Enabling Vendor-Portable GPU Acceleration for Python-Based Quantum Chemistry with SYCL

Brice Videau, Abhishek Bagusetty, Alvaro Vazquez Mayagoitia
Argonne Leadership Computing Facility
Qiming Sun
Quantum Engine LLC
IWOCL 2026, May 7, 2026 PMA Meeting
abagusetty@anl.gov

Agenda

- gpu4pyscf
- Porting efforts with SYCL
- Challenges

Introduction

• GPU4PySCF is a **GPU-accelerated** extension of the Python-based PySCF quantum chemistry framework
• It focuses on speeding up core electronic structure workloads for research and industrial applications

• GPU4PySCF provides a wide and rich set of advanced quantum chemistry methods usually applied in high-throughput community

• Motivation: Users for ALCF Aurora (Intel Datacenter 1550 GPU) reached out to enquire about support to run QC calculations for many small molecules using python based-workflows

• Importantly: gpu4pyscf's integration to several python-based frameworks for AI/ML workflows

```
from pyscf import gto, scf
from gpu4pyscf.lib import utils

mol = gto.M(atom="O 0 0 0; H 0 0 1; H 0 1 0", basis="def2-svp")
mf = scf.RHF(mol).to_gpu()     # convert a PySCF object to GPU4PySCF
e = mf.kernel()
mf_cpu = mf.to_cpu()           # back to CPU
```

gpu4pyscf

Functionalities supported by GPU4PySCF

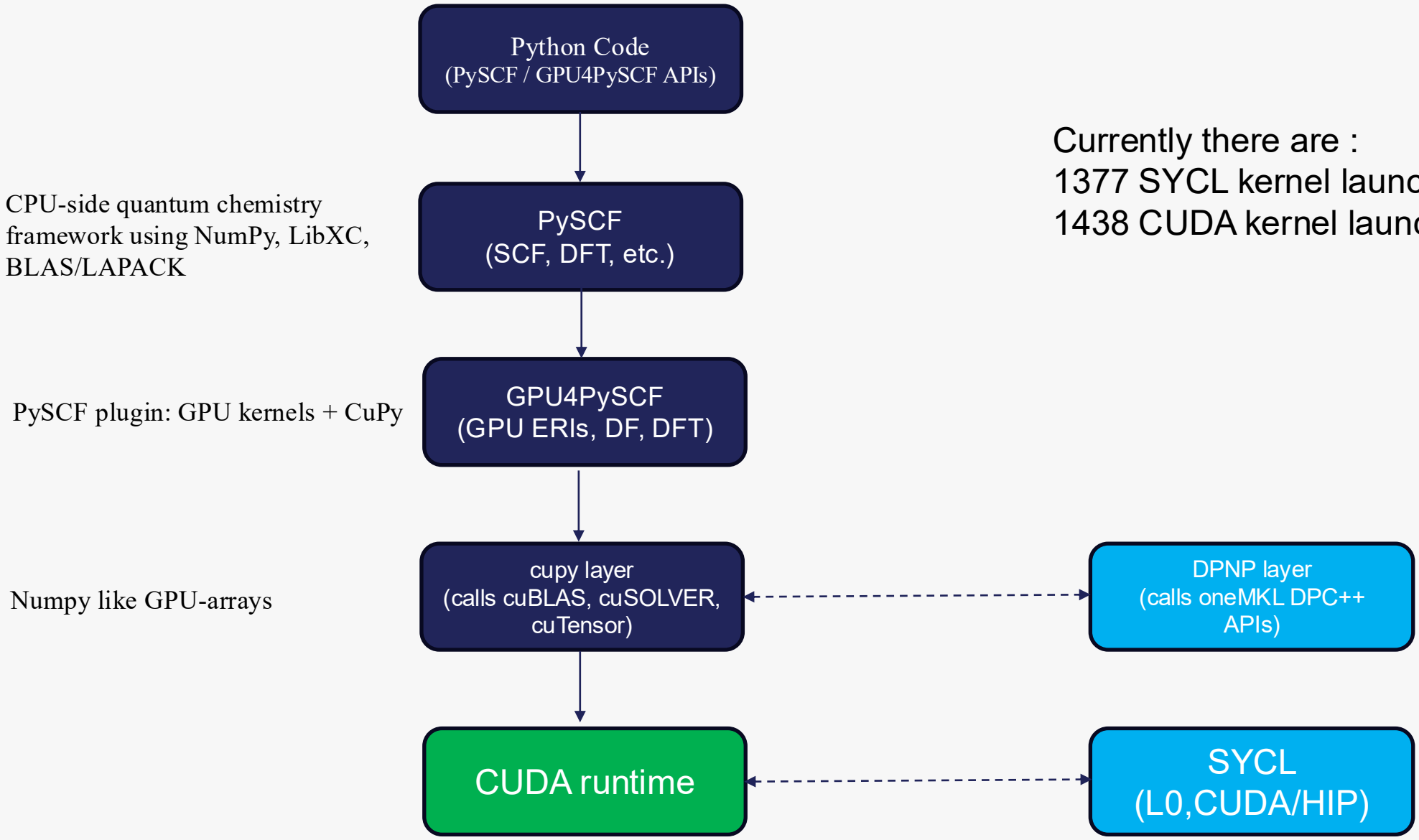
Method	SCF	Gradient	Hessian
direct SCF	O	GPU	CPU
density fitting	O	O	O
LDA	O	O	O
GGA	O	O	O
mGGA	O	O	O
hybrid	O	O	O
unrestricted	O	O	O
PCM solvent	GPU	GPU	FD
SMD solvent	GPU	GPU	FD
dispersion correction	CPU*	CPU*	FD
nonlocal correlation	O	O	NA
ECP	CPU	CPU	CPU
MP2	GPU	CPU	CPU
CCSD	GPU	CPU	NA

