A case study in porting a production scientific supercomputing application to a reconfigurable computer

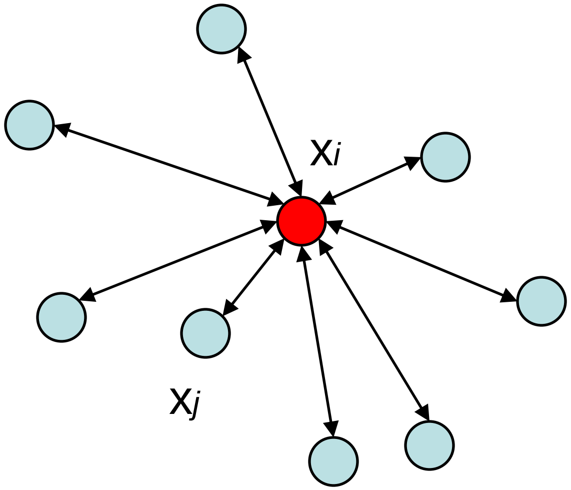
Volodymyr Kindratenko
National Center for Supercomputing Applications (NCSA)
University of Illinois at Urbana-Champaign (UIUC)
kindr@ncsa.uiuc.edu

Presentation outline

- **Molecular dynamics simulation**
  - Basics
  - Optimizations
- **NAMD**
  - ‘Official benchmark’ kernel
  - Benchmark dataset
- **NAMD on SRC-6**
  - Trivially parallel implementation
  - Other approaches
- **Conclusions**

Molecular dynamics simulation

- **Basic principles**
  - each atom is treated as a point mass
  - simple force rules describe the interactions between atoms
  - Newton's equations are integrated to move the atoms

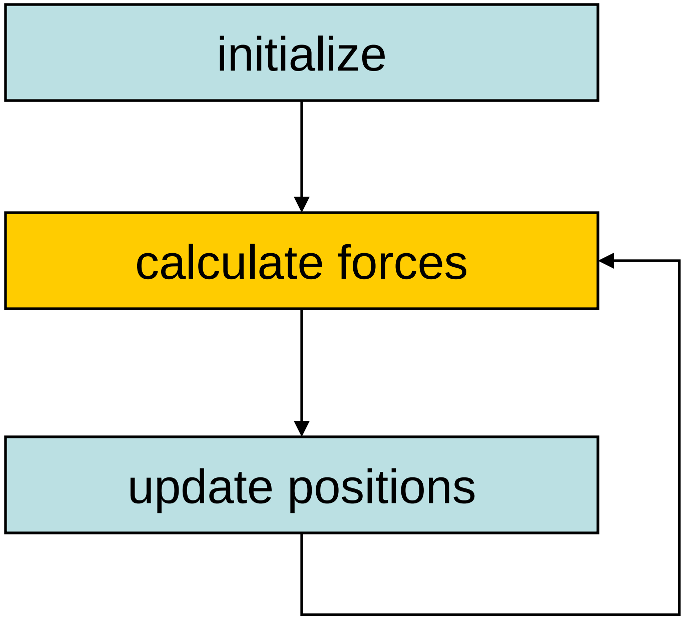


$$F(x_i) := \sum_{i \neq j=1}^{N} f(x_i, x_j)$$

$$F(x_i) = m_i \frac{d^2 x_i}{dt^2}$$

Molecular dynamics simulation

- **Basic algorithm**



$$x_i^k$$

time step

atom index

$$F(x_i^k) := \sum_{i \neq j=1}^{N} f(x_i^k, x_j^k)$$

$$x_i^{k+1} = x_i^k + \mathit{f}(F(x_i^k))$$

Common potential energy function

- $U_{total} = U_{bond} + U_{angle} + U_{dihedral} + U_{vdW} + U_{Coulomb}$
- **Bonded interactions**
  - U_{bond} stretching, U_{angle} bending, and $U_{dihedral}$ torsional
- **Interactions between nonbonded atom pairs**
  - van der Waal’s forces (approximated by a Lennard–Jones 6–12 potential) and

$$U_{\mathrm{vdW}} = \sum_{i} \sum_{j>i} 4\varepsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right]$$

  - electrostatic interactions

$$U_{\mathrm{Coulomb}} = \sum_{i} \sum_{j>i} \frac{q_i q_j}{4\pi\varepsilon_0 r_{ij}}$$

Molecular dynamics simulation

- **Computational complexity**
  - Assuming N atoms
    - computational time is $O(N^2)$

```
for (i=0; i < N; i++)
   for (j=0; j < N; j++)
      $F(i) += f(x_i,x_j)$
```

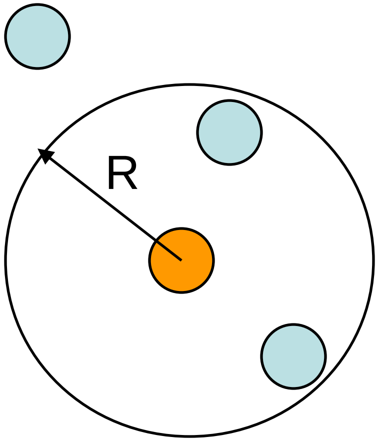
Molecular dynamics simulation

- **Optimizations**
  - Newton's Third Law of Motion $f(x_i, x_j) = -f(x_j, x_i)$
  - Assuming N atoms
    - computational time is O((N-1)N/2)

```
for (i=0; i < N-1; i++)
   for (j=i+1; j < N; j++)
      $F(i) += f(x_i,x_j)$
      $F(j) -= f(x_i,x_j)$
```

Molecular dynamics simulation

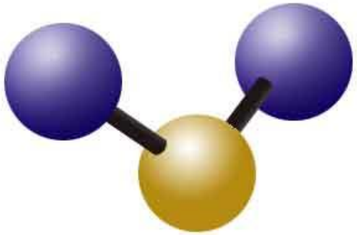
- **Optimizations**
  - – Cutoff radius
    - computational time is O(NR^3)



```
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
        if (distance(i,j) < cutoff distance)
            $F(i) += f(x_i,x_j)$
```

Molecular dynamics simulation

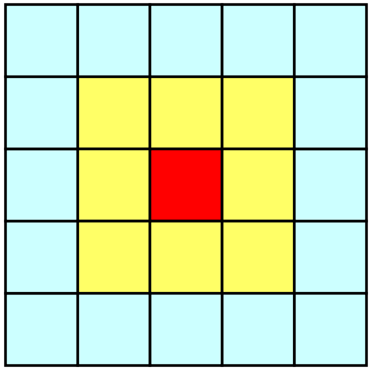
- **Optimizations**
  - Bonded atoms $|x_i^k - x_j^k| = d_{ij}$
    - computational time is O(NM)



```
for (i=0; i < N; i++)
    exclude bonded atoms
    for (j=0; j < M; j++)
        F(i) += f(x_i,x_j)
```

Molecular dynamics simulation

- **Optimizations**
  - Spatial decomposition into M patches
  - Assuming each patch has *N* atoms
    - computational time is $O(14MN^2)$



```
for (k=0; k=M; k++)
    for (i=0; i < N; i++)
        for (j=0; j < N; j++)
            F(i) += f(x_i,x_j)
```

```
for (k=0; k=M; k++)
    for (n=0; n=13; n++)
        for (i=0; i < N; i++)
            for (j=0; j < N; j++)
                F(i) += f(x_i,x_j)
```

Molecular dynamics simulation

- **Optimizations**
  - Lennard–Jones 6–12 potential implemented via a lookup table interpolation
  - The particle-mesh Ewald (PME) method is used to compute electrostatic interactions

Molecular dynamics simulation

- **Are all these optimizations really necessary?**
  - Yes!
  - Instead of O(N^2) complexity, one typically ends up with the O(~N) complexity
    - depends on the dataset
  - A numerical example:
    - 92K atoms, single iteration step

optimization	compute time
spatial decomposition	**~800 seconds**
+ cutoff radius	**~9 seconds**
+ bonded atoms exclusion	**~7 seconds**

NAMD

- **“Official benchmark” kernel**
  - implements all the above optimization techniques
  - applies other code optimization techniques
  - double precision floating point for everything
- **Simplified kernel**
  - Implements everything but bonded atoms
  - Single precision floating point for atom locations and 32 bit integer for forces

NAMD source code

```
void ComputeList::runComputes(PatchList *patchList)
{
    int i;

    for ( i=0; i<numSelfComputes; ++i ) {
        selfComputes[i].doWork(patchList);
    }

    for ( i=0; i<numPairComputes; ++i ) {
        pairComputes[i].doWork(patchList);
    }
}
```

NAMD source code

```
void SelfCompute::doWork(PatchList *patchList)
{
    Patch *p1 = &(patchList->patches[patchId]);
    int doEnergy = patchList->doEnergy;
    nonbonded params;
    params.p[0] = p1->atoms;
    params.p[1] = p1->atoms;
    params.ff[0] = p1->f_nbond;
    params.ff[1] = p1->f_nbond;
    params.numAtoms[0] = p1->numAtoms;
    params.numAtoms[1] = p1->numAtoms;
    params.reduction = patchList->reductionData;
    params.pressureProfileReduction = 0;

    params.minPart = 0; // minPart;
    params.maxPart = 1; // maxPart;
    params.numParts = 1; // numParts;

    calc_both(&params,1);
}
```

```
void PairCompute::doWork(PatchList *patchList)
{
    Patch *p1 = &(patchList->patches[patchId1]);
    Patch *p2 = &(patchList->patches[patchId2]);
    int doEnergy = patchList->doEnergy;
    nonbonded params;
    params.p[0] = p1->image(image1,patchList->lattice);
    params.p[1] = p2->image(image2,patchList->lattice);
    params.ff[0] = p1->f_nbond;
    params.ff[1] = p2->f_nbond;
    params.numAtoms[0] = p1->numAtoms;
    params.numAtoms[1] = p2->numAtoms;
    params.reduction = patchList->reductionData;
    params.pressureProfileReduction = 0;

    params.minPart = 0; // minPart;
    params.maxPart = 1; // maxPart;
    params.numParts = 1; // numParts;

    calc_both(&params,0);
}
```

