MShadow-Notes Documentation

Release

Kaleidoscope

January 22, 2017

Contents

CHAPTER 1

Tensor.h

`./tensor.h` consists of header file of tensor data structure and functions

- CPU and GPU
- Shape
- Stream
- TRValue `:public expr::RValueExp<Container, DType>`
- Tensor `:public TRValue<Tensor<Device, dimension, DType>, Device, dimension, DType>`
- Tensor<Device, 1, DType> `:public TRValue<Tensor<Device, 1, DType>, Device, 1, DType>`

1.1 CPU and GPU

The struct `cpu` and `gpu` here is to guarantee the success of checkings before final evaluation.

```
// device name CPU
struct cpu {
  // whether this device is CPU or not
  static const bool kDevCPU = true;
  // device flag number, identifies this device
  static const int kDevMask = 1 << 0;
};
// device name GPU
struct gpu {
  // whether this device is CPU or not
  static const bool kDevCPU = false;
  // device flag number, identifies this device
  static const int kDevMask = 1 << 1;
};
```

1.2 Shape

The struct `Shape` here is to provide a flexible component for the construction of class Tensor.

1.2.1 Member variables

The `kDimension` defines the dimension of current shape, while the `kSubdim` can be used to infer the lowest dimension in the Tensor.

```
static const int kDimension = dimension;
static const int kSubdim = dimension - 1;
```

With the information of `kDimension`, we can construct an unsigned array `shape_` to store the shape of Tensor.

```
// typedef unsigned index_t;
index_t shape_[kDimension];
```

1.2.2 Constructor

After the difinition of three member variables, we can write the constructor to create a struct `Shape`.

```
// default constructor, do nothing
MSHADOW_XINLINE Shape(void) {}
// constuctor
MSHADOW_XINLINE Shape(const Shape<kDimension> &s) {
  #pragma unroll
  for (int i = 0; i < kDimension; ++i) {
    this->shape_[i] = s[i];
  }
}
```

The code `const Shape<kDimension> &s` as a variable of constructor makes sure that only struct `Shape` with same `kDimension` can be allowed to initiate the new ones.

1.2.3 Overloaded Operator

Operator `[]` is overloaded to return the sub-dimension of required index.

```
MSHADOW_XINLINE index_t &operator[](index_t idx) {
  return shape_[idx];
}
// the returned value is constant
MSHADOW_XINLINE const index_t &operator[](index_t idx) const {
  return shape_[idx];
}
```

Operator == and != are overloaded to check whether two struct `Shape` are same.

```
// Shape<kDimension> implicitly check whether the input variable has the same size
MSHADOW_XINLINE bool operator==(const Shape<kDimension> &s) const {
  #pragma unroll
  for (int i = 0; i < kDimension; ++i) {
    if (s.shape_[i] != this->shape_[i]) return false;
  }
  return true;
}
// return whether two shape not equal
// s:         the shape to compare against
MSHADOW_XINLINE bool operator!=(const Shape<kDimension> &s) const {
  return !(*this == s);
}
```

Operator << is overloaded to output the `shape_` of `Shape`.

```
template<int dim>
  friend std::ostream &operator<<(std::ostream &os, const Shape<dim> &shape);
```

Above is only a declaration, while its definition is in `./tensor_cpu-inl.h`.

```
template<int ndim>
inline std::ostream &operator<<(std::ostream &os, const Shape<ndim> &shape) {
  os << '(';
  for (int i = 0; i < ndim; ++i) {
    if (i != 0) os << ',';
    os << shape[i];
  }
  // python style tuple
  if (ndim == 1) os << ',';
  os << ')';
  return os;
}
```

1.2.4 Member Functions

Size

The `size` function returns the product of all sub-dimensions. e.g. `(5,3,6) -> 90`

```
MSHADOW_XINLINE size_t Size(void) const {
  size_t size = this->shape_[0];
  #pragma unroll
  for (int i = 1; i < kDimension; ++i) {
    size *= this->shape_[i];
  }
  return size;
}
```

FlatTo1D

The `FlatTo1D` function returns a `Shape` with `kDimension=1` and its dimension equals to the product of original `Shape`. e.g. `(5,3,6) -> (90)`

```
MSHADOW_XINLINE Shape<1> FlatTo1D(void) const {
  Shape<1> s;
  s[0] = this->Size();
  return s;
}
```

FlatTo2D

The `FlatTo2D` function returns a `Shape` with `kDimension=2` and its first dimension equals to the product of original `Shape` instead of the lowest dimension, which is put into its second dimension.

```
the reason of doing so will be considered and explained later
```

```
MSHADOW_XINLINE Shape<2> FlatTo2D(void) const {
  Shape<2> s;
  s.shape_[1] = this->shape_[kDimension - 1];
  index_t ymax = 1;
  #pragma unroll
  for (int i = 0; i < kDimension - 1; ++i) {
    ymax *= this->shape_[i];
  }
  s.shape_[0] = ymax;
  return s;
}
```

ProdShape

The function `ProdShape` returns the product of shape in range `[dimstart, dimend)`.

```
MSHADOW_XINLINE index_t ProdShape(int dimstart, int dimend) const {
  index_t num = 1;
  #pragma unroll
  for (int i = dimstart; i < dimend; ++i) {
    num *= this->shape_[i];
  }
  return num;
}
```

SubShape

The function `SubShape` return a new shape, whose `kDimension` is the minus `1` of original one. e.g. `(3,2,6,4) -> (2,6,4)`

```
Since it is majorly built for cuda, its effectiveness will be considered and explained later
```

```
MSHADOW_XINLINE Shape<kSubdim> SubShape(void) const {
  Shape<kSubdim> s;
  // for cuda
  #pragma unroll
  for (int i = 0; i < kSubdim; ++i) {
    s.shape_[i] = this->shape_[i + 1];
  }
  return s;
}
```

Slice

The function `Slice` return a new shape, whose `kDimension` is the `dimend-dimstart` of original one.

```
template<int dimstart, int dimend>
MSHADOW_XINLINE Shape<dimend - dimstart> Slice(void) const {
  Shape<dimend - dimstart> s;
  #pragma unroll
  for (int i = dimstart; i < dimend; ++i) {
    s[i - dimstart] = this->shape_[i];
  }
  return s;
}
```

It can used in the following way, e.g. `(3,4,5,6,7) -> (5,6,7)`.

```
// usage
Shape<5> ss = Shape5(3,4,5,6,7);
Shape<3> sss = ss.Slice<2,5>();
cout << sss <<endl;
// output (5,6,7)
```

1.2.5 Useful Construction

According to the `usage` instruction above, we introduce several construction functions for struct `Shape` as APIs.

```
// useful construction functions to generate shape
MSHADOW_XINLINE Shape<1> Shape1(index_t s0) {
  Shape<1> s; s[0] = s0;
  return s;
}

MSHADOW_XINLINE Shape<2> Shape2(index_t s0, index_t s1) {
  Shape<2> s; s[0] = s0; s[1] = s1;
  return s;
}

MSHADOW_XINLINE Shape<3> Shape3(index_t s0, index_t s1, index_t s2) {
  Shape<3> s;
  s[0] = s0; s[1] = s1; s[2] = s2;
  return s;
}

MSHADOW_XINLINE Shape<4> Shape4(index_t s0, index_t s1,
                                index_t s2, index_t s3) {
  Shape<4> s;
  s[0] = s0; s[1] = s1; s[2] = s2; s[3] = s3;
  return s;
}

MSHADOW_XINLINE Shape<5> Shape5(index_t s0, index_t s1, index_t s2,
                                index_t s3, index_t s4) {
  Shape<5> s;
  s[0] = s0; s[1] = s1; s[2] = s2; s[3] = s3; s[4] = s4;
  return s;
}
```

1.3 Stream

The `Stream` here is only a dummy implementation for CPU, we left it for further discussion when we run into the implementation of GPU.

```
template<typename Device>
struct Stream {
  // this is only a dummy implementation for CPU
  // for GPU, the actual implementation will be specialized in tensor_gpu-inl.h

  //wait for all the computation associated with this stream to complete
  inline void Wait(void) {}
```

```
  // query whether the the stream is idle
  // return true if the stream is idle and all the job have been completed
  inline bool CheckIdle(void) {
    return true;
  }

  // create a blas handle
  inline void CreateBlasHandle() {}
};
```

1.4 TRValue

`:public expr::RValueExp<Container, DType>`

This is Tensor RValue, which is also the super type of all kinds of possible tensors.

```
The meaning of its existence is essential for the understanding of MShadow framework, thus
we postpone its discussion into the core part of this tutorial
```

```
template<typename Container, typename Device, int dimension, typename DType>
struct TRValue: public expr::RValueExp<Container, DType> {
};
```

1.5 Tensor

`:public TRValue<Tensor<Device, dimension, DType>, Device, dimension, DType>`

The struct `Tensor` is the key element in MShadow.

```
//in /mshadow/base.h

//#ifndef MSHADOW_DEFAULT_DTYPE
//#define MSHADOW_DEFAULT_DTYPE = default_real_t
//#endif

//typedef float default_real_t;
//as a result, the default data type of Tensor is float
template<typename Device, int dimension, typename DType MSHADOW_DEFAULT_DTYPE>
struct Tensor: public TRValue<Tensor<Device, dimension, DType>, Device, dimension, DType> {
};
```

```
// Trival Usage
Tensor<cpu, 3> ts(data, Shape3(2,5,2));
```

1.5.1 Member Variables

The variable `kDevCPU` indicates in which type of device the data are stored. And the `kSubdim` is same as in the struct `Shape`.

```
static const bool kDevCPU = Device::kDevCPU;
static const int  kSubdim = dimension - 1;
```

The pointer `dptr_` points to the data wherever it is stored. Besides, the shape of data is controlled by the struct `Shape` to make it flexible to handle.

```
DType *dptr_;
Shape<dimension> shape_;
```

At last, the `stride_` variable is used to deal with pitch allocation in gpu or sse (align x dimension to 64bit) for efficiency.

```
the concept of stream is important for GPU devices, we leave the discussion to there.
```

```
index_t stride_;
// stream where the computation lies
// stream is a device dependency concept
Stream<Device> *stream_;
```

1.5.2 Constructor

```
It is worth noting that the `stride_` is default initialized to be the lowest dimension of
`Shape`. The reason of doing it is remained to be examined and discussed.
```

```
// default constructor
MSHADOW_XINLINE Tensor(void) : stream_(NULL) {}

// constructor from shape
MSHADOW_XINLINE Tensor(const Shape<dimension> &shape): shape_(shape), stream_(NULL) {}

// constructor from data pointer and shape, without stride
MSHADOW_XINLINE Tensor(DType *dptr, const Shape<dimension> &shape)
    : dptr_(dptr), shape_(shape), stride_(shape[kSubdim]), stream_(NULL) {}

// constructor from data pointer, shape and stream, without stride
MSHADOW_XINLINE Tensor(DType *dptr, const Shape<dimension> &shape,Stream<Device> *stream)
    : dptr_(dptr), shape_(shape), stride_(shape[kSubdim]), stream_(stream) {}

// constructor from data pointer, shape, stride and stream
MSHADOW_XINLINE Tensor(DType *dptr, const Shape<dimension> &shape, index_t stride, Stream<Device> *st
    : dptr_(dptr), shape_(shape), stride_(stride), stream_(stream) {}
```

1.5.3 Member Functions

MSize and MemSize

The function `MSize` returns the memory cost of specified tensor, including the aligned x dimension (so it starts with the value of largest dimension of tensor). While the function `MemSize` returns the memory starting from the `startdim`.

```
MSHADOW_XINLINE size_t MSize(void) const {
  return this->MemSize<0>();
}

template<int startdim>
MSHADOW_XINLINE size_t MemSize(void) const {
  size_t memsz = this->stride_;
  #pragma unroll
```

```
  for (int i = startdim; i < kSubdim; ++i) {
    memsz *= this->shape_[i];
  }
  return memsz;
}
```

size

The function `size` return the `shape` of the specified sub-dimension.

```
MSHADOW_XINLINE index_t size(index_t idx) const {
  return shape_[idx];
}
```

FlatTo1D and FlatTo2D

The functions `FlatTo1D` and `FlatTo2D` return a new tensor with same data (unchanged `dptr_`), but different shape (refer to `FlatTo1D` and `FlatTo2D` in the context of `Shape`).

```
MSHADOW_XINLINE Tensor<Device, 1, DType> FlatTo1D(void) const {
  return Tensor<Device, 1, DType>(dptr_, shape_.FlatTo1D(), stride_, stream_);
}

MSHADOW_XINLINE Tensor<Device, 2, DType> FlatTo2D(void) const {
  return Tensor<Device, 2, DType>(dptr_, shape_.FlatTo2D(), stride_, stream_);
}
```

Slice

The function `Slice` returns a new Tensor, which is a subset of the highest dimension. e.g. `(128,3,224,224) -> (64,3,224,224)`

```
MSHADOW_XINLINE Tensor<Device, dimension, DType>
Slice(index_t begin, index_t end) const {
  Shape<dimension> s = this->shape_;
  s[0] = end - begin;
  return Tensor<Device, dimension, DType>(dptr_ + this->MemSize<1>() * begin, s, stride_, stream_);
}
```

1.5.4 Overloaded Operators

Operator `[]`

The operator `[]` is overloaded to return the corresponding index in the highest dimension of `Tensor`.

The code `dptr_ + this->MemSize<1>() * idx` is to fetch the `idx` sub-tensor in `Tensor`, e.g. `(128,3,224,224)[5] -> 5-th (3,224,224)`

```
MSHADOW_XINLINE Tensor<Device, kSubdim, DType> operator[](index_t idx) const {
  return Tensor<Device, kSubdim, DType>(dptr_ + this->MemSize<1>() * idx, shape_.SubShape(), stride_,
}
```

Operator =

The operator = is overloaded to be assignment operator when the `rhs` (right hand side) is also a `Tensor` variable.

```
// implement the assignment of same type
inline Tensor<Device, dimension, DType> &
operator=(const Tensor<Device, dimension, DType> &exp) {
  dptr_ = exp.dptr_;
  shape_ = exp.shape_;
  stride_ = exp.stride_;
  stream_ = exp.stream_;
  return *this;
}
```

However, if the `rhs` is a `scalar`, e.g. `3.0f`, or a `Exp` (expression) type, the operator = is overloaded to trigger the computation, which calls the `__assign` function, defined in its grandfather class `RValueExp`.

```
// we trigger computation at here
template<typename E, int etype>
inline Tensor<Device, dimension, DType> &
operator=(const expr::Exp<E, DType, etype> &exp) {
  return this->__assign(exp);
}

inline Tensor<Device, dimension, DType> &
operator=(const DType &exp) {
  return this->__assign(exp);
}
```

It is worth noting that there are several other assignment related operators are overloaded, but in the grandfather class `RValueExp`. We will do analysis until reaching there.

1.6 Missing Explanations

1.6.1 the usage of `#pragma unroll`:

if the following `for loop` has a constant number of loops, the `for loop` will be expanded in the compile time to accelerate the process. e.g. `for(i = 1; i < 10; i++){...};` will be expanded

Otherwise, if the number of loop is undetermined, it will keep itself same as common loop e.g. `for(i = 1; i < n; i++){...};` will be the same. Since computer will not be able to know the exact value of `n` until the computation time.

1.7 Missing Component

1.7.1 Shape::ConvertLayout

`ConvertLayout` is left to the discussion of MxNet, which uses it to do `convolution`.

```
// Convert shape in src_layout to shape in dst_layout
inline Shape<4> ConvertLayout(const Shape<4>& src, int src_layout, int dst_layout) {
  Shape<4> dst;
  switch (src_layout) {
  case kNCHW:
```

```
    dst = src;
    break;
  case kNHWC:
    dst[0] = src[0];
    dst[2] = src[1];
    dst[3] = src[2];
    dst[1] = src[3];
    break;
  default:
    LOG(FATAL) << "Invalid layout for 4d shape " << src_layout;
  }
  Shape<4> dst2;
  switch (dst_layout) {
  case kNCHW:
    return dst;
  case kNHWC:
    dst2[0] = dst[0];
    dst2[1] = dst[2];
    dst2[2] = dst[3];
    dst2[3] = dst[1];
    break;
  default:
    LOG(FATAL) << "Invalid layout for 4d shape " << src_layout;
  }
  return dst2;
}
// Convert shape in src_layout to shape in dst_layout
inline Shape<5> ConvertLayout(const Shape<5>& src, int src_layout, int dst_layout) {
  Shape<5> dst;
  switch (src_layout) {
  case kNCDHW:
    dst = src;
    break;
  case kNDHWC:
    dst[0] = src[0];
    dst[2] = src[1];
    dst[3] = src[2];
    dst[4] = src[3];
    dst[1] = src[4];
    break;
  default:
    LOG(FATAL) << "Invalid layout for 5d shape " << src_layout;
  }
  Shape<5> dst2;
  switch (dst_layout) {
  case kNCDHW:
    return dst;
  case kNDHWC:
    dst2[0] = dst[0];
    dst2[1] = dst[2];
    dst2[2] = dst[3];
    dst2[3] = dst[4];
    dst2[4] = dst[1];
    break;
  default:
    LOG(FATAL) << "Invalid layout for 5d shape " << src_layout;
  }
  return dst2;
```

```
}
```

1.7.2 Tensor:set_stream and Tensor::CheckContiguous

These two functions are heavily related to GPU implementation, so they are left for further discussion.

```
inline void set_stream(Stream<Device> *stream) {
  this->stream_ = stream;
}

MSHADOW_XINLINE bool CheckContiguous(void) const {
  return this->shape_[dimension - 1] == stride_;
}
```

1.7.3 Tensor<Device, 1, DType>

We must respecialize struct `Tensor` in the `1D` situation, since the implementation of overloaded operator `[]` is different.

It can also be considered as a review of member variables and functions in original `Tensor`.

```
template<typename Device, typename DType>
struct Tensor<Device, 1, DType>:
      public TRValue<Tensor<Device, 1, DType>, Device, 1, DType> {
 public:
  DType *dptr_;
  Shape<1> shape_;
  index_t stride_;
  Stream<Device> *stream_;
  // constructor
  MSHADOW_XINLINE Tensor(void) : stream_(NULL) {}
  MSHADOW_XINLINE Tensor(const Shape<1> &shape)
      : shape_(shape), stream_(NULL) {}
  MSHADOW_XINLINE Tensor(DType *dptr, Shape<1> shape)
      : dptr_(dptr), shape_(shape), stride_(shape[0]), stream_(NULL) {}
  MSHADOW_XINLINE Tensor(DType *dptr, Shape<1> shape, Stream<Device> *stream)
      : dptr_(dptr), shape_(shape), stride_(shape[0]), stream_(stream) {}
  MSHADOW_XINLINE Tensor(DType *dptr, Shape<1> shape,
                         index_t stride, Stream<Device> *stream)
      : dptr_(dptr), shape_(shape), stride_(stride), stream_(stream) {}
  inline void set_stream(Stream<Device> *stream) {
    this->stream_ = stream;
  }
  MSHADOW_XINLINE Tensor<Device, 1, DType> FlatTo1D(void) const {
    return *this;
  }
  MSHADOW_XINLINE Tensor<Device, 2, DType> FlatTo2D(void) const {
    return Tensor<Device, 2, DType>(dptr_, shape_.FlatTo2D(), stride_, stream_);
  }
  MSHADOW_XINLINE Tensor<Device, 1, DType> Slice(index_t begin, index_t end) const {
    Shape<1> s;
    s[0] = end  - begin;
    return Tensor<Device, 1, DType>(dptr_ + begin, s, s[0], stream_);
  }
  MSHADOW_XINLINE bool CheckContiguous(void) const {
    return true;
```

```
  }
  MSHADOW_XINLINE size_t MSize(void) const {
    return shape_[0];
  }
  MSHADOW_XINLINE index_t size(index_t i) const {
    return shape_[0];
  }
  MSHADOW_XINLINE DType &operator[](index_t idx) {
    return dptr_[idx];
  }
  MSHADOW_XINLINE const DType &operator[](index_t idx) const {
    return dptr_[idx];
  }
  // implement the assignment of same type
  inline Tensor<Device, 1, DType> &
  operator=(const Tensor<Device, 1, DType> &exp) {
    dptr_ = exp.dptr_;
    shape_ = exp.shape_;
    stride_ = exp.stride_;
    stream_ = exp.stream_;
    return *this;
  }
  template<typename E, int etype>
  inline Tensor<Device, 1, DType> &
  operator=(const expr::Exp<E, DType, etype> &exp) {
    return this->__assign(exp);
  }
  inline Tensor<Device, 1, DType> &operator=(const DType &exp) {
    return this->__assign(exp);
  }
};
```

CHAPTER 2

Expression.h

`./expression.h` consists of the definitions of abstract expressions and their related expression templates.

