openQxD with QUDA

Anian Altherr, Juan Antonio Fernández De la Garza, Roman Gruber, Tim Harris, Stephanie Maier, Marina Krstic Marinkovic, Letizia Parato

January 30, 2024

ETHZ

Table of contents

Current status

What is done so far

Main goal 1:
We have the O(a) improved QCD+QED Wilson-Clover Dirac operator up and running within QUDA (using C* boundaries in 0,1,2 or 3 spatial directions)

- ✓ SU(3) Clover term (native in QUDA)
- ✓ U(1) Clover term (only by transfer)
- ✓ Dslash with U(3) fields
- ✓ C*-boundaries

Inverters

- Interface to QUDA various solvers

Main goal 2:
All QUDA solvers (including Multigrid) can be called from within openQxD (also for QCD+QED C* lattices!).

- ✓ Solve QCD-only Lattices
- ✓ Solve QCD-only Lattices with C*-boundaries
- ✓ Solve QCD+QED Lattices with C*-boundaries
- ✓ Multigrid

CI/CD

- Using CSCS for CI/CD pipelines

Main goal 3:
Proof-of-concept tests run on Daint via CSCS interface.

- ✓ `devel/*/check*.c`
- ✓ QUDA-tests in `devel/quda`

TODO/ Known issues / Bugs

- EO-preconditioned Dirac operator doesn't work yet

Code

- openQxD implementation:
  - Repo: `https://gitlab.com/rcstar/openQxD-devel`
  - Branch: `feature/quda/main-thesis-release`
  - Pull-request: `https://gitlab.com/rcstar/openQxD-devel/-/merge_requests/41`
- QUDA implementation:
  - Repo: `https://github.com/lattice/quda`
  - Branch: `feature/openqxd`
  - Pull-request: `https://github.com/lattice/quda/pull/1414`
  - Review in progress

Implementation details

Implementation details

- Assume you have a program written in openQxD and you want to use QUDA solvers

```c
int main(int argc, char *argv[]) {
   [...]
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   read_infile(argc, argv);
   [...]

   for (nc = first; nc <= last; nc += step) {
      [...]
      import_cnfg(cnfg_file);
      /* calling solver */
      [...]
   }

   MPI_Finalize();
   exit(0);
}
```

```c
#include "quda_utils.h"

int main(int argc, char *argv[]) {
   [...]
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   read_infile(argc, argv);
   quda_init(); /* Initialize */
   [...]

   for (nc = first; nc <= last; nc += step) {
      [...]
      import_cnfg(cnfg_file);
      /* calling QUDA solver */
      handle = openQCD_qudaSolverSetup(infile, "Solver");
      r = openQCD_qudaInvert(handle, mu, eta, psi, &status);
      openQCD_qudaSolverDestroy(handle);
   }

   openQCD_qudaFinalize(); /* Finalize */
   MPI_Finalize();
   exit(0);
}
```

