BMKernel 1880 Guide

Portability is for people who cannot write new programs.

LINUS TORVALDS

Wanwei Cai
Bitmain Technologies Inc
wanwei.cai@bitmain.com

Mingkang Qin
Bitmain Technologies Inc
mingkang.qin@bitmain.com

Contents

Chapter 1

Introduction

> I claim not to have controlled events, but confess plainly that events have controlled me.
>
> Abraham Lincoln (1864)

BMKernel is a set of simple wrapping utilities over Bitmain's AI chips. To make chip features accessible to software developers, a few abstractions are introduced, none of which are heavyweight. Because at such a low level in software stack, robustness and performance usually weigh much more than flexibility and consistency. So simplicity is the foremost design goal, and portability is not a concern. If a hardware feature behaves differently between two chip versions, its software interface in BMKernel will differ accordingly. Users who are seeking for a stable API to rest upon should never adventure this far. And for those who are brave and curious enough, welcome to the jungle.

Till now, all of Bitmain's AI chips are based on similar architectures, with slightly differing supports for data formats and arithmetic functions. This situation will probably change in the near future. In that case BMKernel will happily evolve by incorporating in entirely new programming models along with existing ones.

The main, if not only, labor BMKernel engages in is formatting users' computation requests into hardware instructions, thus freeing them from bothering about the exact byte or bit locations each computation parameter values should be fitted into. A typical workflow begins by users configuring a BMKernel context. And then they specify their computation requests by calling computation APIs. Finally they acquire the generated hardware instructions. The instructions may be sent to be executed on real hardware, or be processed for further analysis and transformations, all on users' tastes.

Final words on notions introduced in this guide: don't be mislead by their names. The notions are abstract entities BMKernel makes up to explain its programming model. They are nothing more than what BMKernel supposes them to be. Although we have strived to name the notions so that their real world counterparts possess similar behaviors, there are still differences we can not fully erase. For example, in BMKernel, a *chip* is an abstract entity consisting of four components. But when a BMKernel program is to run on a real hardware platform, the platform may not really implement the functions of the four components in a physical chip. Therefore, it is a wise way to think of all notions as entirely new concepts when their names don't help a lot.

Chapter 2

Programming Model

> Considering the current sad state of our computer programs, software development is clearly still a black art, and cannot yet be called an engineering discipline.
>
> William Jefferson Clinton

2.1 The Notion of Chip

The model is based on a notion of *chip*, which consists of a global memory (gmem), a local memory (lmem), a DMA engine (DMA for short) and a computation engine called TPU.

gmem is a memory that is accessible from outside of the chip. All data from external world should be put into gmem first, before any further processing.

lmem is the only momery that TPU has direct access to. Data must be loaded into lmem before being read by TPU, and the result of computation by TPU must also be put into lmem.

DMA moves data between gmem and lmem. Some special data transformations may also be applied during the movement, though computations occur mainly in TPU.

TPU is the main computation unit on chip. It is capable of accessing massive data in lmem every single cycle. TPU behaves in a SIMD-like manner with lmem tightly coupled as a vector register.

A typical workflow of doing computation with BMKernel is:

1. Put data into gmem using some non-BMKernel mechanisms;
2. Move data from gmem into lmem using DMA;
3. Conduct computation using TPU;
4. Move the computation result from lmem into gmem;
5. Retrive the result in gmem using non-BMKernel functions once again.

The non-BMKernel mechanisms used to transfer data between gmem and external world is beyond the scope of this guide. Please refer to complementary documents.

2.2 Data Types

In BMKernel, DMA and TPU manipulate on two data types: tensor and matrix, both of which are aggregates over basic types. The basic types of data contained in tensors and matrice, depending on specific chip versions, may be ones of 8/16/32-bit fixed point integers or 16/32-bit floating point numbers.

Tensor is a set of basic data organized by four dimensions: N, C, H and W. *Matrix*, by contrast, is organized by two: row and column. For a specific tensor or matrix instance, the exact values of its dimensions are collectively called its *shape*.

Consider a tensor of shape (N, C, H, W). The total number of its basic data is $N \times C \times H \times W$. And every basic datum i in it has an index (n_i, c_i, h_i, w_i). Then consider a matrix of shape (R, C), where R and C stand for numbers of its rows and columns, respectively. In this case its basic data total $R \times C$ and every datum i has an index (r_i, c_i). Note that all index values start at 0, that is, (n_i, c_i, h_i, w_i) ranges from $(0, 0, 0, 0)$ to $(N-1, C-1, H-1, W-1)$, and so does (r_i, c_i).

Both DMA and TPU's functions are defined logically on these indexed basic data. For instance, DMA may move a tensor A of shape (N, C, H, W) into a tensor B of shape $(N, C, H \times 2, \frac{W}{2})$, assuming W is a multiple of 2. And during the movement, every datum i in tensor A, with index (n_i, c_i, h_i, w_i), is put into tensor B at index (n_i, c_i, h'_i, w'_i), where:

$$(h'_i, w'_i) = \begin{cases} (h_i \times 2, w_i) & \text{if } w_i < \frac{W}{2}, \\ (h_i \times 2 + 1, w_i - \frac{W}{2}) & \text{if } w_i \geq \frac{W}{2}. \end{cases}$$

This movement is kind of a reshaping along the H, W dimensions, with relative order between basic data unchanged. Another example regarding DMA will be the N/C-transposition of a tensor from of shape (N, C, H, W) to (C, N, H, W), where a basic datum of index (n_i, c_i, h_i, w_i) is moved into index (c_i, n_i, h_i, w_i). In this second example, the relative order between basic data *does* change at a $H \times W$ granularity. Now consider a TPU example of adding two tensors into one. The shapes of these three tensors are all the same, and the adding is defined point-wise. So if the values of basic data at index (n_i, c_i, h_i, w_i) of the two input tensors are a_i and b_i respectively, then the value of basic datum at index (n_i, c_i, h_i, w_i) of the result tensor is $a_i + b_i$. A slightly more complex TPU example is a summation within the H and W dimensions, where a tensor A of shape (N, C, H, W) is summed into a tensor B of shape $(N, C, 1, 1)$. Denoting the value of basic datum at index (n_i, c_i, h_i, w_i) of tensor A/B as $a_{n_i,c_i,h_i,w_i}/b_{n_i,c_i,h_i,w_i}$, the computation can be defined as:

$$b_{n_i,c_i,0,0} = \sum_{h_i=0}^{H-1} \sum_{w_i=0}^{W-1} a_{n_i,c_i,h_i,w_i} \text{ (where } 0 \leq n_i < N,\ 0 \leq c_i < C)$$

2.3 Data Layout

Being memories, gmem and lmem are sequences of addressable bytes. For a given tensor/matrix, the exact gmem/lmem addresses for its every particular byte is called the tensor/matrix's *layout* in gmem/lmem. Layout maps a tensor/matrix's logical structure into a sequence of physical byte addresses in gmem/lmem. It's like a mapping from logical layer, where DMA and TPU's functions are defined, into physical layer where addresses are assigned for each byte. So layout maps DMA and TPU's logical functions into the exact transformation of each byte in gmem/lmem by DMA and TPU. A tensor/matrix's layout is determined by a few factors: its logical dimensions, start address in gmem/lmem, byte-size of basic data and *stride* along each of its dimensions.

Consider a tensor of shape (N, C, H, W), starting at address *Base* in gmem, containing basic data each of $Size_{data}$ bytes and with a stride of (S_N, S_C, S_H, S_W). The address of its basic datum (n_i, c_i, h_i, w_i) is:

$$Base + Size_{data} \times (n_i \times S_N + c_i \times S_C + h_i \times S_H + w_i \times S_W)$$

For a matrix of $Size_{data}$-byte data in gmem, of shape (R, C), starting at *Base*, and with a stride of (S_R, S_C), the address of its datum (r_i, c_i) is computed as:

$$Base + Size_{data} \times (r_i \times S_R + c_i \times S_C)$$

When it comes to layout in lmem, things become a bit more complex. Unlike in gmem, where stride parameters can be used directly in the computation of addresses, stride in lmem works closely with lmem parameters, which we must cover first. As explained earlier, lmem is tightly coupled with TPU as a vector register. To that end, lmem is divided evenly into *slices*. For example, if lmem is of size 1MB and divided into 4 slices, then the size of each slice, called the lmem's *slice size*, is 256KB. When a tensor or matrix is put into lmem, it is kind of scattered into these slices, under directions from stride parameters.

