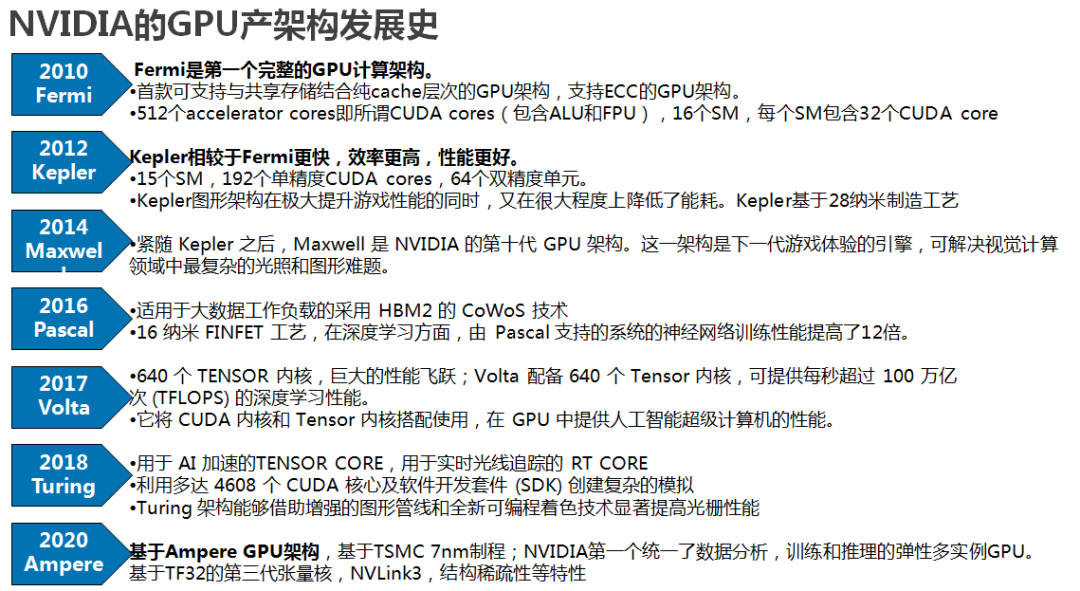
1. **GPU架构**

架构演进: [NVIDIA GPU技术和架构演进](https://www.eet-china.com/mp/a42816.html)



GPU架构的更新体现在SM、TPC的增加，最终体现在GPU浮点计算能力的提升\

* 1. **Volta架构(**[**TeslaV100**](https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf)**)**

**A full GV100 GPU consists of:**

 Six GPCs

Each GPC has:

● Seven TPCs (each including two SMs)

● 14 SMs

 84 Volta SMs

Each SM has:

● 64 FP32 cores

● 64 INT32 cores

● 32 FP64 cores

● 8 Tensor Cores

● Four texture units

 Eight 512-bit memory controllers (4096 bits total)

见下图(\_V100)以及\_V100的Table 1



* + 1. **SM（streaming multiprocessor）**

一个SM负责一个warp线程束的处理,里面有多个CUDACore和TensorCore负责这些线程中的计算. 其中L1 Cache和Share Memory(可编程控制)负责本地缓存以及从Global Mem的加载. Warp Scheduler负责warp及其线程的调度. Register File是线程中访问速度更快的寄存器, L0 Cache是Volta才加的作为寄存器的补充缓存. LD/ST是load/store unit,负责内存操作. SFU是Special function unit，用来做cuda的intrinsic function的，类似于\_\_cos()这种. 一个block通常会分配给一个SM负责。

---\_KPG/p12, \_V100/p17---start

Each SM is partitioned into four processing blocks, called SM sub partitions. The SM sub partitions are the primary processing elements on the SM. Each sub partition contains the following units:

‣ Warp Scheduler (CC7的warp pool为16,CC8的8(\_KPG/p13))

‣ Register File (64K)

‣ L0 Instruction Cache

‣ Dispatch Unit

‣ Execution Units/Pipelines/Cores

‣ 16 Integer Execution units (INT32 Core)

‣ 16 + 8 Floating Point Execution units (16 FP32 Cores, 8 FP64 Cores)

‣ Memory Load/Store units

‣ Special Function unit

‣ 2 mixed-precision Tensor Cores

Shared within an SM across the four SM partitions are:

‣ Unified L1 Data Cache / Shared Memory (共128K而shared可用到96K. L1的bandwidth接近shared的)

‣ 4 Texture units

‣ RT Cores, if available

‣ L1 Instruction Cache

---end

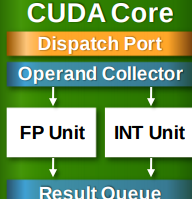
由于FP32和INT32 op有各自的处理单元,所以一些FP和INT的计算指令可并行执行,比如按某个INT类型的下标进行循环做FP计算会比以前更快(\_V100/p23)

The Turing SM inherits the Volta SM's independent thread scheduling model. The SM maintains execution state per thread, including a program counter (PC) and call stack. (\_KPG,所以CC7或以上的thread同步方式是不一样的)

下图:V100的SM架构(\_V100)



* + 1. **CUDA Core (SP)**



也称为SP（streaming processor）是GPU中的基础计算单元,上图的FP64 Cores/ INT Cores/FP32 Cores都是针对不同数据类型的CUDACore, 也就是说可以并行做不同数据类型的运算,当然也可以并行(用TensorCore)做张量运算. 一个CUDACore相当于一个小CPU,可单独执行运算.

* + 1. **Tensor Core**

[请问英伟达GPU的tensor core和cuda core是什么区别？](https://www.zhihu.com/question/451127498)

可以简单的理解为TensorCore是专门用于张量运算的CUDACore,支持更多数据类型; 如果CUDACore内是一个个数运算,TensorCore内就是一个个张量运算.

集成度更高,一个TensorCore内部就集成了多种数据类型混合的张量运算.

要程序控制调用TensorCore一般是针对warp的,见”<CUDA\_C\_Programming\_Guide\_10.2>/B.17. Warp matrix functions”和” [NVIDIA GPU 架构演进 | Chenfan Blog (jcf94.com)](https://jcf94.com/2020/05/24/2020-05-24-nvidia-arch/)”

* + 1. **特定设备相关信息**

比如SM支持的最大blk数目,blk的最大thread数目等等...

见\_CUPG/Table 15 Technical Specifications per Compute Capability

通过cudaGetDeviceProperties()获取(见\_CUPG/3.2.7.1 Device Enumeration)

* 1. **Memory**
     1. **数据流**

当kernel启动时,需要从内存中加载数据到GlobalMem(和显存是同一个东西么???)

一个kernel对应一个grid(含多个block),GlobalMem给所有在跑的grid使用.

GPU会把grid的常用数据加载到L2缓存

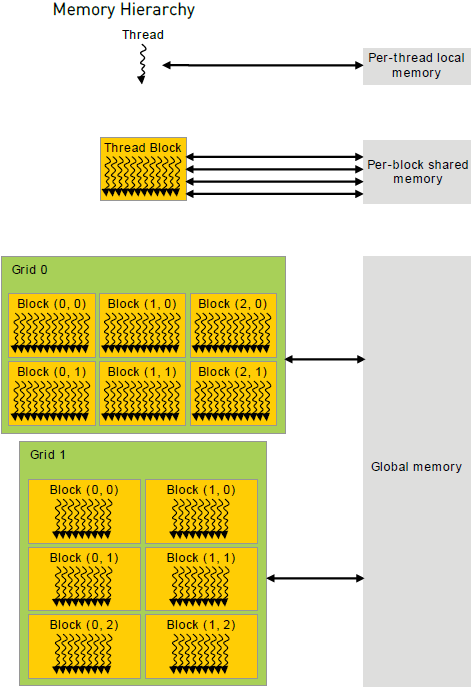
当grid中某block启动时,数据会从GlobalMem和L2加载到thread本地L1缓存和ShareMem. 数据通过LD/ST进行读写.

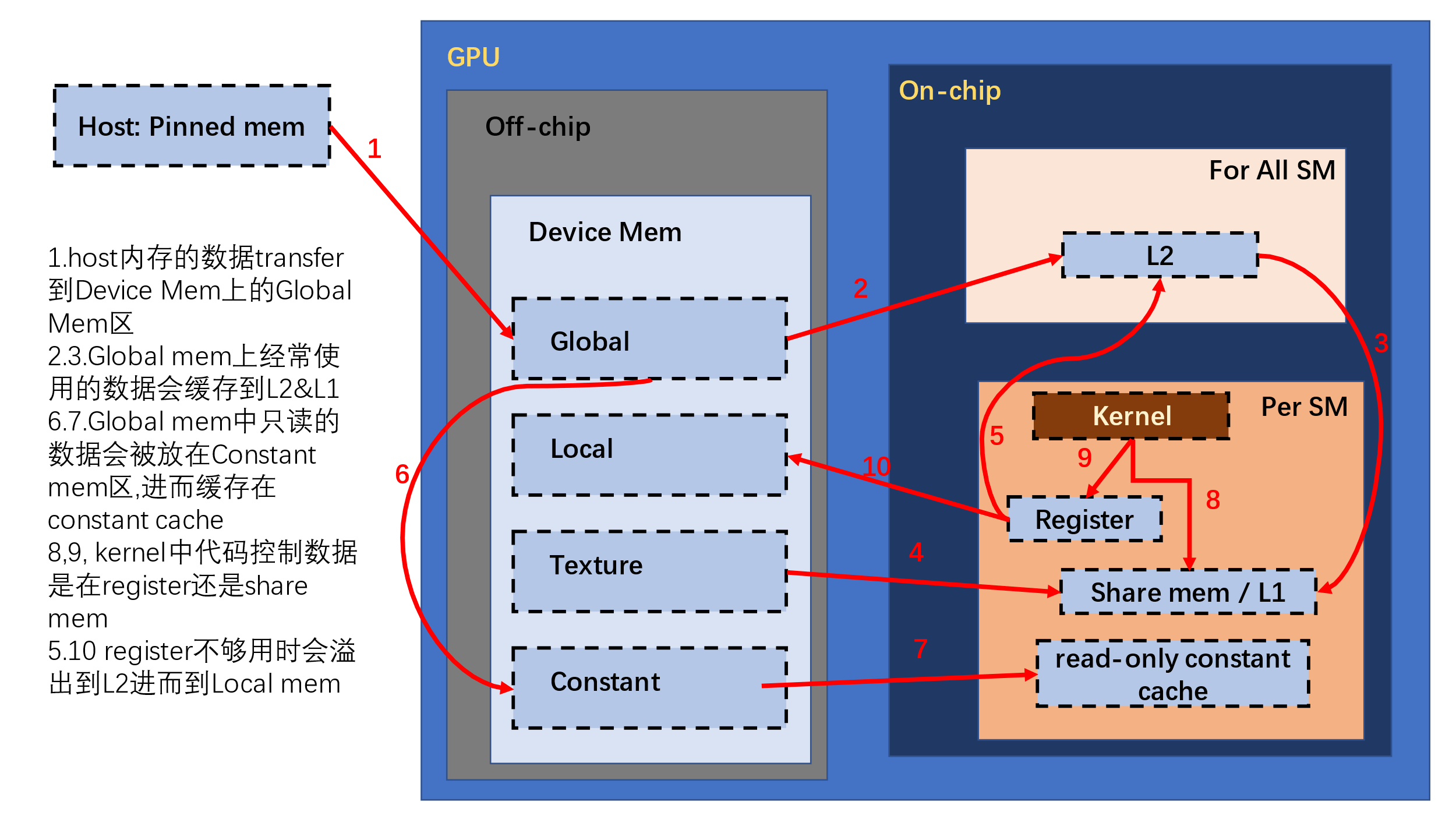
thread中的参数在register(寄存器)中.

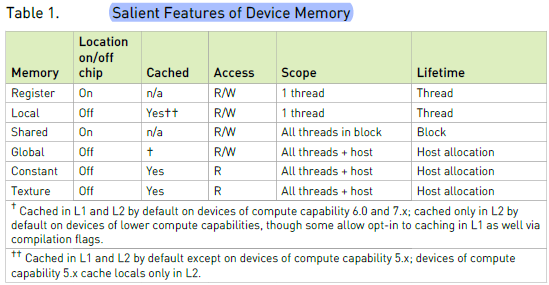
register不够用时,会溢出到L1,都不够用会溢出到LocalMem(GlobalMem的一部分),LocalMem会严重拖慢速度.

