UppASD GPU Project Documentation

Niklas Fejes

August 16, 2013

Summary

The goal of this project has been to implement the basic simulation parts of UppASD on Nvidia's parallel computing platform CUDA. The CUDA implementation is based on UppASD 3.0 and uses UppASD's Fortran code for setting up the system and writing output files, while all calculations are made on the GPU through functions written in CUDA C/C++.

Implemented Parts

The simulation model that has been implemented so far is the simplest model available in UppASD, which is based on the Landau-Lifshitz-Gilbert equations using the Heisenberg Hamiltonian without additional terms:

$$\mathscr{H} = -\frac{1}{2} \sum_{i,j,i \neq j} J_{ij} \, \mathbf{M}_i \cdot \mathbf{M}_j$$

The UppASD code contains several different time evolution algorithms to choose from, but only Depondt's method has been implemented in the CUDA code. However, additional algorithms should be straightforward to implement using the code for Depondt's method as a base.

Program Structure

As the CUDA code is based on the original Fortran code, most of the variable and function names have been kept the same or similar in the new implementation. The structure of the simulation loop is also the same, and all data necessary for the simulations are copied to device memory when the simulations are initiated.

The code also contains a C/C++ implementation (non-CUDA) of the code that served as a predecessor to the CUDA implementation.

For future development the structure of the simulation loop could be made "cleaner" by moving the different moment vectors into the classes they belong to (this has been made for the neighbour list for example), and by using virtual classes for the Hamiltonian and integrator to make it easier to implement other algorithms.

Profiling

There are various ways to profile the program, where the most accurate one is by using `nvprof` or Nvidia Visual Profiler (`nvvp`). (Both requires CUDA compute capability 2.0 or higher.) The code also contains a stopwatch class that may be used instead, which will time the simulation with respect to the different simulation steps. However, using this will prevent asynchronous GPU work, and the total run time will increase. It is possible to disable this GPU synchronization which can be useful to detect unwanted synchronization. For example, with asynchronous stopwatch enabled and "fast copy" disabled, the measurements will appear to take up most of the simulation time as `cudaMemcpy` has to wait for all GPU work to be done before it can copy the data.

Possible Improvements

Reordering of Atoms

The `Heisge` kernel typically takes about 60-70% of the total simulation time, so for future improvements the focus should be put on making this part of the code faster.

The biggest bottleneck in the current implementation is that the GPU for each atom has to read the moments of other atoms that are spread out at different locations in the memory.

When the GPU reads data (from global GPU memory to the multiprocessor) reads from adjacent threads can be coalesced if they read the same or adjacent data. In the `Heisge` kernel, (`HeisgeJijElement` in `cudaHamiltonianCalculations.cu`) each warp (see CUDA guide) processes $10\frac{2}{3}$ atoms (32 threads with one atoms axis each), so if all of those atoms have many neighbours that are the same the warp can under optimal conditions read each neighbour once for all atoms, while it otherwise has to read each neighbour for each atom.

By reordering the atoms and/or the order of the neighbours in the neighbour list so that reads are made optimally, the Heisge kernel should become significantly faster. By changing the order of the reads so that memory alignment is optimal (without taking into account that the the setup is wrong) the `Heisge`-kernel becomes about 15% faster.

If the atoms in each block/warp has enough common neighbours it might also be beneficial to read the moments of the common neighbours to shared memory and then process them.

Spin Transfer Torque

The Fortran code that the CUDA implementation is based on has support for Spin transfer torque (`btorque`), which is only partially implemented in the CUDA code. Those parts that are available are not tested and will not work without changes.

Measurement Queue Size

A potential problem with the current code is that the program dynamically allocates memory for the moments that are queued up for measurement. If the measurements takes more time than the simulation and there are many measurement points this might cause problems. Each queued data takes up [$56 \times$ atoms $\times$ ensembles] bytes of memory, so for big systems which queues a lot of data the system might run out of memory and crash the simulation. For robustness a limit on the measurement queue size should be implemented.

Kernel / Parallelization Code Structure

Since most of the CUDA code is parallelized in the same way the code is written so that all CUDA kernels share the same code base, which is located in the `CudaParallelizationHelper` class. This has the advantage that it becomes easier to write new parallelized functions, and that changes made to the common code applies to all parallelizations in the program.

A simple example of the parallelization is the `CudaCommon::Add` class:

```
class CudaCommon::Add : public CudaParallelizationHelper::Element {
    real *       a;
    const real * b;
    const real * c;
public:
    // Constructor
    Add(cudaMatrix<real,3,3> &A, const cudaMatrix<real,3,3> &B,
            const cudaMatrix<real,3,3> &C) {
        a = A;
        b = B;
        c = C;
    }
    // device "each" function
    __device__ void each(unsigned int element) {
        a[element] = b[element] + c[element];
    }
};
```

Here, the `each` function is called on the GPU once for each element in the matrix, and performs the element-wise addition $A = B + C$.

This code can be called with the following syntax:

```
CudaParallelizationHelper parallel = ...;
cudaMatrix<real,3,3> A = ..., B = ..., C = ...;

parallel.cudaElementCall(CudaCommon::Add(A, B, C));
```

Depending on the parameters needed for the parallelization, there are various base classes in the `CudaParallelizationHelper` class. these are:

`Atom`, `Site`, `AtomSite`, `AtomSiteEnsemble`, `ElementAxisSiteEnsemble`, `Element`

Subclasses of these should have an `each` function with parameters according to the classes name, e.g. subclasses of `AtomSite` should have a function `__device__ each(unsigned int atom, unsigned int site)`. These classes also provides the `N` and `M` system size limits, corresponding to Fortrans `Natom` and `Mensemble`. The difference between `Atom` and `Site` in these names are that `Atom` represents all $N \times M$ atoms in all ensembles, while `Site` represents the N different sites in one ensemble.

