MAT MANUAL

Contents

[in package MGL-MAT]

- **[system]** **"mgl-mat"**

```
- _Version:_ 0.1.0
- _Description:_ [`mat`][6d14] is library for working with multi-dimensional
    arrays which supports efficient interfacing to foreign and CUDA
    code. BLAS and CUBLAS bindings are available.
- _Licence:_ MIT, see COPYING.
- _Author:_ Gábor Melis <mega@retes.hu>
- _Mailto:_ [mega@retes.hu](mailto:mega@retes.hu)
- _Homepage:_ <http://melisgl.github.io/mgl-mat>
- _Bug tracker:_ <https://github.com/melisgl/mgl-mat/issues>
- _Source control:_ [GIT](https://github.com/melisgl/mgl-mat.git)
- *Depends on:* alexandria, bordeaux-threads, cffi, cffi-grovel, cl-cuda,
↪  flexi-streams, ieee-floats, lla, [mgl-pax][6fdb], num-utils, static-vectors,
↪  trivial-garbage
```

1 Links

Here is the official repository and the HTML documentation for the latest version.

2 Introduction

2.1 What's MGL-MAT?

MGL-MAT is library for working with multi-dimensional arrays which supports efficient interfacing to foreign and CUDA code with automatic translations between cuda, foreign and lisp storage. BLAS and CUBLAS bindings are available.

2.2 What kind of matrices are supported?

Currently only row-major single and double float matrices are supported, but it would be easy to add single and double precision complex types too. Other numeric types, such as byte and native integer, can be added too, but they are not supported by CUBLAS. There are no restrictions on the number of dimensions, and reshaping is possible. All functions operate on the visible portion of the matrix (which is subject to displacement and shaping), invisible elements are not affected.

2.3 Where to Get it?

All dependencies are in quicklisp except for CL-CUDA that needs to be fetched from github. Just clone CL-CUDA and MGL-MAT into `quicklisp/local-projects/` and you are set. MGL-MAT itself lives at github, too.

Prettier-than-markdown HTML documentation cross-linked with other libraries is available as part of PAX World.

3 Tutorial

We are going to see how to create matrices, access their contents.

Creating matrices is just like creating lisp arrays:

```
(make-mat '6)
==> #<MAT 6 A #(0.0d0 0.0d0 0.0d0 0.0d0 0.0d0 0.0d0)>

(make-mat '(2 3) :ctype :float :initial-contents '((1 2 3) (4 5 6)))
==> #<MAT 2x3 AB #2A((1.0 2.0 3.0) (4.0 5.0 6.0))>

(make-mat '(2 3 4) :initial-element 1)
==> #<MAT 2x3x4 A #3A(((1.0d0 1.0d0 1.0d0 1.0d0)
-->                   (1.0d0 1.0d0 1.0d0 1.0d0)
-->                   (1.0d0 1.0d0 1.0d0 1.0d0))
-->                  ((1.0d0 1.0d0 1.0d0 1.0d0)
-->                   (1.0d0 1.0d0 1.0d0 1.0d0)
-->                   (1.0d0 1.0d0 1.0d0 1.0d0)))>
```

The most prominent difference from lisp arrays is that `mat`s are always numeric and their element type (called `ctype` here) defaults to `:double`.

Individual elements can be accessed or set with `mref`:

```
(let ((m (make-mat '(2 3))))
  (setf (mref m 0 0) 1)
  (setf (mref m 0 1) (* 2 (mref m 0 0)))
  (incf (mref m 0 2) 4)
  m)
==> #<MAT 2x3 AB #2A((1.0d0 2.0d0 4.0d0) (0.0d0 0.0d0 0.0d0))>
```

Compared to `aref` `mref` is a very expensive operation and it's best used sparingly. Instead, typical code relies much more on matrix level operations:

```
(princ (scal! 2 (fill! 3 (make-mat 4))))
.. #<MAT 4 BF #(6.0d0 6.0d0 6.0d0 6.0d0)>
==> #<MAT 4 ABF #(6.0d0 6.0d0 6.0d0 6.0d0)>
```

The content of a matrix can be accessed in various representations called *facets*. MGL-MAT takes care of synchronizing these facets by copying data around lazily, but doing its best to share storage for facets that allow it.

Notice the `abf` in the printed results. It illustrates that behind the scenes `fill!` worked on the `backing-array` facet (hence the `b`) that's basically a 1d lisp array. `scal!` on the other hand made a foreign call to the BLAS `dscal` function for which it needed the `foreign-array` facet (`f`). Finally, the `a` stands for the `array` facet that was created when the array was printed. All facets are up-to-date (else some of the characters would be lowercase). This is possible because these three facets actually share storage which is never the case for the `cuda-array` facet. Now if we have a CUDA-capable GPU, CUDA can be enabled with `with-cuda*`:

```
(with-cuda* ()
  (princ (scal! 2 (fill! 3 (make-mat 4)))))
.. #<MAT 4 C #(6.0d0 6.0d0 6.0d0 6.0d0)>
==> #<MAT 4 A #(6.0d0 6.0d0 6.0d0 6.0d0)>
```

Note the lonely `c` showing that only the `cuda-array` facet was used for both `fill!` and `scal!`. When `with-cuda*` exits and destroys the CUDA context, it destroys all CUDA facets, moving their data to the `array` facet, so the returned `mat` only has that facet.

When there is no high-level operation that does what we want, we may need to add new operations. This is usually best accomplished by accessing one of the facets directly, as in the following example:

```
(defun logdet (mat)
  "Logarithm of the determinant of MAT. Return -1, 1 or 0 (or
  equivalent) to correct for the sign, as the second value."
  (with-facets ((array (mat 'array :direction :input)))
    (lla:logdet array)))
```

Notice that `logdet` doesn't know about CUDA at all. `with-facets` gives it the content of the matrix as a normal multidimensional lisp array, copying the data from the GPU or elsewhere if necessary. This allows new representations (`facet`s) to be added easily and it also avoids copying if the facet is already up-to-date. Of course, adding CUDA support to `logdet` could make it more efficient.

Adding support for matrices that, for instance, live on a remote machine is thus possible with a new facet type and existing code would continue to work (albeit possibly slowly). Then one could optimize the bottleneck operations by sending commands over the network instead of copying data.

It is a bad idea to conflate resource management policy and algorithms. MGL-MAT does its best to keep them separate.

4 Basics

- **[class]** **mat** *cube*

  A `mat` is a data `cube` that is much like a lisp array, it supports `displacement`, arbitrary `dimensions` and `initial-element` with the usual semantics. However, a `mat` supports different representations of the same data. See Tutorial for an introduction.

- **[reader]** **mat-ctype** *mat (:ctype = *default-mat-ctype*)*

  One of `*supported-ctypes*`. The matrix can hold only values of this type.

- **[reader]** **mat-displacement** *mat (:displacement = 0)*

  A value in the `[0,max-size]` interval. This is like the DISPLACED-INDEX-OFFSET of a lisp array, but displacement is relative to the start of the underlying storage vector.

