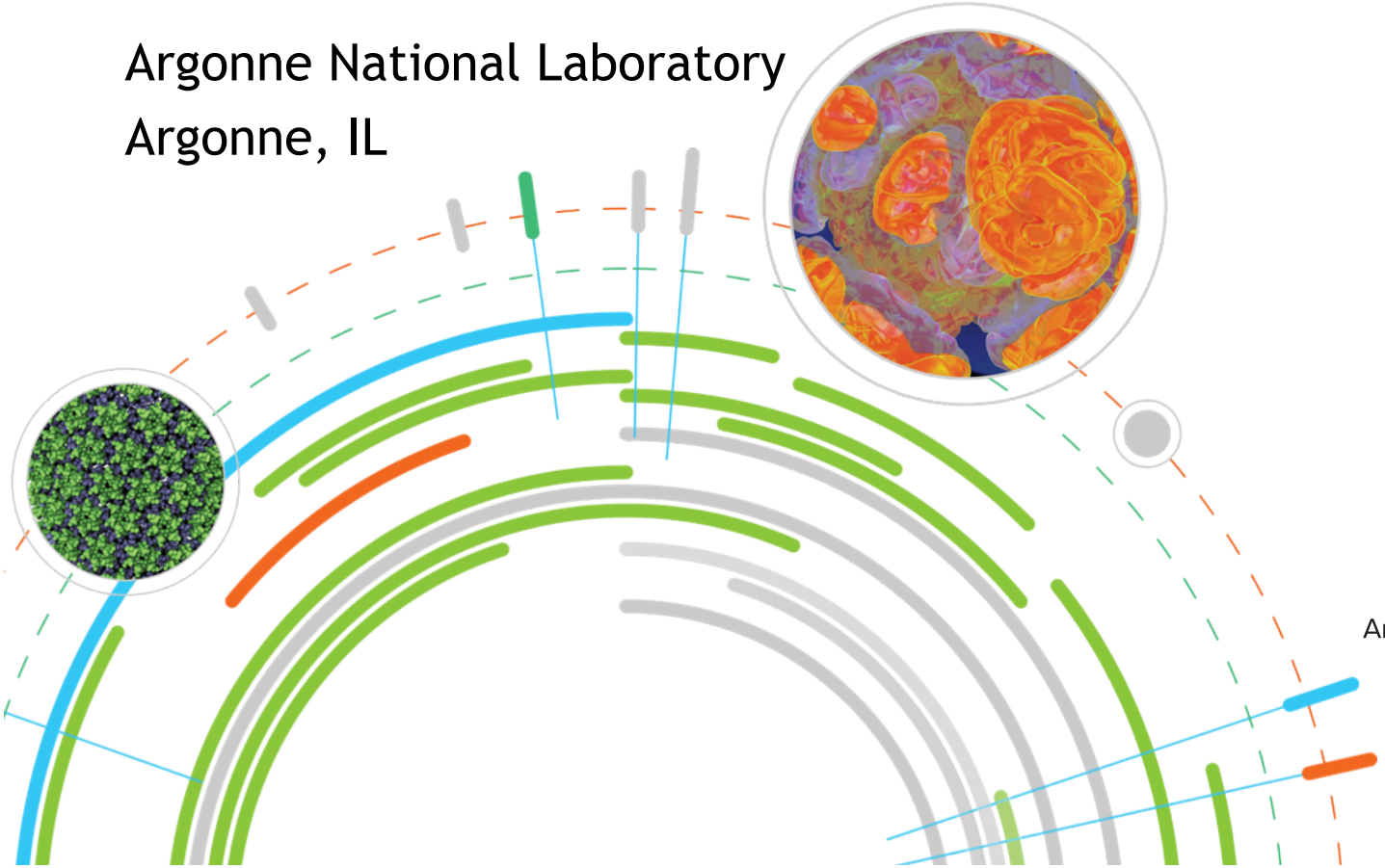
Portability of HACC – a highly tuned cosmology application

Salman Habib, Katrin Heitmann, Hal Finkel,
Adrian Pope*, Nicholas Frontiere, and **Vitali Morozov***

