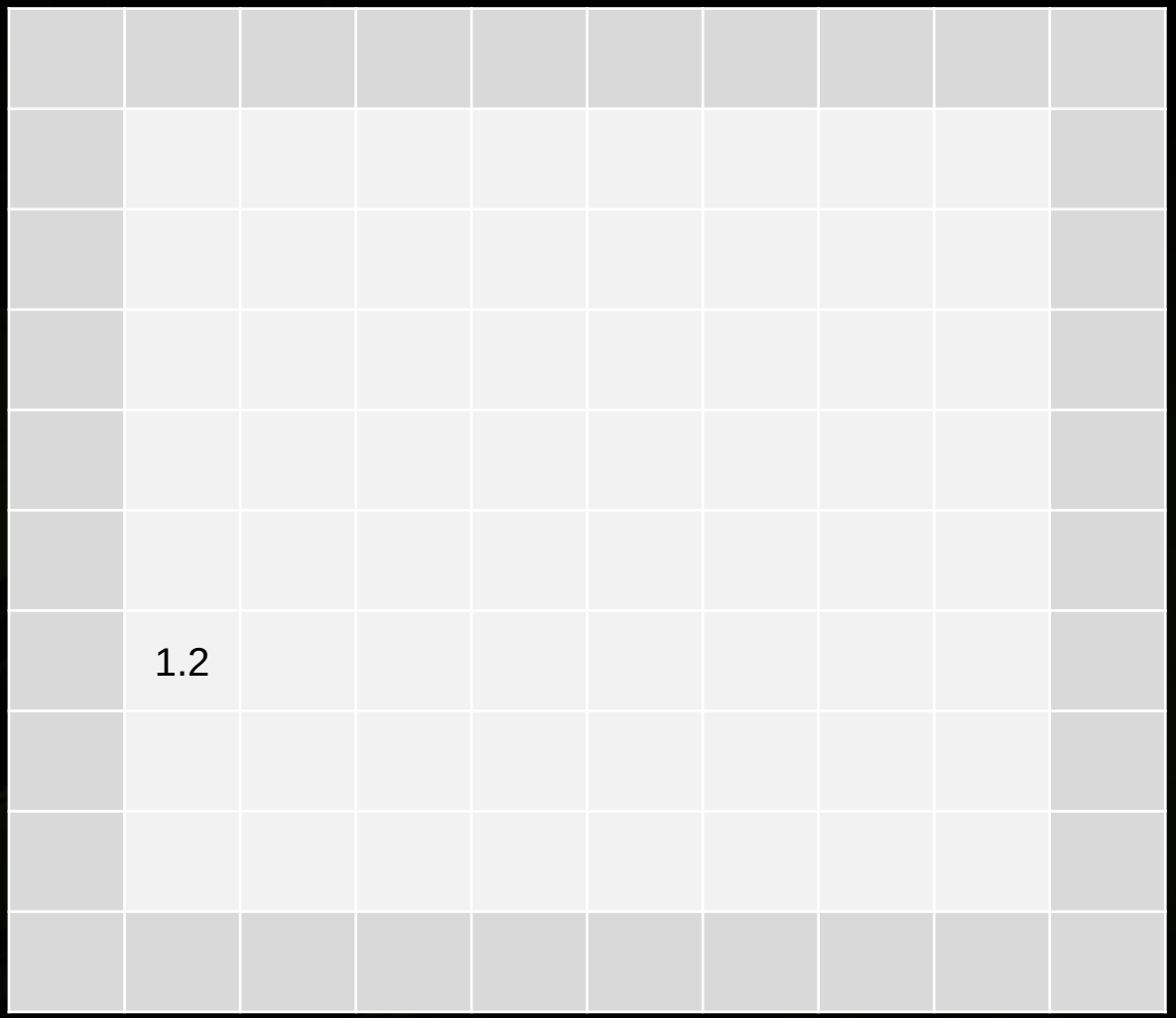
# The OpenCurrent Library



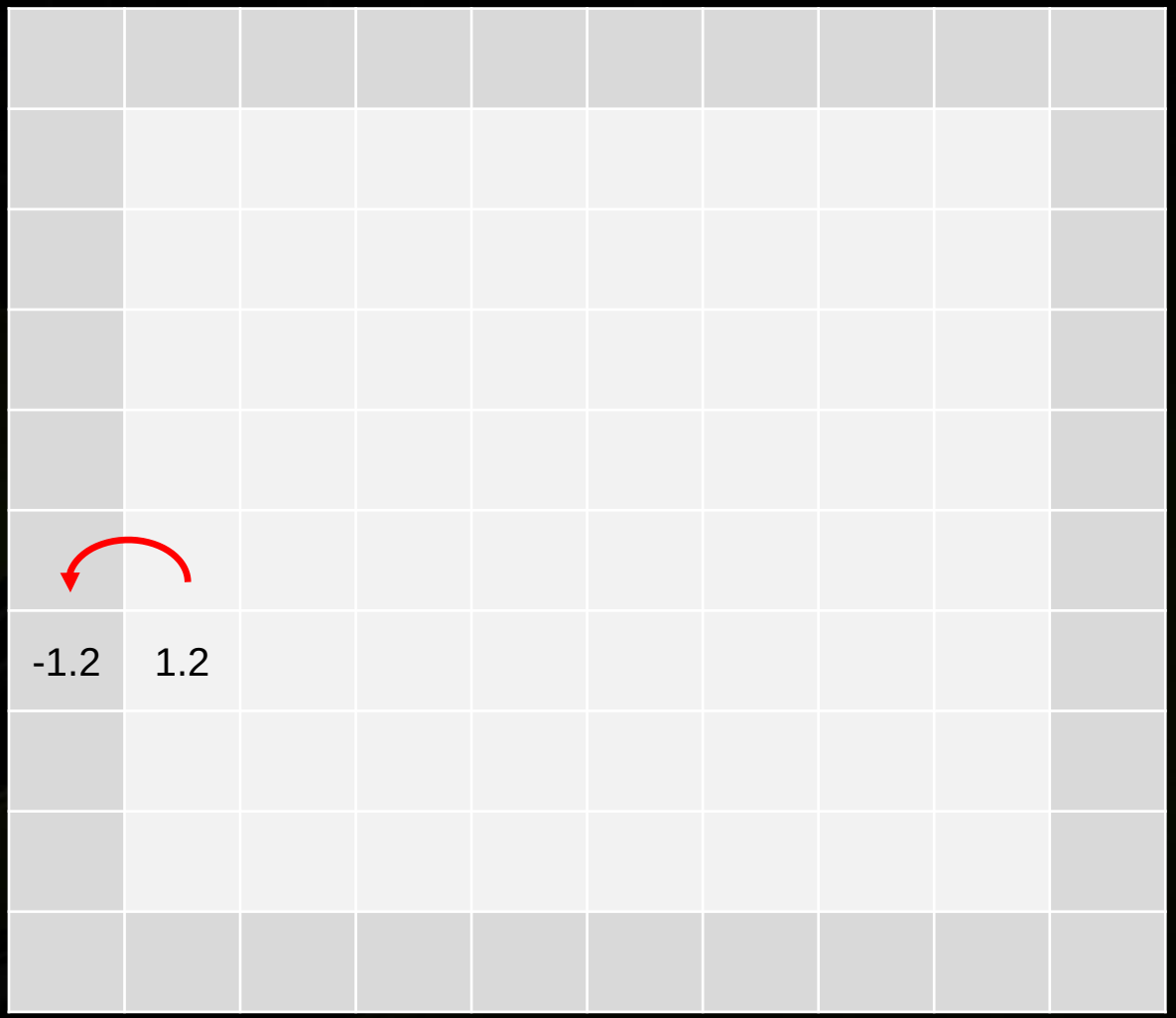



# Storage Classes: Grids

# Grid2DDeviceD – 8x8, padding=1



# Grid2DDeviceD – 8x8, padding=1




**Enforce Dirichelet Condition: value at border = 0**

# Equation Example – Navier-Stokes

# Background: Fluid Mechanics

- Incompressible Navier-Stokes Equations

1. $\partial u/\partial t = -u \cdot \quad u - \quad p$ + Viscosity + …
2. $\cdot u = 0$

- Boundary value problem
- Given u at t=0, first equation defines how to advance forward in time (time marching)
- Second equation is a constraint on pressure

# Solution Strategy:

[1] $\partial u/\partial t = -u \cdot \nabla u - \nabla p + \text{Visc.} + \ldots$

$u^{t+1} = u^t + \Delta t \left[ \partial u/\partial t \right]$ [1]

$u^{t+1} = u^t + \Delta t \left[ -u \cdot \nabla u - \nabla p + \text{Visc.} + \ldots \right]$

3. $\hat{u}^{t+1} = u^t + \Delta t \left[ -u \cdot \nabla u + \text{Visc.} + \ldots \right]$

4. $u^{t+1} = \hat{u}^{t+1} - \Delta t \nabla p$

# What is pressure?

[4] $u^{t+1} = \hat{u}^{t+1} - \Delta t \quad p$

[2] $\cdot u = 0$ [2nd Navier Stokes Equation]

[4]

$\cdot[u^{t+1}] = 0$

$\cdot[\hat{u}^{t+1} - \Delta t \quad p] = 0$

$\cdot \Delta t \quad p = \quad \cdot\hat{u}^{t+1}$

5. $\Delta t \quad ^2p = \quad \cdot\hat{u}^{t+1}$

# Basic CFD Algorithm

Input: Value of $u$ at $t$=0

For $t$ = 1,2,…

1. Time march to get $\hat{u}$

   $\hat{u}^{t+1} = u^t + \Delta t\,[-u\cdot\quad u$ + viscosity + …]

2. Solve for $p$

   $\Delta t\quad {}^2p = \quad \cdot\hat{u}^{t+1}$

3. Subtract $p$ to get $u$

   $u^{t+1} = \hat{u}^{t+1} - \Delta t\quad p$

# Navier-Stokes Equation class

```
class Eqn_IncompressNS3DD : public Equation
{
  Grid3DDeviceD                    _u, _v, _w;
  Grid3DDeviceD                    _derivu_dt, ...;
  ...
  Sol_ProjectDivergence3DDeviceD   _projection_solver;
  Sol_LaplacianCentered3DDeviceD   _u_diffusion_solver, ...;
  ...

public:

  void advance_one_step(double dt) {
