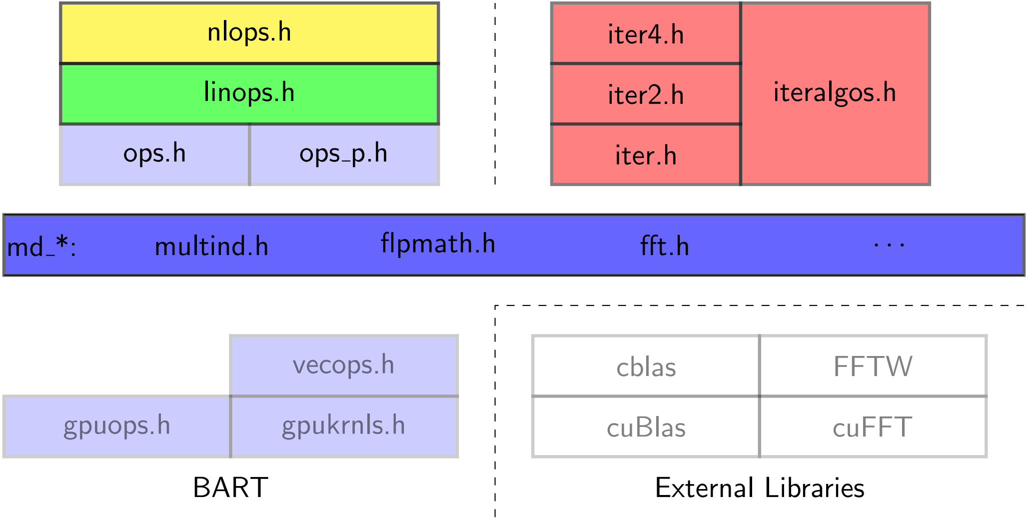
Backend Libraries in BART

Moritz Blumenthal

June 2, 2020

UNIVERSITÄTSMEDIZIN GÖTTINGEN UMG

Overview


nlops.h
linops.h
ops.h
ops_p.h
iter4.h
iter2.h
iter.h
iteralgos.h
md_*:
multind.h
flpmath.h
fft.h
...
vecops.h
gpuops.h
gpukrnls.h
BART
cblas
FFTW
cuBlas
cuFFT
External Libraries

Multi-Dimensional Arrays and Strides - Part I

Memory

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 \end{bmatrix}$$

Strides use element size in bytes!
(Here simplified)

D - dimensional array is described by

- long dims[D] - sizes of dimensions
- long strs[D] - memory access pattern
  - default: column-major order (MATLAB, Fortran,...)

Matrix

	dims	strs
1	3	1
2	2	3

$$A = \begin{pmatrix} a_1 & a_4 \\ a_2 & a_5 \\ a_3 & a_6 \end{pmatrix}$$

Transposed matrix

	dims	strs
1	2	3
2	3	1

$$A^{\mathrm{T}} = \begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \end{pmatrix}$$

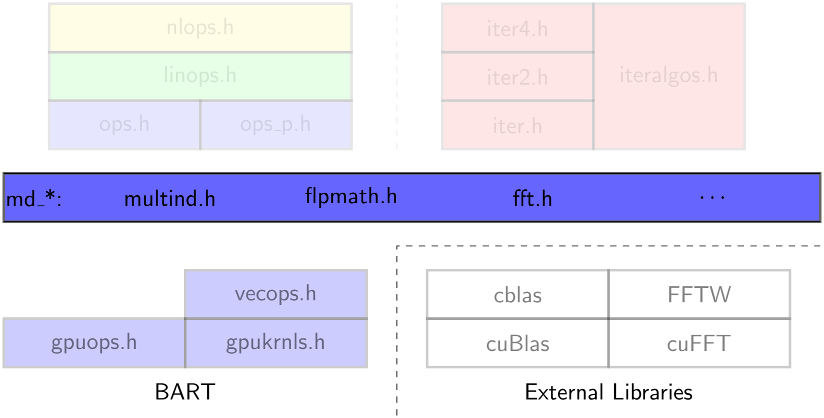
Subvector

	dims	strs
1	2	3

$$B = \begin{pmatrix} a_2 \\ a_5 \end{pmatrix}$$

Offset: 1

md_* Functions



- Consistent interface
- Very flexible due to strides
- Transparent GPU support

md_* Functions - multind.h

Memory allocation

- void* **md_alloc**(int D, long dims[D], size_t size)
- void* **md_alloc_gpu**(int D, long dims[D], size_t size)
- void* **md_alloc_sameplace**(int D, long dims[D], size_t size, void* arg)
- void **md_free**(void* ptr)