NAMD source code

```
void ComputeNonbondedUtil::calc_both( nonbonded *params, int doSelf )
{
   // Bringing stuff into local namespace for speed.
   register const BigReal cutoff2 = ComputeNonbondedUtil:: cutoff2;
   const BigReal dielectric_1 = ComputeNonbondedUtil:: dielectric_1;
   const float* const ljTable = ComputeNonbondedUtil:: ljTable;
   const int ljTable_dim = ComputeNonbondedUtil:: ljTable_dim;

   const BigReal* const table_four = ComputeNonbondedUtil:: table_short;

   const BigReal r2_delta = ComputeNonbondedUtil:: r2_delta;
   const int r2_delta_exp = ComputeNonbondedUtil:: r2_delta_exp;
   const int r2_delta_expc = 64 * (r2_delta_exp - 127);

   const int i_upper = params->numAtoms[0] - ( doSelf ? 1 : 0 );
   const int j_upper = params->numAtoms[1];

   const CompAtom *p_0 = params->p[0];
   const CompAtom *p_1 = params->p[1];  // same as p_0 if doSelf

   Force *f_0 = params->ff[0];
   Force *f_1 = params->ff[1];  // same as f_0 if doSelf
```

NAMD source code

```
for ( int i = 0; i < i_upper; ++i )
{
    const CompAtom &p_i = p_0[i];
    register const BigReal p_i_x = p_i.position.x;
    register const BigReal p_i_y = p_i.position.y;
    register const BigReal p_i_z = p_i.position.z;

    Force & f_i = f_0[i];

    const BigReal kq_i = COLOUMB * p_i.charge * dielectric_1;

    const float* const lj_row = ljTable + 2 * ljTable_dim * p_i.vdw_type;

    // INNER LOOP GOES HERE

 } // for i
```

NAMD source code

```
// INNER LOOP
for ( int j = ( doSelf ? i+1 : 0 ); j < j_upper; ++j)
{
    register const CompAtom *p_j = p_1 + j;

    register const float p_ij_x = p_i_x - p_j->position.x;
    register const float p_ij_y = p_i_y - p_j->position.y;
    register const float p_ij_z = p_i_z - p_j->position.z;
    register float r2 = p_ij_x*p_ij_x + p_ij_y*p_ij_y + p_ij_z*p_ij_z;

    if ( r2 > cutoff2 ) continue;

    float kqq = kq_i * p_j->charge;

    const float A = lj_row[2*p_j->vdw_type];
    const float B = lj_row[2*p_j->vdw_type+1];

    union { float f; int32 i; } r2f;
    r2f.f = r2;
    const int table_i = (r2f.i >> 17) + r2_delta_expc;
    r2f.i &= 0xfffe0000;
    const float diffa = r2 - r2f.f;

    const BigReal* const vdwa_i = table_four + 16*table_i;
    const BigReal* const vdwb_i = table_four + 16*table_i + 4;
    const BigReal* const fast_i = table_four + 16*table_i + 8;

    float fast_d = kqq * fast_i[3] + A * vdwa_i[3] - B * vdwb_i[3];
    float fast_c = kqq * fast_i[2] + A * vdwa_i[2] - B * vdwb_i[2];
    float fast_b = kqq * fast_i[1] + A * vdwa_i[1] - B * vdwb_i[1];

    register float fast_dir = ( 3.0 * diffa * fast_d + 2.0 * fast_c ) * diffa
       + fast_b;
    float force_r = -2.0 * fast_dir;
    if ( force_r > 100.0 ) force_r = 100.0;
    force_r *= IVBIAS;

    Force & f_j = f_1[j];

    register int32 tmp_x = (int32)floor(0.5 + force_r * p_ij_x);
    f_i.x += tmp_x;
    f_j.x -= tmp_x;
    register int32 tmp_y = (int32)floor(0.5 + force_r * p_ij_y);
    f_i.y += tmp_y;
    f_j.y -= tmp_y;
    register int32 tmp_z = (int32)floor(0.5 + force_r * p_ij_z);
    f_i.z += tmp_z;
    f_j.z -= tmp_z;
}
```

NAMD benchmark dataset

- **92224 atoms**
- **144 patches**
  - between 500 and 700 atoms per patch

- **numSelfComputes = 144**
- **numPairComputes = 144*13=1872**
- **calc_both() is called 144+1872=2016 times**
- **accumulated compute time is ~9.28 seconds**
  - SRC host workstation
    - Dual Xeon 2.8 GHz, 1 GB mem

NAND on SRC MAP

- **What needs to be done in order to port NAMD to SRC MAP?**
  - All data structures need to be converted to flat arrays
    - lookup tables
    - input data (atom position, etc.)
    - output data (forces)
  - The code that goes to FPGA should be outsourced to a separate function that works with these flat arrays

NAND on SRC: lookup tables

```
static float* __lj_pars;
static float* __table_four;
static int __n, __m;

// allocate memory, first time only
if (__firsttime)
{
   __in_data = (struct FPGA_input_data *)malloc(mol-
   >numAtoms*sizeof(struct FPGA_input_data));
   __out_data = (struct FPGA_output_data
   *)malloc(mol->numAtoms*sizeof(struct
   FPGA_output_data));

   union { float f; int32 i; } r2f0;
   r2f0.f = cutoff2;
   __n = (r2f0.i >> 17) + r2_delta_expc + 1;

   __table_four =
   (float*)malloc((12*__n)*sizeof(float));

   for (k = 0; k < __n; k++)
      for (j = 0; j < 12; j++)
      {
         __table_four[12*k+j] = table_four[16*k+j];
      }
```

```
   __m = ljTable_dim;

   __lj_pars =
     (float*)malloc((2*__m*__m)*sizeof(float));

   for (k = 0; k < __m; k++)
   {
         const float* const tmp_lj_row = ljTable + 2 *
     ljTable_dim * k;

        for (j = 0; j < __m; j++)
        {
           __lj_pars[2*k*__m+2*j] = tmp_lj_row[2*j];
           __lj_pars[2*k*__m+2*j+1] = tmp_lj_row[2*j+1];
        }
    }
}
```

NAND on SRC: data in

```
// prepare data to go into FPGA
for (i = 0; i < i_upper; i++)
{
    const CompAtom &p_i = p_0[i];
    __in_data[i].p_x = p_i.position.x;
    __in_data[i].p_y = p_i.position.y;
    __in_data[i].p_z = p_i.position.z;
    __in_data[i].p_charge = p_i.charge;

    __in_data[i].p_atomVdwType =
   p_i.vdw_type;

    Force &f_i = f_0[i];
    __in_data[i].f_x = f_i.x;
    __in_data[i].f_y = f_i.y;
    __in_data[i].f_z = f_i.z;
}
```

```
if (!doSelf)
{
    for (j = 0; j < j_upper; j++)
    {
        const CompAtom &p_j = p_1[j];
        __in_data[j+i_upper].p_x =
   p_j.position.x;
        __in_data[j+i_upper].p_y =
   p_j.position.y;
        __in_data[j+i_upper].p_z =
   p_j.position.z;
        __in_data[j+i_upper].p_charge =
   p_j.charge;

        __in_data[j+i_upper].p_atomVdwType =
   p_j.vdw_type;
        Force &f_j = f_1[j];
        __in_data[j+i_upper].f_x = f_j.x;
        __in_data[j+i_upper].f_y = f_j.y;
        __in_data[j+i_upper].f_z = f_j.z;
    }
}
```

NAMD on SRC: data out

```
// get results back
for (i = 0; i < i_upper; i++)
{
    Force &f_i = f_0[i];
    f_i.x = __out_data[i].f_x;
    f_i.y = __out_data[i].f_y;
    f_i.z = __out_data[i].f_z;
}

if (!doSelf)
{
    for (j = 0; j < j_upper; j++)
    {
        Force &f_j = f_1[j];
        f_j.x = __out_data[j+i_upper].f_x;
        f_j.y = __out_data[j+i_upper].f_y;
        f_j.z = __out_data[j+i_upper].f_z;
    }
}
```

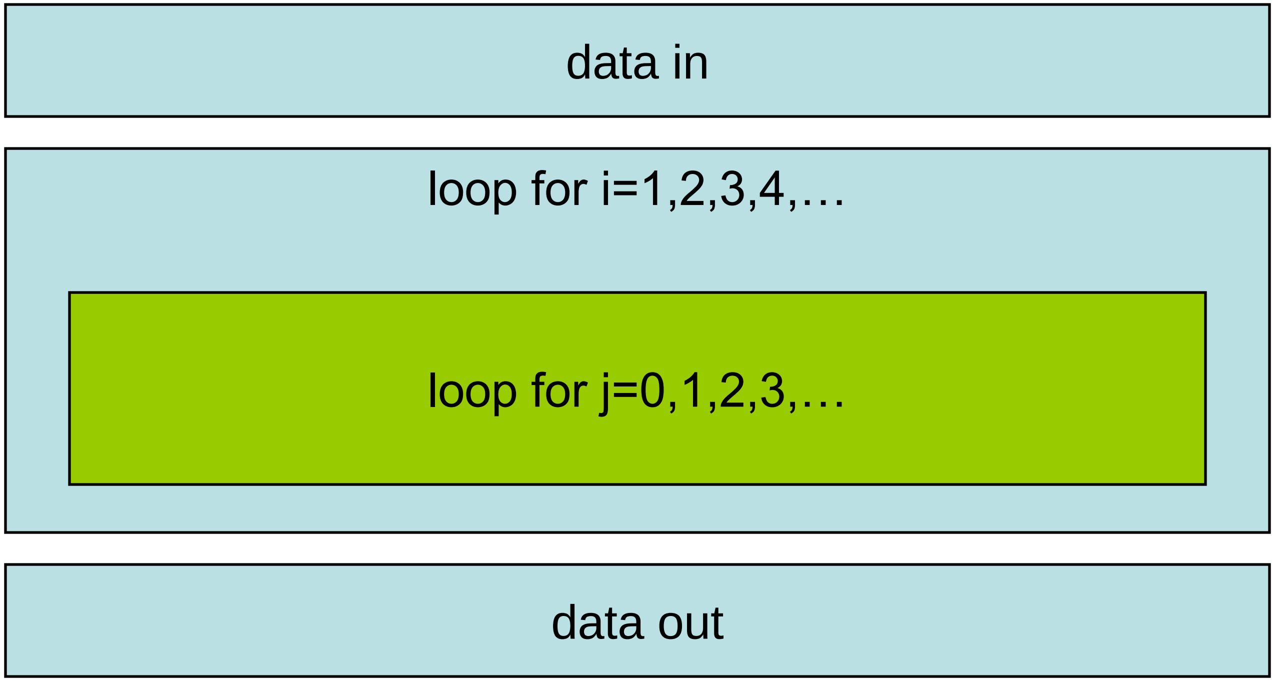
```
// MAP I/O data structures
struct FPGA_input_data
{
    float p_x;
    float p_y;
    float p_z;
    float p_charge;
    int p_atomVdwType;
    int f_x;
    int f_y;
    int f_z;
};

struct FPGA_output_data
{
    int nothing;
    int f_x;
    int f_y;
    int f_z;
};
```

