JCTC Journal of Chemical Theory and Computation

Article

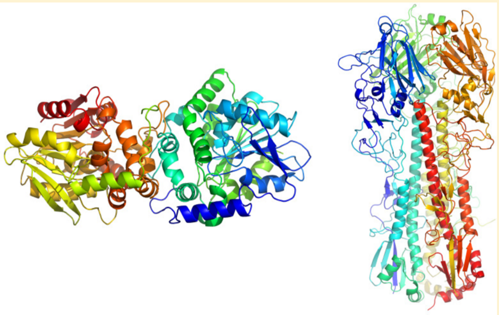
pubs.acs.org/JCTC

GPU-Accelerated Exploration of Biomolecular Energy Landscapes

Rosemary G. Mantell,* Catherine E. Pitt,* and David J. Wales*

University Chemical Laboratories, Lensfield Road, Cambridge CB2 1EW, United Kingdom

ABSTRACT: We present graphics processing unit (GPU)-acceleration of various computational energy landscape methods for biomolecular systems. Basin-hopping global optimization, the doubly nudged elastic band method (DNEB), hybrid eigenvector-following (EF), and a local rigid body framework are described, including details of GPU implementations. We analyze the results for eight different system sizes, and consider the effects of history size for minimization and local rigidification on the overall efficiency. We demonstrate improvement relative to CPU performance of up to 2 orders of magnitude for the largest systems.



1. INTRODUCTION

Basin-hopping global optimization,[1,2] the doubly nudged elastic band method (DNEB),[3] and hybrid eigenvector-following (EF)[4] are well established tools for the location of minima and transition states in the characterization of potential energy landscapes. Their application to biomolecules has so far focused on smaller systems, due to computational expense. We would like to open up the study of larger biomolecules within the computational potential energy landscape approach, through the use of GPU hardware for acceleration.

Graphics processing units (GPUs) were originally designed for fast graphics rendering, but they have become increasingly used in general-purpose computations as massively parallel processors.[5] Relative to a CPU, they have a greater number of transistors devoted to data-processing, than to data-caching and flow control. This feature makes them ideal for computations with high arithmetic intensity (the ratio of arithmetic operations to memory operations).[6] The NVIDIA parallel computing platform and programming model CUDA facilitates the task of exploiting this architecture effectively. CUDA allows the definition of functions known as "kernels", which execute N times in parallel on N threads. Threads are grouped together into blocks, many of which can execute concurrently, forming a grid. GPUs have several different memory spaces available. Each thread can make use of registers, which have extremely fast access, but only have the lifetime of the thread. Low access latency shared memory allows threads within a block to share data. The majority of the available memory is global device memory, which is available to any thread in any block over the lifetime of the application, though access to this resource is relatively slow. Read-only, cached texture memory and constant memory can also be useful in certain applications.

The remainder of this report outlines the potential energy landscape framework, the adaptations for GPUs, and results for a range of system sizes.

2. METHODS

2.1. Basin-hopping global optimization. Basin-hopping global optimization facilitates the exploration of potential energy surfaces. It is particularly useful for locating the global minimum, though other properties of interest may be investigated by performing thermodynamic calculations using the database of local minima found during the search. The coordinate space is explored stepwise using random structural perturbations. For biomolecules, these perturbations may be Cartesian moves of the backbone, rotations of amino acid side chains, or short molecular dynamics (MD) runs. After each perturbation, geometry optimization is performed to find the local minimum associated with this point in coordinate space, usually via L-BFGS[7] minimization (the limited-memory version of the BFGS algorithm, named for Broyden,[8] Fletcher,[9] Goldfarb,[10] and Shanno[11]). The step taken is then either accepted or rejected using a Metropolis criterion. Thus, the landscape is transformed into a series of plateaux or "basins of attraction".[2] The transformed energy can be expressed as

$$\tilde{E}(\mathbf{X}) = \min\{E(\mathbf{X})\} \qquad (1)$$

where min indicates that a local minimization is carried out from the starting coordinates, **X**. The main advantage of this method is the ease with which areas of the landscape separated by high barriers can be explored.

The most time-consuming component of the above process is the calculation of the potential energy and gradient. Much of the previous work employing basin-hopping to investigate biomolecular systems has used the CPU implementation of the AMBER potential interfaced with our GMIN code.[12] This interface is henceforth referred to as "A12GMIN". A GPU-accelerated version of the AMBER potential[13] has now been released. The generalized Born (GB) implicit solvent potential from AMBER 12 was interfaced with GMIN. The DPDP

Received: September 23, 2016
Published: November 7, 2016

ACS Publications © 2016 American Chemical Society

DOI: 10.1021/acs.jctc.6b00934

precision model was used, in which contributions to forces and their accumulation are both performed in double precision. The GPU-accelerated version of GMIN will be referred to as "CUDAGMIN".

Although the potential energy calculation accounts for the majority of the basin-hopping run time, the time taken by the vector operations comprising the L-BFGS routine[7] is still significant. L-BFGS is a quasi-Newton optimization algorithm,[14] where the approximation to the inverse Hessian is represented implicitly through a number of vector pairs that are updated at each iteration. An overview of the method for the minimization of a function, *f*, is expressed in Algorithm 1.[15]

Algorithm 1: L-BFGS

```
Choose a starting point, $\mathbf{x}_0$, and an integer history size, $m > 0$
$k = 0$
repeat
    Choose an initial Hessian approximation, $H_k^0$, e.g. using Eq. 2
    Compute the step direction, $\mathbf{p}_k = -H_k \nabla f_k$, using Algorithm 2
    Compute the step $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$, choosing $\alpha_k$ as in Algorithm 3
    if $k > m$ then
        discard vector pair $\{\mathbf{s}_{k-m}, \mathbf{y}_{k-m}\}$ from storage
    end if
    Compute and store $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k, \mathbf{y}_k = \nabla f_{k+1} - \nabla f_k$
    $k = k + 1$
until convergence.
```

The initial approximation to the Hessian, $\mathbf{H}_k^0$, may vary freely between iterations. One commonly used method is to set $\mathbf{H}_k^0 = \gamma_k \mathbf{I}$, where

$$\gamma_k = \frac{\mathbf{s}_{k-1}^T \mathbf{y}_{k-1}}{\mathbf{y}_{k-1}^T \mathbf{y}_{k-1}} \tag{2}$$

This scaling factor, γ_k, is an estimate of the magnitude of the second derivative along the most recent search direction, ensuring that $\mathbf{p}_k$ is appropriately scaled.[15] The usual procedure for calculating $\mathbf{H}_k \nabla f_k$ is known as the two-loop recursion algorithm and is shown in Algorithm 2.[15]

Algorithm 2: Two-Loop Recursion

```
$\mathbf{q} = \nabla f_k$
for $i = k - 1, k - 2, ..., k - m$ do
    $\alpha_i = \rho_i \mathbf{s}_i^T \mathbf{q}$
    $\mathbf{q} = \mathbf{q} - \alpha_i \mathbf{y}_i$
end for
$\mathbf{r} = \mathbf{H}_k^0 \mathbf{q}$
for $i = k - m, k - m + 1, ..., k - 1$ do
    $\beta = \rho_i \mathbf{y}_i^T \mathbf{r}$
    $\mathbf{r} = \mathbf{r} + \mathbf{s}_i(\alpha_i - \beta)$
end for
stop with result $\mathbf{H}_k \nabla f_k = \mathbf{r}$.
```

A line search is often used to determine an appropriate step length, α_k. However, we have found it more efficient to use a simple procedure that checks whether the step is a descent direction, and ensures that it does not exceed a specified maximum step size and energy rise. This approach is further described in Algorithm 3.

