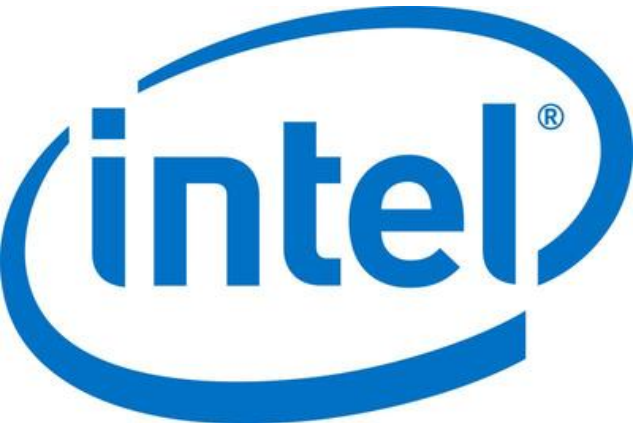
Accelerating Particle-Particle Particle-Mesh Methods for Molecular Dynamics

William McDoniel Ahmed E. Ismail Paolo Bientinesi

High Performance and Automatic Computing Group
Aachen Institute for Advanced Study in Computational Engineering Science
RWTH Aachen University



Intel PCC 2016 Fall Forum
Toulouse

Thanks to: Klaus-Dieter Oertel and Georg Zitzlsberger

Molecular Dynamics

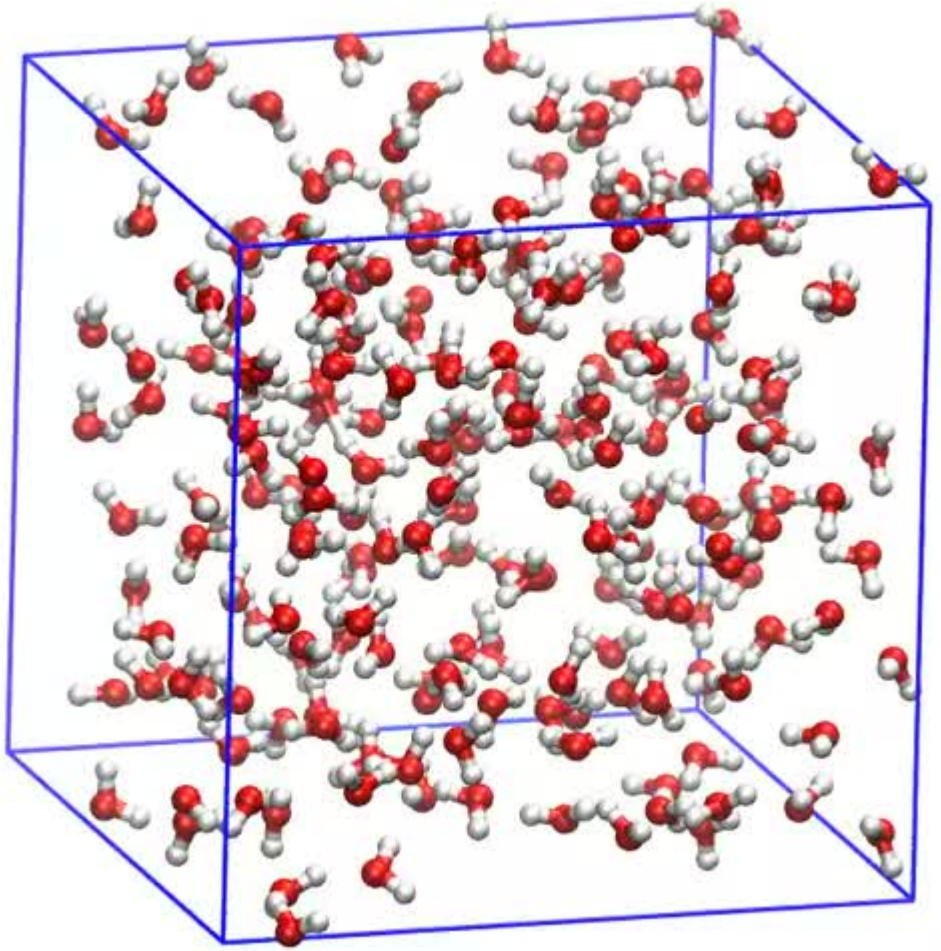
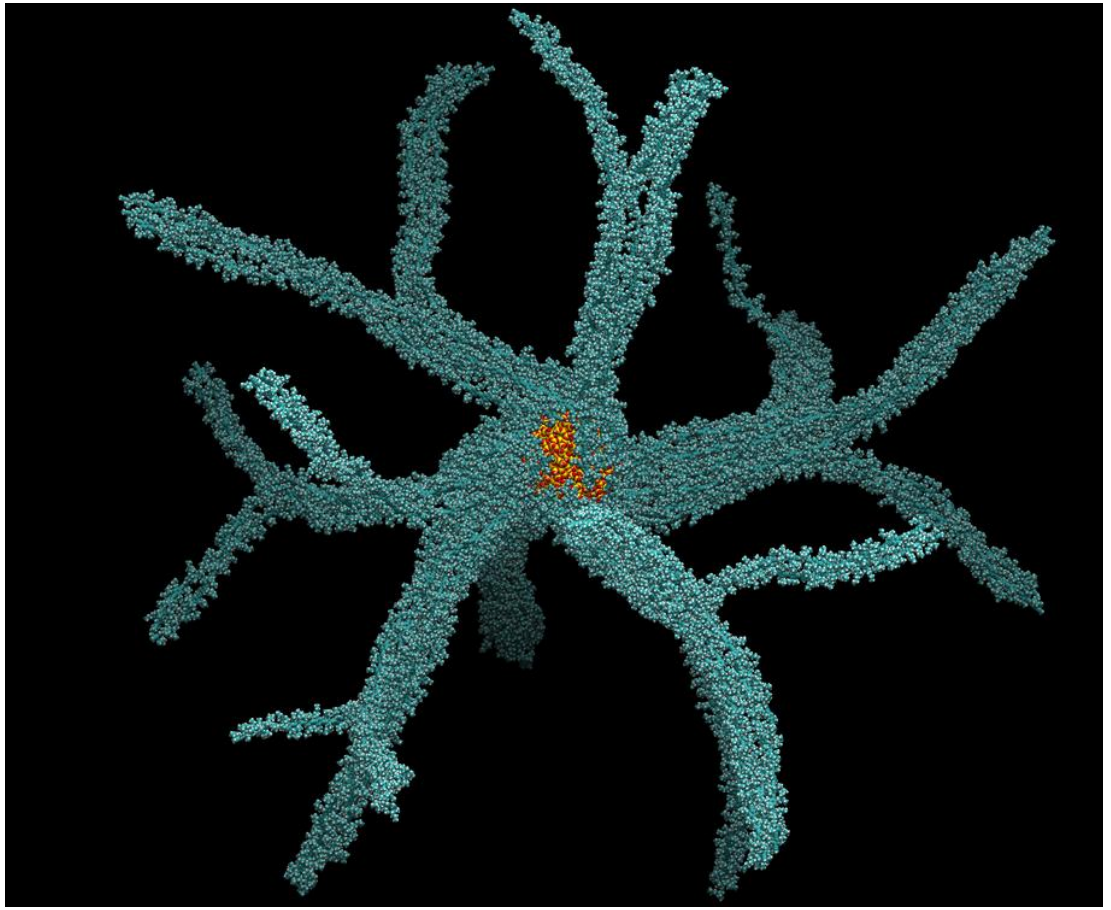


Image from wikimedia

- Simulations the motion of individual molecules
- Widely used in fields from materials science to biology
- Computes the interaction forces between and within molecules according to potential functions

LAMMPS

- Large-scale Atomic-Molecular Massively Parallel Simulator



Sandia National Labs
http://lammps.sandia.gov

Wide collection of potentials

Open source, support for OpenMP, Xeon Phi, and GPU (CUDA and OpenCL)

Intermolecular Forces

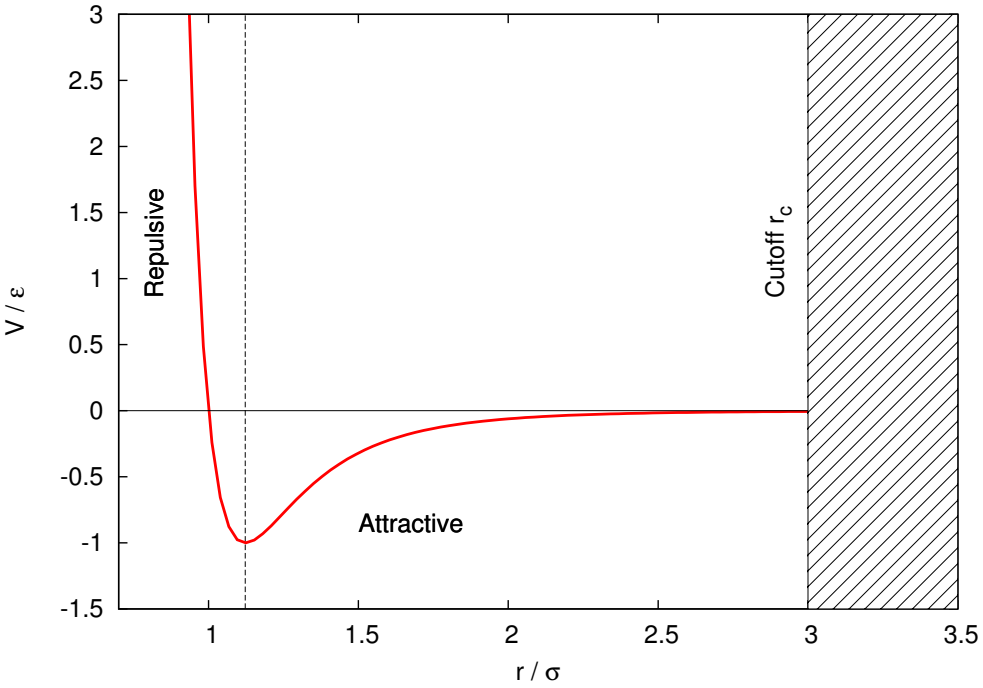
The forces on atoms are commonly taken to be the result of independent pair-wise interactions.

Lennard-Jones potential:

$$\Phi_{LJ} = \sum_{r_{ij}<r_c} 4\,\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right]$$

Where the force on an atom is given by:

$$\vec{F} = -\nabla \Phi$$



But long-range forces can be important!

The electrical potential only decreases as 1/r and doesn't perfectly cancel for polar molecules.

Interfaces can also create asymmetries that inhibit cancellation.

Particle-Particle Particle-Mesh

- PPPM[1] approximates long-range forces without requiring pair-wise calculations.

