密级状态：绝密(　　)　　秘密(　　)　　内部(　　)　　公开(√)

RKNN_Toolkit 自定义算子开发指南

（技术部，图形显示平台中心）

文件状态： [　] 正在修改 [√] 正式发布	当前版本：	V0.1.0
	作　　者：	杨华聪
	完成日期：	2019-08-22
	审　　核：	卓鸿添
	完成日期：	2019-08-22

福州瑞芯微电子股份有限公司

Fuzhou　Rockchips　Semiconductor　Co . , Ltd

更新记录

版本	修改人	修改日期	修改说明	核定人
v0.1.0	杨华聪	2019-08-22	初始版本	卓鸿添

目 录

1 主要功能说明

如果模型含有 RKNN-Toolkit 不支持的算子（operator），那么在模型转换阶段就会失败。这时候可以通过自定义算子功能来添加不支持的算子，从而模型能正常转换和运行。

当前自定义算子功能还属于实验阶段，后续接口可能还会进行调整，目前只能在 Linux x64 平台上使用，并且只支持 TensorFlow 模型。

2 系统依赖说明

1）操作系统：Linux x64

2）依赖软件

a) RKNN-Toolkit 1.2.0 版本及以上

b) gcc 和 gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu

3）目标设备驱动依赖

a) RK1808 平台

RK1808 平台的 librknn_runtime 版本需要 1.2.0 以上，可以在 rk1808 的 shell 中输入如下命令查看：

```
$ strings /usr/lib64/librknn_runtime.so | grep "librknn_runtime version"
```

b) RK3399Pro 平台

RK3399Pro 平台 DRV 版本需要 0.9.9 以上，当运行应用时候会有如下日志打印，可以通过日志中的 DRV 版本判断是否满足要求。

```
=============================================
RKNN VERSION:
   API: 0.9.9 (a949908 build: 2019-08-22 22:20:52)
   DRV: 0.9.9 (c12de8a build: 2019-08-22 20:10:17)
=============================================
```

3 使用说明

3.1 RKNN-Toolkit 自定义算子使用流程

3.1.1 创建自定义算子

1) 自定义算子配置文件

自定义算子通过一个 YAML 格式的配置文件来定义，以下是 RKNN-Toolkit 带的自定义算子示例（配置文件位于 example/custom_op/rknn_custom_op_resize/resize_area.yml）：

```
name: ResizeArea
framework: tensorflow
inputs:
  input:
    type: VX_TYPE_TENSOR
outputs:
  output:
    type: VX_TYPE_TENSOR
params:
  size:
    type: VX_TYPE_ARRAY
  align_corners:
    type: VX_TYPE_BOOL
```

YAML 配置文件的要求如下：

- name：算子名称，需要与原模型的算子名称一致
- framework：原模型框架，目前只支持 tensorflow
- inputs：定义算子输入，每个输入的命名需要不同，每个输入项的配置只需要填写 type。
- outputs：定义算子输出，每个输出的命名需要不同，每个输入项的配置只需要填写 type。
- params：定义算子参数，每个参数的命名需要不同，每个参数项的配置只需填写 type（type 支持 VX_TYPE_ARRAY 和标量类型，标量类型具体参见“读取参数”章节）。

开发者可参考上述配置来编写自己的自定义算子配置。

2) 生成自定义算子代码

编写完自定义算子的配置文件之后，可以通过以下命令来生成自定义算子的代码。

```
    cd example/custom_op/rknn_custom_op_resize
    python3 -m rknn.bin.custom_op --action create --config ./resize_area.yml
--op_path ./resize_area
```

执行完成之后在 resize_area 目录会生成算子代码。

rknn.bin.custom_op 命令的参数如下：

- **--action/-a**：传入“create”执行创建算子代码操作；传入“build”执行编译算子代码操作；
- **--config/-c**：算子配置文件的路径；
- **--op_path/-p**：存放算子代码的路径

