CHAPTER

Application case study—molecular visualization and analysis 15

John Stone

CHAPTER OUTLINE

The previous case study used a statistical estimation application to illustrate the process of selecting an appropriate level of a loop nest for parallel execution, transforming the loops for reduced memory access interference, using constant memory for magnifying the memory bandwidth for read-only data, using registers to reduce the consumption of memory bandwidth, and the use of special hardware functional units to accelerate trigonometry functions. In this case study, we use a molecular dynamics application based on regular grid data structures to illustrate the use of additional practical techniques that achieve global memory access coalescing and improved computation throughput. As we did in the previous case study, we present a series of implementations of an electrostatic potential map calculation kernel, with each version improving upon the previous one. Each version adopts one or more practical techniques. Some of the techniques are in common with the previous case study but some are new: systematic reuse of computational results, thread granularity coarsening, and fast boundary condition checking. This application case study shows that the effective use of these practical techniques can significantly improve the execution throughput of the application.

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00015-7

15.1 BACKGROUND

This case study is based on VMD (Visual Molecular Dynamics) [HDS 1996], a popular software system designed for displaying, animating, and analyzing bio-molecular systems. VMD has more than 200,000 registered users. It is an important foundation for the modern "computational microscope" for biologists to observe the atomic details and dynamics of tiny life forms such as viruses that are too small for traditional microscopy techniques. While it has strong built-in support for analyzing bio-molecular systems such as calculating maps of the electrostatic field that surround a molecular system, it has also been a popular tool for displaying other large data sets such as sequencing data, quantum chemistry calculations, and volumetric data due to its versatility and user extensibility.

While VMD is designed to run on a diverse range of hardware—laptops, desktops, clusters, and supercomputers—most users use VMD as a desktop science application for interactive 3D visualization and analysis. For computation that runs too long for interactive use, VMD can also be used in a batch mode to render movies for later use. A motivation for accelerating VMD is to make batch mode jobs fast enough for interactive use. This can drastically improve the productivity of scientific investigations. With CUDA devices widely available in PCs, such acceleration can have broad impact on the VMD user community. To date, multiple aspects of VMD have been accelerated with CUDA, including electrostatic potential map calculation, ion placement (HSS 2009), calculation and display of molecular orbitals (SSHVHS 2009), molecular surfaces (KSES 2012), radial distribution histograms (LSK 2011), and electron density map quality-of-fit (SMIS 2014), and high fidelity ray tracing of large biomolecular complexes for conventional and panoramic displays (SVS 2013, Stone et al. 2016), and virtual reality headsets (SSS 2016).

The particular calculation used in this case study is the calculation of electrostatic potential maps in 3D grids with uniform spacing. This calculation is often used in placement of ions into a molecular structure for molecular dynamics simulation. Fig. 15.1 shows the placement of ions into a protein structure in preparation for molecular dynamics simulation. In this application, the electrostatic potential map is used to identify spatial locations where ions (red dots) can fit in according to physical laws. The function can also be used to calculate time-averaged potentials during molecular dynamics simulation, which is useful for the simulation process as well as the visualization/analysis of simulation results.

There are several methods for calculating electrostatic potential maps. Among them, Direct Coulomb Summation (DCS) is a highly accurate method that is particularly suitable for GPUs [SPF 2007]. The DCS method calculates the electrostatic potential value of each grid point as the sum of contributions from all atoms in the system. This is illustrated in Fig. 15.2. The contribution of atom i to a lattice point j is the charge of atom i divided by the distance from lattice point j to atom i. Since this needs to be done for all grid points and all atoms, the number of calculations is proportional to the product of the total number of atoms in the system and the total number of grid points. For a realistic molecular system, this product can be very large. Therefore, the calculation of the electrostatic potential map had been traditionally done as a batch job in VMD.

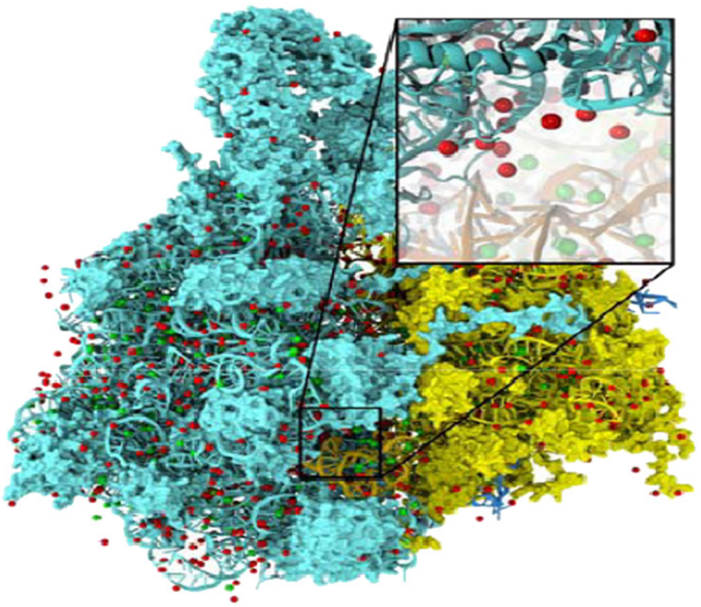


FIGURE 15.1

Electrostatic potential map is used in building stable structures for molecular dynamics simulation.