Four Steps:

1. Determine the charge distribution ρ by mapping particle charges to a grid.

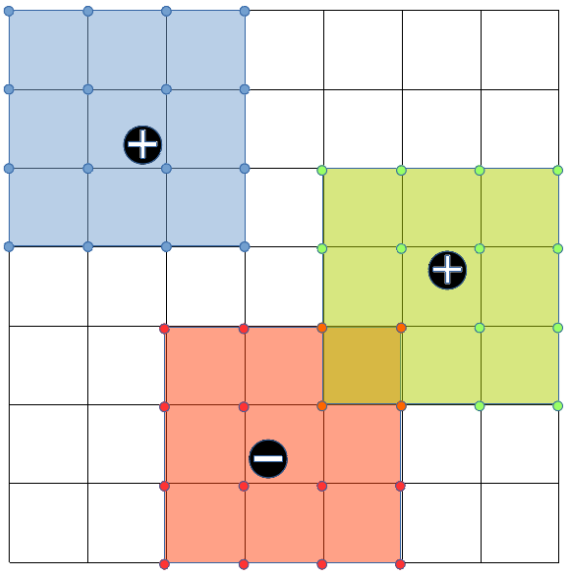
2. Take the Fourier transform of the charge distribution to find the potential:

$$\nabla^2 \Phi = -\frac{\rho}{\epsilon_0}$$

3. Obtain forces due to *all* interactions as the gradient of Φ by inverse Fourier transform:

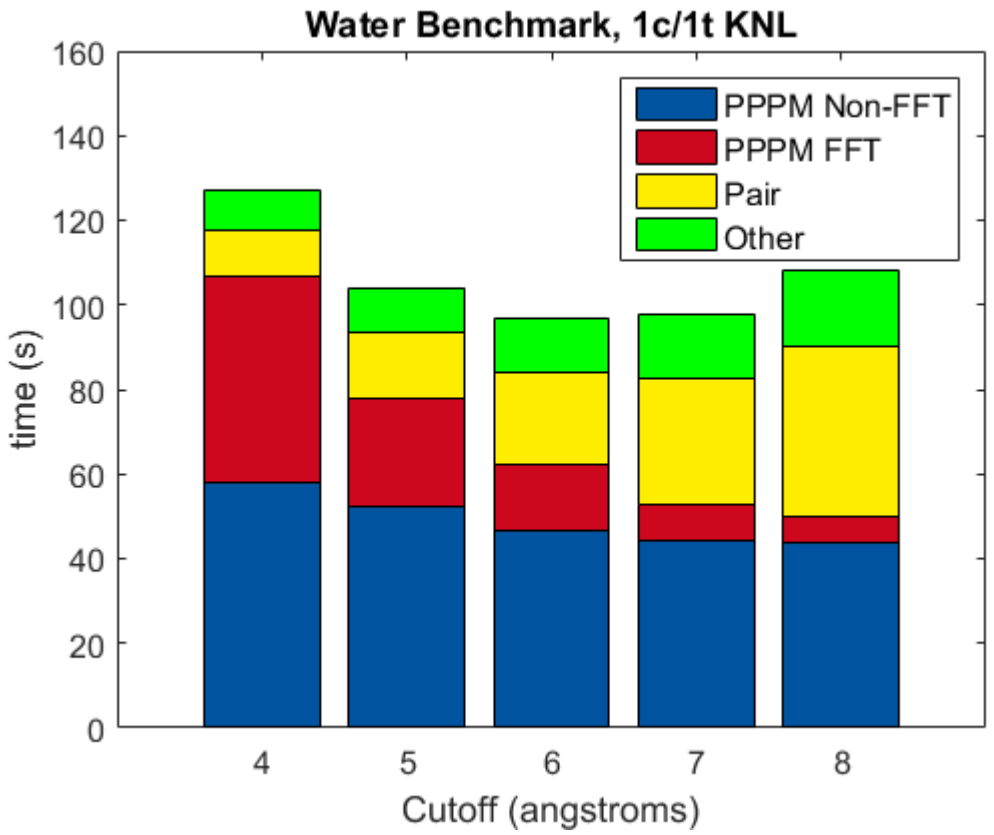
$$\vec{F} = -\nabla \Phi$$

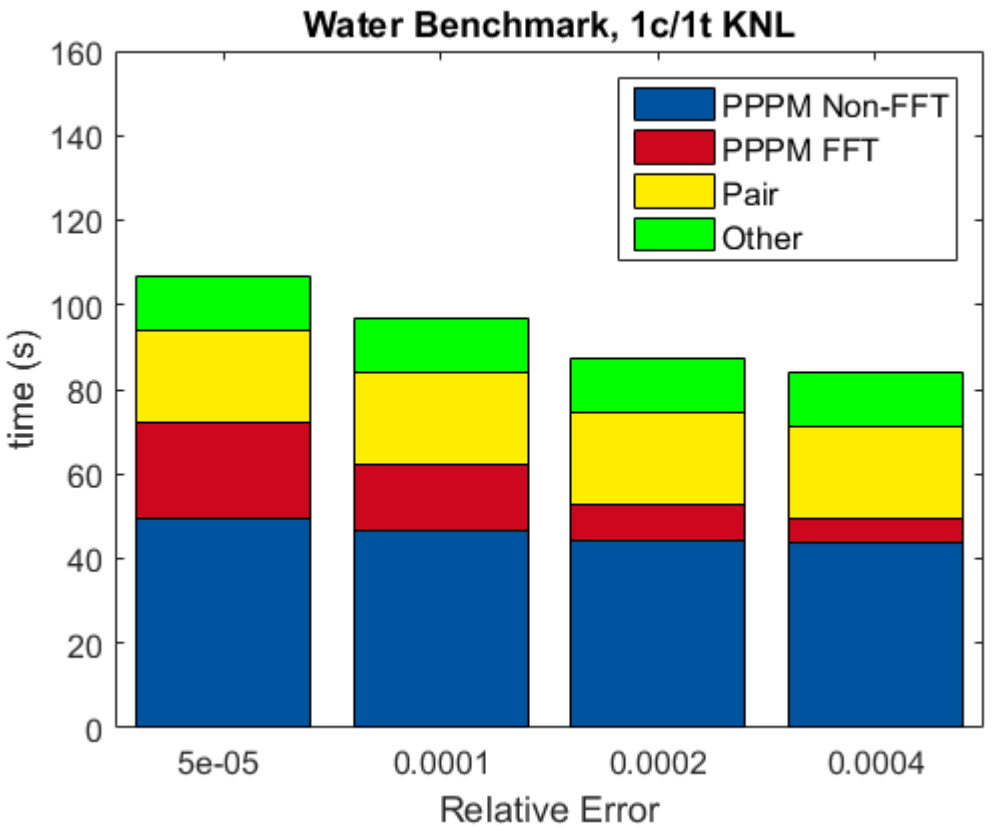
4. Map forces back to the particles.



1. Hockney and Eastwood, 1988

Profiling LAMMPS

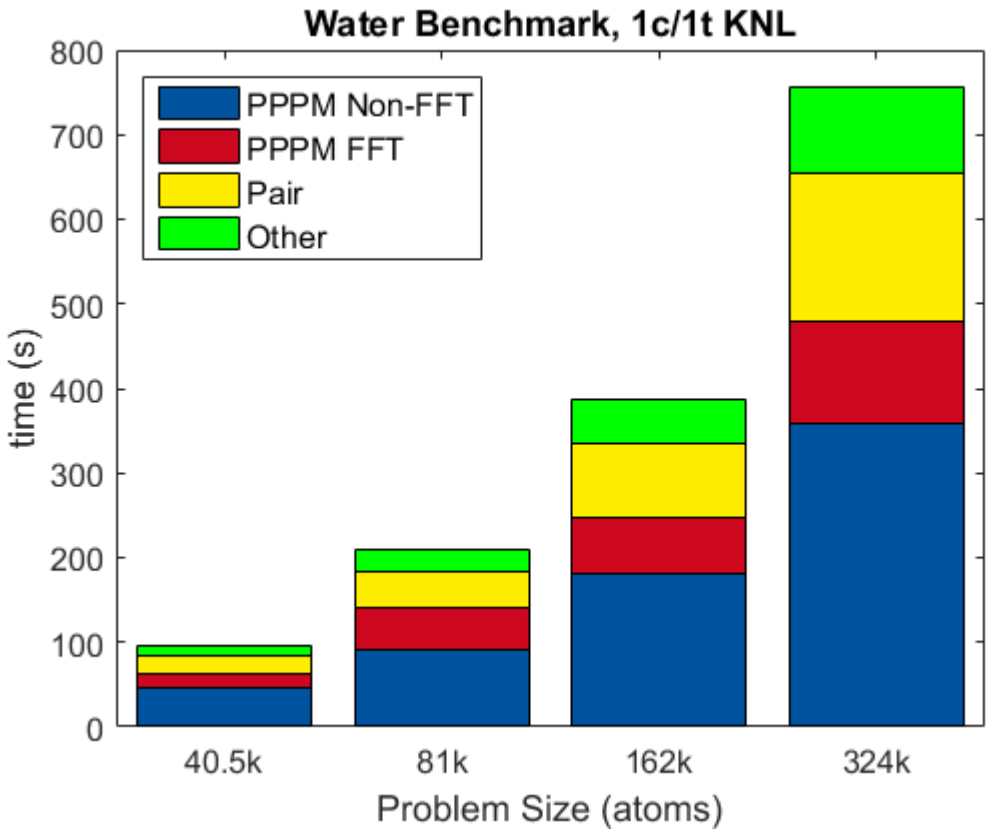




We use the USER-OMP implementation of LAMMPS as a baseline.
Typically: r_c is 6 angstroms, relative error is 0.0001, and stencil size is 5.

The work in FFTs increases rapidly at low cutoffs.
The non-FFT work in PPPM is insensitive to grid size.

Profiling LAMMPS



All parts of the code take time proportional to the size of the problem.

Mapping charges and distributing forces are loops over atoms. The number of FFT grid points is proportional to the domain's volume.

Charge Mapping

```
for (int i = 0; i < nlocal; i++) {

  int nx = part2grid[i][0];
  int ny = part2grid[i][1];
  int nz = part2grid[i][2];
  FFT_SCALAR dx = nx+fshiftone - (x[i].x-lo0)*xi;
  FFT_SCALAR dy = ny+fshiftone - (x[i].y-lo1)*yi;
  FFT_SCALAR dz = nz+fshiftone - (x[i].z-lo2)*zi;

  if( (nz+nlower-nzlo_out)*niy*nix >= jto || (nz-nzlo_out + nupper + 1)*niy*nix <= jfrom )
    continue;

  flt_t rho[3][INTEL_P3M_MAXORDER];

  for (int k = nlower; k <= nupper; k++) {
    FFT_SCALAR r1,r2,r3;
    r1 = r2 = r3 = ZEROF;

    for (int l = order-1; l >= 0; l--) {
      r1 = rho_coeff[l][k] + r1*dx;
      r2 = rho_coeff[l][k] + r2*dy;
      r3 = rho_coeff[l][k] + r3*dz;
    }
    rho[0][k-nlower] = r1;
    rho[1][k-nlower] = r2;
    rho[2][k-nlower] = r3;
  }

  FFT_SCALAR z0 = fdelvolinv * q[i];

  for (int n = nlower; n <= nupper; n++) {
    int mz = (n + nz - nzlo_out)*nix*niy;
    FFT_SCALAR y0 = z0*rho[2][n-nlower];
    for (int m = nlower; m <= nupper; m++) {
      int mzy = mz + (m + ny - nylo_out)*nix;
      FFT_SCALAR x0 = y0*rho[1][m-nlower];
      for (int l = nlower; l <= nupper; l++) {
        int mzyx = mzy + l + nx - nxlo_out;
        if (mzyx >= jto) break;
        if (mzyx < jfrom) continue;
        densityThr[mzyx] += x0*rho[0][l-nlower];
      }
    }
  }
}
}
```

Loop over atoms in MPI rank

Stencil coefficients are polynomials of order stencil size. 3x[stencil size] of them are computed.

