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

nix encompases the language, package manager, and linux distribution.

One of the great advantages of Nix lays in its fundamental approach for how it stores packages. This allows you to have multiple versions of the same package installed, since each application is isolated and given the specific dependencies it needs. This fully solves the classic problem of cyclic dependencies and apps depending on different versions of the same package. This is just one of many advantages that comes with the fundamentally different approach Nix takes.

This is also one of the greatest weaknesses though. Since Nix does not follow the Unix file hiearchy structure(FHS), this means that all applications have to be either recompiled and packaged specifically for Nix, or patched to change the interpreter path and the path to all its dynamically linked libraries. A fully statically linked binary will not have this problem, so a Golang program might work out of the box. Thankfully NixPkgs is the largest and freshest package repository with over 120,000 packages. source: https://repology.org/repositories/graphs

- Why Nix

C. Testing

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

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.

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

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

xtb-gpu: Project for packaging and running compilers and SYCL implementations. xTB-Math: xtb math and theory, report, quantum algorithms, fortran -> python port,

VI. Challenges

A. Outdated and Propriatary Projects

- xtb w/ nvfortran
- onemath with adaptivecpp
- dpcpp on nix
- buildFHSEnv, steam-run, nix-ld etc.

Some binaries are near impossible to patch to work with Nix. Intel's installer has been notoriously difficult because it generates other binaries at runtime that also has to be patched.

Instead of patching a binary that is not linked against nix store paths, we can instead use a FHS compliant sandbox with the Nix function `buildFHSEnv` or sandboxing tools like steam-run and nix-ld. Sadly, running the compiler like this will make it produce binaries that also needs to be patched. Multiple issues for packaging the oneapi toolchain exist in the Nix package repository, and despite my best attempts I have chosen to use `steam-run` for installing the compiler and for running the resulting binaries. This comes with its own problems, which we will dive more into depth with. We also attempted to compile the opensource DPC++ compiler from the LLVM repository, though compiling a compiler turned out to be difficult in its own way, so it was left in search of other solutions. This brought us to the Adaptivecpp compiler, which is already packaged for Nix and worked great. The only problem is that the Onemath repository, which contains SYCL implemenations for rocBLAS etc. cannot be compiled for Adaptivecpp despite having instructions for doing exactly that. In a typical fashion there seems to be no automated tests to avoid the degredation of existing features, which has resulted in outdated documentation and the usual single open issue explaning that it simply does not work.

If reaching a pure solution for Nix is desirable, then fixing the Onemath library to work with the Adaptivecpp compiler would be great. Alternatively, the opensource LLVM DPC++ compiler could be worth another look.

In a desperate struggle to get the library working, we decided to give intel's oneapi toolchain another try.

- oneapi is designed for intel's GPU. Won't work for HIP or CUDA without intel's separate plugins. After installing oneapi for AMD it correctly detects my gpu
- Runtime errors with oneapi dpcpp Selected device: AMD Radeon RX 7900 XT terminate called after throwing an instance of 'sycl::*V1::exception' what(): No kernel named* ZTSZ4mainEUlRT_E_ was found

This was fixed by specifying that we want to compile for AMD GPUs with gfx1100 like this:

```
LD_LIBRARY_PATH=~/intel/oneapi/compiler/2025.2/lib
steam-run ~/intel/oneapi/compiler/2025.2/bin/icpx -fsycl
-fsycl-targets=amdgcn-amd-amdhsa -Xsycl-target-
backend=amdgcn-amd-amdhsa --offload-arch=gfx1100 --rocm-
path=/nix/store/k1dgx54vsclvdlrfipa5yr1xd5s44jm7-rocm-
merged -o hello_sycl hello_sycl.cpp
```

source: https://developer.codeplay.com/products/oneapi/amd/2025.2.0/guides/get-started-guide-amd.html#compile-for-amd-gpus

This gave a new error:

```
dgcn-amd-amdhsa -Xsycl-target-backend=amdgcn-amd-amdhsa
--offload-arch=gfx1100 -o hello_sycl hello_sycl.cpp
icpx: error: cannot find ROCm device library; provide its
path via '--rocm-path' or '--rocm-device-lib-path', or
pass '-nogpulib' to build without ROCm device library
icpx: error: cannot find ROCm device library; provide its
path via '--rocm-path' or '--rocm-device-lib-path', or
pass '-nogpulib' to build without ROCm device library
```

This is once again not as straight forward on with Nix as just specifying /usr.

Instead we have to create a store path with symlinks to the relevant rocm packages.

```
with import <nixpkgs> {};
symlinkJoin {
  name = "rocm-merged";

  paths = with rocmPackages; [
    rocm-core clr rccl miopen rocrand rocblas
    rocsparse hipsparse rocthrust rocprim hipcub roctracer
    rocfft rocsolver hipfft hipsolver hipblas
    rocminfo rocm-comgr rocm-device-libs
    rocm-runtime clr.icd hipify
  ];

  # Fix `setuptools` not being found
  postBuild = ''
    rm -rf $out/nix-support
  '';
}
```

source: https://discourse.nixos.org/t/what-should-rocm-path-be/42396/2

With this we finally get the expected output:

```
LD_LIBRARY_PATH=~/intel/oneapi/compiler/2025.2/lib
steam-run ./hello_sycl

Selected device: AMD Radeon RX 7900 XT
Hello, world! I'm sorry, Dave I'm afraid I can't do that -
HAL
```

Now we can give building Onemath another attempt. To build the rocBLAS backend, we have to tell cmake where to find the rocBLAS config. Thankfully rocBLAS can already be found in the Nix package repository NixPkgs.

```
LD_LIBRARY_PATH=~/intel/oneapi/compiler/2025.2/lib
steam-run /nix/store/dx4bdrs7mq3jfviqhszrc7l35ps9kg64-
cmake-3.31.7/bin/cmake .. -DCMAKE_CXX_COMPILER=/home/
maroka/intel/oneapi/compiler/2025.2/bin/icpx -
DCMAKE_C_COMPILER=/home/maroka/intel/oneapi/
compiler/2025.2/bin/icx -DENABLE_MKLCPU_BACKEND=False -
DENABLE_MKLGPU_BACKEND=False -
DENABLE_ROCBLAS_BACKEND=True -
DBUILD_FUNCTIONAL_TESTS=False -DBUILD_EXAMPLES=False -
Drocblas_DIR=/nix/store/
dabbsd24a5kiw73nr3290hipab3n7zll-rocblas-6.3.3/lib/
cmake/rocblas
```

Compile electro.cpp with Onemath:

```
LD_LIBRARY_PATH=~/intel/oneapi/compiler/2025.2/lib
steam-run ~/intel/oneapi/compiler/2025.2/bin/icpx -fsycl
-fsycl-targets=amdgcn-amd-amdhsa -Xsycl-target-
backend=amdgcn-amd-amdhsa --offload-arch=gfx1100 --rocm-
path=/nix/store/k1dgx54vsclvdlrfipa5yr1xd5s44jm7-rocm-
merged -I/home/maroka/Documents/xtb-gpu/oneMath/include
-I/home/maroka/Documents/xtb-gpu/oneMath/build/bin -o
electro electro.cpp
```

Turns out we cannot call the blas functions inside a kernel, because they launch a kernel themselves and thus expects a SYCL queue as an argument. They can be used as a drop in replacement with atom level parallization, but would require more careful separation of the code to launch it in a separate kernel for all atoms.

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