Deep Learning with OpenCL Caffe

A Guide to Real-Time Inference using Python

Version 1.5.0

Deep Learning with OpenCL Caffe

A Guide to Real-Time Inference using Python

Version 1.5.0

Rahul Atlury and Fabian David Tschopp

Media Press

Atlanta, GA

Media Press
Checkpoint Ave
GA 30008

Background for Rahul Atlury:

Rahul Atlury holds a Master's degree in IT. He is a member of the Intel Channel Board of Advisors (BOA), IoT for the Asian region. He contributes to open source and helps in OS releases for the Intel Edison platforms. Previously he worked as a research associate at the University of Hyderabad and helped with setup and maintenance of the PRAGMA Grid from India. He has worked as a consultant with various companies including the Indian Defense agencies.

Background for Fabian Tschopp:

Fabian Tschopp holds a Bachelor's degree in computer science and is currently a neuroinformatics Master student at the University and ETH ZÃijrich. He is the developer and maintainer of the official BVLC Caffe OpenCL deep learning framework, in collaboration with HHMI Janelia, Intel and AMD. He has in-depth experience with visual computing, deep neural networks, 3D scanning and high performance computing. Fabian Tschopp is an Intel ISEF 2013 alumni and 2nd award winner.

Contents

Chapter 3

Installation and configuration

3.1 Introduction

Basic Linear Algebra Subprograms (BLAS) are the workhorses behind deep learning applications. They are low-level routines that provide the building blocks for performing common linear algebra operations such as, scalar multiplication, vector addition, dot products, linear combinations, and matrix multiplication.

BLAS libraries are highly optimized for specific hardware architectures. The routines in BLAS libraries consist of kernels written in assembly, or C. Kernels include a small-dimension matrix or vector operations also known as **"kernel operations"**. The kernel operations are loops that break down large matrix multiplications into smaller sequences. The loops determine the overall performance of the matrix-matrix product since most of the time is spent there.

Large (2000x2000) matrix multiplication operations are broken into a sequence of smaller (100x100) matrix multiplications. These fixed-size small-dimension operations are hard coded and optimized using intrinsics and specific assembly code to improve performance. The specific assembly code makes use of special floating point units found in devices consisting of features like:

- SIMD-type instructions
- Instruction level parallelism

- Cache-awareness

Different levels of specifications for these kernel operations exist that define the BLAS implementations. The different levels also correlate to both the chronological order and publication, as well as the algorithm complexities.

Level 1: Set of linear algebra routines that operate on vectors only.

Level 2: Set of matrix-vector operations including, a generalized matrix-vector multiplication **(GEMV)**.

Level 3: Set of matrix-matrix product operations including general matrix multiplication **(GEMM)**.

3.2 BLAS libraries

BLAS libraries can be **tuned** and **fused**.

Tuning is the process where parameters are changed to improve the kernel performance according to the hardware.

Fusing combines two kernels to improve performance by exchanging data of the algorithms in cache or local memory instead of CPU or GPU global memory.

Many open-source BLAS libraries exist today, some of the common ones include the following:

Library	Description
ViennaCL	OpenCL Context + BLAS
ISAAC	Tuned OpenCL BLAS
clBLAS	AMD OpenCL BLAS
CLBlast	C. Nugteren OpenCL BLAS

Table 3.1: List of few BLAS libraries

Essentially all of the above are just BLAS libraries. ViennaCL BLAS is very basic: easy to install, but a bit older and slower than state-of-the-art BLAS

libraries. CLBlast has been added to favor i.e. ARM chips (experimental). clBLAS is preferentially used on high-end desktop GPUs by AMD and nVidia.

LibDNN is a fused im2col+GEMM convolution and pooling library with tuning and JIT/RTC support. It has an extensive collection of fast convolution implementations. It is the core of the OpenCL Caffe framework. It handles kernel generation, caching, loading and launching.

im2col and GEMM are usually two separate kernels, and LibDNN fuses them. im2col is an image to column transfer kernel function, which reformats a tensor to a 2D matrix to use GEMM on it. The reformatting of the tensor to a 2D matrix structure takes place within local GPU memory and GPU registers rather than global memory, giving a large speedup.