Loop over cubic stencil and contribute to grid points

USER-OMP Implementation

Charge Mapping

```
for (int i = 0; i < nlocal; i++) {

  int nx = part2grid[i][0];
  int ny = part2grid[i][1];
  int nz = part2grid[i][2];
  FFT_SCALAR dx = nx+fshiftone - (x[i].x-lo0)*xi;
  FFT_SCALAR dy = ny+fshiftone - (x[i].y-lo1)*yi;
  FFT_SCALAR dz = nz+fshiftone - (x[i].z-lo2)*zi;

  if( (nz+nlower-nzlo_out)*niy*nix >= jto || (nz-nzlo_out + nupper + 1)*niy*nix <= jfrom )
    continue;

  flt_t rho[3][INTEL_P3M_MAXORDER];

  for (int k = nlower; k <= nupper; k++) {
    FFT_SCALAR r1,r2,r3;
    r1 = r2 = r3 = ZEROF;

    for (int l = order-1; l >= 0; l--) {
      r1 = rho_coeff[l][k] + r1*dx;
      r2 = rho_coeff[l][k] + r2*dy;
      r3 = rho_coeff[l][k] + r3*dz;
    }
    rho[0][k-nlower] = r1;
    rho[1][k-nlower] = r2;
    rho[2][k-nlower] = r3;
  }

  FFT_SCALAR z0 = fdelvolinv * q[i];

  for (int n = nlower; n <= nupper; n++) {
    int mz = (n + nz - nzlo_out)*nix*niy;
    FFT_SCALAR y0 = z0*rho[2][n-nlower];
    for (int m = nlower; m <= nupper; m++) {
      int mzy = mz + (m + ny - nylo_out)*nix;
      FFT_SCALAR x0 = y0*rho[1][m-nlower];
      for (int l = nlower; l <= nupper; l++) {
        int mzyx = mzy + l + nx - nxlo_out;
        if (mzyx >= jto) break;
        if (mzyx < jfrom) continue;
        densityThr[mzyx] += x0*rho[0][l-nlower];
      }
    }
  }
}
}
```

```
for (int i = 0; i < nlocal; i++) {
```

Loop over atoms in MPI rank

USER-OMP Implementation

Charge Mapping

```
for (int i = 0; i < nlocal; i++) {

  int nx = part2grid[i][0];
  int ny = part2grid[i][1];
  int nz = part2grid[i][2];
  FFT_SCALAR dx = nx+fshiftone - (x[i].x-lo0)*xi;
  FFT_SCALAR dy = ny+fshiftone - (x[i].y-lo1)*yi;
  FFT_SCALAR dz = nz+fshiftone - (x[i].z-lo2)*zi;

  if( (nz+nlower-nzlo_out)*niy*nix >= jto || (nz-nzlo_
    continue;

  flt_t rho[3][INTEL_P3M_MAXORDER];

  for (int k = nlower; k <= nupper; k++) {
    FFT_SCALAR r1,r2,r3;
    r1 = r2 = r3 = ZEROF;

    for (int l = order-1; l >= 0; l--) {
      r1 = rho_coeff[l][k] + r1*dx;
      r2 = rho_coeff[l][k] + r2*dy;
      r3 = rho_coeff[l][k] + r3*dz;
    }
    rho[0][k-nlower] = r1;
    rho[1][k-nlower] = r2;
    rho[2][k-nlower] = r3;
  }

  FFT_SCALAR z0 = fdelvolinv * q[i];

  for (int n = nlower; n <= nupper; n++) {
    int mz = (n + nz - nzlo_out)*nix*niy;
    FFT_SCALAR y0 = z0*rho[2][n-nlower];
    for (int m = nlower; m <= nupper; m++) {
      int mzy = mz + (m + ny - nylo_out)*nix;
      FFT_SCALAR x0 = y0*rho[1][m-nlower];
      for (int l = nlower; l <= nupper; l++) {
        int mzyx = mzy + l + nx - nxlo_out;
        if (mzyx >= jto) break;
        if (mzyx < jfrom) continue;
        densityThr[mzyx] += x0*rho[0][l-nlower];
      }
    }
  }
}
}
```

USER-OMP Implementation

Stencil coefficients are polynomials of order stencil size. 3x[stencil size] are computed.

```
for (int k = nlower; k <= nupper; k++) {
  FFT_SCALAR r1,r2,r3;
  r1 = r2 = r3 = ZEROF;

  for (int l = order-1; l >= 0; l--) {
    r1 = rho_coeff[l][k] + r1*dx;
    r2 = rho_coeff[l][k] + r2*dy;
    r3 = rho_coeff[l][k] + r3*dz;
  }
  rho[0][k-nlower] = r1;
  rho[1][k-nlower] = r2;
  rho[2][k-nlower] = r3;
}
```

Charge Mapping

```
for (int i = 0; i < nlocal; i++) {

  int nx = part2grid[i][0];
  int ny = part2grid[i][1];
  int nz = part2grid[i][2];
  FFT_SCALAR dx = nx+fshiftone - (x[i].x-lo0)*xi;
  FFT_SCALAR dy = ny+fshiftone - (x[i].y-lo1)*yi;
  FFT_SCALAR dz = nz+fshiftone - (x[i].z-lo2)*zi;

  if( (nz+nlower-nzlo_out)*niy*nix >= jto || (nz-nzlo_out + nupper + 1)*niy*nix <= jfrom )
    continue;

  flt_t rho[3][INTEL_P3M_MAXORDER];

  for (int k = nlower; k <= nupper; k++) {
    FFT_SCALAR r1,r2,r3;
    r1 = r2 = r3 = ZEROF;

    for (int l = order-1; l >= 0; l--) {
      r1 = rho_coeff[l][k] + r1*dx;
      r2 = rho_coeff[l][k] + r2*dy;
      r3 = rho_coeff[l][k] + r3*dz;
    }
    rho[0][k-nlower] = r1;
    rho[1][k-nlower] = r2;
    rho[2][k-nlower] = r3;
  }

  FFT_SCALAR z0 = fdelvolinv * q[i];

  for (int n = nlower; n <= nupper; n++) {
    int mz = (n + nz - nzlo_out)*nix*niy;
    FFT_SCALAR y0 = z0*rho[2][n-nlower];
    for (int m = nlower; m <= nupper; m++)
      int mzy = mz + (m + ny - nylo_out)*n
      FFT_SCALAR x0 = y0*rho[1][m-nlower];
      for (int l = nlower; l <= nupper; l+
        int mzyx = mzy + l + nx - nxlo_out
        if (mzyx >= jto) break;
        if (mzyx < jfrom) continue;
        densityThr[mzyx] += x0*rho[0][l-nl
      }
    }
  }
}
```

Loop over cubic stencil and contribute to grid points

```
for (int n = nlower; n <= nupper; n++) {
  int mz = (n + nz - nzlo_out)*nix*niy;
  FFT_SCALAR y0 = z0*rho[2][n-nlower];
  for (int m = nlower; m <= nupper; m++) {
    int mzy = mz + (m + ny - nylo_out)*nix;
    FFT_SCALAR x0 = y0*rho[1][m-nlower];
    for (int l = nlower; l <= nupper; l++) {
      int mzyx = mzy + l + nx - nxlo_out;
      if (mzyx >= jto) break;
      if (mzyx < jfrom) continue;
      densityThr[mzyx] += x0*rho[0][l-nlower];
    }
  }
}
```

USER-OMP Implementation

Charge Mapping

```
for (int i = 0; i < nlocal; i++) {

  int nx = part2grid[i][0];
  int ny = part2grid[i][1];
  int nz = part2grid[i][2];
  FFT_SCALAR dx = nx+fshiftone - (x[i].x-lo0)*xi;
  FFT_SCALAR dy = ny+fshiftone - (x[i].y-lo1)*yi;
  FFT_SCALAR dz = nz+fshiftone - (x[i].z-lo2)*zi;

  if( (nz+nlower-nzlo_out)*niy*nix >= jto || (nz-nzlo_out
    continue;

  flt_t rho[3][INTEL_P3M_MAXORDER];

  for (int k = nlower; k <= nupper; k++) {
    FFT_SCALAR r1,r2,r3;
    r1 = r2 = r3 = ZEROF;

    for (int l = order-1; l >= 0; l--) {
      r1 = rho_coeff[l][k] + r1*dx;
      r2 = rho_coeff[l][k] + r2*dy;
      r3 = rho_coeff[l][k] + r3*dz;
    }
    rho[0][k-nlower] = r1;
    rho[1][k-nlower] = r2;
    rho[2][k-nlower] = r3;
  }

  FFT_SCALAR z0 = fdelvolinv * q[i];

  for (int n = nlower; n <= nupper; n++) {
    int mz = (n + nz - nzlo_out)*nix*niy;
    FFT_SCALAR y0 = z0*rho[2][n-nlower];
    for (int m = nlower; m <= nupper; m++) {
      int mzy = mz + (m + ny - nylo_out)*nix;
      FFT_SCALAR x0 = y0*rho[1][m-nlower];
      for (int l = nlower; l <= nupper; l++) {
        int mzyx = mzy + l + nx - nxlo_out;
        if (mzyx >= jto) break;
        if (mzyx < jfrom) continue;
        densityThr[mzyx] += x0*rho[0][l-nlower];
      }
    }
  }
}
}
```

USER-OMP Implementation

```
  for (int i = jfrom; i < jto; i++) {

    int nx = part2grid[i][0];
    int ny = part2grid[i][1];
    int nz = part2grid[i][2];

    int nysum = nlower + ny - nylo_out;
    int nxsum = nlower + nx - nxlo_out + ngrid*tid;
    int nzsum = (nlower + nz - nzlo_out)*nix*niy + nysum*nix + nxsum;

    FFT_SCALAR dx = nx+fshiftone - (x[i].x-lo0)*xi;
    FFT_SCALAR dy = ny+fshiftone - (x[i].y-lo1)*yi;
    FFT_SCALAR dz = nz+fshiftone - (x[i].z-lo2)*zi;

#pragma simd
    for (int k = nlower; k <= nupper; k++) {
      FFT_SCALAR r1,r2,r3;
      r1 = r2 = r3 = ZEROF;

      for (int l = order-1; l >= 0; l--) {
        r1 = rho_coeff[l][k] + r1*dx;
        r2 = rho_coeff[l][k] + r2*dy;
        r3 = rho_coeff[l][k] + r3*dz;
      }
      rho[0][k-nlower] = r1;
      rho[1][k-nlower] = r2;
      rho[2][k-nlower] = r3;
    }

    FFT_SCALAR z0 = fdelvolinv * q[i];

#pragma loop_count=7
    for (int n = 0; n < tripcount; n++) {
      int mz = n*nix*niy + nzsum;
      FFT_SCALAR y0 = z0*rho[2][n];

#pragma loop_count=7
      for (int m = 0; m < tripcount; m++) {
        int mzy = mz + m*nix;
        FFT_SCALAR x0 = y0*rho[1][m];
#pragma simd
        for (int l = 0; l < 8; l++) {
          int mzyx = mzy + l;
          localDensity[mzyx] += x0*rho[0][l];
        }
      }
    }

  }
}
```