3.1.2 编写自定义算子代码

一个自定义算子的代码清单如下所示：

```
resize_area
├── makefile.linux.aarch64
├── makefile.linux.x64
├── op.yml
├── rknn_kernel_resizearea.c
├── rknn_kernel_resizearea.vx
└── rknn_op_resizearea.py
```

其中开发者需要完成的代码主要有：

1) rknn_op_resizearea.py

该 Python 代码主要用于模型转换时获取 op 参数、计算输出 shape 以及定义输出 Tensor 的计算。详细请参考本文档中的“编写自定义算子 Python 代码”章节。

2) rknn_kernel_resizearea.c

该 C 代码主要包括 kernel 初始化相关回调和 CPU 的 kernel 函数。初始化回调函数用于检查和配置 Kernel 的参数。CPU 的 Kernel 函数的编写请参考本文档中的“编写 CPU Kernel”章节。

3) rknn_kernel_resizearea.vx

可以通过编写 VX Kernel 使用 NPU 中的 PPU 模块进行加速，详细请参考本文档中的“编写 VX Kernel”章节。

4) op.yml

op.yml 文件除了包含开发者定义的算子配置，还增加了几项配置。其中开发者有可能需要修改的配置项是 kernel_index，该配置项用于选择执行哪个 kernel（即 rknn_kernel_resizearea.c 代码中的 vx_kernel_ResizeArea_list 中的哪一项 kernel）。

```
name: ResizeArea
framework: tensorflow
inputs:
  input:
    type: VX_TYPE_TENSOR
outputs:
  output:
    type: VX_TYPE_TENSOR
params:
  size:
    type: VX_TYPE_ARRAY
  align_corners:
    type: VX_TYPE_BOOL
out_binary: ResizeArea.rknnop
op_py_file: rknn_op_resizearea.py
vx_file: rknn_kernel_resizearea.vx
kernel_index: 0
```

3.1.3 编译自定义算子

3.1.3.1 编译环境准备

1) GCC 编译器安装

用户可直接通过系统包管理安装 GCC 编译器：

```
sudo apt-get install gcc
```

2) 交叉编译器安装

首次编译时 RKNN-Toolkit 会检查是否有安装交叉编译器，如果没有会自动联网下载。用户也可以自己手动下载交叉编译器并安装到指定路径，方法如下：

```
wget
https://releases.linaro.org/components/toolchain/binaries/6.3-2017.05/aarch64-linux-gnu/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu.tar.xz
mkdir ~/.rknn
tar xvJf gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu.tar.xz -C ~/.rknn/
```

3.1.3.2 编译算子代码

执行以下命令对自定义算子的代码进行编译，如果代码有错会中止编译并提示错误。当成功编译完成后，在算子的目录会生成.rknnop 文件。

```
python3 -m rknn.bin.custom_op --action build --op_path ./resize_area
```

3.1.4 注册自定义算子

只需要在模型转换前之前调用 register_op 方法并将编译生成的 rknnop 文件路径传入，即可注册算子。如果有多个算子则调用多次 register_op 方法分别注册，参考代码如下所示。

```
rknn.register_op('./resize_area/ResizeArea.rknnop')
rknn.load_tensorflow(…)
```

3.2 编写自定义算子 Python 代码

用户需要在生成的 python 代码中完成几个方法代码的编写，从而能够让 RKNN-Toolkit 获取足够的信息来完成转换和编译出 RKNN 模型。接下来一一介绍需要完成编写的 Python 方法。

3.2.1 实现 load_params_from_tf 函数

在模型转换阶段会回调该函数来从 TensorFlow 的节点对象中得到算子参数。该函数声明如下所示：

函数	load_params_from_tf
功能	获取算子参数。
参数	node_def：该参数是 tf.NodeDef 对象，包括该节点对应的原始模型的信息。
	tensor_data_map：包括了该节点所有的输入常量 Tensor。
返回值	dict：参数字典，每项参数的键需要与 YAML 配置中所一定的参数名称一致。