NAMD on SRC: MAP function

```
void map_compute_func(
        int i_upper, int j_upper,
        int64_t in_data[],
        int64_t out_data[],
        int m, int64_t lj_pars[],
        int n, int64_t table_four[],
        int doSelf, float dielectric_1,
        float cutoff2, int r2_delta_expc,
        int firsttime, int mapnum);
```

NAMD MAP code algorithm

- **Initial implementation**



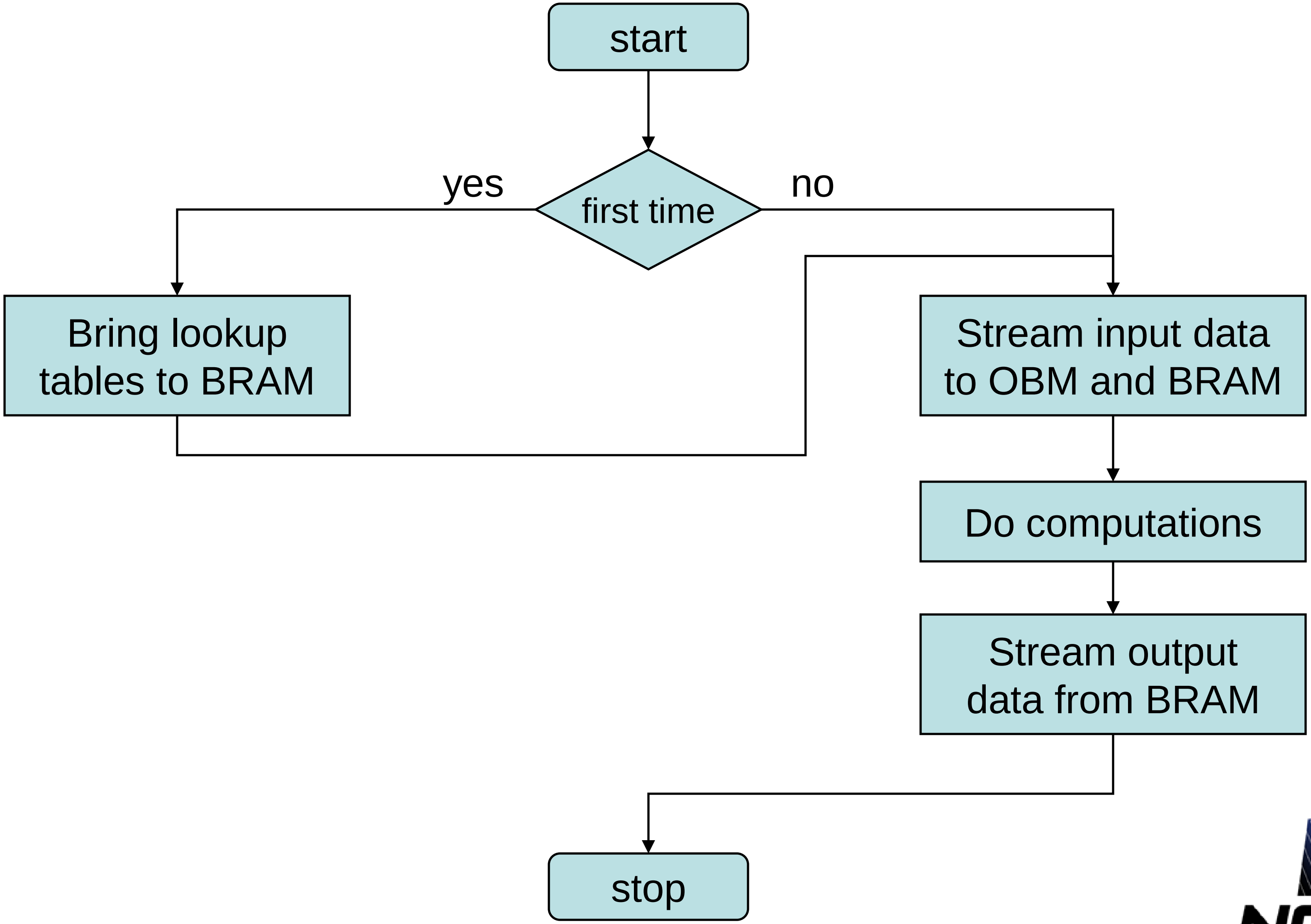
data in

loop for i=1,2,3,4,…

loop for j=0,1,2,3,…

data out

NAMD MAP code algorithm


start
yes
first time
no
Bring lookup tables to BRAM
Stream input data to OBM and BRAM
Do computations
Stream output data from BRAM
stop

NAMD MAP C source code

```
if (firsttime == 1)
{
    // DMA table_four in and copy to 12 internal tables
    DMA_CPU (CM2OBM, AL, MAP_OBM_stripe(1,"A,B,C,D,E,F"), table_four, 1, n*48, 0);  // n*12*sizeof(float)
    wait_DMA(0);

    for (i = 0; i < n; i++)
    {
        split_64to32_flt_flt(AL[i], &tmp_a, &tmp_b);
        c02[i] = tmp_a;
        split_64to32_flt_flt(BL[i], &tmp_a, &tmp_b);
        c03[i] = tmp_b;
        c04[i] = tmp_a;
        split_64to32_flt_flt(CL[i], &tmp_a, &tmp_b);
        c06[i] = tmp_a;
        split_64to32_flt_flt(DL[i], &tmp_a, &tmp_b);
        c07[i] = tmp_b;
        c08[i] = tmp_a;
        split_64to32_flt_flt(EL[i], &tmp_a, &tmp_b);
        c10[i] = tmp_a;
        split_64to32_flt_flt(FL[i], &tmp_a, &tmp_b);
        c11[i] = tmp_b;
        c12[i] = tmp_a;
    }

    // DMA lj_pars in and copy to 2 internal tables
    m2 = m*m;
    DMA_CPU (CM2OBM, FL, MAP_OBM_stripe(1,"F"), lj_pars, 1, (m2*8), 0); // m*m*2*sizeof(float)
    wait_DMA(0);

    for (i = 0; i < m2; i++)
    {
        split_64to32_flt_flt(FL[i], &tmp_b, &tmp_a);
        lj_pars_A[i] = tmp_a;
        lj_pars_B[i] = tmp_b;
    }
}
```

NAMD MAP C source code

```
// how much data to transfer in?
selector_32(doSelf, i_upper, (i_upper + j_upper),
   &size_in);

// DMA data in
#pragma src parallel sections
{
   #pragma src section
   {
      stream_dma_cpu_dual(&S0, &S1,
   PORT_TO_STREAM, EL, DMA_E_F, in_data, 1,
   size_in*32);
   }
```

```
   #pragma src section
   {
      for (k = 0; k < size_in*2; k++)   {
         cg_count_ceil_32 (1, 0, k == 0, 1, &j);
         cg_count_ceil_32 (j==0, 0, k == 0,
   0xffffffff, &i);

         get_stream(&S0, &v0);
         get_stream(&S1, &v1);

         if (j == 0)  {
            AL[i] = v0;
            BL[i] = v1;
         }
         else {
            cl_bram[i] = v0;
            dl_bram[i] = v1;
            cl_bram2[i] = v0;
            dl_bram2[i] = v1;
         }
      }
   }
}
```

NAMD MAP C source code

```
selector_32(doSelf, (i_upper - 1), i_upper, &i_to);
selector_32(doSelf, i_upper, (i_upper + j_upper), &j_to);

for (i = 0; i < i_to; i++)
{
        // read data for the current computation
        split_64to32_flt_flt(AL[i], &in_p_i_y, &in_p_i_x);
        split_64to32_flt_flt(BL[i], &in_p_i_charge,
        &in_p_i_z);
        split_64to32(cl_bram2[i], &in_f_i_x,
        &in_p_i_atomVdwType);
        split_64to32(dl_bram2[i], &in_f_i_z, &in_f_i_y);

        kq_i = in_p_i_charge * dielectric_1; // * COLOUMB;

        selector_32(doSelf, (i + 1), i_upper, &j_from);
```

```
#pragma loop noloop_dep
for (j = j_from; j < j_to; j++)
{
    // read data for the current computation
    split_64to32_flt_flt(AL[j], &in_p_j_y, &in_p_j_x);
    split_64to32_flt_flt(BL[j], &in_p_j_charge,
&in_p_j_z);
    split_64to32(cl_bram[j], &in_f_j_x,
&in_p_j_atomVdwType);
    split_64to32(dl_bram[j], &in_f_j_z, &in_f_j_y);

    // compute distance
    p_ij_x = in_p_i_x - in_p_j_x;
    p_ij_y = in_p_i_y - in_p_j_y;
    p_ij_z = in_p_i_z - in_p_j_z;
    r2 = p_ij_x * p_ij_x + p_ij_y * p_ij_y + p_ij_z *
p_ij_z;
```