Our Implementation

Thread over atoms

#pragma simd
for coefficients

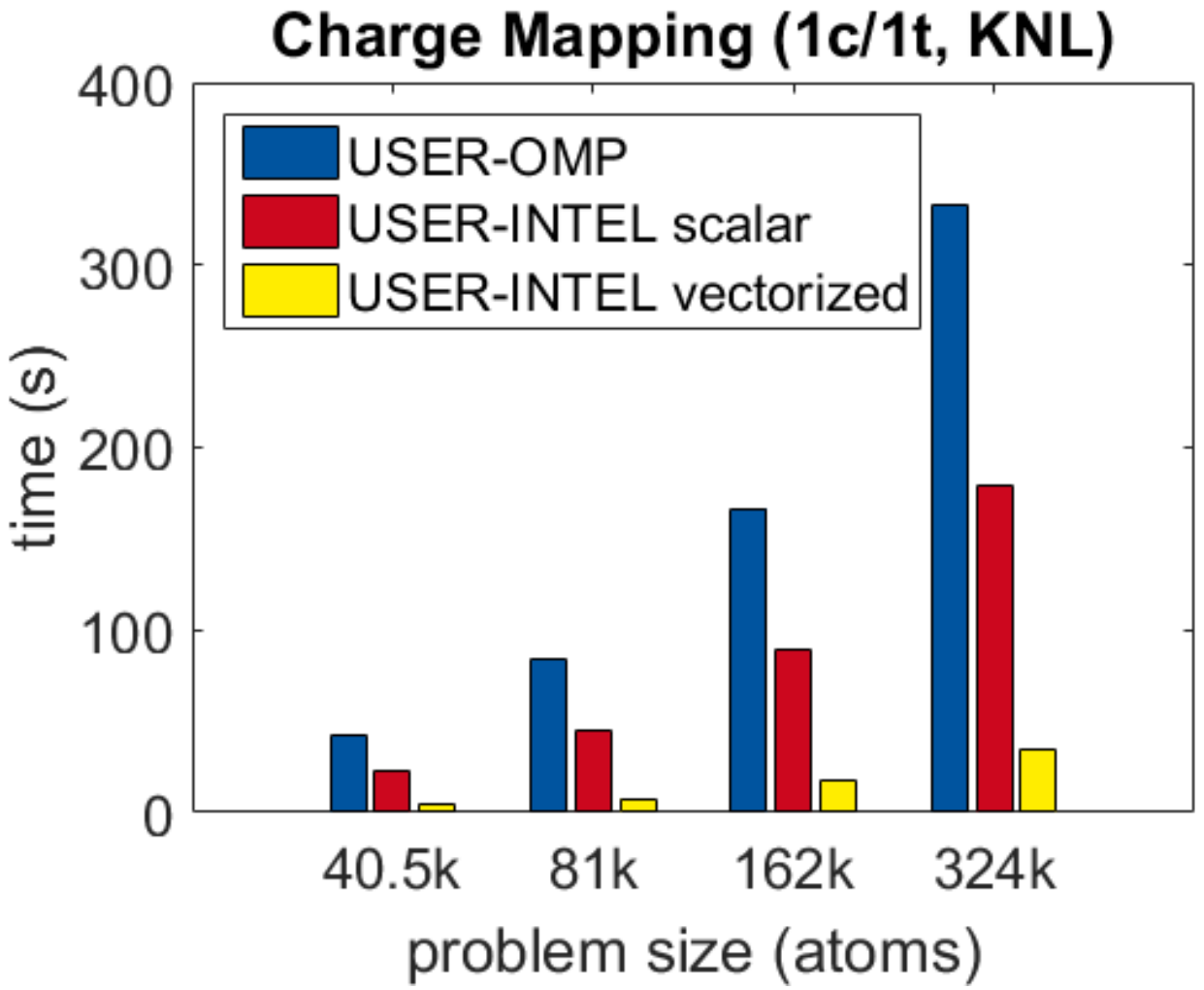
Charge Mapping

Innermost loop vectorized with bigger stencil.
Private grids prevent race conditions.

```
#pragma loop_count=7
    for (int n = 0; n < tripcount; n++) {
      int mz = n*nix*niy + nzsum;
      FFT_SCALAR y0 = z0*rho[2][n];

#pragma loop_count=7
      for (int m = 0; m < tripcount; m++) {
        int mzy = mz + m*nix;
        FFT_SCALAR x0 = y0*rho[1][m];
#pragma simd
        for (int l = 0; l < 8; l++) {
          int mzyx = mzy + l;
          localDensity[mzyx] += x0*rho[0][l];
        }
      }
    }
```

Our Implementation

Charge Mapping



10x speedup over USER-OMP implementation, 5x due to vectorization.
Simulations run using a 7 point stencil.

Distributing Forces

Very similar to charge mapping:
Computes stencil coefficients
Loops over stencil points.

More work and accesses more memory

Water benchmark:
40.5k atoms
884k FFT grid points

```
#if defined(LMP_SIMD_COMPILER)
#pragma vector aligned nontemporal
#pragma simd
#endif
for (int i = iifrom; i < iito; i++) {
  int nx = part2grid[i][0];
  int ny = part2grid[i][1];
  int nz = part2grid[i][2];
  FFT_SCALAR dx = nx+fshiftone - (x[i].x-lo0)*xi;
  FFT_SCALAR dy = ny+fshiftone - (x[i].y-lo1)*yi;
  FFT_SCALAR dz = nz+fshiftone - (x[i].z-lo2)*zi;

  flt_t rho[3][INTEL_P3M_MAXORDER];

  for (int k = nlower; k <= nupper; k++) {
    FFT_SCALAR r1 = rho_coeff[order-1][k];
    FFT_SCALAR r2 = rho_coeff[order-1][k];
    FFT_SCALAR r3 = rho_coeff[order-1][k];
    for (int l = order-2; l >= 0; l--) {
      r1 = rho_coeff[l][k] + r1*dx;
      r2 = rho_coeff[l][k] + r2*dy;
      r3 = rho_coeff[l][k] + r3*dz;
    }
    rho[0][k-nlower] = r1;
    rho[1][k-nlower] = r2;
    rho[2][k-nlower] = r3;
  }

  FFT_SCALAR ekx, eky, ekz;
  ekx = eky = ekz = ZEROF;
  for (int n = nlower; n <= nupper; n++) {
    int mz = n+nz;
    FFT_SCALAR z0 = rho[2][n-nlower];
    for (int m = nlower; m <= nupper; m++) {
      int my = m+ny;
      FFT_SCALAR y0 = z0*rho[1][m-nlower];
      for (int l = nlower; l <= nupper; l++) {
        int mx = l+nx;
        FFT_SCALAR x0 = y0*rho[0][l-nlower];
        ekx -= x0*vdx_brick[mz][my][mx];
        eky -= x0*vdy_brick[mz][my][mx];
        ekz -= x0*vdz_brick[mz][my][mx];
      }
    }
  }
```

#pragma simd around atom loop

Update 3 force components

Distributing Forces

Update 3 force components

```
FFT_SCALAR ekx, eky, ekz;
ekx = eky = ekz = ZEROF;
for (int n = nlower; n <= nupper; n++) {
  int mz = n+nz;
  FFT_SCALAR z0 = rho[2][n-nlower];
  for (int m = nlower; m <= nupper; m++) {
    int my = m+ny;
    FFT_SCALAR y0 = z0*rho[1][m-nlower];
    for (int l = nlower; l <= nupper; l++) {
      int mx = l+nx;
      FFT_SCALAR x0 = y0*rho[0][l-nlower];
      ekx -= x0*vdx_brick[mz][my][mx];
      eky -= x0*vdy_brick[mz][my][mx];
      ekz -= x0*vdz_brick[mz][my][mx];
    }
  }
}
```

```
#if defined(LMP_SIMD_COMPILER)
#pragma vector aligned nontemporal
#pragma simd
#endif
for (int i = iifrom; i < iito; i++) {
  int nx = part2grid[i][0];
  int ny = part2grid[i][1];
  int nz = part2grid[i][2];
  FFT_SCALAR dx = nx+fshiftone - (x[i].x-lo0)*xi;
  FFT_SCALAR dy = ny+fshiftone - (x[i].y-lo1)*yi;
  FFT_SCALAR dz = nz+fshiftone - (x[i].z-lo2)*zi;

  flt_t rho[3][INTEL_P3M_MAXORDER];

  for (int k = nlower; k <= nupper; k++) {
    FFT_SCALAR r1 = rho_coeff[order-1][k];
    FFT_SCALAR r2 = rho_coeff[order-1][k];
    FFT_SCALAR r3 = rho_coeff[order-1][k];
    for (int l = order-2; l >= 0; l--) {
      r1 = rho_coeff[l][k] + r1*dx;
      r2 = rho_coeff[l][k] + r2*dy;
      r3 = rho_coeff[l][k] + r3*dz;
    }
    rho[0][k-nlower] = r1;
    rho[1][k-nlower] = r2;
    rho[2][k-nlower] = r3;
  }

  FFT_SCALAR ekx, eky, ekz;
  ekx = eky = ekz = ZEROF;
  for (int n = nlower; n <= nupper; n++) {
    int mz = n+nz;
    FFT_SCALAR z0 = rho[2][n-nlower];
    for (int m = nlower; m <= nupper; m++) {
      int my = m+ny;
      FFT_SCALAR y0 = z0*rho[1][m-nlower];
      for (int l = nlower; l <= nupper; l++) {
        int mx = l+nx;
        FFT_SCALAR x0 = y0*rho[0][l-nlower];
        ekx -= x0*vdx_brick[mz][my][mx];
        eky -= x0*vdy_brick[mz][my][mx];
        ekz -= x0*vdz_brick[mz][my][mx];
      }
    }
  }
```

Distributing Forces

Inner SIMD

```
  for (int i = iifrom; i < iito; i++) {
    int nx = part2grid[i][0];
    int ny = part2grid[i][1];
    int nz = part2grid[i][2];

    int nxsum = nx + nlower;
    int nysum = ny + nlower;
    int nzsum = nz + nlower;

    FFT_SCALAR dx = nx+fshiftone - (x[i].x-lo0)*xi;
    FFT_SCALAR dy = ny+fshiftone - (x[i].y-lo1)*yi;
    FFT_SCALAR dz = nz+fshiftone - (x[i].z-lo2)*zi;

    #pragma simd
    for (int k = nlower; k <= nupper; k++) {
      FFT_SCALAR r1, r2, r3;
      r1 = r2 = r3 = ZEROF;
      for (int l = order-1; l >= 0; l--) {
        r1 = rho_coeff[l][k] + r1*dx;
        r2 = rho_coeff[l][k] + r2*dy;
        r3 = rho_coeff[l][k] + r3*dz;
      }
      rho[0][k-nlower] = r1;
      rho[1][k-nlower] = r2;
      rho[2][k-nlower] = r3;
    }

    FFT_SCALAR ekx[8]={ZEROF}, eky[8]={ZEROF}, ekz[8]={ZEROF};
    FFT_SCALAR ekxsum, ekysum, ekzsum;
    ekxsum = ekysum = ekzsum = ZEROF;

    for (int n = 0; n < tripcount; n++) {
      int mz = n+nzsum;
      FFT_SCALAR z0 = rho[2][n];

      for (int m = 0; m < tripcount; m++) {
        int my = m+nysum;
        FFT_SCALAR y0 = z0*rho[1][m];
#pragma simd
        for (int l = 0; l < 8; l++) {
          int mx = l+nxsum;
          FFT_SCALAR x0 = y0*rho[0][l];
          ekx[l] -= x0*vdx_brick[mz][my][mx];
          eky[l] -= x0*vdy_brick[mz][my][mx];
          ekz[l] -= x0*vdz_brick[mz][my][mx];
        }
      }
    }

    for (int l = 0; l < tripcount; l++){
        ekxsum += ekx[l];
        ekysum += eky[l];
        ekzsum += ekz[l];
    }
```