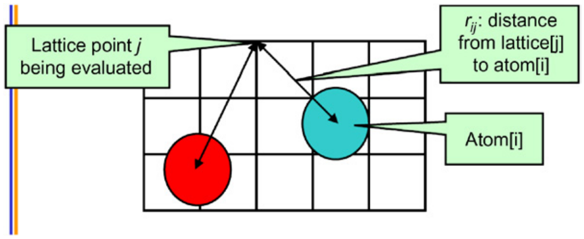

Lattice point *j* being evaluated
r_{ij}: distance from lattice[j] to atom[i]
Atom[i]

FIGURE 15.2

The contribution of atom[i] to the electrostatic potential at lattice point *j* (potential[j]) is atom[i] charge/r_{ij}. In the Direct Coulomb Summation method, the total potential at lattice point *j* is the sum of contributions from all atoms in the system.

15.2 A SIMPLE KERNEL IMPLEMENTATION

Fig. 15.3 shows the base C code of the DCS code. The function is written to process a two-dimensional (2D) slice of a three-dimensional (3D) grid. The function will be called repeatedly for all the slices of the modeled space. The structure of the function is quite simple with three levels of `for` loops. The outer two levels iterate over the *y*-dimension and the *x*-dimension of the grid point space. For each grid point, the innermost for loop iterates over all atoms, calculating the contribution of electrostatic potential energy from all atoms to the grid point. Note that each atom is represented by four consecutive elements of the `atoms[]` array. The first three elements store the *x*, *y*, and *z* coordinates of the atom and the fourth element the electrical charge of the atom. At the end of the innermost loop, the accumulated value of the grid point is written out to the grid data structure. The outer loops then iterate and take the execution to the next grid point.

Note that DCS function in Fig. 15.3 calculates the *x* and *y* coordinates of each grid point on the fly by multiplying the grid point index values by the spacing between grid points. This is a uniform grid method where all grid points are spaced at the same distance in all three dimensions. The function does take advantage of the fact that all the grid points in the same slice have the same *z* coordinate. This value is precalculated by the caller of the function and passed in as a function parameter (`z`).

Based on what we learned from the MRI case study, two attributes of the DCS method should be apparent. First, the computation is massively parallel: the computation of electrostatic potential for each grid point is independent of that of other grid points. As we have seen in the previous case study, there are two alternative approaches to organizing parallel execution. In the first option, we can use each thread to calculate the contribution of one atom to all grid points. This would be a poor choice since each thread would be writing to all grid points, requiring extensive use of atomic memory operations to coordinate the updates done by different threads to each

```
void cenergy(float *energygrid, dim3 grid, float gridspacing, float z, const float *atoms,
                    int numatoms) {
  int i,j,n;
  int atomarrdim = numatoms * 4;
  for (j=0; j<grid.y; j++) {
    float y = gridspacing * (float) j;
    for (i=0; i<grid.x; i++) {
      float x = gridspacing * (float) i;
      float energy = 0.0f;
      for (n=0; n<atomarrdim; n+=4) {    // calculate potential contribution of each atom
        float dx = x - atoms[n    ];
        float dy = y - atoms[n+1];
        float dz = z - atoms[n+2];
        energy += atoms[n+3] / sqrtf(dx*dx + dy*dy + dz*dz);
      }
      energygrid[grid.x*grid.y*k + grid.x*j + i] = energy;
    }
  }
}
```

FIGURE 15.3

Base Coulomb potential calculation code for a 2D slice.

grid point. The second option uses each thread to calculate the accumulated contributions of all atoms to one grid point. This is a preferred approach since each thread will be writing into its own grid point and there is no need to use atomic operations.

We will form a 2D thread grid that matches the 2D potential grid point organization. In order to do so, we need to modify the two outer loops into perfectly nested loops so that we can use each thread to execute one iteration of the two-level loop. We can either perform a loop fission (as we did in the previous case study), or we move the calculation of the *y* coordinate into the inner loop. The former would require us to create a new array to hold all *y* values and result in two kernels communicating data through global memory. The latter increases the number of times that the *y* coordinate will be calculated. In this case, we choose to perform the latter since there is only a small amount of calculation that can be easily accommodated into the inner loop without significant increase in execution time of the inner loop. The amount of work to be absorbed into the inner loop is much smaller than that in the previous case study. The former would have added a kernel launch overhead for a kernel where threads do very little work. The selected transformation allows all *i* and *j* iterations to be executed in parallel. This is a tradeoff between the amount of calculation done and the level of parallelism achieved.