Although many of the individual vector calculations are amenable to parallellization, the loops through the history size, *m*, in Algorithm 2 are necessarily serial. At larger history sizes, a larger number of vector calculations must therefore be performed sequentially. This scaling gives a more accurate step direction, so fewer steps (and hence fewer potential calls) would be required for convergence; however, it is not immediately clear whether a larger history size would result in reduced time for convergence on GPU hardware.

The L-BFGS algorithm has been implemented on GPUs by various other authors.[16−26] In adapting any application to make use of GPU hardware, a decision must be made as to which computations should take place on the GPU and which would better be left on the CPU. In "GPU Computing Gems", Haque and Pande consider this problem with reference to the L-BFGS algorithm. For their application, the function and gradient evaluation are well-suited to efficient parallelization on the GPU because of high data parallelism and arithmetic intensity. Conversely, they find that the L-BFGS direction update is ill-suited to GPU parallelization, as it consists of a large number of sequential, low-dimensional vector operations, requiring frequent thread synchronization and leaving many idle threads. However, leaving the update operations on the CPU necessitates copying large amounts of data between the host and device, which is expensive. In their case, implementing the entire algorithm on the GPU turned out to be the most efficient approach.[16] D'Amore et al.[17] also point out the difficulty of parallelizing the update operations in the L-BFGS routine. Although a dot product computation has a high degree of parallelism, each multiplication requires two read operations from the input vectors and one write operation to the output vector. The calculation is therefore strongly memory bandwidth bound. The upper limit for GPU-acceleration of the dot product is given by the ratio of GPU to CPU memory bandwidth, which is usually less than ten. D'Amore et al. made use of the cuBLAS library in their work, a GPU-accelerated version of the complete standard BLAS library.

Several other authors have also implemented versions of L-BFGS with both the function and gradient evaluation and vector update operations taking place on the GPU, namely Fei et al.,[18] Zhang et al.,[19] and Wetzl et al.[20] In contrast, others have opted to put only the function and gradient evaluation on the GPU.[21−26] In cases where the problem size has low dimensionality, the cost of data transfer between host and device is lowered, and the poor parallelization of the update steps becomes more significant.[23] However, several of these authors speculate that costly memory transfer negatively impacts the performance of their applications.[24−26] The most efficient approach likely depends on the hardware, the nature of the cost function, and the L-BFGS parameters selected.

We chose to port the whole L-BFGS algorithm to GPU so that we could compare it to having just the potential energy function on GPU. Our implementation is a modified version of "CudaLBFGS"[20,27] (Creative Commons Attribution 3.0 Unported License), an existing GPU-accelerated version of L-BFGS, authored by Wetzl and Taubmann. They programmed all the vector operations of Algorithms 1 and 2 on GPU using the NVIDIA cuBLAS library, which delivers 6 to 17 times faster performance than the latest MKL BLAS library.[28] Single-threaded kernels are used to perform operations in device memory not involving vectors or matrices. We also implemented the line search described in Algorithm 3 using cuBLAS, and made various other modifications to make the L-BFGS routine as similar as possible to its CPU equivalent in GMIN. This code, written in a mixture of C++ and CUDA C, was interfaced with the Fortran code of GMIN. The starting coordinates and other relevant parameters are copied from host to device at the start of the calculation, and the energy, gradient, and other useful output values are copied from device to host at the end.

DOI: 10.1021/acs.jctc.6b00934

Algorithm 3: Step Scaling in GMIN

```
if $\mathbf{r}_k^T \nabla f_k > 0$ then                ▷ Check if step, $r$, is a descent direction
    $\mathbf{r}_k = -\mathbf{r}_k$                      ▷ If not, invert the step
end if
$c = 1.0$                                               ▷ Initialise scaling factor for step size
if $c|\mathbf{r}_k| > d$ then                 ▷ Check if initial step larger than allowed maximum, $d$
    $c = d/|\mathbf{r}_k|$                     ▷ If it is, find new scaling factor to reduce it
end if
for $i = 1, 2, ..., 10$ do
    $\mathbf{x}_{k+1} = \mathbf{x}_k - c\mathbf{r}_k$    ▷ Calculate coordinates of new, scaled step
    if $f_{k+1} - f_k < u$ then                          ▷ Check energy change for new step
        break              ▷ If less than allowed maximum, $u$, accept current value of $c$
    else                                             ▷ If energy change greater than $u$
        $c = c/10$                                          ▷ Reduce $c$ by a factor of 10
    end if
end for
stop with result $\alpha_k = c$.
```

Local rigid body framework. The local rigid body framework in our programs, GMIN and OPTIM,[29] provides a way of reducing the number of degrees of freedom, while still retaining full atomistic interactions.[30,31] Rigid body coordinates are used to generate atomic coordinates, prior to the calculation of the energy and gradient. The atomistic forces are then projected to obtain the forces and torques on the rigid bodies, as required for L-BFGS.

The position of each rigid body is specified using three coordinates for the center of geometry, $\mathbf{r}_I$, and three for its orientation, $\mathbf{p}_I$. Capital letters are used here to refer to rigid bodies and lower case letters for atoms or sites. The reference coordinates of the atoms in rigid body I, relative to the center of geometry, are denoted by $\{\mathbf{r}_i^0\}_{i\in I}$. Using an angle-axis framework,[32] the rotation vector can be expressed as $\mathbf{p}_I = \theta_I \hat{\mathbf{p}}_I = (p_I^1, p_I^2, p_I^3)$, where $\hat{\mathbf{p}}_I$ is a unit vector defining the rotation axis and θ_I describes the magnitude of the rotation about that axis. p_I^1, p_I^2, and p_I^3 are the components of the rotation vector in the x, y, and z directions, respectively. The rotation matrix, $\mathbf{R}_I$, can be obtained from Rodrigues' rotation formula[33] as

$$\mathbf{R}_I = \mathbf{I} + (1 - \cos\theta_I)\tilde{\mathbf{p}}_I\tilde{\mathbf{p}}_I + \sin\theta_I\tilde{\mathbf{p}}_I \tag{3}$$

where $\mathbf{I}$ is the identity matrix. $\tilde{\mathbf{p}}_I$ is the skew-symmetric matrix defined as

$$\tilde{\mathbf{p}}_I = \frac{1}{\theta_I}\begin{pmatrix} 0 & -p_I^3 & p_I^2 \\ p_I^3 & 0 & -p_I^1 \\ -p_I^2 & p_I^1 & 0 \end{pmatrix} \tag{4}$$

which is constructed using the rotation vector $\mathbf{p}_I$. The formula

$$\mathbf{r}_{i\in I} = \mathbf{r}_I + \mathbf{R}_I\mathbf{r}_{i\in I}^0 \tag{5}$$

defines the transformation from rigid body coordinates to atomistic coordinates.[30]

The projection of the atomistic forces onto the translational degrees of freedom of the rigid bodies is given by the sum[30]

$$\frac{\partial U}{\partial r_I^k} = \sum_{i\in I}\frac{\partial U}{\partial r_i^k} \tag{6}$$