NAMD MAP C source code

```
if (r2 <= cutoff2)
{
    kqq = kq_i * in_p_j_charge;

    indx_ij = in_p_i_atomVdwType *
                 m+in_p_j_atomVdwType;
    A = lj_pars_A[indx_ij];
    B = lj_pars_B[indx_ij];

    // union { float f; int32 i; } r2f; r2f.f = r2;
    comb_32to64_flt_flt(r2, 0, &r2f);
    split_64to32_int_flt(r2f, &r2f_i, &r2f_f);
    table_i = (r2f_i >> 17) + r2_delta_expc;
    r2f_i &= 0xfffe0000;
    comb_32to64_int_flt(r2f_i, 0, &r2f);
    split_64to32_flt_flt(r2f, &r2f_f, &tmp_a);

    diffa = r2 - r2f_f;

    fast_d = kqq * c12[table_i] + A * c04[table_i] - B *
      c08[table_i];
    fast_c = kqq * c11[table_i] + A * c03[table_i] - B *
      c07[table_i];
    fast_b = kqq * c10[table_i] + A * c02[table_i] - B *
      c06[table_i];

    force_r = -2 * (( 3 * diffa * fast_d + 2 * fast_c ) * diffa +
      fast_b);

    if (force_r > 100)
        force_r = 100.0;

    force_r *= IVBIAS;

    tmp_x = (int32_t)(0.5 + force_r * p_ij_x);
    tmp_y = (int32_t)(0.5 + force_r * p_ij_y);
    tmp_z = (int32_t)(0.5 + force_r * p_ij_z);
}
else
{
    tmp_x = 0;
    tmp_y = 0;
    tmp_z = 0;
}
```

NAMD MAP C source code

```
        cg_accum_add_32(tmp_x, 1, in_f_i_x, j == j_from, &out_f_i_x);
        out_f_j_x = in_f_j_x - tmp_x;

        cg_accum_add_32(tmp_y, 1, in_f_i_y, j == j_from, &out_f_i_y);
        out_f_j_y = in_f_j_y - tmp_y;

        cg_accum_add_32(tmp_z, 1, in_f_i_z, j == j_from, &out_f_i_z);
        out_f_j_z = in_f_j_z - tmp_z;

        // store results for the current calculation
        comb_32to64(out_f_j_x, in_p_j_atomVdwType, &tmp_64);
        cl_bram[j] = tmp_64;
        comb_32to64(out_f_j_z, out_f_j_y, &tmp_64);
        dl_bram[j] = tmp_64;
    }

     // store results for the current calculation
    comb_32to64(out_f_i_x, in_p_i_atomVdwType, &tmp_64);
    cl_bram2[i] = tmp_64;
    comb_32to64(out_f_i_z, out_f_i_y, &tmp_64);
    dl_bram2[i] = tmp_64;
}
```

```
// DMA data out
#pragma src parallel sections
{
  #pragma src section
  {
    for (i = 0; i < size_in; i++)  {
      if (doSelf) {
          split_64to32(cl_bram[i], &out_f_j_x,
   &in_p_j_atomVdwType);
          split_64to32(dl_bram[i], &out_f_j_z, &out_f_j_y);

          split_64to32(cl_bram2[i], &out_f_i_x,
   &in_p_i_atomVdwType);
          split_64to32(dl_bram2[i], &out_f_i_z, &out_f_i_y);

          comb_32to64((out_f_i_x+out_f_j_x), 0, &v0);
          comb_32to64((out_f_i_z+out_f_j_z),
   (out_f_i_y+out_f_j_y), &v1);
      }
      else  {
          selector_64(i < i_upper, cl_bram2[i], cl_bram[i], &v0);
          selector_64(i < i_upper, dl_bram2[i], dl_bram[i], &v1);
      }

      put_stream(&S2, v0, 1);
      put_stream(&S3, v1, 1);
    }
  }
  #pragma src section
  {
    stream_dma_cpu_dual(&S2, &S3, STREAM_TO_PORT, EL,
   DMA_E_F, out_data, 1, size_in*16);
  }
}
```

NAMD MAP results

INNER LOOP SUMMARY

loop on line 71:
clocks per iteration: 1
pipeline depth: 10

loop on line 98:
clocks per iteration: 1
pipeline depth: 10

loop on line 119:
clocks per iteration: 1
pipeline depth: 9

loop on line 165:
clocks per iteration: 1
pipeline depth: 159

loop on line 248:
clocks per iteration: 1
pipeline depth: 4

Device Utilization Summary:

Number of BUFGMUXs	1 out of 16	6%
Number of External IOBs	599 out of 1104	54%
Number of LOCed IOBs	599 out of 599	100%
Number of MULT18X18s	85 out of 144	59%
Number of RAMB16s	89 out of 144	61%
Number of SLICEs	29470 out of 33792	87%

Timing analysis: Actual: 10.000ns

Execution time ~12.05 seconds

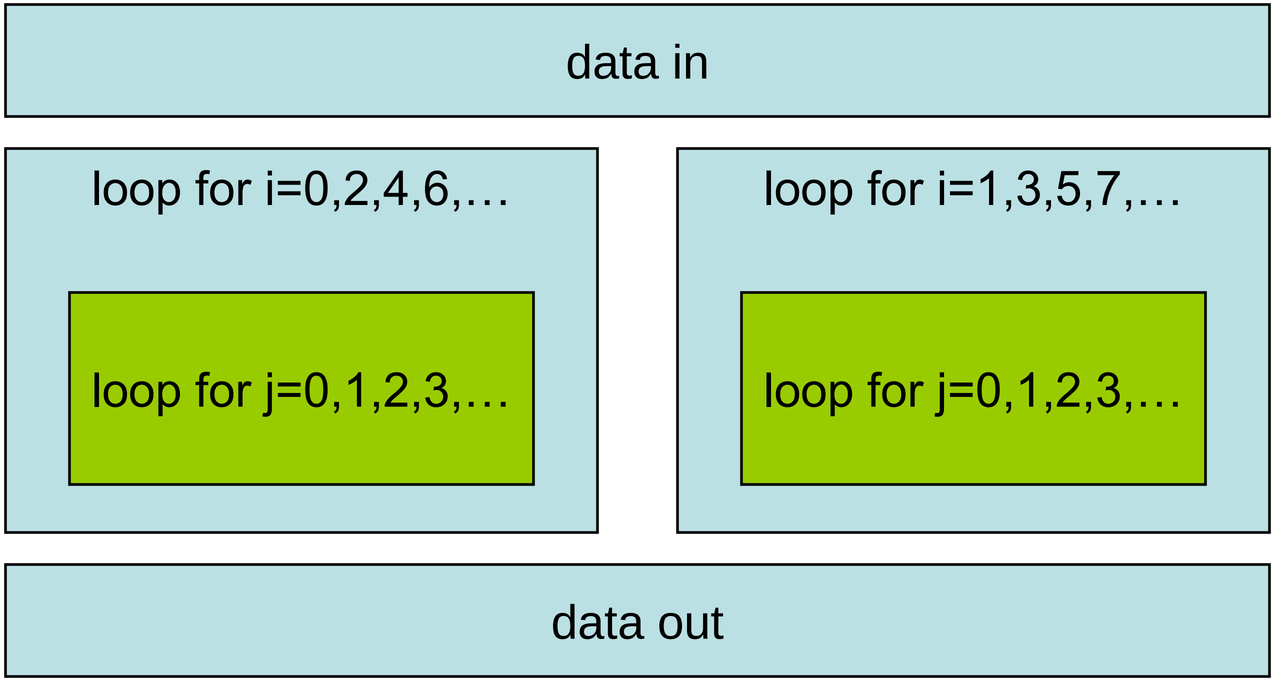
~1.79 seconds due to function call overhead & data DMA in/out and