- **[reader]** **mat-dimensions** *mat (:dimensions)*

  Like `array-dimensions`. It holds a list of dimensions, but it is allowed to pass in scalars too.

- **[function]** **mat-dimension** *mat axis-number*

  Return the dimension along `axis-number`. Similar to `array-dimension`.

- **[reader]** **mat-initial-element** *mat (:initial-element = 0)*

  If non-nil, then when a facet is created, it is filled with `initial-element` coerced to the appropriate numeric type. If `nil`, then no initialization is performed.

- **[reader]** **mat-size** *mat*

  The number of elements in the visible portion of the array. This is always the product of the elements `mat-dimensions` and is similar to `array-total-size`.

- **[reader]** **mat-max-size** *mat (:max-size)*

  The number of elements for which storage may be allocated. This is `displacement` + `mat-size` + `slack` where `slack` is the number of trailing invisible elements.

- **[function]** **make-mat** *dimensions &rest args &key (ctype *default-mat-ctype*) (displacement 0) max-size initial-element initial-contents (synchronization *default-synchronization*) displaced-to (cuda-enabled *default-mat-cuda-enabled*)*

  Return a new `mat` object. If `initial-contents` is given then the matrix contents are initialized with `replace!`. See class `mat` for the description of the rest of the parameters. This is exactly what (`make-instance` 'mat ...) does except `dimensions` is not a keyword argument so that `make-mat` looks more like `make-array`. The semantics of `synchronization` are desribed in the Synchronization section.

  If specified, `displaced-to` must be a `mat` object large enough (in the sense of its `mat-size`), to hold `displacement` plus `(reduce #'* dimensions)` elements. Just like with `make-array`, `initial-element` and `initial-contents` must not be supplied together with `displaced-to`. See Shaping for more.

- **[function]** **array-to-mat** *array &key ctype (synchronization *default-synchronization*)*

  Create a `mat` that's equivalent to `array`. Displacement of the created array will be 0 and the size will be equal to `array-total-size`. If `ctype` is non-nil, then it will be the ctype of the new matrix. Else `array`'s type is converted to a ctype. If there is no corresponding ctype, then `*default-mat-ctype*` is used. Elements of `array` are coerced to `ctype`.

  Also see Synchronization.

- **[function]** **mat-to-array** *mat*

- **[function]** **replace!** *mat seq-of-seqs*

  Replace the contents of `mat` with the elements of `seq-of-seqs`. `seq-of-seqs` is a nested sequence of sequences similar to the `initial-contents` argument of `make-array`. The total number of elements must match the size of `mat`. Returns `mat`.

  `seq-of-seqs` may contain multi-dimensional arrays as *leafs*, so the following is legal:

```
(replace! (make-mat '(1 2 3)) '(#2A((1 2 3) (4 5 6))))
==> #<MAT 1x2x3 AB #3A(((1.0d0 2.0d0 3.0d0) (4.0d0 5.0d0 6.0d0)))>
```

- **[function]** **mref** *mat &rest indices*

  Like `aref` for arrays. Don't use this if you care about performance at all. `setf`able. When set, the value is coerced to the ctype of `mat` with `coerce-to-ctype`. Note that currently `mref` always operates on the `backing-array` facet so it can trigger copying of facets. When it's `setf`'ed, however, it will update the `cuda-array` if cuda is enabled and it is up-to-date or there are no facets at all.

- **[function]** **row-major-mref** *mat index*

  Like `row-major-aref` for arrays. Don't use this if you care about performance at all. `setf`able. When set, the value is coerced to the ctype of `mat` with `coerce-to-ctype`. Note that currently `row-major-mref` always operates on the `backing-array` facet so it can trigger copying of facets. When it's `setf`'ed, however, it will update the `cuda-array` if cuda is enabled and it is up-to-date or there are no facets at all.

- **[function]** **mat-row-major-index** *mat &rest subscripts*

  Like `array-row-major-index` for arrays.

5 Element types

- **[variable]** ***supported-ctypes*** *(:float :double)*

- **[type]** **ctype**

  This is basically `(member :float :double)`.

- **[variable]** ***default-mat-ctype*** *:double*

  By default `mat`s are created with this ctype. One of `:float` or `:double`.

- **[function]** **coerce-to-ctype** *x &key (ctype *default-mat-ctype*)*

  Coerce the scalar `x` to the lisp type corresponding to `ctype`.

6 Printing

- **[variable]** ***print-mat*** *t*

  Controls whether the contents of a `mat` object are printed as an array (subject to the standard printer control variables).

- **[variable]** ***print-mat-facets*** *t*

  Controls whether a summary of existing and up-to-date facets is printed when a `mat` object is printed. The summary that looks like `ABcfh` indicates that all five facets (`array`, `backing-array`, `cuda-array`, `foreign-array`, `cuda-host-array`) are present and the first two are up-to-date. A summary of a single #- indicates that there are no facets.

7 Shaping

We are going to discuss various ways to change the visible portion and dimensions of matrices. Conceptually a matrix has an *underlying non-displaced storage vector*. For `(make-mat 10 :displacement 7 :max-size 21)` this underlying vector looks like this:

```
displacement | visible elements  | slack
. . . . . . . 0 0 0 0 0 0 0 0 0 0 . . . .
```

Whenever a matrix is reshaped (or *displaced to* in lisp terminology), its displacement and dimensions change but the underlying vector does not.

The rules for accessing displaced matrices is the same as always: multiple readers can run in parallel, but attempts to write will result in an error if there are either readers or writers on any of the matrices that share the same underlying vector.

7.1 Comparison to Lisp Arrays

One way to reshape and displace `mat` objects is with `make-mat` and its `displaced-to` argument whose semantics are similar to that of `make-array` in that the displacement is *relative* to the displacement of `displaced-to`.

```
(let* ((base (make-mat 10 :initial-element 5 :displacement 1))
       (mat (make-mat 6 :displaced-to base :displacement 2)))
  (fill! 1 mat)
  (values base mat))
==> #<MAT 1+10+0 A #(5.0d0 5.0d0 1.0d0 1.0d0 1.0d0 1.0d0 1.0d0 1.0d0 5.0d0
-->                 5.0d0)>
==> #<MAT 3+6+2 AB #(1.0d0 1.0d0 1.0d0 1.0d0 1.0d0 1.0d0)>
```

There are important semantic differences compared to lisp arrays all which follow from the fact that displacement operates on the underlying conceptual non-displaced vector.

- Matrices can be displaced and have slack even without `displaced-to` just like `base` in the above example.
- It's legal to alias invisible elements of `displaced-to` as long as the new matrix fits into the underlying storage.
- Negative displacements are allowed with `displaced-to` as long as the adjusted displacement is non-negative.
- Further shaping operations can make invisible portions of the `displaced-to` matrix visible by changing the displacement.
- In contrast to `array-displacement`, `mat-displacement` only returns an offset into the underlying storage vector.