Distributing Forces

Inner SIMD

```
for (int i = iifrom; i < iito; i++) {
  int nx = part2grid[i][0];
  int ny = part2grid[i][1];
  int nz = part2grid[i][2];

  int nxsum = nx + nlower;
  int nysum = ny + nlower;
  int nzsum = nz + nlower;

  FFT_SCALAR dx = nx+fshiftone - (x[i].x-lo0)*xi;
  FFT_SCALAR dy = ny+fshiftone - (x[i].y-lo1)*yi;
  FFT_SCALAR dz = nz+fshiftone - (x[i].z-lo2)*zi;

#prag
for (
  FFT
  r1
  for
    r
    r
    r
  }
  rho
  rho
  rho
}
FFT_S
FFT_S
ekxsu

for (
  int
  FFT

  for
    i
    F
#pragma s
    f
  }
}

for (
    e
    e
    e
}
```

```
    for (int n = 0; n < tripcount; n++) {
      int mz = n+nzsum;
      FFT_SCALAR z0 = rho[2][n];

      for (int m = 0; m < tripcount; m++) {
        int my = m+nysum;
        FFT_SCALAR y0 = z0*rho[1][m];
#pragma simd
        for (int l = 0; l < 8; l++) {
          int mx = l+nxsum;
          FFT_SCALAR x0 = y0*rho[0][l];
          ekx[l] -= x0*vdx_brick[mz][my][mx];
          eky[l] -= x0*vdy_brick[mz][my][mx];
          ekz[l] -= x0*vdz_brick[mz][my][mx];
        }
      }
    }

    for (int l = 0; l < tripcount; l++){
      ekxsum += ekx[l];
      ekysum += eky[l];
      ekzsum += ekz[l];
    }
```

10% faster with tripcount instead of 7

50% faster with 8 instead of 7

Reduction of force component arrays

Distributing Forces

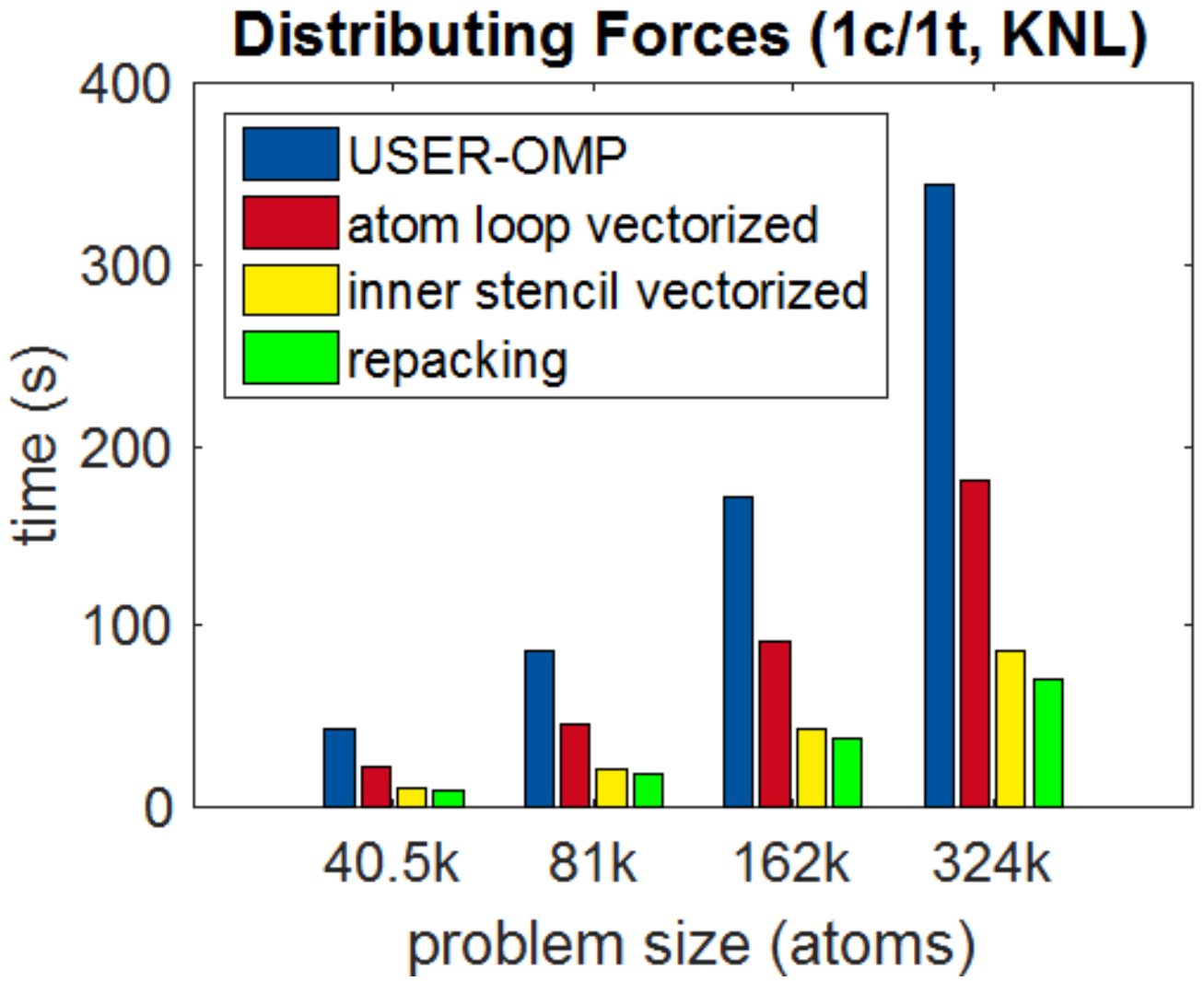
```
for (int iz = 0; iz < niz; iz++) {
  for (int iy = 0; iy < niy; iy++) {
    for ( int ix = 0; ix < nix; ix++) {
    int iter = 2*(iz*niy*nix + iy*nix + ix);
      vdxy_brick[iter] = vdx_brick[nzlo_out + iz][nylo_out + iy][nxlo_out + ix];
      vdxy_brick[iter+1] = vdy_brick[nzlo_out + iz][nylo_out + iy][nxlo_out + ix];
      vdz0_brick[iter] = vdz_brick[nzlo_out + iz][nylo_out + iy][nxlo_out + ix];
      vdz0_brick[iter+1] = 0.;
    }
  }
}
```

```
    for (int n = 0; n < tripcount; n++) {
      int mz = 2*n*nix*niy+nzsum;
      FFT_SCALAR z0 = rho2[n];

      for (int m = 0; m < tripcount; m++) {
        int mzy = mz + 2*m*nix;
        FFT_SCALAR y0 = z0*rho1[m];
#pragma simd
        for (int l = 0; l < 16; l++) {
          FFT_SCALAR x0 = y0*rho0[l];
          ekxy[l] -= x0*vdxy_brick[mzy+l];
          ekz0[l] -= x0*vdz0_brick[mzy+l];
        }
      }
    }

    for (int l = 0; l < 16; l=l+2){
        ekxsum += ekxy[l];
        ekysum += ekxy[l+1];
        ekzsum += ekz0[l];
    }
```