~10.26 seconds due to actual calculations

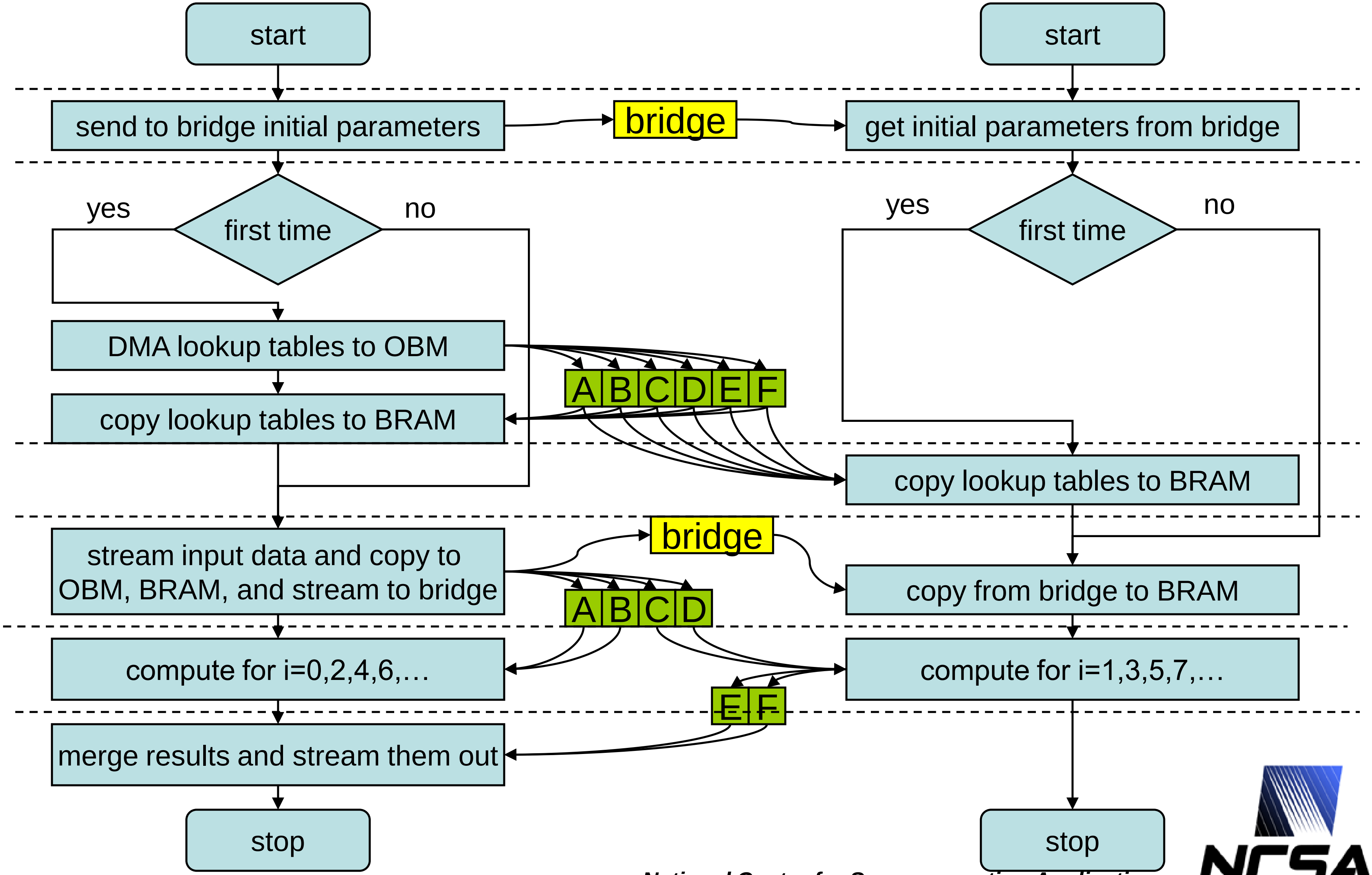
which is ~1.3x slowdown ☹

NAMD MAP code algorithm revised

- **Trivially parallel implementation**



NAMD MAP code algorithm revised



start

send to bridge initial parameters

bridge

get initial parameters from bridge

first time

yes

no

DMA lookup tables to OBM

A B C D E F

copy lookup tables to BRAM

stream input data and copy to OBM, BRAM, and stream to bridge

A B C D

copy from bridge to BRAM

compute for i=0,2,4,6,…

compute for i=1,3,5,7,…

E F

merge results and stream them out

stop

Revised NAMD MAP results

- **Primary chip**

```
INNER LOOP SUMMARY
loop on line 90:
    clocks per iteration:    1
    pipeline depth:         10

loop on line 123:
    clocks per iteration:    1
    pipeline depth:         10

loop on line 146:
    clocks per iteration:    1
    pipeline depth:          9

loop on line 200:
    clocks per iteration:    1
    pipeline depth:        159

loop on line 286:
    clocks per iteration:    1
    pipeline depth:         10
```

- **Secondary chip**

```
INNER LOOP SUMMARY
loop on line 85:
    clocks per iteration:    1
    pipeline depth:         10

loop on line 114:
    clocks per iteration:    1
    pipeline depth:         10

loop on line 138:
    clocks per iteration:    1
    pipeline depth:         10

loop on line 188:
    clocks per iteration:    1
    pipeline depth:        159
```

Revised NAMD MAP results

- Primary chip

Device Utilization Summary:

Number of BUFGMUXs	1 out of 16	6%
Number of External IOBs	805 out of 1104	72%
Number of LOCed IOBs	805 out of 805	100%
Number of MULT18X18s	85 out of 144	59%
Number of RAMB16s	73 out of 144	50%
Number of SLICEs	1239 out of 33792	92%

Timing analysis: Actual: 9.993ns

- Secondary chip

Device Utilization Summary:

Number of BUFGMUXs	1 out of 16	6%
Number of External IOBs	718 out of 1104	65%
Number of LOCed IOBs	718 out of 718	100%
Number of MULT18X18s	88 out of 144	61%
Number of RAMB16s	73 out of 144	50%
Number of SLICEs	26516 out of 33792	78%

Timing analysis: Actual: 9.993ns

Execution time ~6.92 seconds

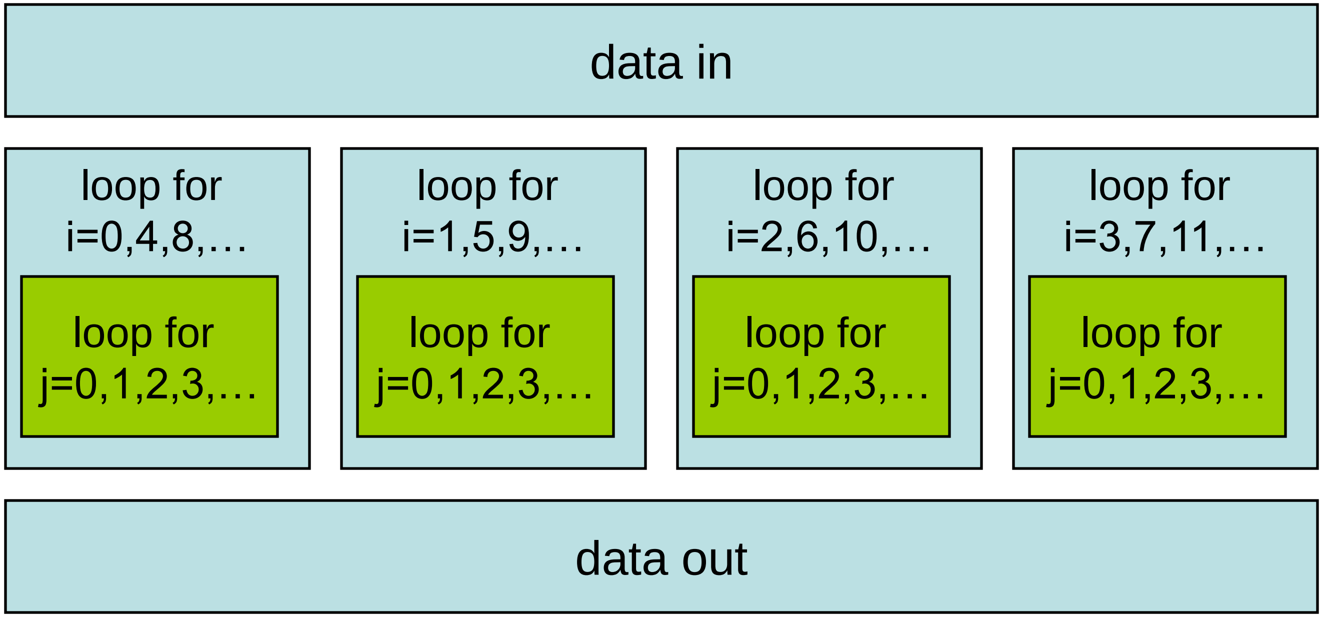
~1.79 seconds due to function call overhead & data DMA in/out and

~5.13 seconds due to actual calculations

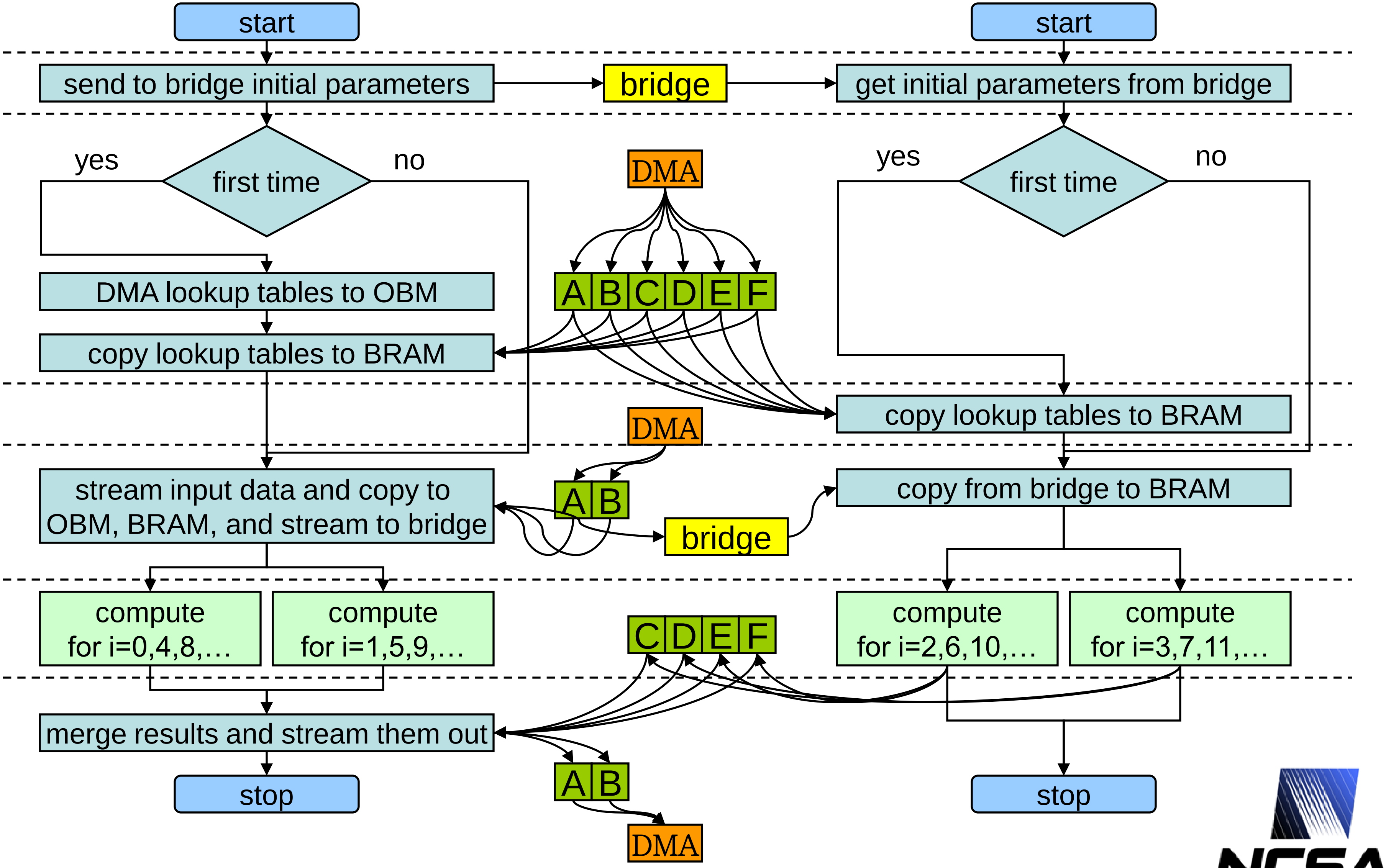
which is ~1.3x speedup ☺

NAMD MAP code algorithm revised

- **Ride on the trivial parallelism**
  - Requires MAP-E (with VP100 FPGAs)
  - Requires SRC_IEEE_V2 multipliers



NAMD MAP code algorithm revised



National Center for Supercomputing Applications

Revised NAMD MAP results

- Primary chip

```
INNER LOOP SUMMARY
loop on line 83:
   clocks per iteration:    1
   pipeline depth:         10

loop on line 133:
   clocks per iteration:    1
   pipeline depth:         10

loop on line 162:
   clocks per iteration:    1
   pipeline depth:          9

loop on line 250:
   clocks per iteration:    1
   pipeline depth:        150

loop on line 364:
   clocks per iteration:    1
   pipeline depth:        150

loop on line 459:
   clocks per iteration:    1
   pipeline depth:         11
```

