rewriting-caffe recode

- rewriting-caffe recode
  - 1. 编写CMakeLists.txt
  - 2. 文件功能说明
  - 3. caffe::string 的来源
  - 4. lint是C/C++ 强大的测试工具
  - 5. CPU_ONLY 是怎么从cmake传给C++内部的
  - 6. device_alternate.hpp
  - 7. common.hpp 解析
    - 7.1 boost::thread_specific_ptr是线程局部存储机制
  - 8. rng.hpp中的iterator_trait
    - 8.1 value type
    - 8.2 difference_type
    - 8.3 iterator_category 迭代器的类别
  - 9.src/caffe/test/CMakeLists.txt中CACHE STRING
    - 9.1 定义
    - 9.2 两种变量的作用域原理和使用
      - 9.2.1 Normal Variables
      - 9.2.2 Cache Variables
  - 10. CMake 的function
  - 10. test_caffe_main.hpp 中的typedef typename
  - 11. 整个工程是怎么执行的呢？
  - 12. 使用cudaMallocHost的技巧
    - 12.1 cudaMallocHost, melloc, cudaMalloc
  - 13. SyncedMemory的设计很巧妙
  - 14. 在GPU模式下，make runtest caffe_gpu_memcpy无法被找到定义undefine reference
  - 15.Vscode 出现#include errors detected. Please update your includePath. IntelliSense
- 错误记录
  - 1. proto编译错误
- 工程实现顺序

按照一步一步的来实现caffe工程

1. 编写CMakeLists.txt

第一步编写CMakeLists.txt文档，有两个目的：

- 为了将整个工程结构有个大概了解，
- 在编写每个模块时，能跑通整个工程

2. 文件功能说明

```
#include "caffe/proto/caffe.pb.h"
```

包含caffe空间定义
如果不添加该文件的include，则会产生caffe没定义; 因为它是第一个生成的.h文件，且含有namespace caffe {}的caffe命名空间定义

3. caffe::string 的来源

是有namespace caffe{
std:string
}
产生的; 这个申明在common.hpp中，
因此开始先写common.hpp

4. lint是C/C++ 强大的测试工具

lint检查C程序中潜在的错误，包括（但不限于）可疑的类型组合、未使用的变量、不可达的代码以及不可移植的代码。lint会产生一系列程序员有必要从头到尾仔细阅读的诊断信息。使用lint的好处是：1.它可以检查出被编译器漏掉的错误; 2.可以关联很多文件进行错误的检查和代码分析,具有较强大灵活性.lint可以检查的错误类型大体如下:

- 可能的空指针
- 在释放内存后使用了指向该内存的指针
- 赋值次序问题
- 拼写错误
- 被0除
- 失败的case语句(遗漏了break语句)
- 不可移植的代码(依赖了特定的机器实现)
- 宏参数没有使用圆括号
- 符号的丢失
- 异常的表达式

- 变量没有初始化
- 可疑的判断语句(例如,if(x=0))
- printf/scanf的格式检查

因此，代码中注释带有//NOLINT时， 表示That's a comment. In this case, it's a comment designed to be read by a static analysis tool to tell it to shut up about this line. 让静态分析工具不要在意这一行.

5. CPU_ONLY 是怎么从cmake传给C++内部的

在CaffeConfig.cmake.in中传入的, **具体怎么转化为#define CPU_ONLY 的，需要考察**
cmake/Templates/CaffeConfig.cmake.in:53:set(Caffe_CPU_ONLY @CPU_ONLY@)

ConfigGen.cmake 产生CaffeTargets.cmake, 这个文件中有：

```
set_target_properties(caffe PROPERTIES
  INTERFACE_COMPILE_DEFINITIONS "USE_LMDB;USE_LEVELDB;CPU_ONLY;USE_OPENCV"
  INTERFACE_INCLUDE_DIRECTORIES "/usr/include;/usr/local/include;/usr/local/include;/usr/include;/home/
)
```

CPU_ONLY 在这传进去的
INTERFACE_COMPILE_DEFINITIONS： 官方说明：

List of public compile definitions requirements for a library.