The second experience that we can apply from the MRI case study is that the electrical charge of every atom will be read by all threads. This is because every atom contributes to every grid point in the DCS method. Furthermore, the values of the atomic electrical charges are not modified during the computation. This means that the atomic charge values can be efficiently stored in the constant memory (in the GPU box in Fig. 15.4).

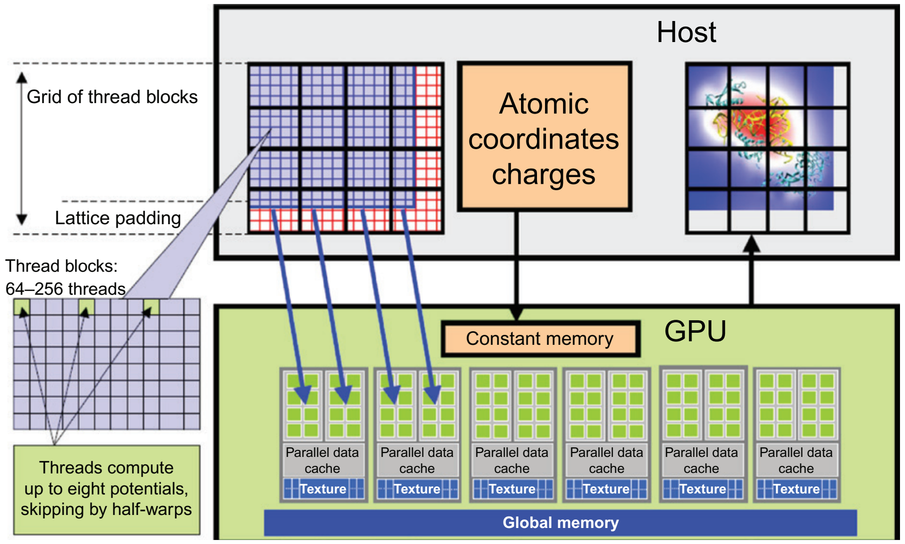


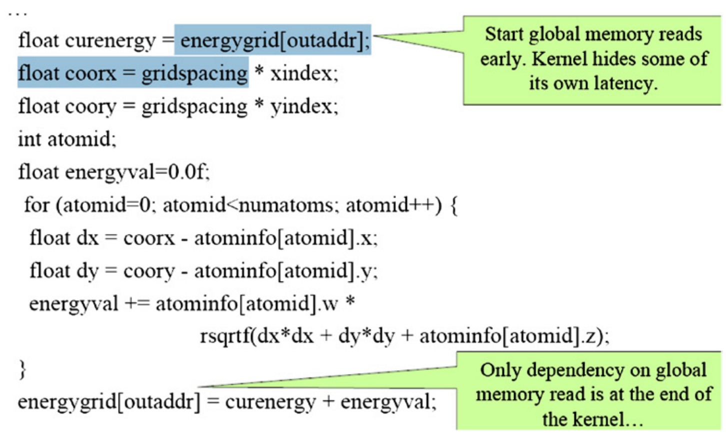
FIGURE 15.4

Overview of the DCS kernel design.

Fig. 15.4 shows an overview of the DCS kernel design. The host program (shown as the Host box) inputs and maintains the atomic charges and their coordinates in the system memory. It also maintains the grid point data structure in the system memory. The DCS kernel is designed to process a 2D slice of the electrostatic potential grid point structure (not to be confused with thread grids). The right-hand side grid in the Host box shows an example of a 2D slice. For each 2D slice, the CPU transfers its grid data to the device global memory. Similar to the *k*-space data, the atom information is divided into chunks to fit into the constant memory. For each chunk of the atom information, the CPU transfers the chunk into the device constant memory, invokes the DCS kernel to calculate the contribution of the current chunk to the current slice, and prepares to transfer the next chunk. After all chunks of the atom information have been processed for the current slice, the slice is transferred back to update the grid point data structure in the CPU system memory. The system moves on to the next slice.

Within each kernel invocation, the thread blocks are organized to calculate the electrostatic potential of tiles of the grid structure. In the simplest kernel, each thread calculates the value at one grid point. In more sophisticated kernels, each thread calculates multiple grid points and exploits the redundancy between the calculations of the grid points to improve execution speed. This is illustrated in the left-hand side portion labeled as "Thread blocks" in Fig. 15.4 and is an example of the granularity adjustment optimization discussed in Chapter 5, Performance Considerations.

