My Project

Generated by Doxygen 1.9.8

Chapter 1

CONTRIBUTING

This documentation assumes you have basic knowledge in C/C++ syntax and are familiar with PauliArray's Python code.

There is currently a reformatting and renaming effort throughout the project. Names like 'paulicpp' will be changed to something more agnostic and appropriate in the near future.

1.0.1 Code Convention

`{MOD}` is a module's name. There can be multiple modules with different names compiled to the same directories.

- `/_src` Contains the C++ source code (.hpp) for every module.
- `/src/bindings` Contains the pybind11 bindings necessary for the translation to and from Python.
- `src/build` Contains the compiled dynamic shared libraries and stub files.

1.0.1.1 Architecture Overview

```
graph TB
    subgraph "Python Layer"
        A[Python User Code]
        B[NumPy Arrays]
        C[Python API]
    end

    subgraph "Binding Layer"
        D[pybind11 Bindings<br/>*_bindings.cpp]
        E[Type Conversion<br/>py::array <-> C++]
        F[Function Exposure<br/>PYBIND11_MODULE]
    end

    subgraph "C++ Core Layer"
        G[Header Files<br/>*.hpp]
        H[Function Operations]
        I[OpenMP Threading]
        J[GIL Management]
    end

    subgraph "Build System"
        K[CMakeLists.txt]
        L[Compiler<br/>clang/gcc]
        M[Dynamic Libraries<br/>.so/.dylib/.dll]
    end

    A --> B
    B --> C
    C --> D
    D --> E
    E --> F
    F --> G
    G --> H
    H --> I
    G --> J

    K --> L
    L --> M
    M --> D
```

1.0.1.2 Build Requirements

To build this library from source, you will need:

- Python 3.10+
- GCC or Clang that supports at least C++20
- A 64-bit Unix-like system
- (Optional) libomp/OpenMP

Note: In theory, Windows can compile the entire project with no issues. However, since MSVC is a completely different beast to GCC or Clang, no build options have been added to support it. MinGW would also probably work.

1.0.2 Nomenclature

1.0.2.1 Z2R

Also know as 'Voids', 'bit_string'.
Is defined as: $\mathbb{Z}_2$ Rows.

- '$\mathbb{Z}_2$' denotes the cyclic group of integers modulo 2 (i.e the set {0, 1}).
- 'Rows' states that the structure is a row (list) of elements.

Thus, the structure is a long list of binary 1s and 0s. This packed representation significantly reduces memory usage and enables efficient bitwise operations.

- Python: `z = np.array([0, 1, 1 0], dtype=np.int8)`. This takes 4 bytes of memory
- C++: `z = 0b0110`. This takes 1 byte of memory

1.0.2.2 CZ2M

Is defined as: C $\mathbb{Z}_2$ Matrix
This name is in reference to the C++ module which operates on Z2Rs to produce matrices.

1.0.2.3 The bitwise_[...] prefix

Any functions with this prefix must be declared inside of `bitops.hpp` and adhere to the following:

1. Must accept any size and shape of NDArrays, as long as every input is the same shape and contiguous.
2. Must return an NDArray of the same shape, size, and dtype as the inputs.
3. Follows as closely as possible NumPy's functions of the same name.

The only exceptions to rule 2 are `bitwise_count()` and `bitwise_dot()`, which both compress the last dimension of the array.

1.0.2.4 General Coding Standards

Avoid Broadcasting in C++: Broadcasting is a complex process that required substantial engineering effort from the NumPy team to implement efficiently. Rather than reimplementing ourself broadcasting in C++, either:

- Perform broadcasting in Python before passing data to C++
- Use pybind11's `vectorize` functionality when dealing with simple functions

Avoid Uncontiguous Data: The speedup made by this library is mainly due to the contiguity of NDArrays, and how this property can be exploited with extreme compiler optimizations (SIMD) or by careful implementation of certain patterns and structure. For uncontiguous data, either: TODO: EXPLAIN CONTIGUITY IN DETAIL!

