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
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

\[
\frac{\partial f_i}{\partial t} + \dot{x} \frac{\partial f_i}{\partial x} - \nabla \phi \frac{\partial f_i}{\partial p} = 0, \quad p = a^2 \dot{x},
\]

\[
\nabla^2 \phi = 4\pi Ga^2 (\rho(x, t) - \langle \rho_{\text{dm}}(t) \rangle) = 4\pi Ga^2 \Omega_{\text{dm}} \delta_{\text{dm}} \rho_{\text{cr}},
\]

\[
\delta_{\text{dm}}(x, t) = \frac{(\rho_{\text{dm}} - \langle \rho_{\text{dm}} \rangle)}{\langle \rho_{\text{dm}} \rangle},
\]

\[
\rho_{\text{dm}}(x, t) = a^{-3} \sum_i m_i \int d^3 p f_i(x, \dot{x}, t).
\]
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

![Diagram of data allocations for the force calculation. A three-dimensional volume data is shown alongside two-dimensional pencil data. The figure illustrates the distribution of points with different colors representing various categories: particles not in reference cell, active particles in reference cell, and passive particles in reference cell. The diagram also highlights overload boundaries and rank boundaries.](image-url)
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 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
  • Microkernels must be tuned for performance
  • Free to use any “non-portable” techniques including assembler programming
  • Possible? YES!
  • Microkernels are soft portable across similar platforms
  • Microkernels can be non-portable (performance prevails!)
  • 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

```c
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

```c
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;
}
Example: Short Force Evaluation

```c
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

```c
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

```c
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

```c
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

```c
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

```c
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;
}
```

**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

```c
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];
        } 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

```c
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];
        } 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

```c
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];
        } 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

```c
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];
        }
        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

```c
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];
        } 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

```c
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

\[ \text{xi0} = \text{vec\_spls}( (\text{double})x1 ) \]; \text{xi1} = \text{vec\_spls}( (\text{double})x2 )
\[ \text{yi0} = \text{vec\_spls}( (\text{double})y1 ) \]; \text{yi1} = \text{vec\_spls}( (\text{double})y2 )
\[ \text{zi0} = \text{vec\_spls}( (\text{double})z1 ) \]; \text{zi1} = \text{vec\_spls}( (\text{double})z2 )
\[ \text{xs} = \text{vec\_spls}( 0. ) \];
\[ \text{ys} = \text{vec\_spls}( 0. ) \];
\[ \text{zs} = \text{vec\_spls}( 0. ) \];
\[ \text{ms} = \text{vec\_spls}( 0. ) \];
\[ \text{for} ( \text{i} = k, \text{j} = k * 4; \text{i} < \text{count}-3; \text{i} = \text{i} + 4, \text{j} = \text{j} + 16 ) \}
\[ \text{xv} = \text{vec\_lda}( \text{j}, \text{xx} ) \]; \text{yv} = \text{vec\_lda}( \text{j}, \text{yy} )
\[ \text{zv} = \text{vec\_lda}( \text{j}, \text{zz} ) \];
\[ \text{wv} = \text{vec\_lda}( \text{j}, \text{mass} ) \];
\[ \text{dv0} = \text{vec\_cmpgt}( \text{xi0}, \text{xv} ) \];
\[ \text{dv1} = \text{vec\_cmplt}( \text{xi1}, \text{xv} ) \];
\[ \text{dv2} = \text{vec\_cmpgt}( \text{yi0}, \text{yv} ) \];
\[ \text{dv3} = \text{vec\_cmplt}( \text{yi1}, \text{yv} ) \];
\[ \text{dv4} = \text{vec\_cmpgt}( \text{zi0}, \text{zv} ) \];
\[ \text{dv5} = \text{vec\_cmplt}( \text{zi1}, \text{zv} ) \];
\[ \text{xi0} = \text{vec\_sel}( \text{xi0}, \text{xv}, \text{dv0} ) \];
\[ \text{xi1} = \text{vec\_sel}( \text{xi1}, \text{xv}, \text{dv1} ) \];
\[ \text{yi0} = \text{vec\_sel}( \text{yi0}, \text{yv}, \text{dv2} ) \]; \text{yi1} = \text{vec\_sel}( \text{yi1}, \text{yv}, \text{dv3} )
\[ \text{zi0} = \text{vec\_sel}( \text{zi0}, \text{zv}, \text{dv4} ) \]; \text{zi1} = \text{vec\_sel}( \text{zi1}, \text{zv}, \text{dv5} )
\[ \text{xs} = \text{vec\_madd}( \text{wv}, \text{xv}, \text{xs} ) \]; \text{ys} = \text{vec\_madd}( \text{wv}, \text{yv}, \text{ys} )
\[ \text{zs} = \text{vec\_madd}( \text{wv}, \text{zv}, \text{zs} ) \]; \text{ms} = \text{vec\_add}( \text{ms}, \text{wv} ) \];

**HACC approach:**
The choice of hard portable solution or non-portable solution
**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