ztachip
Programmer's Guide
v1.0

Author: Vuong Nguyen
https://github.com/ztachip/ztachip
vuongdnguyen@hotmail.com

Table of Contents

1-Introduction

Domain-Specific-Architecture (DSA) is a general trend in computing research in-order to keep up with the future computing demand fuelled by the exponential growth of AI.

With DSA, the goal is to define a domain of applications where a special computing architecture will be more efficient compared to a more general-purpose computing architecture.

ztachip domain are applications that can be expressed as a sequential steps of tensor operations. There are 2 primary types of tensor operations defined:

- Tensor data operations: Involving tensor operations such as tensor data copy, tensor transpose, dimensions resize, data remapping...

- Tensor operator operations: Performing computational tasks on a set of tensors.

But to be productive, DSA would also require a Special-Domain-Language (DSL) to abstract away the complexities of DSA.

The goal for DSL is to provide a programming language that is:

- Easy to use and learn.

- The language hides the complexity of hardware implementation from software users.

- Flexible enough to cover as wide range of applications as possible.

ztachip's DSL is composed of 2 elements:

- tensor programs

- pcore programs

1.1 tensor programs

Tensor programs are codes that run on host processor such as RISCV. It generates and sends proprietary tensor instructions to ztachip's master processor called tensor-core.

Tensor programs have suffix *.m

Tensor programs are primarily C-programs but with some special extensions. There is a special ztachip

compiler that converts these special extensions to tensor instructions before the program can then be compiled with standard RISCV C/C++ compiler.

With tensor programs, programmers express the problems as a sequence of high-level tensor instructions. Computational tasks are presented as tensor operators that will be defined later by pcore programs.

tensor programs can be further executed on 2 separate SW threads with each SW thread emitting tensor instructions to one of the 2 hardware threads called t-threads available from tensor-core.

t-threads improve performance since they allow for tensor data operations from one t-thread to be overlapped with tensor operator operations from the other t-thread.

1.2 pcore programs

pcore programs implement the tensor operators that can be invoked by tensor programs.

pcore programs are executed on an array of ztachip's proprietary VLIW processors called p-cores. VLIW stands for very-long-instruction-word where each instruction can perform many operations at the same time.

pcore programs are files with suffix *.p

pcore programs are multi-threaded with 16 threads. These threads are hardware-based threads without overhead and dispatched in a round-robin fashion. We call these threads to be p-threads.

Tensor operators are presented as operators of C++ object class. Related tensor operators are then grouped together under the same C++ object class. These object classes are single instance objects so they are statically allocated. Due to limited memory available, these objects are overlapped in memory allocation. Therefore, users must partition these objects in such a way that they are not in use at the same time.

1.3 Compare ztachip DSL with traditional programming

With traditional programming, data and execution steps are often interleaved.
Traditional architecture requires data to be loaded first from memory to on-chip registers before computational steps can take place. But this would create a lot of memory round trip delay and stall cycles especially when data is not readily available in cache.

But with ztachip, computation and data transfer with external memory are decoupled from one another. With ztachip, data are loaded/saved between external memory and pcore's internal memory in a streaming fashion with advanced data fetching and no round-trip delay. This results in huge gain in memory bandwidth usage efficiencies. Computational steps are also very efficient without any memory stall cycles since computation reference pcore's internal memory only.

Also, computational steps being represented as tensor operators provide a natural way to express algorithm parallelism where many computational units can perform similar computation on many different elements of tensors.

2-TENSOR PROGRAMS

Tensor programs are C-codes that are executed on host processor such as RISCV.

Tensor program emits tensor instructions to the tensor-core. Tensor instructions perform functions such as

- tensor memory operations such as data copy, resize, reshape...
- Dispatching tensor operator executions defined by p-core programs.

Tensor programs are C programs with special embedded tensor syntax extensions. Tensor extensions are lines that begin with character '>'.
ztachip special compiler replaces these special extensions with appropriate tensor instruction binary that are then being pushed to tensor processor's instruction queue.

There can be up to 2 hardware-based threads running in the tensor core. We call these t-threads.

t-threads are useful in interleaving memory operation cycle from one t-thread with tensor operator execution cycle from the other t-thread. This greatly improves memory bandwidth efficiencies.

2.1 Tensor data object definition

Tensor data objects can be viewed as data variables in tensor programming paradigm.

Tensor data objects are often defined as a sub-tensor of another tensor object. Sub-tensors are defined by specifying index ranges of tensor dimensions.

Tensors can be resided in

- p-core private memory space. In this space, each of p-core's 16 p-threads has its own private memory.
- p-core shared memory space. In this space, all of p-core's 16 p-threads shared the same memory space.
- p-core memory space is further partitioned into 2 separate spaces with each space assigned to each of the two t-threads from tensor-core.
- external DDR memory space.

2.1.1 Tensor data objects in DDR memory space

External DDR tensor has the syntax below. It specifies a tensor as a sub-tensor of another tensor by specifying its dimension index range.

```
DDR(pointer,dim,dim,...)[start:stride:end]...[start:stride:end]
```

pointer	memory address where tensor is stored in DDR memory
dim	tensor dimensions. If omitted than tensor is 1-D spanning the entire DDR memory range.
start	starting index of dimension range.
stride	stride of the dimension range.
end	ending index of dimension range.

2.1.1.1 Example 1

Example below specifies a tensor that is a sub-set of a tensor 100x200 resided at memory location p. The tensor subset has dimension 20x10 spanning row 0-19 and column 20-29 of original tensor.

```
DDR(p,100,200)[0:1:19][20:1:29]
```

2.1.1.2 Example 2

If stride of index range is ignored, then 1 is assumed
Tensor definition below is identical to the one defined in example 1.

```
DDR(p,100,200)[0:19][20:29]
```

2.1.1.3 Example 3

If begin of index range is ignored, then 0 is assumed
Tensor definition below is identical to the one defined in example 1.

```
DDR(p,100,200)[:19][:29]
```

2.1.1.4 Example 4

If end of index range is ignored, then end of dimension range is assumed.
In example below, tensor spans from row index=0 to 99 and spans from column index=20 to 199.

```
DDR(p,100,200)[0:][20:]
```

2.1.2 Tensor data objects in PCORE private memory space

Tensors that are allocated in p-core private memory space have the following syntax

```
PCORE(dim,...)
     [start:stride:end]...[start:stride:end]
     .thread[start:stride:end]
     .class::variable[start:stride:end]...[start:stride:end]
```

dim	p-cores can be arranged as an 1-D array or 2D array. If dim is not specified, then a 1-D arrangement of all available p-cores is assumed.
start	starting index of dimension range.
stride	stride of the dimension range.
end	ending index of dimension range.
thread	referencing the 16 p-threads.
class	class name of variable. variables in pcore's memory space are grouped together by class object.
variable	variable holding tensor data in pcore memory space.

\

2.1.2.1 Example 1

Example below, pcores are arranged as 1-D array of VLIW processors. The tensor spans all 8 p-cores. And at each p-core, we span all 16 threads. And at each thread we reference all elements of variable class::private_variable that is defined by p-core programs as a 2x8 array.

```
PCORE(8)[0:7].thread[0:15].class::private_variable[0:1][0:7]
```

2.1.2.2 Example 2

Example below is similar to previous example except for p-cores being arranged as 4x2 array of processors. Below we span all pcores by row and column indexing.

```
PCORE(4,2)[0:3][0:1].thread[0:15].class::private_variable[0:1][0:7]
```

2.1.3 Tensor data objects in p-core shared memory space

Tensors that are allocated in p-core shared memory space are similar to
tensor in p-core private memory space except that the thread dimension is now omitted.

```
PCORE(dim,...)
     [start:stride:end]...[start:stride:end]
     .class::variable[start:stride:end]...[start:stride:end]
```

dim	p-cores can be arranged as an 1-D array or 2D array. If dim is not specified, then a 1-D arrangement of all available p-cores is assumed.
start	starting index of dimension range.
stride	stride of the dimension range.
end	ending index of dimension range.
class	class name of variable. variables in pcore's memory space are grouped together by class object.
variable	variable holding tensor data in pcore memory space.