Dimension manipulation

- void **md_copy_dims**(int D, long odims[D], long idims[D])
- long* **md_calc_strides**(int D, long str[D], long dim[D], size_t size) (column major)
- long **md_calc_offset**(int D, long str[D], long pos[D])
- void **md_transpose_dims**(int D, int dim1, int dim2, long odims[D], long idims[D])

md_* Functions - multind.h

Copy functions - Two versions (with and without strides)

```
md_copy (int D, long dim[D],                 void* optr,              void* iptr, size_t size);
md_copy2(int D, long dim[D],long ostr[D] ,void* optr,long istr[D],void* iptr, size_t size);
```

- Loops over dimensions and copies each element defined by dims and strs

Copy functions - Usage

- Move data from and to GPU
- Many functions can be derived by setting strides:
  - md_transpose
  - md_flip
  - md_resize
  - md_copy_block
  - md_permute
  - md_fill

md_* Functions - Memory-Mapped I/O (mmio.h)

Properties

- convenient: files accessible in the same way as arrays allocated by md_alloc
- data is loaded/stored as needed

Read arrays

```
int N = 16;
long dims[N];
complex float* in
        = load_cfl("in", N, dims);
...
unmap_cfl(N, dims, in);
```

Store arrays

```
int N = 4;
long dims[4] = {128, 128, 1, 8};
complex float* out
        = create_cfl("out", N, dims);
...
unmap_cfl(N, dims, out);
```

md_* Functions - flpmath.h

Important functions

- md_add, md_sub, md_mul, md_div, md_exp, md_log, ...
- fmac - Fused Multiply-Accumulate: dst += src1 * src2
  - Many functions can be derived using strides: dot, gemv, gemm, convolutions, ...

```
md_fmac (int D, long dim[D],                float* optr,               float* iptr1,               float* iptr2);
md_fmac2(int D, long dim[D],long ostr[D] ,float* optr,long istr1[D],float* iptr1,long istr2[D],float* iptr2);
```

Conventions

- prefix 'z' - complex numbers: zmul vs. mul
- prefix 's' - array-scalar operation: zsmul vs. zmul
- postfix 'c' - complex conjugate: zmulc ($a * \bar{b}$) vs. zmul ($a * b$)
- postfix '2' - strided version

md_* Functions - Strides Part II

Trivial strides

	dims	ostr	istr1	istr2
1	3	1	1	1

$$\mathrm{dst} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$
$$\mathrm{src1} = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix}$$
$$\mathrm{src2} = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}$$

$$\mathrm{dst} = \begin{bmatrix} a_1 b_1 & a_2 b_2 & a_3 b_3 \end{bmatrix}$$

fmac with strides (dst += src1 * src2)

```
for (int i0 = 0; i0 < dim[0]; i0++)
        dst[i0 * ostr[0]]
          += src1[i0 * istr1[0]]
              * src2[i0 * istr2[0]];
```

md_* Functions - Strides Part II

Trivial strides

	dims	ostr	istr1	istr2
1	3	1	1	1

$$\text{dst} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$
$$\text{src1} = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix}$$
$$\text{src2} = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}$$

$$\text{dst} = \begin{bmatrix} a_1 b_1 & a_2 b_2 & a_3 b_3 \end{bmatrix}$$

Dot product

	dims	ostr	istr1	istr2
1	3	0	1	1

$$\text{dst} = \begin{bmatrix} 0 \end{bmatrix}$$
$$\text{src1} = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix}$$
$$\text{src2} = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}$$