LibDNN uses its GEMM implementation, and im2col+GEMM fused kernels require an OpenCL kernel which has both the im2col and GEMM code in it. Specific high-performance OpenCL kernels are available for various devices. The Intel spatial kernels are one such breed of OpenCL kernels that are heavily optimized for Intel HD graphics.

Intel spatial kernels support auto tuning which takes care of tuning the parameters and selecting the fastest kernel within LibDNN for a particular hardware.

LibDNN has im2col+GEMM fused and implemented as JIT/RTC code so that the code complexity can be reduced based on known parameters of the convolution (stride, dilation, pad, kernel size, image sizes). im2col requires memory, and this memory is not needed when fusing GEMM+im2col.

3.3 OpenCL backend

Kernels written for GPUs are based on different ISA than x86, but are generally based on C.

In the OpenCL Caffe library, a versatile backend for various compute devices, based on OpenCL and ViennaCL, is available. The backend is called *Greentea* and is part of the *Project Greentea* consisting of frontend, models and modified Caffe library (see Figure 3.1). LibDNN is one of the many OpenCL toolsets that are part of the greentea project.

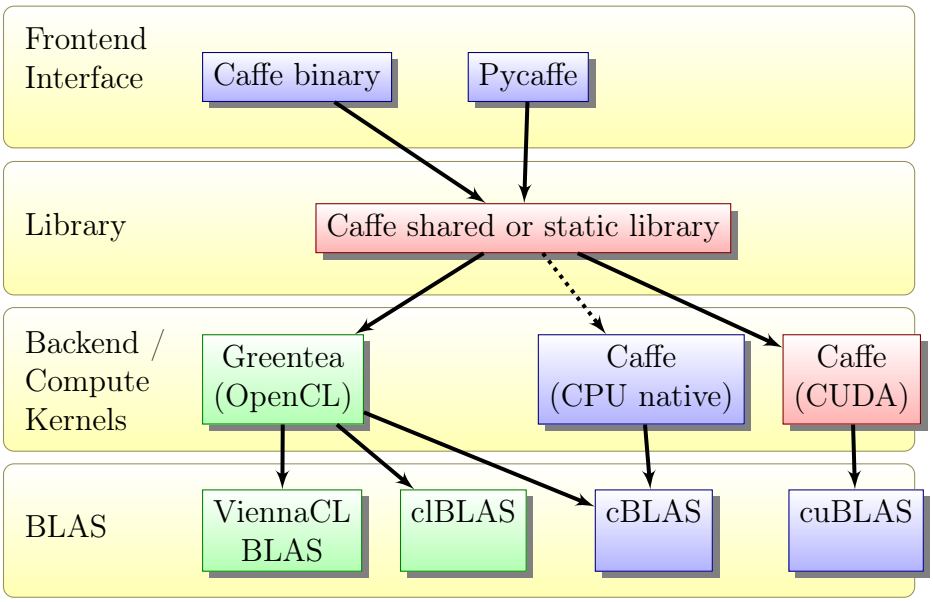

Figure 3.1: *Project Greentea* overview.

Green boxes denote entirely new additions to the Caffe landscape. Red boxes are parts that have been re-implemented or adapted from existing work to fit the needs of this project. Blue parts are mostly unchanged from existing work. Dashed arrows denote deprecated and only partially supported combinations.

Caffe supports CUDA backend. In CUDA Caffe, cuBLAS takes care of the BLAS, cuDNN does convolutions, and the rest is implemented as GPU kernels in CUDA code.

A key feature is that the OpenCL backend in Caffe is **feature equivalent** to the CUDA backend. All GPU layers can be used on both backends. The OpenCL backend is also unit test verified and passes all test cases of the original Caffe library. The tests can be invoked by executing "make runtest" on the source code folder.

It remains possible to compile the Caffe library with support for all backends at once. The compute kernel and BLAS calls can be dispatched dynamically at runtime, depending on what kind of device is selected.

Each of the devices (GPU/CPU) available in the hardware is registered in a new *DeviceContext* object that stores the device and back-end type.

```
$ caffe device_query
```

```
CUDA devices: 0
OpenCL devices: 2

Device id: 0
Device backend: OpenCL
Backend details: Intel(R) Corporation: OpenCL 1.2
Device vendor: Intel(R) Corporation
Name: Intel(R) HD Graphics 4400
Total global memory: 1708759450

Device id: 1
Device backend: OpenCL
Backend details: Intel(R) Corporation: OpenCL 1.2
Device vendor: Intel(R) Corporation
Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
Total global memory: 8513761280
```

After running the above command, the Caffe library enumerates all devices on all enabled backends, starting with CUDA devices and displays them. The selected GPU number determines which *DeviceContext* is set as the default.

3.4 Installation prerequisites

3.4.1 Standard requirements

There are standard requirements for installing/compiling Caffe framework.

- BLAS: Caffe framework requires BLAS for performing backend operations such as matrix and vector computations. BLAS implementations are provided by Intel Math Kernel Library (Intel MKL), ATLAS, openBLAS, ViennaCL and so forth.
- Boost: Caffe framework makes use of the extensive collection of high-quality C++ libraries available from Boost for its math and shared pointer functions.

- leveldb, lmdb, HDF5: They are fast, high-performance storage libraries used by Caffe framework for preparing, storing and managing data.
- protobuf: Caffe framework uses the protobuf library to define the structure and format of data.
- glog, gflags: Caffe framework uses them as essential utilities for logging and for debugging.

3.4.2 OpenCL requirements

LibDNN DNN is integrated into OpenCL Caffe framework, and separate installation is not required. There are two types of **OpenCL modes** when compiling and using the Caffe framework.

- CPU + OpenCL: Faster than traditional Caffe (CPU_ONLY compile) mode but slower than GPU + OpenCL mode. BLAS can still be OpenBLAS and not an OpenCL BLAS.
- GPU + OpenCL: Faster than CPU+OpenCL mode and choice of BLAS does not matter.

The second mode is preferred for real-time performance. The default convolution engine for OpenCL GPUs is LibDNN and is a replacement for cuDNN. There are also new Intel kernels for Intel GPUs available and enabled by default.

Intel spatial kernels included in OpenCL Caffe do not support efficient back propagation and not all shapes of convolutions, but yes, they are the fastest forward propagation on Intel iGPUs.

ISAAC, clBLAS, and CLBlast are strictly optional and can be compiled if required.

3.5 Device specific considerations

INCOMPLETE (this section will show how to get higher performance with example configurations of the Makefile. It is reserved for future, and depends if FP16 code can make it by July/August because in that case, LibDNN will have it is own BLAS as there is currently no good FP16 BLAS)

3.6 Compilation

An Intel device with HD graphics 515 with up to 384 gigaflops of GPU performance is used to demonstration the compilation process. However, any new device that has Intel HD graphics or discrete AMD graphics can be utilized.

3.6.1 Windows

Installation in Windows is fairly simple and an easy process.

Step 1: General

Visual Studio 2015 x64 community edition or higher is needed along with the latest versions of Intel OpenCL SDK and Intel OpenCL run-time. The graphics driver update is done using the Intel update utility.

Install Anaconda version of Python distribution as well Git for Windows. Make sure the latest 64-bit version of CMake for Windows is installed. Create a folder called "projects." Clone or download the OpenCL Caffe framework from URL https://github.com/bvlc/caffe/tree/opencl.

```
C:\> cd projects
C:\Projects> git clone -b opencl https://github.com/BVLC/caffe
C:\Projects> git clone https://github.com/viennacl/viennacl-dev.git
C:\Projects> cd caffe
```

Set the following variables in build_win.cmd file.

```
C:\projects\caffe\scripts\build_win.cmd.

if NOT DEFINED WITH_NINJA set WITH_NINJA=0
if NOT DEFINED PYTHON_VERSION set PYTHON_VERSION=3
```

```
C:\Projects\caffe> scripts\build_win.cmd
```

3.6.2 Linux

Step 1: General OS

Use the URL http://cdimage.ubuntu.com/daily-live/current/ and install Ubuntu 17.0.4 64-bit (at the time of writing).

```
$ cd /home/oreilly
$ sudo su
$ apt-get update
$ apt-get install git
$ apt-get install xz-utils
```

```
$ updatedb; locate libicui18n
$ apt-get install dkms
$ apt-get install mono-devel
$ apt-get install lsb-core

$ add-apt-repository ppa:webupd8team/java
$ apt-get update; apt-get install oracle-java8-installer
$ apt-get install oracle-java8-set-default
```

Step 2: Intel OpenCL SDK and runtime

Download OpenCL SDK from the URL https://software.intel.com/en-us/intel-opencl/download and run the following commands.

