N-Body With CUDA

SciNet Parallel Scientific Computing Course

Aug 31 - Sept 4, 2009

CUDA in General

- CUDA likes *lots* of threads. Many more threads than processors.
- Context switching in CUDA is fast, so one team of threads can work on hardware while another waits for memory.
- Access to main memory very slow - threads have small (~48k) blocks of shared memory split between ~16 compute units.
- For fast CUDA code, must work on data while in shared memory (essentially cache).
- Unlike w/ CPU's *you* have to manually manage cache.

N-Body w/ CUDA

- CUDA likes lots of threads - so, one thread per particle.
- Will process particles in blocks. A team of threads loads the block of particles it owns, then loops through other blocks.
- Atomic add isn't supported for floating-point math, so we will end up doing double work. Unless any of you have clever ideas...

Getting Set Up

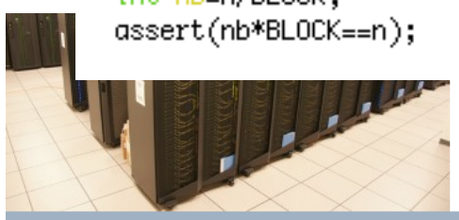
```
/*-------------------------------------------------------------------
float *cuda_vector(int n)
{
  float *vec;
  assert(cudaMalloc ((void **) &vec, sizeof (float) * n)==cudaSuccess);
  cudaMemset(vec,0,sizeof(float)*n);
  return vec;
}
/*-------------------------------------------------------------------
   float *cuda_copy(float *vec, int n)
   //Routine to take in input float vector, allocate space on the GPU for it, and copy the contents over.
   //Returns a pointed to the device vector.
   {
     float *dvec=cuda_vector(n);
     assert(cudaMemcpy(dvec,vec,sizeof(float)*n,cudaMemcpyHostToDevice)==cudaSuccess);
     return dvec;

   }

  void calculate_forces_gpu(NBody *data, int n)
  {
    float *x= (float *)malloc(sizeof(float)*n);
    float *y= (float *)malloc(sizeof(float)*n);
    float *z= (float *)malloc(sizeof(float)*n);

    float *fx= (float *)malloc(sizeof(float)*n);
    float *fy= (float *)malloc(sizeof(float)*n);
    float *fz= (float *)malloc(sizeof(float)*n);
    for (int i=0;i<n;i++) {
      x[i]=data[i].x[0];
      y[i]=data[i].x[1];
      z[i]=data[i].x[2];
    }
    //Copy the particle positions over
    float *dx=cuda_copy(x,n);
    float *dy=cuda_copy(y,n);
    float *dz=cuda_copy(z,n);
    //allocate space for the forces on the GPU.
    float *dfx=cuda_vector(n);
    float *dfy=cuda_vector(n);
    float *dfz=cuda_vector(n);

    int nb=n/BLOCK;
    assert(nb*BLOCK==n);
```



Need to copy particle positions over to the GPU. First extract them from the structure into vectors, then make space on the GPU for positions and forces, and copy positions over.

Note on notation: variables that live on GPU are just regular pointers. To keep them straight, start all device variables w/ extra d, so dy is device vector containing y.

Calling the Kernel

```
//make sure all the data is over before continuing.
cudaThreadSynchronize();
//Now calculate the forces.  We will start up nb blocks of threads, each of size BLOCK
get_forces<<<nb, BLOCK>>> ( n,dx,dy,dz,dfx,dfy,dfz);
//Make sure everybody is done calculating before we try to copy off the device.
cudaThreadSynchronize();
//Now copy the forces off of the GPU.
assert(cudaMemcpy(fx,dfx,sizeof(float)*n,cudaMemcpyDeviceToHost)==cudaSuccess);
assert(cudaMemcpy(fy,dfy,sizeof(float)*n,cudaMemcpyDeviceToHost)==cudaSuccess);
assert(cudaMemcpy(fz,dfz,sizeof(float)*n,cudaMemcpyDeviceToHost)==cudaSuccess);

for (int i=0;i<n;i++) {
  data[i].f[0]=-fx[i]* GRAVCONST*data[i].mass *data[i].mass ;
  data[i].f[1]=-fy[i] * GRAVCONST*data[i].mass *data[i].mass ;
  data[i].f[2]=-fz[i] * GRAVCONST*data[i].mass *data[i].mass ;
}

cudaFree(dx);
cudaFree(dy);
cudaFree(dz);
cudaFree(dfx);
cudaFree(dfy);
cudaFree(dfz);
```

CUDA needs to know how many threads to fire up. The call func<<<a,b>>> (c,d,...)
runs function func on the GPU, starting *a* blocks of threads, each of which has *b* threads in it. *a* and *b* can also be 2- and 3-D.

cudaThreadSynchronize() waits for all threads to finish what they're doing before moving on.