- Secondary chip

```
INNER LOOP SUMMARY
loop on line 78:
   clocks per iteration:    1
   pipeline depth:         10

loop on line 124:
   clocks per iteration:    1
   pipeline depth:         10

loop on line 153:
   clocks per iteration:    1
   pipeline depth:         10

loop on line 232:
   clocks per iteration:    1
   pipeline depth:        150

loop on line 350:
   clocks per iteration:    1
   pipeline depth:        150
```

Revised NAMD MAP results

- Primary chip

Device Utilization Summary:

Number of BUFGMUXs	1 out of 16	6%
Number of External IOBs	832 out of 1164	71%
Number of LOCed IOBs	832 out of 832	100%
Number of MULT18X18s	125 out of 444	28%
Number of RAMB16s	178 out of 444	40%
Number of SLICEs	42990 out of 44096	97%

Timing analysis: Actual: 9.993ns

- Secondary chip

Device Utilization Summary:

Number of BUFGMUXs	1 out of 16	6%
Number of External IOBs	745 out of 1164	64%
Number of LOCed IOBs	745 out of 745	100%
Number of MULT18X18s	128 out of 444	28%
Number of RAMB16s	178 out of 444	40%
Number of SLICEs	38394 out of 44096	87%

Timing analysis: Actual: 9.944ns

Execution time ~3.57 seconds (measured on CPU)

~0.15 seconds due to data DMA in/out and (measured on MAP)

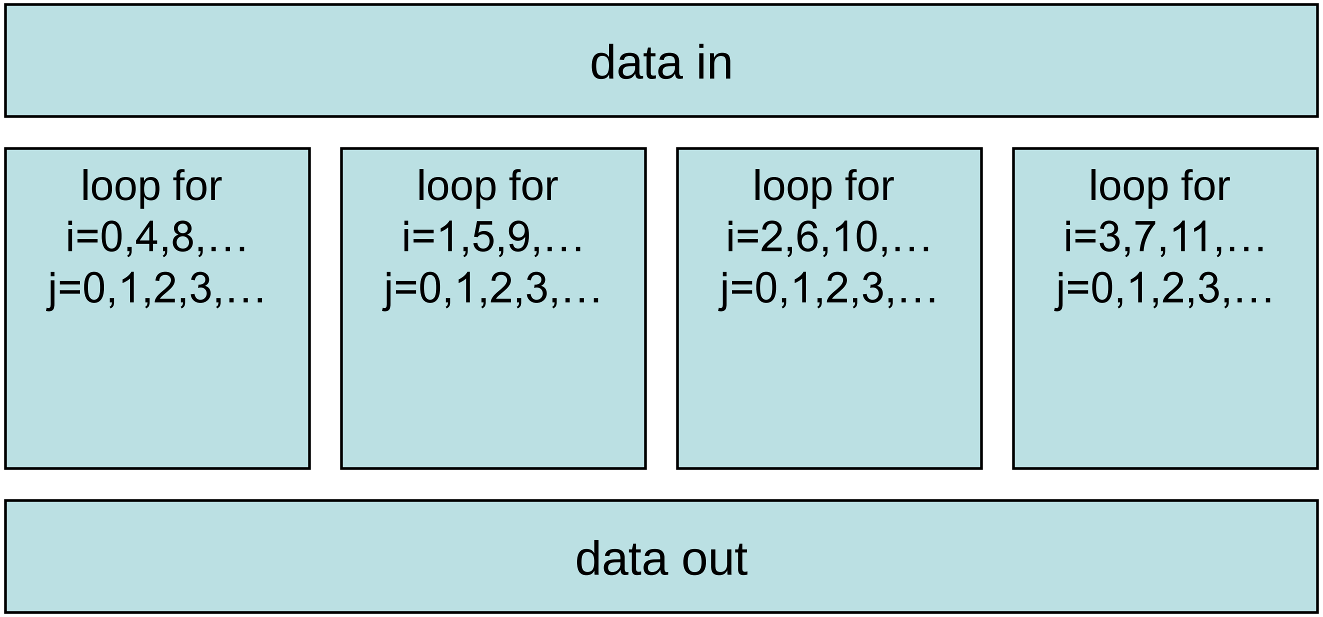
~0.84 seconds due to MAP function call overhead

~2.58 seconds due to actual calculations (measured on MAP)

which is ~2.5x speedup ☺

NAMD MAP code algorithm revised

- **Ride on the trivial parallelism**
  - With nested loops fused



data in

loop for
i=0,4,8,…
j=0,1,2,3,…

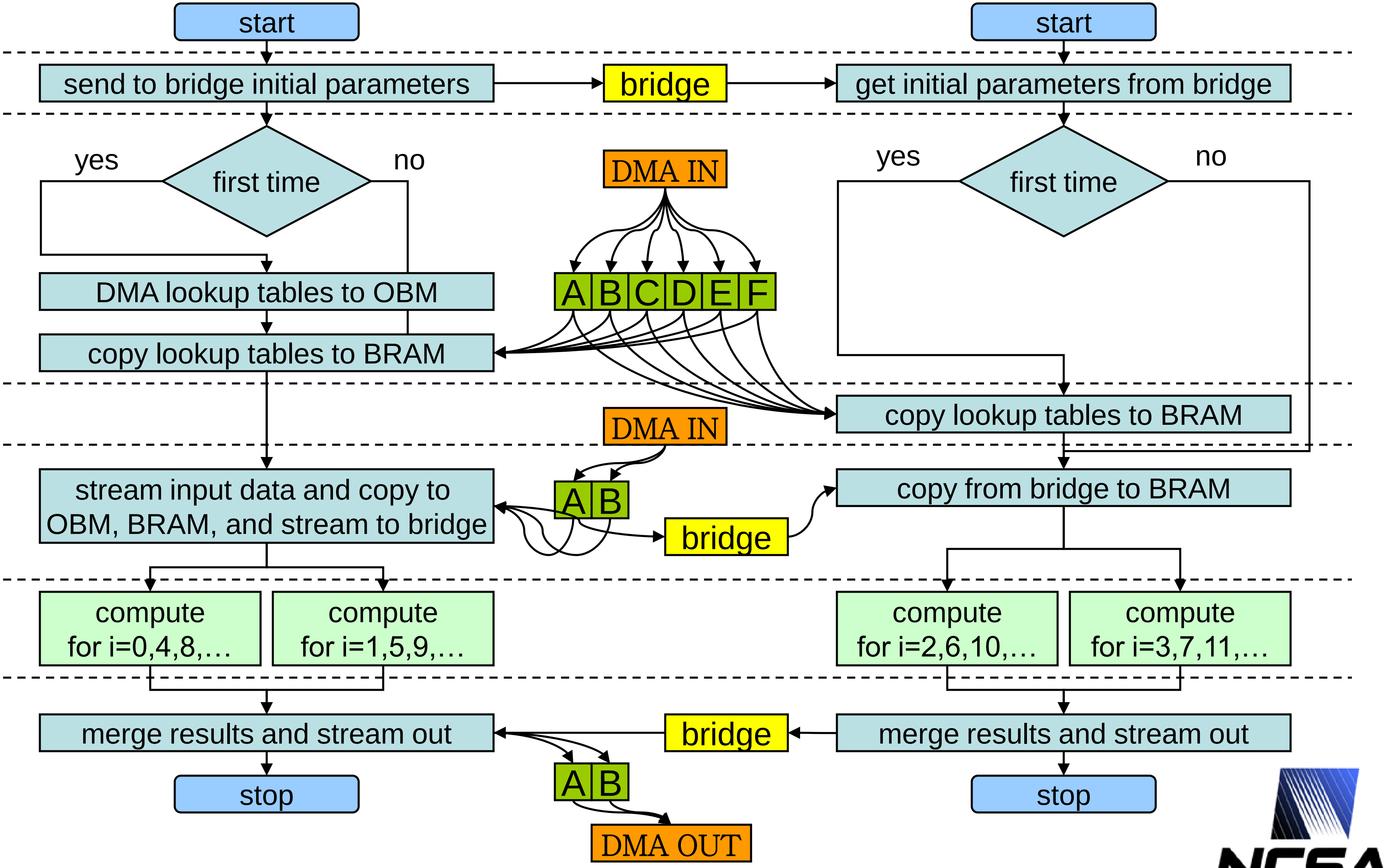
loop for
i=1,5,9,…
j=0,1,2,3,…

loop for
i=2,6,10,…
j=0,1,2,3,…

loop for
i=3,7,11,…
j=0,1,2,3,…

data out

NAMD MAP code algorithm revised


start
start
send to bridge initial parameters
bridge
get initial parameters from bridge
yes
first time
no
DMA IN
yes
first time
no
DMA lookup tables to OBM
A
B
C
D
E
F
copy lookup tables to BRAM
copy lookup tables to BRAM
DMA IN
stream input data and copy to OBM, BRAM, and stream to bridge
A
B
bridge
copy from bridge to BRAM
compute for i=0,4,8,…
compute for i=1,5,9,…
compute for i=2,6,10,…
compute for i=3,7,11,…
merge results and stream out
bridge
merge results and stream out
A
B
stop
stop
DMA OUT

Revised NAMD MAP results

- Primary chip

```
INNER LOOP SUMMARY
loop on line 88:
   clocks per iteration:    1
   pipeline depth:         10

loop on line 138:
   clocks per iteration:    1
   pipeline depth:         10

loop on line 167:
   clocks per iteration:    1
   pipeline depth:          9

loop on line 227:
   clocks per iteration:    1
   pipeline depth:          4

loop on line 239:
   clocks per iteration:    1
   pipeline depth:          4

loop on line 276:
   clocks per iteration:    1
   pipeline depth:        153

loop on line 394:
   clocks per iteration:    1
   pipeline depth:        154

loop on line 523:
   clocks per iteration:    1
   pipeline depth:          7
```