Argonne National Laboratory
Argonne, IL



Argonne **Leadership Computing** Facility

Argonne NATIONAL LABORATORY

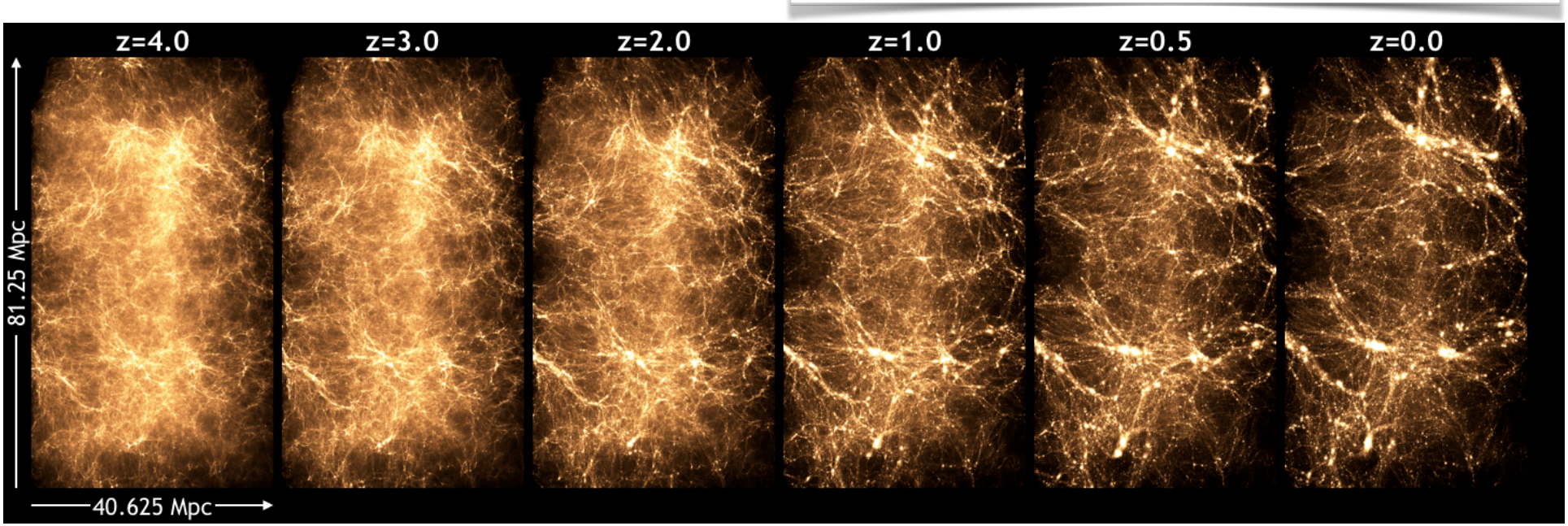
Overview

- HACC - The Hardware Accelerated Cosmology Code
  - Code overview
  - Main design principles
- Performance and Portability
  - Definitions and terminology
  - Portability – hard, soft, and non-portable solutions
  - Microkernels – key to performance achievements
  - Examples – short force evaluation kernel, tree build, cm kernel, 3D FFT
  - Performance portability – effort versus gain
- Conclusions

Large-Scale Structure in an Expanding Universe

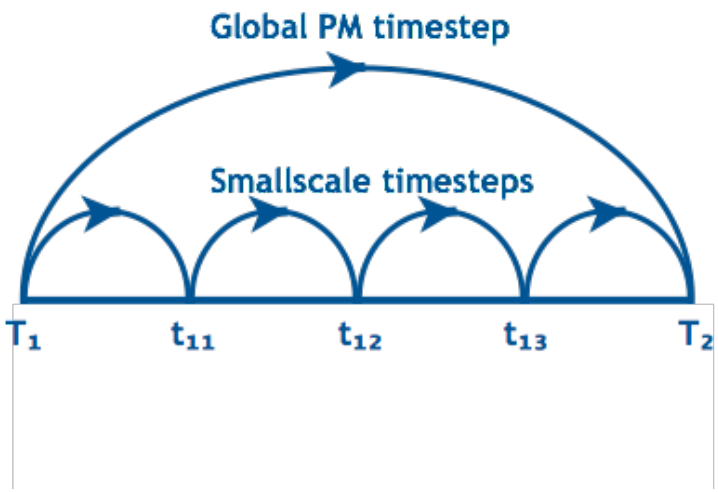
- Gravity dominates on large scales
  - Cosmological Vlasov-Poisson equation
  - Usual PDE methods fail
    - 6D phase space
    - Structure from Jeans instability
  - Use N-body methods
    - Particles hold “real” information
    - Naturally Lagrangian
    - Robust to errors/noise

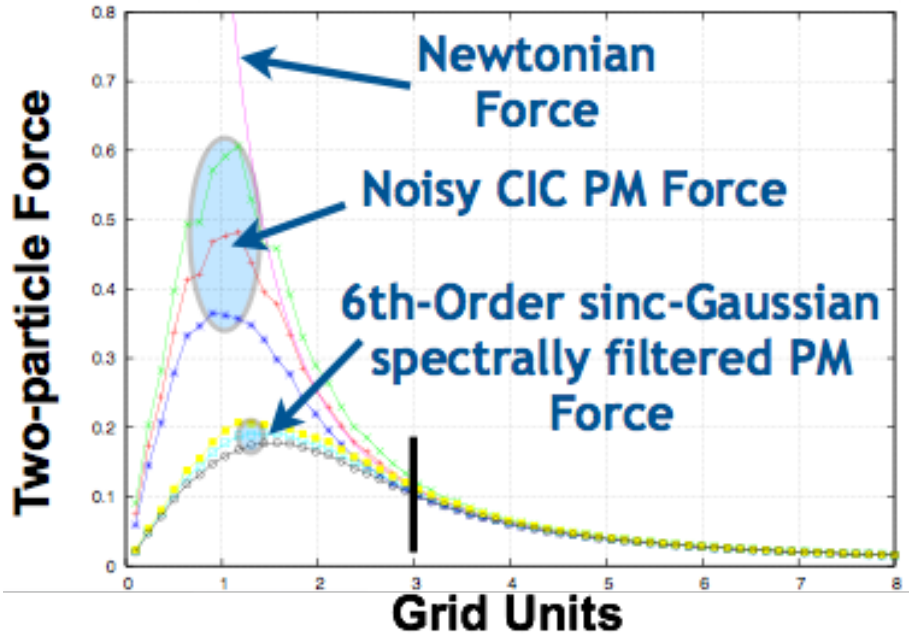
$$
\begin{aligned}
\frac{\partial f_i}{\partial t} + \dot{\mathbf{x}}\frac{\partial f_i}{\partial \mathbf{x}} - \nabla\phi\frac{\partial f_i}{\partial \mathbf{p}} &= 0, \qquad \mathbf{p} = a^2\dot{\mathbf{x}}, \\
\nabla^2\phi &= 4\pi G a^2(\rho(\mathbf{x},t) - \langle\rho_{\rm dm}(t)\rangle) = 4\pi G a^2 \Omega_{\rm dm}\delta_{\rm dm}\rho_{\rm cr}, \\
\delta_{\rm dm}(\mathbf{x},t) &= (\rho_{\rm dm} - \langle\rho_{\rm dm}\rangle)/\langle\rho_{\rm dm}\rangle), \\
\rho_{\rm dm}(\mathbf{x},t) &= a^{-3}\sum_i m_i \int d^3\mathbf{p} f_i(\mathbf{x},\dot{\mathbf{x}},t).
\end{aligned}
$$



