PCCM Introduction

Author: FindDefinition (https://github.com/FindDefinition)

What is PCCM?

PCCM is a C++ code generation framework.

1. Users can write C++ class just like writing python class. All C++ weird stuffs, such as function declaration/implementation, namespace and dependency are handled automatically.
2. C++ template class/template meta programming can be completely replaced by PCCM. We can write meta program in python.
3. PCCM is a python-first library, you can bind a PCCM class by only one decorator: @pccm.pybind.mark. Python annotation will also be generated automatically.

PCCM basic: pccm.Class

Firstly we need to know the mean of python module id of a python class:

In a standard python project, the module id of class is it's import path. For example, you define a class named SpconvOps in spconv/csrc/sparse/all.py, you can import it in python by

```
from spconv.csrc.sparse.all import SpconvOps
```

So the module id of `SpconvOps` is `spconv.csrc.sparse.all`

pccm.Class, corresponding to C++ class, is a basic unit of PCCM. A pccm.Class will generate exactly one C++ class, it's namespace is exactly module id of pccm.Class (can be controlled by set explicit namespace root):

PCCM basic: pccm.Class (Members)

pccm.Class is treated as a unique class definition like a c++ class, so the __init__ function of pccm.Class can't have any arguments. In pccm build system, pccm.Class will be generated once.

To define c++ members, static consts, typedefs and enums, we can use **`add_typedef`, `add_member`, `add_static_const`, `add_enum_class`** in __init__.

We can also add includes for a pccm.Class.

```
class ThrustCustomAllocatorV2(pccm.Class):
    def __init__(self):
        super().__init__()
        self.add_dependency(TensorView)
        self.add_include("functional", "memory")
        self.add_member("alloc_func",
                        "std::function<std::uintptr_t(std::size_t)>",
                        pyanno="Callable[[int], int]")
        self.add_typedef("value_type", "char")
        self.add_static_const("kConstVal", "int", "5")
        self.add_enum_class("Mode", [("kConvolution", 0),
                                     ("kCrossCorrelation", 1)])
```

PCCM basic: pccm.Class (Function inside class)

To define a member function of a class, users need to define python functions with no argument, and wrap it with a decorator: pccm.member_function or pccm.static_function. In python function body, we use pccm.FunctionCode to write all needed stuffs of a c++ function.

In right picture, we create a pccm.FunctionCode instance, then use code.arg to declare c++ arguments, then use code.raw to write raw c++ code. Finally we declare return type and return code object.

```python
@pccm.pybind.mark
@pccm.static_function
def conv_iwo_012_to_abc(self):
    code = pccm.FunctionCode()
    code.arg("op_type", "int")
    code.raw(f"""
    if (op_type == {ConvOpType.kForward.value}){{
        return {{0, 1, 2}};
    }}
    if (op_type == {ConvOpType.kBackwardInput.value}){{
        return {{2, 1, 0}};
    }}
    if (op_type == {ConvOpType.kBackwardWeight.value}){{
        return {{1, 2, 0}};
    }}
    TV_THROW_RT_ERR("unknown op type",op_type);
    """)
    return code.ret("std::array<int, 3>")
```

PCCM basic: pccm.Class (Dependency)

PCCM use DAG to manage dependency graph of pccm.Class. Unlike python dep (import system), pccm use ***self*.add_dependency** to add a dependency of other pccm.Class in __init__. The argument must be Type[pccm.Class]. PCCM will create aliases for all dependency internally, so we can directly use dep class name in c++ code.

In right picture, Test2 depend on Test1. Test2 only need to add dependency, the include/class name alias will be handled internally. We can also find out the argument of add_dependency is a Type, not a object, this due to a pccm.Class is unique in a project.

We can use a dependency graph with arbitrary depth, we only have one requirement: it must be a DAG, i.e., no cycle dependency.

```python
class Test1(pccm.Class):
    def __init__(self):
        super().__init__()
        self.add_dependency(stl.STL)

    @pccm.member_function(const=True, virtual=True)
    def add(self):
        code = pccm.FunctionCode()
        code.arg("a, b", "int")
        code.raw("""
        if (a > 0){
            return b;
        }
        return a + b;
        """)
        return code.ret("int")

class Test2(pccm.Class):
    def __init__(self):
        super().__init__()
        self.add_dependency(Test1)

    @pccm.member_function
    def add(self):
        code = pccm.FunctionCode()
        code.arg("a, b", "int")
        code.raw("""
        auto tst1 = Test1();
        return tst1.add(a, b);
        """)
        return code.ret("int")
```

PCCM basic: pccm.Class (Code Generation)

After we finish a pccm.Class, it's time to generate c++ code.