- Type
- Exp
- ScalarExp: `public Exp<ScalarExp<DType>, DType, type::kMapper>`
- TypecastExp: `public Exp<TypecastExp<DstDType, SrcDType, EType, etype>, DstDType, etype>`
- TransposeExp: `public Exp<TransposeExp<EType, DType>, DType, type::kChainer>`
- ExpEngine
- RValueExp: `public Exp<Container, DType, type::kRValue>`
- DotExp: `public Exp<DotExp<TA, TB, ltrans, rtrans, DType>, DType, type::kComplex>`
- TernaryMapExp: `public Exp<TernaryMapExp<OP, TA, TB, TC, DType, etype>, DType, etype>`
- UnaryMapExp: `public Exp<UnaryMapExp<OP, TA, DType, etype>, DType, etype>`
- BinaryMapExp: `public Exp<BinaryMapExp<OP, TA, TB, DType, etype>, DType, etype>`

2.1 Type

Every expression in MShadow has to own its own type.

- `kRValue`: this expression directly correspnds to a data class, can be used to assign data.
- `kMapper`: this expression contains element-wise tensor operations, can map a expression to same shape.
- `kChainer`: this expression can be chained with other expressions. Most of functions in file `./extension/` are of this type.
- `kComplex`: othercase: e.g dot product.

```
namespace type {
// type expression type are defined as bitmask
// subtype relationshop kRValue < kMapper < kPull < kComplex

const int kRValue = 0;
```

```
const int kMapper = 1;
const int kChainer = 3;
const int kComplex = 7;
}
```

2.2 Exp

The struct `Exp` is the base class for every expression.

Each class inheritated from `Exp` has to put their type into `SubType`, while `DType` is the type of data being manipulated and `exp_type` is to indicate the expression type inheritated from it.

```
template<typename SubType, typename DType, int exp_type>
struct Exp {
 public:
  // return subtype instance of current class
  inline const SubType& self(void) const {
    return *static_cast<const SubType*>(this);
  }
  // return reference of subtype instance of current class
  inline SubType* ptrself(void) {
    return static_cast<SubType*>(this);
  }
};
```

2.3 ScalarExp

```
:  public Exp<ScalarExp<DType>, DType, type::kMapper>
```

The `ScalarExp` is to map a scalar type to a sub-class `ScalarExp` of expression type `Exp`.

For now, we can just think explicit constructor is majorly used for avoiding the implicit consturction of class in declaration. the reason we cannot make the constructor explicit here is that, if the constructor is explicit, the occurence of scalar in either its lhs or rhs will not call this constructor.

```
template<typename DType>
struct ScalarExp: public Exp<ScalarExp<DType>, DType, type::kMapper> {
  // scalar value
  DType scalar_;
  // implicit constructor, MUST NOT BE explicit
  ScalarExp(DType scalar) : scalar_(scalar) {}
};

// this is a type cast function that turns the scalar s with type Dtype into type ScalarExp
template<typename DType>
inline ScalarExp<DType> scalar(DType s) {
  return ScalarExp<DType>(s); // turn the scalar s with type Dtype into type ScalarExp
}
```

```
// usage
scalar<float>(10.0f)
// it is worthy noting that this process is generally implicitly performed
// (by overloaded operator in struct RValueExp)
```

2.4 TypecastExp

: public Exp<TypecastExp<DstDType, SrcDType, EType, etype>, DstDType, etype>

The TypecastExp is to explicitly change the DType of a expression type Exp.

```
template<typename DstDType, typename SrcDType, typename EType, int etype>
struct TypecastExp:
      public Exp<TypecastExp<DstDType, SrcDType, EType, etype>, DstDType, etype> {
  // expression to be typecasted
  const EType &exp;
  // constructor
  explicit TypecastExp(const EType &e) : exp(e) {}
};
// create an Typecast expression
template<typename DstDType, typename SrcDType, typename EType, int etype>
inline TypecastExp<DstDType, SrcDType, EType, (etype|type::kMapper)>
tcast(const Exp<EType, SrcDType, etype> &exp) {
  return TypecastExp<DstDType, SrcDType, EType, (etype|type::kMapper)>(exp.self());
}
```

```
// usage
float data[10];
int data1[10];

Tensor<cpu,2> mat(data,Shape2(5,2));
Tensor<cpu,2,int> mat1(data1,Shape2(5,2));

mat = 3.2f;
mat1 = tcast<int>(mat);

for (index_t i = 0; i < mat.size(0); ++i) {
  for (index_t j = 0; j < mat.size(1); ++j) {
      printf("%.2f ", mat[i][j]);
  }
  printf("\n");
}
for (index_t i = 0; i < mat1.size(0); ++i) {
  for (index_t j = 0; j < mat1.size(1); ++j) {
      printf("%d ", mat1[i][j]);
  }
  printf("\n");
}
// output:
// 3.20 3.20
// 3.20 3.20
// 3.20 3.20
// 3.20 3.20
// 3.20 3.20
// 3 3
// 3 3
// 3 3
// 3 3
// 3 3
```

2.5 TransposeExp

: public Exp<TransposeExp<EType, DType>, DType, type::kChainer>

The TransposeExp is generated by .T() function in class RValueExp.

Through the constructor function, a same type Tensor is generated.

```
The actual transpose actually happens in the `Eval()` function.
Whether its member function `T()` is useful or not remains to be checking.
```

```
template<typename EType, typename DType>
struct TransposeExp: public Exp<TransposeExp<EType, DType>,
                                DType, type::kChainer> {
  // expression to be transposed, the generated one is same
  const EType &exp;
  // constructor
  explicit TransposeExp(const EType &e) : exp(e) {}
  // transpose expression
  inline const EType &T(void) const {
    return exp;
  }
};
```

2.6 ExpEngine

ExpEngine is the struct that starts to interpret all of the expressions.

It is generated by member functions in RValueExp, while its implementation is in ./expr_engine-inl.h , which call the MapExp in ./tensor_cpu-inl.h or ./tensor_gpu-inl.h to begin the interpretion of expressions.

The details of its discussion is left to its implementation codes.

```
template<typename Saver, typename RValue, typename DType>
struct ExpEngine;
```

2.7 RValueExp

: public Exp<Container, DType, type::kRValue>

RValueExp is a special class, that is inheritated by TRValue (Tensor Rvalue), which is the super class of all kinds of tensors.

Since it explicitly set the exp_type (expression type) in Exp to kRValue during inheritance, it correspnds to every situation with a form dst () src, where dst is the destination to receive the result of computation, () is += / -= / *= / /= / =, and src can be a value or expression.

As a result, in this class, we have eight reloaded operator and two assign functions, with a extra transpose function.

```
template<typename Container, typename DType>
class RValueExp: public Exp<Container, DType, type::kRValue> {
 public:
  // transpose of a matrix, return transpose of current expression
```

```
// usage:  *.T()
inline const TransposeExp<Container, DType> T(void) const {
  return TransposeExp<Container, DType>(this->self());
}
// operator overload
// += for scalar
inline Container &operator+=(DType s) {
  ExpEngine<sv::plusto, Container, DType>::Eval(this->ptrself(), scalar<DType>(s));
  return *(this->ptrself());
}
// -= for scalar
inline Container &operator-=(DType s) {
  ExpEngine<sv::minusto, Container, DType>::Eval(this->ptrself(), scalar<DType>(s));
  return *(this->ptrself());
}
// *= for scalar
inline Container &operator*=(DType s) {
  ExpEngine<sv::multo, Container, DType>::Eval(this->ptrself(), scalar<DType>(s));
  return *(this->ptrself());
}
// /= for scalar
inline Container &operator/=(DType s) {
  ExpEngine<sv::divto, Container, DType>::Eval(this->ptrself(), scalar<DType>(s));
  return *(this->ptrself());
}
// assign for scalar
inline Container &__assign(DType s) {
  ExpEngine<sv::saveto, Container, DType>::Eval(this->ptrself(), scalar<DType>(s));
  return *(this->ptrself());
}
// assign for expression
template<typename E, int etype>
inline Container &__assign(const Exp<E, DType, etype> &exp) {
  ExpEngine<sv::saveto, Container, DType>::Eval(this->ptrself(), exp.self());
  return *(this->ptrself());
}
// not sure the difference to above function
inline Container &__assign(const Exp<Container, DType, type::kRValue> &exp);
// += for expression
template<typename E, int etype>
inline Container &operator+=(const Exp<E, DType, etype> &exp) {
  ExpEngine<sv::plusto, Container, DType>::Eval(this->ptrself(), exp.self());
  return *(this->ptrself());
}
// -= for expression
template<typename E, int etype>
inline Container &operator-=(const Exp<E, DType, etype> &exp) {
  ExpEngine<sv::minusto, Container, DType>::Eval(this->ptrself(), exp.self());
  return *(this->ptrself());
}
// *= for expression
template<typename E, int etype>
inline Container &operator*=(const Exp<E, DType, etype> &exp) {
  ExpEngine<sv::multo, Container, DType>::Eval(this->ptrself(), exp.self());
  return *(this->ptrself());
}
// /= for expression
template<typename E, int etype>
```

```
  inline Container &operator/=(const Exp<E, DType, etype> &exp) {
    ExpEngine<sv::divto, Container, DType>::Eval(this->ptrself(), exp.self());
    return *(this->ptrself());
  }
};
```

In every reloaded function, `ExpEngine` is used to trigger the `Eval` function, which use template `MapExp` to `MakePlan` as described above.

At last, the namespace `sv` is defined in the `./base.h` to perform the final assignment.

2.8 DotExp

```
:  public Exp<DotExp<TA, TB, ltrans, rtrans, DType>, DType, type::kComplex>
```

The `DotExp` is an expression to do matrix computation between two `Tensors`.

```
template<typename TA, typename TB, bool ltrans, bool rtrans, typename DType>
struct DotExp: public Exp<DotExp<TA, TB, ltrans, rtrans, DType>, DType, type::kComplex> {
  const TA &lhs_;
  const TB &rhs_;
  DType scale_; // scale over result

  explicit DotExp(const TA &lhs, const TB &rhs, DType scale)
      : lhs_(lhs), rhs_(rhs), scale_(scale) {}
};
```

The generation of struct `DotExp` is triggered by the function `dot()` with following four different styles by consideration of transpose.

- both lhs and rhs are without transpose

```
template<typename TA, typename TB, typename DType>
inline DotExp<TA, TB, false, false, DType>
dot(const RValueExp<TA, DType> &lhs, const RValueExp<TB, DType> &rhs) {
  return DotExp<TA, TB, false, false, DType>(lhs.self(), rhs.self(), DType(1.0f));
}
```

- lhs is with transpose, while rhs is not

```
template<typename TA, typename TB, typename DType>
inline DotExp<TA, TB, true, false, DType>
dot(const TransposeExp<TA, DType> &lhs, const RValueExp<TB, DType> &rhs) {
  return DotExp<TA, TB, true, false, DType>(lhs.exp, rhs.self(), DType(1.0f));
}
```

- rhs is with transpose, while lhs is not

```
template<typename TA, typename TB, typename DType>
inline DotExp<TA, TB, false, true, DType>
dot(const RValueExp<TA, DType> &lhs, const TransposeExp<TB, DType> &rhs) {
  return DotExp<TA, TB, false, true, DType>(lhs.self(), rhs.exp, DType(1.0f));
}
```

- both lhs and rhs are with transpose

```
template<typename TA, typename TB, typename DType>
inline DotExp<TA, TB, true, true, DType>
dot(const TransposeExp<TA, DType> &lhs, const TransposeExp<TB, DType> &rhs) {
```

```
  return DotExp<TA, TB, true, true, DType>(lhs.exp, rhs.exp, DType(1.0f));
}
```

```
the usage of batch_dot is unclear yet!
```

```
template<bool transpose_left, bool transpose_right, typename TA, typename TB, typename DType>
inline DotExp<TA, TB, transpose_left, transpose_right, DType>
batch_dot(const RValueExp<TA, DType> &lhs, const RValueExp<TB, DType> &rhs) {
  return DotExp<TA, TB, transpose_left, transpose_right, DType>(
    lhs.self(), rhs.self(), DType(1.0f));
}
```

2.9 TernaryMapExp

`public Exp<TernaryMapExp<OP, TA, TB, TC, DType, etype>, DType, etype>`

`TernaryMapExp` is designed to handle the ternary operation expression. Its member variables contain three different expressions as their original types.

```
template<typename OP, typename TA, typename TB, typename TC, typename DType, int etype>
struct TernaryMapExp: public Exp<TernaryMapExp<OP, TA, TB, TC, DType, etype>, DType, etype> {
  const TA &item1_;
  const TB &item2_;
  const TC &item3_;

  explicit TernaryMapExp(const TA &item1, const TB &item2, const TC &item3)
      :item1_(item1), item2_(item2), item3_(item3) {}
};
```

The struct `TernaryMapExp` is generated by the function `MakeExp`.

```
template<typename OP, typename TA, typename TB, typename TC, typename DType, int ta, int tb, int tc>
inline TernaryMapExp<OP, TA, TB, TC, DType, (ta|tb|tc|type::kMapper)>
MakeExp(const Exp<TA, DType, ta> &item1, const Exp<TB, DType, tb> &item2,
  const Exp<TC, DType, tc> &item3) {
  return TernaryMapExp<OP, TA, TB, TC, DType,
                       (ta|tb|tc|type::kMapper)>(item1.self(), item2.self(), item3.self());
}
```

The function `F<op>()` provides a short hand for `MakeExp()`, with a operator defined in namespace `op`, which provides `Map()` function to guide the way of evaluation.

```
template<typename OP, typename TA, typename TB, typename TC, typename DType, int ta, int tb, int tc>
inline TernaryMapExp<OP, TA, TB, TC, DType, (ta|tb|tc|type::kMapper)>
F(const Exp<TA, DType, ta> &item1, const Exp<TB, DType, tb> &item2,
 const Exp<TC, DType, tc> &item3) {
  return MakeExp<OP>(item1, item2, item3);
}
```

2.10 UnaryMapExp

`public public Exp<UnaryMapExp<OP, TA, DType, etype>, DType, etype>`

`UnaryMapExp` is designed to handle the unary operation expression. Its member variables contain only one expression as its original type.

```
template<typename OP, typename TA, typename DType, int etype>
struct UnaryMapExp: public Exp<UnaryMapExp<OP, TA, DType, etype>, DType, etype> {
  const TA &src_;
  explicit UnaryMapExp(const TA &src) : src_(src) {}
};
```

The struct `UnaryMapExp` is also generated by the function `MakeExp`.

```
template<typename OP, typename TA, typename DType, int ta>
inline UnaryMapExp<OP, TA, DType, (ta|type::kMapper)>
MakeExp(const Exp<TA, DType, ta> &src) {
  return UnaryMapExp<OP, TA, DType, (ta|type::kMapper)>(src.self());
}
```

Similarly, `UnaryMapExp` also has its own function `F<op>()` providing a short hand for `MakeExp()`, with a operator defined in namespace `op`, which provides `Map()` function to guide the way of evaluation.

```
template<typename OP, typename TA, typename DType, int ta>
inline UnaryMapExp<OP, TA, DType, (ta|type::kMapper)>
F(const Exp<TA, DType, ta> &src) {
  return MakeExp<OP>(src);
}
```

2.11 BinaryMapExp

```
:  public Exp<BinaryMapExp<OP, TA, TB, DType, etype>, DType, etype>
```

`BinaryMapExp` is designed to handle the binary operation expression. Its member variables contain two different expressions as their original type.

```
template<typename OP, typename TA, typename TB, typename DType, int etype>
struct BinaryMapExp: public Exp<BinaryMapExp<OP, TA, TB, DType, etype>,
                                DType, etype> {
  const TA &lhs_;
  const TB &rhs_;
  explicit BinaryMapExp(const TA &lhs, const TB &rhs): lhs_(lhs), rhs_(rhs) {}
};
```

The struct `BinaryMapExp` is also generated by the function `MakeExp`.

```
template<typename OP, typename TA, typename TB, typename DType, int ta, int tb>
inline BinaryMapExp<OP, TA, TB, DType, (ta|tb|type::kMapper)>
MakeExp(const Exp<TA, DType, ta> &lhs, const Exp<TB, DType, tb> &rhs) {
  return BinaryMapExp<OP, TA, TB, DType, (ta|tb|type::kMapper)>(lhs.self(), rhs.self());
}
```

Similarly, `BinaryMapExp` also has its own function `F<op>()` providing a short hand for `MakeExp()`, with a operator defined in namespace `op`, which provides `Map()` function to guide the way of evaluation.

```
template<typename OP, typename TA, typename TB, typename DType, int ta, int tb>
inline BinaryMapExp<OP, TA, TB, DType, (ta|tb|type::kMapper)>
F(const Exp<TA, DType, ta> &lhs, const Exp<TB, DType, tb> &rhs) {
  return MakeExp<OP>(lhs, rhs);
}
```

Moreover, the four basic operators + / − / * / / are also overloaded to make sure we can add an `Exp` type to `built-in scalar type`, e.g. `a+3.0f` where `a` is a `Tensor`. Since the constructor of `ScalarExp` is not explicit, the `3.0f` will be implicitly converted to type `ScalarExp`.

```
template<typename TA, typename TB, typename DType, int ta, int tb>
inline BinaryMapExp<op::plus, TA, TB, DType, (ta|tb|type::kMapper)>
operator+(const Exp<TA, DType, ta> &lhs, const Exp<TB, DType, tb> &rhs) {
  return MakeExp<op::plus>(lhs, rhs);
}

template<typename TA, typename TB, typename DType, int ta, int tb>
inline BinaryMapExp<op::minus, TA, TB, DType, (ta|tb|type::kMapper)>
operator-(const Exp<TA, DType, ta> &lhs, const Exp<TB, DType, tb> &rhs) {
  return MakeExp<op::minus>(lhs, rhs);
}

template<typename TA, typename TB, typename DType, int ta, int tb>
inline BinaryMapExp<op::mul, TA, TB, DType, (ta|tb|type::kMapper)>
operator*(const Exp<TA, DType, ta> &lhs, const Exp<TB, DType, tb> &rhs) {
  return MakeExp<op::mul>(lhs, rhs);
}

template<typename TA, typename TB, typename DType, int ta, int tb>
inline BinaryMapExp<op::div, TA, TB, DType, (ta|tb|type::kMapper)>
operator/(const Exp<TA, DType, ta> &lhs, const Exp<TB, DType, tb> &rhs) {
  return MakeExp<op::div>(lhs, rhs);
}
```

CHAPTER 3

Expr_engine-inl.h

`./expr_engine-inl.h` provides methods to evaluate all of the expressions.

- MakeTensorExp
- Plan
- MakePlan
- ExpInfo
- TypeCheck
- StreamInfo
- ShapeCheck
- ExpEngine

3.1 MakeTensorExp

```
:  public Exp<MakeTensorExp<SubType, SrcExp, dim, DType>, DType,
type::kChainer>
```

`MakeTensorExp` is a base class inheritated by most functions in `./extension/`. Details will be provided when those functions are discussed.

```
template<typename SubType, typename SrcExp, int dim, typename DType>
struct MakeTensorExp : public Exp<MakeTensorExp<SubType, SrcExp, dim, DType>, DType, type::kChainer>
  Shape<dim> shape_;
  inline const SubType& real_self(void) const{
    return *static_cast<const SubType*>(this);
  }
};
```

3.2 Plan

The class `Plan` is the class to provide `Eval()` function, which designs the way to do actual evaluation.

3.2.1 Plan Declaration

The template `Plan` should include the `ExpType`, since its major purpose is to evaluate an expression, and `DType` to guide the evaluation process.

```
template<typename ExpType, typename DType>
class Plan {
 public:
  MSHADOW_XINLINE DType Eval(index_t y, index_t x) const;
};
```

3.2.2 Tensor Plan

The tensor plan includes two member variables `*dptr_` and `stride_` to make sure the program can fetch expected value by `dptr_[y * stride_ + x]`.

Since we will never expect to change the evaluation result from `Eval()` function, it is return type always include the key word `const`.

One exception is in the tensor plan, since it has chances to be the left hand side of the assignment. Thus, it owns a special function `REval()`, whose return type should not be `const`.

