Molecular dynamics on graphics accelerators

Samuel Kupka*

Faculty of Sciences
University of Pavol Jozef Safarik
Kosice / Slovakia

Abstract

Molecular dynamics algorithm optimized for the graphics accelerators is presented in this paper. Parts of it can be used to optimize existing algorithms designed for the classic CPU.

Keywords: molecular dynamics, graphics accelerators, computer simulation, SIMD processor, BrookGPU

1 Introduction

The computer aided simulations are an important part of science work. It can be used to test new theoretical results without a need of the expensive laboratory equipment. Even the personal computers are becoming fast enough to simulate large systems in reasonable time.

The ultimate goal of companies, that create graphics accelerators is to provide their customers with the highest possible computing speed for computer games. Because of those companies, personal computers are now equiped with very fast graphics processors that can be utilized in many useful ways. Running computer aided simulations is one of them.

Molecular dynamics (MD) simulation is a discipline of molecular modeling. It addresses numerical solutions of Newton's equations of motion on an atomistic or similar model of a molecular system to obtain information about its time-dependent properties. It is used to examine the dynamics of atomic-level phenomena that cannot be observed directly [4]. Constraints of the simulated systems are given by the overall computer system performance and time allocated for the MD to produce usable results [3]. There have been many optimalization techniques introduced to the MD simulations [5], but there is still a need to find a new ways to speed up whole application as such. Graphics accelerators (GA) are still viewed as a good possibility, although many of it's internal characteristics are not very fitting for this type of exploitation. This paper introduces algorithm that should compensate some of the bad aspects of the graphics accelerators (or SIMD processors).

*bwpow@ideaz.sk

1.1 Programming language

Most of the programming languages for GA are directed towards shaders or geometry applications. For the implementation of my algorithm, I've chosen BrookGPU[1] [2]. It is a compiler and runtime implementation of the Brook [2] stream programming language for modern graphics hardware. I've chosen this one, because it provides easy access to GPU without a need to understand graphics procedures. This way, source code can be viewed and edited by non-graphics programmers.

Version of BrookGPU which was used in the beginning of the development introduced some important limitations. It did not have support FOR cycles in kernel functions. All cycles had to be unrolled by compiler. It did not provide developer with effective ways to debug programs and some of the functions were available only for the specific hardware. Some of these limitations were removed during the development of BrookGPU and new graphics cards.

2 Basic MD

Molecular dynamics is based on an idea of introducing Newton's equations to a large set of sites (atoms, molecules). The other possible approach is based on statistical model. In real practical MDs, both models are used together to produce most effective system [4] [1]. Algorithm described in this paper utilizes only the first approach.

The most simple basic MD is an $O(N^2)$ algorithm. Let S be a set of all sites, then the algorithm can be described as:

```
Do forever{
 For each site X in S do{
  For each site Y in S do{
   Calculate forces from X to Y
  }
  Update parameters of X
 }
}
```

Because forces between sites are getting less significant with greater distance, idea of neighbooring lists. For ev-

[1] http://graphics.stanford.edu/projects/brookgpu/
[2] http://merrimac.stanford.edu/brook/

ery site, the list of all other significant sites is generated. Simple algorithm may look like this:

```
Do forever{
 For each site X in S do{
  For each site Y in S do{
   If(Distance(X,Y)<MaxDistance){
    Add Y to Neighbour(X)
   }
  }
 }
 Do N steps{
  For each site X in S do{
   For each site Y in Neighbour(X) do{
    Calculate forces from X to Y
   }
   Update parameters of X
  }
 }
}
```

Where N depends on the model used. Of course, there are many improvements and modification of this simple idea, but this paper does not cover those.

3 Proposed algorithm

Proposed algorithm contains CPU and GPU part. The CPU part is responsible for streams preparation and running kernel functions from the GPU part.

Algorithm provides a way to create and keep neighbours lists with constant sizes to be useable in kernel functions.

3.1 Basic idea

The simulation space (or cell) has a shape of the cube. Let N be the number of the sites in the space. We divide the space into the M^3 cube shaped regions. All regions must have same sizes. Constant M is chosen from the formula $C < N/M^3 < 2C$, where C is the constant value used to unroll loops and as the size of the neightbours lists.

Value of the constant C should be from four up to eight times larger than number of GA pipelines. Each kernel function used in the algorithm computes forces for C sites for one call. If C is too small, algorithm gets slow becasue of overhead with every kernel function call. If the constant is too large, algorithm looses performance because of inefective use of neighbours lists.

Lets call the geometrical center of each region the seed for this region. Lets compute distance from each site to the closest seed. This can be done by computing distance from each seed to every site. This operation needs $O(NM^3)$ steps. Lets sort all sites by the distance from the closes seed. This can be done in $O(N^2)$ steps. It is possible to optimize performance of these two procedures, but there won't be any significant change to the whole algorithm. This way we get the list of the closest sites for every seed. Because the sites are not evenly distributed in the space, each list contains different number of sites.

To create neighbours list for each region, get the list of closest sites and divide it into the nonoverlaping sublists, each of size C. Pad the last sublist for every region with $EMPTY$ values. All the sublists for each region form a neighbours list for that region. Neighbours lists can be desribed as spheres with the center in the seed and radius same as maximal distance of all sites in the list from the seed. Spheres from different regions may overlap. Let $MaxDistance$ be the maximal distance of two sites with nominal impact on each other. If the shortest distance of two spheres is more then $MaxDistance$, then the sites covered with one sphere don't have impact on the sites in the second sphere. Let A and B be the two spheres and $IsNear$ M^3xM^3 matrix. For each two spheres A and B, if the shortest distance from A to B is less than $MaxDistance$, set $IsNear[A][B] = 1$ else set $IsNear[A][B] = 0$. This way, we got the list of neighbouring spheres. For this algorithm, we will assume, that if the two spheres are not neighbours ($IsNear$ is set to 0), then all sites within these two spheres are not neighbours as well.