Assume a lmem of size $Size_{total}$ bytes and divided into *Slices* slices. By definition, the slice size $Size_{slice}$ is $\frac{Size_{total}}{Slices}$ bytes. Now consider a tensor of shape (N, C, H, W), starting at address *Base* in lmem, containing basic data each sized $Size_{data}$, and strided by (S_N, S_C, S_H, S_W). Let $align(c_i, slices)$ be the smallest multiple of *slices* that is bigger than or equal to c_i, the address of datum (n_i, c_i, h_i, w_i) is:

$$\begin{aligned}&[(Base + c_i \times Size_{slice}) \bmod Size_{total}] + \\ &\qquad Size_{data} \times (n_i \times S_N + \frac{\frac{Base}{Size_{slice}} + c_i}{slices} \times S_C + h_i \times S_H + w_i \times S_W)\end{aligned}$$

Dear readers, don't be frightened by this formula, for its intuition is fairly simple: the data are scattered along its C dimension into each lmem slice, in turn, N times. Just note that the distance bewteen (n_i, c_i, h_i, w_i) and $(n_i, c_i + 1, h_i, w_i)$ is exactly $Size_{slice}$ bytes. That is, the two data are placed at the same offset within two adjacent lmem slices. Also note that (n_i, c_i, h_i, w_i) and $(n_i, c_i + slices, h_i, w_i)$ are placed in the same slice, apart from each other by S_C bytes.

When a matrix is put into lmem, it is treated as a specially shaped tensor. If a matrix is of shape (R, C), then the corresponding tensor shape is $(R, C', 1, W)$, where $C' \times W \geq C$. And matrix data (r_i, c_i) is located at $(ri, \frac{c_i}{W}, 0, c_i \bmod W)$ in the tensor. The contents of tensor locations that don't have matrix data related when $C' \times W$ is strictly bigger than C are undefined.

A word of caution: the stride must be selected carefully to make data with the same C-dimension index be placed into same lmem slice, otherwise the behavior is undefined. Formally speaking, let $address(n_i, c_i, h_i, w_i)$ be the address of data (n_i, c_i, h_i, w_i) as computed above, then for all Δ_N, Δ_H and Δ_W satisfying $0 \leq \Delta_N < N$, $0 \leq \Delta_H < H$ and $0 \leq \Delta_W < W$, and for all c_i satisfying $0 \leq c_i < C$, the following equation holds:

$$align(address(0, c_i, 0, 0), Size_{slice}) = align(address(\Delta_N, c_i, \Delta_H, \Delta_W), Size_{slice})$$

2.4 Default Strides

The default strides are assumed when user is not allowed or willing to specify the layout of a tensor or matrix, in gmem or lmem. Let's assume the shape of the tensor under discussion is (N, C, H, W), and the matrix (R, C).

For tensors in gmem, the default stride values are:

$$\begin{cases} S_N = C \times H \times W \\ S_C = H \times W \\ S_H = W \\ S_W = 1 \end{cases}$$

For matrice in gmem, the default stride values are:

$$\begin{cases} S_R = C \\ S_C = 1 \end{cases}$$

For tensors or matrice's specially shaped tensors in lmem, the *unaligned* stride values are:

$$\begin{cases} S_N = \frac{align(shape.c, slices)}{slices} \times stride.c \\ S_C = shape.h \times shape.w \\ S_H = shape.w \\ S_W = 1 \end{cases}$$

And the *aligned* stride values are:

$$\begin{cases} S_N = \frac{align(shape.c, slices)}{slices} \times stride.c \\ S_C = align(shape.h \times shape.w, eu\_num) \\ S_H = shape.w \\ S_W = 1 \end{cases}$$

Where *eu_num* is a chip specific parameter.

Chapter 3

System API

That's the thing about people who think they hate computers.
What they really hate is lousy programmers.

Larry Niven and Jerry Pournelle

A typical workflow begins by users configuring a BMKernel context. And then they specify their computation requests by calling computation APIs. Finally they acquire the generated hardware instructions. All the functions involved during the above workflow, excluding computation APIs, are collectively called *system* APIs.

The following code snippet shows how a typical work flow with BMKernel looks like, all system APIs involved will be explained in later sections:

```
bmk1880_info_t info;

/* Omitted: set info structure here */

/* Allocate a bmkernel context */
bmk1880_context_t *ctx = bmk1880_register(&info);

/* Omitted: call computation APIs here */

/* Get generated hardware instructions */
u32 size;
u8 *cmdbuf = bmk1880_acquire_cmdbuf(ctx, &size);

/* User-implemented processing */
do_something(cmdbuf, size);

/* Reset bmkernel context to initial state */
bmk1880_reset(ctx);

/* Omitted again: call computation APIs here */

/* Again, process instructions after reset */
u32 size;
u8 *cmdbuf = bmk1880_acquire_cmdbuf(ctx, &size);
```

```
do_something(cmdbuf, size);

/* Free allocated bmkernel context */
bmk1880_cleanup(ctx);
```

By default, all calls to computation APIs instruct TPU and DMA to function in a serial manner. That is, TPU and DMA will move one after another according to the order of function calls in source code, never at the same time. In real world applications, this serial execution is usually overkill and performance can be gained by exploiting data independence.

Consider an input tensor of shape (N, C, H, W) for example. The tensor's data of different N-dimension indice compute into independent results by all possible computation APIs. Therefore, DMA can safely move data of one N-dimension index into lmem while the data of another index are being computed by TPU, as long as their lmem address ranges don't overlap. This parallel execution of DMA and TPU effectively hides the time of moving data from gmem to lmem and definitely improves TPU's utilization.

BMKernel provides two programming styles to express possible parallelisms between DMA and TPU: one is engine-oriented and the other dependency-oriented.

In the *engine-oriented* style, user wraps functions *bmk1880_parallel_enable* and *bmk1880_parallel_disable* around the group of computations that can be executed in parallel so that DMA and TPU will execute them without synchronizing with each other. Note that computations on same engine are still executed serially in their program order, since there is only one engine of each type in the programming model. Any computation before or after this parallel group will be executed serially against it. In the following code:

```
 1  bmk1880_tl_load(ctx, a0, ga);
 2  bmk1880_tl_load(ctx, b0, gb);
 3
 4  bmk1880_parallel_enable(ctx);
 5  bmk1880_tl_add(ctx, r0, a0, b0);
 6  bmk1880_tl_load(ctx, a1, ga);
 7  bmk1880_tl_load(ctx, b1, gb);
 8  bmk1880_parallel_disable(ctx);
 9
10  bmk1880_parallel_enable(ctx);
11  bmk1880_tl_store(ctx, r0, gr0);
12  bmk1880_tl_add(ctx, r1, a1, b1);
13  bmk1880_parallel_disable(ctx);
14
15  bmk1880_tl_store(ctx, r1, gr1);
```

The computations from line 5 to 7 are one parallel group, and those on line 11 and line 12 are another. The execution order imposed, denoted by line numbers, is $1 \to 2 \to \{5, 6, 7\} \to \{11, 12\} \to 15$.

In the other *dependency-oriented* style, the default serial order implied from pragram order in source code is total canceled by an opening function call to *bmk1880_create_streams*. And user then imposes order manually by putting dependent computations into the same *streams*. Computations in each *stream* are executed serially in their program order. And computations from two different *streams* are independent with each other. Note that the order imposed by each engine still applies here. In the following code:

```
bmk1880_create_streams(2);

```

```
 3  bmk1880_set_stream(0);
 4  bmk1880_tl_load(a0, a0_gaddr);
 5  bmk1880_tl_load(b0, b0_gaddr);
 6  bmk1880_tl_add(c0, a0, b0);
 7  bmk1880_tl_store(c0, c0_gaddr);
 8
 9  bmk1880_set_stream(1);
10  bmk1880_tl_load(a1, a1_gaddr);
11  bmk1880_tl_load(b1, b1_gaddr);
12  bmk1880_tl_add(c1, a1, b1);
13  bmk1880_tl_store(c1, c1_gaddr);
```

Line 1 enables the *dependency-oriented* mode and creates two *streams*. Line 3 and 9 set the current *stream* into 0 and 1 respectively, so that later computations are put into the specified *stream*. The execution order imposed by *stream* 0 and *stream* 1 are $\{4 \rightarrow 5 \rightarrow 6 \rightarrow 7\}$ and $\{10 \rightarrow 11 \rightarrow 12 \rightarrow 13\}$. It can be proved that the expressiveness of *dependency-oriented* style is equivalent to the programming model.