- Rearange the data in Python before passing it to C++
- Explicitly state that slowdowns may occur (idk).

Memory Management: C++ lacks garbage collection or Rust's borrow checker, so you must manually manage memory efficiently:

- Track pointer lifetimes carefully to avoid dangling pointers
- Prefer zero-copy operations whenever possible
- Only allocate new memory when necessary

Type Specifications:

- Use `py::array_t<TYPE>` for typed arrays (e.g., `py::array_t<float>` for NDArrays of floats)
- Use generic `py::array` for Voids (packed binary data)

Binary Data Types: Since C++ has no native 'binary' type, a substitute one must be chosen instead. Use these types because they guarantee exact bit lengths and support both arithmetic and binary operations:

- `uint64_t`: 64-bit unsigned integer (for large data inputs)
- `uint8_t`: 8-bit unsigned integer (for small data)
- `int64_t`: 64-bit signed integer. Use ONLY for returning outputs

Avoid:

- `std::byte` (no arithmetic support) TODO: Clarify size garantees
- `unsigned char` (no size guarantee)
- `unsigned long long int` (unnecessarily verbose)

Do not use `using namespace std;` While convenient, namespace imports create ambiguity in low-level code. Both the standard library and pybind11 may define functions with identical names (e.g., `make_tuple()`), which would result in unpredicable function calls. Explicit namespace prefixes ensure:

- Functions are called from the correct library
- Code behavior is predictable
- Future maintainers understand the code's intent

1.0.2.5 Docstrings

A Python docstring such as:

```
 python

def add_integers(a:int, b:int) -> int:
    """
    Calculates the sum of two integers
    Args:
        a (int): The first operand
        b (int): The second operand
    Returns:
        int: The sum of both inputs
    """
    return a + b
```

Is roughly equivalent to the Doxygen comment:

```
 c++
/**
* @brief Calculates the sum of two integers
*
* @param a The first integer operand
* @param b The second integer operand
* @return int The sum of both inputs
*/
int add_integers(int a, int b) {
    return a + b;
}
```

Additional Doxygen tags include `@deprecated`, `@attention`, and `@warning`, which are used throughout the project where appropriate.

1.0.2.6 Auto-formatter

Since Black is only usable for Python files, we instead use `clang-format`, a highly customizable formatter that we can adjust to look more or less like Black.
All of the options are found inside of the `.clang-format` file. As a base, we use LLVM's coding format and override some specific options:

- `UseTab: Never`: Always forces spaces for indentation
- `IndentWidth: 4`: Use 4-space standard for indentation
- `IndentPPDirectives: AfterHash`: Indent preprocessor directives after their '#'

1.1 Interacting with Python

We use `pybind11` to interface between Python and C++. Pybind11 was chosen over alternatives like Boost.↩ Python for its:

- Seamless NumPy NDArray integration
- Superior performance
- Modern C++ support

Include these headers at the top of files that interact with Python:

```
 C++
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>
namespace py = pybind11;
```

Note: Omit `<pybind11/numpy.h>` if you don't need to pass NumPy arrays.

1.1.0.1 Writing Functions

C++ requires a lot more work and boilerplate in order to get the same result as in Python. The following functions both flip every bit of data inside of an NDArray:
Python: https://numpy.org/doc/stable/reference/generated/numpy.bitwise_↩ invert.html#numpy.bitwise_invert

```
 python
def bitwise_not(voids: NDArray) -> NDArray:
    int_strings = voids_to_int_strings(voids)
    new_int_strings = np.invert(int_strings)
    new_voids = int_strings_to_voids(new_int_strings)

    return new_voids
```

C++:

```
 C++
inline py::array bitwise_not(py::array voids) {
    auto buffer = voids.request();
    const uint64_t *ptr_64 = std::bit_cast<uint64_t *>(buffer.ptr);
    py::array result_arr = py::array(voids.dtype(), buffer.shape);
    // ...
    uint64_t *result_ptr_64 = std::bit_cast<uint64_t *>(result_arr.request().ptr);
    // ...
    for (int i = 0; i < TOTAL_NUM_64_CHUNKS; i++) {
        result_ptr_64[i] = ~ptr_64[i];
    }
    return result_arr
}
```

Lets look at the function declaration:

- `inline` means that the following function will be *expanded* instead of *called* from any other functions. What this means in practice is that the external call will be entirely replaced with the actual code of this function. This increases binary file sizes but reduces traditional function call overhead.
- `py::array` declares an object that is a generic NDArray with no specific type.