Fig. 15.5 shows the resulting CUDA kernel code. We omitted some of the declarations. As was in the MRI case study, the `atominfo[]` array is declared in the constant memory by the host code. The host code divides up the atom information into chunks that fit into the constant memory for each kernel invocation. This means that kernel will be invoked multiple times when there are multiple chunks of atoms. Since this is similar to the MRI case study, we will not show the details.



```
...
float curenergy = energygrid[outaddr];
float coorx = gridspacing * xindex;
float coory = gridspacing * yindex;
int atomid;
float energyval=0.0f;
 for (atomid=0; atomid<numatoms; atomid++) {
  float dx = coorx - atominfo[atomid].x;
  float dy = coory - atominfo[atomid].y;
  energyval += atominfo[atomid].w *
                       rsqrtf(dx*dx + dy*dy + atominfo[atomid].z);
}
energygrid[outaddr] = curenergy + energyval;
```

Start global memory reads early. Kernel hides some of its own latency.

Only dependency on global memory read is at the end of the kernel…

FIGURE 15.5

DCS kernel version 1.

The outer two levels of the loop in Fig. 15.3 have been removed from the kernel code and are replaced by the execution configuration parameters in the kernel invocation. Since this is also similar to one of the steps we took in the MRI case study, we will not show the kernel invocation but leave it as an exercise for the reader. The rest of the kernel code is straightforward and corresponds directly to the original loop body of the innermost loop.

One particular aspect of the kernel is somewhat subtle and worth mentioning. The kernel code calculates the contribution of a chunk of atoms to a grid point. The grid point must be stored in the global memory and updated by each kernel invocation. This means that the kernel needs to read the current grid point value, add the contributions by the current chunk of atoms, and write the updated value to global memory. The code attempts to hide the global memory latency by loading the grid value at the beginning of the kernel and using it at the end of the kernel. This helps to reduce the number of warps needed by the SM scheduler to hide the global memory latency.

The performance of the kernel in Fig. 15.5 is quite good. However, there is definitely room for improvement. A quick glance over the code shows that each thread does nine floating-point operations for every four memory elements accessed. On the surface, this is not a very good ratio. We need a ratio of 10 or more to avoid global memory congestion. However, all four memory accesses are done to `atominfo[]` array. These `atominfo[]` array elements for each atom are cached in a hardware cache memory in each SM and are broadcast to a large number of threads. A calculation similar to that in the MRI case study shows that the massive reuse of memory elements across threads makes the constant cache extremely effective, boosting the effective ratio of floating operations per global memory access much higher than 10. As a result, global memory bandwidth is not a limiting factor for this kernel.

15.3 THREAD GRANULARITY ADJUSTMENT

Although the kernel in Fig. 15.5 avoids global memory bottlenecks through constant caching, it still needs to execute four constant memory access instructions for every nine floating-point operations performed. These memory access instructions consume hardware resources that could be otherwise used to increase the execution throughput of floating-point instructions. More importantly, the execution of these memory access instructions consumes energy, an important limiting factor for many large scale parallel computing systems. This section shows that we can fuse several threads together so that the `atominfo[]` data can be fetched once from the constant memory, stored into registers, and used for multiple grid points.

We observe that all grid points along the same row have the same *y*-coordinate. Therefore, the difference between the *y*-coordinate of an atom and the *y*-coordinate of any grid point along a row has the same value. In the DCS kernel version 1 in Fig. 15.5, this calculation is redundantly done by all threads for all grid points in a row when calculating the distance between the atom and the grid points. We can eliminate this redundancy and improve the execution efficiency.

The idea is to have each thread calculate the electrostatic potential for multiple grid points. The kernel in Fig. 15.7 has each thread calculate four grid points. For each atom, the code calculates `dy`, the difference of the y-coordinates, in line 2. It then calculates the expression `dy*dy` plus the pre-calculated `dz*dz` information and saves it to the auto variable `dysqpdzsq`, which is assigned to a register. This value is the same for all four grid points. Therefore, the calculation of `energyvalx1` through `energyvalx4` can all just use the value stored in the register. Furthermore, the electrical charge information is also accessed from constant memory and stored in the automatic variable `charge`. Similarly, the *x*-coordinate of the atom is also read from constant memory into auto variable `x`. Altogether, this kernel eliminates three accesses to constant memory for `atominfo[atomid].y`, three accesses to constant memory for `atominfo[atomid].x`, three accesses to constant memory for `atominfo[atomid].w`, three floating-point subtraction operations, five floating-point multiply operations, and nine floating-point add operations when processing an atom for four grid points. A quick inspection of the kernel code in Fig. 15.7 shows that each iteration of the loop performs four constant memory accesses, five floating-point subtractions, nine floating-point additions, and five floating-point multiplications for four grid points.

The reader should also verify that the version of DCS kernel in Fig. 15.5 performs 16 constant memory accesses, 8 floating-point subtractions, 12 floating-point additions, and 12 floating-point multiplications, a total of 48 operations for the same four grid points. Going from Figs. 15.5 to 15.7, there is a total reduction from 48 operations down to 25 operations, a sizable reduction. This is translated into about 40% increased execution speed and about the same percentage reduction in energy consumption.

The cost of the optimization is that more registers are used by each thread. This can potentially reduce the number of threads that can be accommodated by each SM. However, as the results show, this is a good tradeoff with an excellent performance improvement.