The accompanying projection of the forces on the atoms onto the rotational degrees of freedom is obtained through the chain rule as

$$\frac{\partial U}{\partial p_I^k} = \sum_{i\in I}\nabla_i U\cdot(\mathbf{R}_I^k\mathbf{r}_i^0) \tag{7}$$

where

$$\frac{\partial \mathbf{r}_i}{\partial p_I^k} = \mathbf{R}_I^k\mathbf{r}_i^0 \tag{8}$$

follows geometrically from eq 5.[31]

The derivative of the rotation matrix $\mathbf{R}_I^k$ (k = 1,2,3) can be expressed as[34]

$$\mathbf{R}_I^k = \frac{p_I^k\sin\theta_I}{\theta_I}\tilde{\mathbf{p}}_I^2 + (1 - \cos\theta_I)(\tilde{\mathbf{p}}_I^k\tilde{\mathbf{p}}_I + \tilde{\mathbf{p}}_I\tilde{\mathbf{p}}_I^k) + \frac{p_I^k\cos\theta_I}{\theta_I}\tilde{\mathbf{p}}_I + \sin\theta_I\tilde{\mathbf{p}}_I^k \tag{9}$$

where, for example,

$$\tilde{\mathbf{p}}_I^1 = \frac{1}{\theta_I}\begin{pmatrix} 0 & p_I^1p_I^3/\theta_I^2 & -p_I^1p_I^2/\theta_I^2 \\ -p_I^1p_I^3/\theta_I^2 & 0 & -(1-(p_I^1)^2/\theta_I^2) \\ p_I^1p_I^2/\theta_I^2 & (1-(p_I^1)^2/\theta_I^2) & 0 \end{pmatrix} \tag{10}$$

If the entirety of the L-BFGS algorithm is to be on the GPU, the coordinate transformation and gradient projection should also take place there to avoid unnecessary memory copying. The rotation matrix of eq 3 is calculated independently for each rigid body, presenting an obvious opportunity for parallelization. A kernel to calculate the rotation matrices was implemented, launching a number of threads equal to the number of rigid bodies, each of which also constructed the intermediate skew-symmetric matrix of eq 4 in registers. The calculations involving loops through small, nine-element matrices were unrolled for performance. The resulting rotation matrices, stored in global memory, were passed as an argument to the subsequent transformation kernel. This kernel encoded the main transformation of eq 5 and utilized a greater degree of parallellization than the previous kernel, with a number of active threads equal to the number of atoms.

For the sum of eq 6, providing the translational forces on the rigid bodies, an approach based on the "Shuffle On Down" algorithm was used to perform a parallel reduction.[35] This procedure makes use of a feature first introduced with the Kepler GPU architecture, namely the shuffle instruction, which allows threads to read the registers of other threads in the same warp (group of 32 threads). There are four shuffle intrinsics, but only "__shfl_down()" is used in this case, specifically the double-precision implementation detailed in ref 35. The final result of the shuffle down instruction is that values held in registers for each thread are shifted to lower thread indices. Reduction within a warp can be achieved by summing pairs of values obtained through a shuffle down operation, where the shift is initially half the warp size and then repeatedly bisected until thread 0 holds the final value, i.e. constructing a reduction tree. Parallel reduction using shuffle instructions in this way is much faster than the approach of using shared memory to exchange data between threads. This function was implemented for summations involving rigid bodies of 32 atoms or fewer. Using a kernel launch with a number of threads equal to 32 times the number of rigid bodies, the register values to be summed are either the appropriate value read from global memory, or zero where the atom does not exist. The first thread of each warp was designated to write the final reduced values to global memory.

The most frequently employed local rigid groupings do not involve more than 32 atoms, and so this approach is sufficient in these cases. However, sometimes larger rigid bodies are needed, so the reduction must be extended to summing more than 32 values. To reduce across a whole block of threads, reductions must first be performed within each warp. The first thread of each warp can then write its partial sum to an array in shared memory. After thread synchronization, the first warp can read these values from shared memory and perform another warp reduction to give the final value.

To reduce across the whole grid, more than one kernel launch is needed for global communication. A kernel was designed such that the number of values to be reduced for each rigid body was equal to the number of atoms in the largest rigid body in the system, rounded up to the nearest multiple of the block size, so that no block contained a mixture of values from different rigid bodies. The kernel launched a number of threads equal to this number multiplied by the number of rigid bodies. A block reduction was performed for all blocks, with the results written to a temporary array in global memory. Another similar kernel was then used to reduce this array in sections (one section per rigid body). This kernel was configured to launch repeatedly until the number of blocks became equal to the number of large rigid bodies, at which point the reduced value for each block was written to the global memory gradient array.

In a similar manner to the coordinate transformation, a kernel using one thread per rigid body was used to calculate the derivatives of the rotation matrices following eq 9. The nine-element loops were unrolled as before and the final values were written to global memory. Due to higher register use, optimal performance required a smaller block size than that for the coordinate transformation kernel.

The derivatives of the rotation matrices were passed in global memory to another kernel, which performed the matrix multiplication and dot product of eq 7 in parallel for each atom. This result was again saved in global memory and passed to another pair of parallel reduction kernels, similar to the previous implementation of the "Shuffle On Down" algorithm, to obtain the rotational forces on the rigid bodies.

In our tests, we used the RMS force formulation described in ref 31, based on a distance measure for angle-axis coordinates that is invariant to global translation and rotation. This gives a more accurate result than the standard procedure, and fewer minimization steps are required for convergence. This method was adapted for GPUs using techniques similar to those already discussed.

2.2. Transition state determination.

In OPTIM, transition states are usually located using the DNEB algorithm to generate initial guesses between two end points, and then these candidates are refined using hybrid EF. These processes both account for a significant proportion of the computational time spent in constructing a kinetic transition network. As with basin-hopping global optimization, the calculation of the potential energy and gradient accounts for the majority of the time taken in both these procedures. The GPU-accelerated AMBER potential has been interfaced with OPTIM, in a similar manner to its interface with GMIN, to form "CUDAOPTIM". This is the GPU equivalent of the existing A12OPTIM code.

DNEB is a double-ended transition state search method, where paths between end points are represented as a series of images connected by springs.[4] In the nudged elastic band method,[36,37] the parallel component of the true gradient and the perpendicular component of the spring gradient are projected out to avoid sliding down and corner cutting, respectively. In the "doubly nudged" modification of this approach, a portion of the spring gradient perpendicular to the path is retained.[3]

As optimization of the images is not converged tightly and steps are not required to be accurate, a small history size of four is usually used. It is unlikely that implementing the whole algorithm on GPU for DNEB would bring much benefit, so only the calculation of the energy and gradient were performed on the GPU.

In hybrid EF, uphill steps are usually taken along the eigenvector associated with the smallest nonzero Hessian eigenvalue, and minimization is performed in the orthogonal subspace until a transition state is reached.[4] Due to the changing geometry, the uphill eigenvector is recalculated frequently. As second derivatives are not available for the AMBER potential on GPU, a variational approach must be used to calculate the smallest eigenvalue and eigenvector.