访问速度由快到慢分别是:

register->L0->L1->Constant缓存->Share Mem->L2->Local Mem->Global Mem







Global一般缓存在L2,除非添加编译选项,否则不缓存在L1

* + 1. **Pinned Mem**

见\_CUPG/CUDA Runtime/Page-Locked Host Memory. 也提到了Page-Locked更多的优化用法.

这不是在GPU上的,而是在主机内存中,但使用Pinned Mem能达到更大的host->device数据传输带宽与速度.

Pinned Mem是稀缺资源,且一块GPU会单独对应一块Pinnied Mem. 过多使用会影响整个os和服务器的性能.

**BestPractice:**

尽可能在允许范围内使用Pinned Mem

很多显存优化的技巧都需要和pinned mem一起使用才快,比如cudaMallocaPitch,所以尽可能用pin mem

* + 1. **Device Mem**

Device men 是off-chip的, 被划分成global mem和local mem ??? 何时被划分?或者说何时Global mem和local mem会生成或可用?

见\_CUBP/Figure 2. Memory spaces on a CUDA device 和 Table 1. Salient Features of Device Memory. Global mem和Local mem都是device mem上的一个区域,它们是给各个SM用的, 前者的作用范围是1 thread后者是All threads + host. 而Constant mem和Texture mem也是device mem上的区域,作用范围均是All threads + host, 但SM无法直接调用, 而是需要先缓存在 constant / texture cache上.

**BestPractice:**

尽可能使用cudaMallocaPitch/Malloca3D/Memcpy2D/Memcpy3D函数(\_CUPG/CUDA Runtime/Device Memory)

* + 1. **Global Mem**

Host application的所有kernel,所有thread可访问. 生命周期与Host application一样. 各SM共享. off-chip,在device mem上.

Global memory is a 49-bit virtual address space that is mapped to physical memory on the device, pinned system memory, or peer memory. Global memory is visible to all threads in the GPU. Global memory is accessed through the SM L1 and GPU L2. (\_KPG/p13)

**访问GlbMem**

(\_CUPG/Performance Guidelines/Maximize Memory Throughput/ Global Memory)

device memory (含GlbMem, LclMem, ConstantMem) is accessed via 32-, 64-, or 128-byte memory transactions. These memory transactions must be naturally aligned: Only the 32-, 64-, or 128-byte segments of device memory that are aligned to their size (i.e., whose first address is a multiple of their size) can be read or written by memory transactions.

warp对GlbMem的访问会转化成上述的memory transaction, 用越少的transaction能获取完warp所需的数据就越好.

文中也给出了如何提高transaction数据传输效率的建议.

**GlbMem的内存对齐**;

(\_CUPG/Performance Guidelines/Maximize Memory Throughput/ Size and Alignment Requirement)

用易于对齐的数据类型(1,2,4,8,16字节);

Any access (via a variable or a pointer) to data residing in global memory compiles to a single global memory instruction if and only if the size of the data type is 1, 2, 4, 8, or 16 bytes and the data is naturally aligned (i.e., its address is a multiple of that size).

If this size and alignment requirement is not fulfilled, the access compiles to multiple instructions with interleaved access patterns that prevent these instructions from fully coalescing.

使用Built-in Vector Types;

The alignment requirement is automatically fulfilled for the Built-in Vector Types

对于structure类型,可用\_\_align\_\_强制对齐.

For structures, the size and alignment requirements can be enforced by the compiler using the alignment specifiers \_\_align\_\_(8) or \_\_align\_\_(16),

为了对齐,有时候需要padding.(使用\_\_aligin\_\_可让编译器自动padding,或者参考oneflow的Pack类做padding)

用GPU的API中\*Malloc()类方法申请的内存编译器会对齐到256byte.

Any address of a variable residing in global memory or returned by one of the memory allocation routines from the driver or runtime API is always aligned to at least 256 bytes.

对于2D Array用\*MallocPitch()快些(在用Pinned Mem的情况下)

[cuda memory alignment](https://stackoverflow.com/questions/12778949/cuda-memory-alignment) 讲了如何用\_\_align\_\_对结构体进行内存对齐

**参考:**

<GPU架构/Memory/GlbMem->L2->L1>

* + 1. **L2**

见\_CUPG/CUDA Runtime/Device Memory L2 Access Management. 也提到了很多可调试的L2的属性.

经常访问的Global Mem中的数据或常用指令会被缓存到L2. 不保证缓存一致性.

cuda11起可以编程操作部分L2

cache line可简单理解为被cache在L1/L2上的数据

All GPU units communicate to main memory through the Level 2 cache, also known as the L2. L2 also includes hardware to perform compression and global atomics.(\_KPG/p14)

\_CUPG/Compute Capability:

L2 cache shared by all SMs. The L2 cache is used to cache accesses to local and global memory[CC3], including temporary register spills[CC5].[CC3,5,6,7,8]

The cache behavior (e.g., whether reads are cached in both L1 and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load or store instruction.[CC3,5,6] [L2,1] 怎么配置???

A cache line is 128 bytes and maps to a 128 byte aligned segment in device memory. Memory accesses that are cached in both L1 and L2 are serviced with 128-byte memory transactions, whereas memory accesses that are cached in L2 only are serviced with 32-byte memory transactions. Caching in L2 only can therefore reduce over-fetch, for example, in the case of scattered memory accesses.[CC3,5,6,7,8] [L2,1] 怎么控制cache在L2还是L1???

capability 5.0. For devices of compute capability 5.2, it is, by default, not cached in the unified L1/texture cache, but caching may be enabled using the following mechanisms: [CC5][L2,1]

**参考:**

<GPU架构/Memory/GlbMem->L2->L1>

* + 1. **L1**

L1和Share Mem其实是同一块内存的不同部分, L1无法编程控制. 在一个SM中由同一block的不同thread共享, 生命周期与block一致. 不保证缓存一致性.

kernel经常访问的数据或常用指令会被GPU放到L1. Register不够时也会溢出到L1.

---\_V100/p12---start

The merger of shared memory and L1 resources enables an increase in shared memory capacity to 96 KB per Volta SM, compared to 64 KB in GP100. (图中L1/Shared总共128K,而share顶多96K,其它是L1)

---\_V100---end

L1, plays a key role in handling global, local, shared, texture, and surface memory reads and writes, as well as reduction and atomic operations. On Volta and Turing architectures there are , there are two L1 caches per TPC, one for each SM.(\_KPG/p14)

it should be understood that the L1 data cache, shared data, and the Texture data cache are one and the same. (\_KPG/p14)

L1 receives requests from two units: the SM and TEX. L1 receives global and local memory requests from the SM and receives texture and surface requests from TEX. These operations access memory in the global memory space, which L1 sends through L2. (\_KPG/p14)

(\_CUPG/Compute Capability:)

An SM has a unified L1/texture cache of 24 KB used to cache reads from global memory. [CC5,6]

大小: 48KB[CC6], 128/96KB[CC7], 192/128KB[CC8]

作用: used by the texture unit that implements the various addressing modes and data filtering mentioned in Texture and Surface Memory.[CC5,6,7,8]

Shared memory is partitioned out of unified data cache, and can be configured to various sizes(不同kernel可通过cudaFuncSetAttribute配置不同的share/L1大小). The remaining data cache serves as an L1 cache. [CC7,8]

L1和Share mem在on-chip的同一块mem上,可通过\*FuncSetCacheConfig()配置它们各占该mem的多少. [CC3]

所以用了read-only cache或L1 cache却没用share mem的地方, CC3的话可以通过cudaFuncSetCacheConfig()之类的函数设置成大L1小share的模式,CC7+的话则通过cudaFuncSetAttribute(MyKernel, cudaFuncAttributePreferredSharedMemoryCarveout, carveout)设置.(\_CUPG/K.6.4(p362))

**作用:**

可划出一部分用于share mem和texture cache. 可配置划出多少给到share mem

若readonly的数据,编译器可让它存在L1上,但需要手工用\_\_ldg()或’const+\_\_restrict\_\_’提高编译器将其保存在L1的几率.

**参考:**

<GPU架构/Memory/GlbMem->L2->L1>

* + 1. **GlbMem->L2->L1**

GlbMem内存分配:

用GPU的API中\*Malloc()类方法申请的内存编译器会对齐到256byte. <GPU架构/Memory/Global Mem>

对齐读取:

<GPU架构/Memory/Global Mem>

用易于对齐的数据类型(1,2,4,8,16字节);

用Built-in Vector Types;

对于structure类型,可用\_\_align\_\_强制对齐.

GlbMem数据传输:

<GPU架构/Memory/Global Mem>

Only the 32-, 64-, or 128-byte segments of device memory that are aligned to their size (i.e., whose first address is a multiple of their size) can be read or written by memory transactions.

如果是CC5以下,对GlbMem的访问会缓存在L2和L1

[CC3] (\_CUPG/Compute Capabilities)

默认是GlbMem->L2->L1, 但可以在nvcc参数加入’ -Xptxas -dlcm=cg’ 禁用L1, 或 -Xptxas -dlcm=ca 显示启用L1.

如果是CC5以上,地GlbMem的访问缓存在L2,而是否缓存在L1是可选的

缓存在L2.

[CC3] (\_CUPG/Compute Capabilities)

A cache line is 128 bytes and maps to a 128 byte aligned segment in device memory.

memory accesses that are cached in L2 only are serviced with 32-byte memory transactions. Caching in L2 only can therefore reduce over-fetch, for example, in the case of scattered memory accesses.

kernel从global mem读写数据,会先访问L2,每次传输以32byte的倍数的块进行, 一个warp中相邻的thread读取传输块中相邻的数据. 如果每个thread访问的数据大于4byte,会分成多个128byte的访问请求(每个thread读8byte,该warp会发出2个128byte的读global请求;16byte则是4个128byte请求) 详见. (\_CUPG/Compute Capabilities/CC3)

对于只读数据, 可通过下述方法额外缓存在unified L1/texture cache中

(\_CUPG/Compute Capabilities 和 Read-Only Data Cache Load Function)和[CC5] (\_CUPG/Compute Capabilities)

使用\_\_ldg()显示指示编译器使用L1或使用const和\_\_restrict\_\_修饰指针以增加编译器使用L1的几率.(\_CUPG/ Read-Only Data Cache Load Function)

使用以下手段将GblMem的数据(默认/强制)缓存到L1:

对于内核整个生命周期不是只读的数据不能缓存在计算能力为5.0的设备的统一L1 /纹理缓存中。 对于计算能力5.2的设备，默认情况下，它不会缓存在统一的L1 /纹理缓存中，但可以使用以下机制启用缓存：[CC5,6,7]

使用PTX参考手册中所述的相应修改器，使用内联汇编执行读取;

使用-Xptxas -dlcm = ca编译标志进行编译，在这种情况下，所有读取都将被缓存，但使用内联汇编使用禁用缓存的修饰符执行的读取除外;

使用-Xptxas -fscm = ca编译标志进行编译，在这种情况下，所有读取都将被缓存，包括使用内联汇编执行的读取，而不管所使用的修改器如何。

当使用上面列出的三种机制启用缓存时，计算能力5.2的设备将全局内存读取缓存到统一的L1 /纹理缓存中

关闭L1对GlbMem数据的缓存: Using -Xptxas="-dlcm=cg" will force the assembler to generate 32 byte rather than 128 byte transactions which bypass the L1 cache. 见[Stack Overflow](https://stackoverflow.com/questions/11274853/is-cuda-shared-memory-also-cached)

仅缓存在L2比在L2+L1好??? [CUDA ---- Memory Access](https://www.cnblogs.com/1024incn/p/4573566.html) 是这么说的,但实验结果只部分支持

**参考:**

[CUDA ---- Memory Access](https://www.cnblogs.com/1024incn/p/4573566.html)  讲了在显存读写时需要注意的项和如何优化,基于< professional cuda c programming>

[How to Access Global Memory Efficiently in CUDA C/C++ Kernels](https://app.yinxiang.com/shard/s30/nl/5421460/4ceb1ad3-e3c8-408a-8ce5-68208799447d)  TODO

<GPU架构/Memory/Global Mem 和 L2 和 L1>

* + 1. **Share Mem**

L1和Share Mem其实是同一块内存的不同部分, Share Mem可使用\_\_shared\_\_关键字编程控制. 在一个SM中由同一block的不同thread共享, 生命周期与block一致.