$$\text{dst} = \begin{bmatrix} a_1 b_1 + a_2 b_2 + a_3 b_3 \end{bmatrix}$$

fmac with strides (dst += src1 * src2)

```
for (int i0 = 0; i0 < dim[0]; i0++)
        dst[i0 * ostr[0]]
          += src1[i0 * istr1[0]]
              * src2[i0 * istr2[0]];
```

md_* Functions - Strides Part II

Matrix-vector-multiplication

	dims	ostr	istr1	istr2	
1	3	1	1	0	loop over cols
2	2	0	3	1	dot product

$$\mathrm{dst} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$
$$\mathrm{src1} = \begin{bmatrix} a_{11} & a_{21} & a_{12} & a_{22} & a_{13} & a_{23} \end{bmatrix}$$
$$\mathrm{src2} = \begin{bmatrix} b_1 & b_2 \end{bmatrix}$$

$$\mathrm{dst} = \begin{bmatrix} a_{11}b_1 + a_{12}b_2 & a_{21}b_1 + a_{22}b_2 & a_{31}b_1 + a_{32}b_2 \end{bmatrix}$$

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} a_{11}b_1 + a_{12}b_2 \\ a_{21}b_1 + a_{22}b_2 \\ a_{31}b_1 + a_{32}b_2 \end{pmatrix}$$

fmac with strides (dst += src1 * src2)

```
for (int i0 = 0; i0 < dim[0]; i0++)
        for (int i1 = 0; i1 < dim[1]; i1++)
                dst[i0 * ostr[0] + i1 * ostr[1]]
                  += src1[i0 * istr1[0] + i1 * istr1[1]]
                     * src2[i0 * istr2[0] + i1 * istr2[1]];
```

md_* Functions - Strides Part II

Convolution

	dims	ostr	istr1	istr2
1	3	1	1	0
2	2	0	1	-1

$$\mathrm{dst} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

$$\mathrm{src1} = \begin{bmatrix} i_1 & i_2 & i_3 & i_4 \end{bmatrix}$$

$$\mathrm{src2} = \begin{bmatrix} k_1 & k_2 \end{bmatrix}$$

$$\mathrm{dst} = \begin{bmatrix} i_1 k_2 + i_2 k_1 & i_2 k_2 + i_3 k_1 & i_3 k_2 + i_4 k_1 \end{bmatrix}$$

$$\begin{pmatrix} o_1 \\ o_2 \\ o_2 \end{pmatrix} = \begin{pmatrix} i_1 k_2 + i_2 k_1 \\ i_2 k_2 + i_3 k_1 \\ i_3 k_2 + i_4 k_1 \end{pmatrix} = \begin{pmatrix} i_1 & i_2 \\ i_2 & i_3 \\ i_3 & i_4 \end{pmatrix} \begin{pmatrix} k_2 \\ k_1 \end{pmatrix}$$

fmac with strides (dst += src1 * src2)

```
for (int i0 = 0; i0 < dim[0]; i0++)
        for (int i1 = 0; i1 < dim[1]; i1++)
                dst[i0 * ostr[0] + i1 * ostr[1]]
                        += src1[i0 * istr1[0] + i1 * istr1[1]]
                           * src2[i0 * istr2[0] + i1 * istr2[1]];
```

md_* Functions - Strides Part II

Convolution

	dims	ostr	istr1	istr2
1	3	1	1	0
2	2	0	1	-1

$$\mathrm{dst} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$
$$\mathrm{src1} = \begin{bmatrix} i_1 & i_2 & i_3 & i_4 \end{bmatrix}$$
$$\mathrm{src2} = \begin{bmatrix} k_1 & k_2 \end{bmatrix}$$

$$\mathrm{dst} = \begin{bmatrix} i_1 k_2 + i_2 k_1 & i_2 k_2 + i_3 k_1 & i_3 k_2 + i_4 k_1 \end{bmatrix}$$

$$\begin{pmatrix} o_1 \\ o_2 \\ o_2 \end{pmatrix} = \begin{pmatrix} i_1 k_2 + i_2 k_1 \\ i_2 k_2 + i_3 k_1 \\ i_3 k_2 + i_4 k_1 \end{pmatrix} = \begin{pmatrix} i_1 & i_2 \\ i_2 & i_3 \\ i_3 & i_4 \end{pmatrix} \begin{pmatrix} k_2 \\ k_1 \end{pmatrix}$$