以下是该函数的示例代码：

```
def load_params_from_tf(self, node_def, tensor_data_map):
    p = dict()
    # set params dict
    p['size'] = tensor_data_map['C:out0'].tolist()
    p['align_corners'] = node_def.attr['align_corners'].b
    return p
```

3.2.2 实现 compute_output_shape 函数

在模型转换阶段会回调该函数来得到算子输出 Tensor 的 Shape 信息。该函数声明如下所示：

函数	compute_output_shape
功能	获取算子输出 Tensor 的 Shape。
参数	inputs_shape：该 op 的所有输入的 shape。
	params：该节点的参数。
返回值	list：输出 Tensor 的 Shape 的列表。

以下是该函数的示例代码：

```
def compute_output_shape(self, inputs_shape, params):
    outputs_shape = [Shape() for i in range(len(self.def_output))]
    # set outputs shape by set_shape()
    in_shape = inputs_shape[0].format('nhwc')
    in_channel = in_shape[-1]
    out_shape = []
    out_shape.extend(params['size'])
    out_shape.append(in_channel)
    outputs_shape[0].set_shape(shape=out_shape, fmt='nhwc')
    return outputs_shape
```

3.2.3 实现 compute_output_tensor 函数

模型量化阶段需要回调该函数来得到 op 的计算输出 Tensor。在该方法中，开发者可以通过调用 TensorFlow 函数或通过 tf.py_func 与 Numpy 来定义算子计算方法。

函数	compute_output_tensor

功能	定义算子输出 Tensor 的计算方法。
参数	const_tensor：该节点的输入常量 tensor。
	inputs_tensor：该节点的输入 tensor 列表，列表每个元素为 tf.Tensor 对象。
	Params：该节点的参数。
返回值	list：输出 Tensor（类型为 tf.Tensor）的列表。

以下是该函数的示例代码：

```
def compute_output_tensor(self, const_tensor, inputs_tensor, params):
    outputs_tensor = list()
    # compute outputs tensor
    out = tf.image.resize_area(inputs_tensor[0],
                               size=params['size'],
                               align_corners=params['align_corners'])
    outputs_tensor.append(out)
    return outputs_tensor
```

3.3 编写 CPU Kernel

开发者可以先实现算子的 CPU 版本 Kernel 函数，可以比较容易编写和验证结果是否正确。生成的 C 代码文件的 cpu_kernel_function 函数为 CPU Kernel 函数，当网络执行到该层节点时，该函数会被回调。函数声明如下所示。

函数	cpu_kernel_function
功能	定义算子输出 Tensor 的计算方法。
参数	vx_node node：表示本节点。
	vx_reference* parameters：按顺序分别存放输入 tensor、输出 tensor 和参数。
	uint32_t paramNum：参数个数。
返回值	vx_status：参考 vx_status_e。

开发者首先需要从函数参数 vx_reference *parameter 读取输入 Tensor 数据，然后进行计算处理，最后将输出数据写入至输出 tensor 中，下面将一一详细介绍。

3.3.1 读写输入输出 Tensor

cpu_kernel_function 的函数参数 vx_reference *parameter 按 C 代码中的变量“kernel_params”所定义的顺序存放输入 Tensor、输出 Tensor 和算子参数。以下面“kernel_params”的定义为例：parameter[0]为输入 tensor；parameter[1]为输出 tensor；parameter[2]和 parameter[3]为算子参数。

```
static vx_param_description_t kernel_params[] =
    {
        {VX_INPUT, VX_TYPE_TENSOR, VX_PARAMETER_STATE_REQUIRED},
        {VX_OUTPUT, VX_TYPE_TENSOR, VX_PARAMETER_STATE_REQUIRED},
        {VX_INPUT, VX_TYPE_ARRAY, VX_PARAMETER_STATE_REQUIRED},
        {VX_INPUT, VX_TYPE_SCALAR, VX_PARAMETER_STATE_REQUIRED}
    };
```

在 OpenVX 中 vx_reference(参考 OpenVX 手册 Object:vx_reference)是一个通用的引用，可以被直接转换为其它类型，如 vx_tensor、vx_array、vx_scalar 等。因为输入 Tensor 都是 vx_tensor 类型，因此这里可以直接将 vx_reference 转为 vx_tensor 类型。

OpenVX 提供了接口能够查询和读写 vx_tensor 对象（参考 OpenVX 手册 Object:Tensor）。

- **vxQueryTensor**：查询 Tensor 属性（支持查询 Tensor 的维度、维度数量、数据类型等）；
- **vxCopyTensorPatch**：往/从 Tensor 拷贝数据。

下面是 vx_tensor 的示例代码（完整代码可以参考 RKNN_Toolkit 带的示例 example/custom_op）：

```
vx_status status;
uint32_t dim_num;
uint32_t dims[4] = {0};
// Read Tensor Attributes
status = vxQueryTensor(tensor, VX_TENSOR_NUMBER_OF_DIMS,
                       &dim_num, sizeof(uint32_t));
status = vxQueryTensor(tensor, VX_TENSOR_DIMS,
                       size, sizeof(uint32_t) * dim_num);
......
// Read Tensor Data
status = vxCopyTensorPatch((vx_tensor)parameters[0],
                                dim_num, view_start, view_end, stride_size,
                                src_buffer, VX_READ_ONLY, 0);
// Write Data to Tensor
status = vxCopyTensorPatch((vx_tensor)parameters[1],
                                dim_num, view_start, view_end, stride_size,
                                dst_buffer, VX_WRITE_ONLY, 0);
```

需要注意的是 vxCopyTensorPatch 的参数中的 view_start、view_end 和 user_stride 参数。

- **view_start**：该参数表示要读取/写入 Tensor 的起始位置（类型为 vx_size[]）；
- **view_end**：该参数表示要读取/写入 Tensor 的结束位置（类型为 vx_size[]）；
- **stride_size**：该参数表示读取/写入 Tensor 的跨度（类型为 vx_size[]），数组每个元素需要按如下赋值。

```
stride_size[0] = sizeof(element dtype)
stride_size[1] = dims[0] * stride_size[1]
stride_size[2] = dims[1] * stride_size[2]
stride_size[3] = dims[2] * stride_size[3]
```

3.3.2 读取参数

目前自定义算子的参数支持标量（VX_TYPE_SCALAR）和数组（VX_ARRAY）两种，和 vx_tensor 一样可以根据定义直接将 vx_reference 转为 vx_scalar 或 vx_array 类型。

1） 读取 vx_scalar 类型参数

vx_scalar 表示标量类型（参考 OpenVX 手册 Object:Scalar），包括的类型如下所示：

原始类型	OpenVX 类型	vx_type_e 枚举值
char	vx_char	VX_TYPE_CHAR
signed char	vx_int8	VX_TYPE_INT8
unsigned char	vx_uint8	VX_TYPE_UINT8
short	vx_int16	VX_TYPE_INT16
unsigned short	vx_uint16	VX_TYPE_UINT16
int	vx_int32	VX_TYPE_INT32
unsigned int	vx_uint32	VX_TYPE_UINT32
long long	vx_int64	VX_TYPE_INT64
unsigned long long	vx_uint64	VX_TYPE_UINT64
	vx_float16	VX_TYPE_FLOAT16
float	vx_float32	VX_TYPE_FLOAT32

double	vx_float64	VX_TYPE_FLOAT64
int	vx_enum	VX_TYPE_ENUM
unsigned int	vx_size	VX_TYPE_SIZE
	vx_bool	VX_TYPE_BOOL

OpenVX 提供了接口能够对 vx_scalar 对象进行查询和读写，主要有：

- **vxCreateScalar**：创建 vx_scalar 对象；
- **vxCreateScalarWithSize**：创建 vx_scalar 对象（有对象大小参数）；
- **vxReleaseScalar**：释放 vx_scalar 对象；
- **vxQueryScalar**：查询 vx_scalar 对象（支持查询 vx_scalar 的类型）；
- **vxCopyScalar**：读写 vx_scalar 对象；
- **vxCopyScalarWithSize**： 读写 vx_scalar 对象（有对象大小参数）。