    _advection_solver.solve();   // update _derivX_dt
    _u_diffusion_solver.solve(); // update _derivu_dt
    _v_diffusion_solver.solve(); // update _derivv_dt
    _w_diffusion_solver.solve(); // update _derivw_dt

    _u.linear_combination(1.0, _u, dt, _deriv_udt));
    _v.linear_combination(1.0, _v, dt, _deriv_vdt));
    _w.linear_combination(1.0, _w, dt, _deriv_wdt));

    _projection_solver.solve(1e-8); // compute & add pressure term
  }
};
```

# Projection Solver class

```
class Sol_PressureProjection3DDeviceD : public Solver
{
  Sol_MultigridPressure3DDeviceD pressure_solver; // contains p
  Sol_Divergence3DDeviceD        divergence_solver;
  Sol_Gradient3DDeviceD          gradient_solver;
  Grid3DDeviceD                 *u, *v, *w; // updated in place
  Grid3DDeviceD                  divergence;

public:

  double solve(double tolerance) {
    double residual;
    apply_3d_mac_boundary_conditions_level1(u,v,w);
    divergence_solver.solve(); // updates divergence
    pressure_solver.solve(residual, tolerance, 15); // calc p
    gradient_solver.solve(); // update u,v,w
    apply_3d_mac_boundary_conditions_level1(u,v,w);
    return residual;
 }
};
```

# How to "Solve for *p*"?

[5] $\Delta t \quad {}^2p = \quad \cdot \hat{u}^{t+1}$

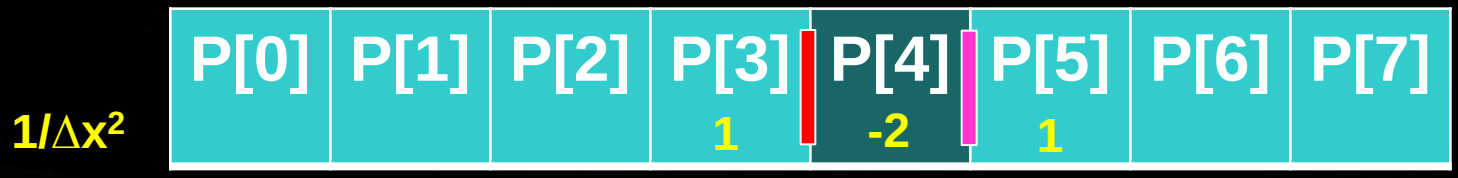
$${}^2p = \frac{\cdot \hat{u}^{t+1}}{\Delta t}$$

(Laplacian of *p* = sum of second derivatives)

$$\partial^2 p/\partial x^2 + \partial^2 p/\partial y^2 + \partial^2 p/\partial z^2 = \text{rhs}$$

$$\partial^2 p/\partial x^2 \approx \frac{\partial(p+\Delta x)/\partial x - \partial p/\partial x}{\Delta x}$$

## To compute Laplacian at P[4]:




## 1st Derivatives on both sides:

$$\partial p/\partial x = \frac{P[4] - P[3]}{\Delta x} \qquad \partial(p+\Delta)/\partial x = \frac{P[5] - P[4]}{\Delta x}$$

## Derivative of 1st Derivatives:

$$\frac{\frac{P[5] - P[4]}{\Delta x} - \frac{P[4] - P[3]}{\Delta x}}{\Delta x} \Rightarrow 1/\Delta x^2 \, ( P[3] - 2P[4] + P[5] )$$