- Secondary chip

```
INNER LOOP SUMMARY
loop on line 84:
   clocks per iteration:    1
   pipeline depth:         10

loop on line 130:
   clocks per iteration:    1
   pipeline depth:         10

loop on line 160:
   clocks per iteration:    1
   pipeline depth:          9

loop on line 207:
   clocks per iteration:    1
   pipeline depth:          4

loop on line 219:
   clocks per iteration:    1
   pipeline depth:          4

loop on line 254:
   clocks per iteration:    1
   pipeline depth:        154

loop on line 372:
   clocks per iteration:    1
   pipeline depth:        154

loop on line 486:
   clocks per iteration:    1
   pipeline depth:          6
```

Revised NAMD MAP results

- Primary chip

Device Utilization Summary:

Number of BUFGMUXs	1 out of 16	6%
Number of External IOBs	832 out of 1164	71%
Number of LOCed IOBs	832 out of 832	100%
Number of MULT18X18s	131 out of 444	29%
Number of RAMB16s	258 out of 444	58%
Number of SLICEs	44094 out of 44096	99%

Timing analysis: Actual: 9.964ns

- Secondary chip

Device Utilization Summary:

Number of BUFGMUXs	1 out of 16	6%
Number of External IOBs	745 out of 1164	64%
Number of LOCed IOBs	745 out of 745	100%
Number of MULT18X18s	134 out of 444	30%
Number of RAMB16s	258 out of 444	58%
Number of SLICEs	40427 out of 44096	91%

Timing analysis: Actual: 9.971ns

Execution time ~3.07 seconds (measured on CPU)

~0.15 seconds due to data DMA in/out and (measured on MAP)

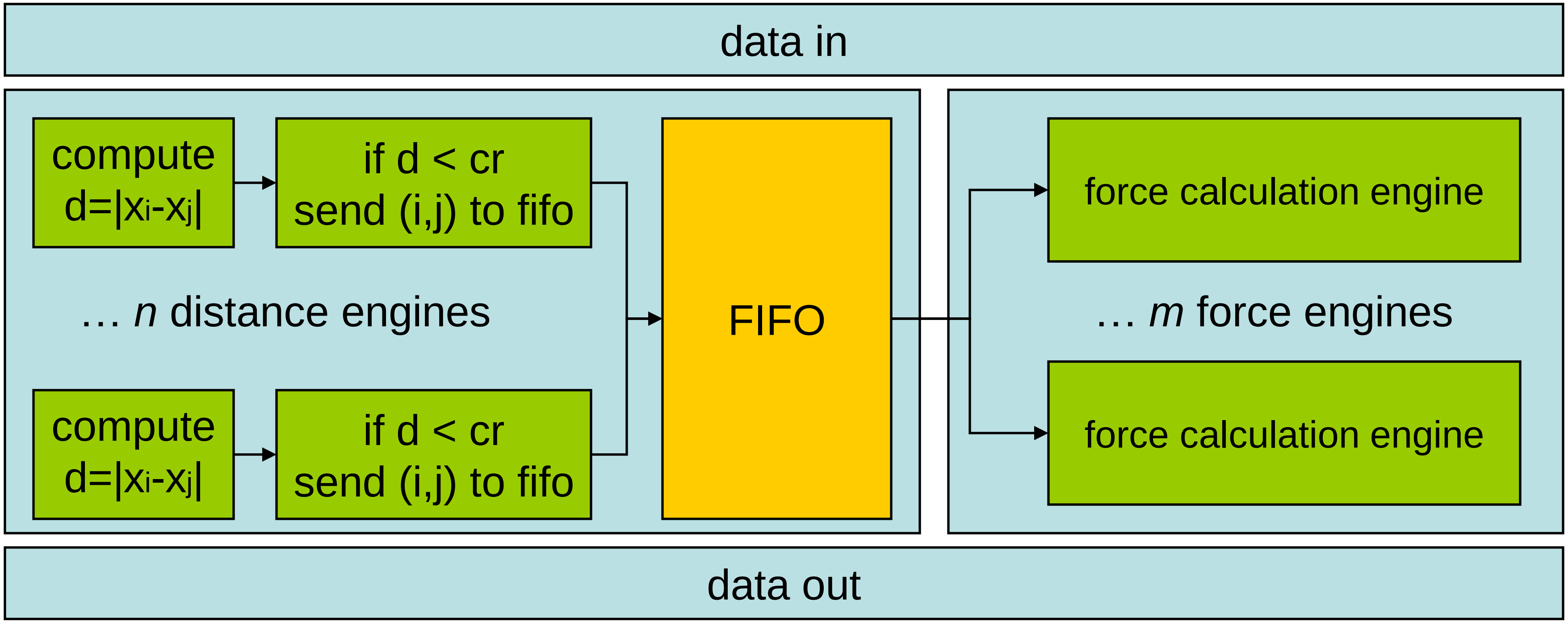
~0.84 seconds due to MAP function call overhead

~2.08 seconds due to actual calculations (measured on MAP)

