Using Graphics Processors to Accelerate Protein Docking Calculations

Dave Ritchie
INRIA Nancy – Grand Est

Protein Docking – To Predict Protein-Protein Interactions

• Protein-protein interactions (PPIs) define the “machinery” of life

• Humans have about 30,000 proteins, each having about 5 PPIs
• Understanding PPIs could lead to immense scientific advances
• Controlling PPIs could have huge therapeutic benefits (new drug molecules)

Why is Protein Docking Difficult?

• Protein docking = predicting protein interactions at the molecular level

• If proteins are rigid => six-dimensional search space
• But proteins are flexible => multi-dimensional space!
• Modeling protein-protein interactions accurately is difficult!

Protein Docking Using 3D Grid Correlations

• Conventional approaches digitise proteins into 3D Cartesian grids...

• ...and use FFTs to calculated 3D TRANSLATIONAL correlations:

\[ C[\Delta x, \Delta y, \Delta z] = \sum_{x,y,z} A[x, y, z] \times B[x + \Delta x, y + \Delta y, z + \Delta z] \]

• BUT – have to rotate one protein and REPEAT, which is EXPENSIVE

• Conventional grid-based FFT docking = SEVERAL CPU-HOURS
Protein Docking Using Polar Fourier Correlations

- Rigid body docking can be considered as a largely ROTATIONAL problem
- This means we should use ANGULAR coordinate systems

With FIVE rotations, we should get a good speed-up?

Docking Needs a 3D “Spherical Polar Fourier” Representation

- Need to introduce special orthonormal Laguerre-Gaussian radial functions, \( R_{nl}(r) \)
- \( R_{nl}(r) = N_n e^{-\rho^2/2} L_{n-1}^{(1/2)}(\rho) ; \quad \rho = r^2/q, \quad q = 20. \)

Surface Skin: \( \sigma(\bar{r}) = \begin{cases} 1; & \bar{r} \in \text{surface skin} \\ 0; & \text{otherwise} \end{cases} \)

Interior: \( \tau(\bar{r}) = \begin{cases} 1; & \bar{r} \in \text{protein atoms} \\ 0; & \text{otherwise} \end{cases} \)

Parametrise as: \( \sigma(\bar{r}) = \sum_{n,m}^{N} \sum_{l=-l}^{l} a_{nml} R_{nl}(r) y_{lm}(\theta, \phi) \)

TRANSLATIONS: \( a_{nml}' = \sum_{n',l'}^{N} T_{nl,n'l'}(R) a_{nml} \)

Some Theory – 2D Spherical Harmonic Surfaces

- Use spherical harmonics (SHs) as orthogonal shape “building blocks”

- Reals SHs \( y_{lm}(\theta, \phi) \), and coefficients \( a_{nlm} \)

- Encode distance from origin as SH series to order L:
  \( r(\theta, \phi) = \sum_{l=0}^{L} \sum_{m=-l}^{l} a_{lm} y_{lm}(\theta, \phi) \)

- Calculate coefficients by numerical integration

  ROTATIONS:
  \( a_{nlm}' = \sum_{n',l'}^{N} R_{nl}(\alpha, \beta, \gamma) a_{n'l'm} \)

- Good for shape-matching, not so good for docking...


SPF Protein Shape-Density Reconstruction

Interior density: \( \tau(\bar{r}) = \sum_{n,m}^{N} a_{nlm} R_{nl}(r) y_{lm}(\theta, \phi) \)

<table>
<thead>
<tr>
<th>Image</th>
<th>Order ( N )</th>
<th>Coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>N = 16</td>
<td>1,496</td>
</tr>
<tr>
<td>C</td>
<td>N = 25</td>
<td>5,525</td>
</tr>
<tr>
<td>D</td>
<td>N = 30</td>
<td>9,455</td>
</tr>
</tbody>
</table>