3.1 bmk1880_register

User allocates a BMKernel context by filling a *bmk1880_info_t* structure and passing it to *bmk1880_register* function. The function returns a handle of the initialized context.

In the *bmk1880_info_t* structure: *chip_version* is an integer describing the version of chip to work with, and can be 1880 or 1880; *cmdbuf* (short for “command buffer”) is a user-allocated buffer to contain generated hardware instructions and *cmdbuf_size* describes its size in bytes. Note that user is responsible to free *cmdbuf* after the use of referring BMKernel context.

```
typedef struct {
  u32 chip_version;
  u8 *cmdbuf;
  u32 cmdbuf_size;
} bmk1880_info_t;

bmk1880_context_t * bmk1880_register(bmk1880_info_t *info);
```

3.2 bmk1880_cleanup

bmk1880_cleanup frees the context previously allocated by *bmk1880_register*.

```
void bmk1880_cleanup(bmk1880_context_t *ctx);
```

3.3 bmk1880_acquire_cmdbuf

bmk1880_acquire_cmdbuf returns a buffer of hardware instructions generated so far and set *(*size)* to buffer’s valid size in bytes. The buffer is an array of *cmd_hdr_t* structures each containing one variable-sized generated hardware instruction.

```
u8 *bmk1880_acquire_cmdbuf(bmk1880_context_t *ctx, u32 *size);

typedef struct {
  u8 engine_id : 4;
  ...
  u8 len;
  u8 cmd[0];
} cmd_hdr_t;
```

In the *cmd_hdr_t* structure, *engine_id* is the identifier of engine on which the contained instruction is supposed to be executed. And *len* indicates in bytes the length of the hardware instruction immediately following this *cmd_hdr_t* structure.

3.4 bmk1880_reset

bmk1880_reset resets current BMKernel context to its initial state as returned by *bmk1880_register*. This function is usually called after *bmk1880_acquire_cmdbuf* to empty the *cmdbuf* buffer.

```
void bmk1880_reset(bmk1880_context_t *ctx);
```

3.5 bmk1880_parallel_enable

bmk1880_parallel_enable claims that following computations on different engines can be executed with no synchornization with each other. This function enables *engine-oriented* parallel programming style.

```
void bmk1880_parallel_enable(bmk1880_context_t *ctx);
```

3.6 bmk1880_parallel_disable

bmk1880_parallel_disable disables *engine-oriented* parallel programming style.

```
void bmk1880_parallel_disable(bmk1880_context_t *ctx);
```

3.7 bmk1880_create_streams

bmk1880_create_streams creates *nr_streams* streams, indexed 0 to (*nr_streams* - 1), that following calls to *bmk1880_set_stream* can refer to. This function enables *dependency-oriented* parallel programming style. Note this style can not be disabled once enabled.

```
void bmk1880_create_streams(
  bmk1880_context_t *ctx,
  int nr_streams);
```

3.8 bmk1880_destroy_streams

bmk1880_destroy_streams destroys all the streams created by the previous call to *bmk1880_create_streams* and resets the system back to serial mode.

```
void bmk1880_destroy_streams(bmk1880_context_t *ctx);
```

3.9 bmk1880_set_stream

bmk1880_set_stream set current stream to stream *i* that has been created by calling *bmk1880_create_streams*. Following computations will be put into this stream until another *bmk1880_set_stream* specifying a different stream index is called.

```
void bmk1880_set_stream(bmk1880_context_t *ctx, int i);
```

3.10 bmk1880_add_dependency

bmk1880_add_dependency further restricts that the computation represented by *before* must take place strictly before that represented by *after*. Both *before* and *after* are pointers returned by some computation API.

```
void bmk1880_add_dependency(
  bmk1880_context_t *ctx,
  bmk1880_op_t *before,
  bmk1880_op_t *after);
```

Chapter 4

Computation API

> A programming language is low level when its programs require attention to the irrelevant.
>
> Alan Jay Perlis

During all kinds of computation, input values are first converted into 32-bit ones before any internal computation, and final 32-bit values are *saturated* into ranges that can be represented by the final 8-bit or 16-bit integer format. That is, if the value before saturation can be represented by the final integer format, it is unchanged. Otherwise it is *saturated* into the maximun or minimum in the final integer format, whichever is nearer to the original value. For example, if the final integer format is `FMT_U8`, then the representable maximum and minimum are 255 and 0 respectively. In this case, any value that is bigger than 255 becomes 255 after saturation, and values smaller than 0 are saturated into 0's.

About *signedness*, one general rule applies to all kinds of computation when not otherwise specified: the result is *unsigned* if and only if all input tensors or matrice are *unsigned.* A tensor or matrix is said to be *signed* if it is of format `FMT_I8`, *unsigned* if `FMT_U8`.

4.1 fmt_t

fmt_t describes the type of basic data in a tensor or matrix. The naming consists of three parts. "`FMT`" is a fixed prefix. A following "`I`" or "`U`" stands for signed integer or unsigned integer respectively. "`8`" describes the bit-width of the type.

```
typedef u32 fmt_t;

#define FMT_I8  4
#define FMT_U8  9
```

Depending on specific API functions, the tensor or matrix's basic data may be 8-bit or 16-bit. If a tensor or matrix's basic data are 16-bit, its high and low 8-bit parts are stored separately in lmem, with low 8-bit parts located at lower addresses. Consider for example a logical 16-bit tensor and assume that the datum at index (n_i, c_i, h_i, w_i) is comprised of v_i^h and v_i^l as its high and low 8-bit part, respectively. Then this tensor is physically stored as two 8-bit tensors, one containing v_i^h at index (n_i, c_i, h_i, w_i) and another one containing v_i^l. Depending on specific API functions, a 16-bit tensor or matrix may be represented by one or two 8-bit *tensor_lmem* structures. Either way, the value of the *tensor_lmem* structure's `fmt` field must be `FMT_I8` or `FMT_U8`.

4.2 shape_t

shape_t describes the shape of a tensor or matrix. *shape_t4* and *shape_t2* are used to construct *shape_t*'s for tensor and matrix, respectively.

```
typedef struct {
  u32 dim;
  u32 n;
  u32 c;
  union {
    u32 h;
    u32 row;
  };
  union {
    u32 w;
    u32 col;
  };
} shape_t;

shape_t shape_t4(int n, int c, int h, int w);
shape_t shape_t2(int row, int col);
```

4.3 stride_t

stride_t describes the stride of a tensor or matrix. *stride_t4* and *stride_t2* are used to construct *stride_t*'s for tensor and matrix, respectively.

```
typedef struct {
  u32 n;
  u32 c;
  union {
    u32 h;
    u32 row;
  };
  union {
    u32 w;
    u32 col;
  };
} stride_t;

stride_t stride_t4(int n, int c, int h, int w);
stride_t stride_t2(int row, int col);
```

4.4 tensor_lmem

tensor_lmem represents a tensor or matrix in *lmem*. *fmt*, *shape*, *stride* are as explained above. If *stride* is `NULL`, *aligned* will be referred as indication of two frequently used stride values.

```
typedef struct {
  fmt_t fmt;
  shape_t shape;
  stride_t *stride;
  bool aligned;
  ...
} tensor_lmem;
```

For tensors, if *aligned* is `false`, the stride values are as in the default **unaligned** stride on page 5. If *aligned* is `true`, the values are as in the default **aligned** stride on page 5. For matrice, stride values are computed by the shapes of corresponding specially shaped tensors, following the same rule.