# Pressure Solver: Poisson Matrix

**Poisson Equation is a sparse linear system**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -2 | 1 | | | | | | 1 |
| 1 | -2 | 1 | | | | | |
| | 1 | -2 | 1 | | | | |
| | | 1 | -2 | 1 | | | |
| | | | 1 | -2 | 1 | | |
| | | | | 1 | -2 | 1 | |
| | | | | | 1 | -2 | 1 |
| 1 | | | | | | 1 | -2 |

| |
|---|
| P[0] |
| P[1] |
| P[2] |
| P[3] |
| P[4] |
| P[5] |
| P[6] |
| P[7] |

$= \Delta x^2$ RHS

# Iterative Poisson Solver

Solve M p = r, where M and r are known

Error is easy to estimate: E = M p' - r

Basic iterative scheme:

Start with a guess for p, call it p'

Until | M p' - r | < tolerance

    p' <= Update(p', M, r)

Return p'

# Serial Gauss-Seidel Relaxation

**Loop until convergence:**

**For each equation j = 1 to n**

**Solve for P[j]**

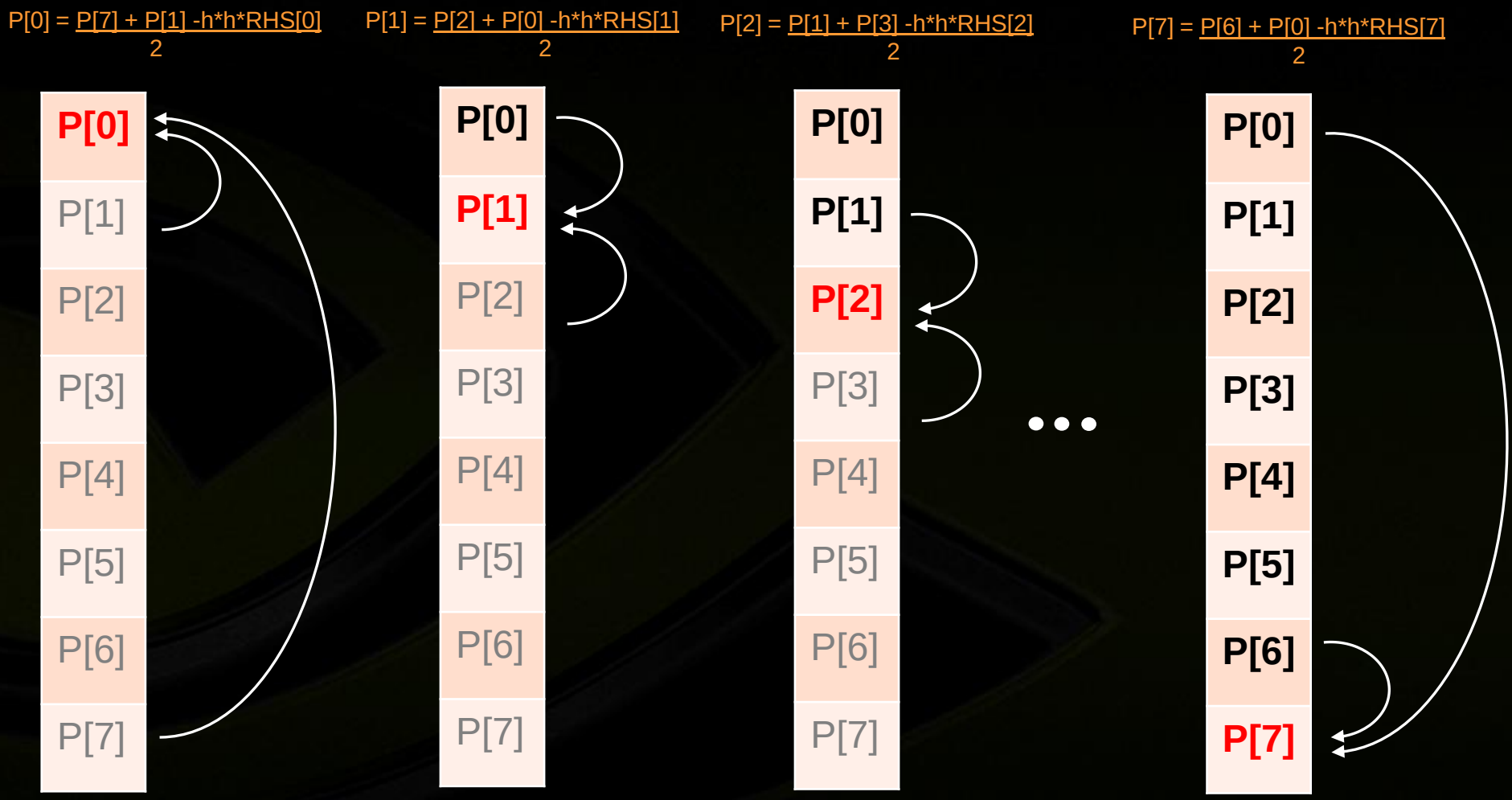
**E.g. equation for** P[1]:

$$P[0] - 2P[1] + P[2] = h*h*RHS[1]$$

**Rearrange terms:**

$$P[1] = \frac{P[0] + P[2] - h*h*RHS[1]}{2}$$

# One Pass of Serial Algorithm



# Red-Black Gauss-Seidel Relaxation

- Can choose any order in which to update equations
  - Convergence rate may change, but convergence still guaranteed
- "Red-black" ordering:


- Red (odd) equations independent of each other
- Black (even) equations independent of each other

# Parallel Gauss-Seidel Relaxation


Loop n times (until convergence)

    For each even equation j = 0 to n-1

        Solve for P[j]