3 vector operations instead of 4: 60% faster

16-iteration loops are faster on KNL, even with extra 0s

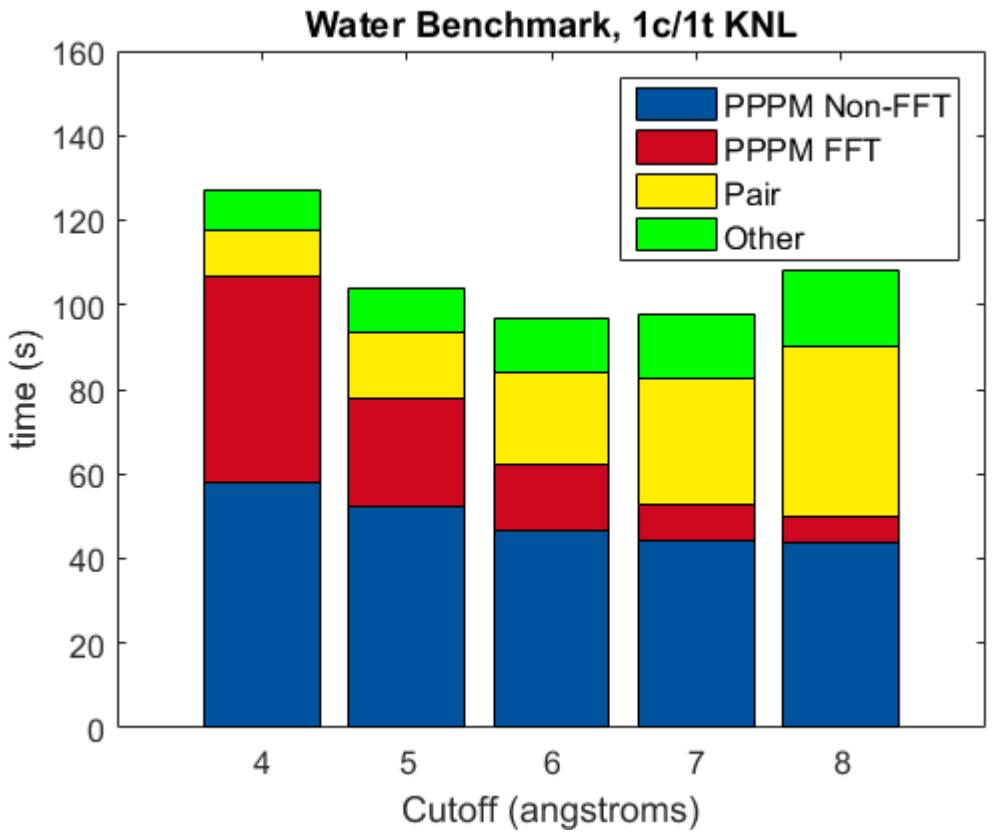
Distributing Forces

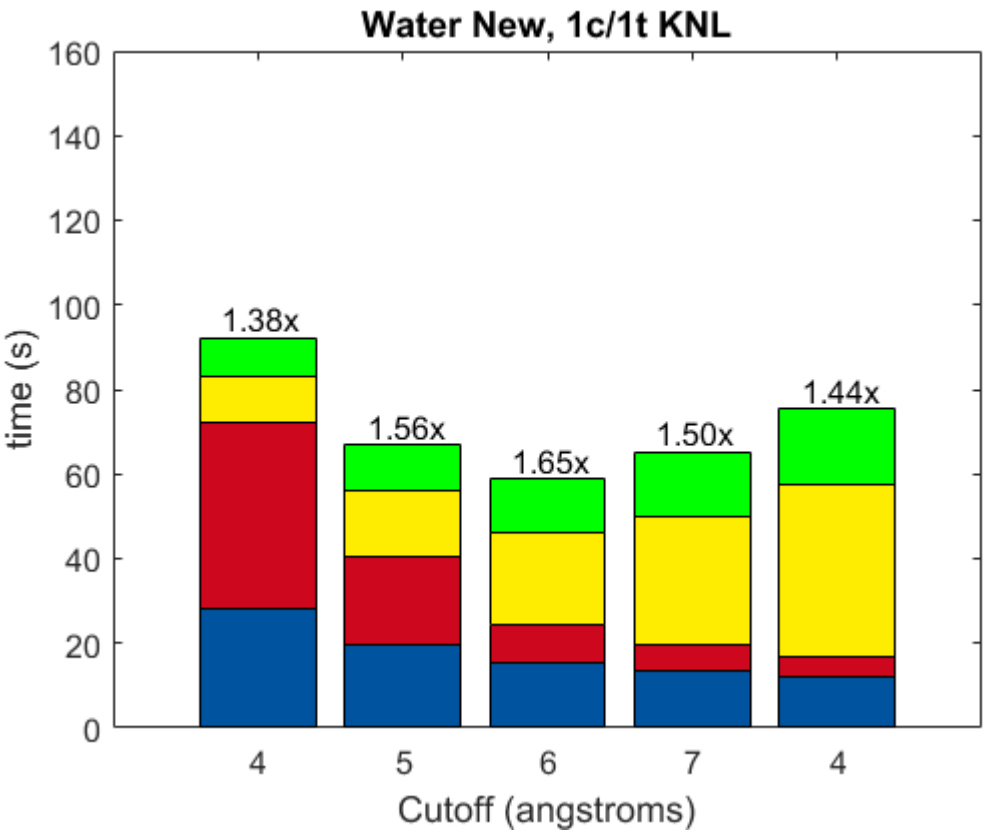


5-6x speedup over the USER-OMP implementation

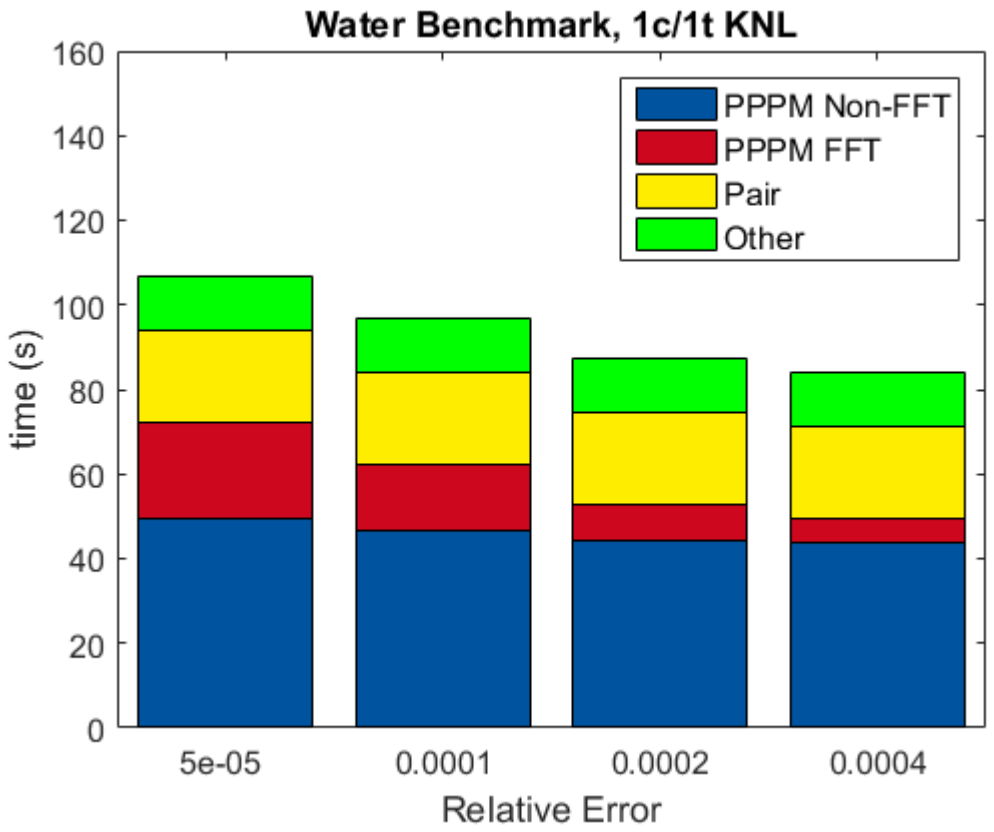
Simulations run using a 7 point stencil

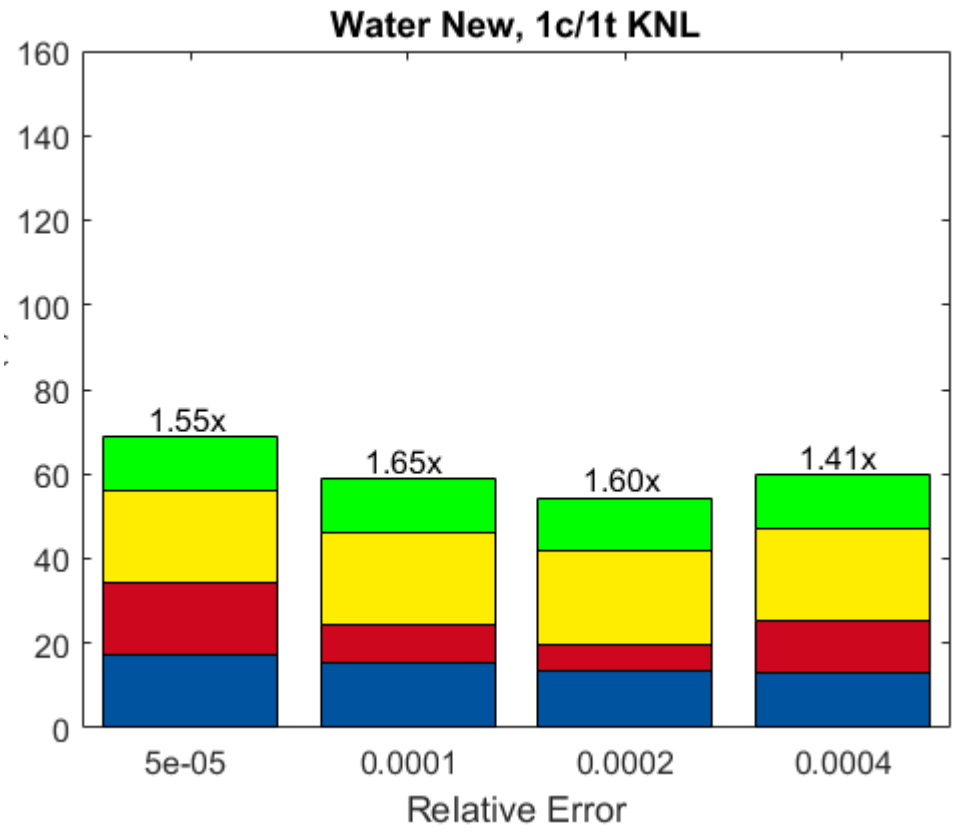