Strides are very powerful

- Very flexible
- Memory efficient

fmac with strides (dst += src1 * src2)

```
for (int i0 = 0; i0 < dim[0]; i0++)
        for (int i1 = 0; i1 < dim[1]; i1++)
                dst[i0 * ostr[0] + i1 * ostr[1]]
                        += src1[i0 * istr1[0] + i1 * istr1[1]]
                           * src2[i0 * istr2[0] + i1 * istr2[1]];
```

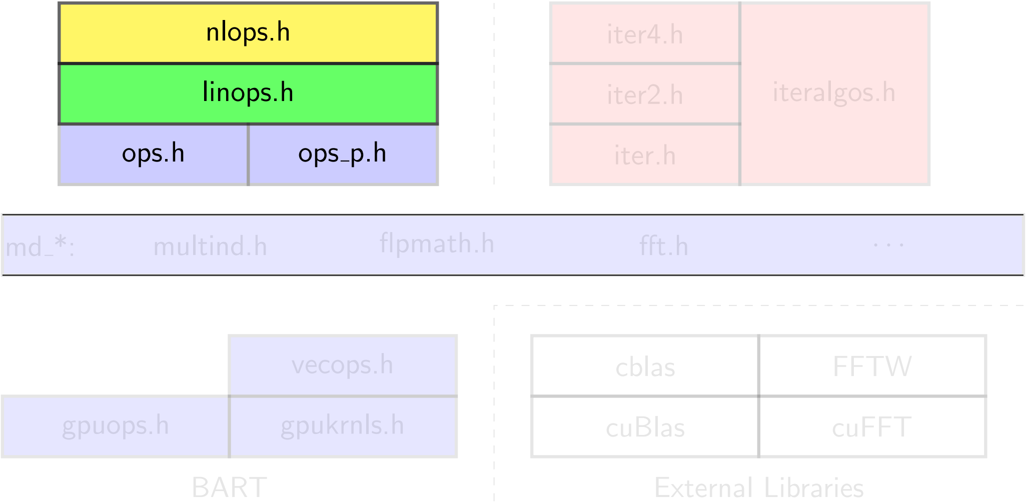
md_* Functions - fft.h

FFT

```
fft(int D, long dims[D], unsigned long flags, complex float* dst,
complex float* src)
fft2(int D, long dims[D], unsigned long flags, long ostr[D], complex
float* dst, long istr[D], complex float* src)
```

- Bitmask for selecting dims: $\text{flags} = \ldots 00010_2$ - select 2nd dimension
- FFTW/cuFFT backend
- fftmod for centered version: fftc=fftmod→fft→fftmod

Operator Frameworks



Linops

- Set of operators: forward, adjoint, normal, norm_inv
- Can be applied with:

```
linop_forward(  struct linop_s* op,
                int DN, long ddims[DN], complex float* dst,
                int SN, long sdims[SN], complex float* src)
```

- Can be chained and added:

```
struct linop_s* linop_chain(struct linop_s* a, struct linop_s* b)
struct linop_s* linop_plus(struct linop_s* a, struct linop_s* b)
```

- Many linear operators available:
  - src/linop/*, e.g.: linop_cdiag_create, linop_transpose_create, linop_fft_create, linop_conv_create, linop_grad_create, linop_sampling_create
  - src/noncart/nufft, e.g.: nufft_create
  - ...

Linops Example: Linop for Unitary FFT

Construct unitary FFT from building blocks:

```
struct linop_s* linop_fftu_create(int N, long dims[N], unsigned int flags)
{       //compute scaling factor
        long fft_dims[N];
        md_select_dims(N, flags, fft_dims, dims);
        complex float scale[1] = { 1. / sqrtf(md_calc_size(N, fft_dims))};
        //create linops for scaling and FFT
        auto lop_fft = linop_fft_create(N, dims, flags);
        auto lop_scale = linop_cdiag_create(N, dims, 0l, scale);
        //return chained operator
        return linop_chain_FF(lop_fft, lop_scale);
}
```