15.4 MEMORY COALESCING

While the performance of the DCS kernel version 2 in Fig. 15.7 is quite high, a quick profiling run reveals that the threads perform memory writes inefficiently. As shown in Figs. 15.6 and 15.7, each thread calculates four neighboring grid points. This seems to be a reasonable choice. However, as we illustrate in Fig. 15.8, the write pattern of adjacent threads in each warp will result in un-coalesced global memory writes.

There are two problems that cause the un-coalesced writes in DCS kernel version 2. First, each thread calculates four adjacent neighboring grid points. Thus, for each statement that accesses the `energygrid[]` array, the threads in a warp are not accessing adjacent locations. Note that two adjacent threads access memory locations that are three elements apart. Thus, the 16 locations to be written by all the threads in warp write are spread out, with three elements in between the loaded/written locations.

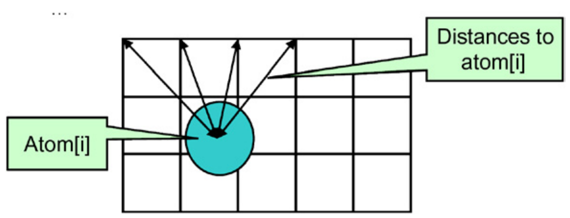
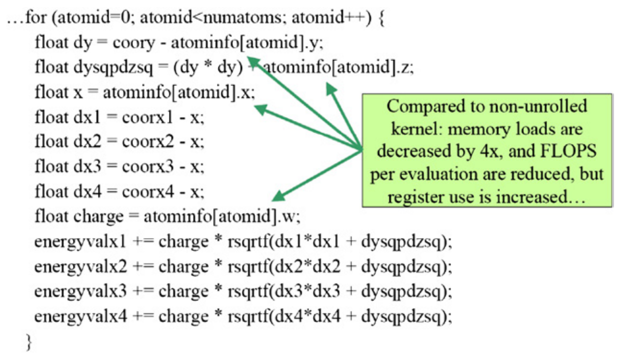


FIGURE 15.6

Reusing computation results among multiple grid points.



```
...for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dysqpdzsq = (dy * dy) + atominfo[atomid].z;
    float x = atominfo[atomid].x;
    float dx1 = coorx1 - x;
    float dx2 = coorx2 - x;
    float dx3 = coorx3 - x;
    float dx4 = coorx4 - x;
    float charge = atominfo[atomid].w;
    energyvalx1 += charge * rsqrtf(dx1*dx1 + dysqpdzsq);
    energyvalx2 += charge * rsqrtf(dx2*dx2 + dysqpdzsq);
    energyvalx3 += charge * rsqrtf(dx3*dx3 + dysqpdzsq);
    energyvalx4 += charge * rsqrtf(dx4*dx4 + dysqpdzsq);
  }
```

Compared to non-unrolled kernel: memory loads are decreased by 4x, and FLOPS per evaluation are reduced, but register use is increased...

FIGURE 15.7

Version 2 of the DCS kernel.

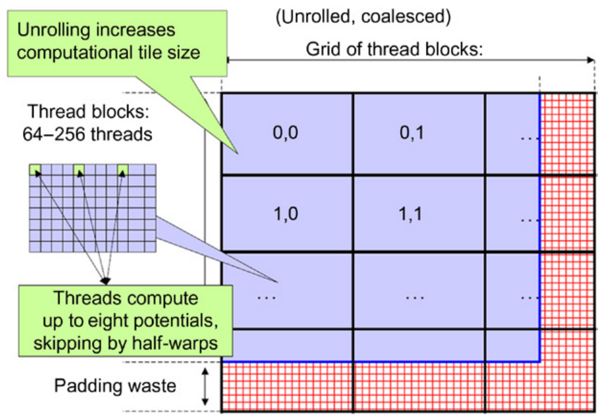
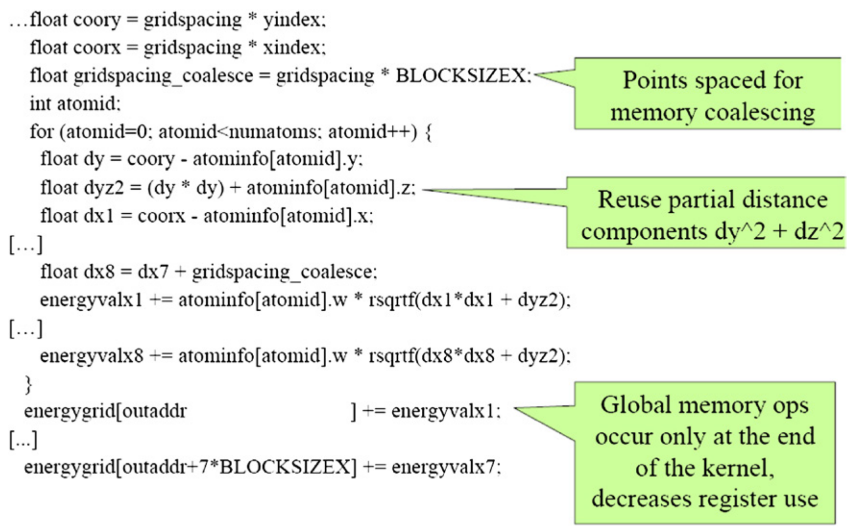