And as for the code within the function:

- `auto buffer = voids.request()`: Creates a buffer object with type inferred at compile time. The `request()` method returns a zero-copy view of the NDArray's data when possible.

- `std::bit_cast<uint64_t *>` is a near zero-overhead reinterpretation of the data underneath.
- `py::array result_arr = py::array(voids.dtype(), buffer.shape)`: Creates a new NDArray with the same dtype and shape as the input. This array can be returned directly to Python.
- `ptr_out_64[i] = ~ptr_64[i]` inverts the chunk of 64 bits and assigns the result to our result pointer

1.1.0.2 Exposing Functions to Python (Bindings)

All C++ functions must be declared in their module's binding file to be accessible from Python. Lets say we want to add `bitwise_not` to our module,

```
 ++
#include "your_source_file.hpp"

PYBIND11_MODULE(your_module_name, m) {
    m.doc() = "The description of your module";

    m.def("bitwise_not", &bitwise_not,
          "Flips each bit in an NDArray",
          py::arg("voids"));
}
```

1.2 Optimization Efforts

1.2.1 General

In most cases, we cast the data of a Voids array to `uint64_t` via `std::bit_cast<>`. This is done because in most modern computers, 64 bits is the largest natively supported integer size. Even when an array's dimension is smaller than 64 bits, treating it as a sequence of raw 64-bit blocks enables highly optimized bitwise manipulation and reduces overhead.

1.2.2 Multi-threading

To fully utilize available hardware, we implement multi-threading wherever safe and beneficial. OpenMP provides simple syntax with powerful results while maintaining code readability.

1.2.2.1 Using OpenMP

Before adding OpenMP, make it **certain** that your code is thread-safe. Data races and memory corruption are difficult to debug. While mutexes can provide thread safety over otherwise unsafe operations, they often cause performance degradation in small functions (our primary use case). Use mutexes cautiously and verify they provide net benefits.
As for the syntax, most *for(...)* loops in the project are prefaced with:

```
 C++
#ifdef USE_OPENMP
    #pragma omp parallel for if (local_variable >= SOME_MACRO) schedule(static)
#endif
    for (i=0; i<local_variable; i++) {
        // Code that does something which could be multi-threaded
    }
```

- `#ifdef` is a preprocessor directive that only compiles it's code block if the specific macro is defined. In this case, `USE_OPENMP` is our own macro, and is only ever defined within CMake's compiling instruction. This is necessary since if OpenMP is not installed but still tries to compile OMP-specific pragmas, the compiler will throw out an error and exit.
- `#pragma omp parallel for` says that we'll be using OpenMP's directives, which will pass the next *for(...)* loop to multiple threads.
- `if (local_variable >= SOME_MACRO)` assures that the multithreading only occurs if the statement evaluates to True. Usually this is with a counter or size of a data point compared to an arbitrarily chosen constant.
- `schedule(static)` assign each loop iterations to threads in a even, round-robin distribution.