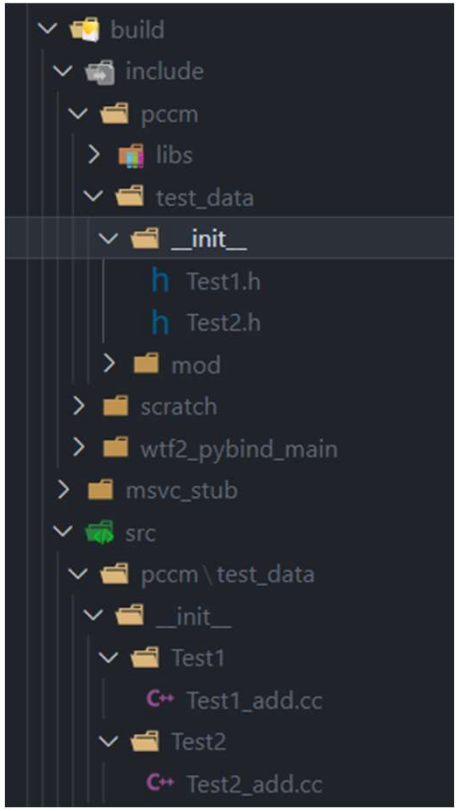
The generation process is described in following steps:

1. Resolve dependency, construct a DAG for all pccm.Class
2. Assign namespace for every pccm.Class based on python module id.
3. Iterate DAG in post-order (leaf first), generate code for every pccm.Class.

PCCM basic: pccm.Class (Code Gen For One Class)

For every pccm.Class, we generate a header file and several implementation files for all (not-header-only) functions, each impl file only have one function implementation.

If you declare an inline (or forceinline in CUDA) function, or specify header_only in pccm decorator, no impl file generated, the impl code will be placed in header.



PCCM basic: pccm.Class (Dependency Example)

```python
class ConvProblemCommon(pccm.Class):
    def __init__(self):
        super().__init__()
        self.add_dependency(bases.ConvEnum, TensorView)
```

See right pictures to see how dependency and aliases are created. PCCM will create an alias named class name for another pccm INSIDE current namespace. Then add includes for them.

With automatic alias generation, we can use dependency code without weird C++ namespace.

```cpp
#pragma once
#include <cumm/conv/bases/ConvEnum.h>
#include <cumm/common/TensorView.h>
namespace cumm {
namespace conv {
namespace params {
using ConvEnum = cumm::conv::bases::ConvEnum;
using TensorView = cumm::common::TensorView;
struct ConvProblemCommon {
};
} // namespace params
} // namespace conv
} // namespace cumm
```

PCCM basic: pybind11

Before talk about pybind11, I want to introduce my C++ programming pattern: Python First. With PCCM, I split the whole c++ project in to many function unit that can be bind by pybind11, then test/debug ONLY ONE PART of them in python. We know that we can't apply any static analysis tool to PCCM, but we receive ability that test part of C++ program in python.

PCCM basic: pybind11

We can bind a c++ function with only one decorator: pccm.pybind.mark. If pccm code generation detect this decorator, the bind code of this pccm.Class will be generated automatically. For whole project, each pccm.Class with bind decorator will generate one bind function, all bind functions will be called in a pybind main file.

In right pictures, we can see the generated pybind code. The PybindClassMixin is a helper mixin for add_pybind_member.