Here we minimize a Rayleigh–Ritz ratio

$$\lambda(\mathbf{x}) = \frac{\mathbf{x}^T\mathbf{H}\mathbf{x}}{\mathbf{x}^2} \tag{11}$$

with respect to the eigenvector $\mathbf{x}$ associated with the smallest eigenvalue $\lambda(\mathbf{x})$ of the Hessian, $\mathbf{H}$. $\lambda(\mathbf{x})$ can be approximated as a numerical second derivative of the energy using the central difference approximation

$$\lambda(\mathbf{x}) \approx \frac{V(\mathbf{X} + \zeta\mathbf{x}) + V(\mathbf{X} - \zeta\mathbf{x}) - 2V(\mathbf{X})}{(\zeta\mathbf{x})^2} \tag{12}$$

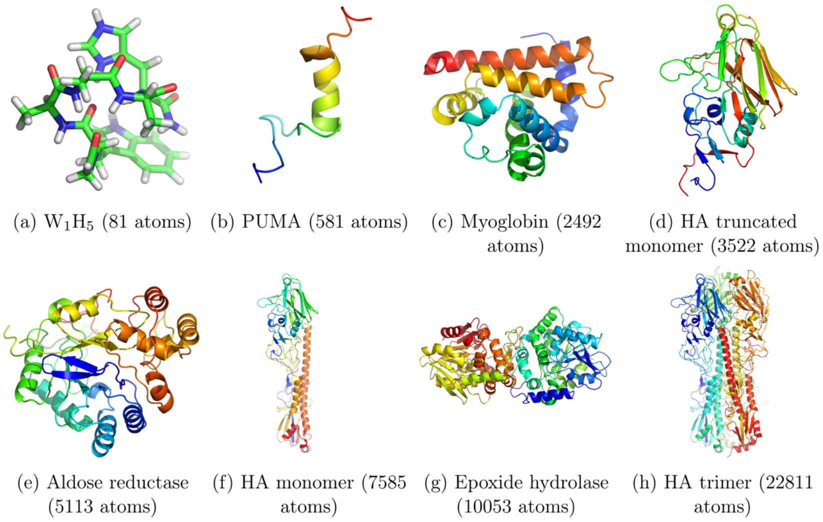
where $V(\mathbf{X})$ is the energy at point $\mathbf{X}$ in nuclear configuration space and $\zeta \ll 1$. Differentiating eq 12 gives

$$\frac{\partial\lambda}{\partial\mathbf{x}} = \frac{\nabla V(\mathbf{X} + \zeta\mathbf{x}) - \nabla V(\mathbf{X} - \zeta\mathbf{x})}{\zeta\mathbf{x}^2} - \frac{2\lambda\mathbf{x}}{\mathbf{x}^2} \tag{13}$$

Roundoff error in eq 12 can result in loss of precision for systems with large values of $V(\mathbf{X})$. An alternative formulation of $\lambda(\mathbf{x})$ using

$$\lambda(\mathbf{x}) \approx \frac{\{\nabla V(\mathbf{X} + \zeta\mathbf{x}) - \nabla V(\mathbf{X} - \zeta\mathbf{x})\}\cdot\mathbf{x}}{2\zeta\mathbf{x}^2} \tag{14}$$

DOI: 10.1021/acs.jctc.6b00934


(a) W_1H_5 (81 atoms) (b) PUMA (581 atoms) (c) Myoglobin (2492 atoms) (d) HA truncated monomer (3522 atoms)

(e) Aldose reductase (5113 atoms) (f) HA monomer (7585 atoms) (g) Epoxide hydrolase (10053 atoms) (h) HA trimer (22811 atoms)

Figure 1. Structures of the biomolecules used in the present tests.

Table 1. L-BFGS Benchmarking (m = 4) Using Two GPU Implementations and a CPU Implementation[a]

System	Number of atoms	Average minimization time for GPU Implementation 1 (m = 4)/ s	Average minimization time for GPU Implementation 2 (m = 4)/ s	Average minimization time for CPU Implementation (m = 4)/ s	Time for CPU Implementation/GPU Implementation 1 (m = 4)	Time for CPU Implementation/GPU Implementation 2 (m = 4)
A	81	2.6	1.2	1.9	0.7	1.6
B	581	5.9	3.4	131.6	22.3	39.2
C	2492	32.8	29.0	4192.1	127.8	144.4
D	3522	40.1	39.4	5937.7	148.1	150.8
E	5113	69.2	71.7	11768.0	169.9	164.2
F	7585	489.9	492.1	86552.8	176.7	175.9
G	10053	1333.4	1374.4	248394.9	186.3	180.7
H	22811	2546.9	2527.1	517567.7	203.2	204.8

[a] GPU Implementation 1 has the entire L-BFGS routine on GPU, including the potential calculation. GPU Implementation 2 has just the potential calculation on GPU. Averages are taken from the minimization of 100 different starting structures from high temperature MD trajectories. The systems used are labeled as follows: "A" is W_1H_5, "B" is PUMA, "C" is myoglobin, "D" is the truncated monomer of HA, "E" is aldose reductase and $NADP^+$, "F" is the full monomer of HA, "G" is epoxide hydrolase, and "H" is the full trimeric HA structure. The ff99SB force field was used for all systems with the modified GB solvent model[47,48] (AMBER input flag igb = 2).

gives a significantly better estimate in these cases, especially when the magnitude of the gradient is small in comparison to the energy.[38]

Components of the eigenvector corresponding to overall translation or rotation are removed by calculating unit vectors for infinitesimal translational and rotational displacements and using these in a projection procedure. The eigenvector is also normalized at every iteration of the minimization.

After the lowest eigenvector has been found, an uphill eigenvector-following step is taken in this direction, with minimization in all orthogonal directions. To prevent minimization in the uphill direction, $\mathbf{x}$, the projection

$$\mathcal{P}\mathbf{G} = \mathbf{G} - (\mathbf{G}\cdot\mathbf{x})\mathbf{x} \tag{15}$$

can be used to remove the component of the gradient along $\mathbf{x}$.

L-BFGS has been found to be faster than the conjugate gradient algorithm for minimization of the Rayleigh–Ritz ratio and also for the subspace minimization,[39] meaning that much of the CUDA code written for basin-hopping global optimization can be reused. $\lambda(\mathbf{x})$ and $\partial\lambda/\partial\,\mathbf{x}$ of eq 12 and eq 13 were calculated using the cuBLAS library. Orthogonalization and normalization of the uphill eigenvector required a mixture of cuBLAS operations and custom kernels with vector length parallelization for more complex vector operations. The determination of the length of the eigenvector-following step took place on CPU, with the actual step itself performed using a cuBLAS call. The projection of gradient components in the subspace minimization, as in eq 15, also used cuBLAS.