Protein Docking Using SPF Density Functions

Favourable: \[ \int (\sigma_A(\tau_A)\tau_B(\tau_B) + \tau_A(\tau_A)\sigma_B(\tau_B))dV \]

Unfavourable: \[ \int \tau_A(\tau_A)\tau_B(\tau_B)dV \]

Score: \[ S_{AB} = \int (\sigma_A\tau_B + \tau_A\sigma_B - Q\tau_A\tau_B)dV \]

Penalty Factor: \( Q = 11 \)

Orthogonality: \[ S_{AB} = \sum_{nm} (a'_{nlm}b''_{nlm} + a''_{nlm}b'_{nlm} - Qb''_{nlm}) \]

Search: 6D space = 1 distance + 5 Euler rotations: \( (R, \beta_A, \gamma_A, \alpha_B, \beta_B, \gamma_B) \)


Hex Polar Fourier Correlation Example – 3D Rotational FFTs

- Set up 3D rotational FFT as a series of matrix multiplications...

Rotate: \[ a'_{nlm} = \sum_{t=1}^{N} R^{(t)}_{nlm}(0, \beta_A, \gamma_A)a_{t} \]

Translate: \[ a''_{nlm} = \sum_{k} T_{nlm,kj}(R)a'_{kj} \]

Real to complex: \[ A_{nlm} = \sum_{t} a''_{nlm}U^{(t)}_{lm}, \quad B_{nlm} = \sum_{t} b_{nlm}U^{(t)}_{lm} \]

Multiply: \[ C_{muv} = \sum_{nl} A^{*}_{nlm}B_{nlv}A_{nlm}^{*} \]

3D FFT: \[ S(\alpha_B, \beta_B, \gamma_B) = \sum_{muv} C_{muv}e^{-i(m\alpha_B + 2u\beta_B + v\gamma_B)} \]

- On one CPU, docking takes from 15 to 30 minutes
- (1D and 5D FFTs can be calculated in a similar way)

Dateline 2007 – “Hex” and “HexServer”

- Hex: “interactive” protein docking (Linux + Windows + Mac)
- 100,000+ lines of C/C++ (FTLK, OpenGL, MKL, Kiss-FFT, ...)

- Hex: 10,000 down-loads, over 100 citations in bio literature...
- HexServer: 100’s docking jobs per month...
- ... not bad, but can we do better ???

2007 – Dave visits Inria Nancy ...
Nvidia Graphics Processors

- Modern GPUs have very high compute performance
- SIMT architecture = simultaneous instructions, multiple threads
  - NVIDIA GPUs:
    - Up to 4Gb memory
    - Up to 240 arithmetic “cores”
    - Up to Tera-flop performance
    - Easy API with C++ syntax
    - Grid of threads SIMT model
- BUT – for best results, need to understand the hardware...

The CUDA Device Architecture

- Typically 8–16 multi-processor blocks, each with 16 thread units
- NB. only a very small amount of fast shared memory is available
- NB. global memory is ABOUT 80x SLOWER than shared memory

An Alternative View of the CUDA Device Architecture

- Reading and writing global memory is like doing slow I/O
- Strategy: aim for “high arithmetic intensity” in fast shared memory

Slow Devices are Not Well Suited for Random Access

- On the GPU, think of global memory as a SLOW device ...
- ... and that accessing array data “against the grain” is like random access
- This explains why 3D FFTs are SLOW on current GPUs...
- Good strategies:
  - avoid unnecessary “I/O” on global memory
  - make threads cooperate by reading consecutive blocks of global memory linearly
  - do “random access” (e.g. to transpose a matrix) only in shared memory
The CUDA Grid-Block Programming Model

- CUDA implements SIMT using a GRID of BLOCKS of THREADS
- Each THREAD executes a simple "kernel" function
- A BLOCK of related threads all execute the same kernel
- The scheduler launches multiple blocks in parallel, making a GRID of blocks