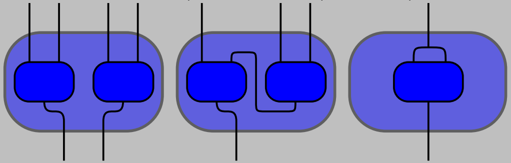
Use the linop:

```
long N = 2; long dims[2] = {128, 128}; usingened int flags = 3; //2^0 + 2^1
auto lop_fftu = linop_fftu_create(N, dims, flags);
// linop_forward/linop_adjoint/linop_normal
linop_adjoint(lop_fftu, N, dims, dst, N, dims, src);
```

Nlops - Work in Progress

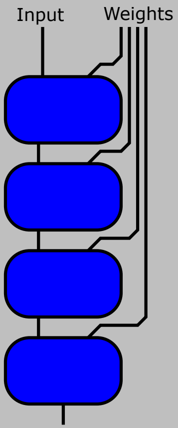
Nonlinear operators

- can have multiple in-/outputs
- support automatic differentiation:
- can be chained, combined, linked, ...

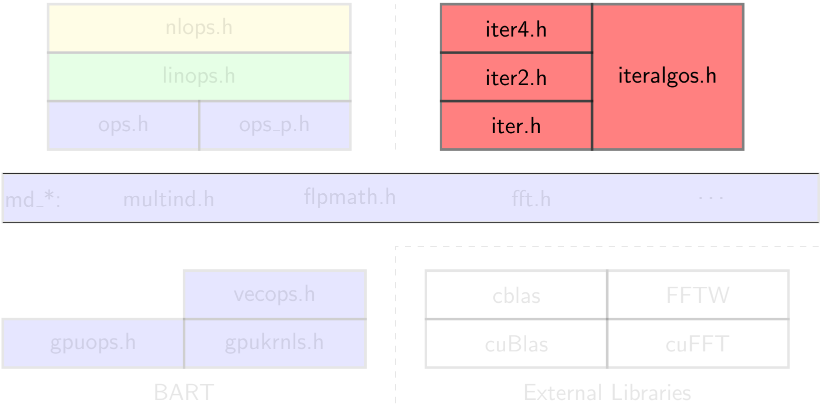


- used for non-linear reconstruction (nlinv, model-based reconstructions)

Upcoming: Neural networks



Iterative Algorithms



- iter* - interfaces for different algorithms
- iteralgos - implementation of algorithms

Iterative Algorithms

iter2

$$x = \arg\min_x \|Ax - y\|_2^2 + \sum_i f_i(B_i x)$$

- Least-squares part: $A^H A x = A^H y$
- Regularization: $\mathrm{prox}_f(x) = \arg\min_x f(x) + \|x - y\|_2^2$
-
```
typedef void (italgo_fun2_f)(iter_conf* conf,
        struct operator_s* normaleq, ... ,
        struct operator_p_s* prox[D], struct linop_s* ops[D],
        ... , float* image, float* imageadj, ...)
```
- Cosistent interface for many algorithms: `iter2_conjgrad`, `iter2_fista`, `iter2_chambolle_pock`, `iter2_admm`, ...

Iterative Algorithms - An Example(rof.c)

TV-Denoising

$$x = \arg\min_{x} \|\mathbb{1}x - y\|_2^2 + \lambda \|\mathrm{TV}x\|_1$$

```
//prepare in-/outputs and parameters
long dims[DIMS];
complex float* in = load_cfl("in", DIMS, dims);
complex float* out = create_cfl("out", DIMS, dims);
float lambda = 0.1; int flags = 6;
//create operators
auto id   = linop_identity_create(DIMS, dims);
auto grad = linop_grad_create(DIMS, dims, DIMS, flags);
auto thresh_prox = prox_thresh_create(DIMS + 1, linop_codomain(grad)->dims,
                                      lambda, MD_BIT(DIMS));
//run iterative algorithm
struct iter_admm_conf conf = iter_admm_defaults;
iter2_admm(     CAST_UP(&conf), id->normal,
                1, MAKE_ARRAY(thresh_prox), MAKE_ARRAY(grad), NULL, NULL,
                2 * md_calc_size(DIMS, dims), (float*)out, (const float*)in,
                NULL);
```

Summary - Backend Libraries in BART