2.1.3.1 Example 1

Example below, pcores are arranged as 1-D array of processors. The referenced tensor spans all 8 p-cores. And at each pcore we reference all elements of variable class::shared_variable that is defined in p-core programs as a 2x8 array. Note that we don't span the thread dimension since shared memory space has no thread dimension.

```
PCORE(8)[0:7].class::shared_variable[0:1][0:7]
```

2.2 Tensor variables.

Tensor variables are tensor alias. They are faster to reference in tensor programs.

Tensor variable name must start with character '$'.

Example below is an example of a tensor variable declaration and definition.

```
// First declare the variable
> VAR $coef_pcore;

// Then assign variable with a tensor definition.
> $coef_pcore := PCORE(8)[0:7].convolution::coef[0:1][0:7];
\
```

Whereever $coef_pcore being referenced, the expression representing the variable will be substituted in its place.

Tensor variables can also have an argument. The argument is specified when tensor variables are being referenced. '$' is tensor variable's argument place holder.

2.2.1 Tensor variables - example 1

If a tensor variable $coef_ddr is defined as

```
> VAR $coef_pcore;
> $coef_ddr := MEM(coef,100,200)[$][0:199];
```

Then

```
$coef_ddr[jj]
```

is the same as below after substituting tensor variable argument $ with jj.

```
MEM(coef,100,200)[jj][0:199];
```

2.2.2 Tensor variables – example 2

If a tensor variable $coef_ddr is defined as

```
> VAR $coef_pcore;
> $coef_ddr := MEM(coef,100,200)[$][0:199];
```

Then

```
$coef_ddr[0:19]
```

is the same as below after substituting tensor variable argument $ with [0:19].

```
MEM(coef,100,200)[0:19][0:199];
```

2.3 Tensor data transfer instructions

These instructions are inserted to tensor programs with special syntax line begins with '>'

These instructions move tensors from one memory space to another.

In addition to tensor memory transfer, these instructions can also perform many functions such as tensor reshape, dimension resize, data remapping...

2.3.1 Tensor data transfer from DDR to PCORE private memory space

```
>PCORE[0:1].THREAD[0:1].myclass::var[0:1] <= DDR(p)[0:7];
```

Example above transfers data from a tensor resided in external DDR memory to a tensor resided in pcore's private memory space named myclass::var.

The result of the transfer is shown below

```
PCORE[0].THREAD[0].myclass::var[0] = p[0]
PCORE[0].THREAD[0].myclass::var[1] = p[1]
PCORE[0].THREAD[1].myclass::var[0] = p[2]
PCORE[0].THREAD[1].myclass::var[1] = p[3]
PCORE[1].THREAD[0].myclass::var[0] = p[4]
PCORE[1].THREAD[0].myclass::var[1] = p[5]
PCORE[1].THREAD[1].myclass::var[0] = p[6]
PCORE[1].THREAD[1].myclass::var[1] = p[7]
```

2.3.2 Tensor data transfer from DDR to PCORE shared memory space.

```
>PCORE[0:1].myclass::shared_var[0:1] <= DDR(p)[0:3];
```

Example above transfers data from a tensor resided in external DDR memory to a tensor resided in pcore's shared memory space named myclass::shared_var. Tensors in PCORE shared memory space have no thread dimension.

The result of the transfer is shown below

```
PCORE[0].myclass::shared_var[0] = p[0]
PCORE[0].myclass::shared_var[1] = p[1]
```

```
PCORE[1].myclass::shared_var[0] = p[2]
PCORE[1].myclass::shared_var[1] = p[3]
```

2.3.3 Tensor data transfer from DDR to tensor operator's parameters

Like C/C++ functions, tensor operators are functions with function parameters declared.

Tensor operator parameters are themselves tensors that reside in pcore private memory space.

Parameters of different operators are overlapped in memory allocation. Therefore, tensor operator parameter's content will be overwritten when a different tensor operator being invoked.

Tensor operator parameters will be explained at more details in chapter 3.

Example1:

```
>PCORE(4)[0:3].THREAD[0:15].myclass::my_op.parm <= DDR(p)[0:63];
```

Example above set parameter parm of tensor operator my_class::my_op with data from DDR memory. parm1 is viewed as a tensor of dimension 4x16

Example2:

```
>PCORE(4)[0:3].THREAD[0:15].myclass::my_op.vparm[0:7] <= DDR(p)[0:511];
```

Example above set parameter parm1 of tensor operator myclass::my_op with data from DDR memory. In this example, in the pcore program, vparm is declared as a vector of width 8, therefore parm1 is viewed as a tensor of dimension 4x16x8

2.3.4 Tensor data transfer from tensor operator's parameters to DDR

Similar to previous section, but now the transfer direction is from tensor operator parameters to DDR memory space.

Tensor operator parameters will be explained at more details in chapter 3.

Example1:

```
>DDR(p)[0:63] <= PCORE(4)[0:3].THREAD[0:15].myclass::my_op.z;
```

Example above transfers parameter z of tensor operator myclass::my_op to DDR memory.
In this example, z is viewed as a tensor of dimension 4x16

Example2:

```
>DDR(p)[0:511] <= PCORE(4)[0:3].THREAD[0:15].myclass::my_op.vz[0:7];
```

Example above transfers tensor operator parameter vz to DDR memory.
In this example, in the pcore program, vz is declared as vector with width 8, then vz is viewed as a tensor of dimension 4x16x8

2.3.5 Setting tensor operator's global parameters

Tensor operator parameters are normally unique and private among pcores and p-threads.

However, tensor operator parameters can be specified to have a global scope.

With global scope, all pcores and all p-threads would share the same parameter.

Tensor operator parameters will be explained at more details in chapter 3.

```
>myclass::my_op.kk <= INT16(2);
```

Example above set parameter kk of tensor operator myclass::my_op to 2. This parameter is shared among all pcores and among all p-threads.

2.3.6 Broadcast transfer to PCORE memory space.

All pcores may receive the same broadcast data by specifying '*' as the PCORE index range.

2.3.6.1 Example

Example below broadcast data to all p-cores.

```
> PCORE(8)[*].thread[0:15].class::var[0:7] <= DDR(p)[0:127];
```

2.3.7 Multi-cast transfer to PCORE memory space.

When pcores are arranged in a 2-D array, all pcores belonged to same row or column can receive the same multi-cast data.

Multi-cast is allowed only when dimensions of pcore array have sizes in power of 2.

Examples below demonstrate the syntax for multi-cast transfers.

2.3.7.1 Example 1

In example below, pcores are arranged in a 4x2 array.
All pcores belonged to the same row receive the same multi-cast data.
In example below, multi-cast transfer is identified with character '*' in the row index of pcore array.

```
> PCORE(4,2)[0:3][*].thread[0:15].class::var[0:7] <= DDR(p)[0:511];
```

2.3.7.2 Example 2

In example below, pcores are arranged in a 4x2 array.
All pcores belonged to the same column receive the same multi-cast data.
In example below, multi-cast transfer is identified with character '*' in the column index of pcore array.

```
> PCORE(4,2)[*][0:1].thread[0:15].class::var[0:7] <= DDR(p)[0:255];
```

2.3.8 Tensor reshaping

```
>PCORE[0].THREAD[0:2].myclass::var[0:3] <= DDR(p,100,3,2)[0][0:2][0:3];
```

Example above transfers data from DDR tensor data object but the DDR dimension indexes are going out-of-bound in some cases.

