CUDA SASS汇编器实现笔记（1）-需求及指导思想

https://zhuanlan.zhihu.com/p/279341172

None Wed Jun, 02 04:59

最近因为有一些实际需求，要做一个功能相对完整的CUDA SASS汇编器。其实去年就做了一点Turing的指令编码，其他部分都是手改。这次希望能把整个汇编器做得功能更完整实用。功能代码部分已完成90%以上，但当前肯定是bug比feature多，所以还在整理测试中。争取这个月底前能有第一个可用的版本开源出来吧！这段时间我也会分享一些实现的心得和理解，毕竟它比较像个黑盒子，猜的东西很多。就着分享的机会，整理一下思路。另外由于是业余时间写的，而且已经很久没这么死磕工程实现的细节了……所以肯定会有很多不到位的地方，也希望大家多交流指正！

首先要说明一下，NV官方文档中有简单的SASS指令的介绍，但是只有Opcode的列表，既没有编码，也没有modifier和operand的介绍，更要命的是也没有任何语义和用法文档。比较迷的是NV官方提供了反汇编的工具nvdisasm和cuobjdump，这也是能够去做汇编的前提。能反汇编却不能汇编，大概是既想让用户有一个针对底层优化的观察口子，又不想维护这个文档和工具吧！另外，SASS与底层架构设计强相关，有些细节也许是NV不想公开。常见的指令集里，x86、arm、power、mips之类，也包括AMD的GPU，指令编码格式都是公开的，很多的汇编器甚至都是开源的。Intel的GPU也有开源的Intel Graphics Compiler，没研究过是底层机器码还是类似PTX的虚拟指令集。SASS作为如此广泛应用的指令集，却如此封闭，确实是有些别致。之前逛NV论坛，时不时可以碰见某位N同学喷NV的闭源策略，也是有趣~

官方的没有，非官方的还是有很多的。比如最早的decuda，后来有asfermi，再到maxas。阿里云据说有内部使用的askepler，只是我没见到过。有些公司可能自己也在做，只是不公开而已。这其中maxas可能是最广为人所知的非官方汇编器，不仅因为它完成度相对非常高，更有较为完整的微架构的解读（主要是control codes，印象中是我见过这个解读最原始的来源），再加上非常细致高效的SGEMM实现做范本，成为很多人学习SASS的重要入门材料。尽管如此，maxas仍然不太能满足我的需求，比如它是用我不熟的perl写的，然后只支持maxwell（pascal大概也行？），指令支持也不太完整，上层feature更是丢失严重。所以我需要自己做一个更完善的汇编器。

动手前当然还是明确一下需求。这个需求的原始驱动力肯定是想写出更极致优化的程序。由于我比较关注的还是上层应用，平时还是高级语言用的多，用Cuda C/C++写的代码占绝大多数，毕竟生产力很重要。偶尔有一些性能非常敏感的kernel会用PTX来写，然后可能会在SASS上做一些微调，但总体流程上不会大改。我极少从零开始写SASS代码。一来是很多SASS的调用惯例手写起来真是非常麻烦，而直接用runtime api生成个对应接口的壳也不是啥难事，调用起来也方便。二来应用里kernel的重要性是有主次的，比如一个模块10个kernel，只有两个kernel性能容易成为瓶颈，那我就只改这两个，其他照抄C代码的编译结果就行。总体来讲，我对官方工具链生

成的代码质量还是比较信任的，大部分优化还是做得非常细致。所以我用SASS的场景主要还是要覆盖一些用C或是PTX会词不达意的场合。因此，从流程上讲，这个汇编器的目的是弥补官方工具中无法精准控制SASS生成的部分，其他能复用官方流程的就尽量复用，这是最正义也是最不损失功能的使用方式。这也意味着所有上层的feature到了这层都不能丢失。关于这个问题其实有点复杂，很多上层功能与SASS指令不一定有直接对应关系，往往需要一些其他代码的配合。另外，CUDA的有些功能感觉是做在driver层的，SASS都看不出来，比如Cooperative group。还有一些比如dynamic parallelism之类就更复杂了，涉及到很多device runtime函数的调用，当然这个我很少用。所以这就要求用户常规的对SASS的修改不能导致上层feature失效。保留上层feature也意味着我可以把大量过程性的代码和流程上的简单优化放到CUDA C或是PTX中来做，这比直接写SASS更简洁高效，适用性也更强。很多公司都会用SASS写一些底层的AI算子实现，一般说来不会涉及太多上层feature，要求其实是比上面的低，这些应该说支持起来没什么难度。