3.2 CPU part

The CPU part generates $IsNear$ matrix and runs the kernel functions.

```
Create regions and get seeds
Do forever{
 label :beginning;

 For each site X in S do{
  Min[X]=INFINITY
  For each seed Z do{
   if(Distance(X,Z)<Min[X]){
    Min[X]=Distance(X,Z);
    Belong[X]=Z;
   }
  }
 }

 Sort all sites X in S by Min[X];

 For each site X in S do{
  Put X into Sphere[Belong[X]];
 }

 For each sphere A in Sphere do{

  For each sublist U in A do{
    PadUnusedPlacesWithEMPTY(U);
  }

  For each sphere B in Sphere do{
   If(ShortestDist(A,B)<MaxDistance){
    IsNear[A,B]=1;
```

```
   }
   else{
    IsNear[A,B]=0;
   }
  }
 }

 Do forever{
  For each sphere A in Sphere do{

   For each sublist U in A do{
    For each sublist V in A do{
     If(U != V){
      kernel add_forces(U,V);
     }
    }
   }

   For each sphere B in Sphere do{
    If(IsNear[A][B]==1){

     For each sublist U in A do{
      For each sublist V in B do{
        kernel add_forces(U,V);
      }
     }

    }
   }

   For each sublist U in A do{
    kernel move_sites(U);
    D = 0;
    reduce max_dist_from_seed(U,A,D);
    if(D>MaxAllowedDistance){
     goto :beginning;
    }
   }

  }
 }
}
```

This is a very simplified scheme of the algorithm.

3.3 GPU part

The GPU part consists of two kernel and one reduce function.

```
kernel add_forces(stream U, static V)
{
 If(U!=EMPTY){
  Vector F;
  For I=1 to C do{
   If(V[I]!=EMPTY){
    F+=ComputeForces(U,V[I]);
   }
  }
  AddForceTo(U,F);
 }
}

kernel move_sites(stream U)
{
 If(U!=EMPTY){
  MoveSite(U);
 }
}

reduce max_dist_from_seed
 (stream U,const A,out D)
{
  E=Distance(U,A);
  if(E>D) D=E;
}
```

The functions called from the kernel functions are dependant on the used physical model as are the constants *MaxDistance* and *MaxAllowedDistance*.

4 Performance

This algorithm was developed as proof-of-concept, so it doesn't contain most of the standard optimization techniques used in modern MD systems. Performance of the algorithm is also dependent on the used GA and physical model. It is optimal only for the short-range forces physical models and large numbers of the sites. The use of the PciExpress BUS instead of AGP gives a big performance boost. Algorithm needs to copy all sites information between System RAM and GA RAM because of the neighbours lists regeneration after every few hundreds steps, depending on the physical model and constants values.

Performance of the GA algorithm compared to the basic algorithms described in section 2 with 1000 sites and simulating 1000 steps.

Computer	GA	Basic	Basic with NL
System A	21.3s	219s	119s
System B	11.1s	137s	63s
System C	3.7s	151s	81s

Table 1: Performance of different algorithms

System A was AMD Athlon 2500+, NVidia 5500 AGP, System B was Intel Pentium4 3.2GHz, NVidia 6600 PciE and System C was AMD Athlon X2 4400+, 2x NVidia 6800 GT PciE.

5 Conclusion and future work

This algorithm shows a very simple way to use GA for molecular dynamics and other similar simulation meth-

ods. The use of BrookGPU gives oportunity to the people not familiar with graphics hardware to edit algorithm and add their own functions. The use of constant fields, as described by this algorithm, may be used to optimize other non-GPU MD systems as well.

In the future, I plan to rewrite most parts of the algorithm to Cg [3] language and integrate it into the GROMACS [4].

6 Acknowledgments

The author thanks to Prof. Aatto Laaksonen and the Fyzikal chemistry department of Stockholm University for great help and technical support. To Doc. Jozef Ulicny, RNDr. Jozef Jirasek and Faculty of Science of University of Pavol Jozef Safarik in Kosice for support and help.

References

[1] J. M. Haile. *Molecular Dynamics Simulation : Elementary Methods*. John Wiley & Sons, Inc., 1997.

[2] Daniel Horn Jeremy Sugerman Kayvon Fatahalian Mike Houston Ian Buck, Tim Foley and Pat Hanrahan. *Brook for GPUs: Stream Computing on Graphics Hardware*. Computer Science Department, Stanford University, SIGGRAPH, 2004.

[3] S. J. Plimpton. *Computational Limits of Classical Molecular-Dynamics Simulations*. Computational Materials Science, 1995.

[4] D. C. Rapaport. *The Art of Molecular Dynamics Simulation, Second edition*. Cambridge University Press, 2004.

[5] B. A. Hendrickson S. J. Plimpton. *Parallel Molecular Dynamics Algorithms for Simulation of Molecular Systems*. American Chemical Society, Symposium Series 592, 1995.

[3] http://developer.nvidia.com/
[4] http://www.gromacs.org/