4.5 tensor_gmem

tensor_gmem represents a tensor or matrix in gmem.

```
typedef struct {
  u64 addr;
  shape_t shape;
  stride_t stride;
} tensor_gmem;
```

4.6 bmk1880_chip_info

bmk1880_chip_info returns a structure describing design parameters of the BM1880 chip.

```
typedef struct {
  u32 version;
  u32 npu_num;
  u32 eu_num;
  u32 lmem_size;
  u32 lmem_banks;
  u32 lmem_bank_size;
} bmk1880_chip_info_t;

bmk1880_chip_info_t bmk1880_chip_info();
```

4.7 bmk1880_tl_prealloc

bmk1880_tl_prealloc allocates a *tensor_lmem* structure on heap memory, and constructs it as dictated by parameters. The parameter *la* is the starting address in lmem. The *tensor_lmem*'s *aligned* field is set to `false`. If the allocation succeeds, a pointer to the constructed structure is returned, `NULL` otherwise.

```
tensor_lmem * bmk1880_tl_prealloc(
  bmk1880_context_t *ctx,
  laddr_t la,
  shape_t s,
  fmt_t fmt);
```

4.8 bmk1880_tl_prealloc_align

Same as *bmk1880_tl_prealloc*, except the *aligned* field is set to `true`.

```
tensor_lmem * bmk1880_tl_prealloc_align(
  bmk1880_context_t *ctx,
  laddr_t la,
  shape_t s,
  fmt_t fmt);
```

4.9 bmk1880_tl_alloc

bmk1880_tl_alloc allocates a *tensor_lmem* structure on heap memory, and constructs it as dictated by parameters. Unlike in *bmk1880_tl_prealloc*, the starting address is not determined from parameters, but assigned by BMKernel automatically. BMKernel manages the starting addresses in lmem by a simple stack. The starting address in each returned *tensor_lmem* increases monotonically against successive *bmk1880_tl_alloc* calls. And the last allocated *tensor_lmem* must be freed first, using function *bmk1880_tl_free* explained soon. If the available memory in lmem is not enough to satisfy an allocation request, or some other error occurs, a `NULL` pointer is returned.

```
tensor_lmem * bmk1880_tl_alloc(
  bmk1880_context_t *ctx,
  shape_t s,
  fmt_t fmt,
  u32 ctrls);
```

tensor_lmem's *aligned* field is set to `false` when *ctrls* is `CTRL_NULL`, and `true` when `CTRL_AL`.

4.10 bmk1880_tl_alloc_bank

bmk1880_tl_alloc_bank allocates memory from a specific lmem bank, as dictated by the *bank_id* parameter.

```
tensor_lmem * bmk1880_tl_alloc_bank(
  bmk1880_context_t *ctx,
  u32 bank_id,
  shape_t s,
  fmt_t fmt,
  u32 ctrls);
```

4.11 bmk1880_tl_free

bmk1880_tl_free frees the *tensor_lmem* structure allocated by *bmk1880_tl_prealloc*, *bmk1880_tl_prealloc_align*, *bmk1880_tl_alloc* and *bmk1880_tl_alloc_bank* back to heap memory. If the structure is allocated by *bmk1880_tl_alloc* or *bmk1880_tl_alloc_bank*, *bmk1880_tl_free* also increases the available lmem memory managed by BMKernel and checks that the *last allocate, first free* rule is obeyed (see *bmk1880_tl_alloc*).

```
void bmk1880_tl_free(
  bmk1880_context_t *ctx, tensor_lmem *tlp);
```

4.12 bmk1880_gdma_copy_gmem

bmk1880_gdma_copy_gmem instructs DMA to copy tensor or matrix within gmem. *src* and *dst* must be both tensors or matrice and must contain 8-bit basic data only. The shapes of *src* and *dst* may be different, as long as their total numbers of basic data equal. When *src* and *dst* are tensors, *ctrls* can be `CTRL_TP`, indicating N/C-transposition. In other cases, *ctrls* must be `CTRL_NULL`.

```
bmk1880_op_t * bmk1880_gdma_copy_gmem(
  bmk1880_context_t *ctx,
  tensor_gmem *dst,
  tensor_gmem *src,
  ctrl_t ctrls);
```

4.13 bmk1880_gdma_copy_lmem

bmk1880_gdma_copy_lmem instructs DMA to copy a tensor (not matrix) within lmem, from *src* to *dst*. The shapes of *src* and *dst* may be different, as long as their total numbers of basic data equal. The basic data must be 8-bit.

```
bmk1880_op_t * bmk1880_gdma_copy_lmem(
  bmk1880_context_t *ctx,
  tensor_lmem *dst,
  tensor_lmem *src);
```

4.14 bmk1880_gdma_load

bmk1880_gdma_load instructs DMA to copy a tensor or matrix from gmem to lmem. The tensor or matrix starts at *gaddr* in gmem, and is strided by default values. When *ctrls* is `CTRL_TP` (instead of `CTRL_NULL`), it indicates N/C-transposition for a tensor, or row/column-transposition for a matrix. The basic data must be 8-bit.

```
bmk1880_op_t * bmk1880_gdma_load(
  bmk1880_context_t *ctx,
  tensor_lmem *t,
  u64 gaddr,
  ctrl_t ctrls);
```

4.15 bmk1880_gdma_store

Similar to *bmk1880_gdma_load*, but copies the tensor or matrix from lmem to gmem.

```
bmk1880_op_t * bmk1880_gdma_store(
  bmk1880_context_t *ctx,
  tensor_lmem *t,
  u64 gaddr,
  ctrl_t ctrls);
```

4.16 bmk1880_gdma_load_stride

Similar to *bmk1880_gdma_load*, but enables users to specify stride values in gmem.

```
bmk1880_op_t * bmk1880_gdma_load_stride(
  bmk1880_context_t *ctx,
  tensor_lmem *t,
  u64 gaddr,
  stride_t stride,
  ctrl_t ctrls);
```

4.17 bmk1880_gdma_store_stride

Similar to *bmk1880_gdma_store*, but enables users to specify stride values in gmem.

```
bmk1880_op_t * bmk1880_gdma_store_stride(
  bmk1880_context_t *ctx,
  tensor_lmem *t,
  u64 gaddr,
  stride_t stride,
  ctrl_t ctrls);
```

4.18 bmk1880_gdma_lrn_shift

bmk1880_gdma_lrn_shift instructs DMA to compute a tensor (not matrix) *dst* from tensor *src*, both of which are of same shape (N, C, H, W). If *right_shift* is `true`, the computation copies datum at index (n_i, c_i, h_i, w_i) in tensor *src* into index $(n_i, c_i + \mathit{lrn\_step}, h_i, w_i)$ in tensor *dst* for each $0 \leq c_i < C - \mathit{lrn\_step}$, and set datum at index (n_i, c_i, h_i, w_i) in tensor *dst* to zero for each $0 \leq c_i < \mathit{lrn\_step}$. If *right_shift* is `false`, the computation copies datum at index (n_i, c_i, h_i, w_i) in tensor *src* into index $(n_i, c_i - \mathit{lrn\_step}, h_i, w_i)$ in tensor *dst* for each $\mathit{lrn\_step} \leq c_i < C$, and set datum at index (n_i, c_i, h_i, w_i) in tensor *dst* to zero for each $C - \mathit{lrn\_step} \leq c_i < C$. The basic data must be 8-bit.

```
bmk1880_op_t * bmk1880_gdma_lrn_shift(
  bmk1880_context_t *ctx,
  tensor_lmem *dst,
  tensor_lmem *src,
  bool right_shift,
  int lrn_step);
```

4.19 bmk1880_tpu_mul

bmk1880_tpu_mul instructs TPU to compute $res_i = (a_i \times b_i) \gg rshift\_width$ for each datum a_i in tensor a and b_i in tensor b, where res_i, a_i and b_i are of same index. All tensors must be of same shape, and the basic data in all *tensor_lmem* structures must be 8-bit. If the result is a 16-bit tensor, *res_high* and *res_low* represent its high and low 8-bit parts, respectively. *res_high* should be `NULL` if the result is 8-bit. *rshift_width* indicates the bits to be shifted to right for each result value before saturation.

```
typedef struct {
  tensor_lmem *res_high;
  tensor_lmem *res_low;
  tensor_lmem *a;
  tensor_lmem *b;
  int rshift_width;
} bmk1880_mul_param_t;

bmk1880_op_t * bmk1880_tpu_mul(
  bmk1880_context_t *ctx,
  const bmk1880_mul_param_t *p);
```