For example, in matrix arithmetic:
- the matrix is divided into a grid of blocks
- one thread calculates one element of the result

CUDA Matrix Multiplication Kernel – Launching a GPU Kernel

- CUDA adds some programming "extensions" to support the grid-block model
- compile with "nvcc" compiler ...
- (here, we assume matrix dimensions are multiples of 16)

void matmul(float *A, float *B, float *C) {
    dim3 dimBlock(16, 16, 1); // set block size (16x16=256 threads)
    dim3 dimGrid(wB/16, hA/16, 1); // set grid size
    matmul<<<dimGrid, dimBlock>>>(wA, wB, hA, A, B, C); // launch instances of kernel function
    cudaThreadSynchronize(); // wait for kernel to finish
}

CUDA Programming Example – Matrix Multiplication Kernel

void matmul(float *A, float *B, float *C) {
    dim3 dimBlock(16, 16, 1); // set block size (16x16=256 threads)
    dim3 dimGrid(wB/16, hA/16, 1); // set grid size
    matmul<<<dimGrid, dimBlock>>>(wA, wB, hA, A, B, C); // launch instances of kernel function
    cudaThreadSynchronize(); // wait for kernel to finish
}

CUDA Programming Example - Matrix Multiplication

- Matrix multiplication C = A * B
- Each thread is responsible for calculating one element: C[i,k]
- A tile size of 16x16 is just right!
- Conventional algorithm: rows and columns
- A tile size of 16x16 is just right!
- Threads co-operate by reading & sharing tiles of A & B
- Multi-processor launches multiple blocks to compute all of C
- Executing thread-blocks concurrently hides global memory latency

CUDA Programming Example – Matrix Multiplication Kernel

void matmul(int wA, int wB, int hA, int hB, float *A, float *B, float *C) {
    dim3 dimBlock(16, 16, 1); // set block size (16x16=256 threads)
    dim3 dimGrid(wB/16, hA/16, 1); // set grid size
    matmul<<<dimGrid, dimBlock>>>(wA, wB, hA, A, B, C); // launch instances of kernel function
    cudaThreadSynchronize(); // wait for kernel to finish
}

CUDA Programming Example – Matrix Multiplication Kernel

__global__ void matmul(int wA, int wB, float *A, float *B, float *C) {
    float Cik = 0.0; // thread-local result variable
    int bx = blockIdx.x, tx = threadIdx.x; // thread subscripts
    int by = blockIdx.y, ty = threadIdx.y; // ("this" thread is one of a 2-D grid)
    __shared__ float a_sub[16][16], b_sub[16][16]; // declare shared memory
    for (int j=0; j<wA; j+=16) { // thread-local loop over tiles of A and B
        int ij = (16*by+ty)*wA + (j+tx); // thread-local array subscripts
        int jk = (j+ty)*wB + (16*bx+tx); // ("this" thread is one of a 2-D grid)
        a_sub[ty][tx] = A[ij]; // copy global data to shared memory ("I/O")
        b_sub[ty][tx] = B[jk]; // copy global data to shared memory ("I/O")
        __syncthreads(); // wait until all memory I/O has finished
    }
    C[(16*by+ty)*wB + (16*bx+tx)] = Cik; // copy local result -> global memory
}
Hex GPU Docking – Rotate and Translate Protein A

1. On CPU, calculate multiple $\beta_A, \gamma_A$ rotations of protein A
2. On CPU, re-index translation matrices and rotated coefficients into regular sparse arrays
3. On GPU, translate multiple protein A coefficients using tiled matrix multiplication

Hex GPU Docking – Perform Multiple 1D FFTs

- Next, calculate multiple 1D FFTs of the form:
  \[ S_{AB}(\alpha_B) = \sum_{m} e^{-im\alpha_B} \sum_{nl} A_{nlm}^*(R, \beta_A, \gamma_A) \times B_{nlm}^*(\beta_B, \gamma_B) \]

