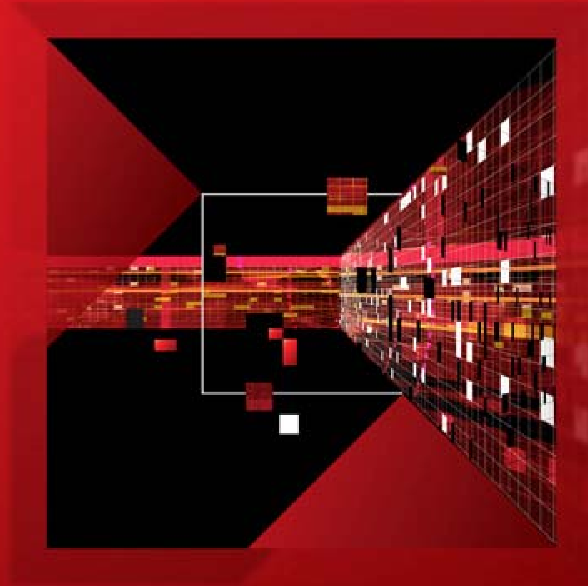
Fusion 11

DEVELOPER SUMMIT

QUANTUM MONTE CARLO ANALYSIS OF BOSE-EINSTEN CONDENSATION

Ivar Ursin Nikolaisen
University of Oslo
Student

THE RANLUX RANDOM NUMBER GENERATOR



WHY WE NEED GOOD PSEUDO-RANDOM NUMBERS

- Monte Carlo algorithms rely on truly random, pseudo-random or quasi-random numbers.
- We do a random walk that samples an integral.
- Bad numbers can give bad results, but can be difficult to detect.
- One sequence of random numbers per work-item.
- Sequences of each work-item must be good, but there should also be no correlations between different work-items.

AMD Fusion11 DEVELOPER SUMMIT

CREATING A GOOD PSEUDO-RANDOM NUMBER GENERATOR IS DIFFICULT

- RANDU: An infamous example of a bad generator.
  - Widespread on IBM mainframes and other systems for many years.
  - An anecdote from Numerical Recipes describes someone noticing that the “random” numbers were generating plots falling in 11 planes, and being told: “We guarantee that each number is random individually, but we don’t guarantee that more than one of them is random.”

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

(courtesy of xkcd.com)

AMD Fusion11 DEVELOPER SUMMIT

SEVERAL CANDIDATES | ran1

- ran1 is an example of a linear congruential generator (LCG) that still sees some use.
- The good:
  - Simple.
  - Fast.
- The bad:
  - Very short period (~10^9). Modern GPU can exhaust it in less than a second.
  - Large state array, especially for such a short period (33 values per generator).
  - No parallel safe theory?

AMD Fusion11 DEVELOPER SUMMIT

SEVERAL CANDIDATES | Mersenne Twister

- A very popular generator.
- The good:
  - Potentially fast.
  - Very long period (depending on state array size).
  - Extensively used and tested by others.
  - Has (complicated) parallel safe theory (not provably good, but likely works well).
- The bad:
  - Fails some statistical tests (linear complexity tests).
  - Very large state array.
    - Can be reduced, but can we then rely on the good track record of the larger state array version?
  - Complicated to implement.
    - Either reduce state array size, or share state array in an entire work-group, necessitating synchronization points.

AMD Fusion11 DEVELOPER SUMMIT

SEVERAL CANDIDATES | RANLUX

- The good:
  - Potentially fast.
  - Very long period (~10^170).
  - Extensively used and tested by others.
  - Has theory for why it is good, both independently and in parallel.
  - No known flaws (passes all known statistical tests).
- The bad:
  - Somewhat large state array (7 float4 variables per work-item in my implementation).

AMD Fusion11 DEVELOPER SUMMIT

SEVERAL CANDIDATES | RANLUX – Interesting Properties

- Has a theory describing why it produces good pseudo-random values.
  - The generator can be viewed as a discrete approximation to a chaotic system.
  - Adjacent values are correlated, but the correlations are exponentially decreasing with distance.
- The theory allows us to select a tradeoff between speed and quality.
  - By discarding values we get fewer correlations. The speed/quality tradeoff is handled through a “luxury” value of 0-4.
    - Even with moderate discarding (luxury value 2) a version that glues two numbers together to form a 48-bit number passed all tests in the TestU01 suite of statistical tests.
  - By discarding enough values all bits of the random numbers can be considered chaotic.
- Some nice consequences.
  - Parallel sequences (different parts of the full period) should not be correlated.
  - Since we can view each generated bit as chaotic we can “glue” values together to create as large pseudo-random numbers as we desire.