4.20 bmk1880_tpu_mul_const

Similar to *bmk1880_tpu_mul*, but tensor b is replaced by an 8-bit constant. The constant is signed if *b_is_signed* is `true`, unsigned otherwise.

```
typedef struct {
  tensor_lmem *res_high;
  tensor_lmem *res_low;
  tensor_lmem *a;
  s8 b;
  bool b_is_signed;
  int rshift_width;
} bmk1880_mul_const_param_t;

bmk1880_op_t * bmk1880_tpu_mul_const(
  bmk1880_context_t *ctx,
  const bmk1880_mul_const_param_t *p);
```

4.21 bmk1880_tpu_mac

bmk1880_tpu_mac instructs TPU to compute $res_i = (a_i \times b_i + (res_i \ll lshift\_width)) \gg rshift\_width$ for each datum a_i in tensor a, b_i in tensor b and res_i represented by *res_high* and *res_low* together, where res_i, a_i and b_i are of same index. All tensors must be of same shape, and the basic data in all *tensor_lmem* structures must be 8-bit. The result is a 16-bit tensor if *res_is_int8* is `false`, or a 8-bit tensor otherwise. *rshift_width* indicates the bits to be shifted to right for each result value before saturation. Note that *res_high* and *res_low* are used both as input res_i's and output res_i's. Input res_i's are fixed to be 16-bit so that both *res_high* and *res_low* must be non-`NULL`. When the result is a 8-bit tensor, it is stored into *res_low*.

```
typedef struct {
  tensor_lmem *res_high;
  tensor_lmem *res_low;
  bool res_is_int8;
  tensor_lmem *a;
  tensor_lmem *b;
  int lshift_width;
  int rshift_width;
} bmk1880_mac_param_t;

bmk1880_op_t * bmk1880_tpu_mac(
  bmk1880_context_t *ctx,
  const bmk1880_mac_param_t *p);
```

4.22 bmk1880_tpu_mac_const

Similar to *bmk1880_tpu_mac*, but tensor b is replaced by an 8-bit constant. The constant is signed if *b_is_signed* is `true`, unsigned otherwise.

```
typedef struct {
  tensor_lmem *res_high;
  tensor_lmem *res_low;
  bool res_is_int8;
  tensor_lmem *a;
  s8 b;
  bool b_is_signed;
  int lshift_width;
  int rshift_width;
} bmk1880_mac_const_param_t;

bmk1880_op_t * bmk1880_tpu_mac_const(
  bmk1880_context_t *ctx,
  const bmk1880_mac_const_param_t *p);
```

4.23 bmk1880_tpu_add

bmk1880_tpu_add instructs TPU to compute $res_i = a_i + b_i$ for each datum a_i in tensor a and b_i in tensor b, where res_i, a_i and b_i are of same index. All tensors must be of same shape, and the basic data in all *tensor_lmem* structures must be 8-bit. Tensor a and tensor b must all be 16-bit so that *a_high* and *b_high* must not be `NULL`. If the result is a 16-bit tensor, *res_high* and *res_low* represent its high and low 8-bit parts, respectively. *res_high* should be `NULL` if the result is 8-bit.

```
typedef struct {
  tensor_lmem *res_high;
  tensor_lmem *res_low;
  tensor_lmem *a_high;
  tensor_lmem *a_low;
  tensor_lmem *b_high;
  tensor_lmem *b_low;
} bmk1880_add_param_t;
```

```
bmk1880_op_t * bmk1880_tpu_add(
  bmk1880_context_t *ctx,
  const bmk1880_add_param_t *p);
```

4.24 bmk1880_tpu_add_const

Similar to *bmk1880_tpu_add*, but tensor *b* is replaced by a 16-bit constant. The constant is signed if *b_is_signed* is `true`, unsigned otherwise.

```
typedef struct {
  tensor_lmem *res_high;
  tensor_lmem *res_low;
  tensor_lmem *a_high;
  tensor_lmem *a_low;
  s16 b;
  bool b_is_signed;
} bmk1880_add_const_param_t;

bmk1880_op_t * bmk1880_tpu_add_const(
  bmk1880_context_t *ctx,
  const bmk1880_add_const_param_t *p);
```

4.25 bmk1880_tpu_sub

bmk1880_tpu_sub instructs TPU to compute $res_i = a_i - b_i$ for each datum a_i in tensor a and b_i in tensor b, where res_i, a_i and b_i are of same index. All tensors must be of same shape, and the basic data in all *tensor_lmem* structures must be 8-bit. Tensor *a* and tensor *b* must all be 16-bit so that *a_high* and *b_high* must not be `NULL`. The result must be signed integers so that the *fmt_t* field in *res_high* and *res_low* must be `FMT_I8`. If the result is a 16-bit tensor, *res_high* and *res_low* represent its high and low 8-bit parts, respectively. *res_high* should be `NULL` if the result is 8-bit.

```
typedef struct {
  tensor_lmem *res_high;
  tensor_lmem *res_low;
  tensor_lmem *a_high;
  tensor_lmem *a_low;
  tensor_lmem *b_high;
  tensor_lmem *b_low;
} bmk1880_sub_param_t;

bmk1880_op_t * bmk1880_tpu_sub(
  bmk1880_context_t *ctx,
  const bmk1880_sub_param_t *p);
```

4.26 bmk1880_tpu_max

bmk1880_tpu_max instructs TPU to compute $res_i = max(a_i, b_i)$ for each datum a_i in tensor a and b_i in tensor b, where res_i, a_i and b_i are of same index. All tensors must be of same shape, and the basic data in all *tensor_lmem* structures must be 8-bit. Tensor a and tensor b must both be signed or unsigned at the same time.

```
typedef struct {
  tensor_lmem *max;
  tensor_lmem *a;
  tensor_lmem *b;
} bmk1880_max_param_t;

bmk1880_op_t * bmk1880_tpu_max(
  bmk1880_context_t *ctx,
  const bmk1880_max_param_t *p);
```

4.27 bmk1880_tpu_min

Similar to *bmk1880_tpu_max*, but computes $res_i = min(a_i, b_i)$.

```
typedef struct {
  tensor_lmem *min;
  tensor_lmem *a;
  tensor_lmem *b;
} bmk1880_min_param_t;

bmk1880_op_t * bmk1880_tpu_min(
  bmk1880_context_t *ctx,
  const bmk1880_min_param_t *p);
```

4.28 bmk1880_tpu_min_const

Similar to *bmk1880_tpu_min*, but tensor b is replaced by an 8-bit constant. The constant is signed if *b_is_signed* is `true`, unsigned otherwise.

```
typedef struct {
  tensor_lmem *min;
  tensor_lmem *a;
  s8 b;
  bool b_is_signed;
} bmk1880_min_const_param_t;

bmk1880_op_t * bmk1880_tpu_min_const(
  bmk1880_context_t *ctx,
  const bmk1880_min_const_param_t *p);
```

4.29 bmk1880_tpu_arith_shift

bmk1880_tpu_arith_shift instructs TPU to compute $res_i = a_i \gg bits_i$ for each datum a_i in tensor a and $bits_i$ in tensor $bits$, where res_i, a_i and $bits_i$ are of same index. All tensors must be of same shape, and the basic data in all *tensor_lmem* structures must be 8-bit. Tensor a must be

16-bit and signed so that the *fmt* fields in *a_high* and *a_low* must be `FMT_I8`. Tensor *bits* must be signed and every datum in it must range in $[-16, 16]$. The result tensor must be 16-bit so that *res_high* must be non-`NULL`.

```
typedef struct {
  tensor_lmem *res_high;
  tensor_lmem *res_low;
  tensor_lmem *a_high;
  tensor_lmem *a_low;
  tensor_lmem *bits;
} bmk1880_arith_shift_param_t;

bmk1880_op_t * bmk1880_tpu_arith_shift(
  bmk1880_context_t *ctx,
  const bmk1880_arith_shift_param_t *p);
```