- ‘O’: carefully optimized for GPU.
- ‘CPU’: only cpu implementation.
- ‘GPU’: drop-in replacement or naive implementation.
- ‘FD’: use finite-difference gradient to approximate the exact Hessian matrix.
- ’NA’: not available.
- ‘CPU*’: DFTD3 [100]/DFTD4 [101] on CPU.

* List is incomplete and not actively maintained since 2024
New methods like TD-DFT, etc were optimized for GPU

https://pyscf.org/user/gpu.html#functionalities-supported-by-gpu4pyscf

Gpu4pyscf - Architecture



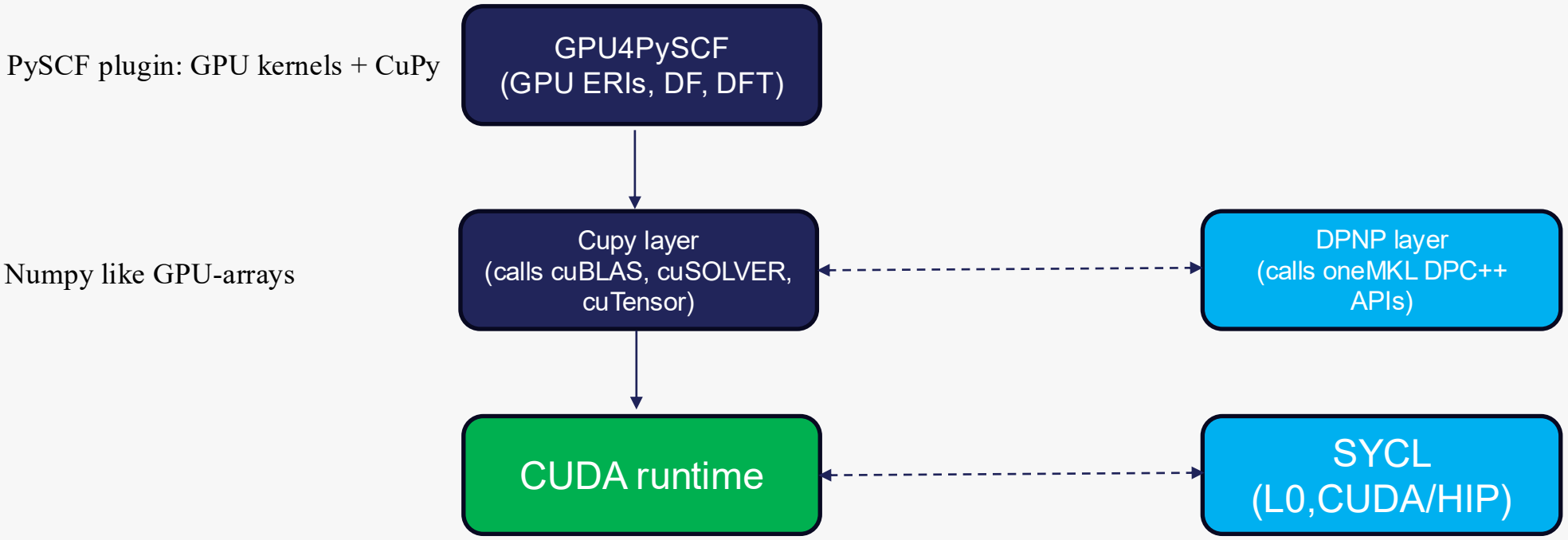
Currently there are :
1377 SYCL kernel launch (ported)
1438 CUDA kernel launch