7.2 Functional Shaping

The following functions are collectively called the functional shaping operations, since they don't alter their arguments in any way. Still, since storage is aliased modification to the returned matrix will affect the original.

- **[function]** **reshape-and-displace** *mat dimensions displacement*

  Return a new matrix of `dimensions` that aliases `mat`'s storage at offset `displacement`. `displacement` 0 is equivalent to the start of the storage of `mat` regardless of `mat`'s displacement.

- **[function]** **reshape** *mat dimensions*

  Return a new matrix of `dimensions` whose displacement is the same as the displacement of `mat`.

- **[function]** **displace** *mat displacement*

  Return a new matrix that aliases `mat`'s storage at offset `displacement`. `displacement` 0 is equivalent to the start of the storage of `mat` regardless of `mat`'s displacement. The returned matrix has the same dimensions as `mat`.

7.3 Destructive Shaping

The following destructive operations don't alter the contents of the matrix, but change what is visible. `adjust!` is the odd one out, it may create a new `mat`.

- **[function]** **reshape-and-displace!** *mat dimensions displacement*

  Change the visible (or active) portion of `mat` by altering its displacement offset and dimensions. Future operations will only affect this visible portion as if the rest of the elements were not there. Return `mat`.

  `displacement` + the new size must not exceed `mat-max-size`. Furthermore, there must be no facets being viewed (with `with-facets`) when calling this function as the identity of the facets is not stable.

- **[function]** **reshape!** *mat dimensions*

  Like `reshape-and-displace!` but only alters the dimensions.

- **[function]** **displace!** *mat displacement*

  Like `reshape-and-displace!` but only alters the displacement.

- **[function]** **reshape-to-row-matrix!** *mat row*

  Reshape the 2d `mat` to make only a single `row` visible. This is made possible by the row-major layout, hence no column counterpart. Return `mat`.

- **[macro]** **with-shape-and-displacement** *(mat &optional (dimensions nil) (displacement nil)) &body body*

  Reshape and displace `mat` if `dimensions` and/or `displacement` is given and restore the original shape and displacement after `body` is executed. If neither is specificed, then nothing will be changed, but `body` is still allowed to alter the shape and displacement.

- **[function]** **adjust!** *mat dimensions displacement &key (destroy-old-p t)*

  Like `reshape-and-displace!` but creates a new matrix if `mat` isn't large enough. If a new matrix is created, the contents are not copied over and the old matrix is destroyed with `destroy-cube` if `destroy-old-p`.

8 Assembling

The functions here assemble a single `mat` from a number of `mat`s.

- **[function]** **stack!** *axis mats mat*

  Stack `mats` along `axis` into `mat` and return `mat`. If `axis` is 0, place `mats` into `mat` below each other starting from the top. If `axis` is 1, place `mats` side by side starting from the left. Higher `axis` are also supported. All dimensions except for `axis` must be the same for all `mats`.

- **[function]** **stack** *axis mats &key (ctype *default-mat-ctype*)*

  Like `stack!` but return a new `mat` of `ctype`.

```
(stack 1 (list (make-mat '(3 2) :initial-element 0)
               (make-mat '(3 1) :initial-element 1)))
==> #<MAT 3x3 B #2A((0.0d0 0.0d0 1.0d0)
-->                 (0.0d0 0.0d0 1.0d0)
-->                 (0.0d0 0.0d0 1.0d0))>
```

9 Caching

Allocating and initializing a `mat` object and its necessary facets can be expensive. The following macros remember the previous value of a binding in the same thread and /place/. Only weak references are constructed so the cached objects can be garbage collected.

While the cache is global, thread safety is guaranteed by having separate subcaches per thread. Each subcache is keyed by a /place/ object that's either explicitly specified or else is unique to each invocation of the caching macro, so different occurrences of caching macros in the source never share data. Still, recursion could lead to data sharing between different invocations of the same function. To prevent this, the cached object is removed from the cache while it is used so other invocations will create a fresh one which isn't particularly efficient but at least it's safe.

- **[macro]** **with-thread-cached-mat** *(var dimensions &rest args &key (place :scratch) (ctype '*default-mat-ctype*) (displacement 0) max-size (initial-element 0) initial-contents) &body body*

  Bind `var` to a matrix of `dimensions`, `ctype`, etc. Cache this matrix, and possibly reuse it later by reshaping it. When `body` exits the cached object is updated with the binding of `var` which `body` may change.

  There is a separate cache for each thread and each `place` (under `eq`). Since every cache holds exactly one `mat` per `ctype`, nested `with-thread-cached-mat` often want to use different `places`. By convention, these places are called `:scratch-1`, `:scratch-2`, etc.

- **[macro]** **with-thread-cached-mats** *specs &body body*

  A shorthand for writing nested `with-thread-cached-mat` calls.

  ```
  (with-thread-cached-mat (a ...)
    (with-thread-cached-mat (b ...)
      ...))
  ```

  is equivalent to:

  ```
  (with-thread-cached-mat ((a ...)
                           (b ...))
    ...)
  ```

- **[macro]** **with-ones** *(var dimensions &key (ctype '*default-mat-ctype*) (place :ones)) &body body*

  Bind `var` to a matrix of `dimensions` whose every element is 1. The matrix is cached for efficiency.

10 BLAS Operations

Only some BLAS functions are implemented, but it should be easy to add more as needed. All of them default to using CUDA, if it is initialized and enabled (see `use-cuda-p`).

Level 1 BLAS operations

- **[function]** **asum** *x &key (n (mat-size x)) (incx 1)*

  Return the l1 norm of `x`, that is, sum of the absolute values of its elements.

- **[function]** **axpy!** *alpha x y &key (n (mat-size x)) (incx 1) (incy 1)*

  Set `y` to `alpha` * `x` + `y`. Return `y`.

- **[function]** **copy!** *x y &key (n (mat-size x)) (incx 1) (incy 1)*

  Copy `x` into `y`. Return `y`.

- **[function]** **dot** *x y &key (n (mat-size x)) (incx 1) (incy 1)*

  Return the dot product of `x` and `y`.

- **[function]** **nrm2** *x &key (n (mat-size x)) (incx 1)*

  Return the l2 norm of `x`, which is the square root of the sum of the squares of its elements.

- **[function]** **scal!** *alpha x &key (n (mat-size x)) (incx 1)*

  Set `x` to `alpha` * `x`. Return `x`.

Level 3 BLAS operations

- **[function]** **gemm!** *alpha a b beta c &key transpose-a? transpose-b? m n k lda ldb ldc*

  Basically `c` = `alpha` * `a'` * `b'` + `beta` * `c`. `a'` is `a` or its transpose depending on `transpose-a?`. `b'` is `b` or its transpose depending on `transpose-b?`. Returns `c`.

  `a'` is an MxK matrix. `b'` is a KxN matrix. `c` is an MxN matrix.

