gristmill Documentation

Release 0.4.0

Jinmo Zhao and Gustavo E Scuseria

Sep 29, 2017

CONTENTS:

CHAPTER

ONE

INTRODUCTION

The `gristmill` package is a pure Python package based on drudge for automatic code generation and optimization, with emphasis on tensor contraction operations.

CHAPTER

TWO

RELEASE HISTORY

2.1 0.2.0

This is mostly a bug fix release. Problems in the handling of bounds in Fortran printer and in the treatment purely scalar intermediates without any external indices are fixed. And the handling of summations are improved with less intermediates by removing duplicated and shallowly-defined ones. Most importantly, the automatic result checker has been fixed.

2.2 0.3.0

This release primarily features the addition of product optimization strategies. It was hoped that the new default `SEARCHED` will provide big improvement over the previous behaviour, which can now be accessed by `BEST`. Unfortunately, the improvement is only marginal. But the new strategy `GREEDY` could be used for large problems defying deeper optimizations.

2.3 0.4.0

This is a very small release. A bug on scalar intermediates is fixed by Roman Schutski and a helper function `mangle_base` is added to mangle name of indexed bases in code printing with greater ease.

CHAPTER

THREE

API REFERENCE

The `gristmill` package can be divided into two orthogonal parts,

The evaluation optimization part, which transforms tensor definitions into a mathematically equivalent definition sequence with less floating-point operations required.

The code generation part, which takes tensor definitions, either optimized or not, into computer code snippets.

3.1 Evaluation Optimization

`gristmill.`**`optimize`**(*computs: typing.Iterable[drudge.drudge.TensorDef], substs=None, interm_fmt='tau^{}', simplify=True, strategy=<Strategy.SEARCHED: 2>*) → typing.List[drudge.drudge.TensorDef]

Optimize the valuation of the given tensor contractions.

This function will transform the given computations, given as tensor definitions, into another list computations mathematically equivalent to the given computation while requiring less floating-point operations (FLOPs).

Parameters

- **`computs`** – The computations, can be given as an iterable of tensor definitions.
- **`substs`** – A dictionary for making substitutions inside the sizes of ranges. All the ranges need to have size in at most one undetermined variable after the substitution so that they can be totally ordered.
- **`interm_fmt`** – The format for the names of the intermediates.
- **`simplify`** – If the input is going to be simplified before processing. It can be disabled when the input is already simplified.
- **`strategy`** – The optimization strategy, as explained in *`Strategy`*.

`class` `gristmill.`**`Strategy`**

The optimization strategy for tensor contractions.

This enumeration type gives possible options for the optimization strategy for tensor contractions. Supported values includes,

`GREEDY` The contraction will be optimized greedily. This should only be used for large inputs where the other strategies cannot finish within a reasonable time.

`BEST` The global minimum of each tensor contraction will be found by the advanced algorithm in gristmill. And only the optimal contraction(s) will be kept for the summation optimization.

`SEARCHED` The same strategy as `BEST` will be attempted for the optimization of contractions. But all evaluations searched in the optimization process will be kept and considered in subsequent summation optimizations.

ALL All possible contraction sequences will be considered for all contractions. This can be extremely slow. But it might be helpful for manageable problems.

gristmill.**verify_eval_seq**(*eval_seq: typing.Sequence[drudge.drudge.TensorDef], res: typing.Sequence[drudge.drudge.TensorDef], simplify=False*) → bool

Verify the correctness of an evaluation sequence for the results.

The last entries of the evaluation sequence should be in one-to-one correspondence with the original form in the `res` argument. This function returns `True` when the evaluation sequence is symbolically equivalent to the given raw form. When a difference is found, `ValueError` will be raised with relevant information.

Note that this function can be very slow for large evaluations. But it is advised to be used for all optimizations in mission-critical tasks.

Parameters

- **eval_seq** – The evaluation sequence to verify, can be the output from *optimize()* directly.
- **res** – The original result to test the evaluation sequence against. It can be the input to *optimize()* directly.
- **simplify** – If simplification is going to be performed after each step of the back-substitution. It is advised for larger complex evaluations.

gristmill.**get_flop_cost**(*eval_seq: typing.Iterable[drudge.drudge.TensorDef], leading=False, ignore_consts=True*)

Get the FLOP cost for the given evaluation sequence.