    For each odd equation j = 1 to n

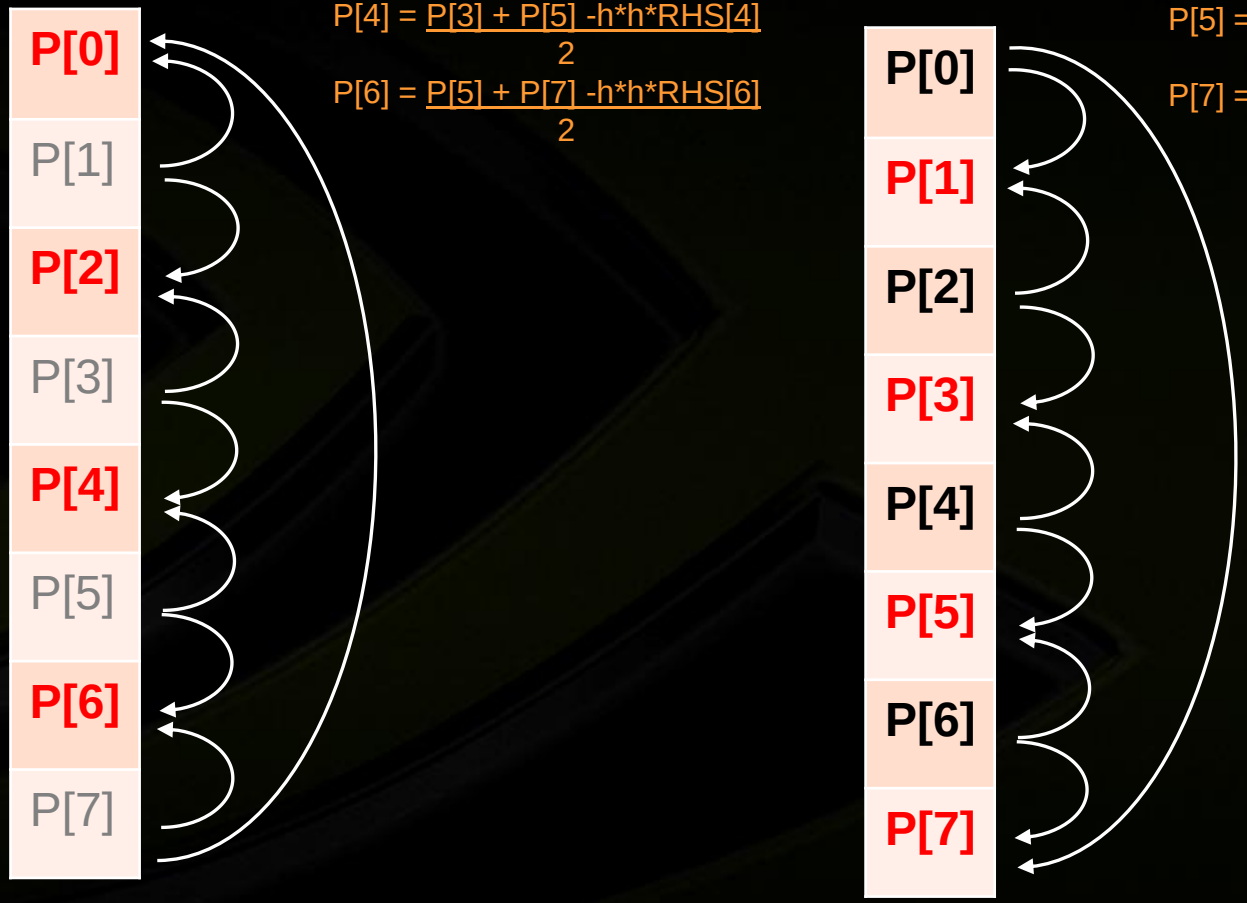
        Solve for P[j]

For loops are parallel – perfect for CUDA kernel

# One Pass of Parallel Algorithm

$$P[0] = \frac{P[7] + P[1] - h*h*RHS[0]}{2}$$
$$P[2] = \frac{P[1] + P[3] - h*h*RHS[2]}{2}$$
$$P[4] = \frac{P[3] + P[5] - h*h*RHS[4]}{2}$$
$$P[6] = \frac{P[5] + P[7] - h*h*RHS[6]}{2}$$

$$P[1] = \frac{P[0] + P[2] - h*h*RHS[1]}{2}$$
$$P[3] = \frac{P[2] + P[4] - h*h*RHS[3]}{2}$$
$$P[5] = \frac{P[4] + P[6] - h*h*RHS[5]}{2}$$
$$P[7] = \frac{P[6] + P[0] - h*h*RHS[7]}{2}$$

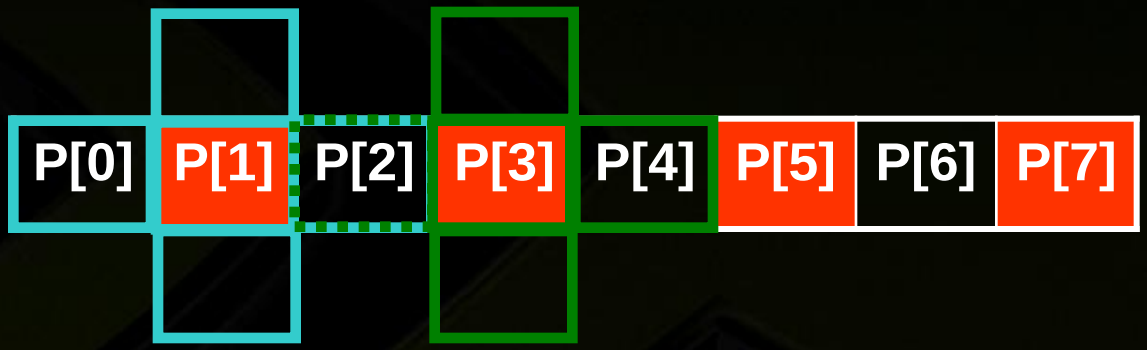


# Relaxation Pseudo-code

```
__global__ void RedBlackGaussSeidel(
  Grid P, Grid RHS, float h, int red_black)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  int j = blockIdx.y*blockDim.y + threadIdx.y;
  i*=2;
  if (j%2 != red_black) i++;
  int idx = j*RHS.jstride + i*RHS.istride;
  P.buf[idx] = 1.0/4.0*(-h*h*R.buf[idx] +
       P.buf[idx + P.istride] + P.buf[idx - P.istride] +
       P.buf[idx + P.jstride] + P.buf[idx - P.jstride]);
}

// on host
relax(int nu) {
  for (int i=0; i < nu; i++) {
    RedBlackGaussSeidel<<<Dg, Db>>>(P, RHS, h, 0);
    RedBlackGaussSeidel<<<Dg, Db>>>(P, RHS, h, 1);
  }
}
```

# Optimizing the Poisson Solver

- Red-Black scheme is cache friendly
  - Lots of reuse between adjacent threads (blue and green)