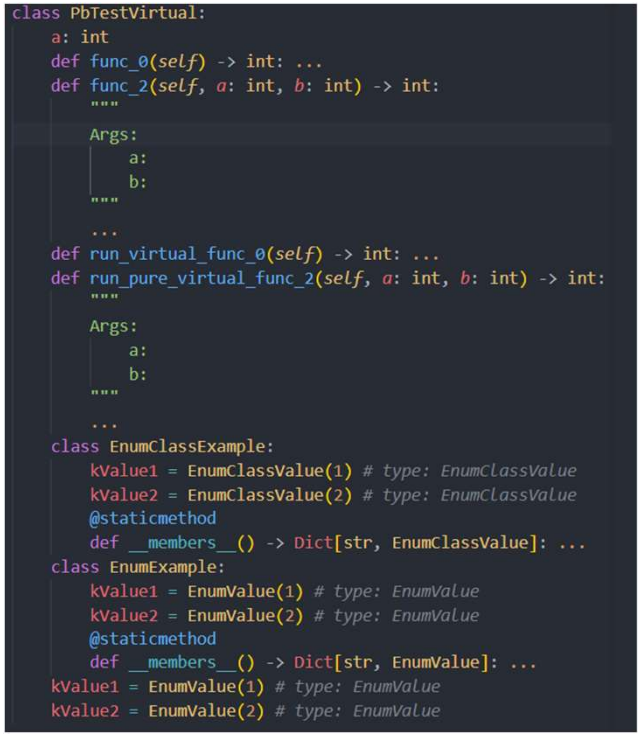
```python
class PbTestVirtual(pccm.Class, pccm.pybind.PybindClassMixin):
    def __init__(self):
        super().__init__()
        self.add_pybind_member("a", "int", "0")
        self.add_enum_class("EnumClassExample", [("kValue1", 1
                                                 ("kValue2", 2
        self.add_enum("EnumExample", [("kValue1", 1), ("kValue
        self.add_member("bbb", "EnumClassExample", "EnumClassE

    @pccm.pybind.mark(virtual=True)
    @pccm.member_function(virtual=True)
    def func_0(self):
        return pccm.FunctionCode("return 0;").ret("int")

    @pccm.pybind.mark(virtual=True)
    @pccm.member_function(virtual=True, pure_virtual=True)
    def func_2(self):
        code = pccm.FunctionCode("").ret("int")
        code.arg("a, b", "int")
        return code

    @pccm.pybind.mark
    @pccm.member_function
    def run_virtual_func_0(self):
        code = pccm.FunctionCode("return func_0();").ret("int"
        return code
```

```cpp
#include <pccm/test_data/mod/PyBindPbTestVirtual.h>
#include <pccm/test_data/mod/PbTestVirtual.h>
namespace pccm {
namespace test_data {
namespace mod {
struct PyPbTestVirtual : public PbTestVirtual {
  using PbTestVirtual::PbTestVirtual;

  virtual int func_0() override {PYBIND11_OVERRIDE(int, PbTest
  /**
   * @param a
   * @param b
   */
  virtual int func_2(int a, int b) override {PYBIND11_OVERRIDE
};
} // namespace mod
} // namespace test_data
} // namespace pccm
namespace pccm {
namespace test_data {
namespace mod {
using PbTestVirtual = pccm::test_data::mod::PbTestVirtual;
void PyBindPbTestVirtual::bind_PbTestVirtual(const pybind11::m
  pybind11::class_<pccm::test_data::mod::PbTestVirtual, pccm::
  module_PbTestVirtual.def(pybind11::init<>());
  module_PbTestVirtual.def("func_0", &pccm::test_data::mod::Pb
  module_PbTestVirtual.def("func_2", &pccm::test_data::mod::Pb
  module_PbTestVirtual.def("run_virtual_func_0", &pccm::test_d
  module_PbTestVirtual.def("run_pure_virtual_func_2", &pccm::t
  module_PbTestVirtual.def_readwrite("a", &pccm::test_data::mo
  pybind11::enum_<PbTestVirtual::EnumClassExample>(module_PbTe
    .value("kValue1", PbTestVirtual::EnumClassExample::kValue1
    .value("kValue2", PbTestVirtual::EnumClassExample::kValue2
  pybind11::enum_<PbTestVirtual::EnumExample>(module_PbTestVir
    .value("kValue1", PbTestVirtual::EnumExample::kValue1)
    .value("kValue2", PbTestVirtual::EnumExample::kValue2)
    .export_values();
}
} // namespace mod
```

PCCM basic: automatic annotation generation

In modern python, we can see that more and more project add annotations for their python/pybind projects. In PCCM, when a pccm.Class is binded, it's python annotation file (pyi) will be generated automatically.