一般用于同一block的thread间通信.

分成多个bank,一般是32个,也就是一个thread一个bank. 内存冲突或bank conflict会造成访问变慢. 可通过api把bank调大以减少bank conflict发生.

The total size of shared memory may be set to 16KB, 32KB or 48KB (with the remaining amount automatically used for L1 Cache) .

SM中共享内存使用太多，会导致SM上活跃的线程数量减少，也会影响程序的运行效率。

compute5或以上:

有32个bank.

每个bank每clock cycle有32bit的bandwidth,刚好4byte,装一个int/float/long

某个compute capbility以上还可以设置bank的bandwidth以适应传输不同的数据类型?

看\_CUBP/Memory Optimization/Share Memory/Share Memory and Memory Banks的注释, 理解基于bank的地址分配与bank conflict.

**\_CUPG/Performance Guidelines/Maximize Memory Throughput:**

消除bank conflict

Shared Memory的一个bank宽度为4B,

**\_CUPG/Compute Capability:**

32个bank,每个bank的bandwidth per clock cycle可设置为32bit/64bit [CC3+]

A shared memory request for a warp does not generate a bank conflict between two threads that access any sub-word within the same 32-bit word or within two 32-bit words whose indices i and j are in the same 64-word aligned segment (i.e., a segment whose first index is a multiple of 64) and such that j=i+32 (even though the addresses of the two sub-words fall in the same bank). In that case, for read accesses, the 32-bit words are broadcast to the requesting threads and for write accesses, each sub-word is written by only one of the threads (which thread performs the write is undefined).也就是若不同thread访问同一bank的32/64bit地址内的不同数据,都不会bank conflict,而是产生broadcast. [CC3+]

可通过cudaFuncSetAttribute设置share mem和L1的大小占比[CC7+]

* + 1. **Register(寄存器)**

仅对thread自己可见,生命周期与thread一致.

保存thread的本地变量,常数长度的数组,指令..

速度快,0 clock. 但read after write 要24 block, 要尽量避免read after write.

一个register有32bit,所以一个变量至少占用一个register,double类型占2个.( \_CUPG/p114)

(\_CUBP/9.2.7 Registers)

Generally, accessing a register consumes zero extra clock cycles per instruction, but delays

may occur due to register read-after-write dependencies and register memory bank conflicts.

The compiler and hardware thread scheduler will schedule instructions as optimally as

possible to avoid register memory bank conflicts. An application has no direct control over

these bank conflicts. In particular, there is no register-related reason to pack data into vector

data types such as float4 or int4 types.

9.2.7.1. Register Pressure

Register pressure occurs when there are not enough registers available for a given task.

Even though each multiprocessor contains thousands of 32-bit registers (see Features and

Technical Specifications of the CUDA C++ Programming Guide), these are partitioned among

concurrent threads. To prevent the compiler from allocating too many registers, use the -

maxrregcount=N compiler command-line option (see nvcc) or the launch bounds kernel

definition qualifier (see Execution Configuration of the CUDA C++ Programming Guide) to control

the maximum number of registers to allocated per thread.

* + 1. **Constant memory&cache (all read-only)**

GlbMem中有一块64KB的Constant Mem, 通过 \_\_device\_\_ \_\_constatn\_\_修饰符可以把变量创建在constant mem中.

对constant mem的数据会加载到constant cache(每SM有8KB,但文献没说在哪儿))

An SM has a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory. (\_CUPG/Compute Capabilities)[对于所有CC]

(\_CUPG/Performance Guidelines/Maximize Memory Throughput/Constant Memory) 说constant mem&cache的访问规则???但没看懂

(\_CUPG/Performance Guidelines/Maximize Memory Throughput/Texture and Surface Memory) 说某些场景用texture/surface memory比constant mem好

[CUDA Constant Memory Best Practices - Stack Overflow](https://stackoverflow.com/questions/18020647/cuda-constant-memory-best-practices)

‘a warp must access a single address in order to have any beneficial optimization/speedup gains from constant memory access’. warp中所有thread都读同一地址才能享受constant cache带来的加速.

the data readed each time must be <= 32bit, because‘if you know that in general your accesses will break the constant memory one 32-bit quantity per cycle rule, then you'll probably be better off leaving the data in ordinary global memory’. 每次读的数据<=32bit,因为constant cache的时钟周期带宽只有32bit.

**参考:**

[gpgpu - Why is the constant memory size limited in CUDA? - Stack Overflow](https://stackoverflow.com/questions/10256402/why-is-the-constant-memory-size-limited-in-cuda)

[Constant Cache - CUDA / CUDA Programming and Performance - NVIDIA Developer Forums](https://forums.developer.nvidia.com/t/constant-cache/61133)

[constant cache - CUDA / CUDA Programming and Performance - NVIDIA Developer Forums](https://forums.developer.nvidia.com/t/constant-cache/33021)

[CUDA Constant Memory Best Practices - Stack Overflow](https://stackoverflow.com/questions/18020647/cuda-constant-memory-best-practices)

(\_CUPG/Performance Guidelines/Maximize Memory Throughput/Constant Memory)

* + 1. **Local Mem**

仅对thread自己可见,生命周期与thread一致. 是device Mem的一部分,也可以被L2缓存, 但速度比register, L1, share mem, constant缓存慢.

Register不够时会先溢出到L1,L2,再不够就会溢出到Local Mem.

One difference between global and local memory is that local memory is arranged such that consecutive 32-bit words are accessed by consecutive thread IDs. Accesses are therefore fully coalesced as long as all threads in a warp access the same relative address (e.g., same index in an array variable, same member in a structure variable, etc.).(\_KPG/p13)

最佳实践:

尽量不要溢出到local mem:

\_CUPG/Performance Guidelines/Maximize Memory Throughput: 因此不要: 使用长度不确定的array; 使用小structure和array; 不要使用太多register(每个SM有一定的register限制, 可使用-ptx或-keep编译选项可知local mem使用情况. 使用cuobjdump查看cubin,去看看是否过度使用register)

kernel中访问本地array的idx不能是动态的,否则也会被放在lcl mem. (\_b1)

访问lcl mem时尽量要保证warp中每个thread都访问同样的地址,比如lcl mem中一个array的同一个idx, 否则会产生instruction replay,敲慢的(\_b1)

* + 1. **Constant mem/cache**

SM内有一块缓存\_\_constant\_\_修饰的常量. ??? 在到底device mem还是在on-chip? constant cache在SM(on-chip), 而constant mem在device mem(off-chip).

但访问速度没有\_\_ldg()缓存的数据快. \_\_ldg()是缓存到那一块?

CC3:

An SM has a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory. [CC3,5,6,7,8]

Data that is read-only for the entire lifetime of the kernel can also be cached in the read-only data cache described in the previous section by reading it using the \_\_ldg() function (see Read-Only Data Cache Load Function). When the compiler detects that the read-only condition is satisfied for some data, it will use \_\_ldg() to read it. The compiler might not always be able to detect that the read-only condition is satisfied for some data. Marking pointers used for loading such data with both the const and \_\_restrict\_\_ qualifiers increases the likelihood that the compiler will detect the read-only condition.

* + 1. **Texture / Surface Cache**

??? 具体是啥,怎么用

在无法有效使用global/share mem的情况下(比如不能byte align,或block sz不能取warp sz倍数),可使用该cache.

??? 可使用专门为它设计的CUDA array. ? 但找不到用它们来做矩阵运算的例子.

(From 5.3 of CUDA manual) Reading device memory through texture or surface fetching present some benefits that can make it an advantageous alternative to reading device memory from global or constant memory:

• If memory reads to global or constant memory will not be coalesced, higher bandwidth can be achieved providing that there is locality in the texture fetches or surface reads (this is less likely for devices of compute capability 2.x given that global memory reads are cached on these devices);

• Addressing calculations are performed outside the kernel by dedicated units;

• Packed data may be broadcast to separate variables in a single operation;

• 8-bit and 16-bit integer input data may be optionally converted to 32-bit floating-point values in the range [0.0, 1.0] or [-1.0, 1.0] (see Section 3.2.4.1.1)

* 1. **资料:**

[Nvidia GPU架构 - Cuda Core，SM，SP等等傻傻分不清？](https://blog.csdn.net/asasasaababab/article/details/80447254)

[Nvidia GPU不同架构L1 cahce的一些区别](https://zhuanlan.zhihu.com/p/273657259)

<NVIDIA TESLA V100 GPU ARCHITECTURE><NVIDIA TESLA V100 GPU volta ARCHITECTURE><volta ARCHITECTURE whitepaper>

<CUDA\_C\_Programming\_Guide\_10.2>/B.17. Warp matrix functions

[NVIDIA GPU技术和架构演进-技术圈)](https://jishuin.proginn.com/p/763bfbd4ef00)

[NVIDIA GPU 架构演进 | Chenfan Blog](https://jcf94.com/2020/05/24/2020-05-24-nvidia-arch/)

[【浅析】CPU中的指令周期、CPU周期和时钟周期](https://zhuanlan.zhihu.com/p/90829922)

[GPU编程3--GPU内存深入了解](https://zhuanlan.zhihu.com/p/158548901)

[GPU处理能力指标（TFLOPS/TOPS）](https://blog.csdn.net/m0_46070173/article/details/106525362), [GFLOPs、GMACs、FMA之间的关系](https://blog.csdn.net/touchwolf/article/details/106997506), [CNN 模型所需的计算力flops是什么？怎么计算？](https://zhuanlan.zhihu.com/p/137719986)

[cuda - Maximum number of concurrent kernels & virtual code architecture - Stack Overflow](https://stackoverflow.com/questions/41091633/maximum-number-of-concurrent-kernels-virtual-code-architecture)

* 1. **最佳实践**

[CUDA ---- Warp解析](https://www.cnblogs.com/1024incn/p/4541313.html)

[GPU Memory Types - Performance Comparison | Microway](https://www.microway.com/hpc-tech-tips/gpu-memory-types-performance-comparison/)

1. **GPU编程CUDA**
   1. **概念**

CUDA runtime和driver API：

平常安装使用的CUDA，即runtimeAPI，是构建在Driver API上的另一层封装，所有的API都是以4个字母cuda开头. Driver API功能更强大但用起来更麻烦,以cu开头. [浅谈Cuda driver API](https://zhuanlan.zhihu.com/p/111602648)

Device

Context:

Stream:

CUDA Stream是指一堆异步的CUDA操作，他们按照host代码调用的顺序执行在device上, Stream维护了这些操作的顺序. 同一个stream里的操作有严格的执行顺序，不同的stream则没有此限制。default stream是同步串行的，且一个进程内的所有线程都在default stream下，需要显式声明default之外的Stream才可以实现多流并发。多流可实现多线程的计算与数据运算并行.

Hyper-Queue:

允许许多个CPU 线程或进程同时加载任务到一个GPU上， 实现CUDA kernels的并发执行.

MPS:

允许多个CPU进程共享同一GPU context，不同进程的kernel和memcpy操作在同一GPU上并发执行，以实现最大化 GPU利用率、减少GPU上下文的切换时间与存储空间

Kernel: 方法

资料

[详解CUDA的Context、Stream、Warp、SM、SP、Kernel、Block、Grid](https://zhuanlan.zhihu.com/p/266633373)

<CUDA\_C\_Programming\_Guide\_11.4>/Driver API/

* + 1. **Context**

类似CPU的进程的上下文. 每个Context有自己的地址空间，之间是隔离的，在一个Context中有效的东西（例如某个指针，指向一段显存；或者某个纹理对象），只能在这一个Context中使用。但一个CUDA Context中的任何一个kernel，挂掉后，则整个Context中的所有东西都会失效.

CUDA Runtime中,一个CUDA程序中对于特定gpu只能有一个context,虽然使用CUDA Driver可创建多个context,但由于多个context带来的创建的消耗和切换的消耗,不建议使用多context(\_CUBP/10.6)

* + 1. **Stream**

见\_CUPG/CUDA Runtime/Stream

注意默认的—default-stream值即legacy时,NULL stream的特殊行为(Implicit Synchronization)

但一般都会使用--default-stream per-thread去除NULL stream的同步影响,增加并发度

stream还可以在一些command执行完后调起host callback,见\_CUPG/CUDA Runtime/Stream/Host Functions(Callbacks)

不同stream可以有不同优先级,SM会优先考虑优先级高的stream的instruction thread,见\_CUPG/CUDA Runtime/Stream/Stream Priorities

\_CUPG/p51: cudaStreamWaitEvent() can therefore be used to synchronize multiple devices with each other.