Overall Speedup



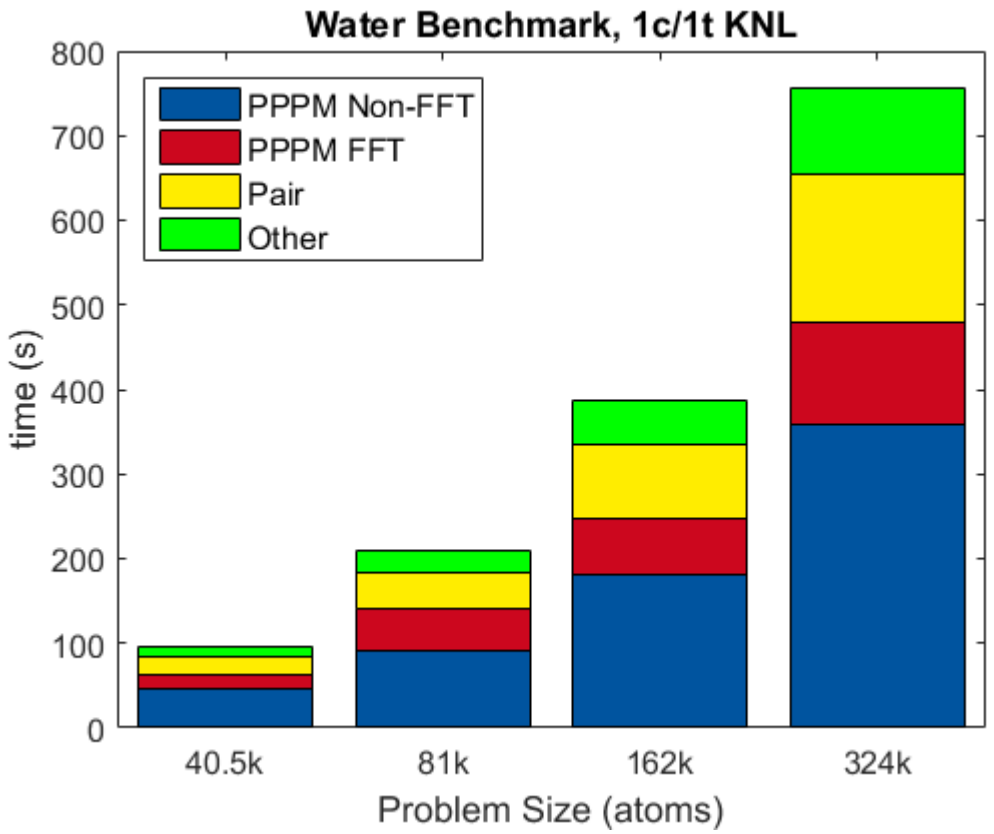


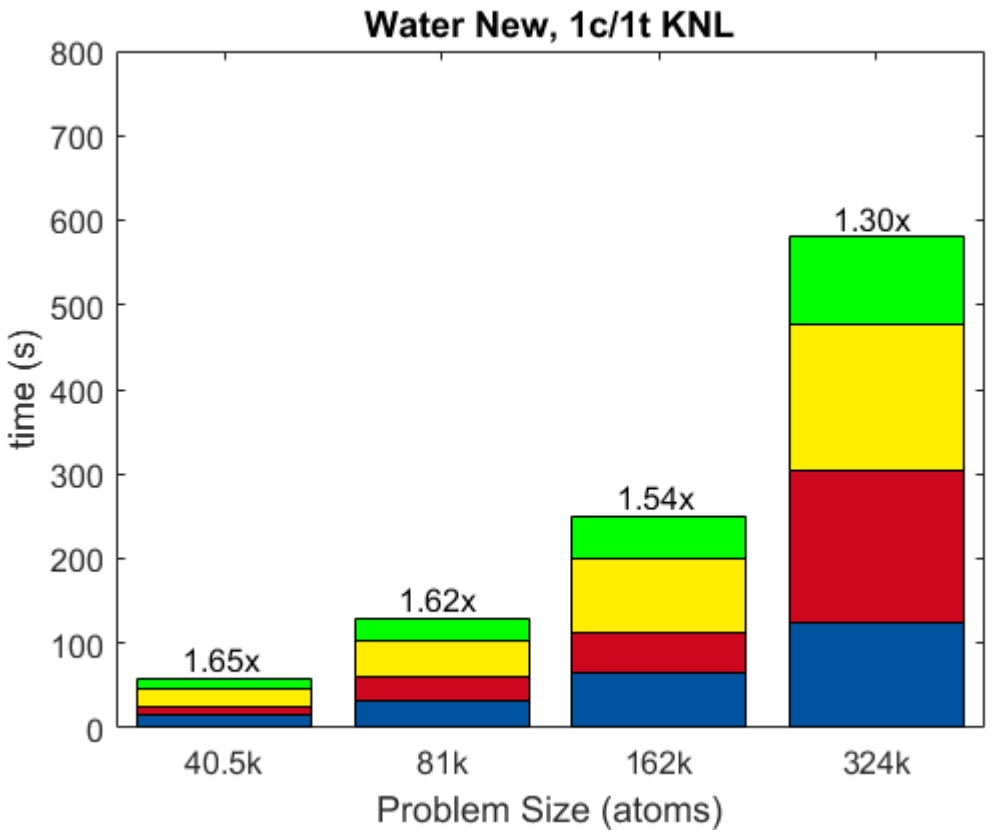
Overall Speedup





Overall Speedup

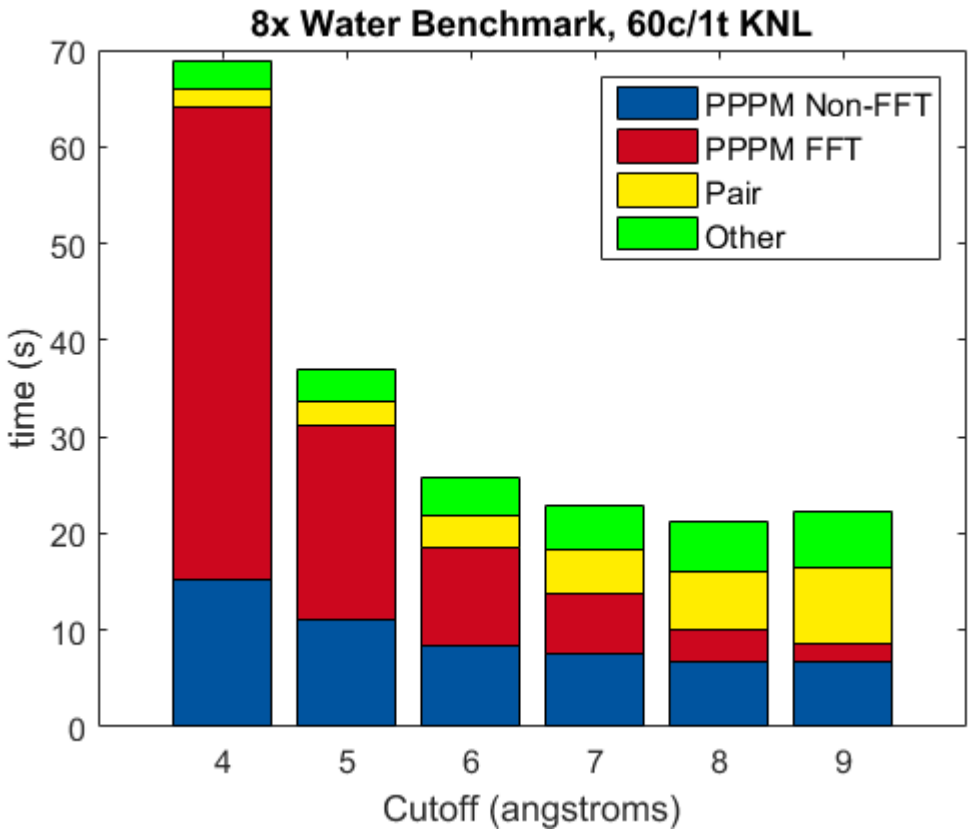


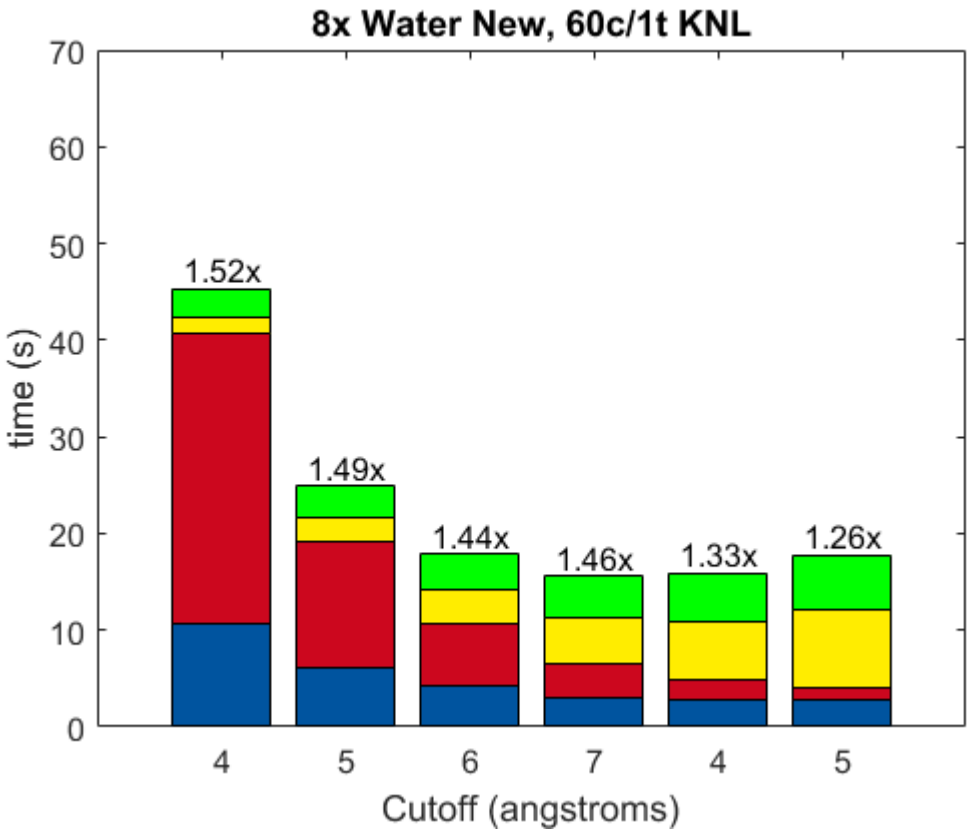


Speedups on 1 core between 1.38x and 1.65x

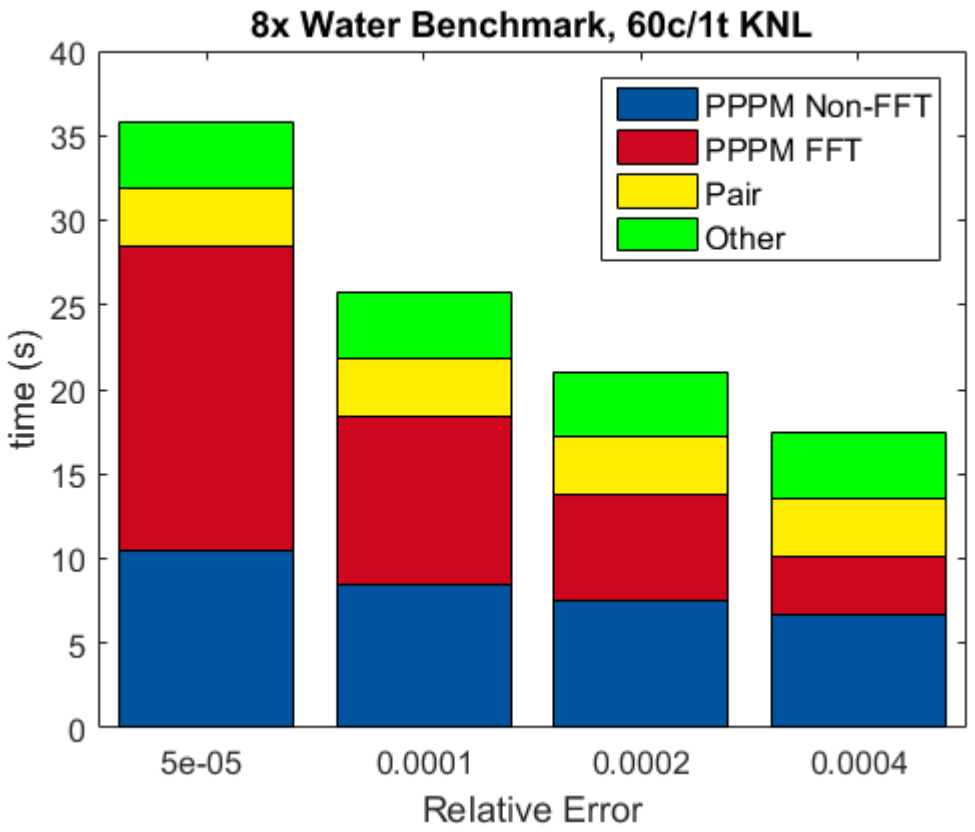
PPPM routines are sped up between 1.48x and 2.65x, sensitive to FFT accuracy.