The right picture shows generated anno code in an example showed before.



```
class PbTestVirtual:
    a: int
    def func_0(self) -> int: ...
    def func_2(self, a: int, b: int) -> int:
        """
        Args:
            a:
            b:
        """
        ...
    def run_virtual_func_0(self) -> int: ...
    def run_pure_virtual_func_2(self, a: int, b: int) -> int:
        """
        Args:
            a:
            b:
        """
        ...
    class EnumClassExample:
        kValue1 = EnumClassValue(1) # type: EnumClassValue
        kValue2 = EnumClassValue(2) # type: EnumClassValue
        @staticmethod
        def __members__() -> Dict[str, EnumClassValue]: ...
    class EnumExample:
        kValue1 = EnumValue(1) # type: EnumValue
        kValue2 = EnumValue(2) # type: EnumValue
        @staticmethod
        def __members__() -> Dict[str, EnumValue]: ...
    kValue1 = EnumValue(1) # type: EnumValue
    kValue2 = EnumValue(2) # type: EnumValue
```

PCCM basic: build system

The build system of PCCM isn't perfect, we offer two functions in PCCM: pccm.builder.build_pybind and pccm.builder.build_library. They accept multiple pccm.Class instance, build DAG and generate code for all pccm.Class.

```
cu = GemmMainUnitTest(SHUFFLE_SIMT_PARAMS + SHUFFLE_VOLTA_PARAMS +
                      SHUFFLE_TURING_PARAMS)
cu.namespace = "cumm.gemm.main"
convcu = ConvMainUnitTest(IMPLGEMM_SIMT_PARAMS + IMPLGEMM_VOLTA_PARAMS +
                          IMPLGEMM_TURING_PARAMS)
convcu.namespace = "cumm.conv.main"
objects_folder = None
if InWindows:
    # windows have command line limit, so we use objects_folder to reduce
    objects_folder = "objects"
pccm.builder.build_pybind([cu, convcu, SpconvOps()],
                          PACKAGE_ROOT / "core_cc",
                          namespace_root=PACKAGE_ROOT,
                          objects_folder=objects_folder,
                          load_library=False)
```

PCCM basic: pccm.ParameterizedClass

Now we will introduce the most important part of PCCM. Recall c++ template classes, we can add type/non-type to template arguments of template class, do some stuffs on these type/non-type, then save result type/non-type by assign a new alias/static const inside class body. This is called template meta programming. We can use meta programming to convert types or do computation in compile time.

There are lots of limitations of c++ template metaprogramming. Firstly, we can't write complex code such as if/for/while to handle types. Secondly, we can't add error handling for meta program. Thirdly, we can't declare meta-type of type arguments, so the intelligence engine can't inference results of meta program.

These drawbacks keep making me puzzled when I learn a heavily-templated HPC project (NVIDIA/CUTLASS). Finally, I decide to abandon CUTLASS code architecture, drop C++ template class, and design a new framework to handle template code, this is the origin of PCCM project.

PCCM basic: pccm.ParameterizedClass

WARNING: we will use pccm.ParamClass as an alias name of pccm.ParameterizedClass because it's too long.

The pccm.ParamClass inherit pccm.Class but have different code generation logic. A most important difference is that pccm.ParamClass can accept arguments in __init__, this means a pccm.ParamClass can be parameterized like a c++ template class. This feature make pccm.ParamClass not unique in a project, it's unique in **instance** level, not **type** level of pccm.Class.

PCCM instantiate pccm.ParamClass instance to namespace in current pccm.Class/ParamClass by generate code in an inner namespace.

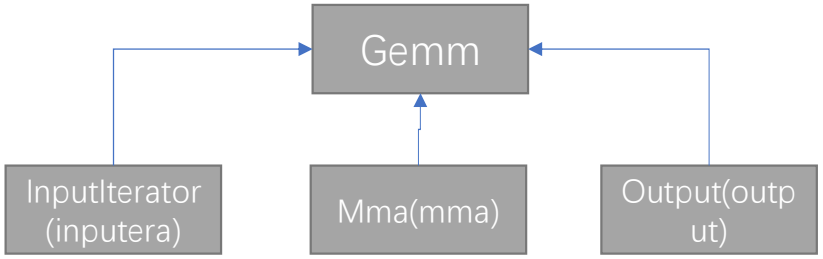
In picture below, in __init__ of a pccm.Class, we create a pccm.ParamClass instance, then use add_param_class to instantiate it to "maskiter" inner namespace and create an alias called "GlobalLoad", now we can access "GlobalLoad" in code in current pccm.Class.

```
self.global_load = memory.GlobalLoad(self.element_per_acc *
                                     self.dtype.itemsize())
self.add_param_class("maskiter", self.global_load, "GlobalLoad")
```

PCCM basic: pccm.ParameterizedClass (Code Gen)

To understand how pccm.ParamClass code generated, here us a real example (a complete gemm CUDA kernel, unnecessary code is deleted). We can see that this gemm class takes many arguments, create many child pccm.ParamClass, then instantiate them in different inner namespace.