Hardware/Hybrid Accelerated Cosmology Code

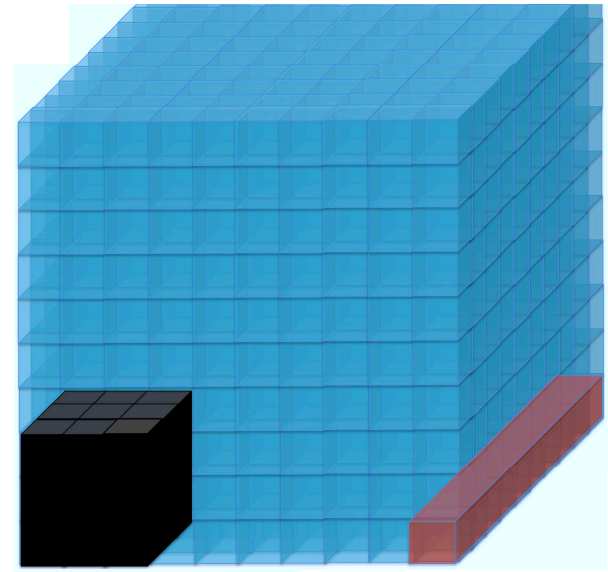
- Design considerations
  - Gravity acts on all length scales
  - Architectural diversity, distributed memory
- Long-range, slowly varying
  - Distributed memory, MPI
  - Spectral/FFT Particle-Mesh (PM) methods
- Short-range, quickly varying
  - Rank-local shared memory
  - Particle-particle comparisons
  - Computationally intense
- Carefully tuned force hand-over
- Time-stepping/integration
  - 2nd order symplectic
  - Standard operator splitting



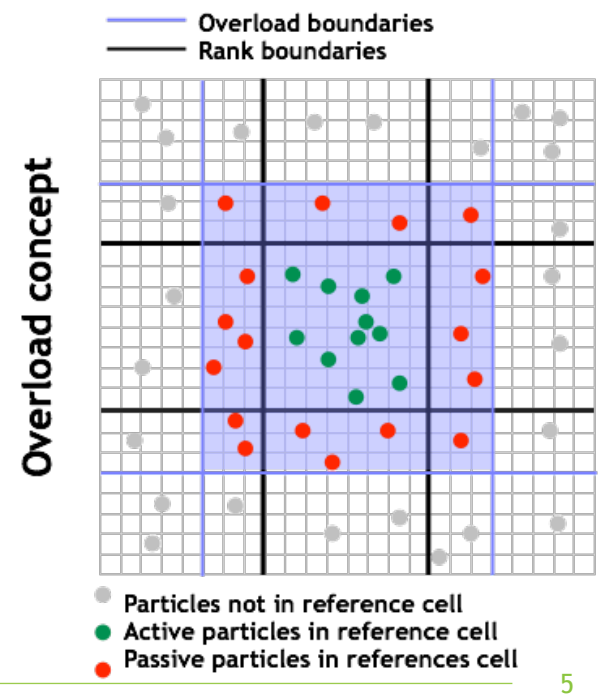


Implementation

- Long-range force
  - Particles in 3D decomposition
  - Custom FFT in 2D/pencil decomposition
    - Grid re-distribution after particle deposit
  - Tested up to ~15000^3 grid, ~10^6 ranks
- Short-range force
  - Instantiate thin particle cache from neighbors
    - Isolate short-range force from communication
  - Finite support, 5th order polynomial
- High performance
  - ACM Gordon Bell Finalist
    - SC12: 14 Pflops (~70% of peak) LLNL/Sequoia
    - SC13: OpenCL on OLCF/Titan
  - "Hero" simulations completed
    - ALCF/Mira: The Outer Rim
    - OLCF/Titan: The Q Continuum



3D Volume Data 2D Pencil Data



HACC: Main Design Principles

- Absolute Performance
  - The code is designed for it as the first-class requirement
  - Isolation of small number of compute-intensive kernels
  - Major focus on data locality
- Algorithmic Flexibility
  - Compute-intensive parts are independent on implementation
  - Compute-intensive parts should be parameterized
- Expert Tuning
  - The tuning cost should be limited by a small isolated subset of plug-in code
- Portable Top Layer
  - Portability of non-performance critical section must be maximized
- Limiting External Dependencies
  - Independence on timescale / productivity / availability of others