3. RESULTS AND DISCUSSION

Here we present results for CUDAGMIN and CUDAOPTIM. Eight different systems of varying sizes (shown in Figure 1) were used to test these methods: the pentapeptide Ac-WAAAH$^+$-NH$_2$ (W_1H_5)[40] (81 atoms), the p53 upregulated modulator of apoptosis (PUMA) protein[41,42] (581 atoms), the myoglobin structure from the AMBER GPU Benchmark Suite[43] (2492 atoms), human aldose reductase with its nicotinamide adenine dinucleotide phosphate ($NADP^+$) cofactor[44] (5113 atoms), an epoxide hydrolase from *Acinetobacter nosocomialis*[45] (10053 atoms), the trimeric hemagglutinin (HA) glycoprotein of the influenza A(H1N1) virus[46] (22811 atoms), a monomeric version of HA (7585 atoms), and finally a truncated version of this monomer (3522 atoms). HA has previously been used to investigate receptor binding. Due to its large size, a truncated monomer structure was employed to make the simulations computationally feasible on CPU. All systems used the AMBER ff99SB force field with an effectively infinite nonbonded cutoff (999.99 Å). Although a

DOI: 10.1021/acs.jctc.6b00934

Table 2. L-BFGS Benchmarking (m = 1000) Using Two GPU Implementations and a CPU Implementation[a]

System	Number of atoms	Average minimization time for GPU Implementation 1 (m = 1000)/s	Average minimization time for GPU Implementation 2 (m = 1000)/s	Average minimization time for CPU Implementation (m = 1000)/s	Time for CPU Implementation/GPU Implementation 1 (m = 1000)	Time for CPU Implementation/GPU Implementation 2 (m = 1000)
A	81	11.3	0.4	0.6	0.1	1.4
B	581	144.5	14.1	92.9	0.6	6.6
C	2492	475.9	201.9	3091.1	6.5	15.3
D	3522	382.1	239.6	4561.2	11.9	19.0
E	5113	421.8	364.4	9123.3	21.6	25.0
F	7585	1249.5	1609.0	48754.5	39.0	30.3
G	10053	2478.7	4217.0	142404.4	57.5	33.8
H	22811	2392.5	4747.1	337077.0	140.9	71.0

[a] Implementation and parameters are identical to those in Table 1.

finite cutoff would be more usual for CPU calculations, this option is not supported on GPU, so an infinite cutoff was used in both cases for a proper comparison. The modified GB solvent model[47,48] (AMBER input flag igb = 2) was used at the salt concentration 0.2 M with a cutoff of 12 Å for the calculation of the effective Born radii. CUDAGMIN and CUDAOPTIM were compiled on the x86_64 architecture running the Ubuntu 14.04.4 operating system, using Intel compiler version 14.0.4 and NVIDIA CUDA Toolkit 6.5.[49] GeForce GTX TITAN Black GPUs with the NVIDIA Linux driver version 346.59 and 2.10 GHz Intel Xeon E5-2620 v2 CPUs were employed. A12GMIN and A12OPTIM were compiled on the x86_64 architecture running the Scientific Linux release 6.2 operating system, using Intel compiler version 14.0.4, and run on 2.67 GHz Intel Xeon X5650 CPUs.

3.1. Basin-hopping global optimization. Basin-hopping global optimization can be viewed in terms of sequential L-BFGS minimizations. Routines other than L-BFGS account for a negligible amount of the overall run time, so it is reasonable to benchmark L-BFGS in isolation so that test minimizations can run concurrently. The calculation of the energy and gradient accounts for the majority of the time spent in L-BFGS, with the percentage of time taken up by linear algebra operations varying with the size of the history of updates, m. As the history size increases, a larger number of vector operations are performed in the sequential loop through the history and fewer L-BFGS steps (and hence fewer potential calls) are needed for convergence, as the directions of the steps taken are more accurate. The optimal history size for a particular system differs between GPU and CPU, so results are presented here for both a small history size of four and a relatively large history size of 1000. For each of the eight systems, 100 configurations were extracted from high temperature MD runs and used as starting structures for minimization. Table 1 shows the average L-BFGS minimization times for these sets of structures for a history size of four. Timings are shown for three different implementations: both the L-BFGS vector operations and the potential function on GPU, L-BFGS on CPU with the potential on GPU, and both L-BFGS and the potential on CPU. The equivalent results for a history size of 1000 are shown in Table 2. All minimizations used a maximum L-BFGS step size of 0.4 Å, with the maximum rise in energy allowed per step capped at 10^{-4} kcal mol^{-1}, and a convergence condition on the root-mean-square (RMS) force of 10^{-5} kcal mol^{-1} Å^{-1}.

Comparing the times shown in Tables 1 and 2 for all three implementations, the CPU-only minimizations are all faster for the larger history size of 1000. However, the calculations where the GPU is used are mostly faster for the smaller history size of four. These CPU results can be explained by the fact that fewer expensive potential calls need to be made for a larger history size. This result is also true for the GPU, but the CPU/GPU speed up ratio for the potential calculation is much greater than that for the vector calculations, so in many cases it is actually fastest to shift the balance toward more potential calls and fewer vector operations, i.e. use a small history size. Comparing times for the two GPU implementations within the same history size, very little difference can be seen in average times for history size m = 4, as vector operations are only a tiny percentage of the overall calculation. However, for history size 1000, where more vector operations are performed, we see that L-BFGS with both the potential and vector calculations on the GPU is slower than the implementation with just the potential calculation on the GPU for small systems, but faster for larger systems. These results occur because BLAS calculations, such as dot products, are actually slower on the GPU relative to CPU for small vectors, as explained in the Methods section. Speed improvements start to be seen for BLAS in the three largest systems. Profiling runs show that cudaMemcpy does not become a significant bottleneck for either GPU implementation, even at these large history sizes, presumably because the potential calculation is sufficiently time-consuming in comparison. Overall, excellent speed ups are obtained for a range of history sizes. The only system for which a significant improvement is not seen is W_1H_5, which is too small for the GPU arithmetic hardware to be fully utilized.[13]

As the speed up for cuBLAS becomes more significant with increasing vector size, it is sensible to optimize the history size to find the fastest balance between the number of potential calls and cuBLAS calls. Referring to Tables 1 and 2, the 22811 atom full trimeric HA system actually minimizes faster with the larger history size of 1000, than for the smaller history size of four. Results are collected in Table 3, showing average minimization times for both GPU implementations for a range of history sizes. Timings for the whole L-BFGS algorithm are faster than those with just the potential on the GPU for all history sizes, except the very smallest. The fastest average minimization time is for the all-GPU implementation with a history size of 75.

3.2. Local rigid body framework. The local rigid body framework[30] is now available on GPU for both CUDAGMIN and CUDAOPTIM. Benchmarking was performed for just the L-BFGS algorithm, as for basin-hopping global optimization. Locally rigid systems were compared to their atomistic equivalents using the implementation of L-BFGS where the whole algorithm is on the GPU. A comparison to CPU timings was not performed, but the average number of L-BFGS steps required was recorded, which will be very similar to that on

Table 3. Variation of HA Minimization Time with History Size Using Two GPU Implementations[a]

History size (m)	Average HA minimization time for GPU Implementation 1/s	Average HA minimization time for GPU Implementation 2/s
4	2546.9	2527.1
10	2035.3	2047.2
50	1817.9	1917.8
75	1788.3	1951.3
100	1836.3	1999.6
250	1880.8	2253.0
500	2010.1	3229.4
750	2223.8	4067.4
1000	2392.5	4747.1

[a] GPU Implementations 1 and 2 are described in Table 1. HA refers to the full trimeric structure ("H" in Tables 1 and 2).