Tensor data associated with out-of-bound dimension index are substituted with zero for read operation and skipped for write operation.

The result of the transfer is listed below

```
PCORE[0].THREAD[0].myclass::var[0] = p[0][0][0]
PCORE[0].THREAD[0].myclass::var[1] = p[0][0][1]
PCORE[0].THREAD[0].myclass::var[2] = p[0][0][2] <== Out-of-bound index. Substituted with zero.
PCORE[0].THREAD[0].myclass::var[3] = p[0][0][3] <== Out-of-bound index. Substituted with zero.
PCORE[0].THREAD[1].myclass::var[0] = p[0][1][0]
PCORE[0].THREAD[1].myclass::var[1] = p[0][1][1]
PCORE[0].THREAD[1].myclass::var[2] = p[0][1][2] <== Out-of-bound index. Substituted with zero.
PCORE[0].THREAD[1].myclass::var[3] = p[0][1][3] <== Out-of-bound index. Substituted with zero.
PCORE[0].THREAD[2].myclass::var[0] = p[0][2][0]
PCORE[0].THREAD[2].myclass::var[1] = p[0][2][1]
PCORE[0].THREAD[2].myclass::var[2] = p[0][2][2] <== Out-of-bound index. Substituted with zero.
PCORE[0].THREAD[2].myclass::var[3] = p[0][2][3] <== Out-of-bound index. Substituted with zero.
```

2.3.9 Setting Out-of-bound access values

During tensor reshaping, out-of-bound values are substituted with zero values by default.
However, different value can be specified instead with the use of PAD keyword.
The example below, data transfer of out-of-bound items will have values of 0xff.

```
>PCORE[0].THREAD[0:2].myclass::var[0:3] <= PAD(0xff) DDR(p,100,3,2)[0][0:2][0:3];
```

2.3.10 Dimension casting

You can cast the dimension of the components of pcore tensor to different dimension.

Example:

```
>PCORE(8)[0:7].THREAD[0:15].myclass::myvar[0:15] <= DDR(p)[0:8*16*16-1];
```

Now recast dimension of thread component from vector of 16 to array of 4x4. And recast myvar dimension from vector of 16 to array of 4x4. Note that the access pattern of the casted components is also changed from 1-D indexing to 2-D indexing access pattern.

```
>PCORE(8)[0:7].THREAD(4,4)[0:3][0:3].myclass::myvar(4,4)[0:3][0:3] <= DDR(p)[0:8*16*16-1];
```

2.3.11 Data reordering

By default, the tensor data access order starts with index going from right to left.

However, the order can be changed from the default by using FOR directive. The order starts with index without a corresponding FOR directive going from right to left and then the order continues with index with a corresponding FOR directive but in the order of the FOR directives going from right to left.

In example below, the access pattern starts with PCORE component indexing and then J for thread component indexing and then K for myvar component indexing.

```
>FOR(K=0:3) FOR(J=0:1) PCORE[0:2].THREAD[J].myclass::myvar[K] <= DDR(p)[0:2*3*4-1];
```

The result of the transfer above is listed below

```
PCORE[0].THREAD[0].myclass::myvar[0] = p[0]
PCORE[1].THREAD[0].myclass::myvar[0] = p[1]
PCORE[2].THREAD[0].myclass::myvar[0] = p[2]
PCORE[0].THREAD[1].myclass::myvar[0] = p[3]
PCORE[1].THREAD[1].myclass::myvar[0] = p[4]
PCORE[2].THREAD[1].myclass::myvar[0] = p[5]
PCORE[0].THREAD[0].myclass::myvar[1] = p[6]
PCORE[1].THREAD[0].myclass::myvar[1] = p[7]
PCORE[2].THREAD[0].myclass::myvar[1] = p[8]
PCORE[0].THREAD[1].myclass::myvar[1] = p[9]
PCORE[1].THREAD[1].myclass::myvar[1] = p[10]
PCORE[2].THREAD[1].myclass::myvar[1] = p[11]
PCORE[0].THREAD[0].myclass::myvar[2] = p[12]
PCORE[1].THREAD[0].myclass::myvar[2] = p[13]
PCORE[2].THREAD[0].myclass::myvar[2] = p[14]
PCORE[0].THREAD[1].myclass::myvar[2] = p[15]
PCORE[1].THREAD[1].myclass::myvar[2] = p[16]
PCORE[2].THREAD[1].myclass::myvar[2] = p[17]
PCORE[0].THREAD[0].myclass::myvar[3] = p[18]
PCORE[1].THREAD[0].myclass::myvar[3] = p[19]
PCORE[2].THREAD[0].myclass::myvar[3] = p[20]
PCORE[0].THREAD[1].myclass::myvar[3] = p[21]
PCORE[1].THREAD[1].myclass::myvar[3] = p[22]
PCORE[2].THREAD[1].myclass::myvar[3] = p[23]
```

2.3.12 Concurrent transfer

pcores are VLIW vector processors that operate on data in vector units. Therefore, to be efficient, ztachip requires data to be in vector format.

However, this restriction may not match application data format. And transferring scalar data is not efficient since transfer is done on one scalar element at a time instead of a whole vector at a time.

Concurrent transfer is the solution to this problem.

Concurrent transfers allow transfers to/from pcore's memory space to happen concurrently on all pcores at the same time. For example, it would take 8 clocks to access all 8 scalar values when they are not in continuous memory location. But with concurrent transfer mode, all pcores are accessing its memory

space at the same time, the net result is that we achieve an access rate of 1 vector data-unit per clock to/from pcore memory space even when data are not aligned in vector format.
Concurrent transfer is enabled with directive CONCURRENT.

CONCURRENT directive applies only to pcore tensors.

However, concurrent transfers are only allowed for the 2 scenarios described in the following sub-sections.

2.3.12.1 Tensor concurrent transfer – scenario 1.

```
>FOR(I=0:7) PCORE(8)[0:7].THREAD[0:15].myclass::myvar(8,8)[:][I] <= DDR(p,1024,8)[0:1023][0:7];
```

We show below the accessing pattern resulting from tensor transfer instruction above. Note that it takes 8 clocks to set a vector unit from DDR tensor to pcore memory space since data access in vector mode to pcore memory space is not possible.

```
PCORE[0].THREAD[0].myclass::myvar(8,8)[0][0] <= p[0];
PCORE[0].THREAD[0].myclass::myvar(8,8)[1][0] <= p[1];
PCORE[0].THREAD[0].myclass::myvar(8,8)[2][0] <= p[2];
PCORE[0].THREAD[0].myclass::myvar(8,8)[3][0] <= p[3];
PCORE[0].THREAD[0].myclass::myvar(8,8)[4][0] <= p[4];
PCORE[0].THREAD[0].myclass::myvar(8,8)[5][0] <= p[5];
PCORE[0].THREAD[0].myclass::myvar(8,8)[6][0] <= p[6];
PCORE[0].THREAD[0].myclass::myvar(8,8)[7][0] <= p[7];
  :
  :
```

By adding CONCURRENT directive, the transfer is now rearranged and interleaved automatically among all the pcores so that the 8-clock access cycles are now overlapped between all the pcores.

This way transfer above can still occur in vector mode at rate of 1 vector per clock.

The transfer now becomes...

```
>CONCURRENT FOR(I=0:7) PCORE(8)[0:7].THREAD[0:15].myclass::myvar(8,8)[:][I] <= DDR(p,1024,8)[0:1023][0:7];
```

This type of concurrent transfers is very useful since it is a very common scenario when your data are not vectors but you like to group scalar data as elements of vectors for pcores to process them simultaneously in vector mode.

2.3.12.2 Tensor concurrent transfer – scenario 2

```
>FOR(I=0:7) FOR(J=0:7) PCORE(8)[0:7].THREAD(2,8)[:][:].myclass::myvar[J] <= DDR(p,1024,8)[0:1023][0:7];
```

We show below the accessing pattern resulting from tensor transfer instruction above. Note that it takes 8 clocks to set a vector unit from DDR tensor to pcore memory space.

```
PCORE[0].THREAD[0][0].myclass::myvar[0] <= p[0];
PCORE[0].THREAD[0][1].myclass::myvar[0] <= p[1];
PCORE[0].THREAD[0][2].myclass::myvar[0] <= p[2];
PCORE[0].THREAD[0][3].myclass::myvar[0] <= p[3];
PCORE[0].THREAD[0][4].myclass::myvar[0] <= p[4];
PCORE[0].THREAD[0][5].myclass::myvar[0] <= p[5];
PCORE[0].THREAD[0][6].myclass::myvar[0] <= p[6];
PCORE[0].THREAD[0][7].myclass::myvar[0] <= p[7];
 :
 :
```

By adding CONCURRENT directive, the transfer is rearranged and interleaved among all the pcores so that the 8-clock access cycles are now overlapped between all the pcores.

This way transfer can still occur in vector mode at rate of 1 vector per clock.

The transfer now becomes...

```
>CONCURRENT FOR(I=0:7) FOR(J=0:7) PCORE(8)[0:7].THREAD(2,8)[:][:].myclass::myvar[J] <= DDR(p)[0:8191];
```

2.3.13 Tensor transfer with boundary check disabled.

Normally the dimension index is bounded by the tensor dimension. If index is out-of-bound, then a read operation is padded with zero and a write operation is skipped.

However, you can disable this behaviour by adding '+' after tensor dimension.

Example below shows DDR tensor that are accessed with index being out-of-bound.

```
>PCORE[0].THREAD[0:2].myclass::myfunc.var[0:3] <= DDR(p,100,2,2)[0][0:2][0:3];
```

For example p[0][2][2] and p[0][2][3] are both out-of-bound.

Now we disable boundary check on DDR tensor by adding a '+' after the dimension that we would like boundary check to be disabled.

```
>PCORE[0].THREAD[0:2].myclass::myfunc.var[0:3] <= DDR(p,100,2+,2+)[0][0:2][0:3];
```

Now boundary check of the last 2 dimensions of DDR tensor is disabled. Read/Write access to out-of-bound index is carried out as if the index is valid.

2.3.14 Tensor transfer with overlay dimension

Tensor can be defined to have an overlapping dimension definition.

This is useful in-order to add additional boundary checks on the tensor read/write access.

2.3.14.1 Tensor transfer with overlapped dimension - Example 1

```
DDR(p,1000(100,200))[:][:]
```

Tensor above is defined as dimension 100x200 but also constrained as 1-D dimension 1000.

Boundary check is performed on both dimension definitions.

This tensor is being recognized in a tensor data transfer operation as a 100x200 tensor. The other dimension definition of 1000 is only for boundary check purposes.

2.3.14.2 Tensor transfer with overlapped dimension - Example 2

```
DDR(p,32,152(10,16))[:][:]
```

Tensor above is defined as dimension 32x10x16 but also as dimension 32x152

Boundary check is performed on both dimension definitions.

This tensor is being recognized in a tensor data transfer operation as a 32x10x16 tensor. The other dimension definition of 32x152 is only for boundary check purposes.

2.3.15 Tensor padding values

Tensor read/write accesses are checked against dimension boundary.

Any out-of-bound read accesses are padded with zero by default.

Any out-of-bound write accesses are skipped.

But you can set different padding values for out-of-bound read accesses with PAD keyword.

In example below, any out-of-bound read accesses from DDR tensor are padded with 0xff instead of zero.

```
>PCORE[0].THREAD[0:2].myclass::myfunc.var[0:3] <= PAD(0xff) DDR(p,100,2,2)[0][0:2][0:3];
```

2.3.16 Tensor transfer data types

DTYPE directive specifies the data type of the tensor used during a data transfer operation.

Tensors data transfer can have the following data types

UINT8	Unsigned 8-bit integer
INT8	Signed 8-bit integer
INT16	Signed 12-bit, or 16-bit integer depends on ztachip configuration.

When applied to a tensor in DDR memory space, DTYPE directive implies the data type of the tensor.

When applied to a tensor from pcore memory space, since pcore memory data types are always INT16, DTYPE directive implies a data conversion when data are read or written to pcore memory space.

Below is an example of tensor data transfer in UINT8 data format.

```
DTYPE(UINT8) PCORE[0].THREAD[0:2].myclass::var[0:3] <= DTYPE(UINT8) DDR(p,100,2,2)[0][0:2][0:3];
```

or statement below will also have the same effects.

```
int fmt=UINT8;

DTYPE(fmt) PCORE[0].THREAD[0:2].myclass::var[0:3] <= DTYPE(fmt) DDR(p,100,2,2)[0][0:2][0:3];
```

2.3.17 Tensor data transfer in REPEAT mode

Data transfer may run in repeat mode when REPEAT directive is specified.

REPEAT keyword applies to source tensor only.

When specified, data access from source tensor will be repeated until the destination tensor received all the requested amount of data.

For example, below, var variables for all pcore and p-threads are filled with the same vector values from DDR(p)[0:3].

```
PCORE[0:7].THREAD[0:15].myclass::var[0:3] <= REPEAT DDR(p)[0:3];
```

2.3.18 Tensor data remap

Transfer to/from PCORE memory space can go through a data remapping process.

A stream processor performs lookup/interpolation translation from input values to output values as data are retrieved from or written to pcore's memory space.

The data remapping is performed based on remap tables that are generated by host program before they

are loaded to master tensor core.

Procedures to create and load these remap tables will be covered in later sections.

There can be up to 4 different remap tables active at same time.

2.3.18.1 Tensor data remap – Example 1

```
> REMAP(0) PCORE(NP)[0:NP-1].THREAD[0:NT-1].convolution.input[0:7] <= DDR(pin)[0:NP*NT*8-1];
```

In the example above, data just before being written to pcore's memory space are being remapped by table 0

2.3.18.2 Tensor data remap – Example 2

```
> DDR(pout)[0:NP*NT*8-1] <= REMAP(1) PCORE(NP)[0:NP-1].THREAD[0:NT-1].convolution.output[0:7];
```

In the example above, data retrieved from pcore's memory space are being remapped by remap table 1

2.3.19 BARRIER

Tensor data transfers may be executed out of order.

However, when BARRIER directive is specified, then all transfers up to that point must be completed before any instructions after BARRIER can be executed.

Example below, BARRIER is introduced to ensure data retrieval from myvar is completed before setting myvar with new values.

```
>DDR(p)[0:8*16*16-1] <= PCORE(8)[:].THREAD[0:15].myclass::myvar[0:15];

>BARRIER;

>PCORE(8)[:].THREAD[0:15].myclass::myvar[0:15] <= DDR(p)[0:8*16*16-1];
```

2.3.20 LATEST

When LATEST directive is specified with a source tensor, then all pending writes to that source tensor must be completed first before the tensor can be used as source tensor for a data transfer operation.

For example, below, the LATEST directive ensures that there are no outstanding write transactions to PCORE memory space before it can proceed with the instruction.

```
>PCORE(8)[:].THREAD[0:15].myclass::myvar[0:15] <= DDR(p)[0:8*16*16-1];

>DDR(p)[0:8*16*16-1] <= LATEST PCORE(8)[:].THREAD[0:15].myclass::myvar[0:15];
```

2.3.21 CONSTANT FILLER

You may fill destination tensor with a constant value.

Constant may have types UINT8,INT8 or INT16.

Example below fills pcore's myvar variables with constant unsigned 8-bit value 0xAA.

```
>DTYPE(UINT8) PCORE(8)[:].THREAD[0:15].myclass::myvar[0:15] <= UINT8(0xAA);
```

Example below fills pcore's myvar variables with constant signed 8-bit value 0x12.

```
>DTYPE(INT8) PCORE(8)[:].THREAD[0:15].myclass::myvar[0:15] <= INT8(0x12);
```

Example below fills pcore's myvar variables with constant signed 16-bit value 0x123.

```
>DTYPE(INT16) PCORE(8)[:].THREAD[0:15].myclass::myvar[0:15] <= INT16(0x123);
```

2.4 Tensor operator execution

Tensor operators are functions defined in pcore programs. They perform numerical calculation on a set of tensors reside in pcore memory space. Tensor operators will be described in more details at later sections.

tensor programs invoke tensor operators using the syntax described in the following sub-sections.

Tensor core ensures that all memory operations to/from pcore memory space are completed before tensor operators can commence.

Tensor operators are executed under one of the two t-threads with each t-thread having its own pcore memory space. This is an important concept since it allows tensor operator to be executed on t-thread A while t-thread B still busy with its memory operation.

2.4.1 Tensor operator syntax

Example below invokes a tensor operator defined as covolution::exe in pcore program

```
> EXE_LOCKSTEP(convolution::exe);
```

Example below invokes the tensor operator but the execution is limited only to 4 pcores.

```
> EXE_LOCKSTEP(convolution::exe,4);
```

Example below invokes tensor operator but the execution is limited only to 4 p-cores and 8 p-threads per pcore.

```
> EXE_LOCKSTEP(convolution::exe,4,8);
```

Example below invokes tensor operator in 2 steps. First a function pointer is computed and assigned to a 32-bit integer variable and then tensor operator is invoked via this function pointer. Example below will have similar effects as previous example.

```
uint32_t func;
func=ztaBuildKernelFunc($convolution::exe,4,8);
> EXE_LOCKSTEP(func);
```

ztaBuildKernelFunc will be covered in more details at later sections.

2.5 HOST SUPPORTING FUNCTIONS

Functions described in following sections are functions running on host processors in-order to support ztachip execution framework.

2.5.1 `void ztaInit()`

This is the first function to be called by host processor.

Performs necessary ztachip framework initialization.

Parameters:

None

Return value:

None

2.5.2 `void ztaDualHartExecute(void(*func)(void *,int),void *pparm)`

To execute a tensor program but in a multi-threaded execution mode. There are 2 threads being spawned with each thread emitting instructions to one of the 2 available t-threads.
The spawned threads are multi-tasking but in cooperation mode. Meaning the thread must explicitly yield control for execution to switch control over to the other thread.

Parameters

func	Thread entry function. This function's first parameter has same value as pparm, and second parameter is to identify which t-thread (0 or 1)
pparm	pointer to be passed to func's first parameter.

Return value:
None

2.5.3 `void ztaTaskYield()`

For thread spawned by ztaDualHardExecute to yield control to the other thread.

Parameters:
None

Return value:
None

2.5.4 `uint32_t ztaBuildKernelFunc(uint32_t _func,int num_pcore,int num_tid)`

Generates function pointer to a tensor operator.

Parameters

_func	Identifier of tensor operator. Tensor operator is identified as $class::operator
num_pcore	Number of pcores to be used for the execution of this tensor operator.
num_tid	Number of p-threads per pcore to be used for the execution of this tensor operator.

Return value:
Function pointer to the tensor operator.

Example below shows how tensor operator is invoked via function pointer generated by this function.

```
uint32_t func;
func=ztaBuildKernelFunc($convolution::exe,4,8);
> EXE_LOCKSTEP(func);
```

2.5.5 `void *ztaAllocSharedMem(int _size)`

Allocates memory that will be used to share data between ztachip and host processor.

Parameters

_size	Memory block size to be allocated in bytes

Return value
Pointer to allocated memory block.

2.5.6 `void *ztaFreeSharedMem(void *p)`

Free memory block allocated by ztaAllocSharedMem.

Parameters

p	Memory block to be freed. This is the memory block previously allocated by ztaAllocSharedMem.

Return value:
None

2.5.7 `void *ztaBuildSpuBundle(int numSpuImg,...)`

This function builds the lookup table to perform tensor data-remapping functions as described in 2.3.13.

As tensors are copied between external memory and pcore memory space, data-remap transforms data before data is written to pcore memory space or just after data is read from pcore memory space.

This function generates a lookup table that in turn will be uploaded to tensor core to perform data remapping.

Parameters

numSpuImg	Number of data-remapping functions to be generated. Up to 4 different data-remapping functions are allowed. The following 4 parameters below are to be repeated numSpuImg times.
remap_func	Function that represents the data remapping. This function will be called to get the expected remapping output for every possible input. remap_func has the following definition: int16_t remap_func(// Return expected remapping output for an input int16_t in, // Input void *pparm, // Same as pparm below uint32_t parm, // Same as parm below uint32_t parm2 // Same as parm2 below)
pparm	Parameter to be passed to remap_func
parm	Parameter to be passed to remap_func
parm2	Parameter to be passed to remap_func

Return value:
Pointer to remapping tables.

2.5.7.1 Example of building data-remap table.

Example below produces a data-remap lookup table for 2 functions.

First function set negative value to zero.

Second function returns the absolute value of the input.

```
:
P =  ztaBuildSpuBundle(2,remap1,0,0,0,remap2,0,0,0);
:

int16_t remap1(int16_t input,void *pparm,uint32_t parm,uint32_t parm2) {
   if(input < 0)
      return 0;
   else
```

```
      return input;
}
int16_t remap2(int16_t input,void *pparm,uint32_t parm,uint32_t parm2) {
   if(input < 0)
      return -input;
   else
      return input;
}
```

2.5.8 void ztaInitPcore(uint16_t *_image)

This function uploads pcore code images to pcore processors.

This must be done as first step in a tensor program.

Parameters

_image	Pointer to memory block that holds pcore compiled binary image. ztachip toolchain generates pcore binary image as it compiles pcore programs. For example, a file named example.p.img is generated as ztachip toolchain compiles a pcore program named example.p. Defined in example.p.img is a array called zta_pcore_img[] that is initialized with the pcore binary image. zta_pcore_img is then used as parameter to ztaInitPcore to upload the pcore image to ztachip.

Return value:
None

2.5.9 void ztaInitStream(uint16_t *_spu)

This function uploads the data-remap tables that are generated by ztaBuildSpuBundle as described in 2.5.7
Data-remap is then used by tensor programs as described in 2.3.13

Parameters

_spu	data-remap table that is generated by ztaBuildSpuBundle.

Return value:

None

2.5.10 void ztaJobDone(unsigned int job_id)

This function is called at the end of tensor program.

This signals ztachip to send a notification message when tensor program execution has been completed.

ZtaReadResponse as described in 2.5.11 is then called to retrieve the notification message.

Parameters

job_id	Task identifier to be returned in the notification message at the completion of tensor program.

Return value:

None

2.5.11 `bool ztaReadResponse(uint32_t *resp)`

This function retrieves responses back from ztachip.

ztachip sends a notification message to host at the completion of a tensor program as indicated by ztaJobDone

Parameters

resp	To return the identifier of the completed task. The task identifier is specified in the function call ztaJobDone()

Return value:

true if there is a message available, false otherwise.

3-PCORE PROGRAMS

PCORE programs are codes that are executed on the array of pcore processors.

These programs have a C/C++-like syntax, but not all C/C++ features are supported.

pcore program files have suffix *.p

They implement tensor operators that can then be invoked by tensor programs. Tensor operators are essentially functions that perform numerical computation on a set of tensors resided in pcore memory space.

All pcore processors execute the same instructions in lockstep. If there are branching instructions, all pcores must branch the same way. If branching is not always the same for all pcores, vector mask techniques are used instead to achieve similar results. Vector mask is a common technique to enable/disable computational results from being written back to memory.

pcore execution is then further scheduled into thread block. There are 16 threads per pcore processor. We call them p-threads.

p-threads scheduling is hardware based with zero-overhead.
pcore thread executions are pipe-lined and interleaved between different threads. This allows for 1 instruction to be executed per clock even when each instruction may take many clock cycles to complete.

pcore has a VLIW architecture. VLIW stands for very-long-instruction-word where each instruction contain multiple operations.

3.1 VLIW instructions

PCORE processor has a VLIW architecture. Each instruction can perform up to 3 operations:

- vector operations
- scalar operations
- control operations

3.1.1 vector operations

Vector operations are executed in the vector ALU units.
Implement operations such as

- FMA

- FMS
- ADD
- SUBTRACT
- MULTIPLY
- SHIFT
- COMPARISON

Each operators have the following mode of memory addressing mode

- Addressing via pointer+index
- Addressing via pointer
- Addressing shared memory
- Addressing private memory
- Addressing shared memory + index
- Addressing private memory + index

3.1.2 scalar operations

Scalar operations are executed in the scalar ALU units.
Implement integer operations such as ADD,SUB,MUL,SHIFT. These operations are normally used for tasks such as array indexing, address calculation, loop counter...

3.1.3 control operations

Perform conditional branching

3.2 General syntax

pcore programs define tensor operators as objects with their own attributes and operations.
All these objects are defined as single instance objects. Meaning you don't have to allocate the objects and the objects are statically allocated.

Due to limited memory available from pcore memory space, all tensor objects are overlapped in memory allocation. Therefore, only one object can be in used at a time.

Example below is a simple pcore program vector_math.p which defines some vector operations.

Tensor operators for vector addition and subtraction are grouped under object vector_math.
There are 3 tensors x,y,z associated with vector_math object. These tensors have 3 dimensions since they span all the pcores, and then span all the 16 p-threads for each pcore, and at each p-thread there is a vector variable of width 8 defined. If there are 4 pcores available, then x,y,z are tensors of dimension 4x16x8

Tensor program can then assign tensor x,y with data from external memory. And then tensor program

can save results from tensor z back to external memory.

Tensor program can then invoke tensor operators vector_math::add and vector_math::subtract.

```
// vector_math.p

class vector_math;

vint16 vector_math::x;
vint16 vector_math::y;
vint16 vector_math::z;

// Tensor operator for vector addition
_kernel_ void vector_math::add()
{
    z=x+y;
}
// Tensor operator for vector subtraction
_kernel_ void vector_math::subtract()
{
    z=x-y;
}
```

3.3 Class declaration

All tensor operators and variables are grouped under object classes.

Each class must be declared at beginning of the file.

Example:

```
class vector_math; // Defines object class vector_math.
```

3.4 Memory layout

Different object classes are overplayed in memory allocation.

Which means only one object class can be in use at a time.

3.5 Tensor variables

Tensor variables are tensors allocated in pcore memory space.

Similar to tensor operators, tensor variables are also grouped under object class.

3.5.1 Data types

3.5.1.1 int16

Scalar of 12-bit or 16-bit integer depending on ztachip configuration. This data type is used for math operations with vector ALU.

Example:

```
int16 myclass::x; // variable of type int16 and belonged to object class myclass.
```

3.5.1.2 vint16

vector of 12-bit or 16-bit integers depending on ztachip configuration. This data type is used with vector ALU for math operations.

Example:

```
vint16 myclass::x; // variable of type vint16 and belonged to class myclass.
```

3.5.1.3 int32

32-bit integer type. Variables of this type are used to hold accumulator computing values.

Example:

```
int32 myclass::_A; // _A is a 32bit accumulator.
```

3.5.1.4 vint32

vector of 32-bit integer type. Variables of this type are used to hold accumulator computing values.

Example:

```
vint32 myclass::_A; // _A is a vector of 32bit accumulator.
```

3.5.1.5 int

scalar 12-bit integer. This data type is used with scalar ALU. Variable of this type is normally not used for computation purposes but rather used for tasks such as loop index, array index, pointer...

Example:

```
int myclass::i; // i is variable of type int and belonged to class myclass.
```

*3.5.1.6 int16 **

pointer to an int16 variables.

Example:

```
int16 * *myclass::p; // p is variable of type int16* and belonged to class myclass.
```

*3.5.1.7 vint16 **

Pointer to a vint16 variable.

Example:

```
vint16 *myclass::p; // p is variable of type int16 * and belonged to class myclass.
```

3.5.2 Variable types

3.5.2.1 Private variables

There is an instance of the variable allocated for each thread.

Example1:

```
// separate instance of my_private_variable are allocated for each p-thread
int16 myclass::my_private_variable;
```

If we have 4 pcores and there are 16 p-threads per pcore, tensor program would see the variable above as a tensor below:

```
int16 my_private_variable[4][16];
```

Example below shows how tensor program can set values to this tensor.

```
>PCORE(4)[0:3].THREAD[0:15].myclass::my_private_variable <= DDR(p)[0:63];
```

Example2:

```
// separate instance of my_private_variable are allocated for each p-thread
vint16 myclass::my_private_variable;
```

If we have 4 pcores and there are 16 p-threads per pcore and vector width is 8, then tensor program would see the variable above as a tensor below:

```
int16 my_private_variable[4][16][8];
```

Example below shows how tensor program can set values to this tensor.

```
>PCORE(4)[0:3].THREAD[0:15].myclass::my_private_variable[0:7] <= DDR(p)[0:511];
```

3.5.2.2 Shared variables

There is only one instance of the variable per pcore. The variable is shared among all the p-threads within the same pcore processor.

Shared variables are preceded with keyword `_share`

Example1:

```
// my_shared_variable is shared among all the p-threads in the same PCORE.
_share int16 myclass::my_shared_variable;
```

If we have 4 pcores, tensor program would see the variable above as a tensor below:

```
int16 my_shared_variable[4];
```

Example below shows how tensor program can set values to this tensor.

```
>PCORE(4)[0:3].myclass::my_shared_variable <= DDR(p)[0:3];
```

Example2:

```
// my_shared_variable is shared among all the p-threads in the same PCORE.
_share vint16 myclass::my_shared_variable;
```

If we have 4 pcores and vector width is 8 then tensor program would see the variable above as a tensor below:

```
int16 my_shared_variable[4][8];
```

Example below shows how tensor program can set values to this tensor.

```
>PCORE(4)[0:3].myclass::my_shared_variable[0:7] <= DDR(p)[0:31];
```

3.5.2.3 Parameter variables

Tensor operators can have variables as function parameters.

Example1:

```
_kernel_ void vector_math::add(int16 x,int16 y,int16 z)
{
    z=x+y;
}
```

x,y,z in example above are tensors declared as tensor operator parameters.

If we have 4 pcores and 16 threads per pcore, then x,y,z are tensors with dimension 4x16

And x is set from tensor program as followed:

```
>PCORE(4)[0:3].THREAD[0:15].vector_math::add.x <= DDR(p)[0:63];
```

Example2:

```
_kernel_ void vector_math::add(vint16 x,vint16 y,vint16 z)
{
    z=x+y;
}
```

x,y,z in example above are tensors declared as tensor operator parameter.

If we have 4 pcores with 16 threads per pcore and vint16 are vector type with width=8, then x,y,z are tensors with dimension 4x16x8

And x can be set from tensor program as followed:

```
>PCORE(4)[0:3].THREAD[0:15].vector_math::add.x[0:7] <= DDR(p)[0:63];
```

3.5.2.4 Global variables

Variables of global scope are defined as function parameters with prefix _global.

All pcores and p-threads shared the same global variables.

Global variables are read-only.

Global variables can be set by tensor programs as described in 2.3.5

Global variables can only be of int type.

Example:

```
_kernel_ void vector_math::add(_global int kk)
{
    z[kk] = x[kk]+y[kk];
}
```

In example above, parameter kk of tensor operator vector_math::add is shared among all pcores and p-threads.

And tensor program would set parameter kk as followed:

```
> vector_add::add.kk <= INT16(2); // Set kk=2
```

3.6 Tensor operators

Tensor operators are functions that can be invoked by tensor programs. These functions have a void return type and preceded with keyword `_kernel_`

Tensor operators must be defined as an operation of a class as defined in 3.4

They may optionally have an input parameter list. Parameter list are temporary variables that have a lifetime of the function call only.

When another function being called, the parameter list of the new functions may overwrite the parameter variables of the previous function call.

For example:

```
_kernel_ void matrix::my_function(vint16 x,vint16 y)
{
     z=x+y;
}
```

Tensor operator above can be invoked from tensor program as followed.

```
>EXE_LOCKSTEP(matrix::my_function);
```

3.7 _VMASK

This is a special integer variable where each bit activates/deactivates write operation to a vector lane. When bit is set, then write operation to the corresponding vector lane is enabled.

When bit is clear, then write operation to the corresponding vector lane is skipped.

Example:

```
vint16 x,y;

_VMASK=1;  // enables only the first lane of the vector ALU.
x=y;       // equivalent to x[0]=y[0];

_VMASK=3;  // enables the first and second lane of vector ALU.
x=y;       // Equivalent to x[0]=y[0];x[1]=y[1];

_VMASK=-1; // Enable all lanes
x=y;       // Equivalent to x[0]=y[0];x[1]=y[1];...;x[7]=y[7];
```

3.8 COMPARISON OPERATORS

Since p-cores are all executed in lock-step mode, comparison operators described in subsequent sections are alternative ways to have conditional execution.

3.8.1 GE(v1,v2)

Test for v1 >= v2. Result is an integer where each bit corresponds to condition being true for the vector lane.

Return value is an int type. The result is normally assigned to _VMASK.

Example:

```
// Doing element wise comparison of vector v1 and v2
_VMASK=GE(v1,v2);
// Set v1 elements to v2 elements if v1 element if greater or equal to v2 element.
v1=v2;
```

3.8.2 GT(v1,v2)

Test for v1 > v2. Result is an integer where each bit corresponds to condition being true for the vector lane.

Return value is an int type. The result is normally assigned to _VMASK.

For example:

```
// Doing element wise comparison of vector v1 and v2
_VMASK=GT(v1,v2);
// Set v1 elements to v2 elements if v1 element if greater than v2 element.
v1=v2;
```

3.8.3 LE(v1,v2)

Test for v1 <= v2. Result is an integer where each bit corresponds to condition being true for the vector lane.

Return value is an int type. The result is normally assigned to _VMASK.

For example:

```
// Doing element wise comparison of vector v1 and v2

_VMASK=LE(v1,v2);

// Set v1 elements to v2 elements if v1 element if less than or equal to v2 element.
v1=v2;
```

3.8.4 LT(v1,v2)

Test for v1 < v2. Result is an integer where each bit corresponds to condition being true for the vector lane.

Return value is an int type. The result is normally assigned to _VMASK.

For example:

```
// Doing element wise comparison of vector v1 and v2

_VMASK=LT(v1,v2);

// Set v1 elements to v2 elements if v1 element if less than v2 element.
v1=v2;
```

3.8.5 EQ(v1,v2)

Test for v1 == v2. Result is an integer where each bit corresponds to condition being true for the vector lane.

Return value is an int type. The result is normally assigned to _VMASK.

For example:

```
// Doing element wise comparison of vector v1 and v2

_VMASK=EQ(v1,v2);

// Set v1 elements to v2 elements if v1 element if is equalled to v2 element.
v1=v2;
```

3.8.6 NE(v1,v2)

Test for v1 != v2. Result is an integer where each bit corresponds to condition being true for the vector lane.

Return value is an int type. The result is normally assigned to _VMASK.

For example:

```
// Doing element wise comparison of vector v1 and v2

_VMASK=NE(v1,v2);

// Set v1 elements to v2 elements if v1 element not equal to v2 element.
v1=v2;
```

4-DEBUGGING

Debugging tensor programs is the same as debugging any C/C++ programs using GDB debugger.

With GDB debugger, we can execute tensor programs in single step mode.

GDB debugger provides commands to read/write DDR external memory.

For debugging purposes, p-core memory space is also mapped to memory address that are reachable by GDB debugger. ztachip toolchain generates a XML file to show where each p-core program symbols are mapped to memory address.

Unlike tensor program, p-core program debugging cannot be single stepped. But p-core tensor operators are typically compact functions and inspecting the input and output to the function is normally sufficient. Example below is common way to debug a tensor operator.

- Using GDB debugger to pause tensor program just before EXE_LOCKSTEP.
- Inspect appropriate symbols in p-core memory space that represent input to the tensor operators.
- Using GDB debugger to step over EXE_LOCKSTEP.
- Now inspect appropriate symbols in p-core memory space that represent output from tensor operators.

5 -BUILD PROCEDURES

For each acceleration function, there are 2 files to be built: a tensor program and a pcore program.

Tensor program has suffix *.m

pcore program has suffix *.p

ztachip/SW/compiler/compiler is used to compile tensor programs and pcore programs.

5.1 Example

In example below, we would like to build a ztachip acceleration function implemented in example.m and example.p

- Using ztachip compiler to translate tensor instructions embedded in tensor program example.m to special binary code. The generated file is example.m.c
- Using ztachip compiler to compile pcore program example.p. The generated file is example.p.img. This file is then being included in the tensor program.
- Compiles example.m.c using host (RISCV) C/C++ compiler.
-

For more details about build procedure, reference ztachip/SW/makefile.kernels

6 -EXAMPLES

The following sections are some examples of tensor and pcore programs.

Procedure to build tensor programs is based on ztachip/SW/makefile

Procedure to build pcore programs is based on ztachip/SW/makefile.kernels

6.1 Basic example

Example below is a simple vector addition example. It simply takes 2 vectors x and y from external memory and perform vector addition x+y before saving the result that is tensor z to external memory.