  `lda` is the width of the matrix `a` (not of `a'`). If `a` is not transposed, then `k` <= `lda`, if it's transposed then `m` <= `lda`.

  `ldb` is the width of the matrix `b` (not of `b'`). If `b` is not transposed, then `n` <= `ldb`, if it's transposed then `k` <= `ldb`.

  In the example below M=3, N=2, K=5, LDA=6, LDB=3, LDC=4. The cells marked with + do not feature in the calculation.

```
            N
          --+
          --+
        K -B+
          --+
          --+
          +++
    K
  -----+  --++
M --A--+  -C++
  -----+  --++
  ++++++  ++++
```

11 Destructive API

- **[function]** **.square!** *x &key (n (mat-size x))*

  Set `x` to its elementwise square. Return `x`.

- **[function]** **.sqrt!** *x &key (n (mat-size x))*

Set x to its elementwise square root. Return x.

- **[function]** **.log!** *x &key (n (mat-size x))*

  Set x to its elementwise natural logarithm. Return x.

- **[function]** **.exp!** *x &key (n (mat-size x))*

  Apply exp elementwise to x in a destructive manner. Return x.

- **[function]** **.expt!** *x power*

  Raise matrix x to power in an elementwise manner. Return x. Note that CUDA and non-CUDA implementations may disagree on the treatment of NaNs, infinities and complex results. In particular, the lisp implementation always computes the realpart of the results while CUDA's pow() returns NaNs instead of complex numbers.

- **[function]** **.inv!** *x &key (n (mat-size x))*

  Set x to its elementwise inverse (/ 1 x). Return x.

- **[function]** **.logistic!** *x &key (n (mat-size x))*

  Destructively apply the logistic function to x in an elementwise manner. Return x.

- **[function]** **.+!** *alpha x*

  Add the scalar alpha to each element of x destructively modifying x. Return x.

- **[function]** **.*!** *x y*

- **[function]** **geem!** *alpha a b beta c*

  Like gemm!, but multiplication is elementwise. This is not a standard BLAS routine.

- **[function]** **geerv!** *alpha a x beta b*

  GEneric Elementwise Row - Vector multiplication. B = beta * B + alpha a .* X* where x* is a matrix of the same shape as a whose every row is x. Perform element-wise multiplication on each row of a with the vector x and add the scaled result to the corresponding row of b. Return b. This is not a standard BLAS routine.

- **[function]** **.<!** *x y*

  For each element of x and y set y to 1 if the element in y is greater than the element in x, and to 0 otherwise. Return y.

- **[function]** **.min!** *alpha x*

  Set each element of x to alpha if it's greater than alpha. Return x.

- **[function]** **.max!** *alpha x*

  Set each element of x to alpha if it's less than alpha. Return x.

- **[function]** **add-sign!** *alpha a beta b*

Add the elementwise sign (-1, 0 or 1 for negative, zero and positive numbers respectively) of `a` times `alpha` to `beta` * `b`. Return `b`.

- **[function]** **fill!** *alpha x &key (n (mat-size x))*

  Fill matrix `x` with `alpha`. Return `x`.

- **[function]** **sum!** *x y &key axis (alpha 1) (beta 0)*

  Sum matrix `x` along `axis` and add `alpha` * `sum`s to `beta` * `y` destructively modifying `y`. Return `y`. On a 2d matrix (nothing else is supported currently), if `axis` is 0, then columns are summed, if `axis` is 1 then rows are summed.

- **[function]** **scale-rows!** *scales a &key (result a)*

  Set `result` to `diag(scales)*a` and return it. `a` is an `MxN` matrix, `scales` is treated as a length `m` vector.

- **[function]** **scale-columns!** *scales a &key (result a)*

  Set `result` to `a*diag(scales)` and return it. `a` is an `MxN` matrix, `scales` is treated as a length `n` vector.

- **[function]** **.sin!** *x &key (n (mat-size x))*

  Apply `sin` elementwise to `x` in a destructive manner. Return `x`.

- **[function]** **.cos!** *x &key (n (mat-size x))*

  Apply `cos` elementwise to `x` in a destructive manner. Return `x`.

- **[function]** **.tan!** *x &key (n (mat-size x))*

  Apply `tan` elementwise to `x` in a destructive manner. Return `x`.

- **[function]** **.sinh!** *x &key (n (mat-size x))*

  Apply `sinh` elementwise to `x` in a destructive manner. Return `x`.

- **[function]** **.cosh!** *x &key (n (mat-size x))*

  Apply `cosh` elementwise to `x` in a destructive manner. Return `x`.

- **[function]** **.tanh!** *x &key (n (mat-size x))*

  Apply `tanh` elementwise to `x` in a destructive manner. Return `x`.

Finally, some neural network operations.

- **[function]** **convolve!** *x w y &key start stride anchor batched*

  `y` = `y` + conv(`x`, `w`) and return `y`. If `batched`, then the first dimension of `x` and `y` is the number of elements in the batch (B), else B is assumed to be 1. The rest of the dimensions encode the input (`x`) and output (`y`) N dimensional feature maps. `start`, `stride` and `anchor` are lists of length N. `start` is the multi-dimensional index of the first element of the input feature map (for each element in the batch) for which the convolution must be computed. Then (`elt stride` (- N 1)) is added to the last element of `start` and so on until

(array-dimension x 1) is reached. Then the last element of start is reset, (elt stride (- N 2)) is added to the first but last element of start and we scan the last dimension again. Take a 2d example, start is (0 0), stride is (1 2), and x is a B*2x7 matrix.

w is:

```
1 2 1
2 4 2
1 2 1
```

and anchor is (1 1) which refers to the element of w whose value is

4. This anchor point of w is placed over elements of x whose multi dimensional index is in numbers in this figure (only one element in the batch is shown):

   0,0 . 0,2 . 0,4 . 0,6 1,0 . 1,2 . 1,4 . 1,6

When applying w at position P of x, the convolution is the sum of the products of overlapping elements of x and w when w's anchor is placed at P. Elements of w over the edges of x are multiplied with 0 so are effectively ignored. The order of application of w to positions defined by start, stride and anchor is undefined.

y must be a B*2x4 (or 2x4 if not batched) matrix in this example, just large enough to hold the results of the convolutions.

- **[function]** **derive-convolve!** *x xd w wd yd &key start stride anchor batched*

  Add the dF/dX to xd and and dF/dW to wd where yd is dF/dY for some function F where Y is the result of convolution with the same arguments.

- **[function]** **max-pool!** *x y &key start stride anchor batched pool-dimensions*

- **[function]** **derive-max-pool!** *x xd y yd &key start stride anchor batched pool-dimensions*

  Add the dF/dX to xd and and dF/dW to WD where yd is dF/dY for some function F where y is the result of max-pool! with the same arguments.

12 Non-destructive API

- **[function]** **copy-mat** *a*

  Return a copy of the active portion with regards to displacement and shape of a.