- Software-managed cache in shared memory get complex fast
- Naïve kernel =>

  good balance between code complexity / performance

# The problem with Gauss-Seidel

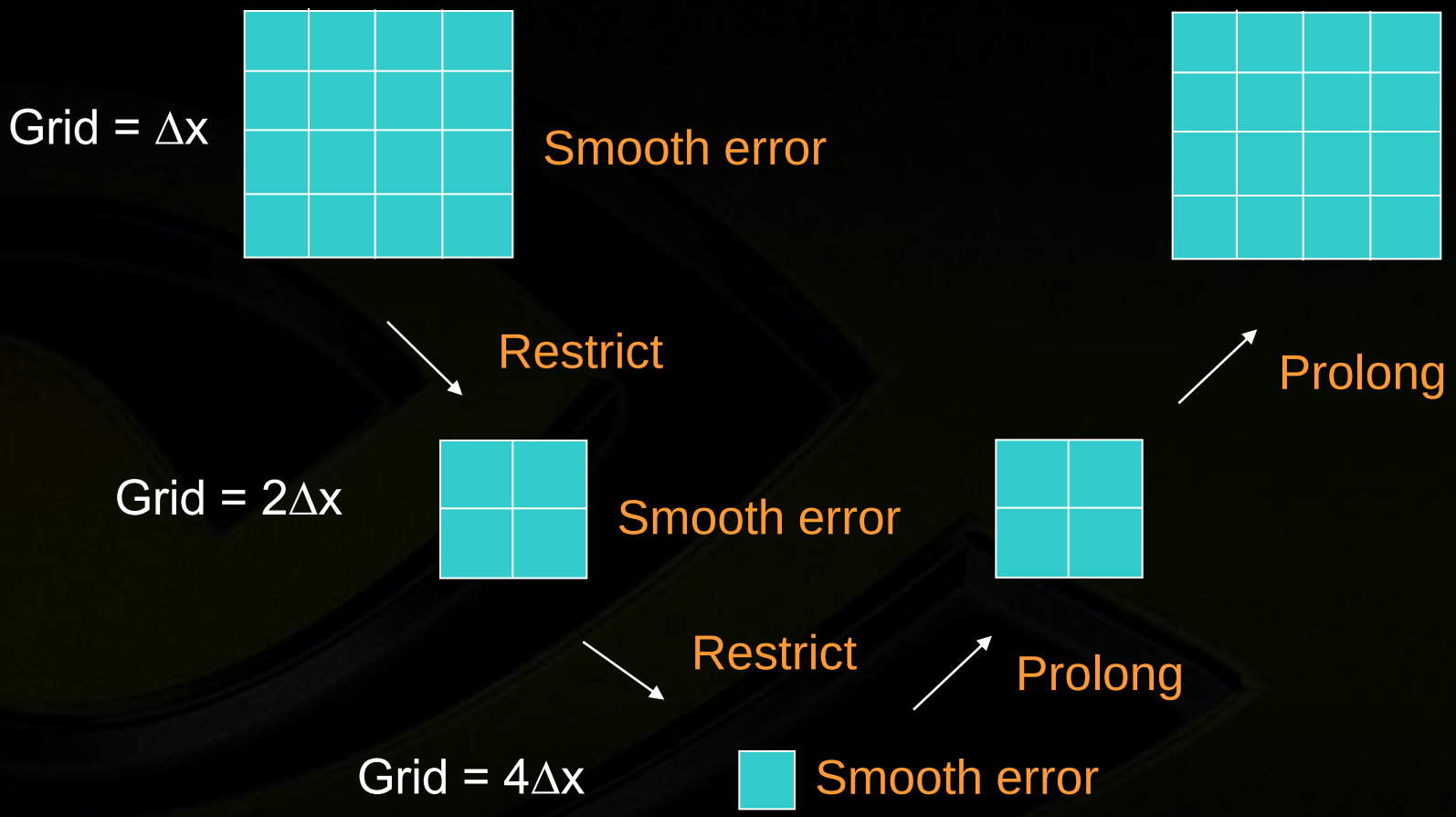
- Split Error into two parts:
  - local (high frequency) vs. global (low frequency)
- Update equations entirely local (hence parallel):

$$P[i] = \frac{P[i-1] + P[i+1] - h*h*RHS[i]}{2}$$

- Local error is reduced quickly
- Global error is reduced very slowly

- => G.S. reduces error at grid scale only ($\Delta x$) – "smoothes" errors, but doesn't reduce overall