which is 3x speedup ☺

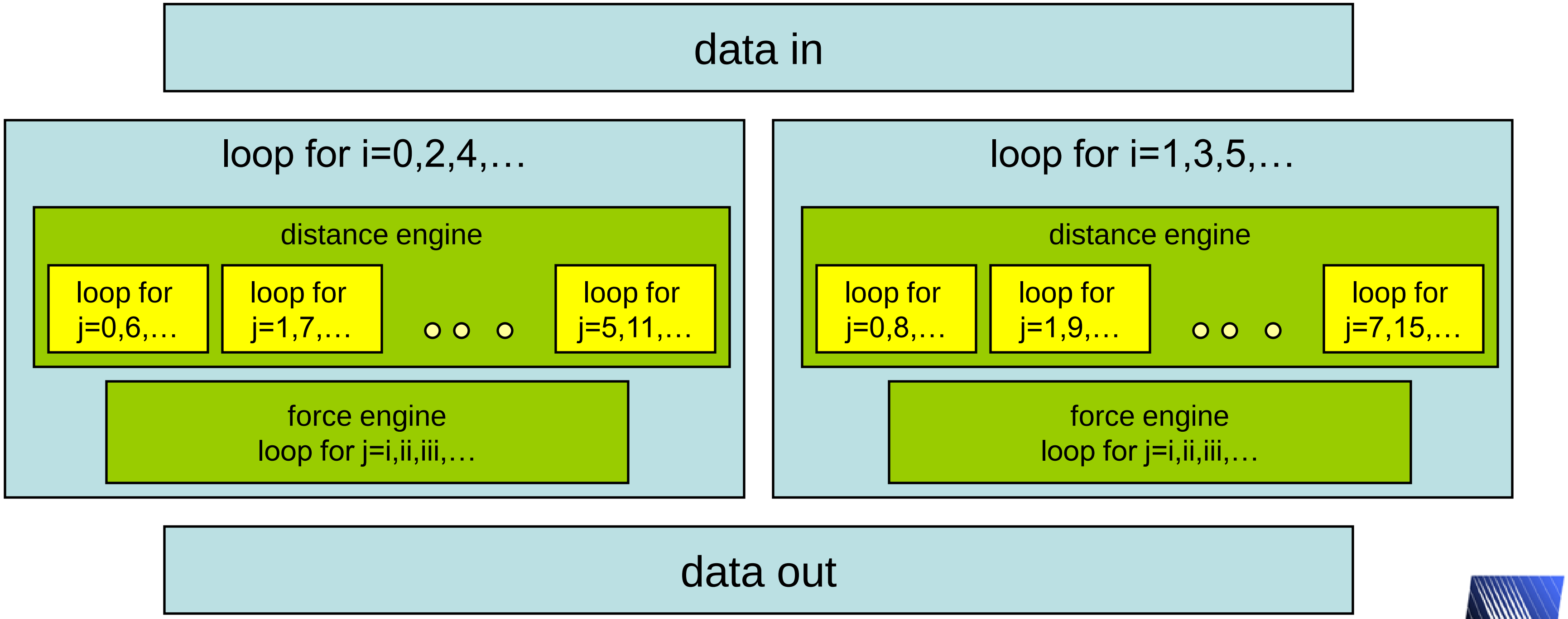
NAMD MAP code algorithm revised

- **De-coupled calculations**
  - Cannot be easily implemented

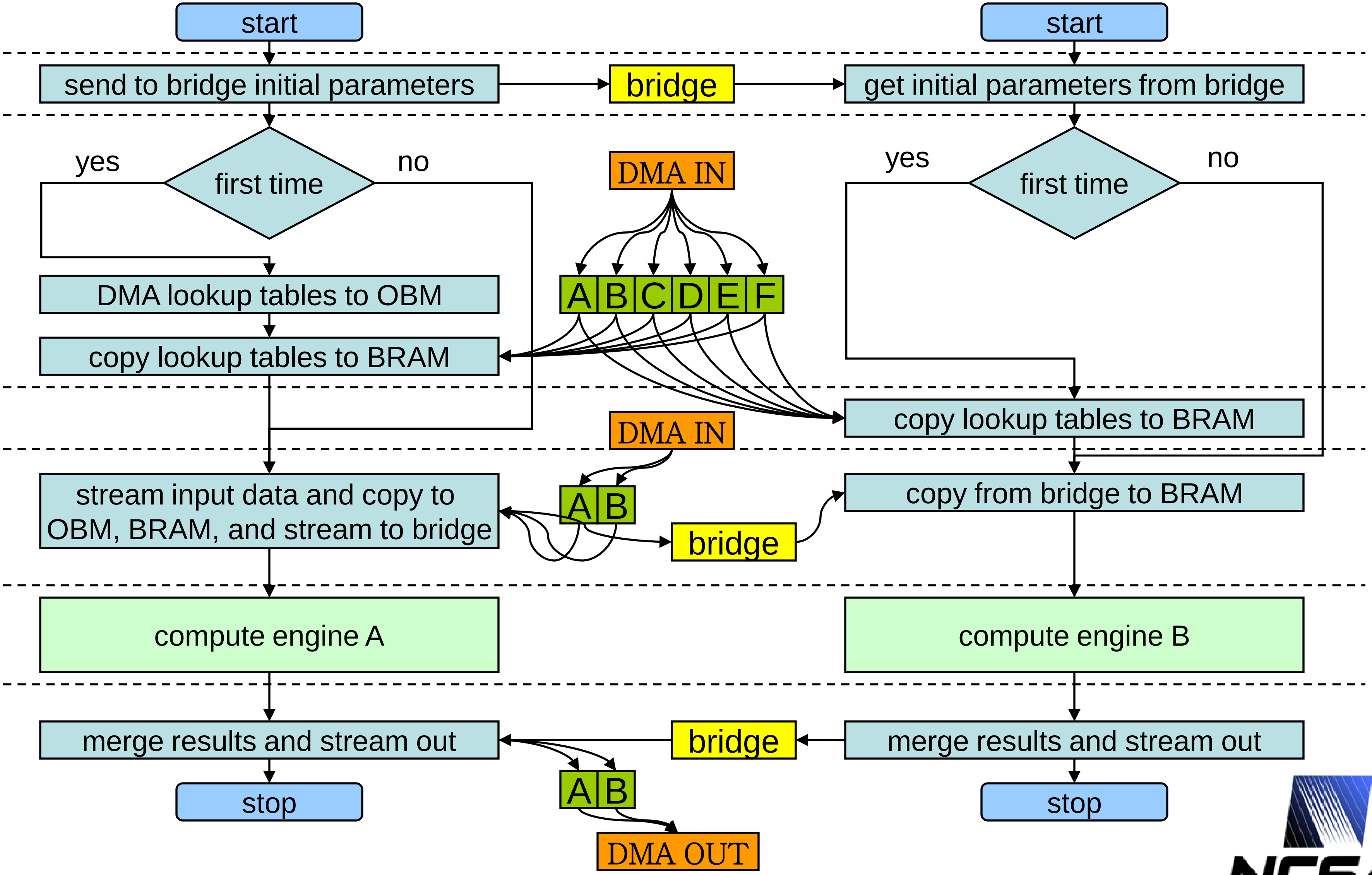


NAMD MAP code algorithm revised

- **De-coupled calculations**
  - But a simpler (and less efficient) version can be implemented

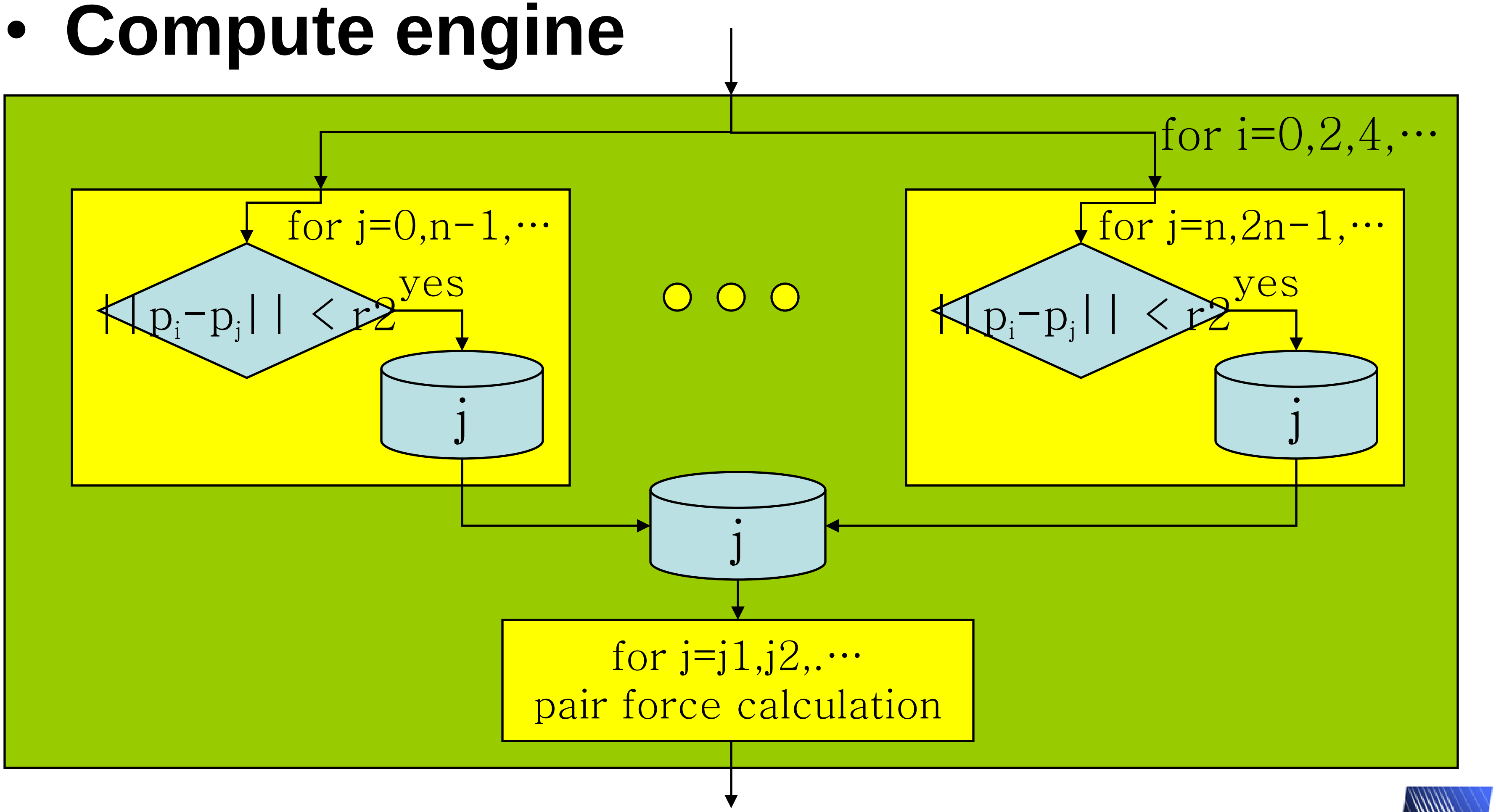


NAMD MAP code algorithm revised


start
start
send to bridge initial parameters
bridge
get initial parameters from bridge
yes
first time
no
DMA IN
yes
first time
no
A
B
C
D
E
F
DMA lookup tables to OBM
copy lookup tables to BRAM
copy lookup tables to BRAM
DMA IN
stream input data and copy to OBM, BRAM, and stream to bridge
A
B
bridge
copy from bridge to BRAM
compute engine A
compute engine B
merge results and stream out
bridge
merge results and stream out
A
B
stop
stop
DMA OUT

NAMD MAP code algorithm revised

- **Compute engine**



Revised NAMD MAP results

- Primary chip

```
INNER LOOP SUMMARY
loop on line 105:
    clocks per iteration:    1
    pipeline depth:         10

loop on line 143:
    clocks per iteration:    1
    pipeline depth:         10

loop on line 171:
    clocks per iteration:    1
    pipeline depth:         10

loop on line 277:
    clocks per iteration:    1
    pipeline depth:         31

loop on line 296:
    clocks per iteration:    1
    pipeline depth:         31

loop on line 315:
    clocks per iteration:    1
    pipeline depth:         31

loop on line 334:
    clocks per iteration:    1
    pipeline depth:         31

loop on line 353:
    clocks per iteration:    1
    pipeline depth:         31

loop on line 372:
    clocks per iteration:    1
    pipeline depth:         31

loop on line 396:
    clocks per iteration:    1
    pipeline depth:        151

loop on line 505:
    clocks per iteration:    1
    pipeline depth:          6
```

- Secondary chip

```
INNER LOOP SUMMARY
loop on line 105:
    clocks per iteration:    1
    pipeline depth:         10

loop on line 139:
    clocks per iteration:    1
    pipeline depth:         10

loop on line 169:
    clocks per iteration:    1
    pipeline depth:         10

loop on line 259:
    clocks per iteration:    1
    pipeline depth:         31

loop on line 278:
    clocks per iteration:    1
    pipeline depth:         31

loop on line 297:
    clocks per iteration:    1
    pipeline depth:         31

loop on line 316:
    clocks per iteration:    1
    pipeline depth:         31

loop on line 335:
    clocks per iteration:    1
    pipeline depth:         31

loop on line 354:
    clocks per iteration:    1
    pipeline depth:         31

loop on line 373:
    clocks per iteration:    1
    pipeline depth:         31

loop on line 392:
    clocks per iteration:    1
    pipeline depth:         31

loop on line 416:
    clocks per iteration:    1
    pipeline depth:        151

loop on line 507:
    clocks per iteration:    1
    pipeline depth:          5
```

Revised NAMD MAP results

- Primary chip

Device Utilization Summary:

Number of BUFGMUXs	1 out of 16	6%
Number of External IOBs	832 out of 1164	71%
Number of LOCed IOBs	832 out of 832	100%
Number of MULT18X18s	178 out of 444	40%
Number of RAMB16s	132 out of 444	29%
Number of SLICEs	44094 out of 44096	99%

Timing analysis: Actual: 9.994ns

- Secondary chip

Device Utilization Summary:

Number of BUFGMUXs	1 out of 16	6%
Number of External IOBs	745 out of 1164	64%
Number of LOCed IOBs	745 out of 745	100%
Number of MULT18X18s	139 out of 444	31%
Number of RAMB16s	150 out of 444	33%
Number of SLICEs	40427 out of 44096	91%

Timing analysis: Actual: 9.989ns

Execution time ~3.02 seconds (measured on CPU)

~0.15 seconds due to data DMA in/out and (measured on MAP)

~0.85 seconds due to MAP function call overhead

~2.02 seconds due to actual calculations (measured on MAP)

which is 3x speedup ☺

Conclusions

- Best speedup is 3x
  - on MAP-E (VP100) system
- FPGA size and speed are the biggest problems

- A 3x performance increase of this heavily optimized code is significant in that it illustrates the potential of reconfigurable system technology. Remember that it is a 100 MHz FPGA achieving a 3x application performance improvement over a 2.8 GHz CPU, and FPGAs are on a faster technology growth curve than CPUs.

Acknowledgements

- **James Phillips**, Theoretical and Computational Biophysics Group, Beckman Institute, University of Illinois at Urbana-Champaign
- **David Caliga**, SRC Computers, Inc.
- **Dan Poznanovic**, SRC Computers, Inc.
- **Jeff Hammes**, SRC Computers, Inc.
- **David Pointer**, Innovative Systems Laboratory, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign
- **Craig Steffen**, Innovative Systems Laboratory, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign