SUPPORTING INFORMATION

Revisiting Molecular Dynamics on a CPU/GPU system: Water Kernel and SHAKE Parallelization

A. Peter Ruymgaart and Ron Elber*
Department of Chemistry and Biochemistry, Institute for Computational Engineering and Sciences, University of Texas at Austin, Austin, TX 78712

APPENDIX A Preconditioned Conjugate Gradient parallel implementation (C)
Global double arrays of size number of constraints: r, x, b, Ax, Dinv, Lambda and vSig.

Parallel PCG main algorithm:

```
AxSparse(x, Ax, istart, iend);
PCG segment 1(istart, iend);
do while () {
AxSparseCGpartTwo(istart, iend, tid);
//-- barrier
alpha  = rtz/ptAp;
CGpartThree(istart, iend, tid, alpha, x);
//-- barrier
beta = rtznew / rtz;
FastAdd(vb, vx, vb, beta, istart, iend);
}
```

Parallel PCG Segment 1:

```
AxSparse (see text)
for (int n=istart; n<iend; n++) { //-- for loop not on GPU
double q = vSig[n] - Ax[n];
double p = Dinv[n] * q;
r[n] = q;
x[n] = p;
b[n] = p;
dot += q*q; //-- on the GPU, no + here
}
ttldot[thread] = dot; //-- not on GPU
v = dot; //-- on GPU
On GPU here, In kernel parallel reduction (Appendix B).
```

Note minimal array access here. The most important observation here is that array access is minimized. By storing vSig[n] - Ax[n] in local variable q, the Ax and Sig[] arrays do not need to be accessed again when making the conditioning matrix multiplication, during assignment of r, x, b and during the calculation of the partial dot product. We also gain from the fact that numerous operations are done in a single loop over the number of vector rows rather than having individual loops for each operation. The GPU version of this segment is identical except it lacks the loop since each thread corresponds to one row and the partial dot product is a single scalar per thread. In the other parts of the PCG, memory access is minimized in the same way.

Parallel PCG Segment 2 (contains local implementation of AxSparse):

```
for (int b=istart; b<iend; b++) { //-- for loop not on GPU
double elm = 0.0;
int elfirst = mSHKcmprow[b];
int ellast = mSHKcmprow[b+1];
```

```
for (int rel=elfirst; rel<ellast; rel++)  {
        int col = mSHKcol[rel];
        double xrc = mSHKelemDP[rel];
        elm += xrc * vb[col];
}

Ax[b] = elm;
dotpAp += b[b] * elm;
dotvrsvx += r[b] * x[b];
}
```

Parallel PCG Segment 3:

```
for (int n=istart; n<iend; n++) { //-- for loop not on GPU
Lambda[n] += alpha*b[n];
double vrselm = r[n] - alpha*Ax[n];
double vxnew = Dinv[n] * vrselm;
r[n] = vrselm;
x[n] = vxnew;
dot += vrselm*vrselm;
dotvrsvx += vxnew*vrselm;
}
```

Note again minimal array access here by locally storing and reusing vrselm and vxnew.

APPENDIX B GPU In-kernel parallel reduction with global thread barrier (CUDA C)

Parallel shared memory reduction code segment for blocksize of 128 (used inside kernels)

This reduction performs addition of all vector elements of max vector size 128*128 and reduces the data from 128 blocks to one single scalar. The array elem is of type __shared__ volatile float and since in shared memory can be accessed only by the same block. The array sdiCGred resides in global memory and is needed for the final block reduction. Boolean isLastBlockDone resides in global memory: it is set to true by the last thread of the last block of the block reduction as shown below.

```
__shared__ volatile float elem[128];
int n = blockIdx.x * blockDim.x + threadIdx.x;
int tid = threadIdx.x;
float v = 0.0;
//------ v is a locally used & stored variable: value calculated here ------
//------ reduction of v down to one block ------
elem[tid] = v;
__syncthreads();
if (tid < 64) {
elem[tid] += elem[tid + 64];
elem[tid] += elem[tid + 32];
```

```
elem[tid] += elem[tid + 16];
elem[tid] += elem[tid +  8];
elem[tid] += elem[tid +  4];
elem[tid] += elem[tid +  2];
elem[tid] += elem[tid +  1];
}
//----- global thread barrier ----
if (!tid) {
sdiCGred[blockid] = elem[0];
//-- the below is following example B5 in CUDA programming guide 4.0
unsigned int value = atomicInc(&sdiRedCount, 1000000);
isLastBlockDone = (value == (nblocks - 1));
__threadfence();
}
//------ final reduction --------
__syncthreads();
if (isLastBlockDone) //-- this should only happen to one block
{
v = sdiCGred[tid];
elem[tid] = v;
__syncthreads();

if (tid < 64) {
        elem[tid] += elem[tid + 64];
        elem[tid] += elem[tid + 32];
        elem[tid] += elem[tid + 16];
        elem[tid] += elem[tid +  8];
        elem[tid] += elem[tid +  4];
        elem[tid] += elem[tid +  2];
        elem[tid] += elem[tid +  1];
}
if (!tid)
{
        sdiCGred[0] = elem[0]; //-- this is the final scalar result
}
__threadfence_block();

}
```

APPENDIX C GPU water water interaction non bonded force kernel (CUDA C)

```
//------------------------- TIP3P water-water pair parameters --------------
#define OaOa  582001.1521f
#define ObOb 595.067236f
#define HqHq 57.743463584f
#define OqOq 230.974340666f
```

```
#define OqHq -115.48704875f
#define  SH256_KNOTSbyR 256.0/12.0
#define  SH256_InvKNOTSbyR 12.0/256.0
__shared__ volatile float FelX[256], FelY[256], FelZ[256];
//------------------- Lennard Jones and  electrostatic (real space) -------------------
__device__ inline float ElecAndVdwForce(float r, float r2, float A, float B, float C, float
valid)
{
        //-- Lennard Jones force
        float invr2 = 1.0f/r2; float invr6 = invr2*invr2*invr2;
        float df = A*invr6*invr6*invr2 + B*invr6*invr2;
        //-- Electrostatic (real space) force
        int ind = r * SH256_KNOTSbyR * valid;
        float findx = (float)ind;
        float dx = r - findx * SH256_InvKNOTSbyR * valid;
        df -= (FelX[ind] + (FelY[ind]*dx + FelZ[ind])*dx)*C*valid;
        df *= valid;
        return df;
}
//------------------- electrostatic (real space) only ---------------------------------

// only shared memory access in this inline function
//-------------------------------------------------------------------------
__device__ inline float ElecForce(float rx, float ry, float rz, float C, float valid)
{
        float r2 = rx*rx + ry*ry + rz*rz;
        float r = sqrt(r2);
        int ind = r * SH256_KNOTSbyR * valid;
        float findx = (float)ind;
        float dx = r - findx * SH256_InvKNOTSbyR * valid;
        return (FelX[ind] + (FelY[ind]*dx + FelZ[ind])*dx)*C*valid;
}
//------------------------  Force only kernel --------------------------
// spdCoorX, Y, Z and Fx, Fy, Fz are pointers to global device
// float arrays containing atomic
// coordinates and forces respectively.
//  spcoord is a pointer to gloabel float4 array of atomic coordinates
//-------------------------------------------------------------------------
__global__ void GPU_CalcNBF_listELFL_H2OxH2OSH256() {
        //-- each thread 1 water molecule, nWat is water index
        int nWat = blockIdx.x * blockDim.x + threadIdx.x;
        int io = sdiWaterList[nWat];
        //-- load force lookup into shared memory - - -
        //-- tex1D reads are not normalized and no filtering
        if (threadIdx.x < 128) {
        float4 F = tex1D(tELFLa, threadIdx.x);
```

```
FelX[threadIdx.x] = F.x; FelY[threadIdx.x] = F.y; FelZ[threadIdx.x] = F.z;
F = tex1D(tELFLa, threadIdx.x+128);
FelX[threadIdx.x+128] = F.x; FelY[threadIdx.x+128] = F.y;
FelZ[threadIdx.x+128] = F.z;
}
//---- 18 registers for water i coordinates and force
float iOx = spdCoorX[io], iOy = spdCoorY[io], iOz = spdCoorZ[io];
float iH1x = spdCoorX[io+1], iH1y = spdCoorY[io+1], iH1z = spdCoorZ[io+1];
float iH2x = spdCoorX[io+2], iH2y = spdCoorY[io+2], iH2z = spdCoorZ[io+2];
float AtioFx = Fx[io], AtioFy = Fy[io], AtioFz = Fz[io];
float Atih1Fx = Fx[io+1], Atih1Fy = Fy[io+1], Atih1Fz = Fz[io+1];
float Atih2Fx = Fx[io+2], Atih2Fy = Fy[io+2], Atih2Fz = Fz[io+2];
//-- obtain some values from constant memory
float cellx = sdUcellX, celly = sdUcellY, cellz = sdUcellZ;
float cutmax = sdUcellMaxCut, cut2 = sdUcellInnerCutoff2;
__syncthreads(); //-- shared mem table must be loaded
//--- loop over water neighbors (no sense trying to unroll this)
int nnbrs = spdiNrWatNbrWaters[nWat];
for (int n=0; n<MAX_NR_NB_NBRS/*nnbrs*/; n++){
if (n < nnbrs)   {
        int jo = spdiWatNbrWaters[n*WATSPACING + nWat];
        //--------- Oxygen j, read coordinates ---------------
        float4 j = spcoord[jo]; // OR  float4 j = tex1D(tCoord, jo);
        float rx = iOx - j.x, ry = iOy - j.y, rz = iOz - j.z;
        //---------- symmetry ---------
        float sx = 0.0, sy = 0.0, sz = 0.0;
        float symop;
        symop = (rx > cutmax); symx -= symop * cellx;
        symop = (ry > cutmax); symy -= symop * celly;
        symop = (rz > cutmax); symz -= symop * cellz;
        symop = (rx < -1.0f*cutmax); symx += symop * cellx;
        symop = (ry < -1.0f*cutmax); symy += symop * celly;
        symop = (rz < -1.0f*cutmax); symz += symop * cellz;

        //--------- water i Oxygen - water j oxygen ------------
        rx += symx; ry += symy; rz += symz;
        float r2 = rx*rx + ry*ry + rz*rz;
        float r = sqrt(r2);
        float valid = 0.0f;
        if (r2 < cut2) valid = 1.0f;
        float df = ElecAndVdwForce(r, r2, -OaOa12,ObOb6,OqOq, valid);
        AtioFx += df*rx; AtioFy += df*ry; AtioFz += df*rz;
        //------------ water i Hydrogen 1 - water j oxygen ----------
        rx = iH1x – j.x + sx; ry = iH1y – j.y + sy; rz = iH1z – j.z + sz;
        df = ElecForce(rx,ry,rz, OqHq, valid);
        Atih1Fx -= df*rx; Atih1Fy -= df*ry; Atih1Fz -= df*rz;
```

```
                //------------ water i Hydrogen 2 - water j oxygen ------------
                rx = iH2x – j.x + sx; ry = iH2y – j.y + sy; rz = iH2z – j.z + sz;
                df = ElecForce(rx,ry,rz, OqHq, valid);
                Atih2Fx -= df*rx; Atih2Fy -= df*ry; Atih2Fz -= df*rz;

                //--------- read water J hydrogen 1 coordinates ----------------
                j = spcoord[jo+1]; // OR j = tex1D(tCoord, jo+1);
                //------------ water Oi - water j hydrogen 1--------------------
                rx = iOx – j.x + sx; ry = iOy – j.y + sy; rz = iOz – j.z + sz;
                df = ElecForce(rx,ry,rz, OqHq, valid);
                AtioFx -= df*rx; AtioFy -= df*ry; AtioFz -= df*rz;
                //------------ water i Hydrogen 1 - water j Hydrogen 1 ---------
                rx = iH1x – j.x + sx; ry = iH1y – j.y + sy; rz = iH1z – j.z + sz;
                df = ElecForce(rx,ry,rz, HqHq, valid);
                Atih1Fx -= df*rx; Atih1Fy -= df*ry; Atih1Fz -= df*rz;
                //------------ water i Hydrogen 2 - water j Hydrogen 1 ----------
                rx = iH2x – j.x + sx; ry = iH2y – j.y + sy; rz = iH2z – j.z + sz;
                df = ElecForce(rx,ry,rz, HqHq, valid);
                Atih2Fx -= df*rx; Atih2Fy -= df*ry; Atih2Fz -= df*rz;

                //--------- read water J hydrogen 2 coordinates ----------------
                j = spcoord[jo+2]; // OR j = tex1D(tCoord, jo+2);
                //---------------------- water Oi - H2j -------------------------
                rx = iOx – j.x + sx; ry = iOy – j.y + sy; rz = iOz – j.z + sz;
                df = ElecForce(rx,ry,rz, OqHq, valid);

                AtioFx -= df*rx; AtioFy -= df*ry; AtioFz -= df*rz;
                //------------ water i Hydrogen 1 - water j Hydrogen 2 ---------
                rx = iH1x – j.x + sx; ry = iH1y – j.y + sy; rz = iH1z – j.z + sz;
                df = ElecForce(rx,ry,rz, HqHq, valid);
                Atih1Fx -= df*rx; Atih1Fy -= df*ry; Atih1Fz -= df*rz;
                //------------ water i Hydrogen 2 - water j Hydrogen 2 ----------
                rx = iH2x – j.x + sx; ry = iH2y – j.y + sy; rz = iH2z – j.z + sz;
                df = ElecForce(rx,ry,rz, HqHq, valid);
                Atih2Fx -= df*rx; Atih2Fy -= df*ry; Atih2Fz -= df*rz;
        }
        __syncthreads();
        } //-- end loop over water neighbors

        //-- write force results to global RAM --
        Fx[io] = AtioFx; Fy[io] = AtioFy; Fz[io] = AtioFz;
        Fx[io+1] = Atih1Fx; Fy[io+1] = Atih1Fy; Fz[io+1] = Atih1Fz;
        Fx[io+2] = Atih2Fx; Fy[io+2] = Atih2Fy; Fz[io+2] = Atih2Fz;
}
//---------------------------- end of function ---------------------------------------------
```