Gpu4pyscf - Architecture

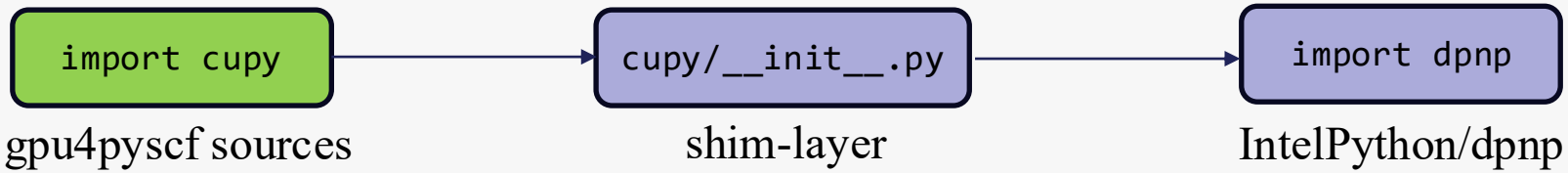
- Core-developers of gpu4pyscf also maintain a mirror at ByteDance-Seed. Their main focus is Nvidia hardware

To support Portable APIs:

- Minimal and modular code-changes to support SYCL ecosystem and relevant libraries (oneMKL, oneDPL, DPNP, DPCTL)
- Maintainers are agonistic to the oneAPI infrastructure
- Agreed to make minimal changes to existing codebase:
  - CUDA kernels (require a very minimal if-def`s to handle SYCL kernel launch & very specific SYCL functionality)
  - Modular – Python infrastructure to support SYCL is contained to a set of files (to mimic Nvidia's cupy support via monkey shim layer patch using Intel's DPNP)



Gpu4pyscf – cupy mapping to dpnp



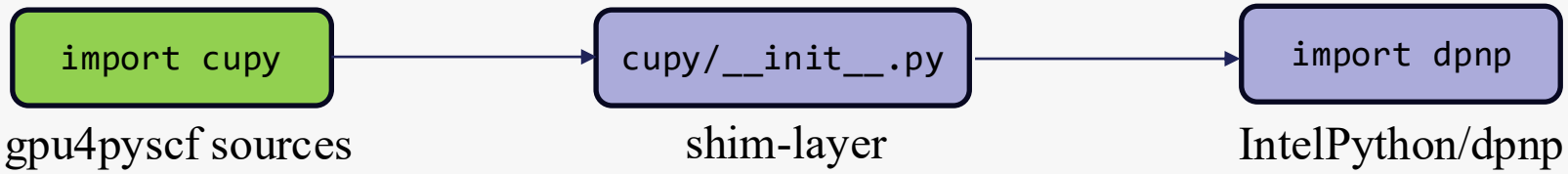
Array Creation	SYCL / dpnp target	Effort	Notes

`cupy.array(x)`
`cupy.asarray(x)`
`cupy.zeros/ones/empty`
`cupy.eye/arange/linspace`
	`dpnp.array(x)`
	`dpnp.asarray(x)`
	`dpnp.zeros/ones/empty`
	`dpnp.eye/arange/linspace`
		DROP-IN	All direct 1-to-1 mappings since dpnp mirrors the NumPy/CuPy API closely

Math & Element-wise	SYCL / dpnp target	Effort	Notes

`cupy.dot/einsum/matmul`
`cupy.sum/max/min/abs`
`cupy.multiply/add/exp/log`
	`dpnp.dot/einsum/matmul`
	`dpnp.sum/max/min/abs`
	`dpnp.multiply/add/exp/log`
		DROP-IN	Also direct 1-to-1 mappings

Gpu4pyscf – cupy mapping to dpnp



Device & Memory Management	SYCL / dpnp target	Effort	Notes
`cupy.cuda.Device(id)`	`dpctl.select_device_gpu(id)` `dpctl.SyclDevice`	Work-around	
`cupy.cuda.get_current_stream()`	`C++ SYCL (in-order) queue cached`	Work-around	
`cupy.get_default_memory_pool()` `.free_all_blocks()`	`dev.get_info<sycl::ext::intel::info::device::free_memory>()`	Work-around	

`mempool = cupy.get_default_memory_pool()`
`used_mem = mempool.used_bytes()`
`mem_limit = mempool.get_limit()`
	`dev.get_info<sycl::ext::intel::info::device::free_memory>()`
	`C++ API accessed via Python cytpes interfaced as shared-libs`
		Work-around	There is no equivalent memory pool infrastructure in oneAPI ecosystem

`array.data.ptr (raw pointer)`	`array.data.ptr`	Functionality was added in DPNP	https://github.com/IntelPython/dpnp/issues/2475