Targets may populate this property to publish the compile definitions required to compile against the headers for the target. The target_compile_definitions() command populates this property with values given to the PUBLIC and INTERFACE keywords. Projects may also get and set the property directly.

When target dependencies are specified using target_link_libraries(), CMake will read this property from all target dependencies to determine the build properties of the consumer.

Contents of INTERFACE_COMPILE_DEFINITIONS may use “generator expressions” with the syntax $<...>. See the [cmake-generator-expressions(7)] manual for available expressions. See the [cmake-buildsystem(7)] -manual for more on defining buildsystem properties.

[cmake-generator-expressions(7)]---: 也就是说这个INTERFACE_COMPILE_DEFINITIONS或者target_compile_definition的设置，会在编译的时候控制一些用宏定义为判断条件的代码是否编译编译，也就是控制条件性的定义
Generator expressions are evaluated during build system generation to produce information specific to each build configuration.

Generator expressions are allowed in the context of many target properties, such as LINK_LIBRARIES, INCLUDE_DIRECTORIES, COMPILE_DEFINITIONS and others. They may also be used when using commands to populate those properties, such as target_link_libraries(), target_include_directories(), target_compile_definitions() and others.

They enable conditional linking, conditional definitions used when compiling, conditional include directories, and more. The conditions may be based on the build configuration, target properties, platform information or any other queryable information.

Generator expressions have the form $<...>. To avoid confusion, this page deviates from most of the CMake documentation in that it omits angular brackets <...> around placeholders like condition, string, target, among others.

Generator expressions can be nested, as shown in most of the examples below.

6. device_alternate.hpp

配置 cuda的错误检查宏，以及配置block数目和threads数目

7. common.hpp 解析

这里有点绕，特别是Caffe类里面有个RNG，RNG这个类里面还有个Generator类
在RNG里面会用到Caffe里面的Get()函数来获取一个新的Caffe类的实例（如果不存在的话）。
然后RNG里面用到了Generator。Generator是实际产生随机数的。
（1）Generator类
该类有两个构造函数：

```
Generator()//用系统的熵池或者时间来初始化随机数
explicit Generator(unsigned int seed)// 用给定的种子初始化
```

（2）RNG类
RNG类内部有generator_，generator_是Generator类的实例
该类有三个构造函数：

```
RNG(); //利用系统的熵池或者时间来初始化RNG内部的generator_
explicit RNG(unsigned int seed); // 利用给定的seed来初始化内部的generator_
explicit RNG(const RNG&);// 用其他的RNG内部的generator_设置为当前的generator_
```

（3）Caffe类
1)含有一个Get函数，该函数利用Boost的局部线程存储功能实现

```
// Make sure each thread can have different values.
// boost::thread_specific_ptr是线程局部存储机制
// 一开始的值是NULL
static boost::thread_specific_ptr<Caffe> thread_instance_;

Caffe& Caffe::Get() {
  if (!thread_instance_.get()) {// 如果当前线程没有caffe实例
    thread_instance_.reset(new Caffe());// 则新建一个caffe的实例并返回
  }
  return *(thread_instance_.get());
```

2)此外该类还有

```
SetDevice
DeviceQuery
mode
set_mode
set_random_seed
solver_count
set_solver_count
root_solver
set_root_solver
```

等成员函数
3)内部还有一些比较技巧性的东西比如：

```
// CUDA: various checks for different function calls.
//  防止重定义cudaError_t，这个以前在linux代码里面看过
// 实际上就是利用变量的局部声明
#define CUDA_CHECK(condition) \
  /* Code block avoids redefinition of cudaError_t error */ \
  do { \
    cudaError_t error = condition; \
    CHECK_EQ(error, cudaSuccess) << " " << cudaGetErrorString(error); \
  } while (0)
```

7.1 boost::thread_specific_ptr是线程局部存储机制

大多数函数都不是可重入的。这也就是说在某一个线程已经调用了一个函数时，如果你再调用同一个函数，那么这样是不安全的。一个不可重入的函数通过连续的调用来保存静态变量或者是返回一个指向静态数据的指针。 举例来说，std::strtok就是不可重入的，因为它使用静态变量来保存要被分割成符号的字符串。