Use cudaFree to free memory allocated on the GPU.

Setting Up Inside the Kernel

```
//function to calculate forces on the GPU, to be called from the host.
__global__ void get_forces(int n, float *x, float *y, float *z, float *fx, float *fy, float *fz)
{
  //The __shared__ memory is a small block of cache shared between a block of threads.
  //To run fast, each block of threads copies global thread records into a local fast-memory
  //area.
  __shared__ float lx[BLOCK],ly[BLOCK],lz[BLOCK],lfx[BLOCK],lfy[BLOCK],lfz[BLOCK];
  __shared__ float lx2[BLOCK],ly2[BLOCK],lz2[BLOCK];

  //which particle do I own?
  int myind=blockIdx.x*blockDim.x+threadIdx.x;

  //These are now vector calls copying in contiguous chunks of memory into the local shared memory.
  lx[threadIdx.x]=x[myind];
  ly[threadIdx.x]=y[myind];
  lz[threadIdx.x]=z[myind];

  //clear the forces on the particle I own.
  lfx[threadIdx.x]=0;
  lfy[threadIdx.x]=0;
  lfz[threadIdx.x]=0;

  //make sure everybody is done before progressing.
  //syncthreads applies to a single block of threads, so
  //it is extremely fast - something like 4 clock cycles.
  __syncthreads();
  float dx,dy,dz,r,r3;
  int i,blck,nblock;
  nblock=n/BLOCK;
  int myind2;
```

We don't want to work on variables in main memory. Instead, copy blocks of data to __shared__ memory. This is local cache shared between blocks of threads - very fast, and all threads in a team see it. So, if have 32 thread blocks, and each thread loads just one #, threads can see all 32 #'s. I have put l(ell) at the beginning of __shared__ variable names.

Variables blockIdx, blockDim, threadIdx are automatically set by CUDA so threads know which ones they are. 0<=threadIdx.x <blockDim.x

Doing the Force Calculations

```
//goal is to pull blocks of particles into __shared__ memory, then loop over all
//of these particles while they're around.  I have yet to figure out a good way to take
//advantage of the force from my particle on other particles, so going to be doing
//double work.
for (blck=0;blck<nblock;blck++)  {

  //which particle in the new block do I own?
  myind2=blck*blockDim.x+threadIdx.x;
  //load the particle block into cache.
  lx2[threadIdx.x]=x[myind2];
  ly2[threadIdx.x]=y[myind2];
  lz2[threadIdx.x]=z[myind2];

  //make sure all the data is loaded before going on.
  __syncthreads();
  for (i=0;i<BLOCK;i++) {
    //everybody calculates the force from particle i in the block on
    //themselves.
    dx=lx[threadIdx.x]-lx2[i];
    dy=ly[threadIdx.x]-ly2[i];
    dz=lz[threadIdx.x]-lz2[i];
    r=dx*dx+dy*dy+dz*dz+EPS*EPS;
    //rsqrt is the reciprocal square root.
    r=rsqrt(r);
    r3=r*r*r;
    lfx[threadIdx.x]+=r3*dx;
    lfy[threadIdx.x]+=r3*dy;
    lfz[threadIdx.x]+=r3*dz;
    //end loop over particles in the temporary block
  }
  //why do I need a syncthreads here?  I get wrong answers without it.
  __syncthreads();
  //end loop over blocks.
}
```

Each team of threads owns one block of particles. Loop over other blocks. Process by loading another block of particle positions in, calculating their forces on me, and accumulating. When done, put the forces in __shared__ back in main GPU memory.

Different threads accessing shared blocks of memory can be tricky. Think very carefully about thread synchronization. The __syncthreads() command is a barrier for a block of threads, you will need it.

Don’t Forget to Send Things Back

```
  //now I have all the forces on the particles owned by this block of threads.  Send 'em back.
  fx[myind]=lfx[threadIdx.x];
  fy[myind]=lfy[threadIdx.x];
  fz[myind]=lfz[threadIdx.x];


}
/*----------------------------------------------------------------------------*/
```

Forces are calculated in __shared__ memory. Don’t forget to send them back to main GPU memory, so the CPU can grab them.

Code lives on SciNet in /scratch/sievers/cuda

And now, watch the GPU in action.