More keywords exist, but they are specific to certain behaviors that are much less common in this project

1.2.2.2 Understanding the GIL

Python's Global Interpreter Lock allows only one thread to execute Python bytecode at a time. This simplifies Python's memory management but prevents true multi-threading. Only multiprocessing can achieve true parallelism under the GIL.
However, it is possible to get freed from the shackles of the GIL within C++. This can be done by declaring a section like this:

```
 ++
{
  py::gil_scoped_release release;
  // very heavy CPU loop (Real multithreading allowed)
}
// The GIL is automaticaly reaquired here
```

Releasing the GIL has some overhead, so it is only useful when code needs to be truly parallelized. **Release the GIL when:**

- Operations are CPU-intensive and long-running
- No Python API calls are needed
- Using OpenMP or manual threading

See `unordered_unique()`'s management of the GIL and OpenMP for more information. It is the very last function inside of `paulicpp/pauliarray/src/paulicpp.hpp`
Note: The Python language is working towards removing the GIL. This could take many years before it is accomplished, buf if/when it finished, there will likely be some breakage around any parts explicitly managing the GIL. See PEP 703

1.2.3 Building

WIP

1.2.3.1 pyproject.toml

`scikit-build-core` is the build backend as Flit not currently compatible with custom C++ compilation steps.
sasssaaa

1.2.3.2 CMake

CMake is a cross-platform build tool. It has clean integration with pybind11 and `scikit-build-core` for `pip install` support.
The configuration file is `CMakeLists.txt`, found in the project root.
Build Options:

- `USE_OPENMP`: Enable/disable OpenMP multi-threading
  - Default: `ON`
  - Disable manually in `CMakeLists.txt` or via pip: `pip install . --config-settings=cmake.define.USE_OPENMP=OFF`

1.3 Name

1.3.1 Why C++?

1. Performance: Most operations can be sped-up with either precise bit-on-bit algorithms and/or extreme compiler optimizations (like SIMD). Results are usually between 1.5-9x faster than pure NumPy
   (a) Memory: Handling Pauli strings within C++ allows for packed representations with next to no compromises. Less RAM

1.3.2 3. Flexibility: C++ offers total control over what a function needs to do. This is especially useful when XYZ

1.4 External tools and libraries

The tools used for this project are:

- pybind11
- pybind-stubgen
- CMake
- clang-format

1.4.1 Why pybind11?

Pybind11 was chosen for its minimal boilerplate, seamless NumPy integration, and modern C++ support. It allows us to expose C++ functions and classes to Python with very little overhead, making it easy to maintain and extend the codebase.
Since pybind is chosen as the interface for this project, the stub generator is obviously pybind-stubgen. However, some problems have appeared with it, notably when trying to add external libraries. It may get replaced with mypy's stubgen in the future.
Other alternatives for optimizing code to work with Python are:

1. **Numba**
   - While Numba is excellent for accelerating generic Python code, our data structures are highly specialized and already optimized in C++. Numba would offer limited additional benefit and cannot easily accelerate C++ code.
2. **Cython**
   - Cython is powerful for optimizing Python code and interfacing with C, but it often requires rewriting parts (if not most) of functions to take full advantage of its features. At that point, we might as well rewrite everything in pure C/C++
3. **Boost.Python**
   - Boost.Python is feature-rich but introduces significant overhead and complexity. It requires more boilerplate code, demands loads more dependencies, and is generally slower compared to pybind11.

1.5 Data structures

Handling long lists of 1s and 0s is at the heart of this project, and needs to be done as efficiently as possible. The current way all data is managed is throught a packed representation of integers.
Example: `ex = 0b01100111` stored in a unsigned integer.

- Drawback: for extreme cases where almost all entries are consecutively 1s or 0s, this method is slower and takes more memory space than a sparse implementation
- Benefits:
  - Enables native bitwise ops (AND/OR/XOR/NOT), popcount, shifts, etc.
  - SIMD, simpler/faster multithreading on most operations

Binary list

- Example: `ex = [0,1,1,0,0,1,1,1]`
- Drawback: each element takes up a whole byte while only one bit is needed. Poor memory density and difficult SIMD exploitation makes this approach slow in almost any scenario.

Sparse intervals

- Example: `ex = [(1,3),(5,8)]` denotes runs of consecutive 1s.

- Drawback: good for very very sparse patterns but degrades rapidely to O(n). Enormous overhead for most cases.

Chapter 2

Project Name

Accelerator project which handles arrays of packed binary values.