有两种方法可以让不可重用的函数变成可重用的函数。第一种方法就是改变接口，用指针或引用代替原先使用静态数据的地方。比方说，POSIX定义了strok_r，std::strtok中的一个可重入的变量，它用一个额外的char**参数来代替静态数据。这种方法很简单，而且提供了可能的最佳效果。但是这样必须改变公共接口，也就意味着必须改代码。另一种方法不用改变公有接口，而是用本地存储线程（thread local storage）来代替静态数据（有时也被成为特殊线程存储，thread-specific storage）。

Boost线程库提供了智能指针boost::thread_specific_ptr来访问本地存储线程。每一个线程第一次使用这个智能指针的实例时，它的初值是NULL，所以必须要先检查这个它的只是否为空，并且为它赋值。Boost线程库保证本地存储线程中保存的数据会在线程结束后被清除。

List5是一个使用boost::thread_specific_ptr的简单例子。其中创建了两个线程来初始化本地存储线程，并有10次循环，每一次都会增加智能指针指向的值，并将其输出到std::cout上（由于std::cout是一个共享资源，所以通过互斥体进行同步）。main线程等待这两个线程结束后就退出。从这个例子输出可以明白的看出每个线程都处理属于自己的数据实例，尽管它们都是使用同一个boost::thread_specific_ptr。

----------另一个解释

对于线程局部存储的概念，正如字面意思，每个变量在每个线程中都有一份独立的拷贝。通过使用线程局部存储技术，可以避免线程间的同步问题，并且不同的线程可以使用不同的日志设置。通过 Boost 库的智能指针 boost::thread_specific_ptr 来存取线程局部存储，每个线程在第一次试图获取这个智能指针的实例时需要对它进行初始化，并且线程局部存储的数据在线程退出时由 Boost 库来释放。

使用线程局部变量的多线程日志的优势：

使用 static 的线程局部变量很容易能实现线程级别的单例日志系统；
通过智能指针 boost::thread_specific_ptr 来存取线程局部存储的生成和清除工作简单方便；
使用线程局部变量很容易实现对一个已有的单例日志系统进行多线程支持的改造，并且不用改动任何原来的日志接口；

8. rng.hpp中的iterator_trait

这个类的作用是得到某个迭代器的相关信息。比如 iterator_traits<vector::iterator>::difference_type。

这个API用于返回模板的具体相关信息，包括迭代器每一项之间的距离difference type， 迭代器每一项的值。iterator_traits负责萃取迭代器的特性。

1、value type 用来表示迭代器所指对象的型别；
2、difference type 用来表示两个迭代器之间的距离；
3、reference 为引用类型；
4、pointer 为指针类型；
5、iterator_category 表明迭代器的类型；

8.1 value type

用来表示迭代器所指对象的型别；

8.2 difference_type

用来表示两个迭代器之间的距离；头尾之间的距离为容器的最大容量。
difference_type是一种常用的迭代器型别，用来表示两个迭代器之间的距离，因此它也可以用来表示一个容器的最大容量，因为对于连续空间的容器而言，头尾之间的距离就是其最大容量。

8.3 iterator_category 迭代器的类别

返回迭代器类别，
迭代器被分为五类：
1、Input Iterator：这种迭代器所指对象，不允许外界改变，只读（read only）；
2、Output Iterator：唯写（write only）；
3、Forward Iterator：允许「写入型」算法（例如 replace()）在此种迭代器所形成的区间上做读写动作；
4、Bidirectional Iterator：可双向移动。某些算法需要逆向走访某个迭代器区间（例如逆向拷贝某范围内的元素），就可以使用 Bidirectional Iterators；
5、Random Access Iterator：前四种迭代器都只供应一部份指标算术能力（前3种支持 operator++ ，第4种再加上 operator--），第5种则涵盖所有指标算术能力，包括 p+n, p-n, p[n], p1-p2, p1<p2.

```
input_iterator_tag
output_iterator_tag
forward_iterator_tag
bidirectional_iterator_tag
random_access_iterator_tag
```