- **[function]** **copy-row** *a row*

  Return row of a as a new 1d matrix.

- **[function]** **copy-column** *a column*

  Return column of a as a new 1d matrix.

- **[function]** **mat-as-scalar** *a*

  Return the first element of a. a must be of size 1.

- **[function]** **scalar-as-mat** *x &key (ctype (lisp->ctype (type-of x)))*

  Return a matrix of one dimension and one element: `x`. `ctype`, the type of the matrix, defaults to the ctype corresponding to the type of `x`.

- **[function]** **m=** *a b*

  Check whether `a` and `b`, which must be matrices of the same size, are elementwise equal.

- **[function]** **transpose** *a*

  Return the transpose of `a`.

- **[function]** **m*** *a b &key transpose-a? transpose-b?*

  Compute op(`a`) * op(`b`). Where op is either the identity or the transpose operation depending on `transpose-a?` and `transpose-b?`.

- **[function]** **mm*** *m &rest args*

  Convenience function to multiply several matrices.

  (mm* a b c) => a * b * c

- **[function]** **m-** *a b*

  Return `a` - `b`.

- **[function]** **m+** *a b*

  Return `a` + `b`.

- **[function]** **invert** *a*

  Return the inverse of `a`.

- **[function]** **logdet** *mat*

  Logarithm of the determinant of `mat`. Return -1, 1 or 0 (or equivalent) to correct for the sign, as the second value.

13 Mappings

- **[function]** **map-concat** *fn mats mat &key key pass-raw-p*

  Call `fn` with each element of `mats` and `mat` temporarily reshaped to the dimensions of the current element of `mats` and return `mat`. For the next element the displacement is increased so that there is no overlap.

  `mats` is keyed by `key` just like the CL sequence functions. Normally, `fn` is called with the matrix returned by `key`. However, if `pass-raw-p`, then the matrix returned by `key` is only used to calculate dimensions and the element of `mats` that was passed to `key` is passed to `fn`, too.

```
(map-concat #'copy! (list (make-mat 2) (make-mat 4 :initial-element 1))
            (make-mat '(2 3)))
==> #<MAT 2x3 AB #2A((0.0d0 0.0d0 1.0d0) (1.0d0 1.0d0 1.0d0))>
```

- **[function]** **map-displacements** *fn mat dimensions &key (displacement-start 0) displacement-step*

  Call `fn` with `mat` reshaped to `dimensions`, first displaced by `displacement-start` that's incremented by `displacement-step` each iteration while there are enough elements left for `dimensions` at the current displacement. Returns `mat`.

```
(let ((mat (make-mat 14 :initial-contents '(-1 0 1 2 3
                                            4 5 6 7
                                            8 9 10 11 12))))
  (reshape-and-displace! mat '(4 3) 1)
  (map-displacements #'print mat 4))
..
.. #<MAT 1+4+9 B #(0.0d0 1.0d0 2.0d0 3.0d0)>
.. #<MAT 5+4+5 B #(4.0d0 5.0d0 6.0d0 7.0d0)>
.. #<MAT 9+4+1 B #(8.0d0 9.0d0 10.0d0 11.0d0)>
```

- **[function]** **map-mats-into** *result-mat fn &rest mats*

  Like `cl:map-into` but for `mat` objects. Destructively modifies `result-mat` to contain the results of applying `fn` to each element in the argument `mats` in turn.

14 Random numbers

Unless noted these work efficiently with CUDA.

- **[generic-function]** **copy-random-state** *state*

  Return a copy of `state` be it a lisp or cuda random state.

- **[function]** **uniform-random!** *mat &key (limit 1)*

  Fill `mat` with random numbers sampled uniformly from the [0,LIMIT) interval of `mat`'s type.

- **[function]** **gaussian-random!** *mat &key (mean 0) (stddev 1)*

  Fill `mat` with independent normally distributed random numbers with `mean` and `stddev`.

- **[function]** **mv-gaussian-random** *&key means covariances*

  Return a column vector of samples from the multivariate normal distribution defined by `means` (Nx1) and `covariances` (NxN). No CUDA implementation.

- **[function]** **orthogonal-random!** *m &key (scale 1)*

  Fill the matrix `m` with random values in such a way that `m^t * m` is the identity matrix (or something close if `m` is wide). Return `m`.

15 I/O

- **[variable]** ***mat-headers*** *t*

  If true, a header with `mat-ctype` and `mat-size` is written by `write-mat` before the contents and `read-mat` checks that these match the matrix into which it is reading.

- **[generic-function]** **write-mat** *mat stream*

  Write `mat` to binary `stream` in portable binary format. Return `mat`. Displacement and size are taken into account, only visible elements are written. Also see `*mat-headers*`.

- **[generic-function]** **read-mat** *mat stream*

  Destructively modify the visible portion (with regards to displacement and shape) of `mat` by reading `mat-size` number of elements from binary `stream`. Return `mat`. Also see `*mat-headers*`.

16 Debugging

The largest class of bugs has to do with synchronization of facets being broken. This is almost always caused by an operation that mispecifies the `direction` argument of `with-facet`. For example, the matrix argument of `scal!` should be accessed with direciton `:io`. But if it's `:input` instead, then subsequent access to the `array`(`0` `1`) facet will not see the changes made by `axpy!`, and if it's `:output`, then any changes made to the `array` facet since the last update of the `cuda-array` facet will not be copied and from the wrong input `scal!` will compute the wrong result.

Using the SLIME inspector or trying to access the `cuda-array` facet from threads other than the one in which the corresponding CUDA context was initialized will fail. For now, the easy way out is to debug the code with CUDA disabled (see `*cuda-enabled*`).

Another thing that tends to come up is figuring out where memory is used.

- **[function]** **mat-room** *&key (stream *standard-output*) (verbose t)*

  Calls `foreign-room` and `cuda-room`.

- **[macro]** **with-mat-counters** *(&key count n-bytes) &body body*

  Count all `mat` allocations and also the number of bytes they may require. *May require* here really means an upper bound, because `(make-mat (expt 2 60))` doesn't actually uses memory until one of its facets is accessed (don't simply evaluate it though, printing the result will access the `array` facet if `*print-mat*`). Also, while facets today all require the same number of bytes, this may change in the future. This is a debugging tool, don't use it in production.

```
(with-mat-counters (:count count :n-bytes n-bytes)
  (assert (= count 0))
  (assert (= n-bytes 0))
  (make-mat '(2 3) :ctype :double)
  (assert (= count 1))
```

```
(assert (= n-bytes (* 2 3 8)))
(with-mat-counters (:n-bytes n-bytes-1 :count count-1)
  (make-mat '7 :ctype :float)
  (assert (= count-1 1))
  (assert (= n-bytes-1 (* 7 4))))
(assert (= n-bytes (+ (* 2 3 8) (* 7 4))))
(assert (= count 2)))
```

17 Facet API

17.1 Facets

A `mat` is a `cube` (see Cube Manual) whose facets are different representations of numeric arrays. These facets can be accessed with `with-facets` with one of the following `facet-name` locatives:

- **[facet-name] backing-array**

  The corresponding facet's value is a one dimensional lisp array or a static vector that also looks exactly like a lisp array but is allocated in foreign memory. See `*foreign-array-strategy*`.