Unrolling increases computational tile size
(Unrolled, coalesced)
Grid of thread blocks:
Thread blocks: 64–256 threads
0,0
0,1
...
1,0
1,1
...
...
...
...
Threads compute up to eight potentials, skipping by half-warps
Padding waste

FIGURE 15.8

Organizing threads and memory layout for coalesced writes.



```
...float coory = gridspacing * yindex;
   float coorx = gridspacing * xindex;
   float gridspacing_coalesce = gridspacing * BLOCKSIZEX;
   int atomid;
   for (atomid=0; atomid<numatoms; atomid++) {
     float dy = coory - atominfo[atomid].y;
     float dyz2 = (dy * dy) + atominfo[atomid].z;
     float dx1 = coorx - atominfo[atomid].x;
[...]
     float dx8 = dx7 + gridspacing_coalesce;
     energyvalx1 += atominfo[atomid].w * rsqrtf(dx1*dx1 + dyz2);
[...]
     energyvalx8 += atominfo[atomid].w * rsqrtf(dx8*dx8 + dyz2);
  }
  energygrid[outaddr                    ] += energyvalx1;
[...]
  energygrid[outaddr+7*BLOCKSIZEX] += energyvalx7;
```

Points spaced for memory coalescing
Reuse partial distance components dy^2 + dz^2
Global memory ops occur only at the end of the kernel, decreases register use

FIGURE 15.9

DCS kernel version 3.

This problem can be solved by assigning adjacent grid points to adjacent threads in each half-warp. Most previous generation devices form coalesced memory accesses on a half-warp basis. Assuming that we still want to have each thread calculate four grid points, we first assign 16 consecutive grid points to the 16 threads in a half-warp. We then assign the next 16 consecutive grid points to the same 16 threads. We repeat the assignment until each thread has the number of grid points desired. This assignment is illustrated in Fig. 15.8. For more recent devices, the number of threads in a coalesced access has increased to 32. Thus, we may need to assign the grid points on a full-warp basis so that the grid points to be processed by each thread will be 32 grid points apart from each other.

The kernel code with coarsened thread granularity and warp-aware assignment of grid points to threads is shown in Fig. 15.9. Note that the *x*-coordinates used to calculate the atom-to-grid-point distances for a thread's assigned grid points are offset by the value of the variable `gridspacing_coalesce`, which is the original grid-spacing times the constant `BLOCKSIZEX` (set as 16). This reflects the fact that the *x*-coordinates of the 8 grid points assigned to a thread are 16 grid points away from each other. Also, after the end of the loop, memory writes to the `energygrid` array are indexed by `outaddr, outaddr+BLOCKSIZEX, …, outaddr+7*BLOCKSIZEX`. Each of these indices is one `BLOCKSIZEX` (16) away from the previous one. The detailed thread block organization for this kernel is left as an exercise.

The other cause of un-coalesced memory writes is the layout of the `energygrid` array, which is a 3D array. If the *x*-dimension of the array is not a multiple of the half-warp size (16), the beginning location of the row 1, as well as those of the subsequent rows will no longer be at the 16-word boundaries. For example, if the `energygrid` array starts at location 0 and the *x*-dimension has 1000 elements, row 1 of the array will start at location 1000, which is not a 16-word boundary. The nearest 16-word boundaries are 992 and 1008. Therefore, starting at row 1, the accesses to the `energygrid` by threads in a half-warp will span two 16-word units in the global memory address space.

In some devices, this means that the half-warp accesses will not be coalesced, even though they write to consecutive locations. This problem can be corrected by padding each row with additional elements so that the total length of the *x*-dimension is a multiple of 16. This can require adding up to 15 elements, or 60 bytes to each row, as shown in Fig. 15.8. In our example, since the *x*-dimension has 1000 elements, we need to pad 8 elements at the end of each row so that the number of words in each row is a multiple of 16 (1008).

If we want to avoid `if`-statements for handling boundary conditions, we will need to make the *x*-dimension a multiple of the number of grid points processed by each thread block. Each block has 16 threads in the *x*-dimension. With the kernel of Fig. 15.9, the number of elements in the *x*-dimension needs to be a multiple of $8 \times 16 = 128$. This is because each thread actually writes eight elements in each iteration. Thus, one may need to pad up to 127 elements, or 1016 bytes to each row. In our example, the nearest multiple of 128 that we can pad from 1000 is 1024. Therefore, we will need to pad 24 elements at the end of each row to avoid adding if-statements for handling the boundary condition.