Moreover, bacause tensor value lies in a pre-allocated memory space, the return type of `REval()` function should be a reference type. To keep consistency, the `Eval()` function in tensor plan also be made as a reference type but const.

```
template <typename Device, int dim, typename DType>
class Plan<Tensor<Device, dim, DType>, DType> {
 public:
  explicit Plan(const Tensor<Device, dim, DType> &t) : dptr_(t.dptr_), stride_(t.stride_) {}

  MSHADOW_XINLINE DType &REval(index_t y, index_t x) {
    return dptr_[y * stride_ + x];
  }

  MSHADOW_XINLINE const DType &Eval(index_t y, index_t x) const {
    return dptr_[y * stride_ + x];
  }

 private:
  DType  *dptr_;
  index_t stride_;
};
```

Because of the special case for 1d tensor (no stride_), the tensor plan for 1d case is specifically defined.

```
template <typename Device, typename DType>
class Plan<Tensor<Device, 1, DType>, DType> {
 public:
  explicit Plan(const Tensor<Device, 1, DType> &t) : dptr_(t.dptr_) {}
  MSHADOW_XINLINE DType &REval(index_t y, index_t x) {
    return dptr_[x];
  }
  MSHADOW_XINLINE const DType &Eval(index_t y, index_t x) const {
    return dptr_[x];
  }

 private:
```

```
  DType  *dptr_;
};
```

3.2.3 Scalar Plan

The usage of scalar plan includes

- It has a explicit constructor that takes in a built-in scalar type value and assign the value with type `DType` to its private member variable
- It has a public member function `Eval()` that return its private member variable `scalar_`, no matter what the input is

```
template<typename DType>
class Plan<ScalarExp<DType>, DType> {
 public:
  explicit Plan(DType scalar) : scalar_(scalar) {}

  MSHADOW_XINLINE DType Eval(index_t y, index_t x) const {
    return scalar_;
  }

 private:
  DType scalar_;
};
```

3.2.4 Typecast Plan

The usage of typecast expression includes

- it is used for the type cast expression `TypecastExp`, e.g. tcast(x)
- it has an explicit constructor that takes in a `Plan` type as its private member variable. The reason of doing so is it uses the original plan for evaluation, and only convert its returned type at last.
- it has a function called `Eval()`, where the type cast happens by calling `DstDType(src_.Eval(y, x))`

```
template<typename DstDType, typename SrcDType, typename EType, int etype>
class Plan<TypecastExp<DstDType, SrcDType, EType, etype>, DstDType> {
 public:
  explicit Plan(const Plan<EType, SrcDType> &src) : src_(src) {}

  MSHADOW_XINLINE DstDType Eval(index_t y, index_t x) const {
    return DstDType(src_.Eval(y, x));
  }

 private:
  Plan<EType, SrcDType> src_;
};
```

3.2.5 Ternary Plan

The usage of ternary plan includes

- it has an explicit constructor that takes in 3 expressions as its private member variables

- it has a function called `Eval()`, where the computation happens by calling `OP::Map(item1_.Eval(y, x), item2_.Eval(y, x), item3_.Eval(y, x));`

```
template<typename OP, typename TA, typename TB, typename TC, int etype, typename DType>
class Plan<TernaryMapExp<OP, TA, TB, TC, DType, etype>, DType> {
 public:
  explicit Plan(const Plan<TA, DType> &item1, const Plan<TB, DType> &item2, const Plan<TC, DType> &it
      : item1_(item1), item2_(item2), item3_(item3) {}

  MSHADOW_XINLINE DType Eval(index_t y, index_t x) const {
    return OP::Map(item1_.Eval(y, x), item2_.Eval(y, x), item3_.Eval(y, x));
  }

 private:
  Plan<TA, DType> item1_;
  Plan<TB, DType> item2_;
  Plan<TC, DType> item3_;
};
```

3.2.6 Binary Plan

The usage of binary plan includes

- it has an explicit constructor that takes in 2 expressions as its private member variables
- it has a function called `Eval()`, where the computation happens by calling `OP::Map(lhs_.Eval(y, x), rhs_.Eval(y, x));`

```
template<typename OP, typename TA, typename TB, int etype, typename DType>
class Plan<BinaryMapExp<OP, TA, TB, DType, etype>, DType> {
 public:
  explicit Plan(const Plan<TA, DType> &lhs, const Plan<TB, DType> &rhs)
      : lhs_(lhs), rhs_(rhs) {}

  MSHADOW_XINLINE DType Eval(index_t y, index_t x) const {
    return OP::Map(lhs_.Eval(y, x), rhs_.Eval(y, x));
  }

 private:
  Plan<TA, DType> lhs_;
  Plan<TB, DType> rhs_;
};
```

3.2.7 Unary Plan

The usage of unary plan includes

- it has an explicit constructor that takes in 1 expression as its private member variables
- it has a function called `Eval()`, where the computation happens by calling `OP::Map(src_.Eval(y, x));`

```
template<typename OP, typename TA, int etype, typename DType>
class Plan<UnaryMapExp<OP, TA, DType, etype>, DType> {
 public:
  explicit Plan(const Plan<TA, DType> &src) : src_(src) {}
```

```
  MSHADOW_XINLINE DType Eval(index_t y, index_t x) const {
    return OP::Map(src_.Eval(y, x));
  }

 private:
  Plan<TA, DType> src_;
};
```

3.2.8 MakeTensorExp Plan

Since the expression type `MakeTensorExp` is only a base class of several extension functions (will be discussed in `./extensions/`), the MakeTensorExp plan also only consider its input `Plan` as its member variables, and use the `Eval()` function of input plan as `Eval()` function of itself.

```
Since any expression should be composed by some types of calling, e.g. `MakeExp()`.
However, the `MakeTensorExp` lacks of it, since it is only a general base class.
Thus, I think this Plan will also never be called. Remain unproven!!!
```

```
template<typename SubType, typename SrcExp, int dim, typename DType>
struct Plan<MakeTensorExp<SubType, SrcExp, dim, DType>, DType> {
 public:
  Plan(const Plan<SubType, DType> &src) : src_(src) {}

  MSHADOW_XINLINE DType Eval(index_t y, index_t x) const {
    return src_.Eval(y, x);
  }

 private:
  Plan<SubType, DType> src_;
};
```

3.2.9 Transpose Plan

The usage of transpose plan includes

- it has an explicit constructor that takes in a Plan `src` to initiate a same one as its private member variable. Since it is only a transpose expression, it still need the original `src` to do the evaluation, and only show the transpose effect in the special design in the `Eval()` function (the interchage of `y` and `x`).
- it has a member function called `Eval()`, which do the transpose in the computation by calling `src_.Eval(x, y)`, instead of `src_.Eval(y, x)`;

```
template<typename EType, typename DType>
class Plan<TransposeExp<EType, DType>, DType> {
 public:
  explicit Plan(const Plan<EType, DType> &src) : src_(src) {}

  MSHADOW_XINLINE DType Eval(index_t y, index_t x) const {
    return src_.Eval(x, y);
  }

 private:
  Plan<EType, DType> src_;
};
```

3.3 MakePlan

The major purpose of `MakePlan` is to transform an Expression to its corresponding `Plan`.

3.3.1 Tensor MakePlan

The input variable of Tensor MakePlan is `RValueExp<T, DType> &e`, which is the grandfather class of `Tensor`. The reason of such design choice is compatibility, since a father class can safely refer to its children class.

```
template<typename T, typename DType>
inline Plan<T, DType> MakePlan(const RValueExp<T, DType> &e) {
  return Plan<T, DType>(e.self());
}
```

3.3.2 Scalar MakePlan

The Scalar MakePlan takes in a `ScalarExp`, and uses its member variable `scalar_` to initiate `Plan<ScalarExp<DType>, DType>`.

```
template<typename DType>
inline Plan<ScalarExp<DType>, DType> MakePlan(const ScalarExp<DType> &e) {
  return Plan<ScalarExp<DType>, DType>(e.scalar_);
}
```

3.3.3 Typecast MakePlan

Since the Typecast Plan only use the `Eval()` function of original plan to do `Eval()` of itself, with a enforced typecast at the end, the only thing Typecast MakePlan needs to do is input a plan of original expression

```
template<typename DstDType, typename SrcDType, typename EType, int etype>
inline Plan<TypecastExp<DstDType, SrcDType, EType, etype>, DstDType>
MakePlan(const TypecastExp<DstDType, SrcDType, EType, etype> &e) {
  return Plan<TypecastExp<DstDType, SrcDType, EType, etype>, DstDType>(MakePlan(e.exp));
}
```

3.3.4 Ternary MakePlan

Since the Ternary Plan require three Plans to do the evaluation, the Ternary MakePlan just provides the plan of its components.

```
// declaration
template<typename OP, typename TA, typename TB, typename TC, typename DType, int etype>
inline Plan<TernaryMapExp<OP, TA, TB, TC, DType, etype>, DType>
MakePlan(const TernaryMapExp<OP, TA, TB, TC, DType, etype> &e);

// definition
template<typename OP, typename TA, typename TB, typename TC, typename DType, int etype>
inline Plan<TernaryMapExp<OP, TA, TB, TC, DType, etype>, DType>
MakePlan(const TernaryMapExp<OP, TA, TB, TC, DType, etype> &e) {
  return Plan<TernaryMapExp<OP, TA, TB, TC, DType, etype>,
              DType>(MakePlan(e.item1_), MakePlan(e.item2_), MakePlan(e.item3_));
}
```

3.3.5 Binary MakePlan

Since the Binary Plan require two Plans to do the evaluation, the Biary MakePlan just provides the plan of its components.

```
// declaration
template<typename OP, typename TA, typename TB, typename DType, int etype>
inline Plan<BinaryMapExp<OP, TA, TB, DType, etype>, DType>
MakePlan(const BinaryMapExp<OP, TA, TB, DType, etype> &e);

// definition
template<typename OP, typename TA, typename TB, typename DType, int etype>
inline Plan<BinaryMapExp<OP, TA, TB, DType, etype>, DType>
MakePlan(const BinaryMapExp<OP, TA, TB, DType, etype> &e) {
  return Plan<BinaryMapExp<OP, TA, TB, DType, etype>,
              DType>(MakePlan(e.lhs_), MakePlan(e.rhs_));
}
```

3.3.6 Unary MakePlan

Since the Unary Plan require one Plan to do the evaluation, the Unary MakePlan just provides the plan of its component.

```
template<typename OP, typename TA, typename DType, int etype>
inline Plan<UnaryMapExp<OP, TA, DType, etype>, DType>
MakePlan(const UnaryMapExp<OP, TA, DType, etype> &e) {
  return Plan<UnaryMapExp<OP, TA, DType, etype>, DType>(MakePlan(e.src_));
}
```

3.3.7 MakeTensorExp MakePlan

In `2.8`, we have claimed that the expression type MakeTensorExp is only a base class of several extension functions (will be discussed in ./extensions/), and use the `Eval()` function of input plan as `Eval()` function of itself.

Therefore, in MakeTensorExp MakePlan, the input `MakeTensorExp<T, SrcExp, dim, DType> &e` is kind of like `RValueExp<T, DType> &e`, since both of them are the father class of other classes. Same as Tensor MakePlan, it returns `Plan<T, DType>(e.real_self());` as a Plan of its `SubType`.

```
template<typename T, typename SrcExp, int dim, typename DType>
inline Plan<T, DType>
MakePlan(const MakeTensorExp<T, SrcExp, dim, DType> &e) {
  return Plan<T, DType>(e.real_self());
}
```

3.3.8 Transpose MakePlan

Since the Transpose Plan only use the `Eval()` function of original plan to do `Eval()` of itself, with a interchage of `y` and `x`, the only thing Transpose MakePlan needs to do is input a plan of original expression

```
template<typename T, typename DType>
inline Plan<TransposeExp<T, DType>, DType>
MakePlan(const TransposeExp<T, DType> &e) {
  return Plan<TransposeExp<T, DType>, DType>(MakePlan(e.exp));
}
```

3.4 ExpInfo

`ExpInfo` is a template struct providing expression infomation to help the `TypeCheck`.

- `kDim`: is the dimension of expression related variables
- `kDevice`: is the device where the expression related variables stored

The default configuration is:

```
template<typename E>
struct ExpInfo {
  static const int kDim = -1;
  static const int kDevMask = 0;
};
```

For Tensor ExpInfo, information can be directly extracted from the `Tensor`:

```
template<typename Device, int dim, typename DType>
struct ExpInfo<Tensor<Device, dim, DType> > {
  static const int kDim = dim;
  static const int kDevMask = Device::kDevMask;
  // kDevMask = 1 for cpu (01), = 2 for gpu (10)
};
```

For Scalar ExpInfo, `kDim` is always `0`, and `kDevMask` is always `0xffff` to make sure no influence to other variables caused by itself.

```
template<typename DType>
struct ExpInfo< ScalarExp<DType> > {
  static const int kDim = 0;
  static const int kDevMask = 0xffff;
};
```

For Typecast ExpInfo, the information can be directly copied from the information of original expresion

```
template<typename DstDType, typename SrcDType, typename EType, int etype>
struct ExpInfo<TypecastExp<DstDType, SrcDType, EType, etype> > {
  static const int kDim = ExpInfo<EType>::kDim;
  static const int kDevMask = ExpInfo<EType>::kDevMask;
};
```

For Ternary ExpInfo, its information is from three different expressions:

```
template<typename OP, typename TA, typename TB, typename TC, typename DType, int etype>
struct ExpInfo<TernaryMapExp<OP, TA, TB, TC, DType, etype> > {
  static const int kDimItem1 = ExpInfo<TA>::kDim;
  static const int kDimItem2 = ExpInfo<TB>::kDim;
  static const int kDimItem3 = ExpInfo<TC>::kDim;
  static const int kDim = kDimItem1;
  static const int kDevMask = ExpInfo<TA>::kDevMask & ExpInfo<TB>::kDevMask & ExpInfo<TC>::kDevMask;
};
```

For Binary ExpInfo, its information is from two different expressions:

```
template<typename OP, typename TA, typename TB, typename DType, int etype>
struct ExpInfo<BinaryMapExp<OP, TA, TB, DType, etype> > {
  static const int kDimLhs = ExpInfo<TA>::kDim;
  static const int kDimRhs = ExpInfo<TB>::kDim;
  static const int kDim = (kDimLhs >= 0 && kDimRhs >= 0) ?\
      (kDimLhs == 0 ?\
```

```
       kDimRhs :\
       ((kDimRhs == 0 || kDimLhs == kDimRhs) ? kDimLhs : -1)) : -1;
  static const int kDevMask = ExpInfo<TA>::kDevMask & ExpInfo<TB>::kDevMask;
};
```

For Unary ExpInfo and Transpose ExpInfo, its information is also simply the copy of its variable:

```
template<typename OP, typename TA, typename DType, int etype>
struct ExpInfo<UnaryMapExp<OP, TA, DType, etype> > {
  static const int kDim = ExpInfo<TA>::kDim;
  static const int kDevMask = ExpInfo<TA>::kDevMask;
};

template<typename E, typename DType>
struct ExpInfo<TransposeExp<E, DType> > {
  static const int kDim = ExpInfo<E>::kDim;
  static const int kDevMask = ExpInfo<E>::kDevMask;
};
```

At last, the MakeTensorExp Info is determined by `SrcExp` as input expression variable of its `SubType`

```
the reason of its design is unclear for now, remain to be checking !!!
```

```
template<typename T, typename SrcExp, int dim, typename DType>
struct ExpInfo<MakeTensorExp<T, SrcExp, dim, DType> > {
  static const int kDimSrc = ExpInfo<SrcExp>::kDim;
  static const int kDim = kDimSrc >= 0 ? dim : -1;
  static const int kDevMask = ExpInfo<SrcExp>::kDevMask;
};
```

3.5 TypeCheck

`TypeCheck` is a template to do type checking. The checking happens in compile time.

- `kExpDim`: is the dimension of expression.
- `kDevPass`: checks whether the expression device type matches the provided type. This is to make sure that the expression related variables are all in one specified device.
- `kMapPass`: checks whether the expression can be mapped to expression of dim
- `kRedPass`: checks whether the expression can be reduced to expression of dim

```
The usage of kRedPass is unclear yet!!!
```

```
template<typename Device, int dim, typename DType, typename E>
struct TypeCheck {
  static const int kExpDim = ExpInfo<E>::kDim;
  static const bool kDevPass = (ExpInfo<E>::kDevMask & Device::kDevMask) != 0;
  static const bool kMapPass = (kExpDim == 0 || kExpDim == dim) && kDevPass;
  static const bool kRedPass = (kExpDim > dim) && kDevPass;
};
```

The usage of TypeCheck is like `expr::TypeCheckPass<expr::TypeCheck<cpu, dim, DType, E>::kMapPass> ::Error_All_Tensor_in_Exp_Must_Have_Same_Type();`. If the `TypeCheck<cpu, dim, DType, E>::kMapPass>` returns `false`, then the corresponding function `Error_All_Tensor_in_Exp_Must_Have_Same_Type()` will not be found, which leads to the fail of compile.

```
template<bool kPass>
struct TypeCheckPass;
template<>
struct TypeCheckPass<false> {};
template<>
struct TypeCheckPass<true> {
  inline static void Error_All_Tensor_in_Exp_Must_Have_Same_Type(void) {}
  inline static void Error_TypeCheck_Not_Pass_For_Reduce_Exp(void) {}
  inline static void Error_Expression_Does_Not_Meet_Dimension_Req(void) {}
};
```

3.6 StreamInfo

The `StreamInfo` returns the stream of a `Tensor` in its stored `Device`.

```
The usage of it is unclear yet !!!
```

```
template<typename Device, typename E>
struct StreamInfo {
  inline static Stream<Device> *Get(const E &t);
};

template<int dim, typename Device, typename DType>
struct StreamInfo<Device, Tensor<Device, dim, DType> > {
  inline static Stream<Device> *Get(const Tensor<Device, dim, DType> &t) {
    return t.stream_;
  }
};
```

3.7 ShapeCheck

`ShapeCheck` is a runtime shape checking template to get the shape of an expression, reporting error if shape mismatch

3.7.1 Declaration

```
template<int dim, typename E>
struct ShapeCheck {
  inline static Shape<dim> Check(const E &t);
};
```

3.7.2 Tensor ShapeCheck

The Tensor ShapeCheck simply returns its shape.

```
template<int dim, typename Device, typename DType>
struct ShapeCheck<dim, Tensor<Device, dim, DType> > {
  inline static Shape<dim> Check(const Tensor<Device, dim, DType> &t) {
    return t.shape_;
  }
};
```

3.7.3 Scalar ShapeCheck

The Scalar ShapeCheck returns a same `Shape` of provided `dim`, but all of the dimension equal to `0`.

```
template<int dim, typename DType>
struct ShapeCheck<dim, ScalarExp<DType> > {
  inline static Shape<dim> Check(const ScalarExp<DType> &exp) {
    Shape<dim> shape;
    for (int i = 0; i < dim; ++i) {
      shape[i] = 0;
    }
    return shape;
  }
};
```

3.7.4 Typecast ShapeCheck

The Typecast ShapeCheck simply conducts in a way that calls the original `Check()` of the expression wishing to cast it type.

```
template<int dim, typename DstDType, typename SrcDType, typename EType, int etype>
struct ShapeCheck<dim, TypecastExp<DstDType, SrcDType, EType, etype> > {
  inline static Shape<dim>
  Check(const TypecastExp<DstDType, SrcDType, EType, etype> &exp) {
    return ShapeCheck<dim, EType>::Check(exp.exp);
  }
};
```

3.7.5 Ternary ShapeCheck

The Ternary ShapeCheck first fetch the check results of its three variable expressions. Then, it couducts a explicit checking to make sure all of the expressions have same shape. At last, it returns one of the shape of expressions.

```
template<int dim, typename OP, typename TA, typename TB, typename TC, typename DType, int etype>
struct ShapeCheck<dim, TernaryMapExp<OP, TA, TB, TC, DType, etype> > {
  inline static Shape<dim>
  Check(const TernaryMapExp<OP, TA, TB, TC, DType, etype> &t) {
    Shape<dim> shape1 = ShapeCheck<dim, TA>::Check(t.item1_);
    Shape<dim> shape2 = ShapeCheck<dim, TB>::Check(t.item2_);
    Shape<dim> shape3 = ShapeCheck<dim, TC>::Check(t.item3_);
    bool same = (shape1 == shape2) && (shape2 == shape3);
    CHECK(same) << "TernaryMapExp: Shapes of operands are not the same, " <<
      "Shape1=" << shape1 << ", Shape2=" << shape2 << ", Shape3=" << shape3;
    return shape1;
  }
};
```

3.7.6 Binary ShapeCheck

The Binary ShapeCheck first fetch the check results of its two variable expressions. Then, it couducts a explicit checking to make sure all of the expressions have same shape. At last, it returns one of the shape of expressions.

```
template<int dim, typename OP, typename TA, typename TB, typename DType, int etype>
struct ShapeCheck<dim, BinaryMapExp<OP, TA, TB, DType, etype> > {
```

```
  inline static Shape<dim>
  Check(const BinaryMapExp<OP, TA, TB, DType, etype> &t) {
    Shape<dim> shape1 = ShapeCheck<dim, TA>::Check(t.lhs_);
    Shape<dim> shape2 = ShapeCheck<dim, TB>::Check(t.rhs_);
    if (shape1[0] == 0) return shape2;
    if (shape2[0] == 0) return shape1;
    CHECK_EQ(shape1, shape2) << "BinaryMapExp: Shapes of operands are not the same, " <<
      "Shape1=" << shape1 << ", Shape2=" << shape2;
    return shape1;
  }
};
```