Gpu4pyscf – cupyx mapping to dpnp

```
from cupyx.scipy.linalg import solve_triangular, lu_factor, lu_solve, expm, block_diag
```

cupyx function	SYCL / dpnp target	Effort	Notes
`lu_factor`	`dpnp.scipy.linalg.lu_factor`	DROP-IN	Native dpnp implementation; direct 1-to-1 replacement
`lu_solve`	`dpnp.scipy.linalg.lu_solve`		
`expm`	`dpnp.scipy.linalg.expm`		
`solve_traingular`	`onemkl::blas::trsm`	oneMKL	Not in dpnp; mapped to oneMKL TRSM via SYCL interop
`block_diag`	Custom DPNP method	Hand-written	No oneMKL equivalent; manually assembled from dpnp array ops

Gpu4pyscf – Multi-GPU support

- The current support is mostly node-local and multi-GPU support is via cupy Peer-to-Peer (P2P) access and context switching
- cudaStreams are created on discoverable devices (CUDA_VISIBLE_DEVICES) and cached
- 1 cudaStream per device is currently being operated
- CUDA kernels obtain the cudaStream from upstream cupy-native objects and passes it down to low-level GPU kernels

- Multi-GPU support is quite limited for SYCL backend (05/01/2026)
- SYCL queues are created for per-device/per-tile (ZE_AFFINITY_MASK, ZE_FLAT_DEVICE_HIERARCHY=FLAT) for Intel GPUs on Aurora
- Ideal solution – Both DPNP/DPCTL ecosystem + SYCL kernels rely on a single queue per device
- Current solution – IntelPython creates it own SYCL queues in the low-level layers and all the SYCL-kernels use a singleton SYCL queue. Given the `default_context` idiom, no issues with synchronizations yet!
- Multi-gpu support is enabled for Intel backend (but not yet tested)
- Completed: May, 2025 Support for P2P “detection, enable, disable” was implemented in IntelPython/dpctl (https://github.com/IntelPython/dpctl/issues/2073)

gpu4pyscf

Functionalities supported by GPU4PySCF

Method	SCF	Gradient	Hessian
direct SCF	O	GPU	CPU
density fitting	O	O	O
LDA	O	O	O
GGA	O	O	O
mGGA	O	O	O
hybrid	O	O	O
unrestricted	O	O	O
PCM solvent	GPU	GPU	FD
SMD solvent	GPU	GPU	FD
dispersion correction	CPU*	CPU*	FD
nonlocal correlation	O	O	NA
ECP	CPU	CPU	CPU
MP2	GPU	CPU	CPU
CCSD	GPU	CPU	NA

- 'O': carefully optimized for GPU.
- 'CPU': only cpu implementation.
- 'GPU': drop-in replacement or naive implementation.
- 'FD': use finite-difference gradient to approximate the exact Hessian matrix.
- 'NA': not available.
- 'CPU*': DFTD3 [100]/DFTD4 [101] on CPU.

Tested thoroughly using SYCL

Blocked due to lack of equivalent functionality for:

```
from cupyx.scipy.sparse.linalg import LinearOperator, gmres, minres
```

Ported to SYCL but not tested

https://pyscf.org/user/gpu.html#functionalities-supported-by-gpu4pyscf

gpu4pyscf – Libxc integration for SYCL backend

• LibXC is a dependency library written in C. Modified to CUDA to support a subset of commonly used XC-functionals (internally maintained repo)

• Easy to port to SYCL but device-side function pointers are an issue
• Timeline: Intel suggested it would take many months for a fix

• Work-around: ExchCXX is a modern C++ library for the evaluation of exchange-correlation (XC) functionals required for DFT. Supports CUDA, HIP and SYCL backends
• Developed as a part of NWChemEx (Developer: David Young, Azure Quantum Elements)

• Challenge - Ensure the functionality in gpu4pyscf related to libxc is intact but maps to Exchcxx APIs

• Future issues: Many XC functionals are not supported for device-side execution using Exchcxx project. Exchcxx provides a mechanism to fallback to libxc for CPU-execution. Currently this is not yet turned-on!