Finally, there is a potential problem with the last row of thread blocks. Each thread block is 16×16 so there are 16 threads in the *y* dimension. Since the number of rows grid array may not be a multiple of 16, some of the threads may end up writing outside the grid data structure without adding if-statements to handle the boundary conditions. Since the grid data structure is a 3D array, these threads will write into the next slice of grid points. As we discussed in Chapter 3, Scalable parallel execution, we can add a test in the kernel and avoid writing the array elements that are out of the known *y*-dimension size. However, this would have added a number of overhead instructions and incurred control divergence. An alternative solution is to pad the *y*-dimension of the grid structure so that it contains a multiple of tiles covered by thread blocks. This is shown in Fig. 15.8 as the bottom padding in the grid structure. In general, one may need to add up to 15 rows due to this padding.

The cost of padding can be substantial for smaller grid structures. For example, if the potential energy grid has 100×100 grid points in each 2D slice, it would be padded into a 128×112 slice. The total number of grid points increases from 10,000 to 14,336, or a 43% overhead. At such overhead, one should consider much less coarsening. On the other hand, for a 1000×1000 grid, one will need to pad it to $1024 \times 1008 = 1{,}032{,}192$ T, or 3.2% overhead. This makes it very cost-effective to assign eight grid points to each thread. This is the reason why high-performance libraries often have multiple kernels for the same type of computation. When the user calls the library function, the interface would choose the version according to the size and shape of the actual data set.

If we had to pad the entire 3D structure, the grid points would have increased from 100 × 100 × 100 (1,000,000) to 128 × 112 × 112 (1,605,632), or a 60% overhead! This is part of the reason why we calculate the energy grids in 2D slices and use the host code to iterate over these 2D slices. Writing a single kernel to process the entire 3D structure would have incurred a lot more extra overhead. This type of tradeoff appears frequently in simulation models, differential equation solvers, and video processing applications. Decomposing the problem into individual 2D slices also allows multiple GPUs to be used concurrently, with each GPU computing independent slices.

The DCS version 3 kernel shown in Fig. 15.9 achieves about 535.16 GFLOPS or 72.56 billion atom evaluations per second on a Fermi GPU. On a recent GeForce GTX 680 (Kepler 1), it achieves a whopping 1267.26 GFLOPS or 171.83 billion atom evaluations per second! This measured speed of the kernel also includes a slight boost from moving the read access to the energygrid array from the beginning of the kernel to the end of the kernel. The contribution to the grid points are first calculated in the loop. The code loads the original grid point data after the loop, adds the contribution to them, and writes the updated values back. Although this movement exposes more of the global memory latency to each thread, it saves the consumption of eight registers. Since the version 3 kernel is using many registers to hold the atom data and the distances, such savings in number of registers used relieve a critical bottleneck for the kernel. This allows more thread blocks to be assigned to each SM and achieved an overall performance improvement.

15.5 SUMMARY

The relative merit of the three versions of the DCS kernel depends on the dimension lengths of the potential energy grid. However, the DCS version 3 (CUDA-Unroll8clx) will perform consistently better than all others once the grid dimension length is sufficiently large, say 300 × 300 or more.

A detailed comparison of between the sequential code performance on a CPU and the CPU–GPU joint performance shows a commonly observed tradeoff. Fig. 15.10 shows plot of the execution time of a medium-sized grid for varying number of atoms to be evaluated. For 400 atoms or fewer, the CPU performs better. This is because the particular GPU used has a fixed initialization overhead of 110 ms regardless of the number of atoms to be evaluated. Also, for a small number of atoms, the GPU is underutilized, thus the curve of the GPU execution time is quite flat between 100 atoms and 1000 atoms.

The plot in Fig. 15.10 reinforces a commonly held principle that GPUs perform better for large amounts of data. Once the number of atoms reaches 10,000, the GPU is fully utilized. The slope of the CPU and the CPU–GPU execution time becomes virtually identical, with the CPU–GPU execution being consistently 44 × times faster than the sequential CPU execution for all input sizes.

While DCS is a highly accurate method for calculating the electrostatic potential energy map of a molecular system, it is not a scalable method. The number of operations to be performed of the method grows proportionally with the number of