汇编器另一个比较重要的用处就是写一些micro benchmark的case。且不要说SASS这种不开源的指令集，即使是x86、arm之类，都没办法详尽的提供性能状态的方方面面。很多具体的点需要你自己去测试、分析和总结。因为这不仅仅与指令集本身有关，还与实现这个指令集的微架构有关。而指令集作为使用微架构的入口，也是测试分析微架构的主要手段。arXiv上有一些dissecting NV相关微架构的文章，大概就是这些micro benchmark结果的总结。没有汇编器，很多特定的指令序列很难通过上层语言来触发（包括PTX），测试上自然也会很受限制。所以一个功能完整的汇编器会使得micro benchmark更简单直接，从而更能发掘微架构中的各种性能细节。

然后就是易用性和扩展性的需求。比如不同SM版本的SASS指令编码是不一样的，那各个版本最好都要支持（至少是框架上兼容），不然将来新出来一个又要大改。我可不愿意做这种重复劳动。因为kepler已经比较古老了，新的CUDA 11.1甚至SM50都已经deprecated，所以我这里只会支持Maxwell及以后的架构。从我现在的观察来看，除指令编码外，各版本之间的差别并不是特别大，而指令编码部分我这里大部分是自动完成的（control codes部分暂时是交给用户自己做的），因此工作量还可以接受。除此之外，从汇编器的可用性上来讲，比较完善的错误检查还是非常重要的，出错的时候还要尽量给出有价值的修改提示，这部分会尽量有。但是错误提示要想做得很完善，需要做语义分析，所以这里可能只会支持一些简单的，将来看需求再慢慢完善。

最后就是希望能有一些汇编的扩展功能。汇编器本身的功能可以认为是直白的翻译函数，你输入汇编文本，它根据规则生成对应代码，不关心语义，更不会帮你优化。所以汇编里的绝大部分信息都必须显式的提供，比如寄存器的分配，Control codes的指定等等。这些通常都是更上层的编译器的工作，在当前版本还不支持。将来有时间也许可以加入有限的支持，这样写汇编会方便很多。然后还有一些更上层的功能，比如支持预处理变量之类（类似C的宏定义），基于预

处理变量的简单流控制（if-else, for）之类，这个有的时候还是挺有用的，特别是需要生成一系列高度相似kernel的时候（比如做auto-tuning，或者是参数化的kernel）。这个短时间内不会支持，将来再看吧！

关于汇编器的性能，做过底层优化的人大致都有感觉：一个普通的kernel，用Cuda C实现可能一天，调试、测试和优化几天，多数也就差不多了。用PTX则时间多几倍，用SASS估计又比PTX多几倍。一些性能特别敏感的kernel，死磕几个月甚至半年也不是不可能（当然可能不是一个，而是一组）。所以这种任务对汇编器的性能不太敏感，因此我选择用Python来实现，这样实现起来方便，扩展性也相对好一些。有些都只是简单粗暴的实现功能，没怎么做性能上的优化。

从我的角度讲，我对汇编器的用法还是很正义的，主要是对官方工具输出的分析，也是用于相应程序的优化。所有参考资料来源于官方文档，或是网上开源材料，有些甚至就是NV论坛的官泄。所以如果碰上一些问题，有些都可以回溯到官方编译路径的问题，甚至还可以找NV报bug…… 对我来讲，这只是个工具，终极目的还是能更好的做性能优化。汇编器可以打开很多新的可能性，这个可能性比汇编器本身对我更重要，因此我写或是其他人来写都一样，我能用就行。如果有对这个感兴趣的人，有能力和意愿花时间维护的，也欢迎联系我~ 当然做完了还是要开源，这样我也能省点力气，揩一点大家的油…… 如果不想开源，也欢迎讨论交流一下思路~