This function gives the count of floating-point operations, addition and multiplication, involved by the evaluation sequence. Note that the cost of copying and initialization are not counted. And this function is only applicable where the amplitude of the terms are simple products.

Parameters

- **eval_seq** – The evaluation sequence whose FLOP cost is to be estimated. It should be given as an iterable of tensor definitions.
- **leading** – If only the cost terms with leading scaling be given. When multiple symbols are present in the range sizes, terms with the highest total scaling is going to be picked.
- **ignore_consts** – If the cost of scaling with constants can be ignored. $2x_i y_j$ could count as just one FLOP when it is set, otherwise it would be two.

3.2 Code generation

class gristmill.**BasePrinter**(*scal_printer: sympy.printing.printer.Printer, indexed_proc_cb=<function BasePrinter.<lambda>>, add_globals=None, add_filters=None, add_tests=None, add_templ=None*)

The base class for tensor printers.

__init__(*scal_printer: sympy.printing.printer.Printer, indexed_proc_cb=<function BasePrinter.<lambda>>, add_globals=None, add_filters=None, add_tests=None, add_templ=None*)

Initializes a base printer.

Parameters

- **scal_printer** – The SymPy printer for scalar quantities.

- **indexed_proc_cb** – It is going to be called with context nodes with `base` and `indices` (in both the root and for each indexed factors, as described in *transl()*) to do additional processing. For most tasks, *mangle_base()* can be helpful.

transl(*tensor_def: drudge.drudge.TensorDef*) → types.SimpleNamespace
Translate tensor definition into context for template rendering.

This function will translate the given tensor definition into a simple namespace that could be easily used as the context in the actual Jinja template rendering.

The context contains fields,

base A printed form for the base of the tensor definition.

indices A list of external indices. For each entry, keys `index` and `range` are present to give the printed form of the index and the range it is over. For convenience, `lower`, `upper`, and `size` have the printed form of lower/upper bounds and the size of the range. We also have `lower_expr`, `upper_expr`, and `size_expr` for the unprinted expression of them.

terms A list of terms for the tensor, with each entry being a simple namespace with keys,

sums A list of summations in the tensor term. Its entries are in the same format as the external indices for tarrays.

phase + sign or − sign. For the phase of the term.

numerator The printed form of the numerator of the coefficient of the term. It can be a simple `1` string.

denominator The printed form of the denominator.

indexed_factors The indexed factors of the term. Each is given as a simple namespace with key `base` for the printed form of the base, and a key `indices` giving the indices to the key, in the same format as the `indices` field of the base context.

other_factors Factors which are not simple indexed quantity, given as a list of the printed form directly.

The actual content of the context can also be customized by overriding the *proc_ctx()* in subclasses.

proc_ctx(*tensor_def: drudge.drudge.TensorDef, term: typing.Union[drudge.term.Term, NoneType], tensor_entry: types.SimpleNamespace, term_entry: typing.Union[types.SimpleNamespace, NoneType]*)
Make additional processing of the rendering context.

This method can be override to make additional processing on the rendering context described in *transl()* to perform additional customization or to make more information available.

It will be called for each of the terms during the processing. And finally it will be called again with the term given as None for a final processing.

By default, the indexed quantities nodes are processed by the user-given call-back.

render(*templ_name: str*, *ctx: types.SimpleNamespace*) → str
Render the given context for the given template.

Meaningful subclass methods can call this function for actual functionality.

__weakref__
list of weak references to the object (if defined)

gristmill.**mangle_base**(*func*)
Mangle the base names in the indexed nodes in template context.

A function taking the printed string for an indexed base and a list of its indices, as described in *BasePrinter.transl()*, to return a new mangled base name can be given to get a function call-back compatible with the `indexed_proc_cb` argument of *BasePrinter.__init__()* constructor.

This function can also be used as a function decorator.

class gristmill.**ImperativeCodePrinter**(*scal_printer: sympy.printing.printer.Printer*, *print_indexed_cb*, *global_indent=1*, *indent_size=4*, *max_width=80*, *line_cont=''*, *breakable_regex='(\s*[+-]\s*)'*, *stmt_end=''*, *add_globals=None*, *add_filters=None*, *add_tests=None*, *add_templ=None*, ***kwargs*)

Printer for automatic generation of naive imperative code.

This printer supports the printing of the evaluation of tensor expressions by simple loops and arithmetic operations.

This is mostly a base class that is going to be subclassed for different languages. For each language, mostly just the options for the language could be given in the super initializer. Most important ones are the printer for the scalar expressions and the formatter of loops, as well as some definition of literals and operators.

__init__(*scal_printer: sympy.printing.printer.Printer*, *print_indexed_cb*, *global_indent=1*, *indent_size=4*, *max_width=80*, *line_cont=''*, *breakable_regex='(\\s*[+-]\\s*)'*, *stmt_end=''*, *add_globals=None*, *add_filters=None*, *add_tests=None*, *add_templ=None*, ***kwargs*)

Initialize the automatic code printer.

scal_printer A sympy printer used for the printing of scalar expressions.

print_indexed_cb It will be called with the printed base, and the list of indices (as described in *BasePrinter.transl()*) to return the string for the printed form. This will be called after the given processing of indexed nodes.

global_indent The base global indentation of the generated code.

indent_size The size of the indentation.

max_width The maximum width for each line.

line_cont The string used for indicating line continuation.

breakable_regex The regular expression used to break long expressions.

stmt_end The ending of the statements.

index_paren The pair of parenthesis for indexing arrays.

All options to the base class *BasePrinter* are also supported.

proc_ctx(*tensor_def: drudge.drudge.TensorDef*, *term: typing.Union[drudge.term.Term, NoneType]*, *tensor_entry: types.SimpleNamespace*, *term_entry: typing.Union[types.SimpleNamespace, NoneType]*)

Process the context.

The indexed nodes will be printed by user-given printer and given to `indexed` attributes of the same node. Also the term contexts will be given an attribute named `amp` for the whole amplitude part put together.

print_eval(*ctx: types.SimpleNamespace*)

Print the evaluation of a tensor definition.

gristmill.**CCodePrinter**

alias of `wrapper`

class gristmill.**FortranPrinter**(*openmp=True*, ***kwargs*)

Fortran code printer.

In this class, just some parameters for the *new* Fortran programming language is fixed relative to the base *ImperativeCodePrinter*.

__init__(*openmp=True*, ***kwargs*)
Initialize a Fortran code printer.

The printer class, the name of the template, and the line continuation symbol will be set automatically.

print_decl_eval(*tensor_defs: typing.Iterable[drudge.drudge.TensorDef]*, *decl_type='real'*, *explicit_bounds=False*) → typing.Tuple[typing.List[str], typing.List[str]]
Print Fortran declarations and evaluations of tensor definitions.

Parameters

- **tensor_defs** – The tensor definitions to print.
- **decl_type** – The type to be declared for the tarrays.
- **explicit_bounds** – If the lower and upper bounds should be written explicitly in the declaration.

Returns

- *decls* – The list of declaration strings.
- *evals* – The list of evaluation strings.

print_decl(*ctx*, *decl_type*, *explicit_bounds*)
Print the Fortran declaration of the LHS of a tensor definition.

A string will be returned that forms the naive declaration of the given tarrays as local variables.

class gristmill.**EinsumPrinter**(***kwargs*)
Printer for the einsum function.

For tensors that are classical tensor contractions, this printer generates code based on the NumPy `einsum` function. For contractions supported, the code from this printer can also be used for Tensorflow.

__init__(***kwargs*)
Initialize the printer.

All keyword arguments are forwarded to the base class *BasePrinter*.

print_eval(*tensor_defs: typing.Iterable[drudge.drudge.TensorDef]*, *base_indent=4*) → str
Print the evaluation of the tensor definitions.

Parameters

- **tensor_defs** – The tensor definitions for the evaluations.
- **base_indent** – The base indent of the generated code.

Returns

Return type The code for evaluations.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

B

C

E

F

G

I

M

O

P

R

S

T

V