2.1 Installation

To install this package, you can try by using pip:

```
 bash
pip install z2r_accel
```

If there is no available distribution available for your machine, you'll need to build it from source

2.1.1 Building from source

2.1.1.1 Requirements

- Python 3.10+
- GCC or Clang, with versions that support at least C++20
- Unix system
- OpenMP (recommended)

> [!IMPORTANT] If you are on MacOS, it's recommended to manually install Clang and OpenMP with `Brew` by using `brew install llvm libomp`.
>
> On most Linux distributions, GCC/clang are pre-installed and with OpenMP support. No need to do anything extra.

2.1.1.2 Building with OpenMP

Clone this repository, navigate to the the directory where `pyproject.toml` is located and do:

```
pip install .
```

This should automatically compile the C++ code based on your machine and link place it to the correct path.

2.1.1.3 Building without OpenMP

> [!WARNING] Once again, this is **not** the recommended way to install this library!

Clone this repository, navigate to the the directory where `pyproject.toml` is located and do:

```
 console
pip install . --config-settings=cmake.define.USE_OPENMP=OFF
```

2.2 Documentation

Internal documentation can be found on the `repos's wiki`.

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

File Documentation

4.1 bitops.h

```
00001 #pragma once
00002 #include <pybind11/numpy.h>
00003 #include <pybind11/pybind11.h>
00004 namespace py = pybind11;
00005
00006 #include <bit>
00007 #include <cstdint> // uint8_t
00008 #include <cstring>
00009 #include <iostream>
00010 #include <vector>
00011
00012 #ifdef USE_OPENMP
00013     #include <omp.h>
00014 #else
00015     #warning "OpenMP is not enabled"
00016 #endif
00017
00018 // bit_operations
00019 // cz2m
00020
00021 // z2row
00022 // z2r  = voids
00023
00024 // This threshold is completely arbitrary and can be tuned for performance depending on the
00025 // hardware.
00026 #define BOPS_THRESHOLD_PARALLEL 1'000'000
00027
00028 py::array bitwise_and(py::array z2r_1, py::array z2r_2);
00029 py::array bitwise_xor(py::array z2r_1, py::array z2r_2);
00030 py::array bitwise_or(py::array z2r_1, py::array z2r_2);
00031 py::array bitwise_not(py::array voids);
00032 py::array paded_bitwise_not(py::array voids, int num_qubits);
00033 py::object bitwise_count(py::array z2r_1);
00034 py::object bitwise_dot(py::array z2r_1, py::array z2r_2);
00035
00036
00049 template <typename Op> py::object bitwise_core(py::array z2r_1, py::array z2r_2, Op op) {
00050     auto buf1 = z2r_1.request();
00051     auto buf2 = z2r_2.request();
00052
00053     if (buf1.itemsize != buf2.itemsize) {
00054         throw std::runtime_error("Input arrays must have the same itemsize (dtype compatibility).");
00055     }
00056     if (buf1.size != buf2.size) {
00057         throw std::runtime_error("Input arrays must have the same size.");
00058     }
00059
00060     size_t total_bytes = buf1.size * buf1.itemsize;
00061     py::array z2r_out = py::array(z2r_1.dtype(), buf1.shape);
00062     auto buf_out = z2r_out.request();
00063
00064     // Cut the data into 64-bit chunks for faster proceCest ssing
00065     // ptr1_64 and ptr2_64 are the pointers to the input data, ptr_out_64 is the pointer to the
00066     // output data
00067     // TODO: With C++20, we could use std::assume_aligned, which could potentially lead to better
00068     // auto-vectorization by the compiler? ? Since NumPy cant guarantee alignement and we cant
00069     // afford to copy the data just to get better SIMD, I dont know if assume_aligned is worth
00070     // anything.
00071     const uint64_t *ptr1_64 = std::bit_cast<uint64_t *>(buf1.ptr);
00072     const uint64_t *ptr2_64 = std::bit_cast<uint64_t *>(buf2.ptr);
00073     int64_t *ptr_out_64 = std::bit_cast<int64_t *>(buf_out.ptr);
00074
```

```
00075     size_t num_u64_chunks = total_bytes / 8;
00076
00077 #ifdef USE_OPENMP
00078     #pragma omp parallel for if (num_u64_chunks >= BOPS_THRESHOLD_PARALLEL) schedule(static)
00079 #endif
00080     for (size_t i = 0; i < num_u64_chunks; ++i) {
00081         // Applies the bitwise operation.
00082         ptr_out_64[i] = op(ptr1_64[i], ptr2_64[i]);
00083     }
00084
00085     // Handle any bytes that don't fit into a 64-bit chunk (the tail)
00086     size_t tail_bytes = total_bytes % 8;
00087     if (tail_bytes > 0) {
00088         const uint8_t *ptr1_8 = std::bit_cast<uint8_t *>(buf1.ptr);
00089         const uint8_t *ptr2_8 = std::bit_cast<uint8_t *>(buf2.ptr);
00090         int8_t *ptr_out_8 = std::bit_cast<int8_t *>(buf_out.ptr);
00091
00092         size_t start_byte = num_u64_chunks * 8;
00093         for (size_t i = 0; i < tail_bytes; ++i) {
00094             ptr_out_8[start_byte + i] = op(ptr1_8[start_byte + i], ptr2_8[start_byte + i]);
00095         }
00096     }
00097
00098     // If the input arrays were actually scalars (shape == ()), return a scalar as well
00099     if (buf1.size == 1) {
00100         std::vector<ssize_t> shape0{};   // zero-dim
00101         std::vector<ssize_t> strides0{}; // must match ndim (0)
00102         py::array scalar(z2r_out.dtype(), shape0, strides0, buf_out.ptr,
00103                          z2r_out); // TODO: Check if this is zero-copy
00104         return scalar;
00105     }
00106
00107     return z2r_out;
00108 }
00109
00110
```