3.7.7 Unary ShapeCheck

For Unary ShapeCheck, it just fetch the check result of its variable expressions and return it.

```
template<int dim, typename OP, typename TA, typename DType, int etype>
struct ShapeCheck<dim, UnaryMapExp<OP, TA, DType, etype> > {
  inline static Shape<dim> Check(const UnaryMapExp<OP, TA, DType, etype> &t) {
    Shape<dim> s = ShapeCheck<dim, TA>::Check(t.src_);
    return s;
  }
};
```

3.7.8 MakeTensorExp ShapeCheck

MakeTensorExp ShapeCheck simply use the member variable `shape_` of `MakeTensorExp` as its return.

```
It is design strategy needs to be further examined.
```

```
template<int dim, typename SrcExp, typename T, typename DType>
struct ShapeCheck<dim, MakeTensorExp<T, SrcExp, dim, DType> > {
  inline static Shape<dim>
  Check(const MakeTensorExp<T, SrcExp, dim, DType> &t) {
    return t.shape_;
  }
};
```

3.7.9 Transpose ShapeCheck

Transpose ShapeCheck fetchs the checked shape of expression to be transposed, and simply return the swaped lowest two dimensions.

```
The strategy of Transpose needs to be further studied when it comes to application.
```

```
template<int dim, typename E, typename DType>
struct ShapeCheck<dim, TransposeExp<E, DType> > {
  inline static Shape<dim> Check(const TransposeExp<E, DType> &e) {
    // swap the lowest two dimensions
    Shape<dim> s = ShapeCheck<dim, E>::Check(e.exp);
    std::swap(s[0], s[1]);
    return s;
  }
};
```

3.8 ExpEngine

`ExpEngine` is a struct with several overloaded `Eval()` functions, always called by the assignment related functions in `RValueExp` to dispatch simple operations.

```
template<typename SV, typename RV, typename DType>
struct ExpEngine {
  template<typename E>
  inline static void Eval(RV *dst,
                          const Exp<E, DType, type::kMapper> &exp) {
    MapExp<SV>(dst, exp);
  }

  template<typename E>
  inline static void Eval(RV *dst,
                          const Exp<E, DType, type::kChainer> &exp) {
    MapExp<SV>(dst, exp);
  }

  template<typename E>
  inline static void Eval(RV *dst,
                          const Exp<E, DType, type::kRValue> &exp) {
    MapExp<SV>(dst, exp);
  }

  template<typename E>
  inline static void Eval(RV *dst,
                          const Exp<E, DType, type::kComplex> &exp) {
    ExpComplexEngine<SV, RV, E, DType>::Eval(dst->ptrself(), exp.self());
  }
};
```

The `ExpComplexEngine` to evaluate expression with `type::kComplex` majorly deals with `dot()` expression and looks like:

```
template<typename SV, typename RV, typename E, typename DType>
struct ExpComplexEngine {
  inline static void Eval(RV *dst, const E &exp);
};

template<typename SV, typename Device, int dim, int ldim,
         int rdim, bool ltrans, bool rtrans, typename DType>
struct ExpComplexEngine<SV,
                        Tensor<Device, dim, DType>,
                        DotExp<Tensor<Device, ldim, DType>,
                               Tensor<Device, rdim, DType>,
                               ltrans, rtrans, DType>,
                        DType> {
  inline static void Eval(Tensor<Device, dim, DType> *dst,
                          const DotExp<Tensor<Device, ldim, DType>,
                                       Tensor<Device, rdim, DType>,
                                       ltrans, rtrans, DType> &exp) {
    DotEngine<SV, Device, dim, ldim, rdim,
              ltrans, rtrans, DType>::Eval(dst, exp.lhs_, exp.rhs_, exp.scale_);
  }
};
```

The DotEngine will be further discussed in `./dot_engine-inl.h`.

CHAPTER 4

Packet-inl.h

`Packet` is designed to further speed up computation.

- Alignment
- Packet
- Packet Operator
- Saver
- Packet Plan
- Packet Check
- Packet Align Check
- MapPacketPlan

A general `Packet` has two modes to choose:

```
enum PacketArch {
  kPlain,
  kSSE2,
};
```

- `kPlain` is just a plain version using standard C language in case SSE is not available
- `kSSE2` is a set of instructions allowing to perform computations on packets of `128` bits at once. Since a float is `32` bits, this means that `SSE2` instructions can handle `4` floats at once, which also means that, if correctly used, they can make our computation go up to `4x` faster.

```
However, in real applications, e.g. we have chosen `size=50`, so our vectors consist of `50`
float, and `50` is not a multiple of `4`. This means that we cannot hope to do all
of that computation using SSE2 instructions. The second best thing, to which we should aim,
is to handle the `48` first coefficients with SSE2 instructions, since `48` is the biggest
multiple of `4` below `50`, and then handle separately, without SSE2, the `49th` and `50th`
coefficients.
```

As a result, we define a struct containing the required aligned bytes

```
template<PacketArch Arch>
struct AlignBytes {
  // typedef unsigned index_t;
  static const index_t value = 4;
};
```

4.1 Packet

`Packet` is a struct that is designed to handle the evaluation process of expression, in a scalar level, by using SSE2 instructions set for better efficiency.

It has three different basic type `<DType, kPlain>` for standard C computation, `<float, kSSE2>` for float SSE2 optimization, and `<double, kSSE2>` for double SSE2 optimization. All of them will be described in parallel.

4.1.1 Member Variables

- `kSize`: number of float can be handled as a vector
- `data_`: the internal data that is loaded from a pointer pointing to a built-in scalar type

Definition in `<DType, kPlain>`:

```
// since kPlain is only defined as default C computation in case of the absense of SSE2
// it naturally can only handle 1 float a time
static const index_t kSize = 1;
DType data_;
```

Definition in `<float, kSSE2>` and `<double, kSSE2>`:

```
// since float size is 32, it can handle 4 floats a time
static const index_t kSize = 4;
// __m128 is a special type in SSE2
__m128 data_;
```

```
// since double size is 64, it can only handle 2 doubles a time
static const index_t kSize = 2;
__m128d data_;
```

The vectorization makes `4` floats or `2` doubles looks like a `scalar`.

4.1.2 Constructor

Since `Packet` only has `2` member variables, and one of them is static const type, we only need initiate the `data_` in constructor.

Definition in `<DType, kPlain>`:

```
Packet(void) {}
explicit Packet(DType data) : data_(data) {}
```

Definition for `<float, kSSE2>` and `<double, kSSE2>` is same:

```
Packet(void) {}
explicit Packet(__m128d data) : data_(data) {}
```

4.1.3 Member Functions

The member functions in `<DType, kPlain>` just use the simple C / C++ code to realize same function in `<float/double, kSSE2>`.

```
// create a fill with the target value s
MSHADOW_CINLINE static Packet<DType, kPlain> Fill(DType s) {
  return Packet<DType, kPlain>(s);
}

// load from address src, using *src for the value
MSHADOW_CINLINE static Packet<DType, kPlain> Load(const DType* src) {
  return Packet<DType, kPlain>(*src);
}

// load from address src, using *src for the value
MSHADOW_CINLINE static Packet<DType, kPlain> LoadUnAligned(const DType* src) {
  return Packet<DType, kPlain>(*src);
}

// store data into dst
MSHADOW_CINLINE void Store(DType* dst) const {
  *dst = data_;
}
```

Compared to plain implementation, the realization of `<float/double, kSSE2>` requires the usage of `intrinsics`, included in file `<emmintrin.h>`.

- `_mm_set1_ps(w)`: set all the four floats to the value of `w`, e.g. `r0 := r1 := r2 := r3 := w`
- `_mm_load_ps(w)`: load the first four floats pointed by `w`, e.g. `r0 := p[0]`, `r1 := p[1]`, `r2 := p[2]`, `r3 := p[3]`. The address must be 16-byte aligned.
- `_mm_loadu_ps(w)`: same as `_mm_load_ps`, but the address does not need to be 16-byte aligned.
- `_mm_store_ps(*p,a)`: transfer the `4` floats values in `a` to the address pointed by `p`. e.g. `p[0] := a0`, `p[1] := a1`, `p[2] := a2`, `p[3] := a3`

```
MSHADOW_CINLINE static Packet<float, kSSE2> Fill(float s) {
  return Packet<float, kSSE2>(_mm_set1_ps(s));
}

MSHADOW_CINLINE static Packet<float, kSSE2> Load(const float* src) {
  return Packet<float, kSSE2>(_mm_load_ps(src));
}

MSHADOW_CINLINE static Packet<float, kSSE2> LoadUnAligned(const float* src) {
  return Packet<float, kSSE2>(_mm_loadu_ps(src));
}

MSHADOW_CINLINE void Store(float* dst) const {
  _mm_store_ps(dst, data_);
}
```

For `<double, kSSE2>`, the process and instructions are all similar, instead of the final `ps` is changed to `pd` for dealing with `double`.

```
MSHADOW_CINLINE static Packet<double, kSSE2> Fill(double s) {
  return Packet<double, kSSE2>(_mm_set1_pd(s));
}

MSHADOW_CINLINE static Packet<double, kSSE2> Load(const double* src) {
  return Packet<double, kSSE2>(_mm_load_pd(src));
}
```

```
MSHADOW_CINLINE static Packet<double, kSSE2> LoadUnAligned(const double* src) {
  return Packet<double, kSSE2>(_mm_loadu_pd(src));
}

MSHADOW_CINLINE void Store(double* dst) const {
  _mm_store_pd(dst, data_);
}
```

```
We intentionally leave the discussion of Sum() function, since its application is scarce in
our context and has heave extra SSE2 instructions.
```

4.1.4 Overloaded Operators

= operator

The = operator simply change the value of `data_` to the value in right hand side, and return itself by `*this`.

```
// plain
MSHADOW_CINLINE Packet<DType, kPlain>& operator=(DType s) {
  data_ = s;
  return *this;
}

// <float, kSSE2>
MSHADOW_CINLINE Packet<float, kSSE2>& operator=(float s) {
  data_ = _mm_set1_ps(s);
  return *this;
}

// <double, kSSE2>
MSHADOW_CINLINE Packet<double, kSSE2>& operator=(double s) {
  data_ = _mm_set1_pd(s);
  return *this;
}
```

+ / – / * / / operators

For Plain version:

```
template<typename DType>
MSHADOW_CINLINE Packet<DType, kPlain> operator+(const Packet<DType, kPlain>& lhs,
                                                const Packet<DType, kPlain>& rhs) {
  return Packet<DType, kPlain>(lhs.data_ + rhs.data_);
}

template<typename DType>
MSHADOW_CINLINE Packet<DType, kPlain> operator-(const Packet<DType, kPlain>& lhs,
                                                const Packet<DType, kPlain>& rhs) {
  return Packet<DType, kPlain>(lhs.data_ - rhs.data_);
}
template<typename DType>
MSHADOW_CINLINE Packet<DType, kPlain> operator*(const Packet<DType, kPlain>& lhs,
                                                const Packet<DType, kPlain>& rhs) {
  return Packet<DType, kPlain>(lhs.data_ * rhs.data_);
}
```

```
template<typename DType>
MSHADOW_CINLINE Packet<DType, kPlain> operator/(const Packet<DType, kPlain>& lhs,
                                                const Packet<DType, kPlain>& rhs) {
  return Packet<DType, kPlain>(lhs.data_ / rhs.data_);
}
```

For SSE2 version:

```
MSHADOW_CINLINE Packet<float, kSSE2> operator+(const Packet<float, kSSE2>& lhs,
                                               const Packet<float, kSSE2>& rhs) {
  return Packet<float, kSSE2>(_mm_add_ps(lhs.data_, rhs.data_));
}

MSHADOW_CINLINE Packet<double, kSSE2> operator+(const Packet<double, kSSE2>& lhs,
                                                const Packet<double, kSSE2>& rhs) {
  return Packet<double, kSSE2>(_mm_add_pd(lhs.data_, rhs.data_));
}

MSHADOW_CINLINE Packet<float, kSSE2> operator-(const Packet<float, kSSE2>& lhs,
                                               const Packet<float, kSSE2>& rhs) {
  return Packet<float, kSSE2>(_mm_sub_ps(lhs.data_, rhs.data_));
}

MSHADOW_CINLINE Packet<double, kSSE2> operator-(const Packet<double, kSSE2>& lhs,
                                                const Packet<double, kSSE2>& rhs) {
  return Packet<double, kSSE2>(_mm_sub_pd(lhs.data_, rhs.data_));
}

MSHADOW_CINLINE Packet<float, kSSE2> operator*(const Packet<float, kSSE2>& lhs,
                                               const Packet<float, kSSE2>& rhs) {
  return Packet<float, kSSE2>(_mm_mul_ps(lhs.data_, rhs.data_));
}

MSHADOW_CINLINE Packet<double, kSSE2> operator*(const Packet<double, kSSE2>& lhs,
                                                const Packet<double, kSSE2>& rhs) {
  return Packet<double, kSSE2>(_mm_mul_pd(lhs.data_, rhs.data_));
}


MSHADOW_CINLINE Packet<float, kSSE2> operator/(const Packet<float, kSSE2>& lhs,
                                               const Packet<float, kSSE2>& rhs) {
  return Packet<float, kSSE2>(_mm_div_ps(lhs.data_, rhs.data_));
}

MSHADOW_CINLINE Packet<double, kSSE2> operator/(const Packet<double, kSSE2>& lhs,
                                                const Packet<double, kSSE2>& rhs) {
  return Packet<double, kSSE2>(_mm_div_pd(lhs.data_, rhs.data_));
}
```

4.2 Alignment

To make fully use of SSE2 instructions set, Alignment is important in the context.

4.2.1 Aligned and Aligned Free

`AlignedMallocPitch()` is an analog to `cudaMallocPitch()`, which allocates a aligned space with `num_line * lspace` cells.

- `out_pitch`: output parameter, the actuall space allocated for each line
- `lspace`: number of cells required for each line
- `num_line`: number of lines to be allocated

First, according to `lspace`, `out_pitch` can be calculated as the smallest number that is larger than `lspace` and can be divided by `16`. Then, the aligned space to be allocated can be calculated by `out_pitch * num_line`.

```
inline void* AlignedMallocPitch(size_t *out_pitch,
                                size_t lspace,
                                size_t num_line) {
  const index_t bits = AlignBytes<MSHADOW_DEFAULT_PACKET>::value;   // = 4
  const index_t mask = (1 << bits) - 1;   // = 15

  // e.g. ((25 + 15) >> 4) << 4 = 32;
  size_t pitch = ((lspace + mask) >> bits) << bits;

  *out_pitch = pitch;
#ifdef _MSC_VER
  void *res = _aligned_malloc(pitch * num_line, 1 << bits);
#else
  void *res;
  int ret = posix_memalign(&res, 1 << bits, pitch * num_line);
  CHECK_EQ(ret, 0) << "AlignedMallocPitch failed";
#endif
  if (res == NULL) {
    LOG(FATAL) << "AlignedMallocPitch failed";
  }
  return res;
}
```

As usual, since we allocate a space, we should free it after using by

```
inline void AlignedFree(void *ptr) {
#ifdef _MSC_VER
  _aligned_free(ptr);
#else
  free(ptr);
#endif
}
```

4.2.2 CheckAlign

`CheckAlign` is designed to make sure a value or the address of a pointer can be divided by 16.

```
// check whether the value can be divided by 16
template<PacketArch Arch>
inline bool CheckAlign(size_t pitch) {
  const index_t bits = AlignBytes<Arch>::value;
  // the aligned pitch is expected to hold 4 zeros in its lowest 4 bits, or just can be divided by 16

  // thus, when it is & with 15, the result should be zero
  return !(pitch & ((1 << bits) - 1));
```

```
}

// check whether the stored address of a pointer can be divided by 16
template<PacketArch Arch>
inline bool CheckAlign(void *ptr) {
  // reinterpret_cast forces the complier thinking ptr as a size_t type
  return CheckAlign<Arch>(reinterpret_cast<size_t>(ptr));
}
```

4.2.3 UpperAlign and LowerAlign

`UpperAlign` and `LowerAlign` return the required loop times to execute a string with `size`.

According to the example in the beginning, it is a common strategy to use `LowerAlign` and then handle remaining separately. If `UpperAlign` is used, padding is required.

```
template<typename DType, PacketArch Arch>
inline index_t UpperAlign(index_t size) {
  const index_t bits = AlignBytes<MSHADOW_DEFAULT_PACKET>::value;   // = 4
  const index_t mask = (1 << bits) - 1;   // = 15
  const index_t fsize = sizeof(DType);    // = e.g. 4 for float or 8 for double
  return (((size * fsize + mask) >> bits) << bits) / fsize;
}

template<typename DType, PacketArch Arch>
inline index_t LowerAlign(index_t size) {
  const index_t bits = AlignBytes<MSHADOW_DEFAULT_PACKET>::value;   // = 4
  const index_t fsize = sizeof(DType);    // = e.g. 4 for float or 8 for double
  // e.g. (((50 * 4) >> 4) << 4) / 4 = 48 as claimed in the beginning example.
  return (((size * fsize) >> bits) << bits) / fsize;
}
```

4.3 Packet Operator

`PacketOp` is a struct same as the operator defined in the `op` namespace. The different choice of `PacketOp` is trigger by the given typename `OP`.

Its main usage is to deal with the operation between differnt `Packet`. Each `+` / `-` / `*` / `/` is defined in the file `./packet/plain-inl.h` or `./packet/sse-inl.h`.

```
// specialization of operators
template<typename DType, PacketArch Arch>
struct PacketOp<op::plus, DType, Arch> {
  static const bool kEnabled = true;
  MSHADOW_CINLINE static Packet<DType, Arch> Map(const Packet<DType, Arch>& lhs,
                                                 const Packet<DType, Arch>& rhs) {
    return lhs + rhs;
  }
};

template<typename DType, PacketArch Arch>
struct PacketOp<op::minus, DType, Arch> {
  static const bool kEnabled = true;
  MSHADOW_CINLINE static Packet<DType, Arch> Map(const Packet<DType, Arch>& lhs,
                                                 const Packet<DType, Arch>& rhs) {
```

```
    return lhs - rhs;
  }
};
template<typename DType, PacketArch Arch>
struct PacketOp<op::mul, DType, Arch> {
  static const bool kEnabled = true;
  MSHADOW_CINLINE static Packet<DType, Arch> Map(const Packet<DType, Arch>& lhs,
                                                 const Packet<DType, Arch>& rhs) {
    return lhs * rhs;
  }
};

template<typename DType, PacketArch Arch>
struct PacketOp<op::div, DType, Arch> {
  static const bool kEnabled = true;
  MSHADOW_CINLINE static Packet<DType, Arch> Map(const Packet<DType, Arch>& lhs,
                                                 const Packet<DType, Arch>& rhs) {
    return lhs / rhs;
  }
};

template<typename DType, PacketArch Arch>
struct PacketOp<op::identity, DType, Arch> {
  static const bool kEnabled = true;
  MSHADOW_CINLINE static Packet<DType, Arch> Map(const Packet<DType, Arch>& src) {
    return src;
  }
};
```

if the operator is out of range of `op::plus`, `op::minus`, `op::mul`, `op::div` and `op:identity`, the declaration will in its default setting, which leads it fail to pass the checking and back to original C++ implement.

```
template<typename OP, typename DType, PacketArch Arch>
struct PacketOp {
  static const bool kEnabled = false;
};
```

4.4 Saver

`Saver` is the struct that calls its member function `Save()` to do actual evaluation and saving.

It has two formulations. One for evaluation and saving (e.g. `+=` / `-=` / `*=` / `/=`), and one only for saving (e.g. `=`) with operator `sv::saveto`.

For the first `Save()`, we first change the left hand side `*dst` to be a `Packet`, then call `PacketOp::Map()` to finish evaluation. At last, the result is stored to `*dst`.

```
template<typename SV, typename TFloat, PacketArch Arch>
struct Saver{
  MSHADOW_CINLINE static void Save(TFloat *dst, const Packet<TFloat, Arch>& src) {
    Packet<TFloat, Arch> lhs = Packet<TFloat, Arch>::Load(dst);
    Packet<TFloat, Arch> ans = PacketOp<typename SV::OPType, TFloat, Arch>::Map(lhs, src);
    ans.Store(dst);
  }
};
```

For the second `Save()`, we simply store the result to `*dst`.

```
template<typename TFloat, PacketArch Arch>
struct Saver<sv::saveto, TFloat, Arch> {
  MSHADOW_CINLINE static void Save(TFloat *dst, const Packet<TFloat, Arch>& src) {
    src.Store(dst);
  }
};
```

4.5 Packet Plan

Same as the class `Plan` defined in `expr_engine-inl.h`, the class `PacketPlan` is the class to provide `Eval()` function for `Packet`, which is the way to do actual evaluation.