9.src/caffe/test/CMakeLists.txt中CACHE STRING

CMake 变量包含 Normal Variables、Cache Variables。

9.1 定义

- **Normal Variables**： 通过 set(... [PARENT_SCOPE])这个命令来设置的变量就是 Normal Variables。例如 set(MY_VAL "666") ，此时 MY_VAL 变量的值就是 666。
- **Cache Variables**： 通过 set(... CACHE [FORCE])这个命令来设置的变量就是 Cache Variables。例如 set(MY_CACHE_VAL "666" CACHE STRING INTERNAL)，此时 MY_CACHE_VAL 就是一个 CACHE 变量。

9.2 两种变量的作用域原理和使用

9.2.1 Normal Variables

作用域属于整个 CMakeLists.txt 文件，当该文件包含了 add_subdirectory()、include()、macro()、function()语句时，会出现两种不同的效果:

- 包含 add_subdirectory()、function()。（本质是值拷贝）:假设，我们在工程根目录 CMakeLists.txt 文件中使用 add_subdirectory(src) 包含另一个 src 目录，在 src 目录中有另一个 CMakeLists.txt 文件。以根目录 CMake 文件为父目录，src 目录为子目录，此时子目录 CMake 文件会拷贝一份父目录文件的 Normal 变量。需要说明的是，我们在子目录中如果想要修改父目录 CMake 文件包含的 Normal 变量。必须通过 set(... PARENT_SCOPE) 的方式, 再src目录下需要写set(MY_VAL "777" PARENT_SCOPE) # 修改处。
- 包含 include()、macro() （本质有点类似 c 中的 #include 预处理含义），会直接更改

9.2.2 Cache Variables

相当于一个全局变量，我们在同一个 cmake 工程中都可以使用。Cache 变量有以下几点说明

- Cache 变量 CMAKE_INSTALL_PREFIX 默认值是 /usr/local (可以在生成的 CMakeCache.txt 文件中查看)，这时候如果我们 在某个 CMakeLists.txt 中，仍然使用 set(CMAKE_INSTALL_PREFIX "/usr")，那么此时我们 install 的时候，CMake 以后面的 /usr 作为 CMAKE_INSTALL_PREFIX 的值，这是因为 CMake 规定，有一个与 Cache 变量同名的 Normal 变量出现时，后面使用这个变量的值都是以 Normal 为准，如果没有同名的 Normal 变量，CMake 才会自动使用 Cache 变量。
- 所有的 Cache 变量都会出现在 CMakeCache.txt 文件中。这个文件是我们键入 cmake .命令后自动出现的文件。打开这个文件发现，CMake 本身会有一些默认的全局 Cache 变量。例如：CMAKE_INSTALL_PREFIX、CMAKE_BUILD_TYPE、CMAKE_CXX_FLAGSS 等等。可以自行查看。当然，我们自己定义的 Cache 变量也会出现在这个文件中。Cache 变量定义格式为 set(CACHE STRING INTERNAL)。这里的 STRING可以替换为 BOOL FILEPATH PATH ，但是要根据前面 value 类型来确定.

- 修改 Cache 变量。可以通过 set(CACHE INSTERNAL FORCE)，另一种方式是直接在终端中使用 cmake -D var=value ..来设定默认存在的CMake Cache 变量。

10. CMake 的function

```
# Filter out all files that are not included in selected list
# Usage:
#   caffe_leave_only_selected_tests(<filelist_variable> <selected_list>)
function(caffe_leave_only_selected_tests file_list)
  # ARGN 表示<selected_list>的传入, ARGN是后面参数的一个占位符,
  # 如果用caffe_leave_only_selected_tests(var1, var2), 那么ARGN就是var2
  if(NOT ARGN)
    return() # blank list means leave all
  endif()
  # message(STATUS "caffe_leave_only_selected_tests - ARGN: ${ARGN}")
  string(REPLACE "," ";" __selected ${ARGN})
  list(APPEND __selected caffe_main)

  # message(STATUS "caffe_leave_only_selected_tests, file_list: ${${file_list}}")
  set(result "")
  foreach(f ${${file_list}})
    get_filename_component(name ${f} NAME_WE)
    string(REGEX REPLACE "^test_" "" name ${name})
    list(FIND __selected ${name} __index)
    if(NOT __index EQUAL -1)
      list(APPEND result ${f})
    endif()
  endforeach()
  set(${file_list} ${result} PARENT_SCOPE)
endfunction()
```

