Optimizing a Lattice Boltzmann Code for the Xeon Phi

Carlos Rosales and Kent Milfeld

{milfeld,carlos}@tacc.utexas.edu

LBM: Code Structure

Local calculation. Most of the math takes place here.

Boundary corrections for outward f values

Move collision data along velocity directions

Boundary corrections for inward f and g

Original Code

- Multiphase LBM based on Free Energy formulation
- Single 4D array for f and single 4D array for g
- Double-buffered to avoid issues in Stream
- Parallelized using MPI
- Some common optimizations present
  - Fused collision-stream step for g
  - Neighbors calculated instead of read from array
  - Bidirectional MPI exchanges used

Setting Expectations

- Performance ratios between one mic and two E5-2680 sockets:
  - In FP Ops : 1074 / 345 = 3.1x
  - In Memory BW (approx) : 180 / 80 = 2.2x
- So, the best acceleration I can expect is:
  - 3.1x for fp bound code
  - 2.2x for memory bw bound code
- And most likely I will get something in between these two results.

Code Port

- Parallelized outermost loop (in z direction)
  - Innermost loop not broken – vectorization
  - Must be careful with number of threads
    - When using multiple threads per core, if zmax / OMP_NUM_THREADS not a multiple of the number of cores performance will be degraded
- Performance was lacklustre:
  - 1 Phi ~ 1.25 E5-2680 sockets (SNB)

VTUNE Bandwidth Reports

- OK, I cheated. I know the algorithm , so I know it is BW heavy…
- And I checked with Vtune using one of the predefined collections: knc-bandwidth

- The code was using a total of 74.4 GB/s
- That is 46.5% or the 160 GB/s achievable by us, mere mortals, on a SE10P
- That is not particularly good. An algorithm like this should be using more bandwidth that that (in the host Sandy Bridge CPUs the code achieves over 97% of the available node memory BW)

- The memory access pattern of the code must be terribly inefficient
  - This could be in part due to all the effort the compiler is putting into vectorizing operations that are likely pulling data from non-contiguous locations in memory – lots of gather operation.
- A change to SOA may improve things

Why Change to SOA?

- Having two 5D arrays to store the data is inefficient in terms of data reads because each time an element of the array is accessed most of the additional cache line data read is discarded
- The hope is that by having individual velocity arrays we will throw away less data each time we execute a read
- And also that the compiler will be able to vectorize the main loops with a smaller setup overhead

From AOS to SOA

$$g(i,j,k,vel,buf) \rightarrow g_0(m), g_1(m), \ldots, g_{18}(m) \quad m = i + NX \cdot (j + NY \cdot k) + offset$$
$$f(i,j,k,vel,buf) \rightarrow f_0(m), f_1(m), \ldots, f_6(m) \quad m = i + NX \cdot (j + NY \cdot k) + offset$$

- This maintains a doubly-buffered array but in a much simpler form
- New code uses higher BW: 105.2 GB/s (~41% higher)

- Phi performance increases by 2x
- Phi is now 2.5x FASTER than a dual socket CPU node

- Does all this improved performance come from the increased BW? Unlikely. More later…

AOS to SOA Pain

- I made all changes manually
- The process is prone to errors because of the 5D nature of the original arrays, particular when applying boundary conditions.
- Long and tedious, multiple mistakes had to be corrected along the way
- Later on, still more mistakes had to be addressed for the hybrid version

- Worth it? Yes – The code is now setup to take advantage of the next generation of processors.
- Alternatives: While I have not used it, the Kokkos library claims that the whole AOS/SOA issue can be avoided. For an alternative point of view:

https://github.com/kokkos/

Vectorization Efficiency

Code	-O3 -mmic	-O3 –mmic –no-vec	Ratio
SOA	112 MLUPS	26 MLUPS	4.3X
AOS	57 MLUPS	24 MLUPS	2.4X

- Intel 15 gives estimates between 3 and 5 speedup for vectorization of the main loops in the code
- Excellent agreement between measured and predicted improvement
- Reported ratio is across whole program execution, and some sections are certainly not vectorizable due to dependencies.
- Some sections are not vectorized at all for the AOS version
- In this case SOA allows for additional vectorization and reduces overhead

MLUPS=M-Lattice update/s

Latency Sensitivity Study

- Using Vtune Amplifier XE 2013 Update 7

Code	CPI Rate	Max BW (GB/s)	L1 Hit Ratio (%)	Latency Impact
SOA	8.142	105.2	95.6	1018.7
AOS	8.373	74.4	92.6	455.1

- So we may have moved our issue from a problem with achieving a significant fraction of the available BW to a more latency-bound problem!
- One of the main issues affecting the performance of this code is that in either SOA or AOS there are over 50 independent memory streams in flight
- The sheer number of memory streams prevents effective prefetching and
- Solutions to this problem involve a significant refactoring of the code, which we are reticent to make right now