百度智能云

FPGA 文档

目录

产品描述

介绍

FPGA(Field Programmable Gate Array)云服务器是云环境中配备FPGA的计算实例，通过购买FPGA实例，您可以结合自身业务场景，利用百度智能云提供的完善的FPGA加速平台，以及配套的开发、模拟、调试、编译资源，快速地为您的业务构建专属的FPGA硬件加速程序。您也可以从百度智能云直接获取已设计好的、适配您业务的FPGA加速实例，为您有加速需求的业务选择高性能低成本的解决方案。

近些年，FPGA在互联网行业逐渐被应用起来，涉及人脸识别、语音识别、智能家居、智能交通、基因测序、视频、图像、文本数据处理等众多领域。

百度有将近9年FPGA加速器研发和大规模部署的经验，技术处于世界领先水平。百度在FPGA加速领域的论文发表在ASPLOS 2014（国内第二篇，并获最佳论文提名），EUROSYS2014，Hotchips 2014/2016/2017等顶级国际会议。FPGA云服务器大量复用百度内部先进及成熟的技术，您只需要关注自己的业务功能逻辑，即可快速开发出高性能的专属FPGA硬件加速程序。百度也将开放内部的FPGA加速IP及服务，同时会建立第三方云市场，帮助您更快更好地搭建自己的FPGA应用。

基于百度智能云提供的FPGA云服务器，用户可以在管理控制台便捷创建和管理FPGA实例，包括配置云服务器名称、网络IP，管理镜像等。

FPGA云服务器目前正在进行白名单邀测，欢迎提交白名单申请。

优势

超高计算性能

• 每个FPGA实例配备16个CPU核心、64G内存和450G高速本地磁盘 • FPGA加速平台搭建20nm Xilinx XCKU115 FPGA，该FPGA芯片大约包含 150 万个逻辑元件，5520 个数字信号处理 (DSP) 引擎 • FPGA加速平台配备4通道DDR4，每个通道72bit，支持ECC，4个通道共8GB容量；2400MHz速率，提供76GB/s的访存带宽 • FPGA加速平台通过PCIE 3.0 x 8与虚机通信，通信带宽高达8GB/s

安全稳定

• 每个FPGA实例独享一个FPGA加速平台，不会在实例、用户之间共享 • 用户可以将FPGA实例当做普通云主机进行管理，支持指定VPC创建，支持使用安全组进行访问控制 • FPGA实例支持实时的硬件资源使用量、等待队列的平均长度和硬件温度等监控，用户可实时了解硬件使用情况，应对突发情况降低风险

镜像

镜像是云服务器实例运行环境的模板，包括操作系统和预装软件等配置信息。百度智能云为FPGA用户默认提供了专属公共镜像，供在创建时选择：

- FPGA标准开发环境
- CNN图像分类服务
- RSA解密加速服务

操作指南

快速入门

FPGA 云服务器提供完全的设备管理权限和全生命周期的运维管理服务。本文旨在帮助您快速在百度智能云管理控制台购买和管理FPGA云服务器。

FPGA 实例操作流程如下所示：

1. 实名认证及注册百度智能云账号；
2. 申请开通FPGA服务；
3. 创建FPGA实例；
4. 选择适合的镜像类型，部署镜像。

实名认证及注册

请根据自身情况，进行“企业认证”或者“个人认证”，具体请参考认证流程。

如果您在创建FPGA 实例的时候仍未完成实名认证，你可以点击页面上的『认证』提示按钮到实名认证页面完成相关操作。

注册百度智能云账号，请参考注册百度账号，完成注册操作。

开通FPGA

1. 登录百度智能云FPGA官网。
2. 填写FPGA公测申请。

提交申请后，工作人员会对您的信息进行审核，会为符合试用条件的用户开通 FPGA 服务。开通后，登录控制台界面即可查看服务器列表。

创建FPGA实例

操作步骤

1. 登录BCC管理控制台主界面。
2. 登录成功后，选择“产品服务>云服务器 BCC”，显示“实例列表”页面。
3. 点击<创建实例>，选择配置信息。

配置信息	说明
付费方式	预付费（包年包月）、后付费（按需购买）
当前地域	华南-广州、华东-苏州
可用区	可用区是指在同一区域下，电力和网络互相独立的区域，故障会被隔离在一个可用区内。
类型	FPGA 实例
配置	套餐支持16核64GB内存，Xilinx KU115 FPGA 1块
操作系统	支持6.5版本的FPGA操作系统
镜像类型	三种专属镜像：FPGA标准开发环境、CNN图像分类服务、RSA解密加速服务
本地磁盘	支持450G本地磁盘
网络类型	当前服务器所属的虚拟私有网络，缺省情况下系统默认私有网络。

设置实例名称和管理员密码，选择购买时长和数量。

4. 点击“下一步”，确认订单并支付。
5. 购买成功后，可以在“服务器实例列表”页查看。

管理FPGA实例

您可以对当前账户中的FPGA 开启、停止、重启、修改名称以及绑定公网IP等操作。

1. 登录控制台主界面。

2. 登录成功后，选择“产品服务>云服务器 BCC”，显示“实例列表”页面。
3. 查看当前云服务器详情。
4. （可选）绑定公网IP，点击当前FPGA 实例的公网IP图标，在弹框中选择需要绑定的弹性公网IP。如还未创建公网IP，请先购买，参见创建EIP流程。

FPGA标准开发环境

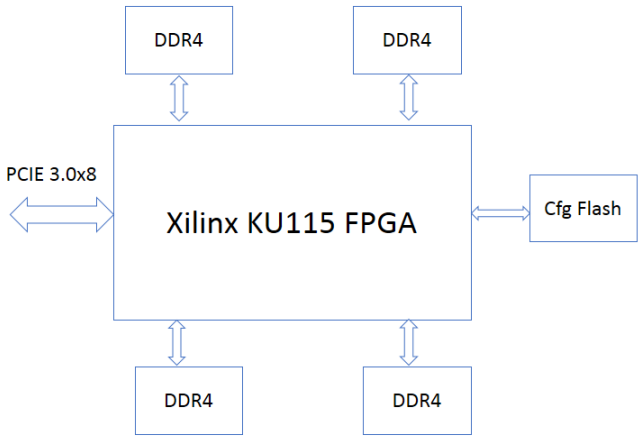
概述

概述

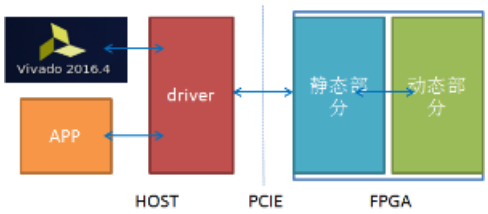
镜像是云服务器实例运行环境的模板，包括操作系统和预装软件等配置。百度智能云为每个FPGA实例默认提供了专属公共镜像，用户可以按需选择适合的镜像类型。

基于百度智能云自研的FPGA加速卡，提供了一套FPGA标准开发环境。您可以使用百度智能云提供的镜像工具包，在FPGA上开发与调试自己的业务功能，或者将已有的功能模块移植到FPGA加速卡上。

百度自研FPGA加速卡使用Xilinx 20nm KU115 FPGA。FPGA板卡带有4通道DDR4，每个通道72bit，带ECC，容量2GB，速率2400Mhz。FPGA通过PCIE 3.0x8和CPU相连。板卡的结构框图如下所示：



基于上面的FPGA板卡，百度还提供的FPGA标准开发环境，其系统结构如下图：



FPGA标准开发环境具有极大的灵活性：

- 您可以自行研发FPGA中动态部分的逻辑，包括KU115芯片的绝大部分资源，以及4个DDR4通道，让FPGA电路完成定制化的功能，
- 百度智能云提供驱动和应用参考设计，您只需修改软件侧的驱动和应用程序，调用FPGA完成特定的功能。
- 直接使用百度提供的工具包更换FPGA中动态部分的逻辑。
- FPGA标准开发环境提供虚拟jtag工具，您可以使用vivado工具对FPGA进行调试。

FPGA 标准开发环境操作包括两部分：

FPGA软件驱动开发

以运行支持PE进行简单浮点向量加功能的示例程序为例：

1. 编译驱动，提供编译示例程序。

2. 运行示例程序。

FPGA逻辑开发

使用工具包开发和调试用户逻辑：

1. 使用Baidu_HW_design_toolkit编译实现您的动态逻辑。

2. 使用bin_pr_tools更换您的动态逻辑。

3. 使用Vivado对您的动态逻辑进行调试。

FPGA软件驱动开发

FPGA软件驱动开发

编译驱动

修改driver/Makefile中的KERNELDIR变量，使之指向当前内核的编译目录，一般为/lib/modules/$(uname -r)/build目录或/usr/src/kernels/$(uname -r)。

```
# d52cbaca0ef8cf4fd3d6354deb5066970fb6511d02d18d15835e6014ed847fb0
obj-m += xdma_xvc.o
xdma_xvc-objs := xdma-core.o xdma-sgm.o xdma-ioctl.o xdma-bit.o xvc-pcie-driver.o xvc-mmconfig_64.o

KERNELDIR ?= /home/work/svn447/sys/sat/tags/linux2-6-32/linux2-6-32_1-12-0-0_PD_BL
PWD := $(shell pwd)
ROOT := $(dir $(M))

XILINXINCLUDE := -I$(ROOT)../include -I$(ROOT)/include

```

执行`make`，如果编译成功，当前目录下会生成xdma_xvc.ko驱动文件，如下图所示：

```
miaotianxiang@yf-inf-fpga-spark00 ~/x/driver> make
make -C /home/work/svn447/sys/sat/tags/linux2-6-32/linux2-6-32_1-12-0-0_PD_BL M=/home/miaotianxiang/xdma-xvc-driver/driver modules
make[1]: Entering directory `/home/work/svn447/sys/sat/tags/linux2-6-32/linux2-6-32_1-12-0-0_PD_BL'
  CC [M]  /home/miaotianxiang/xdma-xvc-driver/driver/xdma-core.o
  CC [M]  /home/miaotianxiang/xdma-xvc-driver/driver/xdma-sgm.o
  CC [M]  /home/miaotianxiang/xdma-xvc-driver/driver/xdma-ioctl.o
  CC [M]  /home/miaotianxiang/xdma-xvc-driver/driver/xdma-bit.o
  CC [M]  /home/miaotianxiang/xdma-xvc-driver/driver/xvc-pcie-driver.o
  CC [M]  /home/miaotianxiang/xdma-xvc-driver/driver/xvc-mmconfig_64.o
  CC [M]  /home/miaotianxiang/xdma-xvc-driver/driver/xvc-mmconfig-shared.o
  CC [M]  /home/miaotianxiang/xdma-xvc-driver/driver/xvc-core.o
  LD [M]  /home/miaotianxiang/xdma-xvc-driver/driver/xdma_xvc.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/miaotianxiang/xdma-xvc-driver/driver/xdma_xvc.mod.o
  LD [M]  /home/miaotianxiang/xdma-xvc-driver/driver/xdma_xvc.ko
make[1]: Leaving directory `/home/work/svn447/sys/sat/tags/linux2-6-32/linux2-6-32_1-12-0-0_PD_BL'
```

执行`insmod xdma_xvc.ko`，装载上一步生成的驱动文件，在/dev目录下会出现如下设备文件/dev/xil_xvc/cfg_ioc0。

```
[root@BBhost_test1514 sample]# ll /dev/xil_xvc/cfg_ioc0
crw-rw---- 1 root root 250, 8 Apr 11 13:38 /dev/xil_xvc/cfg_ioc0
[root@BBhost_test1514 sample]# ll /dev/xdma0_*
crw-rw---- 1 root root 250, 36 Apr 11 13:38 /dev/xdma0_c2h_0
crw-rw---- 1 root root 250,  1 Apr 11 13:38 /dev/xdma0_control
crw-rw---- 1 root root 250, 10 Apr 11 13:38 /dev/xdma0_events_0
crw-rw---- 1 root root 250, 11 Apr 11 13:38 /dev/xdma0_events_1
crw-rw---- 1 root root 250, 20 Apr 11 13:38 /dev/xdma0_events_10
crw-rw---- 1 root root 250, 21 Apr 11 13:38 /dev/xdma0_events_11
crw-rw---- 1 root root 250, 22 Apr 11 13:38 /dev/xdma0_events_12
crw-rw---- 1 root root 250, 23 Apr 11 13:38 /dev/xdma0_events_13
crw-rw---- 1 root root 250, 24 Apr 11 13:38 /dev/xdma0_events_14
crw-rw---- 1 root root 250, 25 Apr 11 13:38 /dev/xdma0_events_15
crw-rw---- 1 root root 250, 12 Apr 11 13:38 /dev/xdma0_events_2
crw-rw---- 1 root root 250, 13 Apr 11 13:38 /dev/xdma0_events_3
crw-rw---- 1 root root 250, 14 Apr 11 13:38 /dev/xdma0_events_4
crw-rw---- 1 root root 250, 15 Apr 11 13:38 /dev/xdma0_events_5
crw-rw---- 1 root root 250, 16 Apr 11 13:38 /dev/xdma0_events_6
crw-rw---- 1 root root 250, 17 Apr 11 13:38 /dev/xdma0_events_7
crw-rw---- 1 root root 250, 18 Apr 11 13:38 /dev/xdma0_events_8
crw-rw---- 1 root root 250, 19 Apr 11 13:38 /dev/xdma0_events_9
crw-rw---- 1 root root 250, 32 Apr 11 13:38 /dev/xdma0_h2c_0
crw-rw---- 1 root root 250,  0 Apr 11 13:38 /dev/xdma0_user
```

编译示例程序

进入sample目录，执行make。如果编译成功，当前目录下生成sample、sample_user_irq等可执行文件，参见下图：

```
miaotianxiang@yf-inf-fpga-spark00 ~/x/sample> make
g++ -c -std=c++0x -o reg_write_32.o reg_write_32.cpp -D_FILE_OFFSET_BITS=64 -D_GNU_SOURCE -D_LARGE_FILE_SOURCE
g++ -c -std=c++0x -o fpga_cloud.o fpga_cloud.cpp -D_FILE_OFFSET_BITS=64 -D_GNU_SOURCE -D_LARGE_FILE_SOURCE
g++ -lrt -lpthread -o reg_write_32 reg_write_32.o fpga_cloud.o -D_FILE_OFFSET_BITS=64 -D_GNU_SOURCE -D_LARGE_FILE_SOURCE
g++ -c -std=c++0x -o reg_read_32.o reg_read_32.cpp -D_FILE_OFFSET_BITS=64 -D_GNU_SOURCE -D_LARGE_FILE_SOURCE
g++ -lrt -lpthread -o reg_read_32 reg_read_32.o fpga_cloud.o -D_FILE_OFFSET_BITS=64 -D_GNU_SOURCE -D_LARGE_FILE_SOURCE
g++ -c -std=c++0x -o sample.o sample.cpp -D_FILE_OFFSET_BITS=64 -D_GNU_SOURCE -D_LARGE_FILE_SOURCE
g++ -lrt -lpthread -o sample sample.o fpga_cloud.o -D_FILE_OFFSET_BITS=64 -D_GNU_SOURCE -D_LARGE_FILE_SOURCE
g++ -c -std=c++0x -o sample_user_irq.o sample_user_irq.cpp -D_FILE_OFFSET_BITS=64 -D_GNU_SOURCE -D_LARGE_FILE_SOURCE
g++ -lrt -lpthread -o sample_user_irq sample_user_irq.o fpga_cloud.o -D_FILE_OFFSET_BITS=64 -D_GNU_SOURCE -D_LARGE_FILE_SOURCE
```

运行示例程序

执行`./sample`，输出如下结果，PE正确地执行了浮点向量加功能。sample使用轮询寄存器方式检查命令结果是否完成。

```
[root@BBhost_test1514 sample]# ./sample
float_a_array :         1.00    2.00    3.00    4.00    1.00    2.00    3.00    4.00
float_b_array :         1.00    1.00    1.00    1.00    1.00    1.00    1.00    1.00
float_c_array :         2.00    3.00    4.00    5.00    2.00    3.00    4.00    5.00
```

执行`./sample_user_irq`，输出如下结果，PE正确地执行了浮点向量加功能。sample_user_irq使用中断方式检查命令结果是否完成。

```
[root@BBhost_test1514 sample]# ./sample_user_irq
float_a_array :         1.00    2.00    3.00    4.00    1.00    2.00    3.00    4.00
float_b_array :         1.00    1.00    1.00    1.00    1.00    1.00    1.00    1.00
float_c_array :         2.00    3.00    4.00    5.00    2.00    3.00    4.00    5.00
```

关键代码示例

```
21
22      /* DMA float_a_array and float_b_array to FPGA */
23      ret = fpga_memcpy(0, 4096,
24      |   |   reinterpret_cast<uint64_t>(&float_a_array),
25      |   |   sizeof(float_a_array), 1);
26      assert(ret == 0);
27      ret = fpga_memcpy(0, 4096 * 2,
28      |   |   reinterpret_cast<uint64_t>(&float_b_array),
29      |   |   sizeof(float_b_array), 1);
30      assert(ret == 0);
31
32      /* wr reg float_a_addr */
33      ret = reg_write_32(0, 64 * 1024 + 4, float_a_addr);
34      assert(ret == 0);
35      /* wr reg float_b_addr */
36      ret = reg_write_32(0, 64 * 1024 + 8, float_b_addr);
37      assert(ret == 0);
38      /* wr reg float_c_addr */
39      ret = reg_write_32(0, 64 * 1024 + 12, float_c_addr);
40      assert(ret == 0);
41      /* wr reg float_array_len */
42      ret = reg_write_32(0, 64 * 1024 + 16, 8);
43      assert(ret == 0);
44      /* wr reg start_pe */
45      ret = reg_write_32(0, 64 * 1024, 1);
46      assert(ret == 0);
47
```

输入数据

写寄存器

```
48      /* rd reg polling_pe_status */
49      uint32_t status;
50      do {
51      |   ret = reg_read_32(0, 64 * 1024 + 36, &status);
52      |   assert(ret == 0);
53      } while (status == 0);
54
55      /* DMA float_c_array from FPGA */
56      ret = fpga_memcpy(0,
57      |   |   reinterpret_cast<uint64_t>(&float_c_array), 4096 * 3,
58      |   |   sizeof(float_c_array), 0);
59      assert(ret == 0);
60
```

轮询状态寄存器

输出数据

```
56      {
57      |   char buf[4];
58      |   int fd = open("/dev/xdma0_events_0", O_RDONLY);
59      |   assert(fd > 0);
60      |   ret = read(fd, buf, sizeof(buf));
61      |   assert(ret != -1);
62      |   close(fd);
63      }
64
65      /* DMA float_c_array from FPGA */
66      ret = fpga_memcpy(0,
67      |   |   reinterpret_cast<uint64_t>(&float_c_array), 4096 * 3,
68      |   |   sizeof(float_c_array), 0);
69      assert(ret == 0);
```

在xdma0_events_0上阻塞读等待中断

输出数据

FPGA逻辑开发

FPGA逻辑开发

使用Baidu_HW_design_toolkit编译实现您的动态逻辑

“Baidu_HW_design_toolkit”工具包，帮助将您开发的动态逻辑实现在FPGA中。

“Baidu_HW_design_toolkit”提供FPGA硬件逻辑所需的环境，只需将自己逻辑所需的相关的文件（如rtl代码，ip核，xdc约束等）放入指定的路径，然后执行脚本，即可生成用于烧写FPGA云服务器的逻辑镜像文件。

“Baidu_HW_design_toolkit”包含了三个子文件夹，build，common_files和usr_files.

- usr_files存放用户的工程设计文件。
- common_files存放FPGA云服务镜像工程的一些通用设计。如静态逻辑的dcp，ddr约束等。通常情况下，不建议您修改common_files目录中的内容。
- build存放制作FPGA云服务器逻辑镜像所要执行的脚本，如果您具备丰富的FPGA开发经验，可以根据自己的需要修改脚本。例如，用更加适合的布局布线策略管理您的工程实现。

“Baidu_HW_design_toolkit”提供了两种流程制作FPGA云服务逻辑镜像，需要准备不同的设计文件:

1. Non_IPI流程

这种方式比较类似传统的FPGA工程实现方式，您需要准备好动态部分逻辑（也就是rp_bd_wrapper.rp_bd_i）的设计文件放入usr_files指定的目录，然后执行build目录下的run_nonIPI.tcl脚本。

2. IPI流程

这种方式采用vivado IP Integrator制作云服务逻辑镜像的动态部分逻辑（也就是rp_bd_wrapper.rp_bd_i）。你需要准备好IPI的设计文件放入usr_files和build下指定的目录，然后执行build目录下的run_IPI.tcl脚本。

使用bin_pr_tools更换您的动态逻辑

“bin_pr_tools”工具包，是更换FPGA动态部分逻辑的必要工具。在使用该工具包前，您需要确保FPGA的驱动程序已经加载。然后运行bin_pr_tools目录下的”load_pr_bin.sh”脚本即可更换您的动态部分逻辑。

```
 $sudo sh load_pr_bin.sh base ./ver2/ver2_pr_region_partial.bin
OK set decouple! ...
OK loading clear bin! ...
OK loading pr region bin! ...
OK unset decouple! ...
OK soft reset rp_bd ...
successfully load custom bitstream!
partial clear bin: ./base/base_pr_region_partial_clear.bin
partial bin: ./ver2/ver2_pr_region_partial.bin
found clear bin base_pr_region_partial_clear.bin in the current partial bin file’s directory
copy bin base_pr_region_partial_clear.bin into ‘last_clear_bin’ directory
```

> **注意：**
>
> 由于更换动态部分逻辑时，需要写入当前动态逻辑对应的clear bin，您务必保存好clear bin文件，以便下次更新动态逻辑时使用。同时bin_pr_tools工具包也会保存新动态逻辑对应的clear bin文件。

使用Vivado对您的动态逻辑进行调试

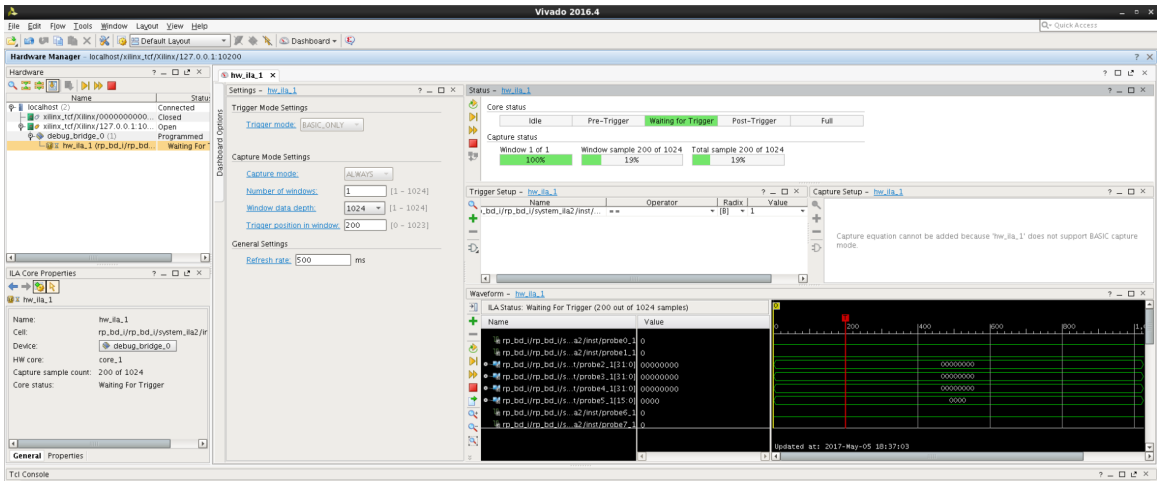
百度智能云提供工具包类似日常使用vivado操作，对您的动态逻辑进行调试。

在使用该工具包前，您需要确保FPGA的驱动程序已经加载。

1. 打开xvc_server工具包，运行xvc_pcie服务。

2. 使用vivado工具，仅需几步就可以通过虚拟jtag识别FPGA设备。

3. 选择动态逻辑对应的probe文件，类似使用Vivado工具，对工程中的ila和vio进行功能调试和信号查看。



FPGA示例工程说明

FPGA示例工程说明

概述

为方便您掌握FPGA云服务器的使用流程，快速创建自己定制的加速卡逻辑，百度智能云提供一个demo工程作为示例。

该demo工程支持了基于FPGA云服务器开发的几个基础功能，主要包括：

- 工程分成静态和动态两部分逻辑，支持基于pcie总线的partial reconfiguration开发及配置流程。
- 静态逻辑支持pcie-3.0-8x xdma，并提供了配套的driver。用户不能修改也无需关注静态部分的逻辑。
- 动态逻辑为用户自定制部分，用户需基于当前提供的接口实现所需功能逻辑。demo工程中的动态逻辑是一个element-wise向量加法模块，基于HLS开发。接口包括：
- 一个axi slave（256bit）和一个axi lite slave（32bit）接口，可分别用于传输逻辑所需的数据和控制命令。
- 4个axi master（512bit），用于连接DDR MIG控制器（可选）。
- 中断、时钟。
- 支持基于pcie总线的ila debug，可在云服务器上的vivado中抓取信号波形进行调试。

您可根据此demo工程的结构及提供的配套脚本了解fpga云服务器的开发流程，并以该工程为基础，修改其中的动态逻辑，实现所需的其他功能。

工程结构

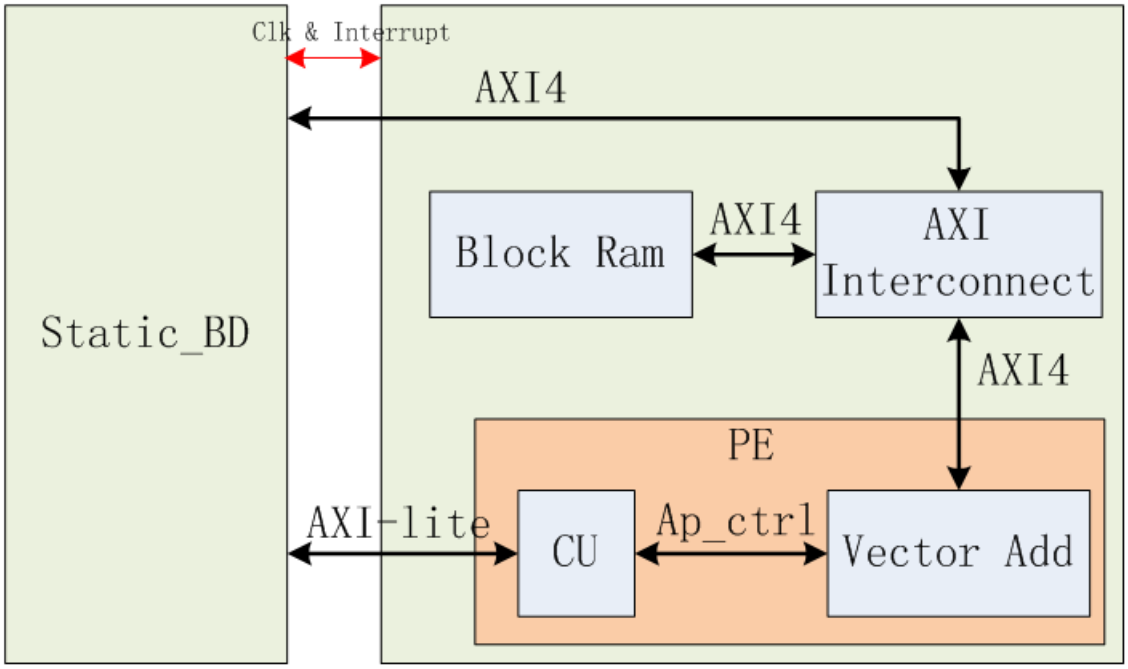
demo工程主要包含了两个部分，分别是static_bd_wrapper和rp_bd_wrapper。

其中static_bd_wrapper属于工程的静态部分，提供了pcie xdma，基于pcie的debug模块，flash控制器等。静态部分的逻辑不暴露给用户，用户不能修改也不用关心静态部分的逻辑。

rp_bd_wrapper则是动态逻辑，这部分逻辑中有rp_bd和其他一些组件。其中只有rp_bd是用户可以修改的内容。其他组件主要用于支持用户利用虚拟jtag进行调试或其他功能，这些组件不需要用户关心，用户不能修改。

rp_bd通过两组AXI总线与static_bd_wrapper传输数据，可以此为基础实现您所需的功能。

demo工程的rp_bd结构框图如下：



模块	说明
Block Ram	rp_bd内部有一个64KB的block ram用来存储计算所用的数据和计算结果，这个block ram可同时被host和卡上的用户逻辑访问，这是通过rp_bd中的一个AXI Interconnect实现的。
AXI Interconnect	AXI Interconnect用于协调两个AXI master访问rp_bd中的Block Ram；static_bd内的xdma输出的AXI4连接到AXI Interconnect的一个slave端口，demo工程提供的drive支持host通过dma访问这个block ram；PE内的Vector Add模块输出的AXI4连接到AXI Interconnect的另一个端口，使得用户逻辑也可以访问这个block ram。
CU	命令处理单元，static_bd输出的AXI-lite接口连接到PE中的CU模块，CU模块解析从AXI lite收到的命令，并产生符合ap_ctrl总线的请求信号与Vector Add模块相连。ap_ctrl是通过HLS综合出的逻辑模块采用的一种标准状态控制总线。有关其详细介绍可以参考Xilinx ug902文档。
Vector Add	计算处理单元，Vector Add模块完成向量加法运算，他是使用HLS高级综合工具开发的，它的控制输入为一组HLS模块使用的ap_ctrl信号；他使用AXI总线协议将Block Ram中的数据读出，进行加法运算后，将数据写回Block Ram中。

PE工作流程

1. 软件发起dma_to_dev将输入向量A，B拷贝至dev；A，B的长度必须8个float数据对齐。（单精度浮点数）
2. 软件通过配置寄存器发起PE计算指令，然后等待PE计算完成。
3. PE计算完成后，通过中断通知CPU上的软件驱动程序。
4. 软件发起dma_from_dev将输出向量C拷贝至host。（单精度浮点数）

CNN图像分类服务

概述

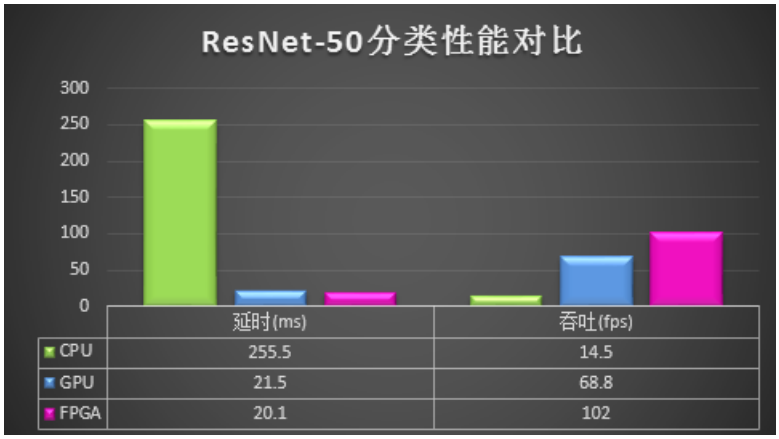
概述

FPGA具有低功耗，低延时，高性能的特点，在深度学习计算领域有很广阔的应用前景。FPGA从2013年开始就应用在许多典型的深度学习模型中，如DNN，RNN，CNN，LSTM等，涵盖了语音识别，自然语言处理，推荐算法，图像识别等广泛的应用领域。

FPGA云服务器提供了基于FPGA的深度卷积神经网络加速服务，单卡提供约3TOPs的定点计算能力，支持典型深度卷积网络算子，如卷积、逆卷积、池化、拼接、切割等。有效加速典型网络结构如VggNet、GoogLeNet、ResNet等。

我们基于FPGA的深度学习硬件，定制优化了主流深度学习平台，如caffe等，您可以直接将深度学习业务切换到FPGA平台，而无需考虑底层硬件细节。

通过使用预先训练好的ResNet-50进行图像分类性能的对比测试，在CPU、GPU和FPGA三类设备的物理机上的测试结果如下：



其中：

- CPU为Intel Xeon CPU E5-2650 v3，使用OpenBLAS占用4核进行测试
- GPU为Nvidia Tesla K40m

测试结果可以明显看到FPGA在时延上跟GPU差不多，比CPU快了超过12倍，在吞吐上FPGA更是比GPU超出1.5倍和比CPU超出近7倍。

CNN图像分类API接口

CNN图像分类API接口

设备管理接口

查询FPGA设备通道数量

定义：int get_fpga_tunnel_num()
功能：获取FPGA设备通道数量
参数：无
返回：当前主机FPGA设备上独立通道数量

初始化FPGA设备句柄

定义：init_fpga_handle(FpgaHandle& fh, const FpgaHandle::Mode mode = FpgaHandle::FPGA_MODE, const int tunnel = 0)
功能：初始化FPGA设备句柄
参数：

参数	说明
fh	FPGA设备句柄
mode	计算模式，取值范围`FPGA_MODE`或者`CPU_MODE`，默认FPGA模式
tunnel	使用设备通道，机器上FPGA设备有两个可以并行的独立计算通道，默认使用0号通道

返回：成功返回0，失败返回-1

释放FPGA设备句柄

定义：int free_fpga_handle(FpgaHandle& fh)
功能：释放FPGA设备句柄
参数：

参数	说明
fh	FPGA设备句柄

返回：成功返回0，失败返回-1

图像分类相关接口

加载训练好的CNN模型

定义：int cnn_fpga_load(FpgaHandle& fh, const std::string& cnn_proto, const std::string& cnn_model, const std::string& cnn_means, const std::string& cnn_labels)
功能：加载训练好的CNN模型
参数：

参数	说明
fh	FPGA设备句柄
cnn_proto	CNN模型定义prototxt文件路径
cnn_model	CNN模型caffemodel权值文件路径
cnn_means	图像均值文件路径
cnn_labels	图像标签文件路径

返回：成功返回0，失败返回-1

使用CNN模型分类图像

定义：int cnn_fpga_classify(FpgaHandle& fh, const cv::Mat& image, const int topk, std::vector& scores, std::vector<std::string>& labels)
功能：使用CNN模型分类图像
参数：

参数	说明
fh	FPGA设备句柄
image	输入图像
topk	概率最大的k个结果
scores	分类概率
labels	分类标签

返回：成功返回0，失败返回-1

释放CNN模型使用资源

定义：int cnn_fpga_free(FpgaHandle& fh)
功能：释放CNN模型使用资源
参数：

参数	说明
fh	FPGA设备句柄

返回：成功返回0，失败返回-1

CNN图像分类DEMO说明

CNN图像分类DEMO说明

目录结构如下：

fpga-cnn
|----demo

|----env.sh
|----fpga_demo.cpp
|----Makefile
|----run_fpga.sh
|----run.sh
|----models

|----ResNet-50

|----ResNet-50.caffemodel
|----ResNet-50.labels
|----ResNet-50.means.binaryproto
|----ResNet-50.prototxt
|----ResNet-50.test.jpg

|----dependency/
|----include/
|----lib/

编译方法

在fpga-cnn/demo目录下运行：make clean && make

运行方法

1.脚本用CPU模式运行ResNet-50分类
在fpga-cnn/demo目录下运行：sh run.sh

2.脚本用FPGA模式运行ResNet-50分类
在fpga-cnn/demo目录下运行：sh run_fpga.sh

3.通过fpga_demo运行
./fpga_demo mode case
其中，mode参数可以是cpu或者fpga，case参数为模型目录和文件名称

CNN图像分类DEMO

CNN图像分类DEMO

为了便于用户开发，FPGA 云服务器包装了CNN图像分类demo：

目录结构

fpga-cnn

|----demo
|----env.sh
|----fpga_demo.cpp
|----Makefile
|----run_fpga.sh
|----run.sh
|----models
|----ResNet-50
|----ResNet-50.caffemodel
|----ResNet-50.labels
|----ResNet-50.means.binaryproto
|----ResNet-50.prototxt
|----ResNet-50.test.jpg |----dependency/
|----include/
|----lib/

编译方法

在fpga-cnn/demo目录下运行：`make clean && make`

运行方法

1. 脚本用CPU模式运行ResNet-50分类

   在fpga-cnn/demo目录下运行：`sh run.sh`

2. 脚本用FPGA模式运行ResNet-50分类

   在fpga-cnn/demo目录下运行：sh run_fpga.sh

3. 通过fpga_demo运行 `./fpga_demo mode case`

   其中，mode参数可以是cpu或者fpga，case参数为模型目录和文件名称

RSA解密加速服务

概述

RSA算法是一种最广为使用的“非对称加密算法”，一般公钥／私钥长度越长，安全性就越好，计算也越复杂。百度智能云https改造中应用了RSA 2048加解密算法，针对高计算复杂度的RSA解密任务，我们运用FPGA上的并行计算资源和定制化的数据通路，提供了高达45000QPS的解密能力（是CPU单线程吞吐率的75倍以上，媲美商用ASIC加解密卡的吞吐率），同时还将提供独具特色的私钥管理方案，令系统安全性得到了质的提升。

软件调用API执行

编译

1. 执行`lspci | grep -i Xilinx`，输出非空，证实FPGA已被正确透传给虚拟机。
2. 编译驱动，进入`rsa-driver`目录，执行`make`。
   - 如提示“No such file or directory”，请修改`Makefile`中的`KERNELDIR`变量，使之指向正确的内核编译目录，一般为`/usr/src/kernels/$(uname -r)`。
   - 如编译时提示符号重定义，请删除源文件中的`PDE_DATA`、`file_inode`、`kvalloc`、`kvfree`等符号。
3. 加载驱动，执行`insmod fpga_drive.ko`。

   检查`/dev/fpga0`的权限是否为0666，如过不是，请执行`chmod 666 /dev/fpga0`。
4. 在openssl系统engine目录创建到`rsa-api/output/so/libfpga_rsa_cpp.so`的软链接，即执行`ln -s /path/to/rsa-`

api/so/libfpga_rsa_cpp.so /usr/lib64/openssl/engines/libfpga_rsa_cpp.so。

5. 通过openssl标准engine接口使用RSA加速功能，在正确加载并初始化engine后即可通过RSA_private_encrypt、RSA_private_decrypt进行RSA私钥加解密。

   FPGA支持密钥长度在2048 bits以下的RSA私钥加解密。如给定密钥长度超出此范围，engine会转交CPU计算，此时性能等同于直接使用CPU处理。

```
#include ‹openssl/rsa.h›
#include ‹openssl/engine.h›
#include ‹openssl/err.h›

OpenSSL_add_all_algorithms();
ERR_load_crypto_strings();
ENGINE_load_dynamic();

/* load engine */
ENGINE *engine = ENGINE_by_id("fpga_rsa_cpp");
if (engine == NULL) {
    LOG(WARNING) ‹‹ "Could not Load fpga_rsa_cpp Engine!";
    return 1;
}
LOG(INFO) ‹‹ "fpga_rsa_cpp Engine successfully loaded";

/* init engine */
int init_ret = ENGINE_init(engine);
int set_ret = ENGINE_set_default_RSA(engine);
LOG(INFO) ‹‹ "engine name = " ‹‹ ENGINE_get_name(engine);
LOG(INFO) ‹‹ "init_ret = " ‹‹ init_ret;
LOG(INFO) ‹‹ "set_ret = " ‹‹ set_ret;

if ((init_ret != 1) || (set_ret != 1)) {
    LOG(WARNING) ‹‹ "Failed to init engine";
    return 1;
}

/* use engine */
RSA_private_decrypt(flen, from, to, rsa, padding);
```

性能测试

qps

执行openssl speed rsa2048 -engine fpga_rsa_cpp -multi 36，在"sign/s"一栏中可以看到qps。正常情况应在40000/s以上。

```
[root@cq02-bcc-online-com2-102.cq02.baidu.com rsa-tools_1-0-1-1_PD_BL]# openssl speed rsa2048 -engine fpga_rsa_cpp -multi 36 2>/dev/null | grep -v -E
"Got|Fork"
 ------------------------------ engine_fpga_init ------------------------------
|   /$$$$$$$$ /$$$$$$$   /$$$$$$   /$$$$$$        /$$$$$$$   /$$$$$$   /$$$$$$   |
|   | $$_____/| $$__  $$ /$$__  $$ /$$__  $$      | $$__  $$ /$$__  $$ /$$__  $$  |
|   | $$      | $$  \ $$| $$  \__/| $$  \ $$      | $$  \ $$| $$  \__/| $$  \ $$  |
|   | $$$$$   | $$$$$$$/| $$ /$$$$| $$$$$$$$      | $$$$$$$/|  $$$$$$ | $$$$$$$$  |
|   | $$__/   | $$____/ | $$|_  $$| $$__  $$      | $$__  $$ \____  $$| $$__  $$  |
|   | $$      | $$      | $$  \ $$| $$  | $$      | $$  \ $$ /$$  \ $$| $$  | $$  |
|   | $$      | $$      |  $$$$$$/| $$  | $$      | $$  | $$|  $$$$$$/| $$  | $$  |
|   |__/      |__/       \______/ |__/  |__/      |__/  |__/ \______/ |__/  |__/  |
 ------------------------------ engine_fpga_init ------------------------------
OpenSSL 1.0.0-fips 29 Mar 2010
built on: Tue May 29 18:16:48 BST 2012
options:bn(64,64) md2(int) rc4(16x,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: gcc -fPIC -DOPENSSL_PIC -DZLIB -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -DKRB5_MIT -m64 -DL_ENDIAN -DTERMIO -Wall -O2 -g -p
ipe -Wall -Wp,-D_FORTIFY_SOURCE=2 -fexceptions -fstack-protector --param=ssp-buffer-size=4 -m64 -mtune=generic -Wa,--noexecstack -DMD32_REG_T=int -DOP
ENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DMD5_ASM -DAES_ASM -DWHIRLPOOL_ASM
                  sign    verify    sign/s verify/s
rsa 2048 bits 0.000022s 0.000007s  45448.2 136141.4
```

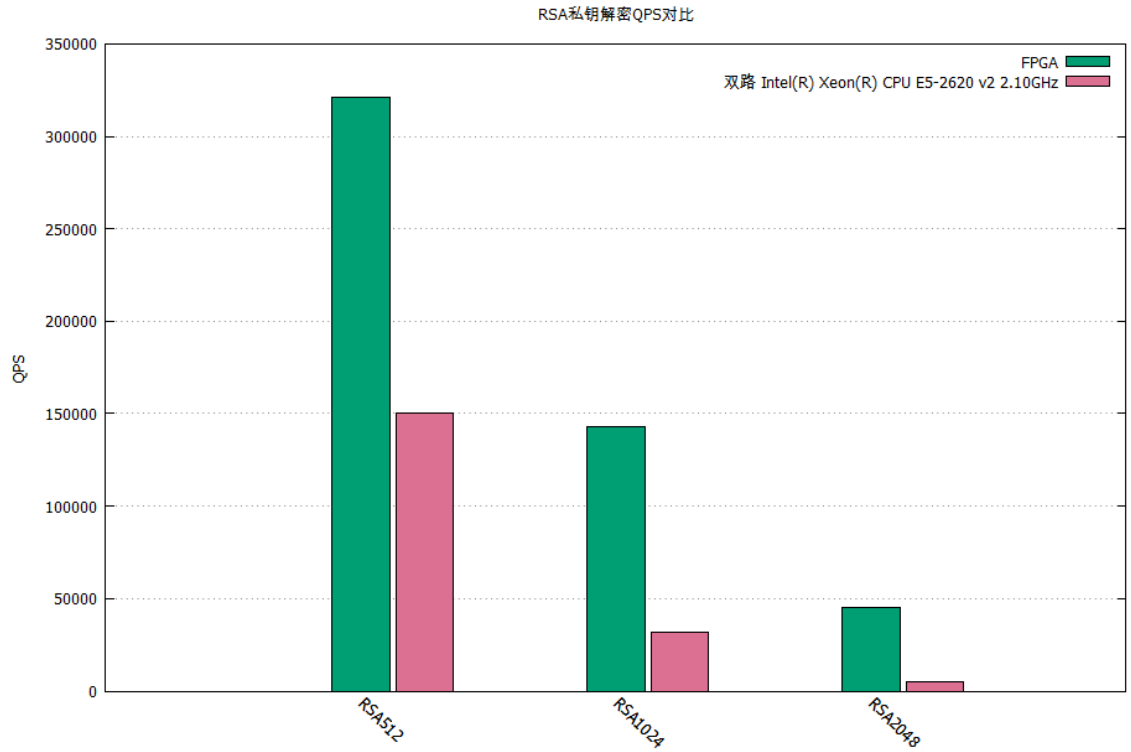
latency

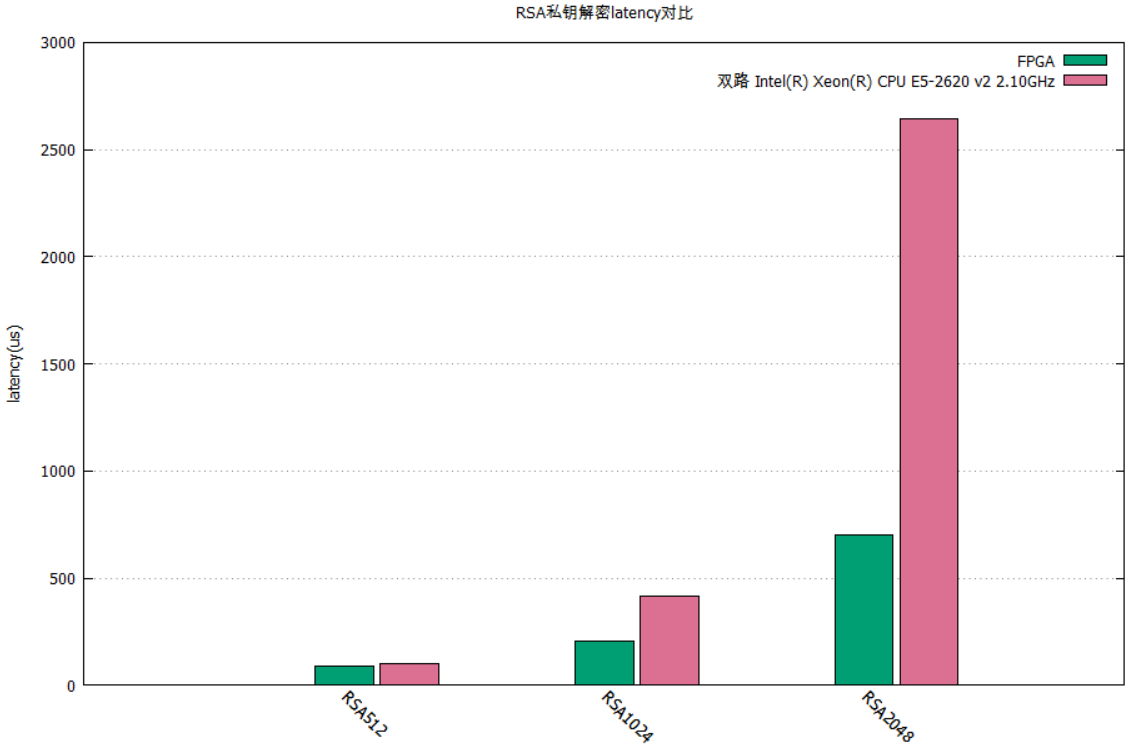
执行`openssl speed rsa2048 -engine fpga_rsa_cpp -multi 1`，在"sign"一栏中可以看到latency。正常情况应在700us左右。

```
[root@cq02-bcc-online-com2-102.cq02.baidu.com rsa-tools_1-0-1-1_PD_BL]# openssl speed rsa2048 -engine fpga_rsa_cpp -multi 1 2>/dev/null | grep -v -E "
Got|Fork"
 ------------------------------ engine_fpga_init ------------------------------
|   /$$$$$$$$ /$$$$$$$   /$$$$$$   /$$$$$$        /$$$$$$$   /$$$$$$   /$$$$$$   |
|   | $$_____/| $$__  $$ /$$__  $$ /$$__  $$      | $$__  $$ /$$__  $$ /$$__  $$  |
|   | $$      | $$  \ $$| $$  \__/| $$  \ $$      | $$  \ $$| $$  \__/| $$  \ $$  |
|   | $$$$$   | $$$$$$$/| $$ /$$$$| $$$$$$$$      | $$$$$$$/|  $$$$$$ | $$$$$$$$  |
|   | $$__/   | $$____/ | $$|_  $$| $$__  $$      | $$__  $$ \____  $$| $$__  $$  |
|   | $$      | $$      | $$  \ $$| $$  | $$      | $$  \ $$ /$$  \ $$| $$  | $$  |
|   | $$      | $$      |  $$$$$$/| $$  | $$      | $$  | $$|  $$$$$$/| $$  | $$  |
|   |__/      |__/       \______/ |__/  |__/      |__/  |__/ \______/ |__/  |__/  |
 ------------------------------ engine_fpga_init ------------------------------
OpenSSL 1.0.0-fips 29 Mar 2010
built on: Tue May 29 18:16:48 BST 2012
options:bn(64,64) md2(int) rc4(16x,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: gcc -fPIC -DOPENSSL_PIC -DZLIB -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -DKRB5_MIT -m64 -DL_ENDIAN -DTERMIO -Wall -O2 -g -p
ipe -Wall -Wp,-D_FORTIFY_SOURCE=2 -fexceptions -fstack-protector --param=ssp-buffer-size=4 -m64 -mtune=generic -Wa,--noexecstack -DMD32_REG_T=int -DOP
ENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DMD5_ASM -DAES_ASM -DWHIRLPOOL_ASM
                  sign    verify    sign/s verify/s
rsa 2048 bits 0.000704s 0.000096s   1420.5  10416.7
```

RSA私钥解密QPS对比

FPGA解决方案与使用双路Intel Xeon E5-2620 v2服务器（超线程开启，共24核）进行纯CPU计算相比，性能优势十分明显，如下图所示。当RSA密钥长度为512 bits、1024 bits和2048 bits时，前者的私钥解密QPS分别是后者的2.13、4.52和9.36倍，latency仅为后者的89%、50%和27%。




RSA私钥解密latency对比
FPGA
双路 Intel(R) Xeon(R) CPU E5-2620 v2 2.10GHz
latency(us)
3000
2500
2000
1500
1000
500
0
RSA512
RSA1024
RSA2048