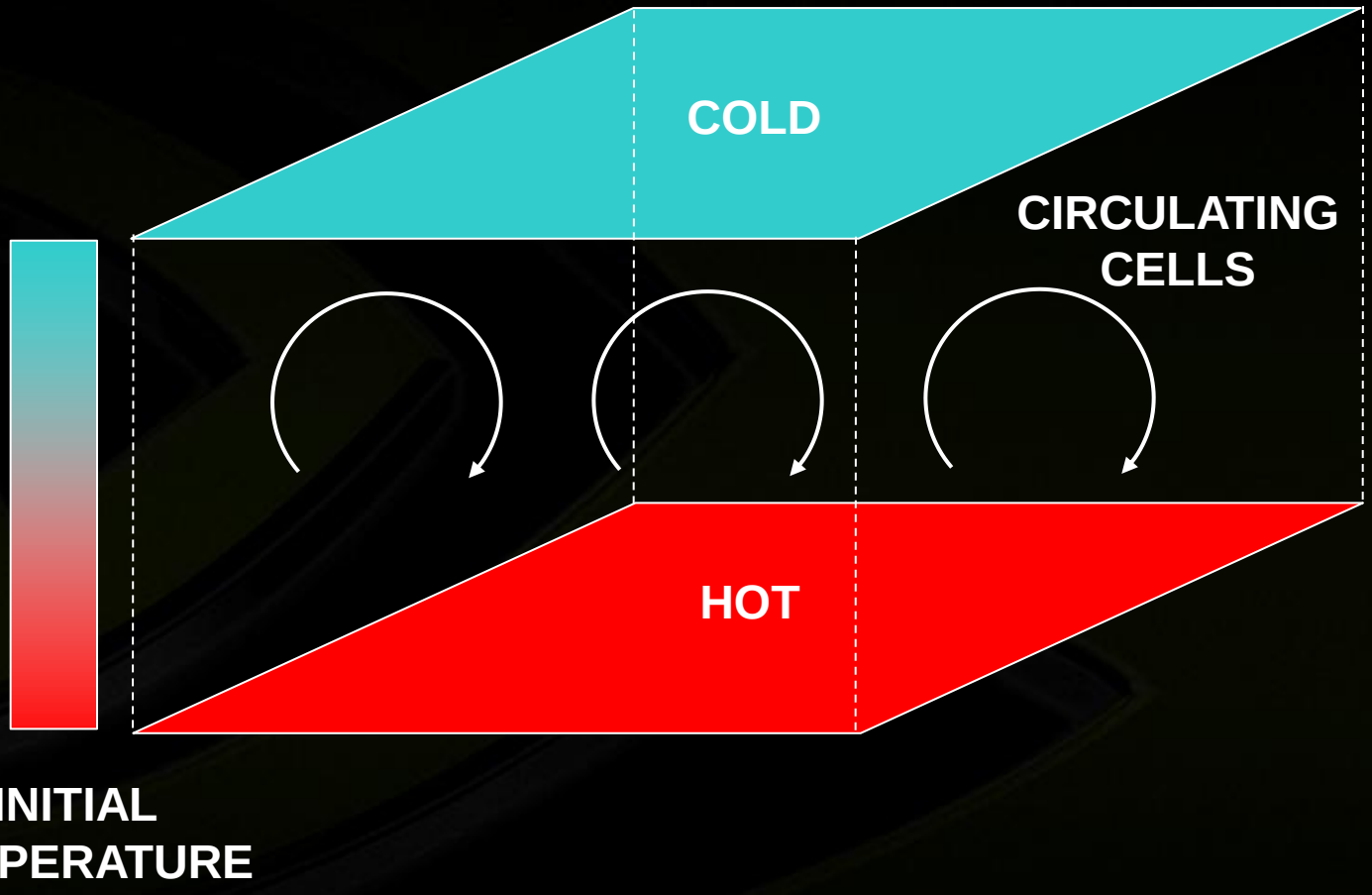
# Multigrid: G.S. at multiple scales



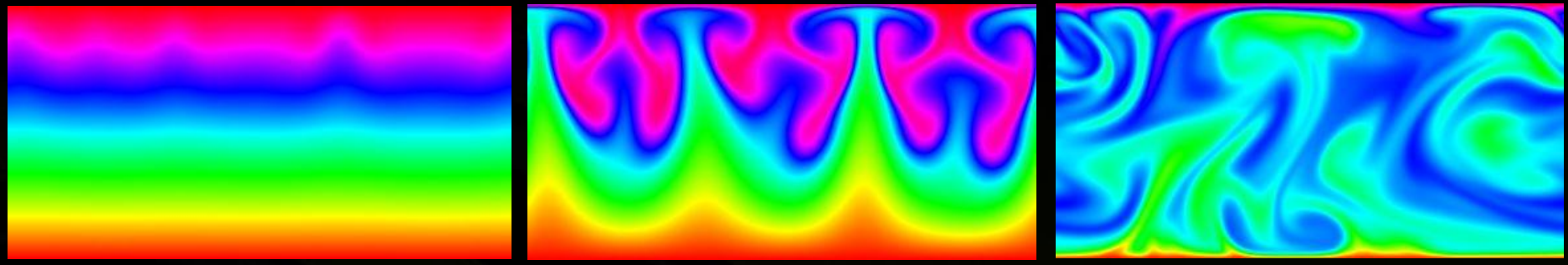
# N.S. Equation Functionality

- Incompressible Navier-Stokes with Boussinesq approximation for thermodynamics
- Globally second-order accurate (time and space)
  - Finite Volume 2nd order centered discretizations
  - Adams-Bashford-2 time stepper
  - Projection method with multigrid pressure solver
- No turbulence model
- Cartesian grid
- Variety of boundary conditions
  - full-slip, no-slip, periodic, Dirichelet / Neumann thermal

# Putting It All Together: Rayleigh-Bénard Convection



# Rayleigh-Bénard Details



- Double precision, second order accurate
- 384 x 384 x 192 grid (max that fits in 4GB)
- Transition from stratified (left) to turbulent (right)
- Validated critical Rayleigh number against theory
- Validated / benchmarked more complex problems against published results & Fortran code

# Validation

- Calculate $Ra_{cr}$ via perturbation analysis
  - Analytical result is 657.51
  - Fully tests linear part of solver

| Resolution | Value | Difference |
|---|---|---|
| 16 x 8 x 16 | 659.69 | - |
| 32 x 16 x 32 | 658.05 | 1.64 |
| 64 x 32 x 64 | 657.65 | 0.40 |
| 128 x 64 x 128 | 657.54 | 0.11 |
| Extrapolated | 657.51 | |

- Non-linear part validated with Nusselt number
  - Ratio of convective heat flux to diffusive heat flux
  - Match published number (no analytic result)

# Benchmark Results

- CUDA (1 Tesla C1060) vs. Fortran (8-core 2.5 GHz Xeon)
- As “apples-to-apples” as possible ($ and manpower)
  - Equal price nodes (~$3k)
  - Skilled programmers in each paradigm

| Resolution | CUDA time/step | Fortran time/step | Speedup |
|---|---|---|---|
| 64 x 64 x 32 | 24 ms | 47 ms | 2.0x |
| 128 x 128 x 64 | 79 ms | 327 ms | 4.1x |
| 256 x 256 x 128 | 498 ms | 4070 ms | 8.2x |
| 384 x 384 x 192 | 1616 ms | 13670 ms | 8.5x |

# Fermi: Single vs Double Precision

- Identical simulation, only difference is precision of buffers & math routines
- Fermi C2050, ECC off (~2x faster than Tesla C1060)
- fp64 incurs penalty of 15% - 49% (< 2x)

| Resolution | fp64 time/step | fp32 time/step | Ratio |
|---|---|---|---|
| $64^3$ | 0.012349 | 0.010778 | 1.15x |
| $128^3$ | 0.041397 | 0.030348 | 1.36x |
| $256^3$ | 0.311494 | 0.208250 | 1.49x |

# Multiple GPU Support

# Synchronous Co-Array Model

- Grid split across multiple GPUs via extra ‘co-dimension’
  ```
  Grid3DDeviceCoD u(“u”); // id so arrays can find each other
  ```
- Access neighbor via grid.co(NBR_ID)
  ```
  Grid3DDeviceCoD *nbr = u.co(this_image()+1);
  ```
- Specify region locally or remotely
  ```
  Region a = u.co(NORTH)->region(0)()(); //remote
  Region b = u.region(0,10)(0,10)(0,10); // local
  ```
- All cooperative routines => all threads participate
  (bulk synchronous)
- Static declaration of communication pattern
  ```
  int handle = CoArrayManager::barrier_allocate(to, from);
  ```
- Dynamic exchange of data
  ```
  CoArrayManager::barrier_exchange(handle)
  ```

# Example: ghost node exchange

```
#pragma omp parallel
{
    initialize_image();
    Grid3DDeviceCoD A("A");

    int east = (this_image() + 1) % num_images();

    Region3D from = A.           region(nx-1)()();
    Region3D to   = A.co(east)->region(  -1)()();

    int handle = barrier_allocate(from, to);

    …
    barrier_exchange(handle);
    …

    barrier_deallocate(handle);
}
```