- **[facet-name] array**

  Same as `backing-array` if the matrix is one-dimensional, all elements are visible (see Shaping), else it's a lisp array displaced to the backing array.

- **[facet-name] foreign-array**

  The facet's value is a `foreign-array` which is an `offset-pointer` wrapping a CFFI pointer. See `*foreign-array-strategy*`.

- **[facet-name] cuda-host-array**

  This facet's value is a basically the same as that of `foreign-array`. In fact, they share storage. The difference is that accessing `cuda-host-array` ensures that the foreign memory region is page-locked and registered with the CUDA Driver API function cuMemHostRegister(). Copying between GPU memory (`cuda-array`) and registered memory is significantly faster than with non-registered memory and also allows overlapping copying with computation. See `with-syncing-cuda-facets`.

- **[facet-name] cuda-array**

  The facet's value is a `cuda-array`, which is an `offset-pointer` wrapping a `cl-cuda.driver-api:cu-device-ptr`, allocated with `cl-cuda.driver-api:cu-mem-alloc` and freed automatically.

Facets bound by with `with-facets` are to be treated as dynamic extent: it is not allowed to keep a reference to them beyond the dynamic scope of `with-facets`.

For example, to implement the `fill!` operation using only the `backing-array`, one could do this:

```
(let ((displacement (mat-displacement x))
      (size (mat-size x)))
 (with-facets ((x* (x 'backing-array :direction :output)))
   (fill x* 1 :start displacement :end (+ displacement size))))
```

`direction` is `:output` because we clobber all values in `x`. Armed with this knowledge about the direction, `with-facets` will not copy data from another facet if the backing array is not up-to-date.

To transpose a 2d matrix with the `array` facet:

```
(destructuring-bind (n-rows n-columns) (mat-dimensions x)
  (with-facets ((x* (x 'array :direction :io)))
    (dotimes (row n-rows)
      (dotimes (column n-columns)
        (setf (aref x* row column) (aref x* column row))))))
```

Note that `direction` is `:io`, because we need the data in this facet to be up-to-date (that's the input part) and we are invalidating all other facets by changing values (that's the output part).

To sum the values of a matrix using the `foreign-array` facet:

```
(let ((sum 0))
  (with-facets ((x* (x 'foreign-array :direction :input)))
    (let ((pointer (offset-pointer x*)))
      (loop for index below (mat-size x)
            do (incf sum (cffi:mem-aref pointer (mat-ctype x) index)))))
  sum)
```

See `direction` for a complete description of `:input`, `:output` and `:io`. For `mat` objects, that needs to be refined. If a `mat` is reshaped and/or displaced in a way that not all elements are visible then those elements are always kept intact and copied around. This is accomplished by turning `:output` into `:io` automatically on such `mat`s.

We have finished our introduction to the various facets. It must be said though that one can do anything without ever accessing a facet directly or even being aware of them as most operations on `mat`s take care of choosing the most appropriate facet behind the scenes. In particular, most operations automatically use CUDA, if available and initialized. See `with-cuda*` for detail.

17.2 Foreign arrays

One facet of `mat` objects is `foreign-array` which is backed by a memory area that can be a pinned lisp array or is allocated in foreign memory depending on `*foreign-array-strategy*`.

- **[class]** **foreign-array**

  `foreign-array` wraps a foreign pointer (in the sense of `cffi:pointerp`). That is, both `offset-pointer` and `base-pointer` return a foreign pointer. There are no other public operations that work with `foreign-array` objects, their sole purpose is represent facets of `mat` objects.

- **[variable]** ***foreign-array-strategy*** "*-see below-*"

One of :pinned, :static and :cuda-host (see type foreign-array-strategy). This variable controls how foreign arrays are handled, and it can be changed at any time.

If it's :pinned (only supported if pinning-supported-p), then no separate storage is allocated for the foreign array. Instead, it aliases the lisp array (via the backing-array facet).

If it's :static, then the lisp backing arrays are allocated statically via the static-vectors library. On some implementations, explicit freeing of static vectors is necessary, this is taken care of by finalizers or can be controlled with with-facet-barrier. destroy-cube and destroy-facet may also be of help.

:cuda-host is the same as :static, but any copies to/from the GPU (i.e. the cuda-array facet) will be done via the cuda-host-array facet whose memory pages will also be locked and registered with cuMemHostRegister which allows quicker and asynchronous copying to and from CUDA land.

The default is :pinned if available, because it's the most efficient. If pinning is not available, then it's :static.

- **[type] foreign-array-strategy**

  One of :pinned, :static and :cuda-host. See *foreign-array-strategy* for their semantics.

- **[function] pinning-supported-p**

  Return true iff the lisp implementation efficiently supports pinning lisp arrays. Pinning ensures that the garbage collector doesn't move the array in memory. Currently this is only supported on SBCL gencgc platforms.

- **[function] foreign-room** *&key (stream *standard-output*) (verbose t)*

  Print a summary of foreign memory usage to stream. If verbose, make the output human easily readable, else try to present it in a very concise way. Sample output with verbose:

```
Foreign memory usage:
foreign arrays: 450 (used bytes: 3,386,295,808)
```

  The same data presented with verbose false:

```
f: 450 (3,386,295,808)
```

17.3 CUDA

- **[function] cuda-available-p** *&key (device-id 0)*

  Check that a cuda context is already in initialized in the current thread or a device with device-id is available.

- **[macro] with-cuda*** *(&key (enabled '*cuda-enabled*) (device-id '*cuda-default-device-id*) (random-seed '*cuda-default-random-seed*) (n-random-states '*cuda-default-n-random-states*) n-pool-bytes) &body body*

Initializes CUDA with with all bells and whistles before `body` and deinitializes it after. Simply wrapping `with-cuda*` around a piece code is enough to make use of the first available CUDA device or fall back on blas and lisp kernels if there is none.

If CUDA is already initialized, then it sets up a facet barrier which destroys `cuda-array` and `cuda-host-array` facets after ensuring that the `array` facet is up-to-date.

Else, if CUDA is available and `enabled`, then in addition to the facet barrier, a CUDA context is set up, `*n-memcpy-host-to-device*`, `*n-memcpy-device-to-host*` are bound to zero, a cublas handle created, and `*curand-state*` is bound to a `curand-xorwow-state` with `n-random-states`, seeded with `random-seed`, and allocation of device memory is limited to `n-pool-bytes` (`nil` means no limit, see CUDA Memory Management).

Else - that is, if CUDA is not available, `body` is simply executed.