**BestPractice:**

\_CUPG/p39:无依赖的指令尽可能在有依赖指令前发出;任何sync行为都尽可能推迟.

使用\_CUPG/CUDA Runtime/Stream/Overlapping Behavior中例程的写法

* + 1. **Event**

见 \_CUPG/CUDA Runtime/Events

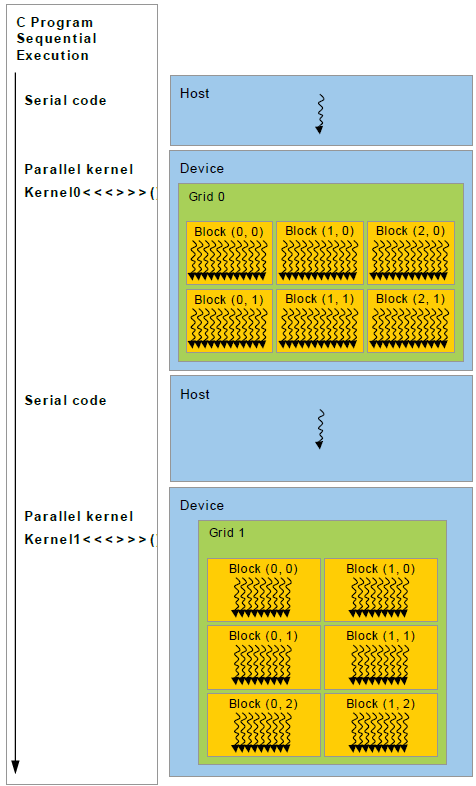
用于统计Elapsed Time. (用cudaEventSynchronize()和cudaEventElapsedTime())

用于stream间(可跨device)的同步调度: \_CUPG/CUDA Runtime/Stream/Explicit Synchronization 和\_CUPG/p51

* 1. **编程模型**

一次对GPU的调用,要写kernel函数, 一个kernel对应一个grid的线程.

kernel的线程层级结构如下:



一个kernel对应一个grid,用dim3表示,一个grid有多个block(dim3),一个block最多1024 thread, 也就是32 warp. grid可以有多少个block请查阅GPU版本对应的compute capability.

* + 1. **grid**

grid中所有线程共享global memory.

**gird\_size的确定:**

kernel启动参数中grid\_size是dim3,block中的总线程数为该dim3的x\*y\*z

约束:

grid\_size 也有上限，为 Maximum x-dimension of a grid of thread blocks ，目前在主流架构上都是 2^31 - 1，对于很多情况都是足够大的值。(GBSZ)

grid\_size 的数值不应低于 GPU 上 SM 的数量，否则会有 SM 处于空闲状态(GBSZ)

一般先确定block\_size再确定grid\_size, 也就是确定总的线程数量 (GBSZ)

要避免tail effect:

GPU 一次可以调度 SM 数量 \* 每个 SM 最大 block 数个 block，因为每个 block 的计算量相等，所以所有 SM 应几乎同时完成这些 block 的计算，然后处理下一批，这其中的每一批被称之为一个 wave。想象如果 grid\_size 恰好比一个 wave 多出一个 block，因为 stream 上的下个 kernel 要等这个 kernel 完全执行完成后才能开始执行，所以第一个 wave 完成后，GPU 上将只有一个 block 在执行，GPU 的实际利用率会很低，这种情况被称之为 tail effect。(GBSZ)

将 grid\_size 设置为精确的一个 wave 可能也无法避免 tail effect，因为 GPU 可能不是被当前 stream 独占的，常见的如 NCCL 执行时会占用一些 SM。所以无特殊情况，可以将 grid\_size 设置为数量足够多的整数个 wave，往往会取得比较理想的结果，如果数量足够多，不是整数个 wave 往往影响也不大。(GBSZ)

**最佳实践:**

除了上面提到的那些的其它最佳实践

* + 1. **block**

Block也叫CTA(Cooperative Thread Array)(\_KPG), 有的中文文献也称为’线程块’(OEW)

同一个 block 中，连续的 32 个线程组成一个 warp，这 32 个线程每次执行同一条指令，也就是所谓的 SIMT，即使最后一个 warp 中有效的线程数量不足 32，也要使用相同的硬件资源(GBSZ)

与 block 对应的硬件级别为 SM，SM 为同一个 block 中的线程提供通信和同步等所需的硬件资源，跨 SM 不支持对应的通信，所以一个 block 中的所有线程都是执行在同一个 SM 上的，而且因为线程之间可能同步，所以一旦 block 开始在 SM 上执行，block 中的所有线程同时在同一个 SM 中执行（并发，不是并行），也就是说 block 调度到 SM 的过程是原子的。SM 允许多于一个 block 在其上并发执行，如果一个 SM 空闲的资源满足一个 block 的执行，那么这个 block 就可以被立即调度到该 SM 上执行，具体的硬件资源一般包括寄存器、shared memory、以及各种调度相关的资源。.blcok中所有线程共享share mem. (GBSZ)

这里的调度相关的资源一般会表现为两个具体的限制，Maximum number of resident blocks per SM 和 Maximum number of resident threads per SM ，也就是 SM 上最大同时执行的 block 数量和线程数量. (GBSZ)

block的线程数最好不要少于128或256.

block的大小决定了kernel所需的资源和运算速度,注意调优. 每个SM有一个32位register集合放在register file中，还有固定数量的shared memory，这些资源都被thread瓜分了，由于资源是有限的，所以，如果thread比较多，那么每个thread占用资源就叫少，thread较少，占用资源就较多，这需要根据自己的要求作出一个平衡。

blk中的threadId都从0开始按blockDim.x连续递增.(\_CUPG/2.2 Thread Hierarchy)

**block\_size的确定:**

kernel启动参数中block\_size是dim3,block中的总线程数为该dim3的x\*y\*z

Maximum number of threads per block 以及 Maximum x- or y-dimension of a block 都是 1024，所以 block\_size 最大可以取 1024。(GBSZ)

一个 kernel 的 block\_size 应大于 SM 上最大线程数和最大 block 数量的比值，否则就无法达到 100% 的 Occupancy，对应不同的架构，这个比值不相同，对于 V100 、 A100、 GTX 1080 Ti 是 2048 / 32 = 64，对于 RTX 3090 是 1536 / 16 = 96，所以为了适配主流架构，如果静态设置 block\_size 不应小于 96。(GBSZ)

考虑到 block 调度的原子性，那么 block\_size 应为 SM 最大线程数的约数，否则也无法达到 100% 的 Occupancy，主流架构的 GPU 的 SM 最大线程数的公约是 512，96 以上的约数还包括 128 和 256，也就是到目前为止，block\_size 的可选值仅剩下 128 / 256 / 512 三个值。(GBSZ) . 但若只考虑特定架构,如V100,blkSz取1024及其比96大的约数也是可以的.

block 调度到 SM 是原子性的，所以 SM 必须满足至少一个 block 运行所需的资源，资源包括 shared memory 和寄存器， (GBSZ)

shared memory 一般都是开发者显式控制的，而如果 block 中线程的数量 \* 每个线程所需的寄存器数量大于 SM 支持的每 block 寄存器最大数量，kernel 就会启动失败。(GBSZ)

目前主流架构上，SM 支持的每 block 寄存器最大数量为 32K 或 64K 个 32bit 寄存器，每个线程最大可使用 255 个 32bit 寄存器，编译器也不会为线程分配更多的寄存器，所以从寄存器的角度来说，每个 SM 至少可以支持 128 或者 256 个线程，block\_size 为 128 可以杜绝因寄存器数量导致的启动失败，但是很少的 kernel 可以用到这么多的寄存器，同时 SM 上只同时执行 128 或者 256 个线程，也可能会有潜在的性能问题。但把 block\_size 设置为 128，相对于 256 和 512 也没有什么损失，128 作为 block\_size 的一个作为调试起点的通用值是非常合适的。(GBSZ).

如果因为 shared\_memory 的限制导致一个 SM 只能同时执行很少的 block，那么增加 block\_size 有机会提高性能，但如果 kernel 中有线程间同步，那么过大的 block\_size 会导致实际的 SM 利用率降低 (GBSZ提出, 在LN1,SX1有对该原则的应用)

在blk内要做线程同步的情况下:(比如blkAllReduce时)

由于 Block 内线程要做同步，当 SM 中正在调度执行的一个 Block 到达同步点时，SM 内可执行 Warp 逐渐减少，若同时执行的 Block 只有一个，则 SM 中可同时执行的 Warp 会在此时逐渐降成0，会导致计算资源空闲，造成浪费，若此时同时有其他 Block 在执行，则在一个 Block 到达同步点时仍然有其他 Block 可以执行。(LN1) 当 block\_size 越小时，SM 可同时调度的 Block 越多，因此在这种情况下 block\_size 越小越好。但是当在调大 block\_size，SM 能同时调度的 Block 数不变的情况下，block\_size 应该是越大越好，越大就有越好的并行度。因此代码中在选择 block\_size 时，对不同 block\_size 都计算了 cudaOccupancyMaxActiveBlocksPerMultiprocessor，若结果相同，使用较大的 block\_size。(LN1 ???具体怎么做的要看代码) 但个人理解是先把blkSz调小到最小可接受的程度,比如256,然后看此时blk数能增大到多少,然后固定blk数目的前提下再调大blkSz,比如到512.

**最佳实践:**

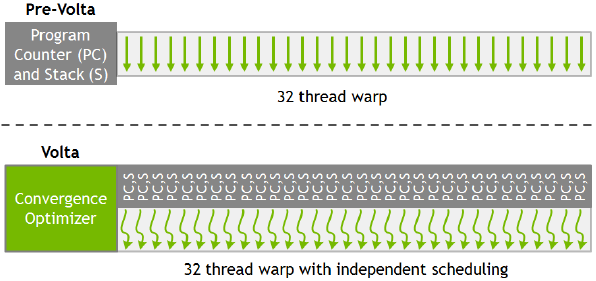
保证block的数目远大于SM的数目

* + 1. **warp**

SM的warp scheduler负责调度warp(32 thread), 同一个warp中的thread可以以任意顺序执行，active warps被SM资源限制。当一个warp空闲时，SM就可以调度驻留在该SM中另一个可用warp。在并发的warp之间切换是没什么消耗的，因为硬件资源早就被分配到所有thread和block，所以新调度的warp的状态已经存储在SM中了。

一般来说一个CUDACore(SP)负责一个thread,但现在Volta的不同数据类型的CUDACore多了,又有了TensorCore,貌似一个thread不同类型的计算可以并行做了.

Volta中warp中每个thread有自己的counter和stack,因此各thread可以有不同控制流分支(也不降低性能),同时也可以通过精细地控制thread的关系来进一步压榨算力. 如下图:



[CUDA ---- Warp解析](https://www.cnblogs.com/1024incn/p/4541313.html) 有很多优化建议:

减少thread中的控制流分支(但貌似在Volta中已不是问题)

SM中warp调度器每个cycle会挑选可获得足够资源(core都有空且所需参数都已准备好)的active warp送去执行，一个被选中的warp称为selected warp，没被选中，但是已经做好准备被执行的称为Eligible warp，没准备好要执行的称为Stalled warp。

CUDA编程中应该重视对计算资源的分配：这些资源限制了active warp的数量。因此，我们必须掌握硬件的一些限制，为了最大化GPU利用率，我们必须最大化active warp的数目。

Latency Hiding: 指令从开始到结束消耗的clock cycle称为指令的latency。当每个cycle都有eligible warp被调度时，计算资源就会得到充分利用，基于此，我们就可以将每个指令的latency隐藏于issue其它warp的指令的过程中。文中有相关方法.

**注意点:**

一个warp有32 thread

threadIdx是从后往前计算的: The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size (Dx, Dy),the thread ID of a thread of index (x, y) is (x + y Dx); for a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is (x + y Dx + z Dx Dy).