Portability

- Hard Portability
  - Requires no code changes and no tuning
  - Should compile and run out of the box
  - Possible? YES!
    - Fortran/C/C++ code with strict language standard, no OS calls, no assumption on I/O
  - Useful? YES!
    - Main logic of the code, data layout, type definitions, initialization stage, …, up to 95% of code base
- Soft Portability
  - Requires "simple" code modifications, no algorithmic changes
    - Simple … hmmm not always
  - Possible? YES!
    - Microkernels must be tuned for performance
    - Free to use any "non portable" techniques including assembler programming
  - Useful? YES!
    - Significant performance gain (examples will follow)
    - Good performance / portability tradeoff (examples will follow)
- Non-portable Solutions
  - Any algorithmic changes in the code
  - Any development of new kernels, including fusion / splitting of old kernels

Portability

- Hard Portability
  - Requires no code changes and no tuning
  - Should co
  - Possible? Y
    - Fortran
  - Useful? YE
    - Main lo code base
- Soft Portability
  - Requires "
    - Simple
  - Possible? Y
    - Microk
    - Free to
  - Useful? YES!
    - Significant performance gain (examples will follow)
    - Good performance / portability tradeoff (examples will follow)
- Non-portable Solutions
  - Any algorithmic changes in the code
  - Any development of new kernels, including fusion / splitting of old kernels

HACC design choices:

Large fraction of the code should be hard portable

Key compute intensive algorithms are localized

Microkernels are soft portable across similar platforms

Microkernels can be non-portable (performance prevails!)

Microkernels – key to performance achievements

- What is a microkernel?
  - One or many routines that perform a complete data processing
    - Algorithmic changes require complete microkernel change
    - Microkernel is specific to the code, not a general purpose math routine
    - BLAS kernels are not qualified as microkernels
  - Uses and preserves a given data layout determined in the main code
  - Uses realistic data sizes and data locality
    - Even better, parameterized data sizes and memory footprint
  - Takes a noticeable fraction of the run time
    - Desired, but not required
- Concurrency
  - Tricky – not too much, not too little
- Art of designing the microkernels
  - Discussed by many in different forums

Example: Short Force Evaluation

```
void ShortForce( int count1, float xxi, float yyi, float zzi, float fsrrmax2, float mp_rsm2,
    float *xx1, float *yy1, float *zz1, float *mass1, float *dxi, float *dyi, float *dzi ) {

    const float ma0 = 0.269327, ma1 = -0.0750978, ma2 = 0.0114808, ma3 = -0.00109313;
    const float ma4 = 0.0000605491, ma5 = -0.00000147177;

    float dxc, dyc, dzc, m, r2, f, xi, yi, zi;    int j;

    xi = 0.f; yi = 0.f; zi = 0.f;

    for ( j = 0; j < count1; j++ ) {
        dxc = xx1[j] - xxi;
        dyc = yy1[j] - yyi;
        dzc = zz1[j] - zzi;

        r2 = dxc * dxc + dyc * dyc + dzc * dzc;

        if ( ( r2 > 0.0f ) && ( r2 < fsrrmax2 ) ) {
            f = r2 + mp_rsm2;
            f =  mass1[j] * ( 1.f / ( f * sqrtf( f ) ) –
                 ( ma0 + r2*(ma1 + r2*(ma2 + r2*(ma3 + r2*(ma4 + r2*ma5))))) );

            xi = xi + f * dxc;
            yi = yi + f * dyc;
            zi = zi + f * dzc;
        }
    }

    *dxi = xi;
    *dyi = yi;
    *dzi = zi;
}
```

90% of run time
C language: 32 lines

Example: Short Force Evaluation

```
void ShortForce( int count1, float xxi, float yyi, float zzi, float fsrrmax2, float mp_rsm2,
    float *xx1, float *yy1, float *zz1, float *mass1, float *dxi, float *dyi, float *dzi ) {

    const float ma0 = 0.269327, ma1 = -0.0750978, ma2 = 0.0114808, ma3 = -0.00109313;
    const float ma4 = 0.0000605491, ma5 = -0.00000147177;

    float dxc, dyc, dzc, m, r2, f, xi, yi, zi;    int j;

    xi = 0.f; yi = 0.f; zi = 0.f;

    for ( j = 0; j < count1; j++ ) {
        dxc = xx1[j] - xxi;
        dyc = yy1[j] - yyi;
        dzc = zz1[j] - zzi;

        r2 = dxc * dxc + dyc * dyc + dzc * dzc;

        if ( ( r2 > 0.0f ) && ( r2 < fsrrmax2 ) ) {
            f = r2 + mp_rsm2;
            f =  mass1[j] * ( 1.f / ( f * sqrtf( f ) ) -
                 ( ma0 + r2*(ma1 + r2*(ma2 + r2*(ma3 + r2*(ma4 + r2*ma5))))) );

            xi = xi + f * dxc;
            yi = yi + f * dyc;
            zi = zi + f * dzc;
        }
    }

    *dxi = xi;
    *dyi = yi;
    *dzi = zi;
}
```