4.30 bmk1880_tl_and_int8

bmk1880_tpu_and_int8 instructs TPU to compute $res_i = a_i \wedge b_i$ for each datum a_i in tensor a and b_i in tensor b, where res_i, a_i and b_i are of same index. All tensors must be of same shape, and the basic data in all *tensor_lmem* structures must be 8-bit.

```
typedef struct {
  tensor_lmem *res;
  tensor_lmem *a;
  tensor_lmem *b;
} bmk1880_and_int8_param_t;

bmk1880_op_t * bmk1880_tpu_and_int8(
  bmk1880_context_t *ctx,
  const bmk1880_and_int8_param_t *p);
```

4.31 bmk1880_tpu_and_int16

Similar to *bmk1880_tpu_and_int8*, but all input and output tensors are 16-bit. So *res_high*, *a_high* and *b_high* must be non-`NULL`.

```
typedef struct {
  tensor_lmem *res_high;
  tensor_lmem *res_low;
  tensor_lmem *a_high;
  tensor_lmem *a_low;
  tensor_lmem *b_high;
  tensor_lmem *b_low;
} bmk1880_and_int16_param_t;

bmk1880_op_t * bmk1880_tpu_and_int16(
  bmk1880_context_t *ctx,
  const bmk1880_and_int16_param_t *p);
```

4.32 bmk1880_tpu_or_int8

Similar to *bmk1880_tpu_and_int8*, but computes $res_i = a_i \vee b_i$.

```
typedef struct {
  tensor_lmem *res;
  tensor_lmem *a;
  tensor_lmem *b;
} bmk1880_or_int8_param_t;

bmk1880_op_t * bmk1880_tpu_or_int8(
  bmk1880_context_t *ctx,
  const bmk1880_or_int8_param_t *p);
```

4.33 bmk1880_tpu_or_int16

Similar to *bmk1880_tpu_and_int16*, but computes $res_i = a_i \vee b_i$.

```
typedef struct {
  tensor_lmem *res_high;
  tensor_lmem *res_low;
  tensor_lmem *a_high;
  tensor_lmem *a_low;
  tensor_lmem *b_high;
  tensor_lmem *b_low;
} bmk1880_or_int16_param_t;

bmk1880_op_t * bmk1880_tpu_or_int16(
  bmk1880_context_t *ctx,
  const bmk1880_or_int16_param_t *p);
```

4.34 bmk1880_tpu_xor_int8

Similar to *bmk1880_tpu_and_int8*, but computes $res_i = a_i \oplus b_i$.

```
typedef struct {
  tensor_lmem *res;
  tensor_lmem *a;
  tensor_lmem *b;
} bmk1880_xor_int8_param_t;

bmk1880_op_t * bmk1880_tpu_xor_int8(
  bmk1880_context_t *ctx,
  const bmk1880_xor_int8_param_t *p);
```

4.35 bmk1880_tpu_xor_int16

Similar to *bmk1880_tpu_and_int16*, but computes $res_i = a_i \oplus b_i$.

```
typedef struct {
  tensor_lmem *res_high;
  tensor_lmem *res_low;
  tensor_lmem *a_high;
  tensor_lmem *a_low;
  tensor_lmem *b_high;
  tensor_lmem *b_low;
} bmk1880_xor_int16_param_t;

bmk1880_op_t * bmk1880_tpu_xor_int16(
  bmk1880_context_t *ctx,
  const bmk1880_xor_int16_param_t *p);
```

4.36 bmk1880_tpu_copy

bmk1880_tpu_copy instructs TPU to copy tensors within lmem, from *src* to *dst*. The basic data must be 8-bit.

```
typedef struct {
  tensor_lmem *dst;
  tensor_lmem *src;
} bmk1880_copy_param_t;

bmk1880_op_t * bmk1880_tpu_copy(
  bmk1880_context_t *ctx,
  const bmk1880_copy_param_t *p);
```

4.37 bmk1880_tpu_copy_with_stride

Similar to *bmk1880_tpu_copy*, but user provides *stride_t* structures specifying the layouts of tensors *dst* and *src*. The basic data must be 8-bit.

```
typedef struct {
  tensor_lmem *dst;
  stride_t dst_stride;
  tensor_lmem *src;
  stride_t src_stride;
} bmk1880_copy_with_stride_param_t;

bmk1880_op_t * bmk1880_tpu_copy_with_stride(
  bmk1880_context_t *ctx,
  const bmk1880_copy_with_stride_param_t *p);
```

4.38 bmk1880_tpu_mdsum

bmk1880_tpu_mdsum instructs TPU to compute a tensor *res* of shape $(1, C, 1, 1)$ from tensor a of shape (N, C, H, W). Every datum res_{c_i} of index $(0, c_i, 0, 0)$ in tensor *res* is computed as $res_{c_i} = \sum_{n_i=0}^{N-1} \sum_{h_i=0}^{H-1} \sum_{w_i=0}^{W-1} a_{n_i,c_i,h_i,w_i}$, where a_{n_i,c_i,h_i,w_i} is a datum of index (n_i, c_i, h_i, w_i) in tensor a. The basic data in all *tensor_lmem* structures must be 8-bit. If the result is a 16-bit tensor, *res_high* and *res_low* represent its high and low 8-bit parts, respectively. *res_high* should be `NULL` if the result is 8-bit. a and res must both be signed or unsigned at the same time.

```
typedef struct {
  tensor_lmem *res_high;
  tensor_lmem *res_low;
  tensor_lmem *a;
} bmk1880_mdsum_param_t;

bmk1880_op_t * bmk1880_tpu_mdsum(
  bmk1880_context_t *ctx,
  const bmk1880_mdsum_param_t *p);
```

4.39 bmk1880_tpu_lut

bmk1880_tpu_lut instructs TPU to compute a tensor res from tensor idx, by using tensor $table$ as a lookup table and values in tensor idx as indice. Tensor $table$ must be of shape $(1, slices, 16, 16)$, where $slices$ is the number of lmem slices. Tensor idx and tensor res must be of same shape. Assuming their shape is (N, C, H, W), the datum res_i of index (n_i, c_i, h_i, w_i) in tensor res is computed from idx_i of same index (n_i, c_i, h_i, w_i) in tensor idx as $res_i = table_i$, where $table_i$ is of index $(0, c_t, \frac{idx_i}{16}, idx_i \bmod 16)$ in tensor $table$, and c_t is the index of lmem slice the datum idx_i resides in. The basic data in all *tensor_lmem* structures must be 8-bit.

```
typedef struct {
  tensor_lmem *ofmap;
  tensor_lmem *ifmap;
  tensor_lmem *table;
} bmk1880_lut_param_t;

bmk1880_op_t * bmk1880_tpu_lut(
  bmk1880_context_t *ctx,
  const bmk1880_lut_param_t *p);
```

4.40 bmk1880_tpu_relu

bmk1880_tpu_relu instructs TPU to compute $res_i = max(0, a_i)$ for each datum a_i in tensor a, where res_i and a_i are of same index. The basic data in all *tensor_lmem* structures must be 8-bit.

```
typedef struct {
  tensor_lmem *ofmap;
  tensor_lmem *ifmap;
} bmk1880_relu_param_t;

bmk1880_op_t * bmk1880_tpu_relu(
  bmk1880_context_t *ctx,
  const bmk1880_relu_param_t *p);
```

4.41 bmk1880_tpu_conv

bmk1880_tpu_conv instructs TPU to compute a tensor *ofmap* from tensor *ifmap*, *weight* and *bias*, by using *ifmap* as input feature map, *weight* as convolution kernel and *bias* as bias to be added into the convolution result. *relu_enable* may be `true`, indicating ReLU activations after adding bias values but before shifting every basic datum. *rshift_width* specifies the number of bits to shift every basic datum rightward after optional ReLU activations.

```
typedef struct {
  tensor_lmem *ofmap;
  tensor_lmem *ifmap;
  tensor_lmem *weight;
  tensor_lmem *bias;
  u8 ins_h, ins_last_h;
  u8 ins_w, ins_last_w;
  u8 pad_top, pad_bottom;
  u8 pad_left, pad_right;
  u8 stride_h, stride_w;
  u8 dilation_h, dilation_w;
  bool relu_enable;
  int rshift_width;
} bmk1880_conv_param_t;

bmk1880_op_t * bmk1880_tpu_conv(
  bmk1880_context_t *ctx,
  const bmk1880_conv_param_t *p);
```

ofmap and *ifmap* must be *aligned* (see *aligned* stride values in section 2.4).