There are 2 components:

- tensor program implemented in example.m
- pcore program implemented in example.p

6.1.1 Tensor program code (example.m)

```
// This file contains the pcore binary image generated when example.p being compiled.
#include “example.p.img”

void do_vector_add(int16_t *x,int16_t *y,int16_t *z,int len) {
   uint32_t resp;
   // Load pcore binary image
   ztaInitPcore(zta_pcore_img);
   for(i=0;i < len;i += 512) {
      // Load x tensor from DDR memory to pcore memory space
      >DTYPE(INT16)PCORE(4)[0:3].THREAD[0:15].example::x[0:7] <= DTYPE(INT16)DDR(x,len)[i:i+511];
      // Load y tensor from DDR memory to pcore memory space
      >DTYPE(INT16)PCORE(4)[0:3].THREAD[0:15].example::y[0:7] <= DTYPE(INT16)DDR(y,len)[i:i+511];
      // Execute tensor operator example::add
      >EXE_LOCKSTEP(example::add);
      // Save result tensor z from pcore memory space to DDR memory.
      DTYPE(INT16)DDR(z,len)[i:i+511] <= DTYPE(INT16)PCORE(4)[0:3].THREAD[0:15].example::z[0:7];
   }
   // The job is all submitted at this point but not necessarily executed yet.
   ztaJobDone(0);
   // Wait for ztachip to indicate that the execution is completed.
   while(!ztaReadResponse(&resp));
}
```

6.1.2 pcore program code (example.p)

```
// Declare class
class example;
// x tensor allocated in pcore private memory space
vint16_t example::x;
// y tensor allocated in pcore private memory space
vint16_t example::y;
// z tensor allocated in pcore private memory space
vint16_t example::z;

// Tensor operator for vector addition
_kernel_ void example::add() {
   z=x+y;
}
```

6.2 Example running dual tensor-threads

This example also performs vector addition like previous example.

However, now we perform vector addition using both t-threads with each t-thread performing addition on half of the vector.

The advantage is that tensor operator execution from one t-thread will overlap with tensor memory operation from the other t-thread, therefore improving overall performance with no memory wait time.

There are 2 components:

- tensor program implemented in example.m
- pcore program implemented in example.p

6.2.1 example.m

```
// This file contains the pcore binary image generated when example.p being compiled.
#include “example.p.img”


typedef struct {
   int16_t *x;
   int16_t *y;
   int16_t *z;
   int len;
} REQUEST;


void do_vector_add(int16_t *x,int16_t *y,int16_t *z,int len) {
   REQUEST req = {x,y,z,len};
   // Load pcore binary image
   ztaInitPcore(zta_pcore_img);
   // Launch dual threads with each emitting tensor instructions to one of the 2 tensor-threads
   ztaDualHartExecute(thread_entry_point,&req);
   // The job is all submitted at this point but not necessarily executed yet.
   ztaJobDone(_req_id);
   // Wait for ztachip to indicate that the execution is completed.
   while(!ztaReadResponse(&resp));
}


// Thread entry function. We have 2 threads matching the 2 tensor threads.
void thread_entry_point(void *p,int pid) {
   REQUEST *req=(REQUEST *)p;
   // First thread takes the first half of vector, second thread takes bottom half
```

```
   for(i=(pid==0)?0:req->len/2;i < (pid==0)?req->len/2:req->len;i += 512) {
      // Load x,y tensors from DDR memory to pcore memory space
      >DTYPE(INT16)PCORE(4)[0:3].THREAD[0:15].example::x[0:7] <= DTYPE(INT16)DDR(x)[i:i+511];
      >DTYPE(INT16)PCORE(4)[0:3].THREAD[0:15].example::y[0:7] <= DTYPE(INT16)DDR(y)[i:i+511];
      // Execute tensor operator example::add
      >EXE_LOCKSTEP(example::add);
      // Yield control for the other thread to take control
      ztaTaskYield();
      // Save result tensor z from pcore memory space to DDR memory.
      // First thread saves first half of vector result, second thread saves bottom half of vector result
      >DTYPE(INT16)DDR(z,(pid==0)?req->len/2:req->len)[i:i+511] <=
      >DTYPE(INT16)PCORE(4)[0:3].THREAD[0:15].example::z[0:7];
   }
}
```

6.2.2 pcore program code (example.p)

```
// Declare class
class example;
// x tensor allocated in pcore private memory space
vint16_t example::x;
// y tensor allocated in pcore private memory space
vint16_t example::y;
// z tensor allocated in pcore private memory space
vint16_t example::z;

// Tensor operator for vector addition
_kernel_ void example::add() {
   z=x+y;
}
```

6.3 Example of using tensor data-remap function.

This example also performs vector addition like previous examples. However, we also divide the result by 3. The division step is performed by tensor data-remapping.

6.3.1 Tensor program code (example.m)

```
// This file contains the pcore binary image generated when example.p being compiled.
#include “example.p.img”

// Remap function. Divide input by 2
int16_t remap_func(int16_t in,void *,uint32_t,uint32_t) {
   return in/3;
}

void do_vector_add(int16_t *x,int16_t *y,int16_t *z,int len) {
   uint32_t resp;
   void *lookup;
   // Load pcore binary image
   ztaInitPcore(zta_pcore_img);
   // Build lookup table
   lookup=ztaBuildSpuBundle(1,remap_func,0,0,0);
   // Load lookup table
   ztaInitStream(lookup);
   for(i=0;i < len;i += 512) {
      // Load x tensor from DDR memory to pcore memory space
      >DTYPE(INT16)PCORE(4)[0:3].THREAD[0:15].example::x[0:7] <= DTYPE(INT16)DDR(x,len)[i:i+511];
      // Load y tensor from DDR memory to pcore memory space
      >DTYPE(INT16)PCORE(4)[0:3].THREAD[0:15].example::y[0:7] <= DTYPE(INT16)DDR(y,len)[i:i+511];
      // Execute tensor operator example::add
      >EXE_LOCKSTEP(example::add);
      // Save result tensor z from pcore memory space to DDR memory.
      // Also performing data-remapping of the result using REMAP directive.
      DTYPE(INT16)DDR(z,len)[i:i+511] <= REMAP(0)DTYPE(INT16)PCORE(4)[0:3].THREAD[0:15].example::z[0:7];
   }
   // The job is all submitted at this point but not necessarily executed yet.
   ztaJobDone(0);
```

```
    // Wait for ztachip to indicate that the execution is completed.
    while(!ztaReadResponse(&resp));
    ztaFreeSharedMem(lookup);
}
```

6.3.2 pcore program code (example.p)

```
// Declare class
class example;
// x tensor allocated in pcore private memory space
vint16_t example::x;
// y tensor allocated in pcore private memory space
vint16_t example::y;
// z tensor allocated in pcore private memory space
vint16_t example::z;

// Tensor operator for vector addition
_kernel_ void example::add() {
    z=x+y;
}
```

6.4 More examples

ztachip provides many vision and AI acceleration functions.

They are excellent examples to be used as reference points for tensor and pcore programming.

The following tensor/pcore programs are available.

Canny edge detection	ztachip/SW/apps/canny/kernels/canny.m ztachip/SW/apps/canny/kernels/canny.p
Colour space conversion	ztachip/SW/apps/color/kernels/color.m ztachip/SW/apps/color/kernels/color.p
Contrast equalizer	ztachip/SW/apps/equalize/kernels/equalize.m ztachip/SW/apps/equalize/kernels/equalize.p
Gaussian blurring	ztachip/SW/apps/gaussian/kernels/gaussian.m ztachip/SW/apps/gaussian/kernels/gaussian.p
Harris Corner detector	ztachip/SW/apps/harris/kernels/harris.m ztachip/SW/apps/harris/kernels/harris.p
Optical Flow	ztachip/SW/apps/of/kernels/of.m ztachip/SW/apps/of/kernels/of.p
Image resize	ztachip/SW/apps/resize/kernels/resize.m ztachip/SW/apps/resize/kernels/resize.p
TensorFlow's CONV layer	ztachip/SW/apps/nn/kernels/conv.m ztachip/SW/apps/nn/kernels/conv.p
TensorFlow's FCN layer.	ztachip/SW/apps/nn/kernels/fcn.m ztachip/SW/apps/nn/kernels/fcn.p