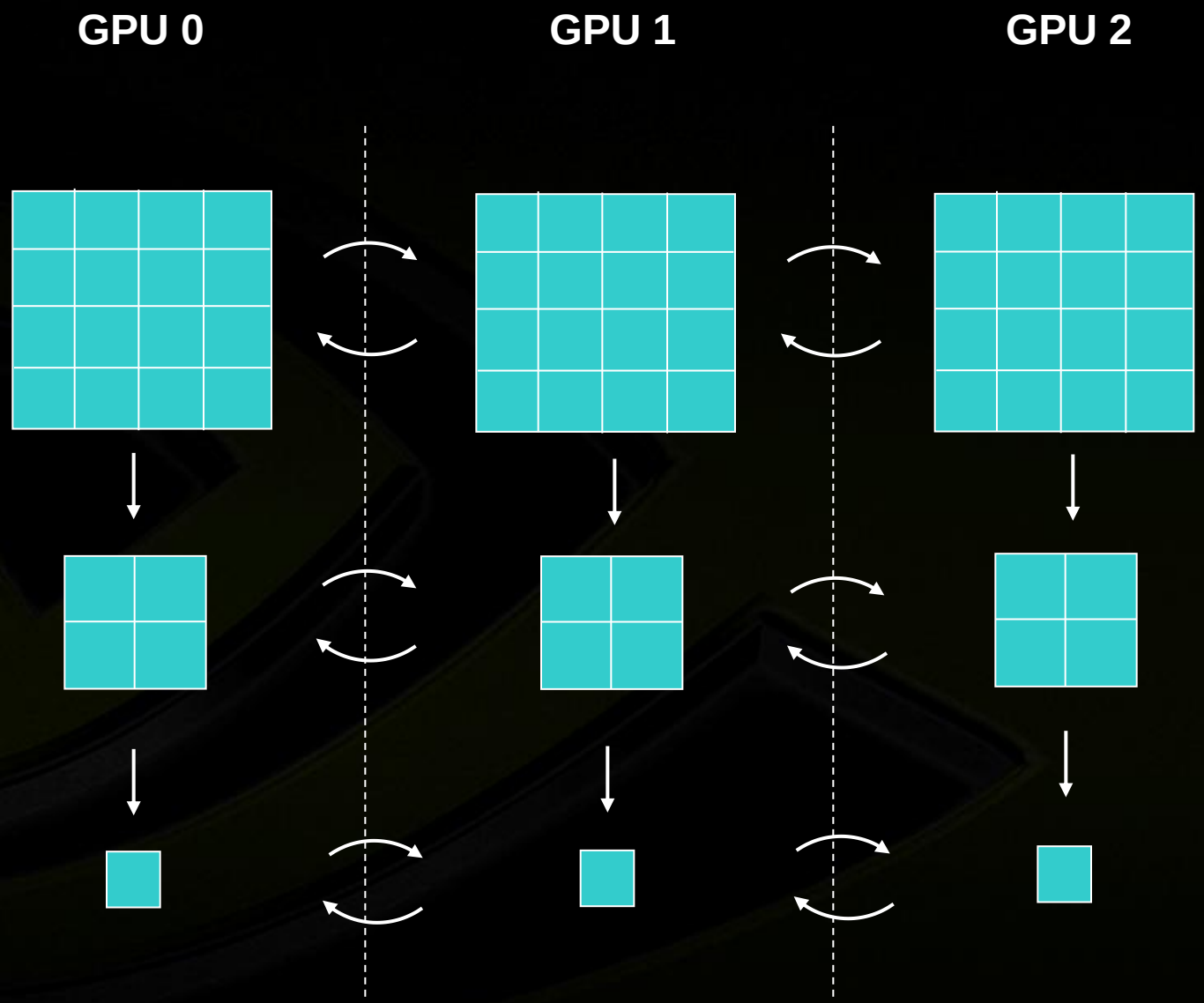
# Scalar advection – single GPU

```
class Eqn_ScalarAdvection3D : public Equation
{
  Term_PassiveAdvection3DDevice       _advection_term;
  double                              _hx, _hy, _hz;
  Grid3DDeviceD                       _u, _v, _w;
  Grid3DDeviceD                       _phi;
  Grid3DDeviceD                       _deriv_phidt;
  BoundaryConditionSet                _bc;

public:

  void advance_one_step(double dt) {
    apply_3d_boundary_conditions(_phi, _bc, _hx, _hy, _hz);

    // _deriv_phidt = - (\nabla \cdot (_u, _v, _w)) _phi
    _advection_term.evaluate());

    // forward Euler
    _phi.linear_combination(1.0, _phi, dt, _deriv_phidt);
  }
};
```

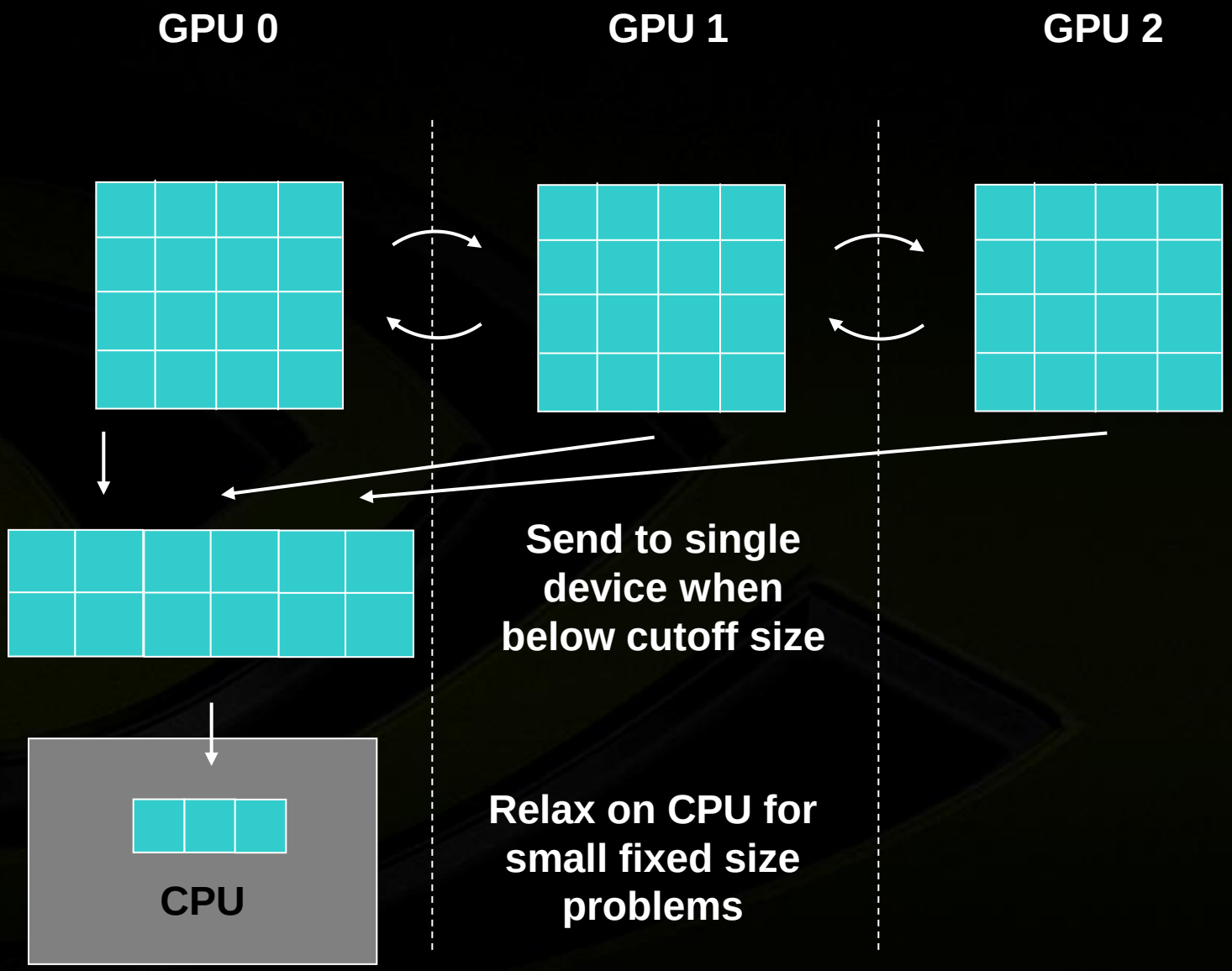
# Scalar advection – multi GPU

```
class Eqn_ScalarAdvection3DMultiGPU : public Equation
{
  Term_PassiveAdvection3DDevice       _advection;
  ... // same as before
  int                                 _easthdl, _westhdl;

public:

  void advance_one_step(double dt) {
    apply_3d_boundary_conditions(_phi, _bc, _hx, _hy, _hz);

    CoArrayManager::io_fence();
    // _deriv_phidt = - (\nabla \cdot (_u, _v, _w)) _phi
    _advection.solve());

    // forward Euler
    _phi.linear_combination(1.0, _phi, dt, _deriv_phidt);
    CoArrayManager::barrier_exchange(_easthdl);
    CoArrayManager::barrier_exchange(_westhdl);
  }
};
```