4.5.1 Declaration

According to the declaration, a `PacketPlan` contains two member functions `EvalPacket()` which use `SSE` optimization, and `Eval()` to do common computation as in `Plan`.

```
typedef packet::PacketArch PacketArch;

// same as plan, but use packet
template<typename ExpType, typename DType, PacketArch Arch>
class PacketPlan {
 public:
  MSHADOW_CINLINE packet::Packet<DType, Arch> EvalPacket(index_t y, index_t x) const;
  MSHADOW_CINLINE DType Eval(index_t y, index_t x) const;
};
```

4.5.2 Tensor PacketPlan

The usage of `EvalPacket()` is to provide the data according to `dptr_[y * stride_ + x]`

```
template <typename Device, int dim, typename DType, PacketArch Arch>
class PacketPlan<Tensor<Device, dim, DType>, DType, Arch> {
 public:
  explicit PacketPlan(const Tensor<Device, dim, DType> &t)
      :dptr_(t.dptr_), stride_(t.stride_) {}
  MSHADOW_CINLINE packet::Packet<DType, Arch> EvalPacket(index_t y, index_t x) const {
    return packet::Packet<DType, Arch>::Load(&dptr_[y * stride_ + x]);
  }
  MSHADOW_CINLINE DType Eval(index_t y, index_t x) const {
    return dptr_[y * stride_ + x];
  }

 private:
  const DType  *dptr_;
  index_t stride_;
};
```

4.5.3 Scalar PacketPlan

The usage of `EvalPacket()` is to provide `scalar_` for every `x` and `y`.

```
template<typename DType, PacketArch Arch>
class PacketPlan<ScalarExp<DType>, DType, Arch> {
 public:
  explicit PacketPlan(DType scalar) : scalar_(scalar) {}
  MSHADOW_CINLINE packet::Packet<DType, Arch> EvalPacket(index_t y, index_t x) const {
    return packet::Packet<DType, Arch>::Fill(scalar_);
  }
  MSHADOW_CINLINE DType Eval(index_t y, index_t x) const {
    return scalar_;
  }

 private:
  DType scalar_;
};
```

4.5.4 Binary PacketPlan

The usage of `EvalPacket()` is to perform `EvalPacket()` for `lhs_` and `rhs_` respectively, and use the pre-defined `OP` to do the combinational evaluation.

```
template<typename OP, typename TA, typename TB, int etype, typename DType, PacketArch Arch>
class PacketPlan<BinaryMapExp<OP, TA, TB, DType, etype>, DType, Arch> {
 public:
  PacketPlan(const PacketPlan<TA, DType, Arch> &lhs, const PacketPlan<TB, DType, Arch> &rhs)
      : lhs_(lhs), rhs_(rhs) {}
  MSHADOW_CINLINE packet::Packet<DType, Arch> EvalPacket(index_t y, index_t x) const {
    return packet::PacketOp<OP, DType, Arch>::Map(lhs_.EvalPacket(y, x), rhs_.EvalPacket(y, x));
  }
  MSHADOW_CINLINE DType Eval(index_t y, index_t x) const {
    return OP::Map(lhs_.Eval(y, x), rhs_.Eval(y, x));
  }

 private:
  PacketPlan<TA, DType, Arch> lhs_;
  PacketPlan<TB, DType, Arch> rhs_;
};
```

4.5.5 Unary PacketPlan

The usage of `EvalPacket()` is to perform `EvalPacket()` for `src_` first, and use the pre-defined `OP` to do the compositional evaluation.

```
template<typename OP, typename TA, int etype, typename DType, PacketArch Arch>
class PacketPlan<UnaryMapExp<OP, TA, DType, etype>, DType, Arch> {
 public:
  PacketPlan(const PacketPlan<TA, DType, Arch> &src) : src_(src) {}
  MSHADOW_CINLINE packet::Packet<DType> EvalPacket(index_t y, index_t x) const {
    return packet::PacketOp<OP, DType, Arch>::Map(src_.EvalPacket(y, x));
  }
  MSHADOW_CINLINE DType Eval(index_t y, index_t x) const {
    return OP::Map(src_.Eval(y, x));
  }

 private:
```

```
  PacketPlan<TA, DType, Arch> src_;
};
```

4.5.6 MapPacketPlan

RValueExp

The `MapPacketPlan()` for `RValueExp` is only to return a `PacketPlan` initiated by the subtype of `RValueExp<T, DType>`, which is always a `Tensor`.

```
template<PacketArch Arch, typename T, typename DType>
inline PacketPlan<T, DType, Arch> MakePacketPlan(const RValueExp<T, DType> &e) {
  return PacketPlan<T, DType, Arch>(e.self());
}
```

ScalarExp

The `MapPacketPlan()` for `ScalarExp` is only to return a `PacketPlan` initiated by the member variable `scalar_`:

```
template<PacketArch Arch, typename DType>
inline PacketPlan<ScalarExp<DType>, DType, Arch> MakePacketPlan(const ScalarExp<DType> &e) {
  return PacketPlan<ScalarExp<DType>, DType, Arch>(e.scalar_);
}
```

BinaryMapExp

The `MapPacketPlan()` for `BinaryMapExp` is to recursively called `MapPacketPlan()` for its `lhs` and `rhs` at first stage, and then return the `PacketPlan` initiated by the two results.

```
template<PacketArch Arch, typename OP, typename TA, typename TB, typename DType, int etype>
inline PacketPlan<BinaryMapExp<OP, TA, TB, DType, etype>, DType, Arch>
MakePacketPlan(const BinaryMapExp<OP, TA, TB, DType, etype> &e);

template<PacketArch Arch, typename OP, typename TA, typename TB, typename DType, int etype>
inline PacketPlan<BinaryMapExp<OP, TA, TB, DType, etype>, DType, Arch>
MakePacketPlan(const BinaryMapExp<OP, TA, TB, DType, etype> &e) {
  return PacketPlan<BinaryMapExp<OP, TA, TB, DType, etype>,
                    DType, Arch>(MakePacketPlan<Arch>(e.lhs_), MakePacketPlan<Arch>(e.rhs_));
}
```

UnaryMapExp

The `MapPacketPlan()` for `UnaryMapExp` is to call `MapPacketPlan()` of its input variable `src`, then construct the `PacketPlan` for the return

```
template<PacketArch Arch, typename OP, typename TA, typename DType, int etype>
inline PacketPlan<UnaryMapExp<OP, TA, DType, etype>, DType, Arch>
MakePacketPlan(const UnaryMapExp<OP, TA, DType, etype> &e) {
  return PacketPlan<UnaryMapExp<OP, TA, DType, etype>, DType, Arch>(MakePacketPlan<Arch>(e.src_));
}
```

4.6 Packet Check

The major usage of `PacketCheck` is to make sure every related `DType` is `float` or `double`, and every related `operator` is defined in the context of `Packet`. Otherwise, the program will back to the original C++ implement.

```
template<PacketArch Arch>
struct PacketCheck<float, Arch> {
  static const bool kPass = true;
};

template<PacketArch Arch>
struct PacketCheck<double, Arch> {
  static const bool kPass = true;
};

template<typename DType, PacketArch Arch>
struct PacketCheck<ScalarExp<DType>, Arch> {
  static const bool kPass = PacketCheck<DType, Arch>::kPass;
};

template<int dim, typename DType, PacketArch Arch>
struct PacketCheck<Tensor<cpu, dim, DType>, Arch> {
  static const bool kPass = PacketCheck<DType, Arch>::kPass;
};

// check both operator implementation and data type in Tensor
template<typename OP, typename TA, typename DType, int etype, PacketArch Arch>
struct PacketCheck<UnaryMapExp<OP, TA, DType, etype>, Arch> {
  static const bool kPass = PacketCheck<TA, Arch>::kPass &&
      packet::PacketOp<OP, DType, Arch>::kEnabled;
};

// check both operator implementation and data type in Tensor
template<typename OP, typename TA, typename TB, typename DType, int etype, PacketArch Arch>
struct PacketCheck< BinaryMapExp<OP, TA, TB, DType, etype>, Arch> {
  static const bool kPass = packet::PacketOp<OP, DType, Arch>::kEnabled &&
      PacketCheck<TA, Arch>::kPass && PacketCheck<TB, Arch>::kPass;
};
```

For some expressions, like `TransposeExp()`, and scalars, like `int` of built-in type which is not implemented in `Packet`, the `PacketCheck` is automatically failed. As a result, commmon C++ code will be used to evaluate such expression.

The following code makes sure the default setting is `false`.

```
template<typename E, PacketArch Arch>
struct PacketCheck{
  static const bool kPass = false;
};
```

4.7 Packet Align Check

`PacketAlignCheck` is to check whether the data is aligned and the expressions has been implemented.

4.7.1 Default

First, for some expressions that is not implemented, like `TransposeExp()`, the check should automatically fail and return the evaluation of expressions to their original implementations. So following codes guarantee the automatic switch.

```
template<int dim, typename E, PacketArch Arch>
struct PacketAlignCheck {
  inline static bool Check(const E &exp) {
    return false;
  }
};
```

4.7.2 Tensor

In the `Check()` function, it checks whether the address is 16-byte aligned, and the `stride_` is available for packet (can be divided by `4` for `float`).

```
For now, I highly wonder the importance of `stride_` checking. Since we can always compute
the parts that can be divided by `4` and calculate the remaining as written by itself in
function `MapPacketPlan()`. However, the author just ignores it obviously.

The reason is remained to be checking !!!
```

```
template<int dim, typename DType, PacketArch Arch>
struct PacketAlignCheck<dim, Tensor<cpu, dim, DType>, Arch> {
  inline static bool Check(const Tensor<cpu, dim, DType> &t) {
    return packet::CheckAlign<Arch>(t.dptr_) &&
        packet::CheckAlign<Arch>(t.stride_ * sizeof(DType));
  }
};
```

4.7.3 ScalarExp

The `ScalarExp` is naturally aligned, so we always return `true`.

```
template<int dim, typename DType, PacketArch Arch>
struct PacketAlignCheck<dim, ScalarExp<DType>, Arch> {
  inline static bool Check(const ScalarExp<DType> &exp) {
    return true;
  }
};
```

4.7.4 BinaryMapExp

It first recursively calls `Check()` function of its `lhs` and `rhs`, and combine them by `&&` in the end.

```
template<int dim, typename OP, typename TA, typename TB,
         typename DType, int etype, PacketArch Arch>
struct PacketAlignCheck<dim, BinaryMapExp<OP, TA, TB, DType, etype>, Arch> {
  inline static bool Check(const BinaryMapExp<OP, TA, TB, DType, etype> &t) {
    return PacketAlignCheck<dim, TA, Arch>::Check(t.lhs_) &&
        PacketAlignCheck<dim, TB, Arch>::Check(t.rhs_);
  }
};
```

4.7.5 UnaryMapExp

It just does `Check()` for its variable `src`

```
template<int dim, typename OP, typename TA, typename DType, int etype, PacketArch Arch>
struct PacketAlignCheck<dim, UnaryMapExp<OP, TA, DType, etype>, Arch> {
  inline static bool Check(const UnaryMapExp<OP, TA, DType, etype> &t) {
    return PacketAlignCheck<dim, TA, Arch>::Check(t.src_);
  }
};
```

4.8 MapPacketPlan

`MapPacketPlan` finishes the evaluation and final assignment.

```
I wonder why the `Tensor` here is 2-D. One possible explanation is the `MapPlan()`
function in `./tensor_cpu-inl.h` is 2-D for the usage of transpose. But the transpose
is not written in the context of Packet. So either the transpose can be added or
just change it to 1-D ???
```

```
template<typename SV, typename E, int dim, typename DType, PacketArch Arch>
inline void MapPacketPlan(Tensor<cpu, dim, DType> _dst,
                          const expr::PacketPlan<E, DType, Arch>& plan) {
  Tensor<cpu, 2, DType> dst = _dst.FlatTo2D();
  const index_t xlen = packet::LowerAlign<DType, Arch>(dst.size(1));
  for (index_t y = 0; y < dst.size(0); ++y) {
    for (index_t x = 0; x < xlen; x += packet::Packet<DType, Arch>::kSize) {
      packet::Saver<SV, DType, Arch>::Save(&dst[y][x], plan.EvalPacket(y, x));
    }
    for (index_t x = xlen; x < dst.size(1); ++x) {
      SV::Save(dst[y][x], plan.Eval(y, x));
    }
  }
}
```

The code above implement a two-step evaluation strategy, but in current code, the second step can not happen. This is due to the check of `stride_`. Its necessary is under question and remains to be checking.

4.9 Missing Explanations

4.9.1 void* pointer

The type void* is a special pointer type that can hold the address of any object.

Like any other pointer, a void* pointer holds an address, but the type of the object at that address is unknown.

4.9.2 size_t keyword

`size_t` is a machine-specific unsigned type that is guaranteed to be large enough to hold the size of any object in memory. The `size_t` type is defined in the `cstddef` header.

4.10 Missing Components

4.10.1 Packet::Sum()

For Plain packet:

```
MSHADOW_CINLINE DType Sum() const {
  return data_;
}
```

For SSE2 packet:

```
MSHADOW_CINLINE float Sum() const {
  __m128 ans  = _mm_add_ps(data_, _mm_movehl_ps(data_, data_));
  __m128 rst  = _mm_add_ss(ans, _mm_shuffle_ps(ans, ans, 1));
#if defined(_MSC_VER) && (_MSC_VER <= 1500) && defined(_WIN64)
  return rst.m128_f32[0];
#else
  float rr = _mm_cvtss_f32(rst);
  return rr;
#endif
}

inline double Sum(void) const {
  __m128d tmp =  _mm_add_sd(data_, _mm_unpackhi_pd(data_, data_));
#if defined(_MSC_VER) && (_MSC_VER <= 1500) && defined(_WIN64)
  return tmp.m128d_f64[0];
#else
  double ans = _mm_cvtsd_f64(tmp);
  return ans;
#endif
}
```

4.10.2 MapPacketPlan for MakeTensorExp

this function will never be called, since the `MakeTensorExp` type will not pass the Check() function. So we tempararily ignore it.

```
template<PacketArch Arch, typename T, int dim, typename DType>
inline PacketPlan<T, DType, Arch>
MakePacketPlan(const MakeTensorExp<T, cpu, dim, DType> &e) {
  return PacketPlan<T, DType, Arch>(e.real_self());
}
```

CHAPTER 5

Tensor_blob.h

`./tensor_blob.h` consists of common representation of arbitrary dimension tensor, which can be used to transforme to normal fixed dimension tensor.

- TShape
- TBlob

5.1 TShape

`TShape` is a dynamic shape class (not a template class) that can hold shape of arbitrary dimension.

The shape will be stored in `data_stack_` when dimension is smaller than `kStackCache`. When it is bigger, it will be stored in `data_heap_`.

5.1.1 Member Variables

- `kStackCache`: size of space in stack.
- `ndim_`: number of dimensions of the shape.
- `num_heap_allocated_`: number of cells allocated in `data_heap_`.
- `data_stack_[kStackCache]`: stack space used to store shape when dimension is small.
- `*data_heap_`: space to store shape when dimension is big.

```
static const index_t kStackCache = 4;
index_t ndim_;
index_t num_heap_allocated_;
index_t data_stack_[kStackCache];
index_t *data_heap_;
```

5.1.2 Constructors

Default Constructor

The default constructor can be a good way to know which variable is required to be initialized.

In this case, we have to initiate `ndim_` (kep variable in the context), `num_heap_allocated_` (to decide whether uses heap), and `data_heap_`.

Since `kStackCache` is a static const, and `data_stack_` is automatically initialized, we can ignore them.

```
TShape()
    : ndim_(0),
      num_heap_allocated_(0),
      data_heap_(NULL) {}
```

Explicit Constructor

The explicit constructor constructs an "all-one" `TShape` with given dimensions.

```
explicit TShape(index_t ndim)
    : ndim_(ndim) {
  if (ndim_ <= kStackCache) {
    data_heap_ = NULL;
    num_heap_allocated_ = 0;
    std::fill_n(data_stack_, ndim_, 1);
  } else {
    data_heap_ = new index_t[ndim_];
    num_heap_allocated_ = ndim_;
    std::fill_n(data_heap_, ndim_, 1);
  }
}
```

Constructor from TShape

The idea is same as the explicit constructor. The only difference is it is an itentical copy by using `std::copy()`.

```
TShape(const TShape &s)
    : ndim_(s.ndim_) {
  if (ndim_ <= kStackCache) {
    data_heap_ = NULL;
    num_heap_allocated_ = 0;
    std::copy(s.data_stack_, s.data_stack_ + ndim_, data_stack_);
  } else {
    data_heap_ = new index_t[ndim_];
    num_heap_allocated_ = ndim_;
    std::copy(s.data_heap_, s.data_heap_ + ndim_, data_heap_);
  }
}
```

Constructor from RandomAccessIterator

The beginning of this constructor is same as a default constructor, the difference lies in the calling of member function `CopyFrom()`.

```
template<typename RandomAccessIterator>
TShape(RandomAccessIterator begin,
       RandomAccessIterator end)
    : ndim_(0),
      num_heap_allocated_(0),
      data_heap_(NULL) {
  this->CopyFrom(begin, end);
}
```

The `CopyFrom()` function first set the dimension of shape by calling `SetDim()`. Then, copy the data from range `[first,end)` to `data()`, which is determined by `ndim_` compared to `kStackCache`.

```
template<typename RandomAccessIterator>
inline void CopyFrom(RandomAccessIterator begin,
                     RandomAccessIterator end) {
  this->SetDim(end - begin);
  std::copy(begin, end, data());
}
```

`SetDim()` also choose to initiate `data_heap_` or not by the value of `ndim_`.

```
inline void SetDim(index_t dim) {
  if (dim > kStackCache &&
      dim > num_heap_allocated_) {
    // data_heap_ can be NULL
    delete [] data_heap_;
    data_heap_ = new index_t[dim];
    num_heap_allocated_ = dim;
  }
  ndim_ = dim;
}

inline index_t *data() {
  return ndim_ <= kStackCache ? data_stack_ : data_heap_;
}
```

Move Constructor from TShape

According to the property of move constructor, we first transfer the member variables from input arguments to the new `TShape`. Then, the original pointer of `s.data_heap_` is reset to `NULL`.

```
TShape(TShape &&s)
    : ndim_(s.ndim_),
      num_heap_allocated_(s.num_heap_allocated_),
      data_heap_(s.data_heap_) {
  if (ndim_ <= kStackCache) {
    std::copy(s.data_stack_, s.data_stack_ + ndim_, data_stack_);
  }
  // remove data heap space from s
  s.data_heap_ = NULL;
}
```

Move Constructor from Shape

Similar to constructor from `RandomAccessIterator`, the beginning of this constructor is same as a default constructor, the difference lies in the calling of member function `CopyFrom()`.

The `CopyFrom()` function takes in two iterators as its input. It first set the dimension of shape by calling `SetDim()`. Then, copy the data from range `[first,end)` to `data()`, which is determined by `ndim_` compared to `kStackCache`.

```
template<int dim>
TShape(Shape<dim> &&s)
    : ndim_(0),
      num_heap_allocated_(0),
      data_heap_(NULL) {
```

```
  this->CopyFrom(s.shape_, s.shape_ + dim);
}
```

Destructor

Since `data_heap_` is the only member variables with a pointer-related type, we explicitly delete it in destructor.

```
~TShape() {
  // data_heap_ can be NULL
  delete [] data_heap_;
}
```

5.1.3 Overloaded Operators

Overloaded =

Overloaded from `Tshape`

It first sets the shape of itself to `shape.ndim_`, then assigns the first address of data (`data_stack_` or `data_heap_`) to a temp variable `src`. At last, it copies the contents pointed by `src` to the `data()` of itself.

```
inline TShape &operator=(const TShape &shape) {
  this->SetDim(shape.ndim_);
  const index_t *src = shape.data();
  std::copy(src, src + ndim_, data());
  return *this;
}
```

Overloaded from `std::vector`

The `CopyFrom()` function takes in two iterators as its input, which is suitable for the two variables `shape.begin()` and `shape.end()`. It first set the dimension of shape by calling `SetDim()`. Then, copy the data from range `[first,end)` to `data()`, which is determined by `ndim_` compared to `kStackCache`.

```
inline TShape &operator=(const std::vector<index_t> &shape) {
  this->CopyFrom(shape.begin(), shape.end());
  return *this;
}
```

Overloaded from `Shape`

It first sets the shape of itself to `dim`. Then, it definite a pointer pointing to `data_stack_` or `data_heap_` by comparing `dim` to `kStackCache`. At last, it just do a element-wise assignment from `shape` to itself.

```
template<int dim>
inline TShape &operator=(const Shape<dim> &shape) {
  this->SetDim(dim);
  index_t *d = dim <= kStackCache ? data_stack_ : data_heap_;
  for (int i = 0; i < dim; ++i) {
    d[i] = shape[i];
  }
```

```
  return *this;
}
```

Overloaded []

It returns the dimension by calling `data()` to fetch the first address, and uses operator `[]` to reach the value.

```
Is it safe to leave the bound check out ???
```

```
inline index_t &operator[](index_t i) {
  return data()[i];
}

inline const index_t &operator[](index_t i) const {
  return data()[i];
}
```

Overloaded ==

Overloaded from TShape

It returns whether two shape equals.

```
inline bool operator==(const TShape &s) const {
  if (ndim_ != s.ndim_) return false;
  if (ndim_ <= kStackCache) {
    for (index_t i = 0; i < ndim_; ++i) {
      if (data_stack_[i] != s.data_stack_[i]) return false;
    }
  } else {
    for (index_t i = 0; i < ndim_; ++i) {
      if (data_heap_[i] != s.data_heap_[i]) return false;
    }
  }
  return true;
}
```

Overloaded from Shape

It returns whether two shape equals.

```
template<int dim>
inline bool operator==(const Shape<dim> &s) const {
  if (ndim_ != dim) return false;
  const index_t *d = dim <= kStackCache ? data_stack_ : data_heap_;
  for (index_t i = 0; i < dim; ++i) {
    if (d[i] != s.shape_[i]) return false;
  }
  return true;
}
```

Overloaded !=

Overloaded from TShape

It returns whether two shape not equals.

```
inline bool operator!=(const TShape &s) const {
  return !(*this == s);
}
```

Overloaded from Shape

It returns whether two shape not equals.

```
template<int dim>
inline bool operator!=(const Shape<dim> &s) const {
  return !(*this == s);
}
```

Overloaded from <<

It outputs a python-style tuple as in the class `Shape`.

```
inline std::ostream &operator<<(std::ostream &os, const TShape &shape) {
  os << '(';
  for (index_t i = 0; i < shape.ndim(); ++i) {
    if (i != 0) os << ',';
    os << shape[i];
  }
  // python style tuple
  if (shape.ndim() == 1) os << ',';
  os << ')';
  return os;
}
```

Overloaded from >>

It reads the input `shape` from the istream and assigns it to a `TShape` type.

There are two `while` loops in the overloaded function for 1-D shape and multi-dimensional shape respectively.

in the first `while` loop, we first peek the next character in a input stream. Then, we judge whether it is a digit or not.

if it is a digit, it means we have a shape with only one dimension. Thus, we input it to a variable named `idx`, set the `ndim_` of `shape` to be `1`, and copy the value of `idx` to `data()` (Refer to the definition of `CopyFrom()`).

Otherwise, we use `is.get()` to extract the character, but do not assign it to anywhere. Again, we judge whether the input is a `(` to support multi-dimension. If it is, we break the `while` loop and move to the second part. If not, we judge whether it is a `space`. If true, we continue the process to see what is the next character. Otherwise, we set the `failbit` to disable the following `istream` and return.

If we receive a `(` in the first part, meaning we will receive the shape of a multi-dimensional tensor, we move to the next part.

In the beginning, we define a `index_t` type `idx` to receive the elements of each dimension, and a `std::vector` to represent the complete shape.

In the second `while` loop, we first fetch the next character by `is>>idx`. It automatically neglects any `space` and check the success of read. It may set a error bit if read fails and break the `while` loop, e.g. input a char `a` to `idx`.

If the input is correct `index_t` type, we push it into the vector, and recursively get the next character until it is not a `space`.

If next is a `L` as the representation of `long` type in Python, we do the fetch again to receive next character.

If next is `,`, then we recursively peek the next character until it is not a space. If the peeked variable is `)`, we break the whole `while` loop. Otherwise we return to the start of `while` loop.

At last, we copy `tmp` to `shape`, even there is a false input, e.g. `(3,4,a)`. We will receive a `tmp` with value `(3,4)` and similarly copy it to `shape`.

```
inline std::istream &operator>>(std::istream &is, TShape &shape) {
  // first part
  while (true) {
    char ch = is.peek();
    if (isdigit(ch)) {
      index_t idx;
      if (is >> idx) {
        shape.CopyFrom(&idx, &idx + 1);
      }
      return is;
    }
    is.get();
    if (ch == '(') break;
    if (!isspace(ch)) {
      is.setstate(std::ios::failbit);
      return is;
    }
  }

  // second part
  index_t idx;
  std::vector<index_t> tmp;
  while (is >> idx) {
    tmp.push_back(idx);
    char ch;
    do {
      ch = is.get();
    } while (isspace(ch));
    if (ch == 'L') {
      ch = is.get();
    }
    if (ch == ',') {
      while (true) {
        ch = is.peek();
        if (isspace(ch)) {
          is.get(); continue;
        }
        if (ch == ')') {
          is.get(); break;
        }
        break;
      }
      if (ch == ')') break;
    } else if (ch == ')') {
      break;
    } else {
```

```
      is.setstate(std::ios::failbit);
      return is;
    }
  }
  shape.CopyFrom(tmp.begin(), tmp.end());
  return is;
}
```

Usage:

```
int main() {
  TShape a;
  cout << (cin >> a) << endl;
  return 0;
}

// correct example
// 3          ->    (3,)
// (3,5)      ->    (3,5)
// (3 , 5)    ->    (3,5)
// (3, 4L, 5) ->    (3,4,5)
// incorrect example
// a          ->    wrong!
// (3,4,a)    ->    (3,4)
```

5.1.4 Member Functions

CopyFrom

The `CopyFrom()` function first set the dimension of shape by calling `SetDim()` with argument equaling the difference between iterators `begin` and `end`. Then, copy the data from range `[begin,end)` to `data()`, which is determined by `ndim_` compared to `kStackCache`.

```
template<typename RandomAccessIterator>
inline void CopyFrom(RandomAccessIterator begin,
                     RandomAccessIterator end) {
  this->SetDim(end - begin);
  std::copy(begin, end, data());
}
```

data

It returns the data content of the `TShape`.

The reason that these two functions can be overloaded is they are called according to the constness of their corresponding variables. If a variable is `const`, then the `const` version will be called, and vice versa.

Actually, it may just according to the implicit `this` argument, by checking `this` is a `const` type or not.

```
inline const index_t *data() const {
  return ndim_ <= kStackCache ? data_stack_ : data_heap_;
}

inline index_t *data() {
  return ndim_ <= kStackCache ? data_stack_ : data_heap_;
}
```

ndim

It simply returns the private member variable `ndim_`.

```
inline index_t ndim(void) const {
  return ndim_;
}
```

Size

It returns the multiplication of the values from all dimensions.

Its return type is `size_t`, which is a machine-related type that is large enough to hold any size can be stored in memory.

```
inline size_t Size(void) const {
  size_t size = 1;
  const index_t *d = this->data();
  for (index_t i = 0; i < ndim_; ++i) {
    size *= d[i];
  }
  return size;
}
```

FlatTo2D

It flattens the higher dimension of `TShape` to second dimension, returns a 2D `shape`.

```
inline Shape<2> FlatTo2D(void) const {
  Shape<2> s;
  if (ndim_ == 0) return Shape2(0, 0);
  const index_t *d = this->data();
  s.shape_[1] = d[ndim_ - 1];
  index_t ymax = 1;
  for (index_t i = 1; i < ndim_; ++i) {
    ymax *= d[i - 1];
  }
  s.shape_[0] = ymax;
  return s;
}
```

FlatTo3D

It flattens the shape into three parts: `[0, axis_begin)`, `[axis_begin, axis_end]` , and `(axis_end, ndim)`.

```
inline Shape<3> FlatTo3D(index_t axis_begin, index_t axis_end) const {
  CHECK(axis_end >= axis_begin);
  Shape<3> s;
  if (ndim_ == 0) return Shape3(0, 0, 0);
  const index_t *d = this->data();
  s.shape_[0] = 1;
  s.shape_[1] = 1;
  s.shape_[2] = 1;
```

```
  for (index_t i = 0; i < axis_begin; ++i) {
    s.shape_[0] *= d[i];
  }
  for (index_t i = axis_begin; i <= axis_end; ++i) {
    s.shape_[1] *= d[i];
  }
  for (index_t i = axis_end + 1; i < ndim_; ++i) {
    s.shape_[2] *= d[i];
  }
  return s;
}
```

It flattens the axis before and after the specified axis, so it becomes 3D tensor.

```
inline Shape<3> FlatTo3D(index_t axis) const {
  return FlatTo3D(axis, axis);
}
```

5.1.5 ProdShape

It returns product of shape in `[dimstart,dimend)`.

```
inline index_t ProdShape(int dimstart, int dimend) const {
  index_t num = 1;
  const index_t *d = this->data();
  for (int i = dimstart; i < dimend; ++i) {
    num *= d[i];
  }
  return num;
}
```

5.1.6 get

It returns the class `shape`, which is a component of `Tensor`.

```
template<int dim>
inline Shape<dim> get(void) const {
  CHECK_EQ(dim, ndim_) << "dimension do not match target dimension " << dim << " vs " << ndim_;
  const index_t *d = this->data();
  Shape<dim> s;
  for (int i = 0; i < dim; ++i) {
    s[i] = d[i];
  }
  return s;
}
```

5.1.7 SetDim (`private`)

Not only does it set the value of `ndim_`, but it decides the usage of `data_stack_` or `data_heap_`.

```
inline void SetDim(index_t dim) {
  if (dim > kStackCache &&
      dim > num_heap_allocated_) {
    // data_heap_ can be NULL
```

```
    delete [] data_heap_;
    data_heap_ = new index_t[dim];
    num_heap_allocated_ = dim;
  }
  ndim_ = dim;
}
```

5.2 TBlob

Tensor blob class (`TBlob`) that can be used to hold tensor of any dimension, any device and any data type. This is only a weak type that can be used to transfer data through interface. `TBlob` itself do not involve any arithmetic operations, but it can be converted to `Tensor` of fixed dimension for further operations.

Like `Tensor`, this data structure is like a pointer class and do not implicit allocated, de-allocate space. This data structure can be helpful to hold tensors of different dimensions and wait for further processing.

5.2.1 Member Variables

There are five member variables belonging to `TBlob`.

- `dptr_`: pointer to the data
- `shape_`: shape of the tensor `TShape`
- `stride_`: storing the stride information in x dimension
- `dev_mask_`: device mask of the corresponding device
- `type_flag_`: typ flag of the tensor blob

```
void *dptr_;
TShape shape_;
index_t stride_;
int dev_mask_;
int type_flag_;
```

5.2.2 Constructor

Default Constructor

Default `TBlob` is set with `dptr_` to be `NULL`, `dev_mask_` to be the mask of `cpu`, and `type_flag_` to be the flag of `default_real_t`, which is actually `float` type.

```
TBlob(void)
    : dptr_(NULL), dev_mask_(cpu::kDevMask),
      type_flag_(DataType<default_real_t>::kFlag) {}
```

Constructor from `TShape`

It constructs `TBlob` from contiguous memory by setting the `stride_` to be the highest dimension of `TShape`.

```
template<typename DType>
TBlob(DType *dptr,
      const TShape &shape,
      int dev_mask)
    : dptr_(dptr), shape_(shape),
      stride_(shape[shape.ndim() - 1]),
      dev_mask_(dev_mask),
      type_flag_(DataType<DType>::kFlag) {}
```

Constructor from `TShape` with type

It constructs `TBlob` from contiguous memory by setting the `stride_` to be the highest dimension of `TShape`, and provides a user-defined `type_flag`.

```
TBlob(void *dptr,
      const TShape &shape,
      int dev_mask,
      int type_flag)
    : dptr_(dptr), shape_(shape),
      stride_(shape[shape.ndim() - 1]),
      dev_mask_(dev_mask),
      type_flag_(type_flag) {}
```

Constructor from `Tensor`

It uses the overloaded assignment operator = to construct `TBlob` from `Tensor`. The detail of = can be referred to following notes.

```
template<typename Device, int dim, typename DType>
TBlob(const Tensor<Device, dim, DType> &src) {
  *this = src;
}
```

5.2.3 Overloaded Operators

Only the assignment operator is overloaded in class `TBlob`. The `rhs` should only be the `Tensor` type.

```
template<typename Device, int dim, typename DType>
inline TBlob
&operator=(const Tensor<Device, dim, DType> &src) {
  dptr_ = src.dptr_;
  shape_ = src.shape_;
  stride_ = src.stride_;
  dev_mask_ = Device::kDevMask;
  type_flag_ = DataType<DType>::kFlag;
  return *this;
}
```

5.2.4 Member Functions

CheckContiguous

It checks whether the `stride_` equals to the highest dimension of `shape_`.

```
inline bool CheckContiguous(void) const {
  return shape_[shape_.ndim() - 1] == stride_;
}
```

FlatTo2D

It first checks the consistency of device and type between the desired return `Tensor` and `TBlob` itself. Then, it calls the constructor of `Tensor`.

```
template<typename Device, typename DType>
inline Tensor<Device, 2, DType> FlatTo2D(Stream<Device> *stream = NULL) const {
  CHECK(Device::kDevMask == dev_mask_)
    << "TBlob.get: device type do not match specified type";
  CHECK(DataType<DType>::kFlag == type_flag_)
    << "TBlob.get_with_shape: data type do not match specified type."
    << "Expected: " << type_flag_ << " v.s. given " << DataType<DType>::kFlag;
  return Tensor<Device, 2, DType>(static_cast<DType*>(dptr_),
                                  shape_.FlatTo2D(), stride_, stream);
}
```

ndim

It simply calls the `ndim()` function, which is a member function of its member variable `shape_` with type `TShape`.

```
inline int ndim(void) const {
  return shape_.ndim();
}
```

size

It returns the size of `i`-th dimension.

```
inline index_t size(index_t idx) const {
  return shape_[idx];
}
```

Size

It returns the total number of elements in `Tensor`.

```
inline index_t Size(void) const {
  return shape_.Size();
}
```

get

It fetches the tensor, with respect to specific dimension `dim`. if it do not match the stored dimension, an error will be issued by the function `get()` of class `TBlob`.

```
template<typename Device, int dim, typename DType>
inline Tensor<Device, dim, DType> get(Stream<Device> *stream = NULL) const {
  CHECK(Device::kDevMask == dev_mask_)
    << "TBlob.get: device type do not match specified type";
  CHECK(DataType<DType>::kFlag == type_flag_)
    << "TBlob.get_with_shape: data type do not match specified type."
    << "Expected: " << type_flag_ << " v.s. given " << DataType<DType>::kFlag;
  return Tensor<Device, dim, DType>(static_cast<DType*>(dptr_),
                                    shape_.get<dim>(),
                                    stride_, stream);
}
```

get_with_shape

It fetches a tensor in given shape. If size do not match the stored size, an error will be issued.

It first checks the consistency of device and type between the desired return `Tensor` and `TBlob` itself. Then, it checks whether the `TBlob` is contiguous by comparing the `stride_` to the highest dimension of `shape_`. Moreover, it also compares the sizes of two shapes to make sure the change of shape will not cause memory leak. At last, it calls the constructor of `Tensor` to create a new one according to the given `shape`.

```
template<typename Device, int dim, typename DType>
inline Tensor<Device, dim, DType> get_with_shape(const Shape<dim> &shape,
                                                 Stream<Device> *stream = NULL) const {
  CHECK(Device::kDevMask == dev_mask_)
    << "TBlob.get: device type do not match specified type";
  CHECK(DataType<DType>::kFlag == type_flag_)
    << "TBlob.get_with_shape: data type do not match specified type."
    << "Expected: " << type_flag_ << " v.s. given " << DataType<DType>::kFlag;
  CHECK_EQ(this->CheckContiguous(), true) << "TBlob.get_reshape: must be contiguous";
  CHECK_EQ(this->shape_.Size(), shape.Size())
    << "TBlob.get_with_shape: new and old shape do not match total elements";
  return Tensor<Device, dim, DType>(static_cast<DType*>(dptr_),
                                    shape,
                                    shape[dim - 1],
                                    stream);
}
```

FlatTo3D

It flattens the tensor to `3` dimension by calling its member function `FlatTo3D()`. Its first version collapses the dimension before and after specified axis.

```
template<typename Device, typename DType>
inline Tensor<Device, 3, DType> FlatTo3D(int axis, Stream<Device> *stream = NULL) const {
  return this->get_with_shape<Device, 3, DType>(
      this->shape_.FlatTo3D(axis), stream);
}
```

Its second version collapses the dimension: `[0, axis_begin)`, `axis_begin, axis_end]`, and `(axis_end, ndim)`.

```
template<typename Device, typename DType>
inline Tensor<Device, 3, DType> FlatTo3D(int axis_begin, int axis_end,
  Stream<Device> *stream = NULL) const {
  return this->get_with_shape<Device, 3, DType>(
```

```
      this->shape_.FlatTo3D(axis_begin, axis_end), stream);
}
```

5.3 Missing Explanation

5.3.1 Algorithm: `std::fill_n`

The `std::fill_n` is defined in `<algorithm>`.

```
template<sclass OutputIt, class Size, class T >
OutputIt fill_n( OutputIt first, Size count, const T& value );
```

a possible implementation:

```
template<class OutputIt, class Size, class T>
OutputIt fill_n(OutputIt first, Size count, const T& value)
{
    for (Size i = 0; i < count; i++) {
        *first++ = value;
    }
    return first;
}
```

Assigns the given `value` to the number of `count` elements in the `first` with the range beginning at `first` if `count>0`. Does nothing otherwise.

5.3.2 Algorithm: `std::copy`

The `std::copy` is defined in `<algorithm>`.

```
template< class InputIt, class OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt d_first );
```

a possible implementation:

```
template<class InputIt, class OutputIt>
OutputIt copy(InputIt first, InputIt last,
              OutputIt d_first)
{
    while (first != last) {
        *d_first++ = *first++;
    }
    return d_first;
}
```

Copies all elements in the range `[first, last)` to `d_first`.

5.3.3 Move Constructor

The move constructor must ensure that the moved-from object is left in a state such that destroying that object will be harmless. In particular, once its resources are moved, the original object must no longer point to those moved resources—responsibility for those resources has been assumed by the newly created object.

Unlike the copy constructor, the move constructor does not allocate any new memory; it takes over the memory in the given argument. Having taken over the memory from its argument, the constructor body sets the pointers in the given object to nullptr.

5.3.4 IOStream: `istream::peak`

```
int peak();
```

It returns the ASCII code of next character in the input sequence, without extracting it: The character is left as the next character to be extracted from the stream.

5.3.5 IOStream: `istream::get`

```
int get();
```

It extracts a single character from the stream.

5.3.6 IOStream: `istream::setstate`

```
void setstate (iostate state);
```

It modifies the current internal error state flags by combining the current flags with those in argument state (as if performing a bitwise OR operation).

5.3.7 IOStream: `std::ios::failbit`

`std::ios::failbit` represents logical error on i/o operation

Once an error has occurred, subsequent IO operations on that stream will fail.

5.3.8 Condition State of `istream`

The easiest way to determine the state of a stream object is to use that object as a condition.

The `while` condition checks the state of the stream returned from the >> expression. If that input operation succeeds, the state remains valid and the condition will succeed. For example,

```
while (is >> idx) { // `>>` input a value to `idx`
    do_some_thing();
}
```

5.4 Missing Explanation

5.4.1 Load and Save

These two functions are related to the functions in `./io.h`, and we leave their explanations to the related parts.

```
template<typename TStream>
inline void Save(TStream *strm) const {
  strm->Write(&ndim_, sizeof(ndim_));
  strm->Write(data(), sizeof(index_t) * ndim_);
}


template<typename TStream>
inline bool Load(TStream *strm) {
  if (strm->Read(&ndim_, sizeof(ndim_)) != sizeof(ndim_)) return false;
  this->SetDim(ndim_);
  size_t nread = sizeof(index_t) * ndim_;
  if (strm->Read(data(), nread) != nread) return false;
  return true;
}
```

CHAPTER 6

/dmlc-core/parameter.h

6.1 Overview

[TOC]

`parameter.h` provides lightweight utilities to do parameter setup and checking.

- Predefined enum and struct
- FieldAccessEntry
- FieldEntryBase
- FieldEntryNumeric
- FieldEntry
- ParamManager
- ParamManagerSingleton
- Parameter
- MACRO

6.2 1. Predefined enum and struct

6.2.1 1.1 ParamInitOption

`ParamInitOption` is the option in parameter initialization.

- `kAllowUnknown`: allow unknown parameters
- `kAllMatch`: need to match exact parameters
- `kAllowHidde`: allow unmatched hidden field with format __*__

```
enum ParamInitOption {
  kAllowUnknown,
  kAllMatch,
  kAllowHidde
};
```

6.2.2 1.2 ParamFieldInfo

`ParamFieldInfo` provides information about a parameter field in string representations.

- `name`: name of the field
- `type`: type of the field in string format
- `type_info_str`: This includes the default value, enum constrain and typename
- `description`: detailed description of the type

```
struct ParamFieldInfo {
  std::string name;
  std::string type;
  std::string type_info_str;
  std::string description;
};
```

6.3 2. FieldAccessEntry

`FieldAccessEntry` is an internal interface to help managge the parameters. Each entry can be used to access one parameter in the `struct` inherited from `Parameter`.

Following is an overview of its member variables and functions, which are all defined as virtual function

```
class FieldAccessEntry {
 public:
  FieldAccessEntry()
      : has_default_(false) {}
  virtual ~FieldAccessEntry() {}
  virtual void SetDefault(void *head) const = 0;
  virtual void Set(void *head, const std::string &value) const = 0;
  virtual void Check(void *head) const {}
  virtual std::string GetStringValue(void *head) const = 0;
  virtual ParamFieldInfo GetFieldInfo() const = 0;

 protected:
  bool has_default_;
  size_t index_;
  std::string key_;
  std::string type_;
  std::string description_;
  virtual void PrintDefaultValueString(std::ostream &os) const = 0;
  friend class ParamManager;
};
```

We add some brief introduction of the concepts in above code.

6.3.1 2.1 Pure Virtual Function

Pure virtual function is a special type of virtual function by adding `= 0` in the back.

A class containing one or more pure virtual functions is called `abstract class`. We cannot declare instances from this class. It only serve as a `base class` for its `derived class`. Moreover, unless all pure virtual functions are defined in the `derived class`, we can still not declare instances using the `derived class`.

6.3.2 2.2 Virtual Destructor

When the `base class` has virtual function, its `destructor` must be defined to be `virtual`. If not, when deconstruct its child classes, the system will call the destructor of `base class`, which will lead to the memory leak, since the variables owned by child classes will not be deleted.

6.4 3. FieldEntryBase

```
template<typename TEntry, typename DType>
class FieldEntryBase : public FieldAccessEntry
```

`FieldEntryBase` is a base class of all `FieldEntry` class, but itself is inherited from `FieldAccessEntry` with all pure virtual function defined.

6.4.1 3.1 Member Variables

```
ptrdiff_t offset_;
DType default_value_;

// together with inherited ones
bool has_default_;
size_t index_;
std::string key_;
std::string type_;
std::string description_;
```

`ptrdiff_t`, like `size_t` (`unsigned`), is a machine-related type (`signed`), defined in `cstddef` head file.

In the context of `dmlc-core`, we use it to identify the offset of the address between the `head` and the `target instance`.

6.4.2 3.2 Member Functions

3.2.1 `public:` Init

`Init()` function sets values for member variables `key_`, `type_`, and `offset_`.

```
inline void Init(const std::string &key,
                 void *head, DType &ref) {
  this->key_ = key;
  if (this->type_.length() == 0) {
    this->type_ = dmlc::type_name<DType>();
  }
  this->offset_ = ((char*)&ref) - ((char*)head);
}
```

The code `this->type_.length() == 0` means the `type_` is only set once and cannot change.

Moreover, the function `dmlc::type_name<DType>()` is defined in `./type_traits.h`.

```
#define DMLC_DECLARE_TYPE_NAME(Type, Name)            \
  template<>                                          \
  inline const char* type_name<Type>() {              \
    return Name;                                      \
```

```
  }

DMLC_DECLARE_TYPE_NAME(float, "float");
DMLC_DECLARE_TYPE_NAME(double, "double");
DMLC_DECLARE_TYPE_NAME(int, "int");
DMLC_DECLARE_TYPE_NAME(uint32_t, "int (non-negative)");
DMLC_DECLARE_TYPE_NAME(uint64_t, "long (non-negative)");
DMLC_DECLARE_TYPE_NAME(std::string, "string");
DMLC_DECLARE_TYPE_NAME(bool, "boolean");
```

3.2.2 protected: Get

`Get()` function fetches the internal representation of parameters.

For example, if this entry corresponds field `param.learning_rate` then `Get(&param)` will return reference to `param.learning_rate`. Its realization thanks to the `offset_` between `head` and the target instance.

```
inline DType &Get(void *head) const {
  return *(DType*)((char*)(head) + offset_);
}
```

First, it converts `head` wity type `void*` to `char*`. Second it adds the `offset_` to `head`. Third, it converts `head` added by `offset_` to `DType*`. Last, it returns the value of `head` with type `DType*`.

Note, `protected` variables or functions can be accessed by `friend` or `hild class`.

3.2.3 public: Set

`Set()` function set the target instance with `value`. In its implementation, it explicitly avoids `space` in the value.

Important Note: the value returned by `Get()` is `DType`. Thanks to the overloaded operator >> in `istringstream`, we can convert the `string` type of `is` to the corresponding `DType`.

```
istream& operator>> (bool& val);
istream& operator>> (short& val);
istream& operator>> (unsigned short& val);
istream& operator>> (int& val);
istream& operator>> (unsigned int& val);
istream& operator>> (long& val);
istream& operator>> (unsigned long& val);
istream& operator>> (float& val);
istream& operator>> (double& val);
istream& operator>> (long double& val);
istream& operator>> (void*& val);
```

```
virtual void Set(void *head, const std::string &value) const {
  std::istringstream is(value);
  is >> this->Get(head);
  // the following codes is designed for avoiding `space` in the value
  if (!is.fail()) {
    while (!is.eof()) {
      int ch = is.get(); // `.get()` returns the next character of istringstream
      if (ch == EOF) {
        is.clear(); break;  // `.clear()` clears all error flags in default
      }
      if (!isspace(ch)) {   // `isspace()` is a C function to check space
        is.setstate(std::ios::failbit); break;  // if we run into space, we manually set the failbit
```

```
                                                    // so the program will go into the next `if()`
      }
    }
  }

  if (is.fail()) {
    std::ostringstream os;
    os << "Invalid Parameter format for " << key_
       << " expect " << type_ << " but value=\'" << value<< '\'';
    throw dmlc::ParamError(os.str());
  }
}
```

The declaration of explicit constructor of `istringstream` is

```
explicit istringstream (const string& str, ios_base::openmode which = ios_base::in);
// str: A string object, whose content is copied.
```

3.2.4 `public:` set_default

In default, class inherited from `FieldEntryBase` do not have a default value. `set_default()` function changes it by setting `has_default_` to be `true` and sets `default_value_` defined by itself to the value of input variable. At last, it returns itself to allow chaining.

```
inline TEntry &set_default(const DType &default_value) {
  default_value_ = default_value;
  has_default_ = true;
  return this->self();
}
```

3.2.5 `public:` self

`self()` first change the type of pointer `this` to the pointer type of its sub-class. Then, it returns the pointed instance of the pointer.

```
inline TEntry &self() {
  return *(static_cast<TEntry*>(this));
}
```

Note: a `static_cast` is useful to perform a conversion that the compiler will not generate automatically.

3.2.6 `public:` SetDefault

Since the function `set_default()` changes the flag `has_default_` to be `true`, we are able to modify the target instance with the pre-set `default_value_`. If it is called before `set_default()`, it will throw out a error.

```
virtual void SetDefault(void *head) const {
  if (!has_default_) {
    std::ostringstream os;
    os << "Required parameter " << key_
       << " of " << type_ << " is not presented";
    throw dmlc::ParamError(os.str());
  } else {
    this->Get(head) = default_value_;
```

```
  }
}
```

3.2.7 public: describe

It simply sets the self-defined member variables `description_` to the input.

```
inline TEntry &describe(const std::string &description) {
  description_ = description;
  return this->self();
}
```

3.2.8 protected: PrintValue

`PrintValue()` simply prints the value with `DType`.

```
virtual void PrintValue(std::ostream &os, DType value) const {
  os << value;
}
```

3.2.9 protected: PrintDefaultValueString

`PrintDefaultValueString()` simply prints the default value.

```
virtual void PrintDefaultValueString(std::ostream &os) const {
  PrintValue(os, default_value_);
}
```

3.2.10 public: GetStringValue

`GetStringValue()` simply prints the value it stores.

```
virtual std::string GetStringValue(void *head) const {
  std::ostringstream os;
  PrintValue(os, this->Get(head));
  return os.str();
}
```

3.2.11 public: GetFieldInfo

It simply sets information of the class into the four member variables in `ParamFieldInfo`, and returns it.

```
virtual ParamFieldInfo GetFieldInfo() const {
  ParamFieldInfo info;
  std::ostringstream os;
  info.name = key_;
  info.type = type_;
  os << type_;
  if (has_default_) {
    os << ',' << " optional, default=";
    PrintDefaultValueString(os);
  } else {
```

```
    os << ", required";
  }
  info.type_info_str = os.str();
  info.description = description_;
  return info;
}
```

6.5 4. FieldEntryNumeric

`FieldEntryNumeric` is a base class for numeric types that have ranges. It is the child class of `FieldEntryBase` and grandchild class of `FieldAccessEntry`.

```
template<typename TEntry, typename DType>
class FieldEntryNumeric : public FieldEntryBase<TEntry, DType> {};
```

6.5.1 4.1 Member Variables

```
bool has_begin_, has_end_;
DType begin_, end_;

// inherited from `FieldEntryBase`
ptrdiff_t offset_;
DType default_value_;

// inherited from `FieldAccessEntry`
bool has_default_;
size_t index_;
std::string key_;
std::string type_;
std::string description_;
```

6.5.2 4.2 Constructor

In default, a `FieldEntryNumeric` type do not have range constraints.

```
FieldEntryNumeric()
    : has_begin_(false), has_end_(false) {}
```

6.5.3 4.3 Member Functions

4.3.1 set_range

It simply sets the range of `begin` and `end`, and turns on the flag `has_begin_` and `has_end_`.

```
virtual TEntry &set_range(DType begin, DType end) {
  begin_ = begin; end_ = end;
  has_begin_ = true; has_end_ = true;
  return this->self();
}
```

4.3.2 set_lower_bound

It simply sets the `begin_` and turns on the flag `has_begin_`.

```
virtual TEntry &set_lower_bound(DType begin) {
  begin_ = begin; has_begin_ = true;
  return this->self();
}
```

4.3.3 Check

It does a simple checking of parameter constraints.

```
virtual void Check(void *head) const {
  FieldEntryBase<TEntry, DType>::Check(head); // not implement
  DType v = this->Get(head);
  if (has_begin_ && has_end_) {
    if (v < begin_ || v > end_) {
      std::ostringstream os;
      os << "value " << v << "for Parameter " << this->key_
         << " exceed bound [" << begin_ << ',' << end_ <<']';
      throw dmlc::ParamError(os.str());
    }
  } else if (has_begin_ && v < begin_) {
      std::ostringstream os;
      os << "value " << v << "for Parameter " << this->key_
         << " should be greater equal to " << begin_;
      throw dmlc::ParamError(os.str());
  } else if (has_end_ && v > end_) {
      std::ostringstream os;
      os << "value " << v << "for Parameter " << this->key_
         << " should be smaller equal to " << end_;
      throw dmlc::ParamError(os.str());
  }
}
```

6.6 5. FieldEntry

`FieldEntry` defines parsing and checking behavior of `DType`. This class can be specialized to implement specific behavior of more settings.

6.6.1 5.1 FieldEntry<typename DType>

It is a general definition of FieldEntry, which only determines it inheriting from a common `FieldEntryBase` class or a specific `FieldEntryNumeric` class for numeric.

```
template<typename DType>
class FieldEntry :
      public IfThenElseType<dmlc::is_arithmetic<DType>::value,
                            FieldEntryNumeric<FieldEntry<DType>, DType>,
                            FieldEntryBase<FieldEntry<DType>, DType> >::Type {
};
```

Basically, it uses a trick to let `FieldEntry` inherit from different classes. Following codes will be helpful for understanding.

```
// all defined in `./type_traits.h`
template<typename T>
struct is_arithmetic {
#if DMLC_USE_CXX11
  /*! \brief the value of the traits */
  static const bool value = std::is_arithmetic<T>::value;
#else
  /*! \brief the value of the traits */
  static const bool value = (dmlc::is_integral<T>::value ||
                             dmlc::is_floating_point<T>::value);
#endif
};

template<typename Then, typename Else>
struct IfThenElseType<true, Then, Else> {
  typedef Then Type;
};

template<typename Then, typename Else>
struct IfThenElseType<false, Then, Else> {
  typedef Else Type;
};
```

6.6.2 5.2 FieldEntry<int>

```
template<>
class FieldEntry<int> : public FieldEntryNumeric<FieldEntry<int>, int> {};
```

It specializes the definition of FieldEntry for `int` type to implement some specific behaviors.

5.2.1 Member Variables

```
bool is_enum_;
std::map<std::string, int> enum_map_;
std::map<int, std::string> enum_back_map_;

// inherited from `FieldEntryNumeric`
bool has_begin_, has_end_;
DType begin_, end_;

// inherited from `FieldEntryBase`
ptrdiff_t offset_;
DType default_value_;

// inherited from `FieldAccessEntry`
bool has_default_;
size_t index_;
std::string key_;
std::string type_;
std::string description_;
```

5.2.2 Constructor

In default, we cannot do enumeration.

```
FieldEntry<int>() : is_enum_(false) {}
```

5.2.3 Member Functions

5.2.3.1 Set

It adds the consideration of enumeration case. If `is_enum_` is `true`, it will first find the corresponding instance of `value` and return its iterator. Then, it will determine whether to set the value according to the return. All in all, the actual set is still performed by its father class.

```
virtual void Set(void *head, const std::string &value) const {
  if (is_enum_) {
    std::map<std::string, int>::const_iterator it = enum_map_.find(value);
    std::ostringstream os;
    if (it == enum_map_.end()) {
      os << "Invalid Input: \'" << value;
      os << "\', valid values are: ";
      PrintEnums(os);
      throw dmlc::ParamError(os.str());
    } else {
      os << it->second;
      Parent::Set(head, os.str());
    }
  } else {
    Parent::Set(head, value);
  }
}
```

5.2.3.2 add_enum

It only works when the `enum_map_.size()` is `0` and the `key` in `enum_map_` has not been set, and `value` in `enum_back_map_` also has not been set.

Its workflow is set the corresponding relation between `key` and `value` directly and reversely. Then, it will turn on the flag `is_enum_` and return itself for chaining.

```
inline FieldEntry<int> &add_enum(const std::string &key, int value) {
  if ((enum_map_.size() != 0 && enum_map_.count(key) != 0) || \
      enum_back_map_.count(value) != 0) {
    std::ostringstream os;
    os << "Enum " << "(" << key << ": " << value << " exisit!" << ")\n";
    os << "Enums: ";
    for (std::map<std::string, int>::const_iterator it = enum_map_.begin();
         it != enum_map_.end(); ++it) {
      os << "(" << it->first << ": " << it->second << "), ";
    }
    throw dmlc::ParamError(os.str());
  }
  enum_map_[key] = value;
  enum_back_map_[value] = key;
  is_enum_ = true;
```

```
  return this->self();
}
```

```
why not use auto here?
may not get the const_iterator?
why not use cbegin()
```

5.2.3.3 PrintValue

`PrintValue()` directly print the input value of type `int` if the flag `is_enum_` is not on. If the flag is on, it first check whether the value is in the `enum_back_map_`. If it is, it returns the value corresponded to `value` in `enum_back_map_`.

```
virtual void PrintValue(std::ostream &os, int value) const {
  if (is_enum_) {
    CHECK_NE(enum_back_map_.count(value), 0U)
        << "Value not found in enum declared";
    os << enum_back_map_.at(value);
  } else {
    os << value;
  }
}
```

5.2.3.4 PrintDefaultValueString

It simply call the function `PrintValue()` to print its `default_value_`.

```
virtual void PrintDefaultValueString(std::ostream &os) const {
  os << '\'';
  PrintValue(os, default_value_);
  os << '\'';
}
```

5.2.3.5 PrintEnums

It simply output every `string` type, which is the key in the `enum_map_`.

```
inline void PrintEnums(std::ostream &os) const {
  os << '{';
  for (std::map<std::string, int>::const_iterator
           it = enum_map_.begin(); it != enum_map_.end(); ++it) {
    if (it != enum_map_.begin()) {
      os << ", ";
    }
    os << "\'" << it->first << '\'';
  }
  os << '}';
}
```

5.2.3.6 GetFieldInfo

If `is_enum_` is not on, it simply call `GetFieldInfo()` of its parent class. Otherwise, it perform its own codes. The only difference is the `PrintEnums(os)`.

```
virtual ParamFieldInfo GetFieldInfo() const {
  if (is_enum_) {
    ParamFieldInfo info;
    std::ostringstream os;
    info.name = key_;
    info.type = type_;
    PrintEnums(os);
    if (has_default_) {
      os << ',' << "optional, default=";
      PrintDefaultValueString(os);
    } else {
      os << ", required";
    }
    info.type_info_str = os.str();
    info.description = description_;
    return info;
  } else {
    return Parent::GetFieldInfo();
  }
}
```

6.6.3 5.3 FieldEntry<std::string>

It specializes the definition of `FieldEntry` for `std::string` type to implement some specific behaviors.

```
template<>
class FieldEntry<std::string>
    : public FieldEntryBase<FieldEntry<std::string>, std::string> {};
```

5.3.1 Member Functions

5.3.1.1 Set

Since the `Set()` function in its base class `FieldEntryBase` has a design for avoiding `space` in the value, it is naturally insuitable for a specific `std::string` type. So it overloads the `Set()` by directly set the target instance as the input `value`.

```
virtual void Set(void *head, const std::string &value) const {
  this->Get(head) = value;
}
```

5.3.1.2 PrintDefaultValueString

It overloads the `PrintDefaultValueString()` to add single quotes on its two sides.

```
virtual void PrintDefaultValueString(std::ostream &os) const {  // NOLINT(*)
  os << '\'' << default_value_ << '\'';
}
```

6.6.4 5.4 FieldEntry<bool>

It specializes the definition of `FieldEntry` for `bool` type to implement some specific behaviors.

```
template<>
class FieldEntry<bool>
    : public FieldEntryBase<FieldEntry<bool>, bool> {};
```

5.4.1 Member Functions

5.4.1.1 Set

It first define a `string` with same size of `value`. Then it transforms the characters in `value` to be lowercase and stores in the defined string. If the value is one of `true`, `false`, `1`, and `0`, it will set the target instance as `true` of `false` accordingly. Otherwise, it will raise an error.

```
virtual void Set(void *head, const std::string &value) const {
  std::string lower_case; lower_case.resize(value.length());
  std::transform(value.begin(), value.end(), lower_case.begin(), ::tolower);
  bool &ref = this->Get(head);  // directly set the value by reference
  if (lower_case == "true") {
    ref = true;
  } else if (lower_case == "false") {
    ref = false;
  } else if (lower_case == "1") {
    ref = true;
  } else if (lower_case == "0") {
    ref = false;
  } else {
    std::ostringstream os;
    os << "Invalid Parameter format for " << key_
       << " expect " << type_ << " but value=\'" << value<< '\'';
    throw dmlc::ParamError(os.str());
  }
}
```

5.4.1.2 PrintValue

Instead of directly output `true` and `false` as `1` and `0`, it outputs them as a `string`.

```
virtual void PrintValue(std::ostream &os, bool value) const {
  if (value) {
    os << "True";
  } else {
    os << "False";
  }
}
```

6.7 6. ParamManager

```
class ParamManager {};
```

`ParamManager` class is to handle parameter settings for each type in the structure. An `ParamManager` will be created for each parameter structures.

6.7.1 6.1 Member Variables

- `name_`: parameter holding struct name
- `entry_`: positional list of entries
- `entry_map_`: map from key to entry

```
std::string name_;
std::vector<FieldAccessEntry*> entry_;
std::map<std::string, FieldAccessEntry*> entry_map_;
```

6.7.2 6.2 Destructor

`ParamManager` explicitly deletes every `entry_` to trigger their destructors.

```
~ParamManager() {
  for (size_t i = 0; i < entry_.size(); ++i) {
    delete entry_[i];
  }
}
```

6.7.3 6.3 Member Functions

6.3.1 AddEntry

`AddEntry()` is an internal function to add entry to manager, and the manager will take over the ownership of the entry.