Performance

Strong scaling of Dirac operator

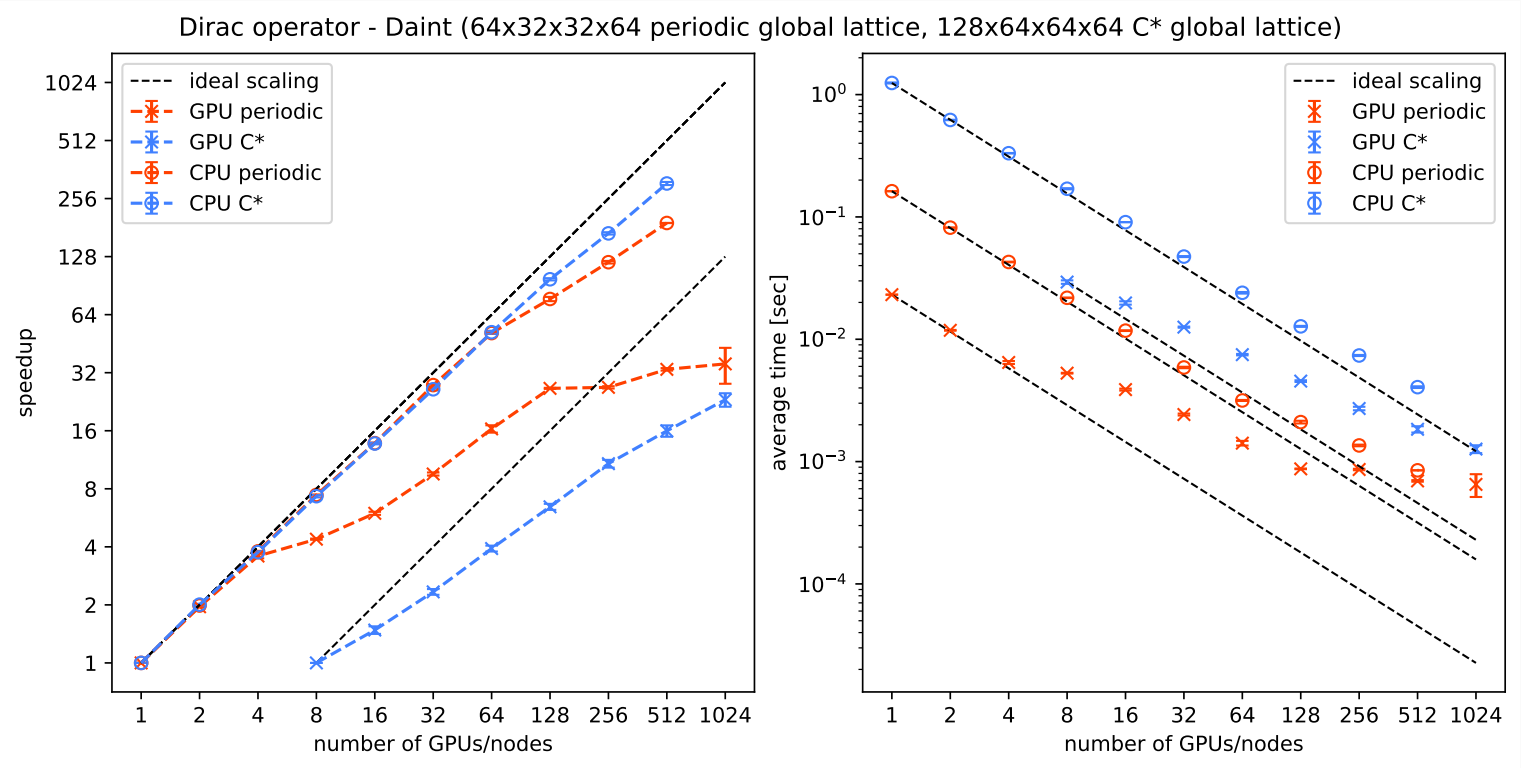


Figure 1: Strong scaling (left) and time for one application (right) of the Dirac operator on Piz Daint on the CPU (circles) and GPU (crosses) using a periodic (red) and a lattice with 3 C* directions (blue). CPU: 32 ranks per node, GPU: 1 P100 per node.

Weak scaling of inverter

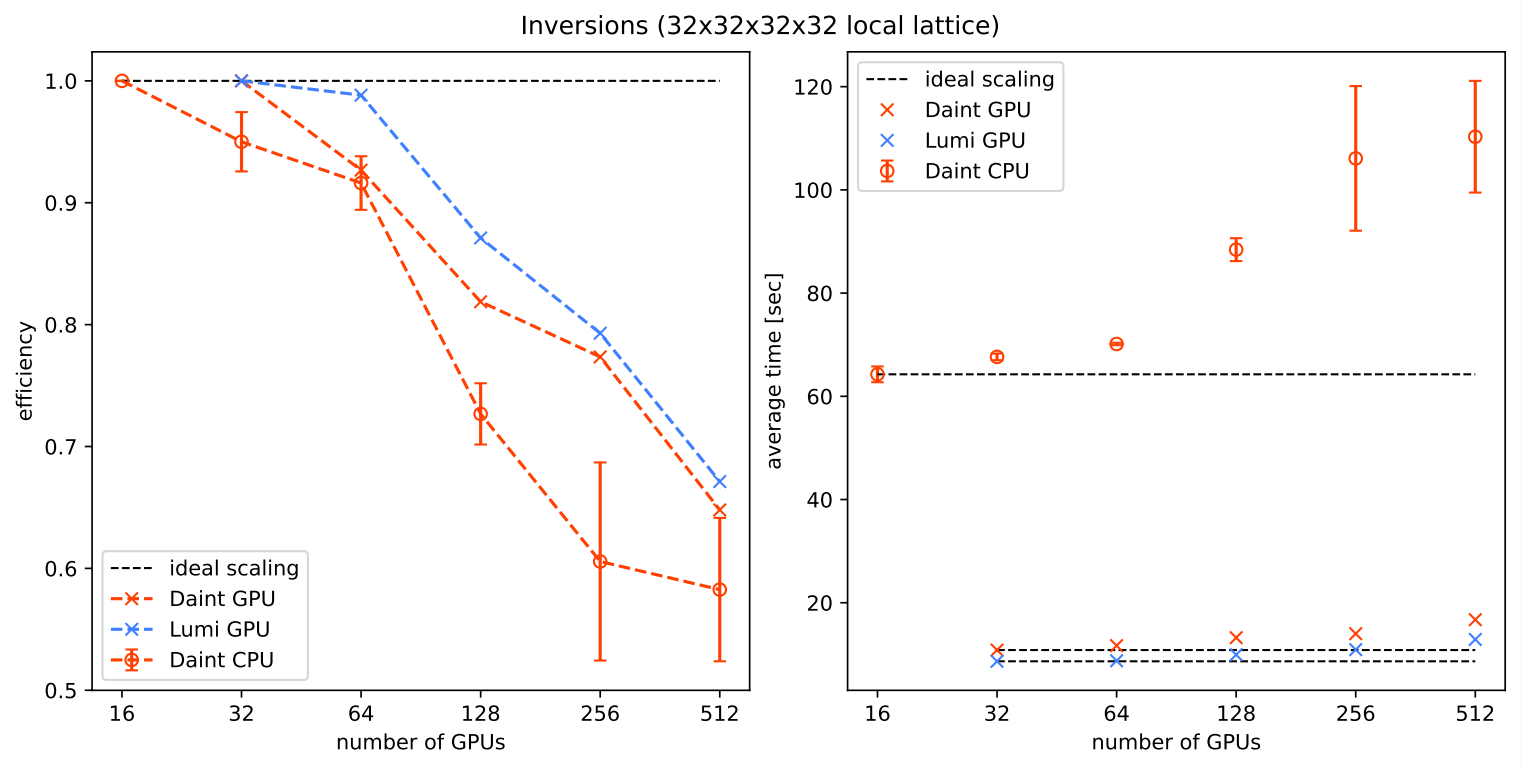


Figure 2: Weak scaling (left) and time to solution (right) for one solve of the Dirac equation on LUMI (blue) and on Piz Daint (red) on the CPU (circle) and GPU (crosses) partition. CPU: `DFL_SAP_GCR`, GPU: GCR with multigrid.

Cost on Daint - Dirac operator

	cost in node-seconds [sec]	
	periodic ($V_{\mathrm{G}} = 64 \times 32 \times 32 \times 64$)	
nodes/GPUs	Piz Daint GPU	Piz Daint CPU
1	0.023146040(628)	0.162241(174)
2	0.023661(178)	0.163499(376)
4	0.025835(708)	0.17126(101)
8	0.042252(302)	0.174588(889)
16	0.06203(113)	0.18817(131)
32	0.07759(142)	0.18814(254)
64	0.09055(397)	0.20222(196)
128	0.111636(701)	0.26917(683)
256	0.22056(302)	0.34688(534)
512	0.35585(533)	0.434372608(0)
1024	0.667(141)	

Cost on LUMI - Dirac operator

		cost in node-seconds [sec]	
		periodic ($V_{\mathrm{G}} = 64 \times 32 \times 32 \times 64$)	
nodes	GPUs	LUMI GPU	LUMI CPU
1	1	0.015923220(595)	
1	2	0.0077102(141)	
1	4	0.005683(714)	
1	8	0.005751(579)	0.044532(772)
2	16	0.008744(472)	0.045384(362)
4	32	0.01111(239)	0.04758(468)
8	64	0.01123(100)	0.047173(458)
16	128	0.01679(148)	0.04909(394)
32	256	0.02214(183)	0.047487(869)
64	512	0.03478(155)	0.052639(898)
128	1024	0.05901(380)	0.06068(222)

Frame Title

Notice that we count 8 GPUs per node on LUMI-G, since one AMD MI250 abstracts itself as 2 GPUs from the point of view of the program.

Specs

The performance measures that follow are obtained on the machines

- Piz Daint multicore at CSCS, Switzerland, 2x Intel® Xeon® E5-2695 v4 @ 2.10GHz (2x 18 cores, 64/128 GB RAM)
- Piz Daint hybrid at CSCS, Switzerland, 1x Intel® Xeon® E5-2690 v3 @ 2.60GHz (12 cores, 64GB RAM) and 1x NVIDIA® Tesla® P100 16GB
- LUMI-C at CSC, Finland, 2x AMD EPYC 7763 @2.45GHz (2x 64 cores, 256-1024 GB RAM)
- LUMI-G at CSC, Finland, 1x AMD® Trento™ (64 cores, 512 GB RAM) and 4x AMD® Instinct™ MI250X GPU 128GB

Outlook

Outlook

(1) Optimisation: Packed format for U(3) fields (QUDAs internal representation)

(3) Optimisation: Calculating U(1) clover term directly within QUDA (no transfer)

(10) Optimisation: No doubling of lattice within QUDA for C* boundaries (effort needed)

(3) Optimisation: Tuning/Understanding of solver parameters

(8) Feature: Interface multiple RHS solvers (effort needed)

(3) Feature: Partitioning of process grid

(2) Feature: Interface eigensolvers (interface written, understanding and tuning of parameters in progress!)

(6) Further developments: Unified fields (HMC, offloading of arbitrary functionality, beyond PASC project)

Further developments

Overview

- Motivation
- Unified fields
- Implicit transfer of fields

```
// PhD Student Alice wrote this code
int main(int argc, char *argv[]) {
   spinor_dble **wsd;
   [...]

   alloc_wsd(N+1); // N = 10
   sd = reserve_wsd(N+1);

   for (i=0; i<N; i++) {
      random_sd(VOLUME, sd[i], 1.0);
      scale_dble(VOLUME, 3.0, sd[i]);
      Dw_dble(0.0, sd[i], sd[N]);
      Dw_dble(0.0, sd[N], sd[i]);
      r = norm_square_dble(VOLUME, 1, sd[i]);
   }

   release_wsd();
}
```

```
// PhD Student Alice wrote this code
int main(int argc, char *argv[]) {
   spinor_dble **wsd;
   [...]

   alloc_wsd(N+1); // N = 10
   sd = reserve_wsd(N+1);

   for (i=0; i<N; i++) {
      random_sd(VOLUME, sd[i], 1.0);  // CPU
      scale_dble(VOLUME, 3.0, sd[i]); // CPU
      Dw_dble(0.0, sd[i], sd[N]);     // GPU overwrite
      Dw_dble(0.0, sd[N], sd[i]);     // GPU overwrite
      r = norm_square_dble(VOLUME, 1, sd[i]); // CPU
   }

   release_wsd();
}
```

Dirac operator

GPU implementation of `Dw_dble`:

- Takes CPU `spinor_dble` as input/output field
- Transfer input/output fields to GPU (H2D)
- Apply Dirac operator on GPU
- Transfer input/output fields to CPU (D2H)

```
#if (defined AVX)
// implementation using AVX intrinsics
functionA(spinor_dble *s) { ... }
#elif (defined x64)
// implementation using SSE2 intrinsics
functionA(spinor_dble *s) { ... }
#else
// default implementation
functionA(spinor_dble *s) { ... }
#endif
```

```
#if (defined AVX)
// implementation using AVX intrinsics
functionA(spinor_dble *s) { ... }
#elif (defined x64)
// implementation using SSE2 intrinsics
functionA(spinor_dble *s) { ... }
#elif (defined GPU_OFFLOADING)
// GPU overloading of the function
functionA(spinor_dble *s) { ... }
#else
// default implementation
functionA(spinor_dble *s) { ... }
#endif
```

Motivation: Requirements

- No changes in existing programs written in openQxD
- No changes in existing functions that operate on spinors
- Fully backwards compatible with the memory layout of openQCD (array of structs)

Motivation: Next problem

- Initial code: all functions implemented in CPU $\rightarrow$ no transfers needed
- Ideal final code: all functions implemented in GPU $\rightarrow$ no transfers needed $\longrightarrow$ we'll probably never reach that
- Intermediate solution: some functions are ported to GPU, but not all of them $\rightarrow$ needs transfers

$\rightarrow$ We don't want to rewrite every program, when a new function is ported to GPU!

```
// PhD Student Alice wrote this code
int main(int argc, char *argv[]) {
   spinor_dble **wsd;
   [...]

   alloc_wsd(N+1); // N = 10
   sd = reserve_wsd(N+1);

   for (i=0; i<N; i++) {
      random_sd(VOLUME, sd[i], 1.0);  // CPU
      scale_dble(VOLUME, 3.0, sd[i]); // GPU overwrite
      Dw_dble(0.0, sd[i], sd[N]);     // GPU overwrite
      Dw_dble(0.0, sd[N], sd[i]);     // GPU overwrite
      r = norm_square_dble(VOLUME, 1, sd[i]); // CPU
   }

   release_wsd();
}
```

```
// PhD Student Alice wrote this code
int main(int argc, char *argv[]) {
   spinor_dble **wsd;
   [...]

   alloc_wsd(N+1); // N = 10
   sd = reserve_wsd(N+1);

   for (i=0; i<N; i++) {
      random_sd(VOLUME, sd[i], 1.0);              // CPU
      scale_dble(VOLUME, 3.0, sd[i]);             // CPU
      // transfer sd[i], sd[N] (H2D)
      Dw_dble(0.0, sd[i], sd[N]);                 // GPU
      // transfer sd[i], sd[N] (D2H)
      // transfer sd[i], sd[N] (H2D)
      Dw_dble(0.0, sd[N], sd[i]);                 // GPU
      // transfer sd[i], sd[N] (D2H)
      r = norm_square_dble(VOLUME, 1, sd[i]);     // CPU
   }

   release_wsd();
}
```

Unified fields

We need to unify fields, and only transfer when needed by the code!

Figure 3: One field to rule them all!

Unified fields

Figure 4: Each field with openQxD corresponds to a field within QUDA.

- Establish a 1-1 correspondence between CPU/GPU fields

$\Longrightarrow$ Everytime (de-)allocating a field $\rightarrow$ (de-)allocate on both devices

$\Longrightarrow$ Maintain consistency among the two fields (CPU/GPU manipulates field)

Maintaining consistency

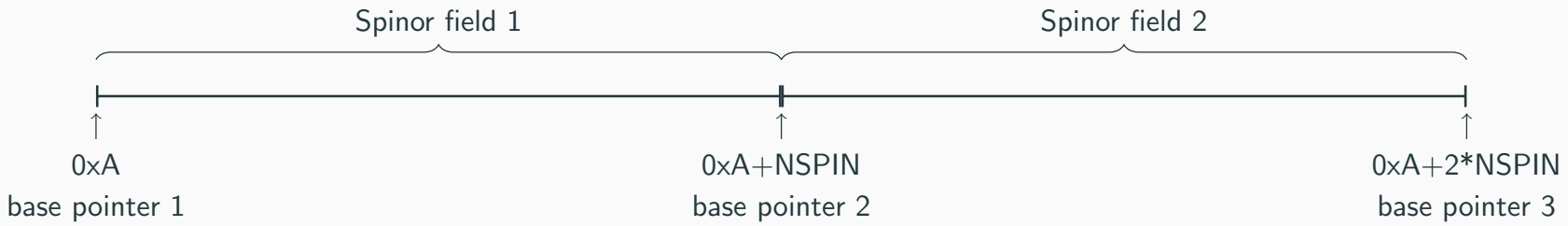


Figure 5: Current field allocation scheme.

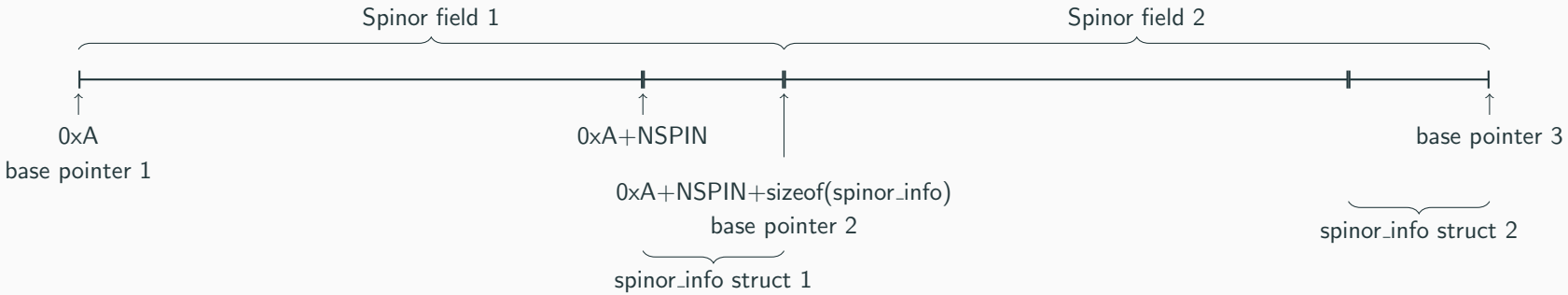


Figure 6: Proposed field allocation scheme.

spinor info struct

Information held by the `spinor_info` struct:

- Field status: `CPU_NEWER`, `GPU_NEWER`, `IN_SYNC`
- GPU pointer: pointer to field on the GPU (i.e. pointer to `ColorSpinorField` instance)
- Other information needed, eg. field size in bytes, ...

$\Longrightarrow$ Need to change allocation, reservation and release functions: `alloc_wsd()`, `reserve_wsd()`, `release_wsd()` + their single precision variants

Procedure

- Functions within openQxD still operate on base pointers (in the same way as before!) $\implies$ they all still work (no change needed)
- GPU functions now take the same CPU base pointer
  - Navigate to the `spinor_info` struct
  - Check if field needs to be transferred
  - Transfer if needed
  - Obtain GPU field pointer from info struct
  - Continue with GPU field
  - Function body
  - Update status field in info struct
  - (Transfer field(s) back to CPU)

```
// PhD Student Alice wrote this code
int main(int argc, char *argv[]) {
   spinor_dble **wsd;
   [...]

   alloc_wsd(N+1); // N = 10
   sd = reserve_wsd(N+1);

   for (i=0; i<N; i++) {
      random_sd(VOLUME, sd[i], 1.0);              // CPU
      scale_dble(VOLUME, 3.0, sd[i]);             // CPU
      // transfer sd[i], sd[N] (H2D) -> solved
      Dw_dble(0.0, sd[i], sd[N]);                 // GPU
      // transfer sd[i], sd[N] (D2H)
      // transfer sd[i], sd[N] (H2D) -> solved
      Dw_dble(0.0, sd[N], sd[i]);                 // GPU
      // transfer sd[i], sd[N] (D2H)
      r = norm_square_dble(VOLUME, 1, sd[i]);     // CPU
   }

   release_wsd();
}
```

Explicit transfers

We could write explicit transfer functions everywhere where needed

- `transfer_H2D(field); // transfer when needed`
- `transfer_D2H(field);`

Disadvantages:

- Makes code cluttered with these calls
- Makes code hard to read
- Makes code hard to maintain (eg: what if a new function was implemented on the GPU?)
- Is a lot of effort (we have many functions in openQxD!)
- If we forget one, code is not consistent anymore

Implicit transfers

- Problem 1: Every function within openQxD that operates on spinor fields has to transfer fields (if needed by the info struct).
- Problem 2: Every function within openQxD that operates on spinor fields has to maintain the spinor info struct.

$\rightarrow$ Ideally: Solve these problems without touching any spinor manipulation function already implemented in openQxD.

$\rightarrow$ Ideally: Without touching any existing code.

Implicit transfers

- Memory protection
- Signals (IPC)

Impicit Tranfers

Memory protection (posix):

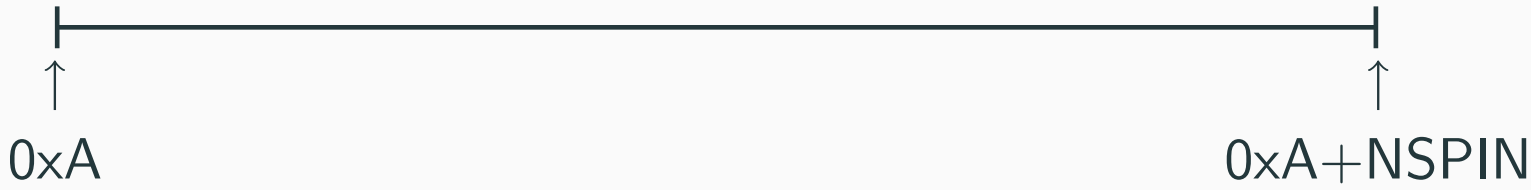
- System call `mprotect()`
- Protect already allocated pages of memory
- Throw segmentation fault upon accessing protected memory

Signals (posix):

- Process can receive signals from other processes (or the OS)
- We can register a signal handler to react upon a signal
  - Signature: `void handler(int signo, siginfo_t *info, void*)`
  - `signo`: signal number

    9 `SIGKILL` (not catchable)
    11 `SIGSEGV` (catchable)
    15 `SIGTERM` (catchable)
    ...
  - `info`: information struct about the signal

Memory protection



Memory protection

Memory protection

Memory protection

→ Accessing 0xB throws SEGV

→ Enter registered signal handler

- Check if it's a memory protection violation
- Navigate to spinor info struct
- Check if field is up2date
- Remove memory protection
- Transfer field if needed
- Update info struct for next GPU function acting on the field
- Exit signal handler and continue code

→ Code continues with up2date field!

Memory protection

- Existing spinor manipulation functions openQxD → untouched (except `alloc,reserve,release_wsd`)
- Existing code written in openQxD → set `GPU_OFFLOADING`, else untouched
- Function implementations on GPU → need to write explicit transfers (but that's OK)

```
#include "quda_utils.h"

// PhD Student Alice wrote this code
int main(int argc, char *argv[]) {
   spinor_dble **wsd;
   [...]

   alloc_wsd(N+1); // N = 10
   sd = reserve_wsd(N+1);

   for (i=0; i<N; i++) {
      random_sd(VOLUME, sd[i], 1.0);              // CPU
      scale_dble(VOLUME, 3.0, sd[i]);             // CPU
      Dw_dble(0.0, sd[i], sd[N]);                 // GPU
      Dw_dble(0.0, sd[N], sd[i]);                 // GPU
      r = norm_square_dble(VOLUME, 1, sd[i]);     // CPU
   }

   release_wsd();
}
```

- Or by compile time flag `GPU_OFFLOADING`

Summary

- Current Status: Inversion, Eigensolvers on GPU
- Next steps: Optimisations
- Further development: Field unification
- Further development: Implicit transfers

Discussion

- Other requirements by people working with the code?
- Other requirements for HMC?
- Contractions on GPU?
- Future directions of the code? (Also regarding new openQCD releases)
- CI/CD: automation of existing checks
- ...