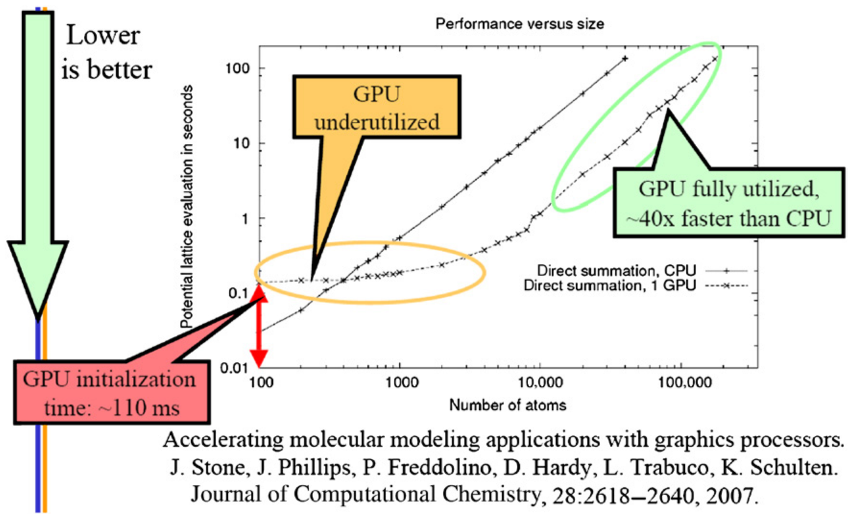


FIGURE 15.10

Single-threads CPU versus CPU–GPU comparison.

atoms and the number of grid points. When we increase the physical volume of the molecular system to be simulated, we should expect that both the number of grid points and the number of atoms to increase proportional to the physical size. As a result, the number of operations to be performed will be approximately proportional to the square of the physical volume. That is, the number of operations to be performed will grow quadratically with the volume of the system being simulated. This makes the use of DCS method not suitable for simulating realistic biological systems. Therefore, one must devise a method whose number of operations grows linearly with the volume of the biological systems being simulated. We will revisit this topic in Chapter 17, Parallel programming and computational thinking.

15.6 EXERCISES

1. Complete the implementation of the DCS kernel as outlined in Fig. 15.5. Fill in all of the missing declarations. Give the kernel launch statement with all the execution configuration parameters.

2. Compare the number of operations (memory loads, floating-point arithmetic, branches) executed in each iteration of the kernel in Fig. 15.7 compared to that in Fig. 15.5. Keep in mind that each iteration of the former corresponds to four iterations of the latter.

3. Complete the implementation of the DCS kernel version 3 in Fig. 15.9. Explain in your own words how the thread accesses are coalesced in this implementation.

4. For the memory padding in Fig. 15.8 and DCS kernel version 3 in Fig. 15.9, show why one needs to pad up to 127 elements in the *x* dimension but only up to 15 elements in the *y* dimension.
5. Give two reasons for adding extra "padding" elements to arrays allocated in the GPU global memory, as shown in Fig. 15.8.
6. Give two potential disadvantages associated with increasing the amount of work done in each CUDA thread, as shown in Section 15.3.

REFERENCES

Humphrey, W., Dalke, A., & Schulten, K. (1996). VMD—Visual molecular dyanmics. *Journal of Molecular Graphics*, *14*, 33–38.

Hardy, D. J., Stone, J. E., & Schulten, K. (2009). Multilevel Summation of Electrostatic Potentials Using Graphics Processing Units. *Parallel Computing*, *28*, 164–177.

Krone, M., Stone, J. E., Ertl, T., & Schulten, K. (2012). Fast Visualization of Gaussian Density Surfaces for Molecular Dynamics and Particle System Trajectories. EuroVis-Short Papers, pp. 67–71.

Levine, B. G., Stone, J. E., & Kohlmeyer, A. (2011). Fast Analysis of Molecular Dynamics Trajectories with Graphics Processing Units—Radial Distribution Function Histogramming. *Journal of Computational Physics*, *230*(9), 3556–3569.

Stone, J. E., McGreevy, R., Isralewitz, B., & Schulten, K. (2014). GPU-Accelerated Analysis and Visualization of Large Structures Solved by Molecular Dynamics Flexible Fitting. *Faraday Discussions*, *169*, 265–283.

Stone, J. E., Phillips, J. C., Freddolino, P. L., Hardy, D. J., Trabuco, L. G., & Schulten, K. (2007). Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, *28*, 2618–2640.

Stone, J.E., Saam, J., Hardy, D.J., Vandivort, K.L., Hwu, W., & Schulten, K. (2009). High Performance Computation and Interactive Display of Molecular Orbitals on GPUs and Multi-core CPUs. In Proceedings of the 2nd Workshop on General-Purpose Processing on Graphics Processing Units, ACM International Conference Proceeding Series, *383*, 9–18.

Stone, J. E., Sener, M., Vandivort, K. L., Barragan, A., Singharoy, A., Teo, I., … Schulten, K. (2016). Atomic Detail Visualization of Photosynthetic Membranes with GPU-Accelerated Ray Tracing. *Journal of Parallel Computing*, *55*, 17–27.

Stone, J. E., Sherman, R., & Schulten, K. (2016). Immersive Molecular Visualization with Omnidirectional Stereoscopic Ray Tracing and Remote Rendering. High Performance Data Analysis and Visualization Workshop, 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1048–1057.

Stone, J. E., Vandivort, K. L., & Schulten, K. (2013). GPU-Accelerated Molecular Visualization on Petascale Supercomputing Platforms. UltraVis'13: Proceedings of the 8th International Workshop on Ultrascale Visualization, pp. 6:1–6:8.