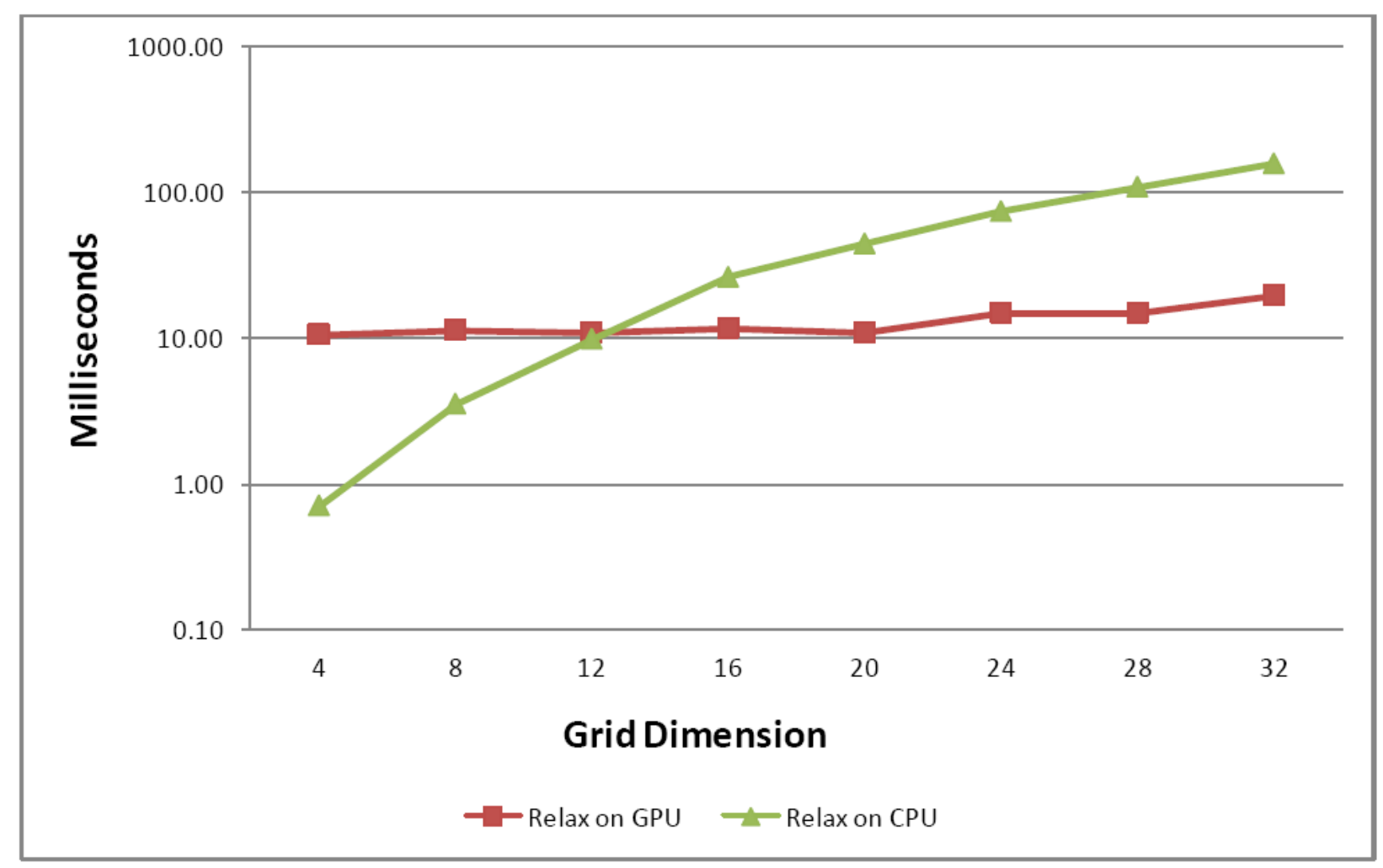
# Naïve multi-GPU multigrid



# Optimized communication pattern



# CPU relaxation vs GPU relaxation



# Choosing Optimal Cutoff

**Coarse relax on CPU**

| 128 | **64** | 32 | 16 | 8 | 4 | 0 |
|---|---|---|---|---|---|---|
| 720 ms | **624 ms** | 628 ms | 660 ms | 695 ms | 732 ms | 738 ms |

**All relax on GPU**

| 128 | 64 | 32 | 16 | 8 | 4 | 0 |
|---|---|---|---|---|---|---|
| 741 ms | 631 ms | 643 ms | 676 ms | 706 ms | 734 ms | 784 ms |



**Copy to single GPU later**

# Multi GPU Results – Incompressible Navier-Stokes

| Single GPU - $256^3$ | Dual GPU – $512 \times 256^2$ (Weak Scaling) | Dual GPU – $256^3$ (Strong Scaling) |
|---|---|---|
| 467 ms / step | 562 ms / step | 306 ms / step |
| x | 83% of linear | 76% of linear |

- Tesla C1060 (haven't run on Fermi yet)
- Not particularly optimized
  - Compute / Transfers not overlapped
- "Strong scaling" isn't apples-to-apples, since aspect ratio changes => more MG iterations

# Future Work – up to you

- OpenCurrent great basis for student projects:
  - Surface tracking
  - Complex boundary conditions for Poisson Solver
  - Experiment with different PDEs on structured grids
  - Stretched grids
  - Curvilinear grids
  - Real-time version (in fact, OpenCurrent is basis for NVIDIA's APEX Turbulence module)
  - Add turbulence model
  - Better parallel linear solvers
  - MPI support
  - Better multi-GPU scaling
  - etc.

# Thanks


**Joint work with Jeroen Molemaker, UCLA**

**Contact me: jocohen@nvdia.com**

**http://research.nvidia.com**
**http://research.nvidia.com/users/jonathan-cohen**

**http://code.google.com/p/opencurrent/**