参考” <CUDA\_C\_Programming\_Guide\_10.2>/Chapter 2. Programming Model”

* + 1. **thread**

有时为每个 element 创建一个线程是可行的，因为线程的创建在 GPU 上是一个开销足够低的操作，但是如果每个线程都包含一个公共的操作，那么线程数的增多，也代表着这部分的开销变大. (GBSZ中有例子) (???这里的’公共’是指对公共数据的操作吗?)

* 1. **SIMT**

(见\_CUPG/Chapter 4. Hardware Implementation)

* + 1. **流程**

kernel启动

Grid的每个blk会分发到不同SM上,每个有资源的SM会负责一个或多个blk.(\_CUPG/4.1)

当SM上的可用资源(register/share mem)无法满足一个block(SM每次能跑多少warp取决于有多少个warp scheduler)使用时,相关kernel启动会失败.(理论可用资源还是实际可用资源???如果理论可用资源能满足,难道不会等待特定时间让其它block释放资源???)(\_CUBP/10.3)

remember that multiple concurrent blocks can reside on a multiprocessor, (\_CUBP/10.3)

每个warp的执行上下文（execution context，如程序计数器和寄存器等）在warp的整个生命周期内都被保存在片上内存（on-chip memory）。因此从一个执行上下文切换到另一个执行上下文是无开销的。在每个指令执行时间里，warp scheduler都会选择一个warp，这个warp中的所有thread都准备好执行它的下一个指令，warp scheduler会向这些thread发送指令并执行。那么等待这个warp准备好的这段时间（即时钟周期数）就是执行延迟（latency），为了掩盖这个延迟，warp scheduler可以在延迟期间发指令给其他的warp，因此一个SM内的warp数越高，延迟掩盖会越充分，利用率和性能也会越高。而一个SM上同时能并发存在多少个warp和block，是跟该SM上的寄存器数量和shared memory大小相关的。(OFO)

warp启动

(见\_CUPG/4.1. SIMT Architecture)

blk中的每32个thread会组成一个warp, 不够32会填充无效thread. 这32个thread的程序地址相同,但各自有自己的instruction address counter和register state,因此可独立执行(CC7或以上才可以). 每个warp由warp scheduler调起执行.

A warp is allocated to a sub partition and resides on the sub partition from launch to completion. A warp is referred to as active or resident when it is mapped to a sub partition. A sub partition manages a fixed size pool of warps. On Volta architectures, the size of the pool is 16 warps. On Turing architectures the size of the pool is 8 warps. (\_KPG/p13). 每个sub partition对应一个warp scheduler(\_CUPG/5.2.3)

指令发射到thread执行:

(\_CUPG/Performance Guidelines/Maximize Utilization/Multiprocessor Level)

每个clock cycle, warp scheduler都会获取一个准备好的指令,发射到active thread上让其执行.(\_CUPG/5.2.3)

instrution-level parallelism: 发射的指令与同warp的其它指令无依赖,可与同warp其他指令并行执行.

thread-level parallelism: 发射的指令来自其它warp. 可掩盖当前warp的sync或latency.

一个kernel的指令是按顺序发射的. (\_CUBP/10.1)

Active warps can be in eligible state if the warp is ready to issue an instruction. This requires the warp to have a decoded instruction, all input dependencies resolved, and for the function unit to be available. (\_KPG/p13)

Statistics on active, eligible and issuing warps can be collected with the *Scheduler Statistics* section. (\_KPG/p13)

warp中有当前的指令在跑的thread叫active thread, 否则叫inactive. Thrad变inactive有很多原因,比如已经跑完指令,或者是warp不满32时拿去做填充的thread等(\_CUPG/3.1)

warp中不同thread都同一地址进行非原子操作时,结果无定义.做原子操作时是串行进行,但顺序不保证.具体看各CC. (\_CUPG/4.1)

指令发射的硬件限制:CC7+中每个SM有4 个sub partition, 每个sub partition有一个warp scheduler, 共4个warp scheduler, 因此每clock cycle可同时发射4个warp的一个指令(\_CUPG/5.2.3)

---\_KPG/p13---start

A warp is stalled when the warp is waiting on

‣ an instruction fetch,

‣ a memory dependency (result of memory instruction),

‣ an execution dependency (result of previous instruction), or

‣ a synchronization barrier.

See *Warp Scheduler States* for the list of stall reasons that can be profiled and the *Warp State Statistics* section for a summary of warp states found in the kernel execution.

---end---

register的分配

(\_CUBP/Calculating Occupancy)

register是一次性分配给整个blk. 所以某SM的某blk占了太多register就会导致其它blk无法放到该SM,导致该SM的occupancy变低,资源利用率低.

而且CC7+的register是以256为批次分配的,也就是说一个blk所需要的没256个register也会被分配256个.

如果blk所需要的register多于device的物理regiser,编译器???会在保证regiser spill与指令数最少的前提下减少register的使用. 另外regiser的使用也可通过maxrregcount编译选项和\_\_launch\_bounds\_\_()来限制 (\_CUPG/p113末尾) (\_CUBP/Register Pressure)

一个register有32bit,所以一个变量至少占用一个register,double类型占2个.( \_CUPG/p114)

可否disable掉register spill???

The most important resource under the compiler's control is the number of registers used by a kernel. Each sub partition has a set of 32-bit registers, which are allocated by the HW in fixed-size chunks. The Launch Statistics section shows the kernel's register usage.(\_KPG/p13)

CC7+中warp相关的同步有较大改动,要看CC7的章节(\_CUPG/K.6)

* + 1. **Warp Diverge**

(见\_CUPG/4.1. SIMT Architecture)

如果kernel中有if-else之类的控制流,很有可能会导致warp diverge, 也就是一个warp中不同thread的执行路径不一样(有些thread走if有些走else). 但不同warp没有diverge一说.

warp在所有thread的执行路径一样时效率最高.

当warp不同thread执行路径不一样时,所有thread所有路径上的指令都会被执行,即使thread\_1并没有走if语句块中的指令A,该指令A也会被放到thread\_1执行,只是执行结果会被丢弃.

所以warp diverge对性能损害很大,要尽量避免.

* + 1. **Occupancy:**

定义

\_CUBP/10.1

the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps

Higher occupancy does not always equate to higher performance-there is a point above which additional occupancy does not improve performance. However, low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation.

A lower occupancy kernel will have more registers available per thread than a higher occupancy kernel, which may result in less register spilling to local memory; in particular, with a high degree of exposed instruction-level parallelism (ILP) it is, in some cases, possible to fully cover latency with a low occupancy. (\_CUBP/10.3)

The total number of CTAs that can run concurrently on a given GPU is referred to as Wave. Consequently, the size of a Wave scales with the number of available SMs of a GPU, but also with the occupancy of the kernel. (\_KPG)

受影响因素:

Per thread resources required by a CUDA kernel might limit the maximum block size in an unwanted way.

受device上供thread使用的资源的影响,比如register和share mem

The number of CTAs that fit on each SM depends on the physical resources required by the CTA. These resource limiters include the number of threads and registers, shared memory utilization, and hardware barriers. (\_KPG)

Register在设备上的限制

\_CUBP/10.1.1

on devices of compute capability 7.0 each multiprocessor has 65,536 32-bit registers and can have a maximum of 2048 simultaneous threads resident (64 warps x 32 threads per warp). This means that in one of these devices, for a multiprocessor to have 100% occupancy, each thread can use at most 32 registers.

maximum number of registers per thread can be set manually at compilation time per-file using the -maxrregcount option or per-kernel using the \_\_launch\_bounds\_\_ qualifier. (软限制)( \_CUBP/10.1)

Share Mem在设备上的限制

???TODO

一个blk所耗费的share mem是statically allocated shared mem与dynamically allocated shared mem之和.( \_CUPG/p113)

如何查看Occupancy相关信息:

(\_CUBP/10.1.1)

The --ptxas options=v option of nvcc details the number of registers used per thread for each kernel

CUDA\_Occupancy\_Calculator.xls and is located in the tools subdirectory of the CUDA Toolkit installation. (\_CUBP/10.1.1) (\_CUPG/5.2.3.1)

occupancy can be determined using the NVIDIA Nsight Compute Profiler. Details about occupancy are displayed in the Occupancy section.

An application can also use the Occupancy API from the CUDA Runtime, e.g. cudaOccupancyMaxActiveBlocksPerMultiprocessor, to dynamically select launch configurations based on runtime parameters. (\_CUBP/10.1.1) (\_CUPG/5.2.3.1)

* + 1. **latency的引起和掩盖**

(\_CUPG/Performance Guidelines/Maximize Utilization/Multiprocessor Level)

warp为下一条指令做的准备工作所消耗的clock cycle是一种latency, 可通过更多的无依赖的指令(不限于同warp/block)发射到active thread执行来掩盖

指令没准备好有以下情况:

指令依赖的数据没准备好,这比较常见. Latency是数据加载时间.

如果依赖的数据在off-chip mem上,latency会很大,需要更多的warp去掩盖.

指令依赖执行中的上一条指令的计算结果,就是下面所说的register dependencies. Latency是所依赖的指令的执行时间.\_CUPG/p113/2段,举了如何掩盖该latency的例子.

指令在kernel的某同步点之后(\_\_syncthreads同步blk或\_\_syncwarp同步warp), 可通过让SM接纳更多的blk/warp来避免空等同步的完成.(<最大化有效并发>有讲如何让SM容纳更多blk)

register dependencies (read-after-write)会引起latency, 可通过更多的可并行instruction或active thread进行掩盖(\_CUBP/10.2)

kernel中计算指令与(从device mem)数据IO指令的比例最好是8:1左右,才能较好的掩盖IO时延(\_b1)

* + 1. **最大化有效并发(Maximize Utilization 和 Occupancy)**

(\_CUPG/Maximize Utilization)

提高性能的关键是用大小不等的任务把所有计算单元(所以是并发的)每时每刻都刚好塞满, 计算资源刚好用完, 太过, 则会用到local mem, 太少, 则会造成资源限制, 算力没完全发挥. 此时需要每个thread/instruction尽可能不互相依赖, 一个blk/warp所用的计算资源要刚刚好. (\_CUBP/10.1)

与occupancy,active warp/thread,相关.

为何叫有效并发呢?

因为高并发度不一定能带来相应计算速度的提升,有时候也需要低并发.

高occupancy可充分使用device的计算资源,并且有更多的active thread去掩盖由于同步,数据传输等引起的latency. (\_CUBP/10.1)

有时候低occupancy使得有register空闲, 可避免register spill to local mem. 另外, 如果指令并行度instruction-level parallelism高的话, 可以在低occupancy也能掩盖其他指令造成的latency. (\_CUBP/10.3)

**kernel并行执行**

(\_CUPG/3.2.6 Asynchronous Concurrent Execution)

串行任务在host,并行任务在device. (\_CUPG/5.2.1)

尽量调用\*async类的api(\_CUPG/3.2.6)

使用多个stream执行kernel (\_CUBP/10.5)

使用非default stream或non-block stream(\_CUBP/10.5)

为stream设置优先级(\_CUPG/Streams/Stream Priorities)

不要使用multi context(\_CUBP/10.6)

**优化grid size和block size:**

中BP:因为要隐藏由于register dependency导致的latency,所以SM上要跑足够多的active threads,即要有足够大的occupancy.(\_CUBP/10.2)

(\_CUBP/10.3)

GridSz要大于device上SM个数,使得每个SM至少有一个block可做,最好每SM可有多个block可做,以免某block被\_\_syncthreads时有其它block顶上,保证SM随时忙碌.

而此时SM同时负责的多个block最好来自一个或少数几个kernel, 以尽可能使用instruction缓存.

blkSz要考虑到其中每个thread占用的资源,以免block占用的资源多于一个SM,启动失败,或者register spill到local mem.

由于所有blk分散到每个SM上跑,超过SM一半计算资源却又无法填满SM的blkSz会导致每个SM只能跑一个blk,却又跑不满计算资源.

中BP:blkSz最好是warpSz=32的倍数,以利于资源的对齐与利用.

最小blkSz不要低于64,而且只在SM能跑多个blk的时候使用该值.

做blkSz的尝试实验时,可从128或256开始.

使用偏小的blkSz,会使得SM能跑多个blk以隐藏latency,特别是kernel中使用了\_\_syncthreads时.

**优化每个thread对share mem的使用:**

(\_CUBP/10.4)

share mem是针对blk共享的,读写某块share mem时,blk中的每个thread可只操作该块mem其中的一部分.减少thread读写数据的消耗,从而提高单thread的计算访存比;减少thread读写数据的时间,从而提高并发度.