Name	Libxc Identifier	ExchCXX Identifier	Device?
Slater Exchange	XC_LDA_X	Kernel::SlaterExchange	Y
Vosko-Wilk-Nusair III	XC_LDA_C_VWN_3	Kernel::VWN3	Y
Vosko-Wilk-Nusair V	XC_LDA_C_VWN_RPA	Kernel::VWN5	Y
Perdew-Burke-Ernzerhof (Exchange)	XC_GGA_X_PBE	Kernel::PBE_X	Y
Perdew-Burke-Ernzerhof (Correlation)	XC_GGA_C_PBE	Kernel::PBE_C	Y
Revised PBE from Zhang & Yang	XC_GGA_X_PBE_R	Kernel::revPBE_X	Y
Perdew-Wang 91 (LDA)	XC_LDA_C_PW	Kernel::PW91_LDA	Y
Perdew-Wang 91 (LDA) Modified	XC_LDA_C_PW_MOD	Kernel::PW91_LDA_MOD	Y
Perdew-Wang 91 (LDA) RPA	XC_LDA_C_PW_RPA	Kernel::PW91_LDA_RPA	Y
Perdew-Zunger 86	XC_LDA_C_PZ	Kernel::PZ86_LDA	Y
Perdew-Zunger 86 Modified	XC_LDA_C_PZ_MOD	Kernel::PZ86_LDA_MOD	Y
Becke Exchange 88	XC_GGA_X_B88	Kernel::B88	Y
Lee-Yang-Parr	XC_GGA_C_LYP	Kernel::LYP	Y

https://github.com/argonne-lcf/AuroraBugTracking/issues/77

Challenges – IntelPython (time-taking) issues

• **data.ptr usage error on dpnp.ndarray** (#2532) — Code that worked in CuPy by accessing array.data.ptr to get a raw device pointer for ctypes/C extension calls silently fails or produces wrong results in dpnp. The USM pointer lives in a completely different interface (__sycl_usm_array_interface__), so it looks like it works but passes a garbage or null pointer to the underlying C kernel — with no immediate error, just wrong numerics downstream

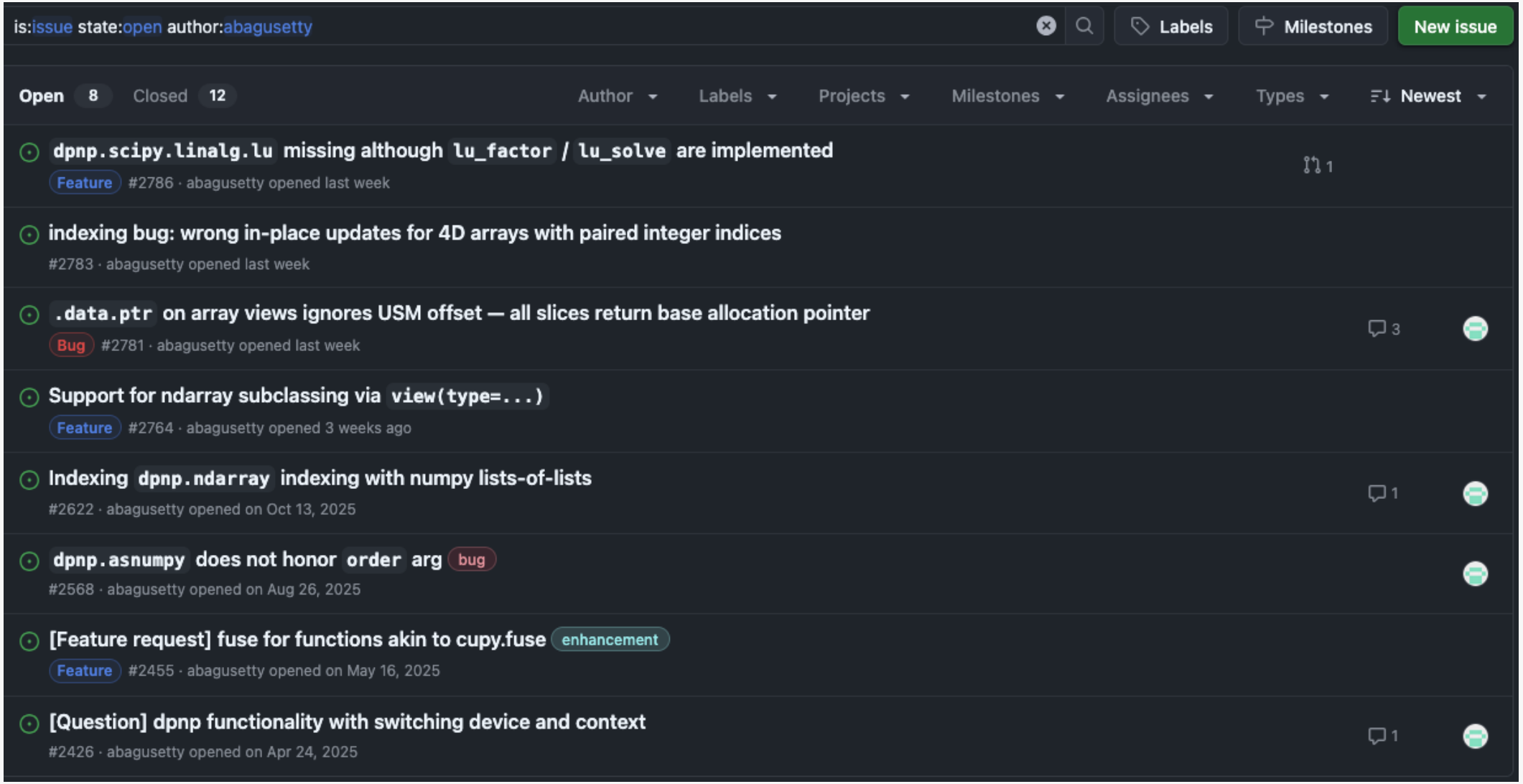
• **DPNP vs NumPy strides mismatch** (#2640) — dpnp arrays can have different internal strides than their NumPy equivalents for the same shape/dtype, causing silent numerical errors or crashes in kernels that assume C-contiguous layout. Extremely hard to debug because the array *prints correctly* but produces wrong results when passed to low-level routines that rely on stride metadata