4.2 cz2m.h

```
00001 #pragma once
00002
00003 #include <pybind11/numpy.h>
00004 #include <pybind11/pybind11.h>
00005 namespace py = pybind11;
00006
00007 #include <complex>
00008 #include <cstdint> // uint8_t
00009 #include <cstring>
00010 #include <iostream>
00011 #include <numeric>
00012 #include <random>
00013 #include <unordered_map>
00014 #include <vector>
00015
00016 #include "bitops.h"
00017
00018 #ifdef USE_OPENMP
00019     #include <omp.h>
00020 #else
00021     #warning "OpenMP is not enabled"
00022 #endif
00023
00024
00025 #define FUNC_THRESHOLD_PARALLEL 100000
00026
00027 // Function declarations
00028 py::tuple tensor(py::array z2, py::array x2, py::array z1, py::array x1);
00029
00030 py::tuple compose(py::array z1, py::array x1, py::array z2, py::array x2);
00031
00032 py::array_t<bool> bitwise_commute_with(py::array z1, py::array x1, py::array z2, py::array x2);
00033
00034 py::tuple random_zx_strings(const std::vector<size_t> &shape);
00035
00036 py::object unique(py::array zx_voids, bool return_index = false, bool return_inverse = false,
00037                   bool return_counts = false);
00038
00039 py::tuple unordered_unique(py::array zx_voids);
00040
00041 py::array row_echelon(py::array voids, int num_qubits);
00042
00043 std::tuple<std::vector<int>, std::vector<int>, std::vector<std::complex<double>>>
00044 sparse_matrix_from_zx_voids(py::array z_voids, py::array x_voids, int num_qubits);
00045
00046 std::vector<std::complex<double>> get_phases(py::array z_voids, py::array x_voids);
```

```
00047
00048 py::array_t<std::complex<double» to_matrix(py::array z_voids, py::array x_voids, int num_qubits);
00049
00050 py::array transpose(py::array voids, int64_t num_bits = -1);
00051
00052 py::array matmul(py::array z2r_a, py::array z2r_b, int a_num_qubits, int b_num_qubits);
00053
00054 py::array concatenate(py::array x1, py::array x2, int axis = 0);
```

Index