weight is of a special layout which is very different from that described in section 2.3. If *ifmap* is of shape $(N_{in}, C_{in}, H_{in}, W_{in})$, *ofmap* is of shape $(N_{out}, C_{out}, H_{out}, W_{out})$ and convolution kernels are of shape (H_{kernel}, W_{kernel}), then *weight* should be of shape $(C_{in}, C_{out}, H_{kernel}, W_{kernel})$. The layout of *weight*, however, is as if it is of shape $(1, C_{out}, H_{kernel} \times W_{kernel}, C_{in})$. This special layout can be precisely defined by applying the following stride values to *weight*'s logical shape $(C_{in}, C_{out}, H_{kernel}, W_{kernel})$:

$$\begin{cases} S_{C_{in}} &= 1 \\ S_{C_{out}} &= C_{in} \times (H_{kernel} \times W_{kernel}) \\ S_{H_{kernel}} &= C_{in} \times W_{kernel} \\ S_{W_{kernel}} &= C_{in} \end{cases}$$

bias may be `NULL`, indicating no bias values. If it is non-`NULL`, and assume *ofmap* is of shape (N, C, H, W), then *bias* must be a 16-bit tensor of shape $(1, C, 1, 1)$. Since a 16-bit tensor is stored as two 8-bit tensors in lmem, *bias*'s *tensor_lmem* structure must be of shape $(2, C, 1, 1)$ and must be *unaligned* (see *unaligned* stride values in section 2.4). During the phase of adding bias, the value of datum at index $(0, c_i, 0, 0)$ in the 16-bit tensor are added to all data in *ofmap* whose C-dimension index is c_i.

param contains detailed convolution parameters that can be classified into four categories by their functions. They are insertion, padding, striding and dilations parameters, which are detailed below. Insertion parameters specify the number of zeros to be inserted into specific locations within *ifmap*. They include *ins_h*, *ins_last_h*, *ins_w* and *ins_last_w*. *ins_h* specifies the number of zeros to be inserted after every non-last basic datum, along the H-dimension. Consider *ifmap* of shape (N, C, H, W) for example. After inserting zeros, *ifmap*$'$ will be of shape (N, C, H', W), where $H' = 1 + (H - 1) \times (ins\_h + 1)$. Denoting as x_{n_i,c_i,h_i,w_i} the value of basic datum at index (n_i, c_i, h_i, w_i) of tensor *ifmap*, and as x'_{n_i,c_i,h_i,w_i} the value of that of tensor *ifmap*$'$, the following holds:

$$x'_{n_i,c_i,h_i,w_i} = \begin{cases} x_{n_i,c_i,\frac{h_i}{ins\_h+1},w_i} & \text{if } h_i \text{ is a multiple of } (ins\_h + 1), \\ 0 & \text{otherwise.} \end{cases}$$

ins_last_h specifies the number of zeros to be inserted only after every last basic datum. Similarly, *ins_w* and *ins_last_w* specify the number of zeros to be inserted along the W-dimension. Padding parameters specify the number of zeros to be inserted around elements within *ifmap*. *pad_top* specifies the number of zeros to be inserted before every first basic datum along the H-dimension. *pad_bottom* specifies the number after every last basic datum along the H-dimension. Similary, *pad_left* and *pad_right* specify the number along the W-dimension. Striding parameters specify the number of basic data convolution kernel should stride over after each convolution step. *stride_h* and *stride_w* specify the number along the H-dimension and W-dimension, respectively. Dilation parameters specify the dilation of the convolution kernel *weight*. That is, $(stride\_h - 1)$ zeros are inserted between each two basic data along the H-dimension. Similary $(stride\_w - 1)$ zeros are inserted along the W-dimension.

4.42 bmk1880_tpu_winograd

Similar to *bmk1880_tpu_conv*, but use winograd algorithm to accelerate the computation. Moreover, *weight* must contain only 3×3 kernels and must be default strided in lmem (see section 2.4). The other parameters, including those in *param*, are similar to those of same names in function *bmk1880_tpu_conv* (see section 4.41).

```
typedef struct {
  tensor_lmem *ofmap;
  tensor_lmem *ifmap;
  tensor_lmem *weight;
  tensor_lmem *bias;
  u8 pad_top, pad_bottom;
  u8 pad_left, pad_right;
  bool relu_enable;
  int rshift_width;
} bmk1880_winograd_param_t;

bmk1880_op_t * bmk1880_tpu_winograd(
  bmk1880_context_t *ctx,
  const bmk1880_winograd_param_t *p);
```

4.43 bmk1880_tpu_depthwise

Similar to *bmk1880_tpu_conv*, but computes a depthwise convolution. Moreover, *weight* is default strided in lmem (see section 2.4). The other parameters, including those in *param*, are similar to those of same names in function *bmk1880_tpu_conv* (see section 4.41).

```
typedef struct {
  tensor_lmem *ofmap;
  tensor_lmem *ifmap;
  tensor_lmem *weight;
  tensor_lmem *bias;
  u8 ins_h, ins_last_h;
  u8 ins_w, ins_last_w;
  u8 pad_top, pad_bottom;
  u8 pad_left, pad_right;
  u8 stride_h, stride_w;
  int rshift_width;
} bmk1880_depthwise_param_t;

bmk1880_op_t * bmk1880_tpu_depthwise(
  bmk1880_context_t *ctx,
  const bmk1880_depthwise_param_t *p);
```

4.44 bmk1880_tpu_max_pooling

bmk1880_tpu_max_pooling instructs TPU to compute a tensor *ofmap* from tensor *ifmap*, by doing a ($kh \times kw$) max pooling over *ifmap*. The size parameters of pooling kernel, *kh* and *kw*, are specified in *param*. Other parameters in *param* are similar to those of same names in *bmk1880_conv_param_t* (see section 4.41). *ofmap* and *ifmap* must be *aligned* (see *aligned* stride values in section 2.4).

```
typedef struct {
  tensor_lmem *ofmap;
  tensor_lmem *ifmap;
  u8 kh, kw;
  u8 pad_top, pad_bottom;
  u8 pad_left, pad_right;
  u8 stride_h, stride_w;
} bmk1880_max_pooling_param_t;

bmk1880_op_t * bmk1880_tpu_max_pooling(
  bmk1880_context_t *ctx,
  const bmk1880_max_pooling_param_t *p);
```

4.45 bmk1880_tpu_avg_pooling

Similar to *bmk1880_tpu_max_pooling*, but does an average pooling over *ifmap* as controlled by *avg_pooling_const*. At every pooling step, all related basic data in *ifmap* are summed together, multiplied by *avg_pooling_const*, and then shifted rightward by *rshift_width* bits.

```
typedef struct {
  tensor_lmem *ofmap;
  tensor_lmem *ifmap;
  u8 kh, kw;
  u8 ins_h, ins_last_h;
  u8 ins_w, ins_last_w;
  u8 pad_top, pad_bottom;
  u8 pad_left, pad_right;
  u8 stride_h, stride_w;
  u8 avg_pooling_const;
  int rshift_width;
} bmk1880_avg_pooling_param_t;

bmk1880_op_t * bmk1880_tpu_avg_pooling(
  bmk1880_context_t *ctx,
  const bmk1880_avg_pooling_param_t *p);
```

4.46 bmk1880_tpu_matrix_mac

bmk1880_tpu_matrix_mac instructs TPU to compute a matrix *res* by multiplying left matrix *left* with right matrix *right*, and then add matrix *bias* (if not `NULL`), and finally shift to right by *rshift_width* bits. Noth that all *tensor_lmem* structures involved must be matrice instead of tensors. *ctrls* may have `CTRL_RELU` or `CTRL_RA` flag set, but not both. After adding *bias* but before right shifting, ReLU activations are performed in which negative values are rectified to 0 if *ctrls* is `CTRL_RELU`, or the original values in *res* are shifted leftward by *lshift_width* bits and then added into the results if *ctrls* is `CTRL_RA`. *res_is_int8* indicates whether the result is 8-bit or 16-bit.

The use of *res* matrix is unusual when *ctrls* is `CTRL_RA` or when *res_is_int8* is `false`. Assume that the result is a matrix of shape (R, C). When *ctrls* is `CTRL_RA`, the original result is a 16-bit matrix of shape (R, C) represented by *res*. Since a 16-bit matrix's high and low 8-bit parts are stored separately as two 8-bit matrice in lmem, *res*'s *tensor_lmem* structure must be of 8-bit format (`FMT_I8` or `FMT_U8`), must be of shape $(R \times 2, C)$, and must be *aligned* (see *aligned* stride values in section 2.4). When *res_is_int8* is `false`, the final result is a 16-bit matrix similarly represented by *res*. When *ctrls* is `CTRL_RA` but *res_is_int8* is `true`, the original result is 16-bit while the final result is 8-bit. In this case, only the low 8-bit parts (located at lower addresses) of the *res* matrix are written with the final result. In the final case where both the original and final result are 8-bit matrice, *res* is a normal 8-bit matrix of shape (R, C).

Note that *bias* is different from those in *bmk1880_tpu_conv*, *bmk1880_winograd* or *bmk1880_tpu_depthwise*. Firstly, it is a matrix. Moreover, if *res* is of shape (R, C), then *bias* must be a 16-bit matrix of shape $(1, C)$. Since a 16-bit matrix's high and low 8-bit parts are stored separately as two 8-bit matrice in lmem, *bias*'s *tensor_lmem* structure must be of shape $(2, C)$ and must be *aligned* (see *aligned* stride values in section 2.4).

res, *left*, *right* and *bias* must all be *aligned* (see *aligned* stride values in section 2.4).

```
typedef struct {
  tensor_lmem *res;
  tensor_lmem *left;
  tensor_lmem *right;
  tensor_lmem *bias;
  int lshift_width;
  int rshift_width;
  bool res_is_int8;
  ctrl_t ctrls;
} bmk1880_matrix_mac_param_t;

bmk1880_op_t * bmk1880_tpu_matrix_mac(
  bmk1880_context_t *ctx,
  const bmk1880_matrix_mac_param_t *p);
```

4.47 bmk1880_tpu_matrix_mac_2

bmk1880_tpu_matrix_mac_2 instructs TPU to compute a matrix *res* by multiplying left matrix *left* with right matrix *right*. *left*, *right* and *res* must be tensors, though the computation is matrix multiplication. *res* and *left* must be of shape $(1, 256, 1, 256)$. *right* must be of shape $(256, 16, 1, 16)$. The basic data in all *tensor_lmem* structures must be 8-bit.

```
typedef struct {
  tensor_lmem *res;
  tensor_lmem *left;
  tensor_lmem *right;
} bmk1880_matrix_mac_2_param_t;

bmk1880_op_t * bmk1880_tpu_matrix_mac_2(
  bmk1880_context_t *ctx,
  const bmk1880_matrix_mac_2_param_t *p);
```

Appendix A

Samples

> Beware of bugs in the above code. I have only proved it correct, not tried it.
>
> Donald E. Knuth

A.1 Print Generated Bytes

```c
#include <bm_kernel.h>
#include <stdlib.h>
#include <assert.h>

void print_cmdbuf(u8 *cmdbuf, u32 size)
{
  printf("%s: the emitted bytes are:\n", __func__);

  char *p = (char *)cmdbuf;
  for (u32 i = 0; i < size; i += 16) {
    printf("\t");
    for (u32 j = 0; j < size - i && j < 16; j++) {
      printf("%02x ", p[i + j]);
    }
    printf("\n");
  }
}

int main()
{
  bmk1880_info_t info;
  info.chip_version = 1880;
  info.cmdbuf_size = 4096;
  info.cmdbuf = (u8 *)malloc(info.cmdbuf_size);
  assert(info.cmdbuf);

  bmk1880_context_t *ctx = bmk1880_register(&info);
  bmk1880_enter(ctx);

  u32 size;
  u8 *cmdbuf = bmk1880_acquire_cmdbuf(&size);
  print_cmdbuf(cmdbuf, size);

  bmk1880_exit();
  bmk1880_cleanup(ctx);

  free(info.cmdbuf);
  return 0;
}
```

A.2 The Effect of bmk1880_reset

```
#include <bm_kernel.h>
#include <stdlib.h>
#include <assert.h>

void print_cmdbuf(u8 *cmdbuf, u32 size)
{
  printf("%s: the emitted bytes are:\n", __func__);

  char *p = (char *)cmdbuf;
  for (u32 i = 0; i < size; i += 16) {
    printf("\t");
    for (u32 j = 0; j < size - i && j < 16; j++) {
      printf("%02x ", p[i + j]);
    }
    printf("\n");
  }
}

void do_print()
{
  u32 size;
  u8 *cmdbuf = bmk1880_acquire_cmdbuf(&size);
  print_cmdbuf(cmdbuf, size);
}

int do_compute()
{
  shape_t s = shape_t4(1, 32, 8, 8);
  tensor_lmem *t = bmk1880_tl_prealloc(0, s, FMT_I8);
  bmk1880_tl_load(t, 0x1000, CTRL_NULL);
}

int main()
{
  bmk1880_info_t info;
  info.chip_version = 1880;
  info.cmdbuf_size = 4096;
  info.cmdbuf = (u8 *)malloc(info.cmdbuf_size);
  assert(info.cmdbuf);

  bmk1880_context_t *ctx = bmk1880_register(&info);

  bmk1880_enter(ctx);

  do_compute();
  do_print();
  bmk1880_reset();
  do_print();

  bmk1880_exit();
  bmk1880_cleanup(ctx);

  free(info.cmdbuf);
  return 0;
}
```

A.3 Multiple Contexts

```
#include <bm_kernel.h>
#include <stdlib.h>
#include <assert.h>

void print_cmdbuf(u8 *cmdbuf, u32 size)
{
  printf("%s: the emitted bytes are:\n", __func__);

  char *p = (char *)cmdbuf;
  for (u32 i = 0; i < size; i += 16) {
    printf("\t");
    for (u32 j = 0; j < size - i && j < 16; j++) {
      printf("%02x ", p[i + j]);
    }
    printf("\n");
  }
}

void do_print()
{
  u32 size;
  u8 *cmdbuf = bmk1880_acquire_cmdbuf(&size);
  print_cmdbuf(cmdbuf, size);
}

int do_compute()
{
  shape_t s = shape_t4(1, 32, 8, 8);
  tensor_lmem *t = bmk1880_tl_prealloc(0, s, FMT_I8);
  bmk1880_gdma_load(t, 0x1000, CTRL_NULL);
}
```

```
int main()
{
  u32 cmdbuf_size = 4096;

  bmk1880_info_t info_0;
  info_0.chip_version = 1880;
  info_0.cmdbuf_size = cmdbuf_size;
  info_0.cmdbuf = (u8 *)malloc(cmdbuf_size);
  assert(info_0.cmdbuf);

  bmk1880_info_t info_1;
  info_1.chip_version = 1880;
  info_1.cmdbuf_size = cmdbuf_size;
  info_1.cmdbuf = (u8 *)malloc(cmdbuf_size);
  assert(info_1.cmdbuf);

  bmk1880_context_t *ctx_0 = bmk1880_register(&info_0);
  bmk1880_context_t *ctx_1 = bmk1880_register(&info_1);

  bmk1880_enter(ctx_0);
  do_compute();
  do_print();
  bmk1880_exit();

  bmk1880_enter(ctx_1);
  do_compute();
  do_print();
  bmk1880_exit();

  bmk1880_cleanup(ctx_0);
  bmk1880_cleanup(ctx_1);
  free(info_0.cmdbuf);
  free(info_1.cmdbuf);
  return 0;
}
```