Table 4. GPU Minimization of the HA Monomer for Locally Rigid and Atomistic Representations[a]

	Average minimization time (GPU Implementation 1)/ s		Average number of L-BFGS steps		Average time per step/s	
History size (m)	Rigid	Atomistic	Rigid	Atomistic	Rigid	Atomistic
4	360.6	296.2	19639	16514	0.018	0.018
10	281.6	236.0	15102	13006	0.019	0.018
25	249.2	219.4	12799	11563	0.019	0.019
50	252.4	230.4	12013	11296	0.021	0.020
75	256.9	245.6	11279	11249	0.023	0.022
100	265.5	265.9	10837	11238	0.024	0.024
250	348.8	385.3	9885	11023	0.035	0.035
500	496.9	581.3	9450	10979	0.053	0.053
750	635.7	719.9	9153	10757	0.069	0.067
1000	757.4	890.1	8858	10561	0.085	0.084

[a] GPU Implementation 1 has the entire L-BFGS routine on GPU, including the potential calculation. Averages were taken for the minimization of 100 different starting structures obtained from a basin-hopping run. The atomistic HA monomer is labeled "F" in Tables 1 and 2. The rigid representation has aromatic rings, peptide bonds, and sp^2 centers grouped as local rigid bodies.

Table 5. GPU Minimization of Aldose Reductase for Locally Rigid and Atomistic Representations[a]

	Average minimization time (GPU Implementation 1)/s		Average number of L-BFGS steps		Average time per step/s	
History size (m)	Rigid	Atomistic	Rigid	Atomistic	Rigid	Atomistic
4	335.8	49.2	32431	5224	0.010	0.009
250	110.9	102.9	4155	3999	0.027	0.026
500	92.8	162.1	2200	3890	0.042	0.041
750	87.9	215.5	1739	3827	0.050	0.056
1000	94.7	265.7	1571	3798	0.059	0.069
1500	82.7	347.5	1391	3713	0.057	0.092
2000	78.6	414.6	1350	3691	0.057	0.109

[a] GPU Implementation 1 has the entire L-BFGS routine on GPU, including the potential calculation. Averages were taken for the minimization of 100 different starting structures obtained from a basin-hopping run. Aldose reductase is complexed with the PAC ligand, and the rigidified version is considered for an FSA simulation with an atomistic layer of radius 16 Å; a local rigid layer of 1 Å with rigid aromatic rings, peptide bonds, and sp^2 centers; and the rest of the protein (3678 atoms) rigidifed as a single large rigid body.[44]

CPU. The same parameters were used for L-BFGS as for the atomistic benchmarking, and two systems with different patterns of rigidification were considered. The results in Table 4 are for a system containing a large number of small local rigid bodies, namely the HA monomer, which was used in the previous benchmarking set with aromatic rings, peptide bonds, and sp^2 centers rigidified (a total of 671 rigid bodies, none exceeding a size of 11 atoms). The results in Table 5 are for a system with one large rigid body and a few small rigid bodies, as might be used in a factorized superposition approach (FSA)[44] calculation. The system in question is aldose reductase with aromatic rings, peptide bonds, and sp^2 centers rigidified between 16 and 17 Å distant from the binding site of the phenylacetic acid (PAC) ligand (a total of 22 bodies, none exceeding a size of 11 atoms), and everything beyond 17 Å grouped as one large rigid body of size 3678 atoms. Starting structures for the test minimizations were taken from a basin-hopping run for the locally rigidified structure, with random perturbations at each step consisting of side-chain group rotations and backbone Cartesian moves. The first 100 starting structures to successfully converge formed the benchmarking set. The average minimization times, the average number of L-BFGS steps taken, and the time per step averaged over the whole set are shown in Tables 4 and 5 for a range of history sizes for both rigid and atomistic systems. Different sets of history sizes are used for the two cases, which were chosen to best illustrate the variations observed.

The average time per step for both systems shows that the coordinate and gradient transformations add very little overhead to each step, as they are well optimized for the GPU. As for basin-hopping global optimization, a large history size gives the optimal minimization time on the CPU. It was found in the original work for rigid body systems on CPUs that minimization takes less time than for atomistic representations, as fewer steps are taken due to the reduction in the number of degrees of freedom.[30] Looking at the average number of L-BFGS steps taken for rigid and atomistic systems in Tables 4 and 5, we see that this result holds only for larger history sizes. At small history sizes, rigid body systems actually take more steps than their atomistic equivalents, an effect that is particularly pronounced for the system with FSA rigidification in Table 5. This result implies that rigid body systems require a more accurate step direction in minimization. Rigid body timings only become lower than atomistic timings for history sizes where fewer steps are taken. On a GPU, due to the competing effects of history size, it follows that the optimal history size for fast minimization may be larger for locally rigid systems than for atomistic. This result is certainly true for the highly rigidified system in Table 5, but the average number of L-BFGS steps does not decrease as steeply with increasing history size for the system with minimal rigidification, so the optimal history size is the same as for atomistic. We note that for both systems the fastest minimization is still for the atomistic system with a fairly small history size. It is likely that individual L-BFGS minimizations will only be faster for locally rigid systems on the GPU when the system is sufficiently large and complex that a large history size is required for the atomistic representation. However, excellent speed ups

DOI: 10.1021/acs.jctc.6b00934

Table 6. Hybrid EF Benchmarking (m = 4) for GPU and CPU[a]

System	Number of atoms	Average time for GPU Implementation (m = 4)/s	Average time for CPU Implementation (m = 4)/s	Time for CPU Implementation/GPU Implementation (m = 4)
B	581	14.8	192.1	13.0
C	2492	20.2	2102.4	104.2
D	3522	25.7	3529.1	137.4
E	5113	51.4	7399.4	143.8
F	7585	197.1	36779.8	186.6
G	10053	195.9	33549.2	171.3
H	22811	1047.6	176550.7	168.5

[a] The GPU implementation has the entire Rayleigh−Ritz L-BFGS and L-BFGS routines with gradient projection on GPU, including the potential calculation. 100 DNEB structures that successfully converged to transition states during a connection attempt between two random minima were used as starting points for the searches. The systems used are labeled as follows: "B" is PUMA, "C" is myoglobin, "D" is the truncated monomer of HA, "E" is aldose reductase and NADP$^+$, "F" is the full monomer of HA, "G" is epoxide hydrolase' and "H" is the full trimeric HA structure. The ff99SB force field was used for all systems with the modified GB solvent model[47,48] (AMBER input flag igb = 2).

Table 7. Hybrid EF Benchmarking (m = 1000) for GPU and CPU[a]

System	Number of atoms	Average time for GPU Implementation (m = 1000)/s	Average time for CPU Implementation (m = 1000)/s	Time for CPU Implementation/GPU Implementation (m = 1000)
B	581	113.3	142.7	1.3
C	2492	215.1	1650.3	7.7
D	3522	186.3	3160.9	17.0
E	5113	238.3	6576.0	27.6
F	7585	669.6	32046.2	47.9
G	10053	449.9	30073.3	66.8
H	22811	1225.4	169234.0	138.1

[a] Implementation and parameters are identical to those in Table 6.

compared to the CPU are indeed obtained for locally rigid systems, and the advantage of the significant reduction in minima on the potential energy surface is still present.