- **[function]** **call-with-cuda** *fn &key ((:enabled *cuda-enabled*) *cuda-enabled*) (device-id *cuda-default-device-id*) (random-seed *cuda-default-random-seed*) (n-random-states *cuda-default-n-random-states*) n-pool-bytes*

  Like `with-cuda*`, but takes a no argument function instead of the macro's `body`.

- **[variable]** ***cuda-enabled*** *t*

  Set or bind this to false to disable all use of cuda. If this is done from within `with-cuda`, *then cuda becomes temporarily disabled. If this is done from outside* `with-cuda`, then it changes the default values of the `enabled` argument of any future `with-cuda*`s which turns off cuda initialization entirely.

- **[accessor]** **cuda-enabled** *mat (:cuda-enabled = *default-mat-cuda-enabled*)*

  The control provided by `*cuda-enabled*` can be too coarse. This flag provides a per-object mechanism to turn cuda off. If it is set to `nil`, then any operation that pays attention to this flag will not create or access the `cuda-array` facet. Implementationally speaking, this is easily accomplished by using `use-cuda-p`.

- **[variable]** ***default-mat-cuda-enabled*** *t*

  The default for `cuda-enabled`.

- **[variable]** ***n-memcpy-host-to-device*** *0*

  Incremented each time a host to device copy is performed. Bound to 0 by `with-cuda*`. Useful for tracking down performance problems.

- **[variable]** ***n-memcpy-device-to-host*** *0*

  Incremented each time a device to host copy is performed. Bound to 0 by `with-cuda*`. Useful for tracking down performance problems.

- **[variable]** ***cuda-default-device-id*** *0*

  The default value of `with-cuda*`'s `:device-id` argument.

- **[variable]** ***cuda-default-random-seed*** *1234*

The default value of `with-cuda*`'s `:random-seed` argument.

- **[variable]** ***cuda-default-n-random-states*** *4096*

  The default value of `with-cuda*`'s `:n-random-states` argument.

17.3.1 CUDA Memory Management

The GPU (called *device* in CUDA terminology) has its own memory and it can only perform computation on data in this *device memory* so there is some copying involved to and from main memory. Efficient algorithms often allocate device memory up front and minimize the amount of copying that has to be done by computing as much as possible on the GPU.

MGL-MAT reduces the cost of device of memory allocations by maintaining a cache of currently unused allocations from which it first tries to satisfy allocation requests. The total size of all the allocated device memory regions (be they in use or currently unused but cached) is never more than `n-pool-bytes` as specified in `with-cuda*`. `n-pool-bytes` being `nil` means no limit.

- **[condition]** **cuda-out-of-memory** *storage-condition*

  If an allocation request cannot be satisfied (either because of `n-pool-bytes` or physical device memory limits being reached), then `cuda-out-of-memory` is signalled.

- **[function]** **cuda-room** *&key (stream *standard-output*) (verbose t)*

  When CUDA is in use (see `use-cuda-p`), print a summary of memory usage in the current CUDA context to `stream`. If `verbose`, make the output human easily readable, else try to present it in a very concise way. Sample output with `verbose`:

```
CUDA memory usage:
device arrays: 450 (used bytes: 3,386,295,808, pooled bytes: 1,816,657,920)
host arrays: 14640 (used bytes: 17,380,147,200)
host->device copies: 154,102,488, device->host copies: 117,136,434
```

  The same data presented with `verbose` false:

```
d: 450 (3,386,295,808 + 1,816,657,920), h: 14640 (17,380,147,200)
h->d: 154,102,488, d->h: 117,136,434
```

That's it about reducing the cost allocations. The other important performance consideration, minimizing the amount copying done, is very hard to do if the data doesn't fit in device memory which is often a very limited resource. In this case the next best thing is to do the copying concurrently with computation.

- **[macro]** **with-syncing-cuda-facets** *(mats-to-cuda mats-to-cuda-host &key (safep '*syncing-cuda-facets-safe-p*)) &body body*

  Update CUDA facets in a possibly asynchronous way while `body` executes. Behind the scenes, a separate CUDA stream is used to copy between registered host memory and device memory. When `with-syncing-cuda-facets` finishes either by returning normally or by a performing a non-local-exit the following are true:

  - All `mat`s in `mats-to-cuda` have an up-to-date `cuda-array` facet.

- All `mat`s in `mats-to-cuda-host` have an up-to-date `cuda-host-array` facet and no `cuda-array`.

It is an error if the same matrix appears in both `mats-to-cuda` and `mats-to-cuda-host`, but the same matrix may appear any number of times in one of them.

If `safep` is true, then the all matrices in either of the two lists are effectively locked for output until `with-syncing-cuda-facets` finishes. With SAFE `nil`, unsafe accesses to facets of these matrices are not detected, but the whole operation has a bit less overhead.

- **[variable]** ***syncing-cuda-facets-safe-p*** *t*

  The default value of the `safep` argument of `with-syncing-cuda-facets`.

Also note that often the easiest thing to do is to prevent the use of CUDA (and consequently the creation of `cuda-array` facets, and allocations). This can be done either by binding `*cuda-enabled*` to `nil` or by setting `cuda-enabled` to `nil` on specific matrices.

18 Writing Extensions

New operations are usually implemented in lisp, CUDA, or by calling a foreign function in, for instance, BLAS, CUBLAS, CURAND.

18.1 Lisp Extensions

- **[macro]** **define-lisp-kernel** *(name &key (ctypes '(:float :double))) (&rest params) &body body*

  This is very much like `define-cuda-kernel` but for normal lisp code. It knows how to deal with `mat` objects and can define the same function for multiple `ctypes`. Example:

```
(define-lisp-kernel (lisp-.+!)
    ((alpha single-float) (x :mat :input) (start-x index) (n index))
  (loop for xi of-type index upfrom start-x
          below (the! index (+ start-x n))
        do (incf (aref x xi) alpha)))
```

  Parameters are either of the form `(<name> <lisp-type)` or `(<name> :mat <direction>)`. In the latter case, the appropriate CFFI pointer is passed to the kernel. `<direction>` is passed on to the `with-facet` that's used to acquire the foreign array. Note that the return type is not declared.

  Both the signature and the body are written as if for single floats, but one function is defined for each ctype in `ctypes` by transforming types, constants and code by substituting them with their ctype equivalents. Currently this means that one needs to write only one kernel for `single-float` and `double-float`. All such functions get the declaration from `*default-lisp-kernel-declarations*`.

  Finally, a dispatcher function with `name` is defined which determines the ctype of the `mat` objects passed for `:mat` typed parameters. It's an error if they are not of the same type.

Scalars declared `single-float` are coerced to that type and the appropriate kernel is called.

- **[variable]** ***default-lisp-kernel-declarations*** *((optimize speed (sb-c:insert-array-bounds-checks 0)))*

  These declarations are added automatically to kernel functions.