AMD Fusion11 DEVELOPER SUMMIT

RANLUXCL | OpenCL Implementation of RANLUX

- C/C++ host code function to initialize the states of parallel generators/work-items.
- Generator proper implemented in OpenCL C, straightforward to use.
- Implementation is compatible with (generates same sequences as) reference fortran77 code.
- Available from https://bitbucket.org/ivarun/ranluxcl/.
- Example kernel code to generate a random vector:

```
#include "ranluxcl.cl"
__kernel void example(__global float4 *ranluxcltab){
    ranluxcl_state_t ranluxclstate;
    ranluxcl_download_seed(&ranluxclstate, ranluxcltab);
    float4 randomvec = ranluxcl (&ranluxclstate);
    ranluxcl_upload_seed(&ranluxclstate, ranluxcltab);
}
```


AMD Fusion11 DEVELOPER SUMMIT

THE BOSE-EINSTEIN CONDENSATE SOLVER

PROBLEM TO BE SOLVED

- Bose Einstein condensation in a collection of trapped neutral atoms.
  - Bosons can occupy the same quantum state as other bosons (fermions can't).
  - Bose-Einstein condensation is a state of matter where a large fraction of the molecules in a gas are in the ground state at the same time. It is achieved by cooling the gas to near absolute zero temperature.
- Particles are trapped in a 3D harmonic oscillator potential.
- Want to find the ground state energy and particle distribution.

METROPOLIS MONTE CARLO ALGORITHM

- Basic outline of the algorithm.

  Generate initial random particle positions

    For 0 to number of Monte Carlo cycles

      For 0 to number of particles

        Move the particle

        Accept or reject move

      Calculate total system energy

- Since we're just gathering statistics we can run the algorithm in parallel.
  - Each work-item is an independent system, with its own set of particles etc.
  - Very simple massively parallel algorithm, no local memory use.
- Parallelize across multiple devices using MPI.

AMD Fusion11 DEVELOPER SUMMIT

TWO APPROACHES

- VMC (Variational Monte Carlo)
  - Need a trial wave function (our best guess) with some parameters we can vary.
  - Vary the parameters to find the lowest energy.
  - The lowest energy we find is an upper bound on the actual ground state energy (no trial wave function can give lower energy than the real ground state wave function).
- DMC (Diffusion Monte Carlo)
  - Same basic algorithm as used for VMC.
    - But we now need several walkers (work-items) in parallel. With VMC we didn't need it (but used it anyway to parallelize the computation).
    - Some walkers are deleted, others multiplied for each Monte Carlo cycle.
  - Don't need a trial wave function (but a good guess can still help).
  - Can find the actual ground state energy and particle distribution with no knowledge of the ground state wave function!

AMD Fusion11 DEVELOPER SUMMIT

PERFORMANCE

- As always, *fair* comparisons are difficult.
- Comparisons done to *similar* C++ implementations.
  - Same overall algorithm, except that C++ implementations sometimes use lookup tables to increase performance.
- Initial learning of OpenCL and quick port took only a few evenings, resulting in ~60 times speedup compared to original single-threaded C++ code.
- Performance of OpenCL implementation on CPU is similar to C++ (actually somewhat faster perhaps because of OpenCL vector types yielding better SSE code).
- Performance on AMD Cypress (HD 5870) is 100-200 times that of single threaded C++ implementations.
- Double precision performance on AMD Cypress surprisingly good.
  - About 65 % of single precision performance. Why not less?
    - Slightly more efficient use of memory controllers in this case.
    - Higher ALU busy and packing values.
    - Small kernels means no register spilling.

CONCLUSIONS

- Can’t make claim that GPU is always this much faster, and there is always the question of further optimizations.
- However learning OpenCL is not terribly difficult, and writing standard, straightforward OpenCL code can give excellent results on both CPUs and GPUs.
- This case simply further demonstrates that there is low-hanging fruit to be had.
  - No complicating factors with difficult memory access patterns or local memory.
  - Many Monte Carlo simulations have similar algorithms, and OpenCL / GPGPU is therefore likely equally suited for those as well.

Disclaimer & Attribution

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.

NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

ALL IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. All other names used in this presentation are for informational purposes only and may be trademarks of their respective owners.

The contents of this presentation were provided by individual(s) and/or company listed on the title page. The information and opinions presented in this presentation may not represent AMD’s positions, strategies or opinions. Unless explicitly stated, AMD is not responsible for the content herein and no endorsements are implied.