90% of run time
C language: 32 lines
Dependency chain

Example: Short Force Evaluation

```
void ShortForce( int count1, float xxi, float yyi, float zzi, float fsrrmax2, float mp_rsm2,
    float *xx1, float *yy1, float *zz1, float *mass1, float *dxi, float *dyi, float *dzi ) {

    const float ma0 = 0.269327, ma1 = -0.0750978, ma2 = 0.0114808, ma3 = -0.00109313;
    const float ma4 = 0.0000605491, ma5 = -0.00000147177;

    float dxc, dyc, dzc, m, r2, f, xi, yi, zi;    int j;

    xi = 0.f; yi = 0.f; zi = 0.f;

    for ( j = 0; j < count1; j++ ) {
        dxc = xx1[j] - xxi;
        dyc = yy1[j] - yyi;
        dzc = zz1[j] - zzi;

        r2 = dxc * dxc + dyc * dyc + dzc * dzc;

        if ( ( r2 > 0.0f ) && ( r2 < fsrrmax2 ) ) {
            f = r2 + mp_rsm2;
            f =  mass1[j] * ( 1.f / ( f * sqrtf( f ) ) –
                 ( ma0 + r2*(ma1 + r2*(ma2 + r2*(ma3 + r2*(ma4 + r2*ma5))))) );

            xi = xi + f * dxc;
            yi = yi + f * dyc;
            zi = zi + f * dzc;
        }
    }

    *dxi = xi;
    *dyi = yi;
    *dzi = zi;
}
```

90% of run time
C language: 32 lines
Dependency chain
Single precision

Example: Short Force Evaluation

```
void ShortForce( int count1, float xxi, float yyi, float zzi, float fsrrmax2, float mp_rsm2,
    float *xx1, float *yy1, float *zz1, float *mass1, float *dxi, float *dyi, float *dzi ) {

    const float ma0 = 0.269327, ma1 = -0.0750978, ma2 = 0.0114808, ma3 = -0.00109313;
    const float ma4 = 0.0000605491, ma5 = -0.00000147177;

    float dxc, dyc, dzc, m, r2, f, xi, yi, zi;    int j;

    xi = 0.f; yi = 0.f; zi = 0.f;

    for ( j = 0; j < count1; j++ ) {
        dxc = xx1[j] - xxi;
        dyc = yy1[j] - yyi;
        dzc = zz1[j] - zzi;

        r2 = dxc * dxc + dyc * dyc + dzc * dzc;

        if ( ( r2 > 0.0f ) && ( r2 < fsrrmax2 ) ) {
            f = r2 + mp_rsm2;
            f =  mass1[j] * ( 1.f / ( f * sqrtf( f ) ) –
                 ( ma0 + r2*(ma1 + r2*(ma2 + r2*(ma3 + r2*(ma4 + r2*ma5))))) );

            xi = xi + f * dxc;
            yi = yi + f * dyc;
            zi = zi + f * dzc;
        }
    }

    *dxi = xi;
    *dyi = yi;
    *dzi = zi;
}
```

90% of run time
C language: 32 lines
Dependency chain
Single precision
Polynomial

Example: Short Force Evaluation

```
void ShortForce( int count1, float xxi, float yyi, float zzi, float fsrrmax2, float mp_rsm2,
    float *xx1, float *yy1, float *zz1, float *mass1, float *dxi, float *dyi, float *dzi ) {

    const float ma0 = 0.269327, ma1 = -0.0750978, ma2 = 0.0114808, ma3 = -0.00109313;
    const float ma4 = 0.0000605491, ma5 = -0.00000147177;

    float dxc, dyc, dzc, m, r2, f, xi, yi, zi;    int j;

    xi = 0.f; yi = 0.f; zi = 0.f;

    for ( j = 0; j < count1; j++ ) {
        dxc = xx1[j] - xxi;
        dyc = yy1[j] - yyi;
        dzc = zz1[j] - zzi;

        r2 = dxc * dxc + dyc * dyc + dzc * dzc;

        if ( ( r2 > 0.0f ) && ( r2 < fsrrmax2 ) ) {
            f = r2 + mp_rsm2;
            f =  mass1[j] * ( 1.f / ( f * sqrtf( f ) ) –
                 ( ma0 + r2*(ma1 + r2*(ma2 + r2*(ma3 + r2*(ma4 + r2*ma5))))) );

            xi = xi + f * dxc;
            yi = yi + f * dyc;
            zi = zi + f * dzc;
        }
    }

    *dxi = xi;
    *dyi = yi;
    *dzi = zi;
}
```

90% of run time
C language: 32 lines
Dependency chain
Single precision
Polynomial
Data reuse