18.2 CUDA Extensions

- **[function]** **use-cuda-p** *&rest mats*

  Return true if cuda is enabled (`*cuda-enabled*`), it's initialized and all `mats` have `cuda-enabled`. Operations of matrices use this to decide whether to go for the CUDA implementation or BLAS/Lisp. It's provided for implementing new operations.

- **[function]** **choose-1d-block-and-grid** *n max-n-warps-per-block*

  Return two values, one suitable as the `:block-dim`, the other as the `:grid-dim` argument for a cuda kernel call where both are one-dimensional (only the first element may be different from 1).

  The number of threads in a block is a multiple of `*cuda-warp-size*`. The number of blocks is between 1 and and `*cuda-max-n-blocks*`. This means that the kernel must be able handle any number of elements in each thread. For example, a strided kernel that adds a constant to each element of a length `n` vector looks like this:

```
(let ((stride (* block-dim-x grid-dim-x)))
  (do ((i (+ (* block-dim-x block-idx-x) thread-idx-x)
          (+ i stride)))
      ((>= i n))
    (set (aref x i) (+ (aref x i) alpha))))
```

  It is often the most efficient to have `max-n-warps-per-block` around

  4. Note that the maximum number of threads per block is limited by hardware (512 for compute capability < 2.0, 1024 for later versions), so `*cuda-max-n-blocks*` times `max-n-warps-per-block` must not exceed that limit.

- **[function]** **choose-2d-block-and-grid** *dimensions max-n-warps-per-block*

  Return two values, one suitable as the `:block-dim`, the other as the `:grid-dim` argument for a cuda kernel call where both are two-dimensional (only the first two elements may be different from 1).

  The number of threads in a block is a multiple of `*cuda-warp-size*`. The number of blocks is between 1 and and `*cuda-max-n-blocks*`. Currently - but this may change - the `block-dim-x` is always `*cuda-warp-size*` and `grid-dim-x` is always 1.

  This means that the kernel must be able handle any number of elements in each thread. For example, a strided kernel that adds a constant to each element of a HEIGHT*WIDTH matrix looks like this:

```
(let ((id-x (+ (* block-dim-x block-idx-x) thread-idx-x))
      (id-y (+ (* block-dim-y block-idx-y) thread-idx-y))
      (stride-x (* block-dim-x grid-dim-x))
      (stride-y (* block-dim-y grid-dim-y)))
  (do ((row id-y (+ row stride-y)))
      ((>= row height))
    (let ((i (* row width)))
      (do ((column id-x (+ column stride-x)))
          ((>= column width))
        (set (aref x i) (+ (aref x i) alpha))
        (incf i stride-x)))))
```

- **[function]** **choose-3d-block-and-grid** *dimensions max-n-warps-per-block*

  Return two values, one suitable as the `:block-dim`, the other as the `:grid-dim` argument for a cuda kernel call where both are two-dimensional (only the first two elements may be different from 1).

  The number of threads in a block is a multiple of `*cuda-warp-size*`. The number of blocks is between 1 and and `*cuda-max-n-blocks*`. Currently - but this may change - the `block-dim-x` is always `*cuda-warp-size*` and `grid-dim-x` is always 1.

  This means that the kernel must be able handle any number of elements in each thread. For example, a strided kernel that adds a constant to each element of a `thickness` * `height` * `width` 3d array looks like this:

```
(let ((id-x (+ (* block-dim-x block-idx-x) thread-idx-x))
      (id-y (+ (* block-dim-y block-idx-y) thread-idx-y))
      (id-z (+ (* block-dim-z block-idx-z) thread-idx-z))
      (stride-x (* block-dim-x grid-dim-x))
      (stride-y (* block-dim-y grid-dim-y))
      (stride-z (* block-dim-z grid-dim-z)))
  (do ((plane id-z (+ plane stride-z)))
      ((>= plane thickness))
    (do ((row id-y (+ row stride-y)))
        ((>= row height))
      (let ((i (* (+ (* plane height) row)
                  width)))
        (do ((column id-x (+ column stride-x)))
            ((>= column width))
          (set (aref x i) (+ (aref x i) alpha))
          (incf i stride-x))))))
```

- **[macro]** **define-cuda-kernel** *(name &key (ctypes '(:float :double))) (return-type params) &body body*

  This is an extended `cl-cuda:defkernel` macro. It knows how to deal with `mat` objects and can define the same function for multiple `ctypes`. Example:

```
(define-cuda-kernel (cuda-.+!)
    (void ((alpha float) (x :mat :input) (n int)))
  (let ((stride (* block-dim-x grid-dim-x)))
```

```
    (do ((i (+ (* block-dim-x block-idx-x) thread-idx-x)
            (+ i stride)))
        ((>= i n))
      (set (aref x i) (+ (aref x i) alpha)))))
```

The signature looks pretty much like in `cl-cuda:defkernel`, but parameters can take the form of `(<name> :mat <direction>)` too, in which case the appropriate `cl-cuda.driver-api:cu-device-ptr` is passed to the kernel. `<direction>` is passed on to the `with-facet` that's used to acquire the cuda array.

Both the signature and the body are written as if for single floats, but one function is defined for each ctype in `ctypes` by transforming types, constants and code by substituting them with their ctype equivalents. Currently this means that one needs to write only one kernel for `float` and `double`.

Finally, a dispatcher function with `name` is defined which determines the ctype of the `mat` objects passed for `:mat` typed parameters. It's an error if they are not of the same type. Scalars declared `float` are coerced to that type and the appropriate kernel is called.

18.2.1 CUBLAS

In a `with-cuda*` BLAS Operations will automatically use CUBLAS. No need to use these at all.

- **[condition]** **cublas-error** *error*
- **[reader]** **cublas-error-function-name** *cublas-error (:function-name)*
- **[reader]** **cublas-error-status** *cublas-error (:status)*
- **[variable]** ***cublas-handle***
- **[function]** **cublas-create** *handle*
- **[function]** **cublas-destroy** *&key (handle *cublas-handle*)*
- **[macro]** **with-cublas-handle** *nil &body body*
- **[function]** **cublas-get-version** *version &key (handle *cublas-handle*)*

18.2.2 CURAND

This the low level CURAND API. You probably want Random numbers instead.

- **[macro]** **with-curand-state** *(state) &body body*
- **[variable]** ***curand-state***
- **[class]** **curand-xorwow-state**
- **[reader]** **n-states** *curand-xorwow-state (:n-states)*
- **[reader]** **states** *curand-xorwow-state (:states)*

19 Indices

Referrer definition type abbreviations:

- *f*: for definitions in the function namespace (macros, compiler macros and also methods)
- *t*: DEFTYPEs, classes, conditions, structs
- *d*: documentation sections and glossary terms
- *l*: definitions of definition types
- *s*: ASDF systems
- *p*: packages
- *n*: named readtables
- *v*: special variables and constants
- *r*: restarts
- *?*: other

19.1 Function and Macro Index

19.2 Variable and Constant Index

19.3 Type Index

19.4 Misc Index