• **TypeError: bool() conversion fails on single-element arrays with ndim > 0** (#2784) — Conditionals like if arr: or assert arr that work in NumPy and CuPy raise a TypeError in dpnp for non-0d single-element arrays. These checks are scattered implicitly across scientific code and fail deep in call stacks, making the root cause hard to trace

• **Floor division // returns incorrect result when left operand is Python int** (#2720) — int // dpnp_array silently produces wrong values due to reflected operator dispatch behavior differing from NumPy. Particularly nasty because the operation succeeds without error and the result looks plausible, only caught by comparing against a CPU reference — which may not always be done

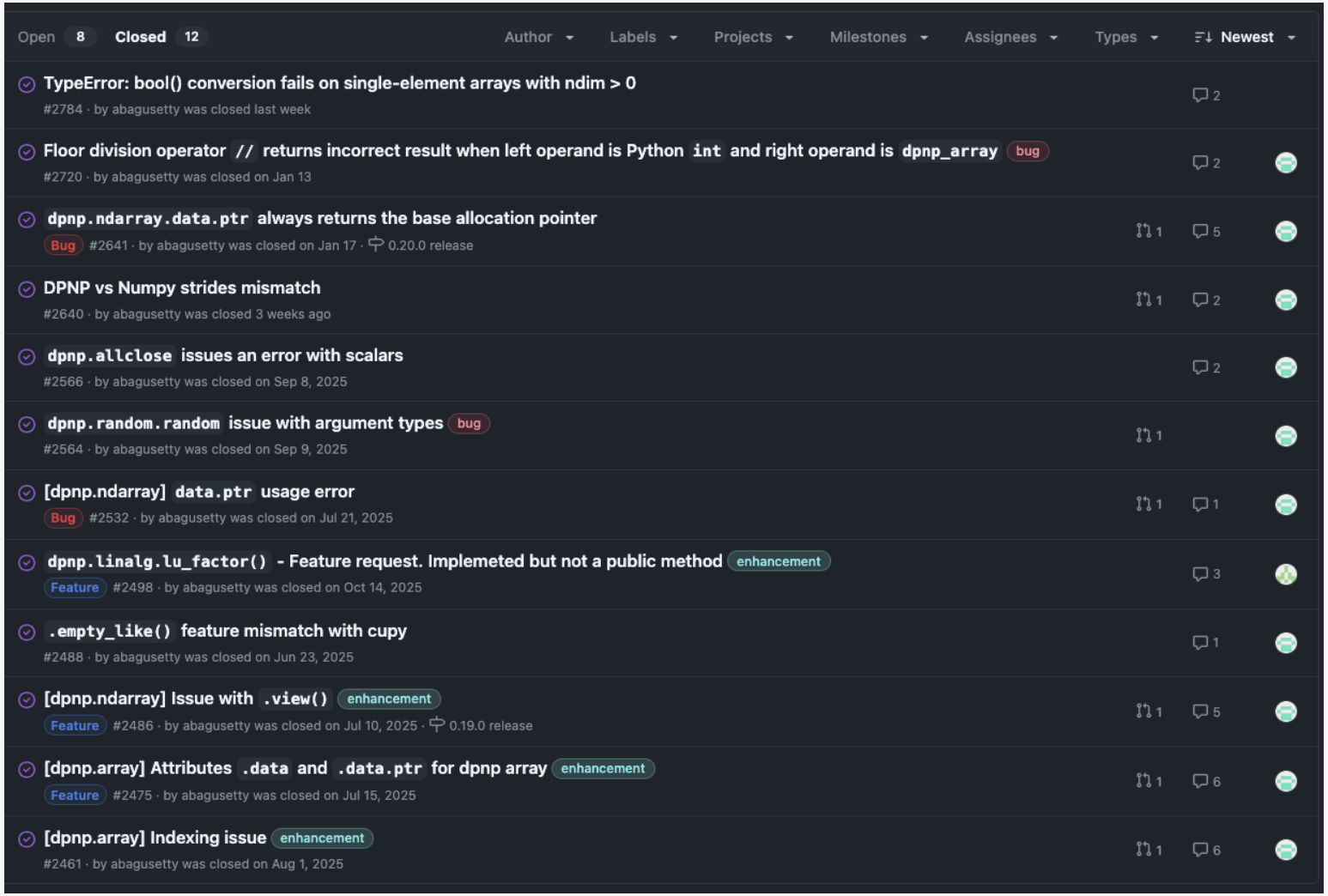
Challenges – IntelPython (Open) issues

- None of the issues opened are “critical”. All the work-arounds are implemented in gpu4pyscf



Challenges – IntelPython (Closed) issues

- All the IntelPython (dpnp) issues are either related to implementation differences between cupy/numpy and dpnp



Challenges - Pointer Dereference in Kernel Launch

• In CUDA, <<...>> passes arguments by value. Dereferencing a host pointer (*envs) copies the struct into kernel registers/memory at launch time — the kernel sees a *device-side value*