10. test_caffe_main.hpp 中的typedef typename

```
typedef typename TypeParam::Dtype Dtype;
```

如果给模板类型添加别名时，需要加上typename

如果不加这个关键字，编译器就不知道TypeParam::Dtype到底是个什么东西？可能是静态成员变量，也有可能是静态成员函数，也有可能是内部类。
加上这个关键字等于手动告诉编译器：TypeParam::Dtype就是一个类型。

也就是说 模板类型在实例化之前，编译器并不知道
typedef创建了存在类型的别名，而typename告诉编译器TypeParam::Dtype是一个类型而不是一个成员。

11. 整个工程是怎么执行的呢？

- 先给caffe 目标，所需要的所有源代码，srcs，然后这样在生成该target时，会将所有的srcs中的.cpp, .cu, 进行编译，这些文件中include的头文件会被找到，然后作为这些源文件的依赖进行编译，这个会自动生成makefile语法中cxxxx.o:xx.hpp,xxx.h的格式
- target_link_libraries给出所有需要的库
- 给出所有需要的 .hpp, .h 文件
- target_compile_definitions 给出一些定义这个是用来控制程序中像'#ifdef CPU_ONLY'这样的宏判断语句的；如下面配置，${Caffe_DEFINITIONS}包含-DCPU_ONLY，-DUSE_MKL等， 这样在编译程序的时候，或根据程序中#ifndef CPU_ONLY 来控制编译哪些代码？

```
# Caffe_DEFINITIONS变量设置
set(Caffe_DEFINITIONS "")
cmake/Dependencies.cmake:63:  list(APPEND Caffe_DEFINITIONS PUBLIC -DUSE_LMDB)
cmake/Dependencies.cmake:65:    list(APPEND Caffe_DEFINITIONS PRIVATE -DALLOW_LMDB_NOLOC
cmake/Dependencies.cmake:74:  list(APPEND Caffe_DEFINITIONS PUBLIC -DUSE_LEVELDB)
cmake/Dependencies.cmake:93:  list(APPEND Caffe_DEFINITIONS PUBLIC -DCPU_ONLY)
cmake/Dependencies.cmake:112:  list(APPEND Caffe_DEFINITIONS PUBLIC -DUSE_OPENCV)
cmake/Dependencies.cmake:132:    list(APPEND Caffe_DEFINITIONS PUBLIC -DUSE_MKL)
cmake/Dependencies.cmake:141:      list(APPEND Caffe_DEFINITIONS PUBLIC -DUSE_ACCELERATE
cmake/Dependencies.cmake:185:      list(APPEND Caffe_DEFINITIONS PRIVATE -DWITH_PYTHON_L
cmake/ConfigGen.cmake:36:    list(APPEND Caffe_DEFINITIONS -DUSE_HDF5)
cmake/Cuda.cmake:257:    list(APPEND Caffe_DEFINITIONS PUBLIC -DUSE_CUDNN)
```

- set_target_properties() 设置一些属性

```
# 主要的工程编译程序
# 因为一些库和anaconda3/libs冲突了，它选的是anaconda3里面的库
add_library(caffe ${srcs})
caffe_default_properties(caffe)
message(STATUS "add lib caffe Caffe_LINKER_LIBS: ${Caffe_LINKER_LIBS} ")
target_link_libraries(caffe ${Caffe_LINKER_LIBS})
target_include_directories(caffe ${Caffe_INCLUDE_DIRS}
                                 PUBLIC
                                 $<BUILD_INTERFACE:${Caffe_INCLUDE_DIR}>
                                 $<INSTALL_INTERFACE:include>)
message(STATUS "caffe definition: -------- ${Caffe_DEFINITIONS}, ${Caffe_COMPILE_OPTIONS
target_compile_definitions(caffe ${Caffe_DEFINITIONS})
if(Caffe_COMPILE_OPTIONS)
  message(STATUS "add lib caffe Caffe_COMPILE_OPTIONS:  ${Caffe_COMPILE_OPTIONS}")
  target_compile_options(caffe ${Caffe_COMPILE_OPTIONS})
endif()
set_target_properties(caffe PROPERTIES
    VERSION   ${CAFFE_TARGET_VERSION}
    SOVERSION ${CAFFE_TARGET_SOVERSION}
    )
```