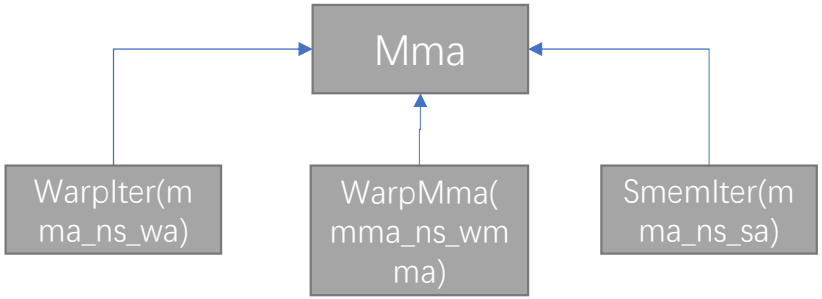
These children are also pccm.ParamClass. Same instantiation processes are applied to every child pccm.ParamClass.




```python
class GemmKernel(pccm.ParameterizedClass):
    def __init__(
            self,
            tile_shape: MetaArray[int],
            warp_tile_shape: MetaArray[int],
            num_stage: int,
            dtype_a: dtypes.DType,
            dtype_b: dtypes.DType,
            dtype_c: dtypes.DType,
            dtype_acc: dtypes.DType,
            dtype_comp: dtypes.DType,
            trans_a: bool,
            trans_b: bool,
            trans_c: bool,
            tensorop: Optional[TensorOpParams] = None,
            algo: GemmAlgo = GemmAlgo.Simt,
            splitk_serial: bool = False,
            splitk_parallel: bool = False,
            need_source: bool = True,
            shuffle_stride: ShuffleStrideType = ShuffleStrideType.NoShuffle,
            access_per_vector: int = 1):
        super().__init__()
        self.add_dependency(TensorView, TensorViewKernel, layout.RowMajor,
                            layout.ColumnMajor, GemmBasicKernel)
        self.add_param_class("gemmutils", GemmUtils(tile_shape), "GemmUtils")

        self.add_param_class("inpitera", self.input_spec.input_iter_a,
                             "InputIteratorA")
        self.add_param_class("inpiterb", self.input_spec.input_iter_b,
                             "InputIteratorB")

        self.add_param_class("gemm_smem_storage", self.gemm_smem_storage,
                             "BlockMmaStorage")
        self.add_param_class("out_smem_storage", self.out_smem_storage,
                             "OutputSmemStorage")
        self.add_param_class("gemm_params", self.gemm_params, "GemmParams")

        self.add_param_class("out_iter", self.output_spec.out_iter, "OutIter")
        self.add_param_class("out_iter_const", self.output_spec.const_out_iter,
                             "ConstOutIter")

        self.add_param_class("out_op", self.output_spec.output_op, "OutputOp")

        self.add_param_class("mma", self.mma_container, "Mma")
        self.add_param_class("output", self.output, "Output")
```

PCCM basic: pccm.ParameterizedClass (Code Gen)

In previous page, the “Mma” class is a child pccm.ParamClass, same as before, mma create (or just use pccm.ParamClass instance from arguments) and instantiate child pccm.ParamClass in its inner namespace.