使用dynamically allocated shared memory(也就是kernel的<<<>>>中的第三个参数),增加该值会减少occupancy,要做实验看下对性能的影响.

其它最佳实践::

Providing the two argument version of \_\_launch\_bounds\_\_(maxThreadsPerBlock,minBlocksPerMultiprocessor) can improve performance in some cases. The right value for minBlocksPerMultiprocessor should be determined using a detailed per kernel analysis.(\_CUBP/10.1)

* 1. **Graph**

见 \_CUPG/CUDA Runtime/CUDA Graphs

更好地支持multi stream, 能基于静态图编译使得multi stream的kernel调用更优化. Cuda exampels中有例子.

cuda11.4加入memory node把内存管理纳入计算图优化

cuda11中才解决了graph的io必须是固定内存地址的问题,其它版本都需要先把数据拷贝到固定地址,好让graph处理.

graph中一些参数变量在不同的执行中变化的问题,可通过以下解决:

\_CUPG/CUDA Runtime/CUDA Graphs/Updating Instantiated Graphs 可update整个graph或其中某些node

\_CUPG/CUDA User Objects 类似于graph所依赖的某个obj的值变了后,可通过更新obj的引用让graph得知新值,但不确定能否解决,需要进一步研究???

可用于kernel较小的场景; 或者,kernel较大而导致并发度和occupancy不高时,把kernel切分成小kernel以提升并发与occupancy(当然也要衡量因此带来的launching kernel的overhead cost)

\_CUPG中也说了使用graph时的限制

经验发现:

需要固定pointer,所以内存要复制到算子内存池,如果不是大计算图而是分成多个小计算图,这种内存复制会消耗大量时间

刚开始的多strm还是串行启动,后续才并行,所以大图会更体现优势.

kernel启动基本无间隔. 所有的kernel计算从0.009减少到0.007,约22%多

资料

[NVIDIA/cuda-samples: Samples for CUDA Developers which demonstrates features in CUDA Toolkit (github.com)](https://github.com/NVIDIA/cuda-samples)

[Getting Started with CUDA Graphs | NVIDIA Developer Blog](https://developer.nvidia.com/blog/cuda-graphs/)

<https://developer.nvidia.com/gtc/2020/video/cwe21914>

[浅谈Cuda driver API](https://zhuanlan.zhihu.com/p/111602648)

* 1. **warp shuffle 和 cooperative\_groups ???**

warp shuffle可基于寄存器进行warp不同thread之间的通信和协同计算

cooperative groups 使得warp shuffle更易用

cooperative groups还封装了thread, warp, block , grid级别不同的协同计算操作

但没验证是cooperative groups的性能更好? 还是原生warp shuffle性能更好???

[Using CUDA Warp-Level Primitives | NVIDIA Developer Blog](https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/)

[Cooperative Groups: Flexible CUDA Thread Programming | NVIDIA Developer Blog](https://developer.nvidia.com/blog/cooperative-groups/)

[CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics | NVIDIA Developer Blog](https://developer.nvidia.com/blog/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/)

这篇只说了同样的功能cooperative group用更少的代码,没说比warp shuffle快, 估计性能应该相似. 同时高版本的automicAdd其实就是基于warp shuffle的思想实现的, 比trust库要快.

* 1. **用 \_\_ldg 和 \_\_restrict\_\_ 操作read only cache**

\_\_ldg()提示(强制???)nvcc把thread中只读的变量(如果是只读的话)缓存在read-only cache(L1 cache的一部分)中,并从缓存读取.

const和\_\_restrict\_\_同时出现,提示nvcc可把对应的指针相关的数据用read only cache缓存

read only cache比constant缓存(用\_\_constant\_\_提示)快且大

[cuda - What is the difference between \_\_ldg() intrinsic and a normal execution? - Stack Overflow](https://stackoverflow.com/questions/26603188/what-is-the-difference-between-ldg-intrinsic-and-a-normal-execution)

[cude的\_\_ldg使用](https://blog.csdn.net/jqw11/article/details/104717743)

[pointers - CUDA: \_\_restrict\_\_ tag usage - Stack Overflow](https://stackoverflow.com/questions/43235899/cuda-restrict-tag-usage)

<CUDA\_C\_Programming\_Guide>/B.2.5. \_\_restrict\_\_/

<CUDA\_C\_Programming\_Guide>/K.3.2. Global Memory/和/K.4.2. Global Memory/和/B.10. Read-Only Data Cache Load Function/

\_\_restrict\_\_ 结合 \_\_ldg更好???

* 1. **Data Transfer**

[How to Optimize Data Transfers in CUDA C/C++ | NVIDIA Developer Blog](https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/)

Let’s start with a few general guidelines for host-device data transfers.

* Minimize the amount of data transferred between host and device when possible, even if that means running kernels on the GPU that get little or no speed-up compared to running them on the host CPU.
* Higher bandwidth is possible between the host and the device when using page-locked (or “pinned”) memory.
* Batching many small transfers into one larger transfer performs much better because it eliminates most of the per-transfer overhead.
* Data transfers between the host and device can sometimes be overlapped with kernel execution and other data transfers.
  + 在加速GPU运算和并行度的同时也合符pipeline并行的macrobatch思想
  + 使用cudaMemcpy2D/3D类似的方法,可一小块一小块地运输数据,利于掩盖运输时间

[How to Overlap Data Transfers in CUDA C/C++ | NVIDIA Developer Blog](https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/)

There are several requirements for this to happen:

* The device must be capable of “concurrent copy and execution”. This can be queried from the deviceOverlap field of a cudaDeviceProp struct, or from the output of the deviceQuery sample included with the CUDA SDK/Toolkit. Nearly all devices with compute capability 1.1 and higher have this capability.
* The kernel execution and the data transfer to be overlapped must both occur in different, non-default streams.
* The host memory involved in the data transfer must be pinned memory.
* 文中也提了,不同调用cudaMemcpyAsync和kernel的顺序在不同的卡上性能差别也挺大的.

使用pinned host memory

资料:

(2012)

(2012)

[使用GPU训练大神经网络之 显存控制策略与提升](https://zhuanlan.zhihu.com/p/114718838) TODO

* 1. **编程算法**
     1. **AllReduce**

[CUDA高性能计算经典问题①：归约](https://app.yinxiang.com/shard/s30/nl/5421460/e0df5529-a693-49f8-9d00-5eb12d4d04ed)

warpAllReduce,用warp原语做all reduce. 见(LN1)和LS

blockAllReduce,先用warp原语做all reduce,再通过ShareMem对blk中所有warpAllReduce的结果做all reduce. 见(LN1)和LS.

* 1. **其它???**

\_\_数学原语是SFU中的???

各种设备原语

rand\_init()在nsight compute上看貌似并不耗性能???

DALI是NV推出的能快速加载数据到devices的框架

* 1. **字节的ls**

ls:1.3->0.6,省50%多,快一倍

用cuda重写.但只支持原始trsf,且由于实现不同trch,无法兼容网上的预训练模型,内部版可兼容.

减少锁,虽然f4可较好利用每thread的缓存拿到靠近存储的数据,但由于tkn id稀疏,原子加要锁4个不同地方,且锁的时间就是thread时间,也就是原子加4次的时间

每个thread只做一次原子加,减少锁时间,增加并行度(执行粒度小所需资源少,可用于塞满调度区间). 提升11%:0.6->0.5多

multi stream: 掩盖数据传输,掩盖kernel拉起,加大并行化; 提升16%.:0.534->不到0.45

但多个stream拉起时还是接近串行, 且由于要固定指针需要拷贝数据到内存池,所以使用ms的操作要足够多,否则无法抵消ms拉起和数据进池的耗时.

进一步fused,一些张量shape没有变的操作,其实可以放到一个kernel,比如dropout和transform,activation之类的. 但要考虑kernel每个thread的时间太长影响并行度,特别是里面有sync的情况(LN)下. 提升4%.:0.448->0.42多

best practice

[CUDA C最佳实践-CUDA Best Practices(二)\_每天get√新知识-CSDN博客](https://blog.csdn.net/fishseeker/article/details/76915489)

其它最佳实践

[Parallel Code: Maximizing your Performance Potential | Microway](https://www.microway.com/hpc-tech-tips/parallel-code-maximizing-your-performance-potential/)

<CUDA\_C\_Programming\_Guide\_11.4>/ Chapter 5. Performance Guidelines

<CUDA STREAMS BEST PRACTICES AND COMMON PITFALL 2014>

<CUDA\_C\_Best\_Practices\_Guide.11.4>及其翻译或笔记:

[解读《CUDA C最佳实践指南》 - 知乎 (zhihu.com)](https://www.zhihu.com/column/c_1323748466020315136)

<https://blog.csdn.net/fishseeker/article/details/76904262>,

<https://blog.csdn.net/fishseeker/article/details/76915489>,

<https://blog.csdn.net/FishSeeker/article/details/76915657>

1. **Profile**
   1. **Timing**

见\_CUBP/Performance Metrics/Timing

如果对整个程序(含cpu,gpu部分)计算耗时,记得使用各种cuda\*Synchronize()或设置\*BLOCK环境变量,使得计时不受异步计算影响.

如果仅对cuda部分计算耗时,可使用event.

* 1. **Bandwidth**

见\_CUBP/Performance Metrics/Bandwidth

计算不同存储(host-device,global-share等)间数据传输的带宽(每秒传多少G)

最佳实践:

高:尽量使用有效带宽(effective bandwidth)衡量性能

* 1. **计算访存比**

某kernel中计算所占的(指令数或时间)与读写数据所占的(指令数或时间)的比值.

计算访存比要适当大,这样才能尽量利用读到的数据. 太大会影响thread的并发度. 太小读的数据的利用率又太低,读数据所耗资源的性价比低.

* 1. **使用Visual Profiler???**

见\_CUBP/Performance Metrics/Throughput Reported by Visual Profiler

罗列了visual profiler所提供的各种指标及其含义

注意kernel所想要(request)读写的数据与实际运行时读写的数据量是不一样的,因为有对齐之类的影响(VP中都能看到这些指标)

* 1. **相关编译选项**

-ptxas-options=-v 或 -Xptxas=-v : 显示每个kernel的register, share mem, lcoal mem的使用情况

1. **调用流程**
   1. **init context**

第一次调用需要ctx的cuda函数时发生.(见\_CUPG/CUDA Runtime/Initialization)

* 1. **分配显存**

Memory allocated through the CUDA Runtime API, such as via cudaMalloc(), is guaranteed to be aligned to at least 256 bytes(见\_CUBP/9.2.1.2)

* 1. **kernel启动的配置**

<<<grid\_size, block\_size, 0, stream>>>

1:gridSz为dim3; 2:blockSz为dim3;

3: ype size\_t and specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory; this dynamically allocated memory is used by any of the variables declared as an external array as mentioned in shared; (???share mem的总额多少以及如何分配) (GBSZ)

4:启动kernel用到的stream

**gridSz和blockSz的确定:**

(见<GPU编程CUDA/GPU编程模型/grid>) (见<GPU编程CUDA/GPU编程模型/block>)

根据上述参考: 普通的 elementwise kernel 或者近似的情形中，block\_size 设置为 128，grid\_size 设置为可以满足足够多的 wave 就可以得到一个比较好的结果了.(GBSZ)

数据量较小的情况下，不会启动过多的线程块。在数据量较大的情况下，尽可能将线程块数目设置为数量足够多的整数个 wave，以保证 GPU 实际利用率够高。基于上述参考资料和原则,可以按场景制定自己的gridSz和blockSz的生成策略, 比如, elem wise计算(OEW)

参考:

* 1. **host传输数据到device:**

见\_CUPG/CUDA Runtime/Page-Locked Host Memory 去优化host mem的访问(使用并优化Pinnned Mem)

见\_CUBP/Memory Optimization/Data Transfer Between Host and Device

尽量用\*async方法

* 1. **Global Mem到Kernel**

见\_CUBP/Memory Optimization/Coalesced Access to Global Memory

牢记\_CUBP/Memory Optimization/Table 1. Salient Features of Device Memory

Compute Capability 6 以上:

从global mem读取的数据都会缓存在L2&L1(???怎样才能尽量控制缓存行为呢)

从global mem读(???写呢)数据都是以32byte的倍数进行数据传输的,而传输是按warp发起传给warp中每个thread的

同warp的不同thread以threadId为segment下标,顺序对齐到32倍数的传输segment中. 因此blockSz为warpSz(32)倍数时比较容易对齐.