12. 使用cudaMallocHost的技巧

syncedMemory 中有个技巧：CaffeMallocHost 中判断ptr指针是否为GPU指针，如果是则用cudaMallocHost生成pinned memory这样高效；如果不是则malloc进行分配。 这样能高效执行GPU->CPU的传输。 而且pinned mem的设置主要对Memcpy函数的高效传输有用。

12.1 cudaMallocHost, melloc, cudaMalloc

分页（英语：Paging），是一种操作系统里存储器管理的一种技术，可以使电脑的主存可以使用存储在辅助存储器中的数据。操作系统会将辅助存储器（通常是磁盘）中的数据分区成固定大小的区块，称为“页”（pages）。当不需要时，将分页由主存（通常是内存）移到辅助存储器；当需要时，再将数据取回，加载主存中。相对于分段，分页允许存储器存储于不连续的区块以维持文件系统的整齐。分页是磁盘和内存间传输数据块的最小单位.

我们用cudaMalloc()为GPU分配内存,用malloc()为CPU分配内存.除此之外,CUDA还提供了自己独有的机制来分配host内存:cudaHostAlloc(). 这个函数和malloc的区别是什么呢?
malloc()分配的标准的,可分页的主机内存(上面有解释到),而cudaHostAlloc()分配的是页锁定的主机内存,也称作固定内存pinned memory,或者不可分页内存,它的一个重要特点是操作系统将不会对这块内存分页并交换到磁盘上,从而保证了内存始终驻留在物理内存中.也正因为如此,操作系统能够安全地使某个应用程序访问该内存的物理地址,因为这块内存将不会被破坏或者重新定位.

由于GPU知道内存的物理地址,因此就可以使用DMA技术来在GPU和CPU之间复制数据.当使用可分页的内存进行复制时(使用malloc),CUDA驱动程序仍会通过dram把数据传给GPU,这时复制操作会执行两遍,第一遍从可分页内存复制一块到临时的页锁定内存,第二遍是再从这个页锁定内存复制到GPU上.当从可分页内存中执行复制时,复制速度将受限制于PCIE总线的传输速度和系统前段速度相对较低的一方.在某些系统中,这些总线在带宽上有着巨大的差异,因此当在GPU和主机之间复制数据时,这种差异会使页锁定主机内存比标准可分页的性能要高大约2倍.即使PCIE的速度于前端总线的速度相等,由于可分页内训需要更多一次的CPU参与复制操作,也会带来额外的开销.

当我们在调用cudaMemcpy(dest, src, ...)时，程序会自动检测dest或者src是否为Pinned Memory，若不是，则会自动将其内容拷入一不可见的Pinned Memory中，然后再进行传输。可以手动指定Pinned Memory，对应的API为：cudaHostAlloc(address, size, option)分配地址，cudaFreeHost(pointer)释放地址。注意，所谓的Pinned Memory都是在Host端的，而不是Device端。

有的人看到这里,在写代码的过程中把所有的malloc都替换成cudaHostAlloc()这样也是不对的.

固定内存是一把双刃剑.当时使用固定内存时,虚拟内存的功能就会失去,尤其是,在应用程序中使用每个页锁定内存时都需要分配物理内存,而且这些内存不能交换到磁盘上.这将会导致系统内存会很快的被耗尽,因此应用程序在物理内存较少的机器上会运行失败,不仅如此,还会影响系统上其他应用程序的性能.
综上所述,建议针对cudaMemcpy()调用中的源内存或者目标内存,才使用页锁定内存,并且在不在使用他们的时候立即释放,而不是在应用程序关闭的时候才释放.我们使用下面的测试实例