以下为 vx_scalar 的示例代码：

```
vx_enum scalar_type;
vx_bool align_corners = 0;
vxQueryScalar((vx_scalar)parameters[3], VX_SCALAR_TYPE,
                &scalar_type, sizeof(vx_enum));
vxCopyScalar((vx_scalar)parameters[3], &align_corners,
                VX_READ_ONLY , VX_MEMORY_TYPE_HOST);
```

2） 读取 vx_array 类型参数

OpenVX 使用 vx_array 类型表示数组（参考 OpenVX 手册 Object:Array），对应的类型枚举值为 VX_TYPE_ARRAY。OpenVX 同样提供了一系列函数能够对 vx_array 对象进行查询和读写，主要有：

- **vxCreateArray**：创建 vx_array 对象；
- **vxReleaseArray**：释放 vx_array 对象；
- **vxQueryArray**：查询 vx_array 对象（支持查询数组项的类型、数组项的数量、数组容量以及数组项的大小）；
- **vxMapArrayRange**：映射 vx_array 对象中的一段范围给用户访问。

以下为 vx_array 的示例代码：

```
int32_t size[2];
vx_enum array_item_type;
vx_size array_item_num;

vxQueryArray((vx_array)parameters[2], VX_ARRAY_ITEMTYPE,
                &array_item_type, sizeof(vx_enum));
vxQueryArray((vx_array)parameters[2], VX_ARRAY_NUMITEMS,
```

```
                    &array_item_num, sizeof(vx_size));

vxCopyArrayRange((vx_array)parameters[2], 0, 2, sizeof(int32_t), (void *)size,
                        VX_READ_ONLY, VX_MEMORY_TYPE_HOST);
```

注意，当前自定义算子的 vx_array 的 item_type 都是 int32_t 类型。

3.3.3 编写算子实现代码

获取算子输入 Tensor 的数据与参数之后，用户可以编写算子的实现代码。使用输入 Tensor 数据时需要注意算子的输入 Tensor 数据类型（因为自定义算子不会被量化，所以输入 Tensor 数据类型都为 **float16**）和排列顺序（默认为 **NCHW**）。

开发者也可以调用 OpenVX 提供的接口函数（参考 OpenVX 手册 Vision Function），需要注意的是，以 vxu 开头的函数才能直接调用得到结果。

3.4 编写 VX Kernel

可以通过编写 VX Kernel 使用 NPU 中的 PPU 模块进行加速。生成的代码中 C 代码包含对 VX kernel 的初始化、反初始化、参数验证等回调函数；vx 扩展名的文件为 kernel 函数。

注意，编写 VX Kernel 需要具备一定的 OpenCL 开发经验。

3.4.1 初始化 Kernel

初始化函数中，开发者需要根据自己需求来配置 Kernel 的执行参数。VX Kernel 执行参数和 OpenCL 的 NDRange 类似，都有 work-item 和 work-group 概念：

- **work-item**：工作线程，每个 Work-item 有唯一的全局 ID（3 个维度[x, y, z]）。
- **work-group**：由一个或多个 work-item 组成，是线程切换的基本单元。

VX Kernel 执行参数的定义如下所示

```
typedef struct _vx_kernel_execution_parameters {
    vx_uint32 workDim;
    vx_size globalWorkOffset[VX_MAX_WORK_ITEM_DIMENSIONS];
    vx_size globalWorkScale[VX_MAX_WORK_ITEM_DIMENSIONS];
    vx_size localWorkSize[VX_MAX_WORK_ITEM_DIMENSIONS];
```

```
    vx_size globalWorkSize[VX_MAX_WORK_ITEM_DIMENSIONS];
} vx_kernel_execution_parameters_t;
```