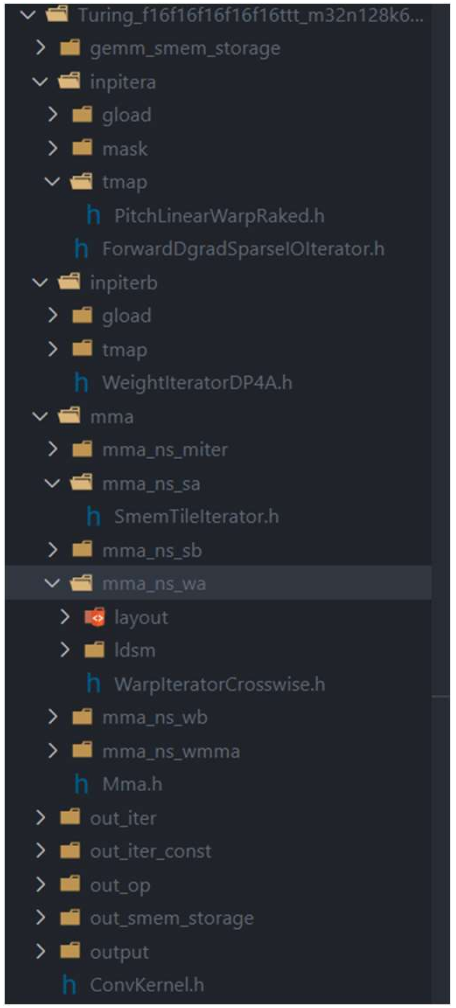



```python
class Mma(pccm.ParameterizedClass):
    """a strange behavier exists in ptx compiler
    if we construct iterators in main kernel, compiled code is different from
    construct them inside a class.
    """
    def __init__(self,
                 dtype_acc: dtypes.DType,
                 partk: int,
                 num_stage: int,
                 spec: bases.Mma,
                 smem_storage: BlockMmaStorage,
                 first_input_clear: bool = True,
                 clear_mask: bool = True,
                 mask_sparse: bool = False,
                 increment_k_first=False,
                 is_sparse_wgrad: bool = False):
        super().__init__()
        self.dtype_acc = dtype_acc
        miter = MaskIGemmIterator(increment_k_first)
        self.add_param_class("mma_ns_miter", miter, "MaskIGemmIterator")

        self.add_param_class("mma_ns_wa", spec.warp_iter_a, "WarpIterA")
        self.add_param_class("mma_ns_wb", spec.warp_iter_b, "WarpIterB")
        self.add_param_class("mma_ns_sa", spec.smem_iter_a, "SmemIterA")
        self.add_param_class("mma_ns_sb", spec.smem_iter_b, "SmemIterB")
        if is_sparse_wgrad:
            self.add_param_class("gl_wgrad", GlobalLoad(4), "GlobalLoad")
        self.add_param_class("mma_ns_gm", smem_storage, "GemmStorage")
        self.accumulator_fragment = array_type(dtype_acc,
                                               spec.accumulator_size)
        self.add_param_class("mma_ns_ia", self.input_spec.input_iter_a,
                             "InputIteratorA")
        self.add_param_class("mma_ns_ib", self.input_spec.input_iter_b,
                             "InputIteratorB")
        self.add_param_class("mma_ns_wmma", spec.warp_mma, "WarpMma")

        self.add_member("warp_iter_A", "WarpIterA")
        self.add_member("warp_iter_B", "WarpIterB")
        self.add_member("smem_iter_A", "SmemIterA")
        self.add_member("smem_iter_B", "SmemIterB")
```

PCCM basic: pccm.ParameterizedClass (Code Gen)

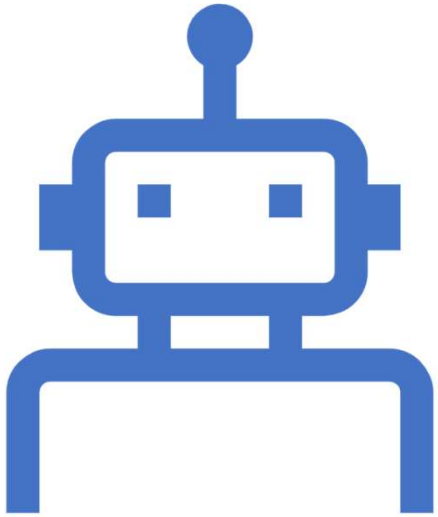
The right picture shows generated headers finally. The nested pccm.ParamClass become an instantiation tree.

Unlike c++ template classes, pccm.ParamClass based metaprogramming use full featured python as meta language, enable complex type handling and compile-time computing, error handling and full-featured intelligence engine support in meta program.



PCCM basic: pccm.ParameterizedClass (Usage)

pccm.ParamClass inherits pccm.Class, so it supports all features of pccm.Class except c++ inherit, don't use inherit in pccm.ParamClass.



PCCM examples

Spconv: Sparse Convolution Library,
https://github.com/traveller59/spconv

Cumm: CUda Matrix Multiple Library,
https://github.com/FindDefinition/cumm

Thanks!