Performance Comparision

Performance comparisions has been made on the UppASD example system “bccFe” with 8192 ($16 \times 16 \times 16$ cell) atoms, and 8 ensembles:

System	Simulation Time
Intel(R) Core(TM) i5 CPU (4 cores at 2.67GHz)	201.6 s
GeForce GTX 670 GPU	29.5 s

Compilation Parameters

There are various parameters that can be set in the `Makefile` to change the behaviour of the program. Note that for these to take effect the program must be rebuilt. The parameters are added to the compiler through additional definitions that can be enabled by uncommenting lines in the `Makefile`, or adding `-D<FLAG>` to the compiler parameters.

`DEBUG`: Enables error checking in the matrix class and classes deriving from it.

`MATRIX_ERROR_INTERRUPT`: Interrupts the program if an error occurs in a matrix. Can be used with `gdb` to detect where in the code an error occured.

`DUMMY_STOPWATCH`: Disables the stopwatch.

`ASYNC_STOPWATCH`: Normally the stopwatch waits for the GPU to finish all its work before measuring the time. This behaviour can be disabled with this flag.

`USE_BIG_GRID`: With this flag the code does not use CUDA's built in 2-D and 3-D block indexing and instead uses one big 1-D block that is splits up in the kernels. Enabling this flag seems to generate slightly faster code.

`USE_FAST_COPY`: This flag enables utilization of asynchronous kernel work and device to host memory copy, which may increase speed if the system is large. With this flag, the simulation loop never has to synchronize with the GPU and thus allows the CPU to "work ahead" of the GPU. If the GPU would have an unlimited call stack this would mean that all calculations could be queued up at the start of the program, and most of the calculations would be performed while the main thread waits in the final device synchronization. This method requires slightly more memory that are page locked, which according to the CUDA Best Practices Guide may "reduce overall system performance". The fast copy also relies on "`cudaCallbacks`", which sometimes seems to block the GPU work without reason if the CPU is under high workload.

`SINGLE_PREC`: Makes the program use single precision (`floats`) instead of double precision. This also requires that the Fortran part of UppASD uses single precision.

`NVPROF`: Adds profiling information for Nvidia Visual Profiler so that the progress of the measurements can be visualized.

Simulation Parameters

Three additional simulation parameters have been added that controls the GPU implementation:

`gpu_mode`: Determines what code to use. (0=Fortran, 1=CUDA (default), 2=C/C++)

`gpu_rng`: Determines what RNG to use. (0=CURAND Default (default), 1=XORWOW, 2=MRG32K3A, 3=MTGP32)

`gpu_rng_seed`: The seed to use for the RNG. (if 0 the seed is initialize with `time()` (default))

Files

Helper files

`cudamake.mk`

Contains the CUDA-specific make flags and rules.

`stopwatch.hpp stopwatchPool.cpp/hpp stopwatchDeviceSync.hpp stopwatch_fortran.cpp`

Different utilities for measuring time.

`matrix.hpp fortMatrix.hpp cudaMatrix.hpp hostMatrix.hpp`

Matrix classes for different types of memory. The matrix data is stored in column-major order (Fortran style).

`real_type.h`

Defines the "`real`" data type which is either `double` (64-bit) or `float` (32-bit) depending on `SINGLE_PREC`.

`c_helper.f90 c_helper.h`

Helper functions for calling Fortran code from C/C++ code.

`fort_helper.cpp`

Helper subroutines for calling C/C++ code from Fortran.

`cudaParallelizationHelper.cu/hpp`

A helper class that simplifies parallelization over atoms and provides a common base for all CUDA parallelized methods.

`gridHelper.hpp`

Class for splitting up the grid over the atoms. Used by `cudaParallelizationHelper.cu`.

`fortranData.cpp/hpp`

Class for interfacing Fortrans data members with C/C++.

`cudaCommon.hpp`

Various parallelized vector/matrix operations that may be useful for multiple classes.

`cudaEventPool.cu/hpp`

Provides a "pool" of CUDA events so that events may be reused between time steps without having to synchronize with the GPU.

CUDA Algorithm Files

`cudaMdSimulation.cu/hpp`

Contains the simulation loop and handles the initialization of the simulation. This class corresponds to the `sd_mphase()` routine in `0sd.f90`.

`cudaHamiltonianCalculations.cu/hpp`

Contains the Hamiltonian calculation algorithm. (See `applyhamiltonian.f90`)

`cudaDepondtIntegrator.cu/hpp`

Contains the Depondt integrator algorithm. (See `depondt.f90`)

`cudaMomentUpdater.cu/hpp`

Contains the moment updater. (See `updatemoments.f90`)

`cudaMeasurement.cu/hpp`

Wrapper for Fortrans measurement routines. This class handles the copying between device and host for measurements, and queues the moments for measurement.

`cudaThermfield.cu/hpp`

The thermfield class which is responsible for generating the thermal field. (Separated from the Depondt integrator.)

`measurementQueue.cpp/hpp`

This class queues up moments and performs the measurements in a separate thread so that the GPU does not have to wait for the measurements to finish.

C/C++ Algorithm Files

`depondt_c_wrapper.cpp depondtIntegrator.cpp/hpp hamiltonianCalculations.cpp/hpp mdSimulation.cpp/hpp momentUpdater.cpp/hpp randomnum.cpp/hpp thermfield.cpp/hpp`

The C/C++ implementation. (Structured in the same way as the CUDA implementation.)