Example: Short Force Evaluation

```
void ShortForce( int count1, float xxi, float yyi, float zzi, float fsrrmax2, float mp_rsm2,
    float *xx1, float *yy1, float *zz1, float *mass1, float *dxi, float *dyi, float *dzi ) {

    const float ma0 = 0.269327, ma1 = -0.0750978, ma2 = 0.0114808, ma3 = -0.00109313;
    const float ma4 = 0.0000605491, ma5 = -0.00000147177;

    float dxc, dyc, dzc, m, r2, f, xi, yi, zi;    int j;

    xi = 0.f; yi = 0.f; zi = 0.f;

    for ( j = 0; j < count1; j++ ) {
        dxc = xx1[j] - xxi;
        dyc = yy1[j] - yyi;
        dzc = zz1[j] - zzi;

        r2 = dxc * dxc + dyc * dyc + dzc * dzc;

        if ( ( r2 > 0.0f ) && ( r2 < fsrrmax2 ) ) {
            f = r2 + mp_rsm2;
            f =  mass1[j] * ( 1.f / ( f * sqrtf( f ) ) –
                 ( ma0 + r2*(ma1 + r2*(ma2 + r2*(ma3 + r2*(ma4 + r2*ma5))))) );

            xi = xi + f * dxc;
            yi = yi + f * dyc;
            zi = zi + f * dzc;
        }
    }

    *dxi = xi;
    *dyi = yi;
    *dzi = zi;
}
```

90% of run time
C language: 32 lines
Dependency chain
Single precision
Polynomial
Data reuse
Division/Square root

Example: Short Force Evaluation

```
void ShortForce( int count1, float xxi, float yyi, float zzi, float fsrrmax2, float mp_rsm2,
    float *xx1, float *yy1, float *zz1, float *mass1, float *dxi, float *dyi, float *dzi ) {

    const float ma0 = 0.269327, ma1 = -0.0750978, ma2 = 0.0114808, ma3 = -0.00109313;
    const float ma4 = 0.0000605491, ma5 = -0.00000147177;

    float dxc, dyc, dzc, m, r2, f, xi, yi, zi;    int j;

    xi = 0.f; yi = 0.f; zi = 0.f;

    for ( j = 0; j < count1; j++ ) {
        dxc = xx1[j] - xxi;
        dyc = yy1[j] - yyi;
        dzc = zz1[j] - zzi;

        r2 = dxc * dxc + dyc * dyc + dzc * dzc;

        if ( ( r2 > 0.0f ) && ( r2 < fsrrmax2 ) ) {
            f = r2 + mp_rsm2;
            f =  mass1[j] * ( 1.f / ( f * sqrtf( f ) ) –
                 ( ma0 + r2*(ma1 + r2*(ma2 + r2*(ma3 + r2*(ma4 + r2*ma5))))) );

            xi = xi + f * dxc;
            yi = yi + f * dyc;
            zi = zi + f * dzc;
        }
    }

    *dxi = xi;
    *dyi = yi;
    *dzi = zi;
}
```

90% of run time
C language: 32 lines
Dependency chain
Single precision
Polynomial
Data reuse
Division/Square root
Complex conditional

Example: Short Force Evaluation

```
void ShortForce( int count1, float xxi, float yyi, float zzi, float fsrrmax2, float mp_rsm2,
    float *xx1, float *yy1, float *zz1, float *mass1, float *dxi, float *dyi, float *dzi ) {

    const float ma0 = 0.269327, ma1 = -0.0750978, ma2 = 0.0114808, ma3 = -0.00109313;
    const float ma4 = 0.0000605491, ma5 = -0.00000147177;

    float dxc, dyc, dzc, m, r2, f, xi, yi, zi;    int j;

    xi = 0.f; yi = 0.f; zi = 0.f;

    for ( j = 0; j < count1; j++ ) {
        dxc = xx1[j] - xxi;
        dyc = yy1[j] - yyi;
        dzc = zz1[j] - zzi;

        r2 = dxc * dxc + dyc * dyc + dzc * dzc;

        if ( ( r2 > 0.0f ) && ( r2 < fsrrmax2 ) ) {
            f = r2 + mp_rsm2;
            f =  mass1[j] * ( 1.f / ( f * sqrtf( f ) ) –
                 ( ma0 + r2*(ma1 + r2*(ma2 + r2*(ma3 + r2*(ma4 + r2*ma5))))) );

            xi = xi + f * dxc;
            yi = yi + f * dyc;
            zi = zi + f * dzc;
        }
    }

    *dxi = xi;
    *dyi = yi;
    *dzi = zi;
}
```

Portability?

90% of run time
C language: 32 lines
Dependency chain
Single precision
Polynomial
Data reuse
Division/Square root
Complex conditional

Example: Short Force Evaluation tuning

```
void ShortForce( int count1, float xxi, float yyi, float zzi, float fsrrmax2, float mp_rsm2,
    float *xx1, float *yy1, float *zz1, float *mass1, float *dxi, float *dyi, float *dzi ) {

    const f
    const f

    float d

    xi = 0.

    for ( j
        dxc
        dyc
        dzc

        r2

        if

        }
    }

    *dxi =
    *dyi = yi;
    *dzi = zi;
}
```

BG/Q Hard portable solution:

37,128,240 cycles, 7.5% of peak

BG/Q Soft portable solution:

4,180,445 cycles, 68% of peak

Intrinsics, 6 month effort, x8.89 speedup

KNL Hard portable solution:

Baseline and compiler optimized code

Compiler tuning possible AFTER assembler

KNL Soft portable:

Significant performance improvement

Assembler, 3 month effort, measurable speedup

Example: CM Bounding Box Computation

```
void cm( int count, float *xx, float *yy, float *zz, float *mass, float *xmin, float *xmax, float *xc){
  double x = 0, y = 0, z = 0, m = 0;
  for (int i = 0; i < count; ++i) {
    if (i == 0) {
        xmin[0] = xmax[0] = xx[0];
        xmin[1] = xmax[1] = yy[0];
        xmin[2] = xmax[2] = zz[0];
    }
    else {
        xmin[0] = fminf(xmin[0], xx[i]);
        xmax[0] = fmaxf(xmax[0], xx[i]);
        xmin[1] = fminf(xmin[1], yy[i]);
        xmax[1] = fmaxf(xmax[1], yy[i]);
        xmin[2] = fminf(xmin[2], zz[i]);
        xmax[2] = fmaxf(xmax[2], zz[i]);
    }

    float w = mass[i];
    x += w*xx[i];
    y += w*yy[i];
    z += w*zz[i];
    m += w;
  }

  xc[0] = (float) (x/m);
  xc[1] = (float) (y/m);
  xc[2] = (float) (z/m);
}
```

5% of run time
C language: 28 lines

Example: CM Bounding Box Computation

```
void cm( int count, float *xx, float *yy, float *zz, float *mass, float *xmin, float *xmax, float *xc){
  double x = 0, y = 0, z = 0, m = 0;
  for (int i = 0; i < count; ++i) {
    if (i == 0) {
        xmin[0] = xmax[0] = xx[0];
        xmin[1] = xmax[1] = yy[0];
        xmin[2] = xmax[2] = zz[0];
    }
    else {
        xmin[0] = fminf(xmin[0], xx[i]);
        xmax[0] = fmaxf(xmax[0], xx[i]);
        xmin[1] = fminf(xmin[1], yy[i]);
        xmax[1] = fmaxf(xmax[1], yy[i]);
        xmin[2] = fminf(xmin[2], zz[i]);
        xmax[2] = fmaxf(xmax[2], zz[i]);
    }

    float w = mass[i];
    x += w*xx[i];
    y += w*yy[i];
    z += w*zz[i];
    m += w;
  }

  xc[0] = (float) (x/m);
  xc[1] = (float) (y/m);
  xc[2] = (float) (z/m);
}
```

5% of run time
C language: 28 lines
Conditional in loop

Example: CM Bounding Box Computation

```
void cm( int count, float *xx, float *yy, float *zz, float *mass, float *xmin, float *xmax, float *xc){
  double x = 0, y = 0, z = 0, m = 0;
  for (int i = 0; i < count; ++i) {
    if (i == 0) {
        xmin[0] = xmax[0] = xx[0];
        xmin[1] = xmax[1] = yy[0];
        xmin[2] = xmax[2] = zz[0];
    }
    else {
        xmin[0] = fminf(xmin[0], xx[i]);
        xmax[0] = fmaxf(xmax[0], xx[i]);
        xmin[1] = fminf(xmin[1], yy[i]);
        xmax[1] = fmaxf(xmax[1], yy[i]);
        xmin[2] = fminf(xmin[2], zz[i]);
        xmax[2] = fmaxf(xmax[2], zz[i]);
    }

    float w = mass[i];
    x += w*xx[i];
    y += w*yy[i];
    z += w*zz[i];
    m += w;
  }

  xc[0] = (float) (x/m);
  xc[1] = (float) (y/m);
  xc[2] = (float) (z/m);
}
```

5% of run time
C language: 28 lines
Conditional in loop
Function calls
Accumulation

Example: CM Bounding Box Computation

```
void cm( int count, float *xx, float *yy, float *zz, float *mass, float *xmin, float *xmax, float *xc){
  double x = 0, y = 0, z = 0, m = 0;
  for (int i = 0; i < count; ++i) {
    if (i == 0) {
        xmin[0] = xmax[0] = xx[0];
        xmin[1] = xmax[1] = yy[0];
        xmin[2] = xmax[2] = zz[0];
    }
    else {
        xmin[0] = fminf(xmin[0], xx[i]);
        xmax[0] = fmaxf(xmax[0], xx[i]);
        xmin[1] = fminf(xmin[1], yy[i]);
        xmax[1] = fmaxf(xmax[1], yy[i]);
        xmin[2] = fminf(xmin[2], zz[i]);
        xmax[2] = fmaxf(xmax[2], zz[i]);
    }

    float w = mass[i];
    x += w*xx[i];
    y += w*yy[i];
    z += w*zz[i];
    m += w;
  }

  xc[0] = (float) (x/m);
  xc[1] = (float) (y/m);
  xc[2] = (float) (z/m);
}
```

5% of run time
C language: 28 lines
Conditional in loop
Function calls
Accumulation

Example: CM Bounding Box Computation

```
void cm( int count, float *xx, float *yy, float *zz, float *mass, float *xmin, float *xmax, float *xc){
  double x = 0, y = 0, z = 0, m = 0;
  for (int i = 0; i < count; ++i) {
    if (i == 0) {
        xmin[0] = xmax[0] = xx[0];
        xmin[1] = xmax[1] = yy[0];
        xmin[2] = xmax[2] = zz[0];
    }
    else {
        xmin[0] = fminf(xmin[0], xx[i]);
        xmax[0] = fmaxf(xmax[0], xx[i]);
        xmin[1] = fminf(xmin[1], yy[i]);
        xmax[1] = fmaxf(xmax[1], yy[i]);
        xmin[2] = fminf(xmin[2], zz[i]);
        xmax[2] = fmaxf(xmax[2], zz[i]);
    }

    float w = mass[i];
    x += w*xx[i];
    y += w*yy[i];
    z += w*zz[i];
    m += w;
  }

  xc[0] = (float) (x/m);
  xc[1] = (float) (y/m);
  xc[2] = (float) (z/m);
}
```

Portability?