4. On GPU, cross-multiply transformed A with rotated B coefficients (as above)
5. On GPU, perform batch of 1D FFTs using cuFFT and save best orientations

Results – GPU v’s CPU Docking Performance

- Key Hex functions implemented using only 5 or 6 CUDA kernels
- 1D and 3D FFTs are calculated using Nvidia’s cuFFT library
- Here, GPU = Nvidia FX-5800, CPU = Intel i7-965

- Hex 1D correlations are up to 100x faster on FX-5800 than on iCore7
- Overall, including set-up, Hex 1D FFT is about 45x faster on FX-5800 than on iCore7

Results – Multiple GPUs and CPUs

- With Multi-threading, we can use as many GPUs and CPUs as are available

- For best performance: use 2 GPUs alone, or 6 CPUs plus 2 GPUs
- With 2 GPUs, docking takes only about 15 seconds – very important for large-scale!
Results – Speed Comparison with ZDOCK and PIPER

- Hex: 52000 x 812 rotations, 50 translations (0.8 Å steps)
- ZDOCK: 54000 x 6 deg rotations, 92Å 3D grid (1.2 Å cells)
- PIPER: 54000 x 6 deg rotations, 128Å 3D grid (1.0Å cells)
- Hardware: GTX 285 (240 cores, 1.48 GHz)

<table>
<thead>
<tr>
<th></th>
<th>ZDOCK</th>
<th>PIPER</th>
<th>PIPER</th>
<th>Hex</th>
<th>Hex</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kallikrein A / BPTI (233 / 58 residues)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td>1xCPU</td>
<td>1xCPU</td>
<td>1xGPU</td>
<td>1xCPU</td>
<td>4xCPU</td>
<td>1xGPU</td>
</tr>
<tr>
<td>3D</td>
<td>7,172</td>
<td>468,625</td>
<td>26,372</td>
<td>224</td>
<td>60</td>
<td>84</td>
</tr>
<tr>
<td>(3D)</td>
<td>(1.195)</td>
<td>(42,602)</td>
<td>(2,398)</td>
<td>224</td>
<td>60</td>
<td>84</td>
</tr>
<tr>
<td>1D</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>676</td>
<td>243</td>
<td>15</td>
</tr>
</tbody>
</table>

# execution times in seconds
* (times scaled to two-term potential, as in Hex)

Conclusions and Future Prospects

- Protein-protein docking on a GPU now takes only a few seconds:
  - This was implemented using only 5 or 6 GPU kernels
  - But a lot of low-level CPU code had to be re-written

- Was it worth the effort?
  - Yes! Hex with GPU is ultra-fast, and gives very good publicity
  - Yes! the multi-threaded CPU version is now also much faster

- Also, opens door for other high-throughput multi-shape comparison:
  - All-vs-all docking?
  - Full 3D small-molecule virtual screening?

Dateline 2012 – “Hex” and “HexServer”

- Multi-threaded Hex: first (only) docking program to get full benefit of GPUs
- Hex: Over 22,000 down-loads, over 280 citations in bio literature...
- HexServer: About 1,000 docking jobs per month...

Ritchie and Venkatraman (2010) Bioinformatics 26 2398–2405
Macindoe et al. (2010), Nucleic acids Research, 38 W445–W449

Acknowledgments

BBSRC 1996–2000
EPSRC 2000–2006
Nvidia 2008–2012
ANR 2009–2012

Software & Papers: http://hex.loria.fr/
HexServer: http://hexserver.loria.fr/
Exploiting Prior Knowledge in SPF Docking

- Knowledge of even only one key residue can reduce search space enormously...
- This accelerates the calculation and helps to reduce false-positive predictions

5D FFT Correlations from Complex Overlap Expressions
(Report, Kozakov, Vajda, 2008 Bioinformatics, 24, 1865–1873)