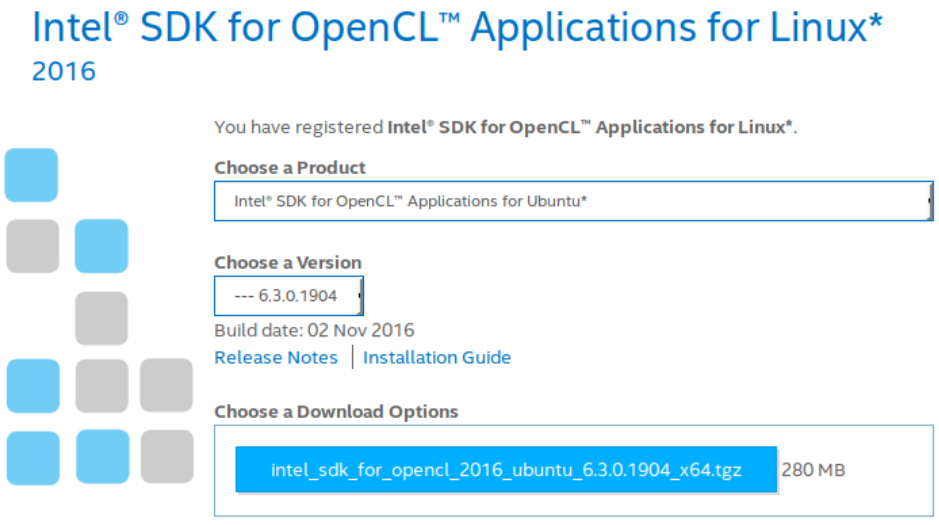

Figure 3.2: opencl.

```
$ tar -xvzf /home/oreilly/intel_sdk_for_opencl_2016_ubuntu_6.3.0.1904_x64.tg
$ cd /home/oreilly/intel_sdk_for_opencl_2016_ubuntu_6.3.0.1904_x64
$ ./install_GUI.sh
```

You can ignore missing prerequisite files libicui18n.so.52()(64bit), and libicuuc.so.52()(64bit) as newer versions are already present.

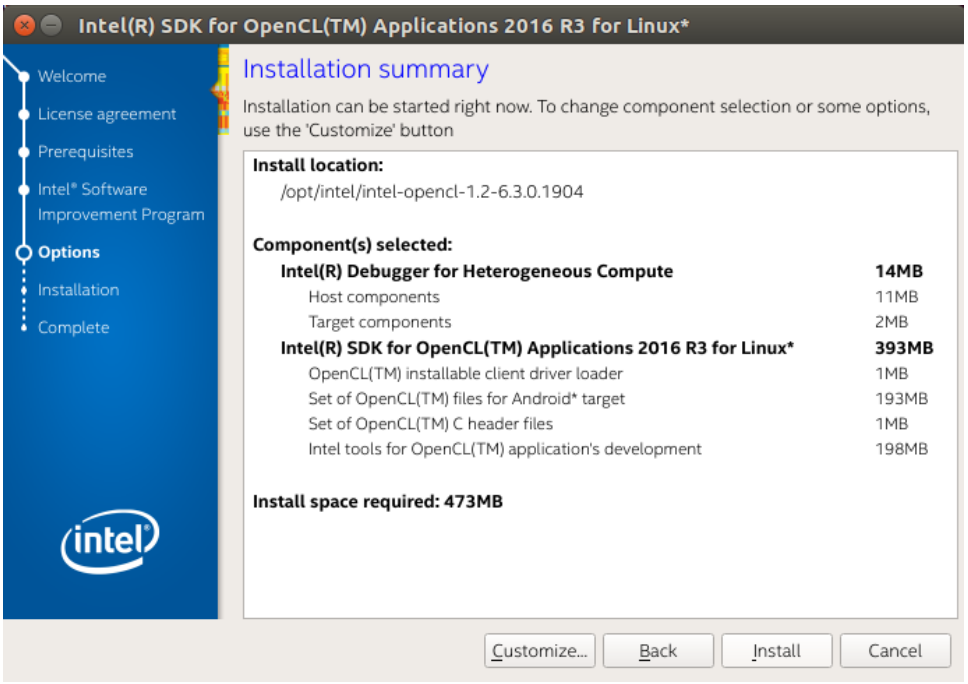

Figure 3.3: install.

Download OpenCL runtime from URL https://software.intel.com/en-us/articles/opencl-drivers and run the following commands.

```
$ unzip /home/oreilly/SRB4.1_linux64.zip
$ cd /home/oreilly/
$ tar -xJf intel-opencl-cpu-r4.1-61547.x86_64.tar.xz
$ tar -xJf intel-opencl-devel-r4.1-61547.x86_64.tar.xz
$ tar -xJf intel-opencl-r4.1-61547.x86_64.tar.xz
```

The extracted opt, etc, directories need to be copied to the root of the file system, and clinfo utility is required to query and display OpenCL information.

```
$ cp -avr opt/ /
$ cp -avr etc/ /

$ apt-get install cmake-qt-gui
$ apt-get install clinfo vim
```

Step 3: Clone ViennaCL, OpenCL Caffe

Clone the OpenCL branch of Caffe framework, and clone or download the latest ViennaCL from the URL http://viennacl.sourforge.net/viennacl-download.html and extract.

```
$ cd /home/oreilly
$ git clone https://github.com/viennacl/viennacl-dev.git
$ cd /home/oreilly/viennacl-dev
$ mkdir build
```

```
$ cd /home/oreilly/viennacl-dev/build
$ cmake -DBUILD_TESTING=OFF -DBUILD_EXAMPLES=OFF
-DCMAKE_INSTALL_PREFIX=$HOME/local
-DOPENCL_LIBRARY=/opt/intel/opencl/libOpenCL.so ..

$ make -j8
$ make install
$ git clone -b opencl https://github.com/BVLC/caffe
```

Step 5: Install dependencies

```
$ apt-get install libprotobuf-dev
$ apt-get install libleveldb-dev

$ apt-get install libsnappy-dev
$ apt-get install libhdf5-serial-dev
```

```
$ apt-get install protobuf-compiler
$ apt-get install --no-install-recommends libboost-all-dev

$ apt-get install libgflags-dev
$ apt-get install libgoogle-glog-dev

$ apt-get install liblmdb-dev
$ apt-get install libopenblas-dev
```

```
$ apt-get install -y cython
$ apt-get install python3-dev python3-skimage
$ apt-get install python3-pip
$ pip3 install pyzmq jsonschema pillow numpy scipy ipython jupyter pyyaml
```

```
$ apt-get install build-essential
$ apt-get install cmake git libgtk2.0-dev pkg-config
$ apt-get install libavcodec-dev libavformat-dev libswscale-dev
$ apt-get install libtbb2 libtbb-dev libjpeg-dev
$ apt-get install libpng-dev libtiff5-dev libdc1394-22-dev
$ apt-get install libjasper-dev
$ apt-get install doxygen
```

Step 6: Compile OpenCV 3

Compile and install OpenCV 3.

```
$ cd /home/oreilly
$ git clone https://github.com/opencv/opencv.git
$ git clone https://github.com/opencv/opencv_contrib.git
$ cd /home/oreilly/opencv
$ mkdir build
$ cmake-gui
```

The source code and build folders must be filled as shown in figure 3.4 and upon pressing the configure button, a pop-up window enables the selection of Unix Makefiles as the generator for the project.

Post configure, the OpenCL and OpenCV paths must be set, and the generate button generates the makefiles. The make command compiles the OpenCV package.

```
$ cd /home/oreilly/opencv/build
$ make -j8
$ make install
$ ldconfig
```

Make install command will copy the compiled libraries to system directories such as:

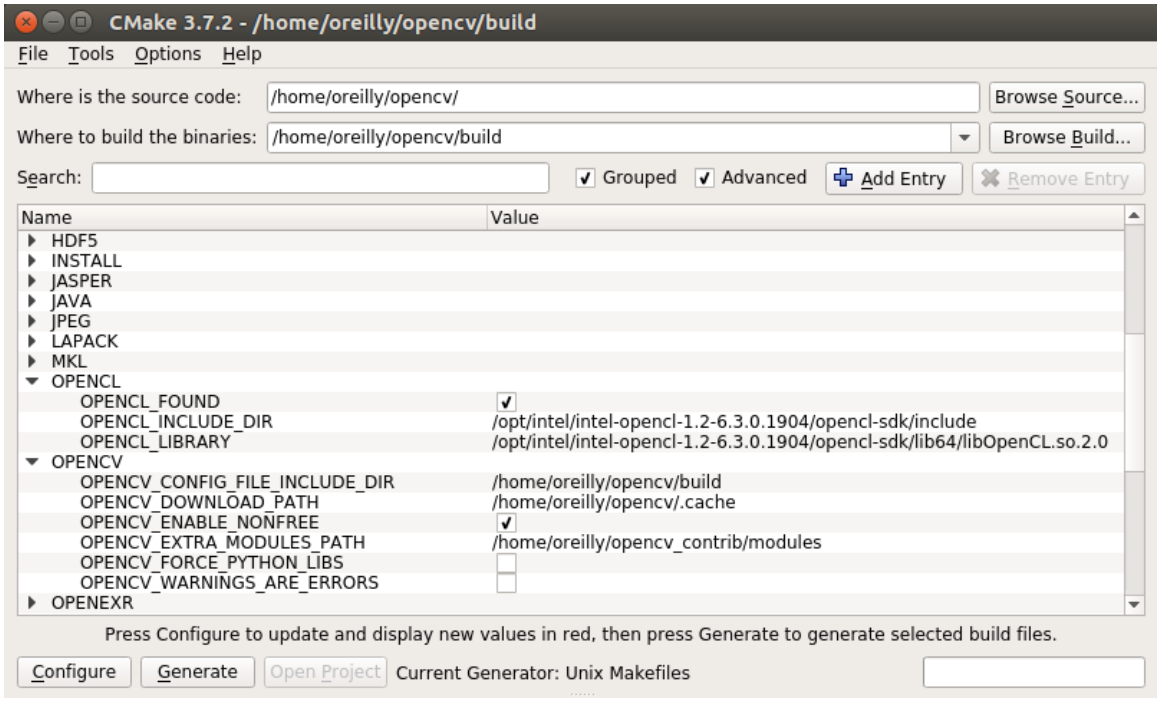

Figure 3.4: CMake Gui

```
/usr/local/include/
/usr/local/lib/
/usr/local/share/
/usr/local/lib/python3.5/dist-packages/cv2.cpython-35m-x86_64-linux-gnu.so
```

OpenCL Caffe framework can now be compiled.

```
$ cd /home/oreilly/caffe
$ cp Makefile.config.example to Makefile.config
```

In addition to existing items, modify the makefile to uncomment and include as follows:

```
USE_GREENTEA := 1
DISABLE_DEVICE_HOST_UNIFIED_MEMORY := 0
USE_LIBDNN := 1
USE_INTEL_SPATIAL := 1
VIENNACL_DIR = ../viennacl-dev
OPENCV_VERSION := 3
BLAS := open
PYTHON_LIBRARIES := boost_python3 python3.5m
PYTHON_INCLUDE := /usr/include/python3.5m \
          /usr/lib/pthong3.5/dist-packages/numpy/core/include
WITH_PYTHON_LAYER := 1
INCLUDE_DIRS := $(PYTHON_INCLUDE) /usr/local/include \
/usr/include/hdf5/serial
```

Determine the correction version present for boost python and libhdf5 to symlink taking into account their respective versions.

```
$ ln -s /usr/lib/x86_64-linux-gnu/libboost_python-py35.so
/usr/lib/x86_64-linux-gnu/libboost_python3.so
$ ln -s /usr/lib/x86_64-linux-gnu/libhdf5_serial.so.100
/usr/lib/x86_64-linux-gnu/libhdf5.so
$ ln -s /usr/lib/x86_64-linux-gnu/libhdf5_serial_hl.so.100
/usr/lib/x86_64-linux-gnu/libhdf5_hl.so
```

```
$ make all -j8
$ make pycaffe -j8
$ make runtest -j8
$ make install
```

The Caffe module for Python _caffe.so should now be available in the directory /home/oreilly/caffe/python/caffe.

The entire caffe folder is itself the module. It can just be added to the python path or added to the python script itself.

```
$ echo $PYTHONPATH=/home/oreilly/caffe/python:$PYTHONPATH
```

3.7 Testing

```
$ /home/orelly/caffe/tools/caffe device_query -gpu all
```