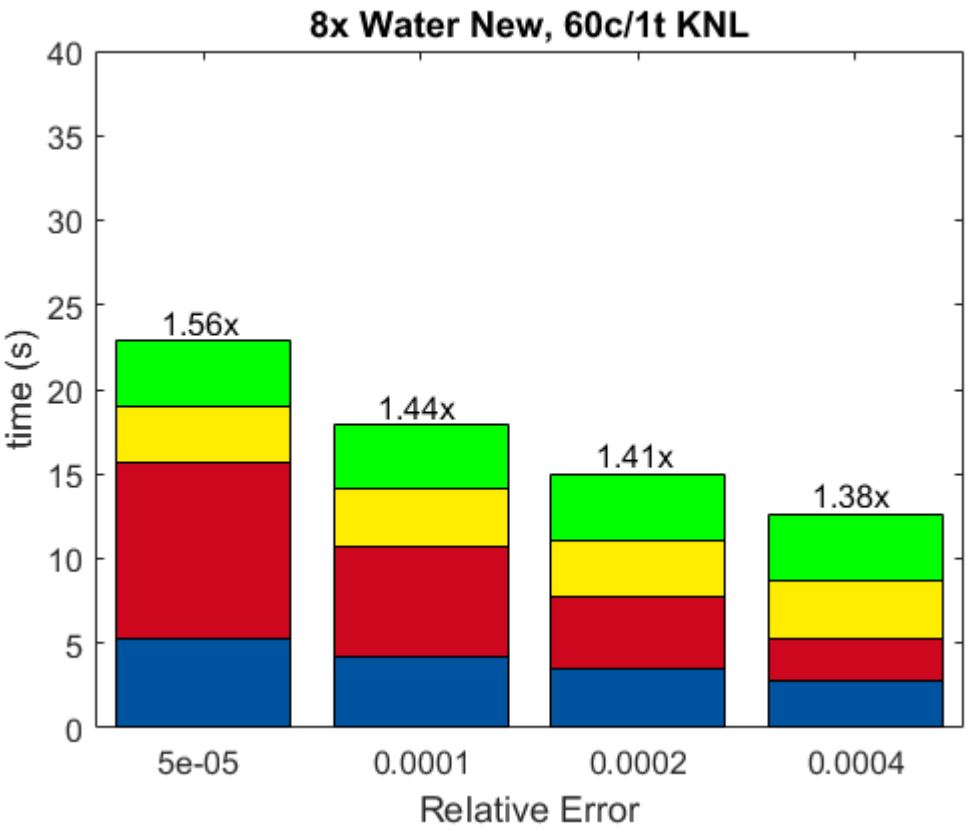
Overall Speedup - Parallel



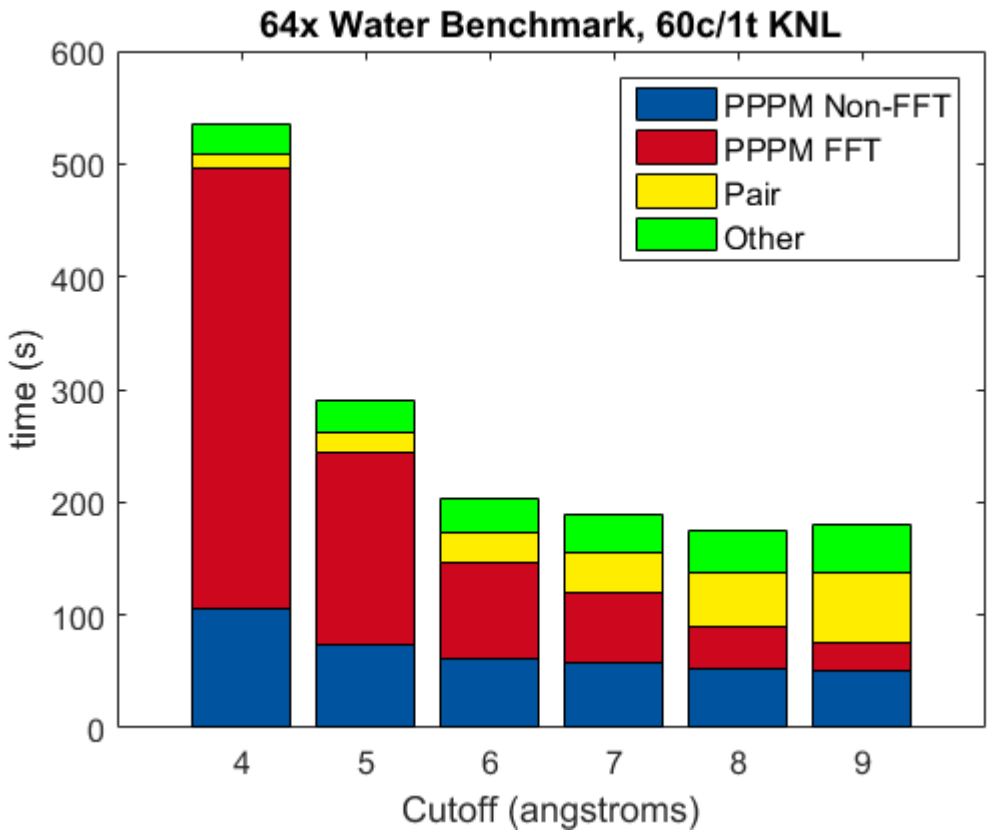


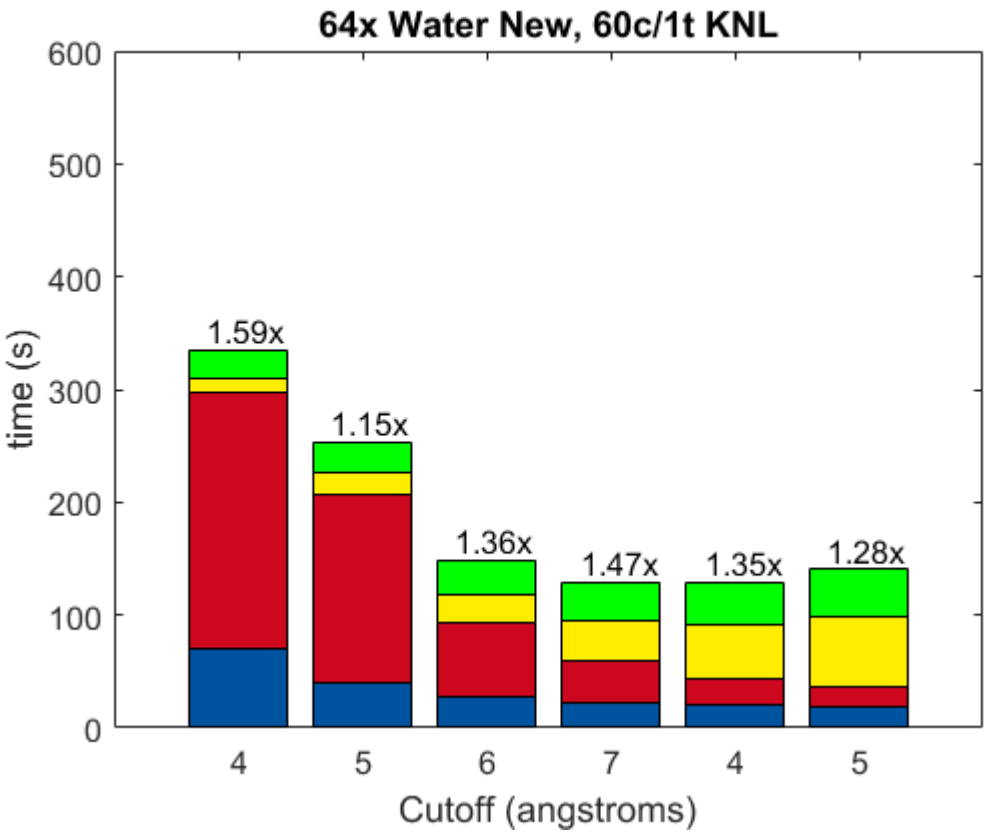
Overall Speedup - Parallel



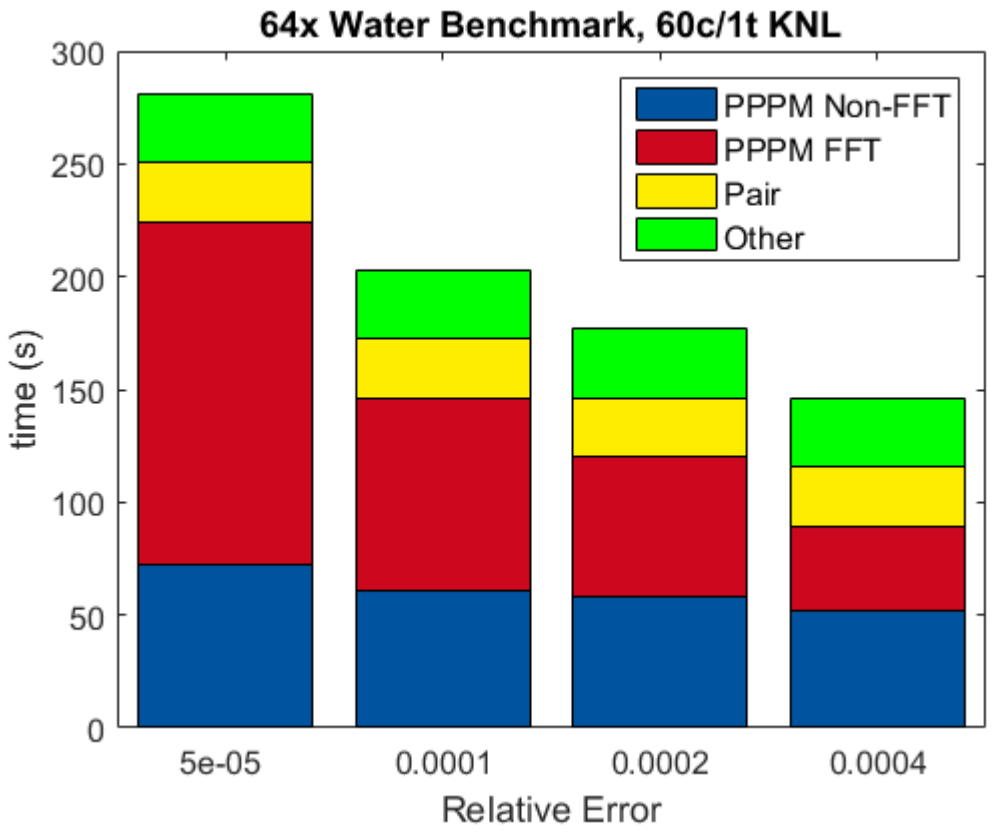


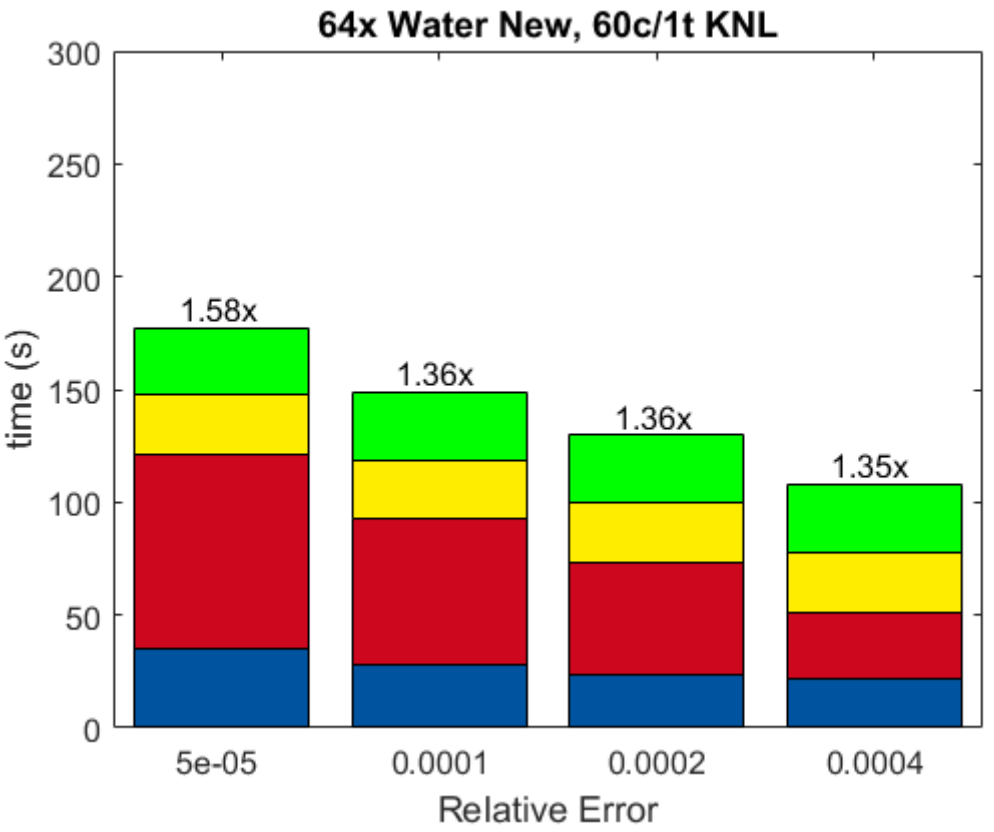
Overall Speedup - Parallel





Overall Speedup - Parallel





Overall Speedup - Parallel

Speedups on 60 cores are about 1.5x for small problems for the fastest overall parameters.

PPPM routines are still sped up by about 2x for cases with good parameters

Optimal parameters depend on problem size

Future Work

- Study 1D vs 3D FFT MKL calls
- Adjust optimizations for alternative differentation mode
- Apply optimizations to P3M for Lennard-Jones (dispersion)