Complex SHs, $Y_{lm}$:

$$y_{lm}(\theta, \phi) = \sum_{l} U_{lm}^{(j)} Y_{l}(\theta, \phi)$$

Complex coefficients:

$$A_{nlm} = \sum_{l} a_{nlm} U_{lm}^{(j)}$$

Complex overlap:

$$S = \sum_{l} D_{m}^{(j)}(0, \beta_A, \gamma_A) A_{k}^{*(m)} R_{k,nl}^{(m)}(R) D_{m}^{(l)}(\alpha_B, \beta_B, \gamma_B) B_{nl}$$

Collect coefficients:

$$S_{j,l}^{(m)}(R) = \sum_{k} A_{k}^{*(m)} R_{k,nl}^{(m)}(R) B_{nl}, \quad k > j; n > l$$

To give:

$$S = \sum_{j,l} D_{m}^{(j)}(0, \beta_A, \gamma_A) S_{j,l}^{(m)}(R) D_{m}^{(l)}(\alpha_B, \beta_B, \gamma_B)$$

Expand as exponentials:

$$D_{m}^{(l)}(\alpha, \beta, \gamma) = \sum_{l} \Gamma_{lm}^{(j)} e^{-i\alpha m n} e^{-i\beta r} e^{-i\gamma}$$

Hence:

$$S = \sum_{j,l} \Gamma_{lm}^{(j)} S_{j,l}^{(m)}(R) \Gamma_{lm}^{(l)} e^{-i(\alpha m n + \beta r + \gamma l)}$$

Translation Matrices From Fourier-Bessel Transform Theory

Using spherical Bessel transforms:

$$\tilde{R}_{nl}(\beta) = \sqrt{\frac{2}{\pi}} \int_{0}^{\infty} R_{nl}(r) j_{l}(\beta r) r^{2} dr; \quad R_{nl}(r) = \sqrt{\frac{2}{\pi}} \int_{0}^{\infty} \tilde{R}_{nl}(\beta) j_{l}(\beta r) \beta^{2} d\beta$$

it can be shown that

$$T_{nm}^{(j)} R (R) = \sum_{k=|l-1|}^{l+1} A_{k}^{(lm)} \tilde{R}_{nm}^{(k)}(\beta) j_{k}(\beta R) \beta^{2} d\beta$$

where

$$A_{k}^{(lm)} = (-1)^{l+1} \frac{(2l+1)(2l+3)}{4l+1} \left( \begin{array}{ccc} l+1 & l+1 & l+1 \\ 0 & 0 & 0 \end{array} \right) (l+1)$$

- Can derive analytic formulae for both GTO and ETO radial functions
- Requires high precision math library (GMP)...
- Calculate once for $R = 1, 2, 3, ... 50 \AA$ and store on disk ($\sim 200$ Mb)
Inside Hex – High Order FFTs and GPUs

- The SPF gives an analytic way to calculate TRANSLATIONAL + ROTATIONAL correlations:

In particular:

\[ S_{AB} = \sum_{js_{x_{s_{x_{x}}}}v_{s_{x_{x_{x}}}}} A_{s_{x_{x}}r_{s_{x_{x_{x}}}}}^{r_{s_{x_{x_{x}}}}}(R) A_{s_{x_{x}}r_{s_{x_{x_{x}}}}}^{r_{s_{x_{x_{x}}}}}(e^{-i(r_{s_{x_{x}}}-\alpha_{s_{x_{x}}})}) \]

- This allows high order FFTs to be used – 1D, 3D, and 5D
- It also allows calculations to be easily ported to modern GPUs
  - Up to 240 arithmetic “cores”
  - Grid of threads SIMT model
  - Correlation speed-up \( \geq 100x \)
  - Overall speed-up = 45x
- GPU docking takes 15 seconds (475x faster than ZDOCK) – very important for large-scale!

D.W. Ritchie, et al. HealthGrid (2010), To Appear
D.W. Ritchie, V. Venkatraman (2010), In review