• In SYCL, parallel_for captures lambda arguments by value at submission time. If you pass *envs directly inside the lambda, the SYCL runtime tries to copy the struct (which may contain device pointers or non-trivially copyable members) through the *host stack* at enqueue time

```
dim3 threads(THREADSX, THREADSY);
dim3 blocks((ntasks_ij+THREADSX-1)/THREADSX, (ntasks_kl+THREADSY-1)/THREADSY);

GINTint2e_get_veff_ip1_kernel<3, NABLAGSIZE3> <<<blocks, threads>>>(*envs, *jk, *offsets);
```



```
sycl::range<2> threads(THREADSY, THREADSX);
sycl::range<2> blocks((ntasks_kl+THREADSY-1)/THREADSY, (ntasks_ij+THREADSX-1)/THREADSX);

stream.parallel_for<class GINTint2e_get_veff_ip1_kernel_3_sycl>(sycl::nd_range<2>(blocks * threads,
threads), [=](auto item) { GINTint2e_get_veff_ip1_kernel<3, NABLAGSIZE3> (*envs, *jk, *offsets); });
```

```
auto dev_envs = *envs;
auto dev_jk = *jk;
auto dev_offsets = *offsets;

stream.parallel_for<class GINTint2e_get_veff_ip1_kernel_3_sycl>(sycl::nd_range<2>(blocks * threads, threads),
[=](auto item) { GINTint2e_get_veff_ip1_kernel<3, NABLAGSIZE3> (dev_envs, dev_jk, dev_offsets); });
```

Challenges – Inconvenience with Shared/Local Memory Allocations

• Dynamic shared memory passed as a explicit argument allocated via SYCL `local_accessor`, static shared memory allocations via `group_local_memory_for_overwrite`