13. SyncedMemory的设计很巧妙

有四个状态， 用to_cpu, to_gpu进行状态之间的转移判断； 利用cpu_data,mutable_cpu_data,gpu_data,mutable_gpu_data进行不同需求的数据读取； 也就是一个任务要分离出需要的接口函数，以及确定好那些是内部的固定的功能函数如to_cpu, to_gpu；

14. 在GPU模式下，make runtest caffe_gpu_memcpy无法被找到定义undefine reference

可能链接库的顺序不对， 而且添加了caffe::GlobalInit(&argc,&argv) make就出该错误;
真正原因： 是.hpp中的声明，和.cu中的定义不一样导致的

```
# declare in math_functions.hpp
void caffe_gpu_memcpy(const size_t n, const void* x, const void* y); // wrong
// 应该将 第二个const去掉
void caffe_gpu_memcpy(const size_t n, const void* x, void* y);

# define in math_functions.cu
void caffe_gpu_memcpy(const size_t n, const void* x, void* y){
        if (x != y){
            // x ---> y
            CUDA_CHECK(cudaMemcpy(y, x, n, cudaMemcpyDefault)); // NOLINT(caffe/alt_fn)
        }
    }
```

15.Vscode 出现#include errors detected. Please update your includePath. IntelliSense

shift+ctrl+p, 然后输入C/C++ 然后找C/C++ configure file (JSON); 然后配置如下：

```
{
    "configurations": [
        {
            "name": "Linux",
            "includePath": [
            # 一些.h 如driver_types.h 无法被加入，但是能跳转过去； 需要添加该文件所在路径，因此添加了一下可能会需要
                "/usr/include/**",
                "/usr/local/cuda/include/**",
                "/usr/local/include/**",
                "${workspaceFolder}/**"
            ],
            "defines": [],
            "compilerPath": "/usr/bin/gcc",
            "cStandard": "c11",
            "cppStandard": "c++11",
            "intelliSenseMode": "gcc-x64"
        }
    ],
    "version": 4
}
```

错误记录

1. proto编译错误

```
[ 14%] Building CXX object src/caffe/CMakeFiles/caffeproto.dir/__/__/include/caffe/proto
c++: fatal error: no input files
compilation terminated.
/bin/sh: 1: -fPIC: not found
/bin/sh: 1: -Wall: not found
src/caffe/CMakeFiles/caffeproto.dir/build.make:74: recipe for target 'src/caffe/CMakeFil
make[2]: *** [src/caffe/CMakeFiles/caffeproto.dir/__/__/include/caffe/proto/caffe.pb.cc.
CMakeFiles/Makefile2:183: recipe for target 'src/caffe/CMakeFiles/caffeproto.dir/all' fa
make[1]: *** [src/caffe/CMakeFiles/caffeproto.dir/all] Error 2
Makefile:127: recipe for target 'all' failed
make: *** [all] Error 2

查看  prtoc --version  2.6.1，  不是版本的问题
```

根据提示可以发现：

-fPIC not found

-Wall not found

查看错误处代码：

```
src/caffe/CMakeFiles/caffeproto.dir/__/__/include/caffe/proto/caffe.pb.cc.o: include/caf
        @$(CMAKE_COMMAND) -E cmake_echo_color --switch=$(COLOR) --green --progress-dir=/
        cd /home/lshm/Projects/rewrite_caffe/build/src/caffe && /usr/bin/c++   $(CXX_DEF

1) 去掉-fPIC, -Wall  编译成功
2）c++ 换成g++ 也不行

通过打印发现，${CXX_FLAGS}的内容是 ";-fPIC;-Wall"，而一般这两个编译选项是不带";"分号的，
因此赋值时出错；  而CXX_FLAGS是make的系统变量，它是由CMAKE_CXX_FLAGS的值生成的. 而根目录下的CMake
  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS}" -fPIC -Wall)
对照原来的caffe的CMakeLists.txt是
  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fPIC -Wall")
所以修改后错误消失
```

工程实现顺序

1. caffe 一半，然后需要common.hpp
2. common.hpp 有需要cudnn， device_alternate,
3. 写common的测试代码test_common.cpp， 而它又需要syncedmem.hpp和math_functions.hpp
4.