注意理解例程\_CUBP/A copy kernel that illustrates misaligned accesses和Figure 3&4&5, 理解如何通过对齐来优化访问.

Figure 3上的注释.

若无法对齐,如果访问是顺序的,那么上一轮访问所多拉取的数据会被缓存,这轮访问时与上一轮threadId所相邻的thread会用上cache.因此性能损失比理论上少些.

防止Strided Accesses:

见\_CUBP/Memory Optimization/Coalesced Access to Global Memory/Strided Acesses

比按顺序但不对齐的访问更损bandwidth,因为stride中不需要的数据无法通过cache让相邻的thread访问.

解决方案见\_CUBP/Memory Optimization/Share Memory in Matrix Multiplication(C=AA^T)和相关例程,关注注释.

**\_CUPG/Performance Guidelines/Maximize Memory Throughput:**

尽可能对齐;

用易于对齐的数据类型(1,2,4,8,16字节);使用Built-in Vector Types;对于structure类型,可用\_\_align\_\_强制对齐.

为了对齐,有时候需要padding.

\*Malloc()申请的内存至少对齐到256byte.

对于2D Array用\*MallocPitch()快些(在用Pinned Mem的情况下)

最佳实践:

高:确保对global mem的访问是对齐的

高:尽可能消除对global mem的strided acesses

* 1. **向量化读和操作数据**

使用向量化数据类型从global mem读数据,能带来更少的指令更大的吞吐.CUDA中有内置的向量化类型 (NVT)(\_CUPG/B.3 Built-in Vector Types)

oneflow为了支持更多数据类型的向量,自己实现了支持任何数据类型的向量化类,并支持对无法整除向量化pack的数据块的剩余数据的处理.(OEW)(LN1)

其中对无法整除的剩余数据的处理,在(OEW/调用链)对ApplyGeneric的说明中描述.(???oneflow在这方面的代码挺有用)

CUDA也对half2做了优化, 提供了half2进行加减乘除的API. 参考(\_CUPG/Performance Guidelines/Maximize Memory Throughput/Arithmetic Instructions/Half Precision Arithmetic)

oneflow对CUDA相关half2的转换和计算进行了封装(OEW/针对half2数据类型优化)

* 1. **constant mem的使用**

对于一些kernel中全程只读的数据,用cosnt和\_\_restrict\_\_修饰, 使其放在read only costant cache中

* 1. **L2的使用**

CUDA11可细粒度控制那些数据使用L2.

如果对global mem的访问难以对齐,那尽量让相邻thread访问到L2上的相关数据.

对global mem的访问会缓存到L2,也会先读L2,找不到再都global mem

* 1. **L1的使用**

\_CUPG/5.3: 只能编程控制L1和share mem各自的大小比例,因为它们在同一块mem上

???L1确定只用于texture and surface的话,是否可以把整块mem都用于share mem

* 1. **Kernel中使用Share Mem**

见\_CUBP/Memory Optimization/Shared Memory

理解share mem的地址分配与bank conflict,关注注释(\_CUBP/9.2.3.1)

cc5以上每bank每clock cycle的bandwidth为32bit,但可设置为64bit以支持双精度,或设为其它???

细看并补充每个例程的相关注释.

例程Using shared memory to improve the global memory load efficiency in matrix multiplication说了一个warp内计算要访问的数据可先用share mem保存; 此时要保证global mem的访问是对齐(coalesced access)的,share mem的访问是无bank conflic的,同时充分利用带宽而不浪费,尽量用syncwarp.

但提到的cache evict机制和规则是啥???

例程Improvement by reading additional data into shared memory说了何时以及如何去做一个基于block的share mem加载.

例程An optimized handling of strided accesses using coalesced reads from global memory:

对于global mem的访问指针最好是随着threadIdx.x连续移动的,否则会出现strided access

同时说了如何发现和解决share mem访问的bank conflic

\_CUBP/9.2.3.4

CUDA11或以上可以aync访问share mem,此时会跳过L1,L2,速度更快.

无论sync还是async访问,存取的数据类型最好8/16byte,最好是build-in vector类型(比如int4,float4),这样传输内容更压缩,效率更高.但没说为何???

例程中sync循环访问global和share mem时,copy\_count(循环次数)最好是4的倍数.但没说为何???

(SX1)在使用ShareMem时的bank conflic处理,该处理方法与(\_CUBP)提到的也不同,???需要进一步看源码

不要用太多register,防止溢出到local mem

host计算与kernel计算是异步的

见\_CUPG/CUDA Runtime/Concurrent Execution between Host and Device

使用Async结尾的内存传输相关函数

见\_CUPG/CUDA Runtime/Overlap of Data Transfer and Kernel Execution 和 Concurrent Data Transfers

见\_CUBP/Memory Optimizations/Data Transfer Between Host and Device

例程Staged concurrent copy and execute和Concurrent copy and execute

kernel并发计算

见\_CUPG/CUDA Runtime/Concurrent Kernel Execution 在计算资源和前提条件允许的情况下,kernel是并发执行的

\_CUPG/Table 15表示v100也就是compute7的kernel并发数最大为128

1. **LS**

(基本参考\_LS)

* 1. **整体优化**

attn相关的QKV操作用的都是cublas,有论文(\_p1)尝试用cutblas做更多融合,但效果不如cublas.

同时\_p1也尝试过cuDNN的self attn,效果还不如tf,pt

把所有操作都放到cpp层,并使用graph,去掉中间没层对pytorch的调用以及input,mask等内存复制,提升10%左右

使用mult stream反而会比ls低,使用权重multi stream提升降低到5%,可见occupancy已经很高或遇到了并行度以外的其它瓶颈.

参考参数:

batchSz=16,tknLen=512,embSz=768,mediSz(MLP中间的hidSz)=3072

通过nvs知道如下耗时kernel:

ker\_attn\_softmax\_bw~ls\_dropout\_act\_bias\_bwd\_kernel~5%

ker\_attn\_softmax~ls\_dropout\_kernel~ls\_dropout\_bwd\_kernel~4%

ls\_dropout\_act\_bias\_kernel~2%

然后一个个优化

‘SoftMax’优化了ker\_attn\_softmax\_bw和ls\_dropout\_bwd\_kernel, 提升不到2%

‘Dropout’优化了ls\_dropout\_act\_bias\_bwd\_kernel, 以及ln的w和b的bwd, 提升2%左右

brt emb的实现,以及每个thread中减少锁(不用F4),减少Stall Barrier,提升2%左右

共最多提升15%

* 1. **SoftMax**

ker\_attn\_softmax\_bw占5% :

看roofline和SQL:

nvc的roofline上可见带宽已占满, SOL Mem也93%了, 但计算访存比并不理想(才0.36), SQL SM 7.7%

方向应该是优化访存带宽的同时, 提升计算访存比, 为每个加载到thread的数据进行更多的计算

看nvc 的mem:

L1命中率6%,不及理论最大的20%,见下面的’数据吞吐优化’

看nvc的warp state:

stall LG throttle占大部分: 可见访问dev mem过于频繁

stall long scoreboard次之: 访问L1/TEX较慢, 很可能因为命中率低造成

见’数据吞吐优化’,加大L1占比,在变长tkn中有提升

看nvc的sass:

stall的确发生在LDG(glb mem IO)指令上(sample data最多,指令数目也多)

见’数据吞吐优化’,加大L1占比,在变长tkn中有提升

数据吞吐优化:

16batchSz,512tkn前提下,总数据IO是batchSz\*heads\*tknLen\*tknLen=12582912F4,每thread处理一个tkn的1/16sftMx,读grad和inp各4F4,写grad4F4共12F4,一个warp的IO不到7KB,一个blk约28KB,而v100的L1+share men是128K, 应该是够的, 且最大命中率大概应是28/128~20%. 而且blkDim.x=32,所以对lbl mem的读取是对齐的,blkDim.y=4,32\*4=128,对cache line来说也是对齐的.数据读取上应该影响不大.

数据吞吐速度已经成为整体计算瓶颈, ker\_attn\_softmax\_bw要处理的数据量是ker\_layer\_norm的10倍左右,耗时也是10倍左右,虽然后者的计算还复杂些,计算访存比略高,且包含了几次sync blk

一般会用register,share mem,L1,L2,减少对glb mem的IO,减少带宽占用

由于这里不需要跨thread计算,没有跨warp的数据复用, 单个数加载到thread后经过计算和warp shuf就可以得到结果,所以用register来保存数据就足够

使用F4提高IO效率, 提升1%左右:

通过nvc的sass看,LDG指令虽然每个指令的stall差不多,但指令数目减少大半

同时由于用不到share,所以把L1 cache的所有缓存都给到L1,在tkn变长时提升不到1%. (CC7可调L1占比)

此时nvc的L1命中率提升到10%左右

与上一步ls\_dropout\_bwd\_kernel融合后,对mask, sftMx\_outputed用const,\_\_restrict\_\_,\_\_ldg()操作,使其尽量被放到read-only cache中(也就是L1)

sass对应指令从LDG.E.SYS变为LDG.E.CONSTANT.SYS,可见编译器对这些数据的IO已经尽量放在read-only上了

L1命中率10%多

这使得’ ls\_dropout\_bwd\_kernel融合’能提升1%多

计算访存比:

考虑到FFMA约1:1,nv的blog说8:1较好,但经验上是10+:1左右较好,因为v100计算单元太多,可并行计算

与上一步ls\_dropout\_bwd\_kernel融合,减少glb mem的IO,提高计算访存比到(8/3+3/3)=11/6~2/1,作用不大,但减少了IO,提升1%多(还有上面提到的)

提升2%左右

* 1. **Dropout**

ls\_dropout\_act\_bias\_bwd\_kernel 占 5%

roofline看并没完全达到内存带宽与算力极限

occupancy看:

blk粒度小点的话,occupancy可以更高

blkDim.x=y=32,导致划分出的blk比较少,一个SM分1个多点,达不到2-3个,导致IO得不到掩盖

但由于使用的算法,blkDim只能32\*32,

这个算法可以使得都是每个blk负责32个hidSz元素的情况下,比原始的每个mediSz循环batchTkn次汇总bias grad的做法多了31个warp,也就是每个warp占的资源和时间都少些,occupancy比原始做法高些,但在glb mem访问相似的同时多了share的使用和blk sync

前后都是调用cublas,无法进一步融合

进一步优化:

使用F4

使用const, \_\_restrict\_\_, \_\_ldg()尽量把IO放到read-only cache

把ln中对w和b的bwd,分得更细blk,并用multi stream中的低权重stream去塞满,没有满的那些occupancy

之前的multi stream没把blk所占的资源拆得更细,所以未必能见缝插针地占算力

提升2%多,相当于那两个ln的bwd不用再串行做了

* 1. **LN**

[[LightSeq: Transformer高性能加速库](https://app.yinxiang.com/Home.action?login=true#n=93cb0d0b-5fc0-4ea8-904d-3fde6a99b955&s=s30&ses=3&sh=5&sds=5&x=lightseq&)]

方差用的是: E(x^2)-E(x)^2, 而不是原LN中求高斯分布的方差公式, 前者更像是求uniform分布的方差, 减少了跨warp的归约操作和计算量

sum(x^2)和sum(x)可在一次block reduce<sum>中做

而原LN需要求sum(x)以求均值,然后再求sum(x-μ)来求方差,需要两次block reduce<sum>

使用float4

总数据IO是batchTkn\*embSz=1572864F4,每thread是embSz=192F4(若threadIdx=0则多写一个F到vars)

计算访存比,4:1(threadIdx=0时20+:6),nv的blog说8:1较好,但经验上是10+:1左右较好,因为v100计算单元太多

个人觉得,

重点是计算量与数据加载量的比值, 以及是否有其它计算中的warp来掩盖数据加载时间, 所以一个warp对应一个tokn的LN计算可避免block sync的使用, 同时有可能计算量也可以掩盖数据加载时间.而且此时是warp之间掩盖,而不是blk之间掩盖,可能掩盖的可能性会高些.