5% of run time
C language: 28 lines
Conditional in loop
Function calls
Accumulation

Example: CM Bounding Box Computation

```
void cm( int count, float *xx, float *yy, float *zz, float *mass, float *xmin, float *xmax, float *xc) {
  float x = 0.f, y = 0.f, z = 0.f, m = 0.f, w, xmin0, xmin1, xmin2, xmax0, xmax1, xmax2; int i;

  xmin0 = xmin1 = xmin2 = FLT_MAX;
  xmax0 = xmax1 = xmax2 = FLT_MIN;

#pragma vector aligned
  for ( i = 0; i < count; ++i )
  {
    if ( xmin0 > xx[i] ) xmin0 = xx[i];
    if ( xmin1 > yy[i] ) xmin1 = yy[i];
    if ( xmin2 > zz[i] ) xmin2 = zz[i];

    if ( xmax0 < xx[i] ) xmax0 = xx[i];
    if ( xmax1 < yy[i] ) xmax1 = yy[i];
    if ( xmax2 < zz[i] ) xmax2 = zz[i];

    w = mass[i];

    x += w * xx[i];
    y += w * yy[i];
    z += w * zz[i];
    m += w;
  }

  xc[0] = x / m;    xc[1] = y / m;    xc[2] = z / m;

  xmin[0] = xmin0;   xmin[1] = xmin1;   xmin[2] = xmin2;

  xmax[0] = xmax0;   xmax[1] = xmax1;   xmax[2] = xmax2;
}
```

HACC approach:

The choice of

hard portable solution

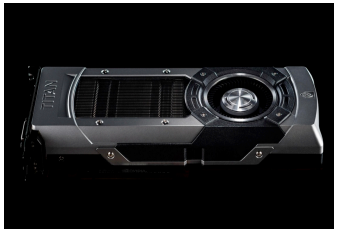
Example: CM Bounding Box Computation

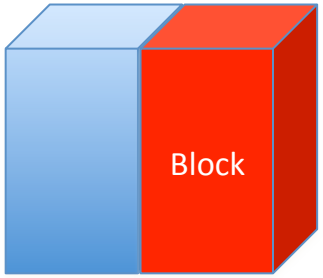
```
xi0 = vec_splats( (double)x1 );  xi1 = vec_splats( (double)x2 );
yi0 = vec_splats( (double)y1 );  yi1 = vec_splats( (double)y2 );
zi0 = vec_splats( (double)z1 );  zi1 = vec_splats( (double)z2 );

xs = vec_splats( 0. );
ys = vec_splats( 0. );
zs = vec_splats( 0. );
ms = vec_splats( 0. );

for ( i = k, j = k * 4; i < count-3;
            i = i + 4, j = j + 16 )
{
  xv = vec_lda( j, xx );  yv = vec_lda( j, yy );
  zv = vec_lda( j, zz );
  wv = vec_lda( j, mass );

  dv0 = vec_cmpgt( xi0, xv );
  dv1 = vec_cmplt( xi1, xv );
  dv2 = vec_cmpgt( yi0, yv );
  dv3 = vec_cmplt( yi1, yv );
  dv4 = vec_cmpgt( zi0, zv );
  dv5 = vec_cmplt( zi1, zv );

  xi0 = vec_sel( xi0, xv, dv0 );
  xi1 = vec_sel( xi1, xv, dv1 );
  yi0 = vec_sel( yi0, yv, dv2 ); yi1 = vec_sel( yi1, yv, dv3 );
  zi0 = vec_sel( zi0, zv, dv4 ); zi1 = vec_sel( zi1, zv, dv5 );

  xs = vec_madd( wv, xv, xs );   ys = vec_madd( wv, yv, ys );
  zs = vec_madd( wv, zv, zs );   ms = vec_add( ms, wv );
}
```

HACC approach:

The choice of
hard portable solution
or
non-portable solution

HACC on Titan: GPU Implementation (Schematic)





TreePM code: Portable, similar to CPU code

- Kernels are written in CUDA or OpenCL
  - Non-portable
- Simplicity
- Maintenance
- Flexibility

P3M Implementation: Non-portable

- Spatial data pushed to device in large blocks
- Data decomposed in one dimension
- Data sub-partitioned into chaining mesh cubes
- Forces between particles in a cube and neighboring cubes
- Natural parallelism and simplicity
- Large block sizes ensure computation time exceeds memory transfer latency by a large factor

Conclusion

- HACC essential requirements to code development
  - Absolute performance – critical first-class citizen
  - Absolute throughput – both hero and parametric study runs
- HACC approach
  - Running across a variety of platforms
    - (BUT, not all possible platforms)
  - Developing across a few small compute-intensive kernels
  - Supporting for algorithmic flexibility
  - Encouraging for expert tuning to the lowest possible level
    - (for maximal performance)
  - Pushing maximal portability for non-critical parts
  - Limiting non-essential external dependencies