参数	类型	描述
workDim	vx_uint32	work-item 的维度，有效值为 1，2，3。
GlobalWorkOffset[i]	vx_size	work-item[i]的全局 ID 的初始偏移。
globalWorkScale[i]	vx_size	work-item[i]的全局 ID 的步进值。
localWorkSize[i]	vx_size	work-group 的大小。即一个 work-group 包含多少个 work-item。
globalWorkSize[i]	vx_size	work-item 总的大小（globalWorkSize 必须是 localWorkSize 的整数倍）。

3.4.2 读取 Tensor 数据

在 VX Kernel 中读取输入 Tensor 数据可以使用 VXC_ReadImage2DArray 函数。

函数	VXC_ReadImage2DArray(Dest, Image, Coord, Offset, Info)
功能	该函数可以从图像中读取最多 128bit 数据，也可以用于从 3 维 Tensor 中读取数据。
参数	Dest：读取数据存放位置。
	Image：要读取的 Image（Tensor）。
	Coord：要读取的坐标（int4）。
	Offset：XY 的偏移位置，可以使用以下宏函数设置： VXC_5BITOFFSET_XY(offsetX, offsetY) 其中 offsetX 和 offsetY 值范围是-16~15。
	Info：控制信息，可以使用以下宏函数设置： VXC_MODIFIER (StartBin, EndBin, SourceBin, RoundingMode, Clamp) ● StartBin：目标存放的起始位置 ● EndBin：目标存放的结束位置 ● SourceBin：源起始位置 ● RoundingMode：舍入模式，可以设置的值有（VXC_RM_TowardZero：向下取整；VXC_RM_TowardInf：向上取整；VXC_RM_ToNearestEven：最近取整）

	● Clamp：目标类型较小时是否使用 Clamp 进行数据缩短。0 为 truncated；1 为 Clamp。
返回值	void

以下为示例代码：

```
int4 Coord0 = int4(0,0,0,0);
VXC_ReadImage(Dst0, Image, Coord0, VXC_5BITOFFSET_XY(0,0),
                    VXC_MODIFIER(0, 15, 0, VXC_RM_TowardZero, 0));
```

假设输入的 Image 为 3 维的 Tensor（NCHW），那么上述代码将从第一个通道的第一行读取 16 个 Dest0 类型的数据。

3.4.3 写入 Tensor 数据

在 VX Kernel 中读取输入 Tensor 数据可以使用 VXC_WriteImage2DArray 函数。

函数	VXC_WriteImage2DArray (Dest, Image, Coord, Offset, Info)
功能	该函数可以写入最多 128bit 数据到图像，也可以写入数据到 3 维 Tensor 中数据。
参数	Image：要写入的 Image（Tensor）对象。
	Coord：要写入的坐标（int4）。
	Color：写入的数据。
	Info：控制信息，可以使用以下宏函数设置： VXC_MODIFIER (StartBin, EndBin, SourceBin, RoundingMode, Clamp) ● StartBin：目标存放的起始位置 ● EndBin：目标存放的结束位置 ● SourceBin：源起始位置 ● RoundingMode：舍入模式，可以设置的值有（VXC_RM_TowardZero：向下取整；VXC_RM_TowardInf：向上取整；VXC_RM_ToNearestEven：最近取整） ● Clamp：目标类型较小时是否使用 Clamp 进行数据缩短。0 为 truncated；1 为 Clamp。

返回值	void

以下为示例代码

```
int4 Coord0 = int4(0,0,0,0);
VXC_WriteImage(Image, Coord0, Val0,
                    VXC_MODIFIER(0, 15, 0, VXC_RM_TowardZero, 0));
```

假设输出的 Image 为 3 维的 Tensor（NCHW），那么上述代码将 16 个 Val0 类型的数据写入第一个通道的第一行。

3.4.4 编写算子实现代码

除了 Tensor 的读取和写入函数，VX Kernel 的语法和 OpenCL1.2 Kernel 的编写基本一致，开发者可以参考 OpenCL1.2 官方指南来编写算子的实现代码。使用输入 Tensor 数据时需要注意算子的输入 Tensor 数据类型（因为自定义算子不会被量化，所以输入 Tensor 数据类型都为 **float16**）和排列顺序（默认为 **NCHW**）。