先大致说一下预期的使用流程：

1. 先用CUDA C/C++写一个.cu的壳，所有kernel的名字及输入参数配置、shared memory配置、contant memory配置，也包括一些模块级变量（比如CU文件里定义的global变量，texture或surface的reference等）都需要与最终kernel一致。可能的话，kernel一些简单流程代码也可以用C实现大致框架，因为这些都会被继承下去。汇编器里更改这些东西非常不方便，隐藏规则很多，我觉得多数时候也不会频繁改这部分东西，所以尽量把这些放到上层，走官方路径。
2. 用NVCC编译cu文件，用keep参数（大概相当于clang的save-temps）得到中间的cubin文件。
3. 用nvdisasm对cubin进行反汇编。尽管是反汇编，但要根据生成的反汇编文本直接回到cubin还是挺难的。至少根据我现在的观察，从section的排布到symbol的定义，里面有非常多隐藏的惯例。我也不想挨个检查，所以我会把cubin中的这些信息也dump出来，从而继承所有原cubin的各种ELF信息。这些绝大部分都是前面那个“壳”决定的，大部分SASS的修改不会影响这个壳，这也是我决定套壳而不是从零开始写的原因。这些ELF信息和cubin的nvdisasm反汇编输出组成一个新的cuasm文件。这就是用户需要修改的汇编文本了。
4. 然后就是最耗时的SASS修改了。所有的汇编文本的调整都是对cuasm进行的修改。
5. 汇编器把修改后的cuasm汇编成cubin文件。
6. 有了cubin文件，按理说我们就可以用driver api去load这个cubin，然后调用其中的kernel代码。但我偷懒能力很强，我不这么弄。因为这不仅需要在原来的runtime api外另写一套代

码，改起来也很啰嗦。所以我这里会用nvcc的dryrun参数（类似clang的-###），得到底层的编译命令序列，然后把原来ptxas编译生成cubin的那一步改掉，相当于把cubin掉包。然后其他命令序列不变。这样我就可以基于原来的runtime api直接生成可执行文件（前提是前面那个壳里的关键配置不能改，否则编译完不匹配）。这样修改和测试会非常的方便，使用起来和直接用CUDA C写的没啥区别，放在大项目里也更方便集成。当然，更tricky的玩法也有，大家可以自己摸索~ 我记得LLVM也有PTX的后端，接上ptxas后跟这个流程上应该也是相似的，只不过我没这么用过。

大致列一下接下来要写的几块内容，计划一两周一更吧！不过呢，懒起来谁也挡不住，争取吧！

1. SASS指令编码的自动化实现，这个主要是通过分析cuobjdump的反汇编得到的，其中涉及到一些数学算法的实现，整体还算简单有效。这个算是完成95%以上了，极个别坑没填。
2. Cubin是个ELF文件，虽说咱对ELF的结构完全没兴趣，分析这些又烦又无趣，但没办法，不搞清楚就没法写。这个应该说已经完成80%左右，不过这里面坑太多，先看看吧！
3. 到这期就基本上可以把源代码完全放出来了。可以介绍一下基本用法，几个简单的Tutorial。不想太早把代码放出来主要是因为当前实现还比较乱，我会对实现做整理和重构，很多用户接口都可能需要调整，不希望造成不必要的困扰。
4. Control codes的自动生成。这个我勉强有点思路，还在研究算法。根据我现在的观察，安培架构的SM80和SM86在指令编码本身与Turing只有个别微调，但是微架构的变化使得control codes还是有显著变化。这个要一代一代的手调也是很烦，我在看有没有什么好方法可以自动做。
5. 然后就可以分享一些micro benchmark的case，不会专门测指令throughput和latency之类，这个略平淡，而且已经有人测过很多了。我主要会关注更细致的任务调度上的点，或者是特定任务模式下的性能之类。
6. 最后就是一个大头了：基于这个汇编器，做一些比较有意思的应用~ 算是计划，做不做再说吧！看心情~

最后再谈点感想。会写汇编是个不错的技能，表示你对底层实现方式有一定的了解。但它仍然只是你做事情的一个工具，不要把会用某个工具作为体现你水平的方式。有些人觉得我现在的CUDA C程序性能不好，那我把它改成用PTX或是SASS写就会快。类似论调以前常常听到，比如认为用python或matlab很慢的程序，用C改写就会快了，这是同一个类型。显然，这些都是片面的。用底层实现能更快，一定是因为你有一些优化方式无法用上层语言来表达。其中有一些确实是上层的表述能力有缺失，但也有很多的时候是你没用好或是没能充分挖掘上层语言的潜力。任何一个广泛应用且久经考验的系统，都一定会充分考虑到常用的需求，除非这不是它的设计需求。所以，既要扩展你的技术能力，也要抑制自己使用复杂工具带来的虚荣心。根据自己的实际需求和技术状态，选择合适的方法和工具。毕竟，你做的事情和产出比用什么工具更重要。