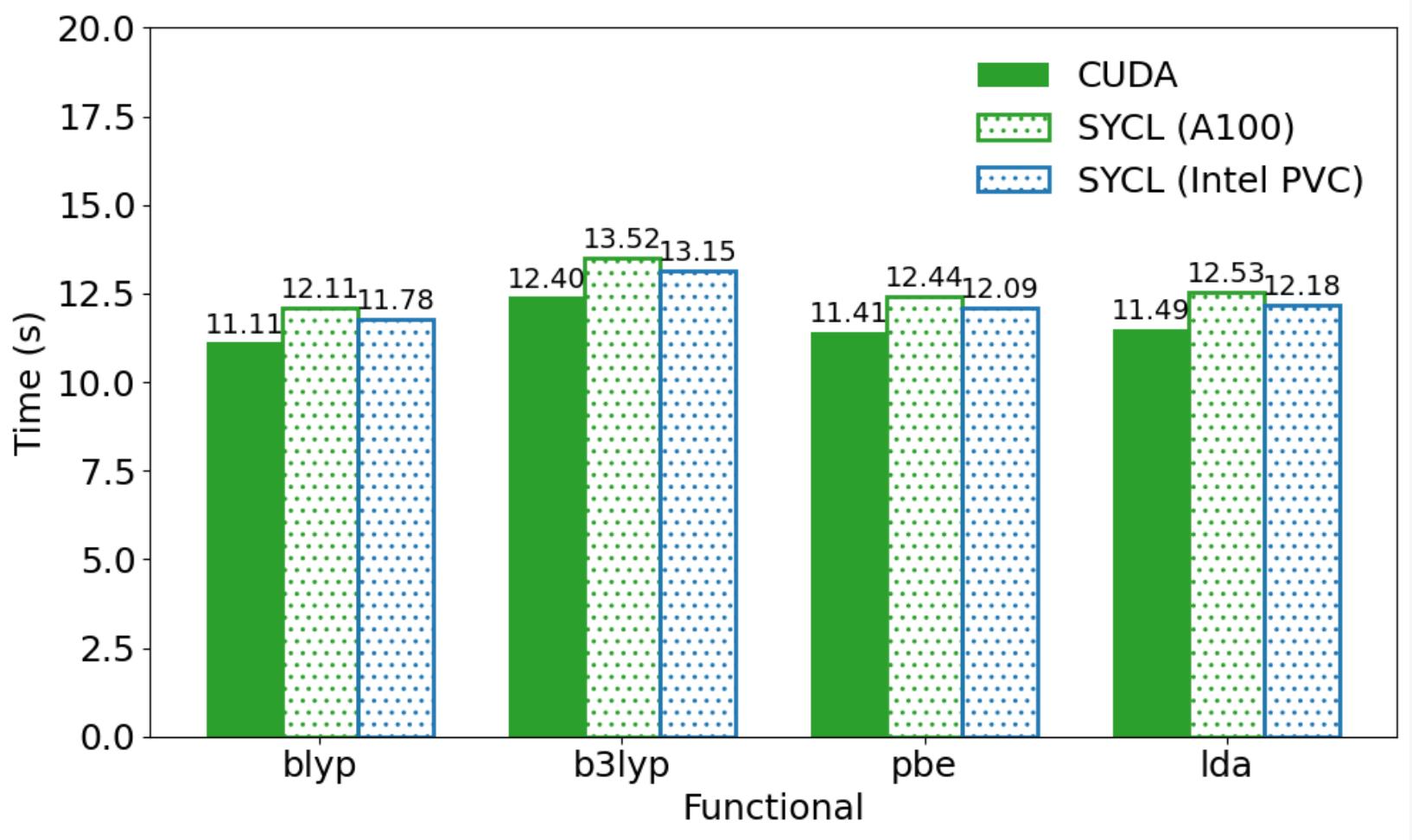
```
void rys_vjk_ip1_kernel(RysIntEnvVars envs, JKMatrix jk, BoundsInfo bounds,
                        int *pool, int reserved_shm_size, int nfij, int nfkl
                        #ifdef USE_SYCL
                        , sycl::nd_item<2> &item, double *shared_memory
                        #endif
                        )
{
    #ifdef USE_SYCL
    int threadIdx_x = item.get_local_id(1);
    ...

    auto thread_block = item.get_group();
    int &ntasks     = *sycl::ext::oneapi::group_local_memory_for_overwrite<int>(thread_block);
    int &ish        = *sycl::ext::oneapi::group_local_memory_for_overwrite<int>(thread_block);
    int &jsh        = *sycl::ext::oneapi::group_local_memory_for_overwrite<int>(thread_block);
    double (&ri)[3] = *sycl::ext::oneapi::group_local_memory_for_overwrite<double[3]>(thread_block);
    #else
    int threadIdx_x = threadIdx.x;
    ...

    __shared__ int ntasks;
    __shared__ int ish, jsh
    __shared__ double ri[3];
    extern __shared__ double shared_memory[];
        #endif
```

Performance

- Very initial performance test set
- DFT benchmarks
- Single GPU benchmarks on CUDA and SYCL (A100 & PVC 1-tile)
- Runtime is synchronized with `ZE_SERIALIZE=2` & `CUDA_LAUNCH_BLOCKING=1`



Ongoing & Future work:

- More Debugging for hessians (incorrect results)
- SYCL on AMD is a pain !
  - Majority of the code-structure is assumed with 32 as warpSize
- SYCL on Nvidia results are promising on A100 & H100

Conclusion

- **GPU4PySCF as a portability case study** combines Python array programming with performance-critical CUDA kernels for quantum chemistry
- **Goal:** Port GPU4PySCF from CUDA to SYCL without changing the project's codebase scope
- **Two-layer methodology:**
  - *Python layer:* Interface CuPy with DPNP for array operations
  - *Kernel layer:* Port CUDA kernels to SYCL using a thin compatibility header
- **Next steps:** Evaluate performance, analyze results, and identify tuning opportunities

  Performance: Ensure DPCTL & DPNP Intel Python APIs provide a stable default SYCL queue (in-order) preferably to be used by user provided SYCL kernels in gpu4pyscf. This avoids the creation of a singleton SYCL queue per device by the user
- **Future work**: Look into SYCL (HIP backend) targeting 64 wavefront size GPU devices