3.3. Transition state determination.

Hybrid EF[4] and DNEB[3] optimizations account for the majority of the time spent in determining transition states with the OPTIM program. As explained in the Methods section, gradient-only hybrid EF consists of two different modified versions of L-BFGS with an uphill eigenvector-following step in between. This overall process of finding a transition state starting from coordinates proposed by DNEB was analyzed as a whole, rather than for the separate components. The implementation of L-BFGS with the entire routine on GPU was used in these tests. The same systems were used as for basin-hopping global optimization, excluding the smallest, where too few unique transition states were found. Short basin-hopping runs were performed to generate a distinct second end point for each system. Connection attempts using OPTIM were then performed to try and locate some transition states between the two end points. The first 100 starting coordinates generated by the DNEB procedure, which went on to successfully converge to transition states, were saved as the reference coordinates.

The approximation of eq 14 was used to calculate the eigenvector in all cases. A maximum L-BFGS step size of 0.4 Å with a maximum allowed rise in energy of 10^{-4} kcal mol^{-1} was used in both versions of L-BFGS. The maximum number of iterations in the Rayleigh−Ritz ratio minimization for calculating the smallest nonzero Hessian eigenvalue[4] was set to 1000, and a convergence criterion of 0.01 kcal mol^{-1} Å^{-3} was adopted for the RMS gradient, with the additional constraint that the percentage change of the eigenvalue must have fallen below 1% for the final two steps. The subspace minimization was limited to a maximum of 20 iterations before the eigenvalue converged, as judged by modulus overlap with the previous vector better than 0.9999, and then increased to a maximum of 200 iterations after the eigenvalue converged. A trust radius of 0.5 Å was used for adjusting the size of the maximum uphill step along the eigenvector,[50] with the step size constrained between 0.01 and 0.5 Å. A maximum of 1000 iterations of hybrid EF was allowed. Transition state convergence was deemed to have occurred when the RMS force fell below 10^{-5} kcal mol^{-1} Å^{-1} and the magnitude of the hybrid EF step fell below 0.02 Å. The same initial guess for the eigenvector was used for all 100 starting DNEB structures, and the time taken to converge to a transition state was recorded for both CPU and GPU tests.

Table 6 shows the average time taken for hybrid EF for both GPU and CPU and the corresponding speed ups using a small history size of four. The equivalent results for large history sizes of 1000 are shown in Table 7. Again, the GPU implementation is fastest with a history size of four and the CPU implementation is fastest with a history size of 1000. Excellent speed ups are obtained on GPU hardware, though slightly below those found for basin-hopping global optimization, due to the greater number of dot products and other vector operations performed in projecting out components of the gradient in the subspace minimization.

The analysis of DNEB is fairly straightforward, as there is little variation in time taken for a particular system with a fixed set of parameters. Usually, a fixed maximum number of steps is used, rather than continuing until convergence. In this case, 10 images were employed in the DNEB and the maximum number of iterations used to optimize these was set to 300. A small history size of m = 4 was employed. The convergence tolerance for the RMS DNEB force on all the images was set to 0.01 kcal mol^{-1} Å^{-1}.

DOI: 10.1021/acs.jctc.6b00934

Table 8. DNEB Benchmarking (m = 4) for GPU and CPU[a]

System	Number of atoms	Average time for GPU Implementation (m = 4)/s	Average time for CPU Implementation (m = 4)/s	Time for CPU Implementation/GPU Implementation (m = 4)
B	581	2.8	90.1	32.5
C	2492	10.5	1203.8	114.2
D	3522	18.6	2275.3	122.2
E	5113	30.9	4593.6	148.5
F	7585	59.5	9569.0	160.8
G	10053	100.1	16535.6	165.2
H	22811	452.9	80680.2	178.1

[a] The GPU implementation has just the potential calculation on GPU. The average of three DNEB runs of 300 iterations each was taken from a connection attempt between two random minima. The systems used are the same as those in Tables 6 and 7.

The speed ups are shown in Table 8 along with the average times taken for both CPU and GPU (only the potential is implemented on GPU). These times are an average of those for the first three individual DNEB runs performed in the connection attempts used for generating the DNEB structures for benchmarking hybrid EF. Again, speed ups of more than 2 orders of magnitude have been obtained, providing a significant acceleration for the complete process of transition state location.

4. CONCLUSIONS

This contribution has detailed our implementation of various key components of computational energy landscapes theory on GPU hardware. We first accelerated basin-hopping global optimization in the GMIN code, using a version of L-BFGS adapted for CUDA, and an interface to the GPU-accelerated AMBER potential.[13] These results were then extended to form the basis of a GPU-accelerated version of hybrid eigenvector-following, one component of transition state location in the OPTIM code. DNEB, the other component employed in OPTIM for transition state characterization, was also accelerated using the interfaced potential. Additionally, we adapted our local rigid body framework, which can be be used in either GMIN or OPTIM, for GPU hardware. Tests performed for system sizes in the range of 81 to 22811 atoms gave a speed up relative to CPU of up to 2 orders of magnitude on Titan Black GPUs. Hence, it will now be feasible to explore much larger biological systems than previously accessible using energy landscape methods, opening up a wide array of new applications.

■ AUTHOR INFORMATION

Corresponding Authors

*E-mail: rgm38@cam.ac.uk.
*E-mail: cen1001@cam.ac.uk.
*E-mail: dw34@cam.ac.uk.

ORCID

Rosemary G. Mantell: 0000-0003-4029-3445

Notes

The authors declare no competing financial interest.
Additional data related to this publication is available on Zenodo: https://doi.org/10.5281/zenodo.163633.

■ ACKNOWLEDGMENTS

The authors gratefully acknowledge funding from the EPSRC (grant reference EP/L504920/1).

■ REFERENCES

(1) Li, Z.; Scheraga, H. A. *Proc. Natl. Acad. Sci. U. S. A.* **1987**, *84*, 6611−6615.

(2) Wales, D. J.; Doye, J. P. K. *J. Phys. Chem. A* **1997**, *101*, 5111−5116.

(3) Trygubenko, S. A.; Wales, D. J. *J. Chem. Phys.* **2004**, *120*, 2082−2094.

(4) Munro, L. J.; Wales, D. J. *Phys. Rev. B: Condens. Matter Mater. Phys.* **1999**, *59*, 3969−3980.

(5) Sanders, J.; Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed.; Addison-Wesley Professional: Boston, 2010; Chapter 1, pp 4−6.

(6) *CUDA C Programming Guide.* http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (accessed Sep 21, 2016).

(7) Nocedal, J. *Math. Comput.* **1980**, *35*, 773−782.

(8) Broyden, C. G. *IMA J. Appl. Math.* **1970**, *6*, 222−231.

(9) Fletcher, R. *Comput. J.* **1970**, *13*, 317−322.

(10) Goldfarb, D. *Math. Comput.* **1970**, *24*, 23−26.

(11) Shanno, D. F. *Math. Comput.* **1970**, *24*, 647−656.

(12) *GMIN: A program for finding global minima and calculating thermodynamic properties from basin-sampling.* http://www-wales.ch.cam.ac.uk/GMIN/ (accessed Sep 21, 2016).

(13) Götz, A. W.; Williamson, M. J.; Xu, D.; Poole, D.; Le Grand, S.; Walker, R. C. *J. Chem. Theory Comput.* **2012**, *8*, 1542−1555.

(14) Dennis, J. E., Jr.; Moré, J. J. *SIAM Rev.* **1977**, *19*, 46−89.