It first sets the member variable `index_` in `FieldAccessEntry` to current `entry_.size()`. So, every `FieldEntry` can be sorted according to the order of add. Second, if the `key` is already in the `entry_map_`, it will raise an error. Otherwise, it will push the object of type `FieldAccessEntry` into its `entry_`, and set the map connection between `key` and the object instance.

```
inline void AddEntry(const std::string &key, FieldAccessEntry *e) {
  e->index_ = entry_.size();
  if (entry_map_.count(key) != 0) {
    LOG(FATAL) << "key " << key << " has already been registered in " << name_;
  }
  entry_.push_back(e);
  entry_map_[key] = e;
}
```

6.3.2 AddAlias

`AddAlias()` is to set an alias to existent entry. It first checks whether the `field` name has already registered, and `alias` has not been registered. If both is true, it create a new map connection between `alias` and the object instance.

```
inline void AddAlias(const std::string& field, const std::string& alias) {
  if (entry_map_.count(field) == 0) {
    LOG(FATAL) << "key " << field << " has not been registered in " << name_;
  }
  if (entry_map_.count(alias) != 0) {
    LOG(FATAL) << "Alias " << alias << " has already been registered in " << name_;
  }
```

```
  entry_map_[alias] = entry_map_[field];
}
```

6.3.3 set_name

`set_name()` simply set its member variable `name_` to be the name of `struct`.

```
inline void set_name(const std::string &name) {
  name_ = name;
}
```

6.3.4 GetFieldInfo

`GetFieldInfo()` simply builds a `vector` to store the field information of every entry respectively and returns it.

```
inline std::vector<ParamFieldInfo> GetFieldInfo() const {
  std::vector<ParamFieldInfo> ret(entry_.size());
  for (size_t i = 0; i < entry_.size(); ++i) {
    ret[i] = entry_[i]->GetFieldInfo();
  }
  return ret;
}
```

6.3.5 PrintDocString

`PrintDocString()` is designed for readible docstring.

For every component in `entry_`, it first fetches its corresponding `ParamFieldInfo`, and output its member variable `name` and `type_info_str`. If it has `description`, that will also be output.

```
inline void PrintDocString(std::ostream &os) const {
  for (size_t i = 0; i < entry_.size(); ++i) {
    ParamFieldInfo info = entry_[i]->GetFieldInfo();
    os << info.name << " : " << info.type_info_str << '\n';
    if (info.description.length() != 0) {
      os << "    " << info.description << '\n';
    }
  }
}
```

6.3.6 GetDict

It gets the internal parameters and push them into a vector of pairs. The `first` value in the pair is the name of the instance, and the `second` value is the content of the internal parameters been converted to `string`.

```
inline std::vector<std::pair<std::string, std::string> > GetDict(void * head) const {
  std::vector<std::pair<std::string, std::string> > ret;
  for (std::map<std::string, FieldAccessEntry*>::const_iterator
          it = entry_map_.begin(); it != entry_map_.end(); ++it) {
    ret.push_back(std::make_pair(it->first, it->second->GetStringValue(head)));
  }
  return ret;
}
```

6.3.7 Find

It finds the entry in `entry_map_` by the `key` value.

```
inline FieldAccessEntry *Find(const std::string &key) const {
  std::map<std::string, FieldAccessEntry*>::const_iterator it =
      entry_map_.find(key);
  if (it == entry_map_.end()) return NULL;
  return it->second;
}
```

6.3.8 RunInit

It does the initialization to every entry stored, by the contents inside of `RandomAccessIterator`.

It first checks whether the `key` inside of `it->first` has been registered in `entry_map_`. If it is, it fetches the class `FieldEntry` corresponding to the `key`, and set its actual value to be `it->second`. Then, it will call `Check()` to examine the satisfication of constraints, and insert the processed `FieldEntry` into `selected_args`.

Otherwise, it will check several extra conditions to determine whether keep the program running or raise an error. We will leave the discussion until we meet the case that uses them.

After the initialization, it checks every component inside `entry_map_`, if there are components not being initialized, the function `SetDefault()` will be called.

```
template<typename RandomAccessIterator>
inline void RunInit(void *head,
                    RandomAccessIterator begin,
                    RandomAccessIterator end,
                    std::vector<std::pair<std::string, std::string> > *unknown_args,
                    parameter::ParamInitOption option) const {
  std::set<FieldAccessEntry*> selected_args;
  for (RandomAccessIterator it = begin; it != end; ++it) {
    FieldAccessEntry *e = Find(it->first);
    if (e != NULL) {
      e->Set(head, it->second);
      e->Check(head);
      selected_args.insert(e);
    } else {
      if (unknown_args != NULL) {
        unknown_args->push_back(*it);
      } else {
        if (option != parameter::kAllowUnknown) {
          if (option == parameter::kAllowHidden &&
              it->first.length() > 4 &&
              it->first.find("__") == 0 &&
              it->first.rfind("__") == it->first.length()-2) {
            continue;
          }
          std::ostringstream os;
          os << "Cannot find argument \'" << it->first << "\', Possible Arguments:\n";
          os << "----------------\n";
          PrintDocString(os);
          throw dmlc::ParamError(os.str());
        }
      }
    }
  }
```

```
  for (std::map<std::string, FieldAccessEntry*>::const_iterator it = entry_map_.begin();
       it != entry_map_.end(); ++it) {
    if (selected_args.count(it->second) == 0) {
      it->second->SetDefault(head);
    }
  }
}
```

6.8 7. ParamManagerSingleton

In the execution, the typename `PType` will be replaced by the type of use-defined `struct`. In fact, the process of its initialization is actually also the initialization process for `ParamManager`.

It first defines a `param` with `PType`. Then, call the member function `__DECLARE__()` of `PType` with pointer `this` passing into it. Inside the `__DECLARE__`, we do the initialization of `PType` by manipulating the member variable `manager` with type `ParamManager` through `this` pointer. At last, it set the name of the `manager` according to the name of the user-defined `struct`.

```
template<typename PType>
struct ParamManagerSingleton {
  ParamManager manager;
  explicit ParamManagerSingleton(const std::string &param_name) {
    PType param;
    param.__DECLARE__(this);
    manager.set_name(param_name);
  }
};
```

The detailed discussion of this class should be combined with the following contents.

6.9 8. Macro

In `./parameter.h` we introduce several macros to declare parameter.

6.9.1 8.1 DMLC_DECLARE_PARAMETER

`DMLC_DECLARE_PARAMETER` is used inside the `struct` as a declaration or definition of member functions.

The usage of `DMLC_DECLARE_PARAMETER` is firstly declare a function named `__MANAGER__()`, then define a function named `__DECLARE__()` whose input variable is with type `ParamManagerSingleton`.

```
#define DMLC_DECLARE_PARAMETER(PType)                                  \
  static ::dmlc::parameter::ParamManager *__MANAGER__();               \
  inline void __DECLARE__(::dmlc::parameter::ParamManagerSingleton<PType> *manager) \
```

6.9.2 8.2 DMLC_DECLARE_FIELD

`DMLC_DECLARE_FIELD` should follow right behind `DMLC_DECLARE_PARAMETER`, it call the `DECLARE()` function of class `Parameter` to register parameter.

```
#define DMLC_DECLARE_FIELD(FieldName)  this->DECLARE(manager, #FieldName, FieldName)
```

6.9.3 8.3 DMLC_DECLARE_ALIAS

DMLC_DECLARE_ALIAS is also used inside the struct to store parameters. It always appears after DMLC_DECLARE_PARAMETER and DMLC_DECLARE_FIELD to set alias of defined parameters by calling member function AddAlias() of class Parameter.

```
#define DMLC_DECLARE_ALIAS(FieldName, AliasName)  manager->manager.AddAlias(#FieldName, #AliasName)
```

6.9.4 8.4 DMLC_REGISTER_PARAMETER

This macro need to be put in a source file so that the __MANAGER__() function declared inside the struct can be defined.

The usage of DMLC_REGISTER_PARAMETER is to define the function __MANAGER__() declared in the struct, and call the defined function __MANAGER__() (I wonder the reason of its existence).

```
#define DMLC_REGISTER_PARAMETER(PType)                                     \
  ::dmlc::parameter::ParamManager *PType::__MANAGER__() {                  \
    static ::dmlc::parameter::ParamManagerSingleton<PType> inst(#PType); \
    return &inst.manager;                                                  \
  }                                                                        \
  static DMLC_ATTRIBUTE_UNUSED ::dmlc::parameter::ParamManager&            \
  __make__ ## PType ## ParamManager__ =                                    \
      (*PType::__MANAGER__())
```

We indicates __make__ ## PType ## ParamManager__ will never be used in the context. It only triggers the function __MANAGER__() and accpets its return.

6.10 9. Parameter

Parameter is the base type that every parameter struct should inheritate from.

The following code is a complete example to setup parameters. Note: It is important for every parameter struct inherited from class Parameter<PType>, where PType is itself.

```
struct Param : public dmlc::Parameter<Param> {
  float learning_rate;
  int num_hidden;
  std::string name;
  // declare parameters in header file
  DMLC_DECLARE_PARAMETER(Param) {
    DMLC_DECLARE_FIELD(num_hidden).set_range(0, 1000);
    DMLC_DECLARE_FIELD(learning_rate).set_default(0.01f);
    DMLC_DECLARE_FIELD(name).set_default("hello");
  }
};

// register it in cc file
DMLC_REGISTER_PARAMETER(Param);
```

The only difference between a normal struct is that we will need to declare all the fields, as well as their default value or constraints.

6.10.1 9.1 Member Functions

9.1.1 protected: DECLARE

`DECLARE()` function is the most important member function of `Parameter`. Everytime we want to manipulate the stored parameter inside `struct`, it will be called.

Inside of it, it constructs and inits a new instance with type `FieldEntry<DType>`. The `DType` is determined by the type of its input variable `ref`. At last, it add the entry into the `ParamManager`.

```
template<typename DType>
inline parameter::FieldEntry<DType>& DECLARE(parameter::ParamManagerSingleton<PType> *manager,
                                             const std::string &key, DType &ref) {
  parameter::FieldEntry<DType> *e = new parameter::FieldEntry<DType>();
  e->Init(key, this->head(), ref);
  manager->manager.AddEntry(key, e);
  return *e;
}
```

At here, we can conclude the execution process of the calling `PType::__MANAGER__()`.

- (1) It initializes a new `ParamManagerSingleton`. Inside of the constructor, a temporal `struct` will be built. And the `__DECLARE__()` function of the temporal `struct` will be called to initialize a new `FieldEntry<DType>` added as a new entry of the `ParamManager`.
- (2) The `ParamManager` will be returned for further execution of the `struct` like `Init()`, `__DICT__()` or `__DOC__()`.

9.1.2 public: Init

It initializes the parameter by keyword arguments, stored in a container, which can be a `map` or a `vector` containing `pair`. Inside this function, parameter `struct` will bo initialized, and every parameter will be checked and error will be thrown if something is wrong.

```
template<typename Container>
inline void Init(const Container &kwargs,
                 parameter::ParamInitOption option = parameter::kAllowHidden) {
  PType::__MANAGER__()->RunInit(static_cast<PType*>(this),
                                kwargs.begin(), kwargs.end(),
                                NULL,
                                option);
}
```

Followe the example before, the usage of `Init()` function is:

```
int main() {
   MyParam param;
   std::vector<std::pair<std::string, std::string> > param_data = {
     {"num_hidden", "100"},
     {"learning_rate", "0.1f"},
     {"name", "MyNet"}
   };
   // set the parameters
   param.Init(param_data);
   return 0;
}
```

The execution process is as follows. First, it calls the `__MANAGER__()` function of the `struct`.

```
ParamManager *PType::__MANAGER__() {
  static ::dmlc::parameter::ParamManagerSingleton<PType> inst(#PType);
  return &inst.manager;
}
```

Inside the `__MANAGER__()`, a `ParamManagerSingleton` will be created, whose constructor will call the member function `__DECLARE__()` of the `struct`. Inside struct, every parameter related default value or constraints will all be set. Then, it returns the member variable `manager` with type `ParamManager`.

Second, the member function `RunInit()` of returned `manager` will be called. We only consider the case that all parameter are correctly set, so all parameters are set inside the `RunInit()` referred to its description in this notebook.

```
Let us summarize the usage of `./parameter.h`. It is major difference to normal structure is it
introduce more information into the struct like default value, constraints, and descriptions.

Indeed, the parameter is stored in the same way as usual. But, the difference is the way to
manipulate them. Everytime we want to manipulate them, a temporal `ParamManager` will be created
to handle it.

Once we finish our task, we will run into the end of function `Init()`, which will automatically
trigger the destructor of `ParamManager`.
```

9.1.3 `private:` head

`head()` returns head pointer of child structure

```
inline PType *head() const {
  return static_cast<PType*>(const_cast<Parameter<PType>*>(this));
}
```

9.1.4 `public:` *_DICT_*

It gets the internal parameters and push them into a vector of pairs. The `first` value in the `pair` is the name of the instance, and the `second` value is the content of the internal parameters been converted to `string`. Then, it use `range constructor` to initialize a `map`.

Note of range constructor, We can also initialize an associative container from a range of values, so long as those values can be converted to the type of the container.

```
inline std::map<std::string, std::string> __DICT__() const {
  std::vector<std::pair<std::string, std::string> > vec
      = PType::__MANAGER__()->GetDict(this->head());
  return std::map<std::string, std::string>(vec.begin(), vec.end());
}
```

9.1.5 `public:` *_FIELDS_*

It simply return the result of function `GetFieldInfo()` of a `ParamManager` type.

```
inline static std::vector<ParamFieldInfo> __FIELDS__() {
  return PType::__MANAGER__()->GetFieldInfo();
}
```

9.1.6 public: *_DOC_*

It simply output the result of function `PrintDocString()` of a `ParamManager` type.

```
inline static std::string __DOC__() {
  std::ostringstream os;
  PType::__MANAGER__()->PrintDocString(os);
  return os.str();
}
```

6.11 Missing Explanation

6.11.1 std::string::length()

`size_t length() const;`

`std::string::length()` returns the length of the string, in terms of bytes. e.g.

```
#include <iostream>
#include <string>

int main ()
{
  std::string str ("Test string");
  std::cout << "The size of str is " << str.length() << " bytes.\n";
  return 0;
}

// output: The size of str is 11 bytes
```

6.11.2 map::find()

`map::find(k)` searches the container for an element with a key equivalent to k and returns an iterator to it if found, otherwise it returns an iterator to map::end.

6.11.3 map::size()

`map::size()` returns the number of elements in the map container.

6.11.4 map::count()

`map::count(k)` searches the container for elements with a key equivalent to `k` and returns the number of matches.

Because all elements in a map container are unique, the function can only return `1` (if the element is found) or `0` (otherwise).

6.11.5 map::at()

`std::at(k)` returns a reference to the mapped value of the element identified with key `k`.

If `k` does not match the key of any element in the container, the function throws an `out_of_range` exception.

6.11.6 std::resize()

```
void resize (size_t n);
void resize (size_t n, char c);
```

`std::resize()` resizes the string to a length of `n` characters.

If `n` is smaller than the current string length, the current value is shortened to its first `n` character, removing the characters beyond the `n`-th.

If `n` is greater than the current string length, the current content is extended by inserting at the end as many characters as needed to reach a size of `n`. If `c` is specified, the new elements are initialized as copies of `c`, otherwise, they are value-initialized characters (null characters).

6.11.7 std::transform()

```
template< class InputIt, class OutputIt, class UnaryOperation >
OutputIt transform(InputIt first1, InputIt last1, OutputIt d_first, UnaryOperation unary_op);
```

`std::transform()` applies the given function to a range and stores the result in another range, beginning at `d_first`. The unary operation `unary_op` is applied to the range defined by `[first1, last1)`.

6.11.8 std::tolower()

`std::tolower()` converts the given character to lowercase according to the character conversion rules.

6.11.9 Special Character in `#define`

- `##`: is to combine two operatees, e.g. `#define conn(x,y) x##y` will result `int n = conn(123,456)` as `n = 123456`
- `#@`: is to add single quotes, e.g. `#define ToChar(x) #@x` will result `char a = ToChar(1)` as `a = '1'`
- `#` : is to add double quotes, e.g. `#define ToString(x) #x` will result `char* str = ToString(123)` as `str = "123"`

6.11.10 DMLC_ATTRIBUTE_UNUSED

Normally, the compiler warns if a variable is declared but is never referenced. `__attribute__((unused))` informs the compiler that you expect a variable to be unused and tells it not to issue a warning if it is not used. So, in `dmlc-core`, we define

```
#if defined(__GNUC__)
#define DMLC_ATTRIBUTE_UNUSED __attribute__((unused))
#else
#define DMLC_ATTRIBUTE_UNUSED
#endif
```

6.12 Missing Component

6.12.1 Parameter::InitAllowUnknown

We will consider it later, when we meet it

```
template<typename Container>
inline std::vector<std::pair<std::string, std::string> >
InitAllowUnknown(const Container &kwargs) {
  std::vector<std::pair<std::string, std::string> > unknown;
  PType::__MANAGER__()->RunInit(static_cast<PType*>(this),
                                kwargs.begin(), kwargs.end(),
                                &unknown, parameter::kAllowUnknown);
  return unknown;
}
```

6.12.2 Parameter::Save

We will leave it to the discussion of `./json.h`

```
inline void Save(dmlc::JSONWriter *writer) const {
  writer->Write(this->__DICT__());
}
```

6.12.3 Parameter::Load

We will leave it to the discussion of `./json.h`

```
inline void Load(dmlc::JSONReader *reader) {
  std::map<std::string, std::string> kwargs;
  reader->Read(&kwargs);
  this->Init(kwargs);
}
```

6.12.4 FieldEntry<optional >

Currently, we have no idea of its usage.

```
template<>
class FieldEntry<optional<int> > : public FieldEntryBase<FieldEntry<optional<int> >, optional<int> >
 public:
  // construct
  FieldEntry<optional<int> >() : is_enum_(false) {}
  // parent
  typedef FieldEntryBase<FieldEntry<optional<int> >, optional<int> > Parent;
  // override set
  virtual void Set(void *head, const std::string &value) const {
    if (is_enum_ && value != "None") {
      std::map<std::string, int>::const_iterator it = enum_map_.find(value);
      std::ostringstream os;
      if (it == enum_map_.end()) {
        os << "Invalid Input: \'" << value;
        os << "\', valid values are: ";
```

```
        PrintEnums(os);
        throw dmlc::ParamError(os.str());
      } else {
        os << it->second;
        Parent::Set(head, os.str());
      }
    } else {
      Parent::Set(head, value);
    }
  }
  virtual ParamFieldInfo GetFieldInfo() const {
    if (is_enum_) {
      ParamFieldInfo info;
      std::ostringstream os;
      info.name = key_;
      info.type = type_;
      PrintEnums(os);
      if (has_default_) {
        os << ',' << "optional, default=";
        PrintDefaultValueString(os);
      } else {
        os << ", required";
      }
      info.type_info_str = os.str();
      info.description = description_;
      return info;
    } else {
      return Parent::GetFieldInfo();
    }
  }
  // add enum
  inline FieldEntry<optional<int> > &add_enum(const std::string &key, int value) {
    CHECK_NE(key, "None") << "None is reserved for empty optional<int>";
    if ((enum_map_.size() != 0 && enum_map_.count(key) != 0) || \
        enum_back_map_.count(value) != 0) {
      std::ostringstream os;
      os << "Enum " << "(" << key << ": " << value << " exisit!" << ")\n";
      os << "Enums: ";
      for (std::map<std::string, int>::const_iterator it = enum_map_.begin();
           it != enum_map_.end(); ++it) {
        os << "(" << it->first << ": " << it->second << "), ";
      }
      throw dmlc::ParamError(os.str());
    }
    enum_map_[key] = value;
    enum_back_map_[value] = key;
    is_enum_ = true;
    return this->self();
  }

 protected:
  // enum flag
  bool is_enum_;
  // enum map
  std::map<std::string, int> enum_map_;
  // enum map
  std::map<int, std::string> enum_back_map_;
  // override print behavior
```

```
  virtual void PrintDefaultValueString(std::ostream &os) const { // NOLINT(*)
    os << '\'';
    PrintValue(os, default_value_);
    os << '\'';
  }
  // override print default
  virtual void PrintValue(std::ostream &os, optional<int> value) const {  // NOLINT(*)
    if (is_enum_) {
      if (!value) {
        os << "None";
      } else {
        CHECK_NE(enum_back_map_.count(value.value()), 0)
            << "Value not found in enum declared";
        os << enum_back_map_.at(value.value());
      }
    } else {
      os << value;
    }
  }


 private:
  inline void PrintEnums(std::ostream &os) const {  // NOLINT(*)
    os << "{None";
    for (std::map<std::string, int>::const_iterator
             it = enum_map_.begin(); it != enum_map_.end(); ++it) {
      os << ", ";
      os << "\'" << it->first << '\'';
    }
    os << '}';
  }
};
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search