A Lockstep Parallel xTB-GFN2 Implementation

Anton Marius Nielsen
Department of Computer Science
University of Copenhagen

Asmus Tørsleff
Quantum Information Science
University of Copenhagen

***Abstract*—Abstract.**

I. Introduction

A. Paper overview

II. Theory

A. xTB GFN2

B. Quantum Implementation

C. Parallel Computing on GPU

- memory coalescing
- array of objects
- object of arrays
- row-major vs column-major order
- host and device memory
- global vs shared vs register memory
- flattening
- barriers and reductions etc.
- block size, warp size
- throughput and latency
- molecule level parallization and lower level parallization

a) *SYCL:*

- hardware diagnostic
- portability
- CUDA vs SYCL vs HIP
- Tooling comparison? (no SYCL LSP afaik)

III. Related Work

- xTB w/ nvfortran. Only old version is supported.
- the xTB python project?

IV. Methodology

A. Porting Fortran to Python

- Differences (row vs column major etc.)
- Done as preparation for lockstep implementation

B. Reproducability with Nix

- What is Nix
- Why Nix

C. Testing

Reproducable tests with Nix. Comparison with reference implementation by exporting and importing input and output as binary data.

TABLE I: The Planets of the Solar System and Their Average Distance from the Sun

Planet	Distance (million km)
Mercury	57.9
Venus	108.2
Earth	149.6
Mars	227.9
Jupiter	778.6
Saturn	1,433.5
Uranus	2,872.5
Neptune	4,495.1

D. Reproducable Builds

Reproducable builds of xTB and nvhpc with Nix.

$$a + b = \gamma \tag{1}$$

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

Fig. 1: A circle representing the Sun.

In Fig. 1 you can see a common representation of the Sun, which is a star that is located at the center of the solar system.

In Table I, you see the planets of the solar system and their average distance from the Sun. The distances were calculated with (1) that we presented in Section IV.

V. Code Structure

VI. Challenges

A. Outdated and Propriatary Projects

- xtb w/ nvfortran
- onemath with adaptivecpp
- dpcpp on nix

B. Implementation Deviating from xTB Paper

- Implementing the equations from the paper does not give the same results as the reference implementation.

VII. Results

A. Benchmarks

VIII. Reflection

IX. Future Work

X. Conclusion

XI. Parallel Computing

Find out how large nbf can get, aka how many iterations the loops in electro can be. Based on that we would choose between the cpu or gpu version, or only use the cpu version if iterations are at most < 100k or such.

XII. Lockstep Parallel Electrostatics and Self-Consistent-Charges

XIII. Lockstep-Parallel Computing of Molecule Energies

Parallelising internal loops of functions that run for single molecules at a time have too few loops to gain any speedups. The overhead of copying the data between host and device, spinning up ALUs, and flattening arrays for so few iterations far outweighs the actual work done by the loops. The parallel SYCL version is times slower than running the sequencial version. The remaining work done is highly optimized linear algebra functions from BLAS. In conclusion we must look at a higher level to perform parallelism on multiple molecules simultaniously to see a meaningful workload for a GPGPU.

C++ port of the original Fortran code for computing electrostatics and self-consistent-charges.

```
std::tuple<double, double> electro(
  int nbf,
  std::vector<double> H0,
  std::vector<std::vector<double>> P,
  std::vector<double> dq,
  std::vector<double> dqsh,
  std::vector<double> atomicGam,
  std::vector<double> shellGam,
  std::vector<std::vector<double>> jmat,
  std::vector<double> shift
  ) {

 int k = 0;
 double h = 0;
 for (int i = 0; i < nbf; i++) {
  for (int j = 0; j < i; j++) {
   h += P[i][j] * H0[k];
   k += 1;
  }
  h += P[i][i] * H0[k] * 0.5;
  k += 1;
 }

 double es = get_isotropic_electrostatic_energy(dq, dqsh,
atomicGam, shellGam, jmat, shift);
 double scc = es + 2.0 * h * evtoau;

 return std::make_tuple(es, scc);
}
```

Parallel version using reduction with SYCL.

```
std::tuple<double, double> electro_sycl(
  int nbf,
  std::vector<double> H0,
  std::vector<double> P_flat,
  std::vector<double> dq,
  std::vector<double> dqsh,
  std::vector<double> atomicGam,
  std::vector<double> shellGam,
  std::vector<std::vector<double>> jmat,
  std::vector<double> shift
  ) {

 queue q{gpu_selector_v};

 size_t H0_size = H0.size();

 double* h_out = malloc_shared<double>(1, q);
 *h_out = 0.0;

 double* P_usm = sycl::malloc_shared<double>(nbf * nbf,
q);
 double* H0_usm = sycl::malloc_shared<double>(H0_size,
q);
 std::copy(P_flat.begin(), P_flat.end(), P_usm);
 std::copy(H0.begin(), H0.end(), H0_usm);

 q.submit([&](sycl::handler& cgh) {
   auto reduction = sycl::reduction(h_out, plus<>());

   cgh.parallel_for(sycl::range<1>(H0_size), reduction,
[=](sycl::id<1> idx, auto& sum) {
     int k = idx[0];

     // Inverse triangular index calculation:
     int i = static_cast<int>((std::sqrt(8.0 * k + 1) -
1) / 2);
     int j = k - i * (i + 1) / 2;

     double val = P_usm[i * nbf + j] * H0_usm[k];
     if (i == j) val *= 0.5;

     sum += val;
   });
 });

 q.wait();

 double h = *h_out;
 free(h_out, q);
 free(P_usm, q);
 free(H0_usm, q);

 double es = get_isotropic_electrostatic_energy(dq, dqsh,
atomicGam, shellGam, jmat, shift);
 double scc = es + 2.0 * h * evtoau;

 return std::make_tuple(es, scc);
}
```

Benchmark:

- Show that the parallel version is slower due to overhead

Electrostatics and self-consistent-charges computation:
Tested with: Caffeine
iterations: 2211
sequential avg over 5 runs: 32.2 µs
parallel avg over 5 runs: 7644.4 µs

- Now present a lockstep-parallel version on multiple molecules.
- Write about xnack's performance degradation with USM in SYCL.
- xnack is supported on mi250, but not my 7900xt

References