(15) Nocedal, J.; Wright, S. J. *Numerical Optimization*, 2nd ed.; Springer: New York, 2006; Chapter 7, pp 176−180.

(16) Haque, I. S.; Pande, V. S. In *GPU Computing Gems: Emerald ed.*, 1st ed.; Hwu, W.-m. W., Ed.; Applications of GPU Computing Series; Morgan Kaufmann: Boston, 2011; pp 19−34.

(17) D'Amore, L.; Laccetti, G.; Romano, D.; Scotti, G.; Murli, A. *Int. J. Comput. Math.* **2015**, *92*, 59−76.

(18) Fei, Y.; Rong, G.; Wang, B.; Wang, W. *Comput. Graph.* **2014**, *40*, 1−9.

(19) Zhang, Q.; Bao, H.; Rao, C.; Peng, Z. *Opt. Rev.* **2015**, *22*, 741−752.

(20) Wetzl, J.; Taubmann, O.; Haase, S.; Köhler, T.; Kraus, M.; Hornegger, J. In *Bildverarbeitung für die Medizin 2013: Algorithmen - Systeme - Anwendungen. Proceedings des Workshops vom 3. bis 5. März 2013 in Heidelberg*; Meinzer, H.-P., Deserno, M. T., Handels, H., Tolxdorff, T., Eds.; Springer: Berlin, Heidelberg, 2013; Chapter GPU-Accelerated Time-of-Flight Super-Resolution for Image-Guided Surgery, pp 21−26.

(21) Yatawatta, S., Kazemi, S., Zaroubi, S. *Innovative Parallel Computing (InPar)*; Institute of Electrical and Electronics Engineers (IEEE): San Jose, CA, 2012; Chapter GPU Accelerated Nonlinear Optimization in Radio Interferometric Calibration, pp 1−6.

(22) Sukhwani, B.; Herbordt, M. C. In *Numerical Computations with GPUs*, 1st ed.; Kindratenko, V., Ed.; Springer International Publishing: Cham, Switzerland, 2014; pp 379−405.

(23) Gates, M.; Heath, M. T.; Lambros, J. *Int. J. High Perform. Comput. Appl.* **2015**, *29*, 92−106.

(24) Gu, J.; Zhu, M.; Zhou, Z.; Zhang, F.; Lin, Z.; Zhang, Q.; Breternitz, M. Implementation and Evaluation of Deep Neural

Networks (DNN) on Mainstream Heterogeneous Systems. *Proceedings of 5th Asia-Pacific Workshop on Systems*, New York, NY, USA, 2014; pp 12:1−12:7.

(25) Rong, G.; Liu, Y.; Wang, W.; Yin, X.; Gu, X.; Guo, X. *IEEE Trans. Vis. Comput. Graphics* **2011**, *17*, 345−356.

(26) Martinez, J.; Claux, F.; Lefebvre, S. *Raster2Mesh: Rasterization based CVT meshing*; [Research Report] RR-8684, 2015; p 27.

(27) cudaLBFGS. https://github.com/jwetzl/CudaLBFGS (accessed Oct 1, 2013).

(28) cuBLAS. https://developer.nvidia.com/cublas (accessed Sep 21, 2016).

(29) OPTIM: A program for optimizing geometries and calculating reaction pathways. http://www-wales.ch.cam.ac.uk/OPTIM/ (accessed Sep 21, 2016).

(30) Kusumaatmaja, H.; Whittleston, C. S.; Wales, D. J. *J. Chem. Theory Comput.* **2012**, *8*, 5159−5165.

(31) Rühle, V.; Kusumaatmaja, H.; Chakrabarti, D.; Wales, D. J. *J. Chem. Theory Comput.* **2013**, *9*, 4026−4034.

(32) Wales, D. J. *Philos. Trans. R. Soc., A* **2005**, *363*, 357−377.

(33) ROTATION. http://www.mech.utah.edu/~brannon/public/rotation.pdf (accessed Sep 21, 2016).

(34) Chakrabarti, D.; Wales, D. J. *Phys. Chem. Chem. Phys.* **2009**, *11*, 1970−1976.

(35) Faster Parallel Reductions on Kepler. https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/ (accessed Sep 21, 2016).

(36) Henkelman, G.; Uberuaga, B. P.; Jónsson, H. *J. Chem. Phys.* **2000**, *113*, 9901−9904.

(37) Henkelman, G.; Jónsson, H. *J. Chem. Phys.* **2000**, *113*, 9978−9985.

(38) Wales, D. J.; Carr, J. M.; Khalili, M.; de Souza, V. K.; Strodel, B.; Whittleston, C. S. In *Proteins: Energy, Heat and Signal Flow*; Leitner, D. M., Straub, J. E., Eds.; *Computation in Chemistry*; CRC Press: Boca Raton, FL, 2010; Vol. *1*; pp 318−319.

(39) Kumeda, Y.; Wales, D. J.; Munro, L. J. *Chem. Phys. Lett.* **2001**, *341*, 185−194.

(40) De Sancho, D.; Best, R. B. *J. Am. Chem. Soc.* **2011**, *133*, 6809−6816.

(41) Yu, J.; Zhang, L.; Hwang, P. M.; Kinzler, K. W.; Vogelstein, B. *Mol. Cell* **2001**, *7*, 673−682.

(42) Day, C. L.; Smits, C.; Fan, F. C.; Lee, E. F.; Fairlie, W. D.; Hinds, M. G. *J. Mol. Biol.* **2008**, *380*, 958−971.

(43) AMBER 12 NVIDIA GPU ACCELERATION SUPPORT: Benchmarks. http://ambermd.org/gpus12/benchmarks.htm (accessed Apr 1, 2014).

(44) Mochizuki, K.; Whittleston, C. S.; Somani, S.; Kusumaatmaja, H.; Wales, D. J. *Phys. Chem. Chem. Phys.* **2014**, *16*, 2842−2853.

(45) Bahl, C. D.; Hvorecny, K. L.; Bridges, A. A.; Ballok, A. E.; Bomberger, J. M.; Cady, K. C.; O'Toole, G. A.; Madden, D. R. *J. Biol. Chem.* **2014**, *289*, 7460−7469.

(46) Chutinimitkul, S.; Herfst, S.; Steel, J.; Lowen, A. C.; Ye, J.; van Riel, D.; Schrauwen, E. J.; Bestebroer, T. M.; Koel, B.; Burke, D. F.; Sutherland-Cash, K. H.; Whittleston, C. S.; Russell, C. A.; Wales, D. J.; Smith, D. J.; Jonges, M.; Meijer, A.; Koopmans, M.; Rimmelzwaan, G. F.; Kuiken, T.; Osterhaus, A. D.; García-Sastre, A.; Perez, D. R.; Fouchier, R. A. *J. Virol.* **2010**, *84*, 11802−11813.

(47) Onufriev, A.; Bashford, D.; Case, D. A. *J. Phys. Chem. B* **2000**, *104*, 3712−3720.

(48) Onufriev, A.; Bashford, D.; Case, D. A. *Proteins: Struct., Funct., Genet.* **2004**, *55*, 383−394.

(49) CUDA Toolkit 6.5. https://developer.nvidia.com/cuda-toolkit-65 (accessed Sep 21, 2016).

(50) Wales, D. J.; Walsh, T. R. *J. Chem. Phys.* **1996**, *105*, 6957−6971.