从forward代码看,如果每个warp计算一个tokn的ln,需要一次warp sync,计算访存比大概8:1左右,应该可以掩盖吧,但如果batch\_token\_num不大的话可能并行度和occupancy不高

* 1. **根据不同的序列长度调用不同kernel**

ds只支持句子长度为8倍数

ds支持2次幂的emb sz

* 1. **emb**

对于词嵌入层，LightSeq 将词表查找与放缩、位置向量融合以及 dropout 操作都写成了一个核函数。

* 1. **使用Float4**

采用float4,指令数减少,增大数据吞吐,减少延迟,数据传输效率高.但要处理末尾剩余的元素.

[CUDA Pro Tip: Increase Performance with Vectorized Memory Access | NVIDIA Developer Blog](https://developer.nvidia.com/blog/cuda-pro-tip-increase-performance-with-vectorized-memory-access/)

优点:一个指令传输更多数据;一次传输的有效数据更多,其它元数据占比变少;传输同样数据需要的指令更少;

缺点:由于每个thread能处理的数据多了,所以并发度降低.

* 1. **ptx调tensor core**

TODO:

需要什么样的shape, shape方面的约束,

half编程

<https://forums.developer.nvidia.com/t/how-to-define-a-vector-type-half8/68341>

没half8,用ptx以unit4加载,才能编译成LDG.E.128加大吞吐. Uint4和float4都支持这种编译.

通过half\_t的玩法,用unit4,计算时强转为half2,可用HFMA2/HADD2指令(优于cutls)

无padding适配,数据处理时尽量符合tc约束

Smem->Reg->Sum->Epilogue

嵌入式汇编

切断自动编译,做pipeline并行

<https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html#Using-Assembly-Language-with-C>

效果:

Pytorch半精度,速度提升25-35%,显存节约40-60%,精度下降不到2%.

https://www.zhihu.com/question/268914381 指出用了TC的cublas比不用的快6倍虽然理论能快12倍,但访存是瓶颈.

我们可快接近8倍,因对half的处理增大了访存吞吐…

* 1. **Pipeline并行**

双buff

* 1. **Gemm融合其它**
  2. **约束**

大量优化的使用产生了一些约束:

因快速取模所以emb sz要为2次幂(约束已去除),

因使用float4, tkn num???, seq\_len(softmax用到)和tkn\_emb(LN用到)要被4整除

要让cublas使用TensorCore,???要被4或16整除(???很多tensorCore例子都基于4x4矩阵,但也有文档说内部实现基于16x16x16矩阵???)

inference的ker\_refresh\_cache()要求batchSz和beamSz必须是2次幂

* 1. **其它:**

用召回后计算的思路处理大规模beam search

内存池,按最大有可能用到的量预先占好内存,使用时从池中分配.因为cublas很容易打满GPU算力和带宽,内存多了也没用

参数拉平和连续化成一个参数, 降低参数更新期间拷贝、同步、计算的次数。实验分析发现，进行了这项优化后，优化器性能提升了 40%-50%。

算子融合

TODO:

主干与分支计算

没用到share mem的kernel加大L1分量

nvidia出品的Megatron也有做layernorm，mask+softmax，triangle（取上三角）+mask+softmax的融合: https://github.com/NVIDIA/Megatron-LM/tree/main/megatron/fused\_kernels

貌似LS的float4没有处理末尾剩余的元素

* 1. **tmp**

推理时权重的提前转置,适配cublas

decod推理时是没step有beamSz个tkn,同时decode这些tkn

1. **DL性能优化**
   1. **Elem wise 操作**

oneflow做了以下优化:(OEW)

针对elem wise情况设置gridSz和blkSz: 参考<调用流程/kernel启动的配置>, 同时在GetNumBlock()中有自己的设置策略. 以保证’ 数据量较小的情况下，不会启动过多的线程块。在数据量较大的情况下，尽可能将线程块数目设置为数量足够多的整数个 wave，以保证 GPU 实际利用率够高。’(OEW ???但他这样做貌似不大妥,blkSz固定256?)

调用链上大量使用向量化增加吞吐,在做向量化转换的同时做了计算操作.

向量化上支持除了CUDA内置类型以外的更多数据类型, 同时也处理了无法整除的剩余数据.参考<调用流程/向量化读GlbMem数据>

对half2操作的易用与兼容上的优化参考<调用流程/向量化读GlbMem数据>

对一元,二元,多元elem wise操作的兼容,与更容易扩展到不同的elem wise操作(OEW)

* 1. **Permute/Transpose**

以下来自(PT)

尽量用int32\_t表示idx,若元素太多才用int64(PT)

大小为1的维度可以直接去除 (PT)

连续排列的维度可以合并成一个维度(PT中有例子)

更大的访问粒度:(使用向量化数据类型) (PT)

CUDA支持的访问粒度为1B，2B，4B，8B，16B，粒度越大性能越好

最后一个维度是作为整体来移动的，即permutation[n-1]==x.dims[n-1]，且大小是新访问粒度的倍数 (PT中有例子,但不大懂)

保证数据指针满足新访问粒度的对齐要求(???何谓新访问粒度?具体怎么对齐)

使用share mem处理permute涉及不连续元素的情况:(PT) ???没懂

但实际场景中还存在矩阵转置的情况，此时无法应用第三条增大访问粒度的优化操作，并且不满足访存合并要求，导致性能不佳(PT). 不能用上述第三条. 未合并的访存导致实际读取数据量远大于写入数据量(7-8倍)。(???怎么用share mem解决的没看懂,要结合[An Efficient Matrix Transpose in CUDA](https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/) 看下)

在使用share mem做permute时, 针对half数据类型，且转置维度均能被2整除的情况下，我们可以进一步利用half2来合并 (PT)

Shared Memory的一个bank宽度为4B, 那么加载到Shared Memory的时候，我们可以将两个half数据合并为half2类型进行加载。

取列元素的时候，因为元素分布在两个不同的bank上，不能合并成half2直接取。需要构造一个临时的half2对象，分别将两个bank上的half元素存储到该half2对象，再写回到Global Memory里。(PT) ???没看懂

‘对于像Transpose这样的数据重排列的Kernel，我们并不关心里面数据具体的值，而是仅涉及到数据的 搬运，此时将小数据类型（如2字节的half类型）合并成一个大的数据类型（如8字节的int64），可以使 得计算量减少到原来的1/4。在通常的Transpose Kernel实现中，坐标变换涉及到整数除法，速度很慢，性能瓶颈主要来自于整数除法，而通过合并数据类型，可以减少坐标变换的次数，从而加速Kernel 的执行时间’(OFO)

???

看oneflow: GetMovementSize函数来确定访问粒度的大小

Permute的带宽比原生Copy还高一点，是因为Copy Kernel里没有做unroll指令间并行优化，而Permute Kernel内部做了相关优化(???具体怎么做)

基于int32，int64类型的快速除法, 可用于坐标转换, 但具体怎么做

* 1. **LN:**

oneflow做得优化(LN1)

方差计算: 使用Welford 算法在N较少也可保证精度,同时速度更快. 而naive算法在N小时精度插,且分子有可能为0. (LN1)

但(LN1)中提到的naive算法貌似在LS中使用也没啥问题, 估计是因为tkn\_emb的维度是N,而N也足够大,使得naive算法的精度还可以.

向量化访存: 参考(LN1)<调用流程/向量化读GlbMem数据>和(OEW)

使用寄存器和ShareMem暂存中间数据,减少访问GlbMem次数.

寄存器资源和 Shared memory 资源都是有限的，如果 num\_cols 过大，就会超出资源的使用限制，因此我们针对不同 num\_cols 采用不同的实现

num\_cols<=1024时, 以 Warp 为单位处理一行或两行，将输入 x 存储到寄存器中。用warp级别原语.

对warp内的thread又进行了分group,若num\_cols太小,groupSz会小些,进而一个warp可以处理多行.但分group没用CUDA的cooperative groups API.

warp内分group的sz由num\_cols, thread\_group\_width决定,进而决定rows\_per\_access即每group处理的行数.

对’ cols 不是 warp\_size 的整数倍’的情况做了padding处理

使用warpAllReduce <GPU编程CUDA/编程算法/AllReduce>

num\_cols>1024, 每blk处理一行, 通过warpAllReduce->ShareMem->warpAllReduce的方式做BlkAllReduce<GPU编程CUDA/编程算法/AllReduce>

采用 cudaOccupancyMaxActiveBlocksPerMultiprocessor 函数判断当前硬件资源条件下 Kernel 是否能成功启动

由于需要blk内同步以及使用ShareMem,所以blkSz的设置有特定方法.<GPU编程CUDA/编程模型/block>

num\_cols再大些,就只能直接读写GlbMem了:

也就是cudaOccupancyMaxActiveBlocksPerMultiprocessor返回0说明kernel对ShareMem和寄存器的要求无法满足时,就只能读写GlbMem

一个 Block 处理一行的元素，不使用 Shared Memory，重复读输入 x

和num\_cols>1024时的处理类似,只是无法用ShareMem,但由于部分数据可被Cache,所以不一定慢.

每blk负责一行的情况下: 文中’block\_size 越大，SM 中能同时并行执行的 block 数就越少，对 Cache 的需求就越少，就有更多机会命中 Cache，因此我们使用较大的 block\_size。’跟oneflow沟通后是说blk少证明行少,固然对cache的需求就少,而cache是不同blk都可以访问的,并发每blk独立占有

oneflow的LN其实支持乘gamma加beta的玩法: 在源码的LOAD和STORE的具体实现中,是可以更灵活地在数据的读入和写出点扩展不同的计算操作的,gamma和beta的操作是在此发生的.

* 1. **SoftMax**

oneflow的实现:( SX1)

与(LN1)一样,num\_cols<=1024,大于1024,无法使用ShareMem时的处理是相似的. 只是BlkAllReduce没有自己写,用的是Cub的BlockReduce(???会比LN1的实现更好?)

另外向量化与half2的处理也与(OEW)相似

与(LN1)不同的貌似就是在使用ShareMem时的bank conflic处理,该处理方法与(\_CUBP)提到的也不同,???需要进一步看源码

* 1. **Activatioin Ckpt**

虽然activation ckpt可以减少做后向传播求导时的计算量,但也增加了访存压力,所以为了平衡访存量与计算量,有时候宁愿重新计算而不在forward时保存activation ckpt.

(OFO)中也提到一种用bitset减少relu时activation ckpt的大小的方法,但仅使用在求导时可以对activation ckpt中为0的不进行计算或数值为0,1的情况. ???而且我也没看懂bitset的用途和如何做的???

我的dropout融合其实也是减少了没必要的activation ckpt,用上一个的算子的activation ckpt来携带哪个神经元被drop掉的信息.

* 1. **通用:**

[Increase Performance with Vectorized Memory Access](https://app.yinxiang.com/shard/s30/nl/5421460/65dbdaa5-66cf-4309-b60a-e4f983e6fe51) 提到使用vectorized(向量化)内存操作可以减少总的指令数，减少延迟，提高带宽利用率, 增加数据吞吐与更好利用cache,但oneflow的LN和softmax代码中找不到如何使用float4和float2的代码.

如果用warpReduce,当warp\_size变小时,warpReduce的轮数也变少. (LN1)

padding 代表当前是否做了 padding，若 cols 不是 warp\_size 的整数倍，我们会把它padding 到最近的整数倍处理。

\*\*\*高性能padding要学习,对gpu性能影响大.当然,如果能控制输入,使得不用pad就更好. (LN1)(SX1)

此外，由于 Block 内线程要做同步，当 SM 中正在调度执行的一个 Block 到达同步点时，SM 内可执行 Warp 逐渐减少，若同时执行的 Block 只有一个，则 SM 中可同时执行的 Warp 会在此时逐渐降成0，会导致计算资源空闲，造成浪费，若此时同时有其他 Block 在执行，则在一个 Block 到达同步点时仍然有其他 Block 可以执行。(LN1)

循环展开:

‘#pragma unroll’在for的条件判断复杂时,不一定能展开循环,需要简化循环条件. (PT中有例子)

算子融合

将element-wise的Kernel跟上一个计算Kernel合并. 减少显存访问,加大访存计算比

借助shared memory合并带有Reduce计算的Kernel

--(OFO)—start--

GPU最主要提供的是两种资源：计算资源和显存带宽资源。如果我们能将这两种资源充分利用，且对资源的需求无法再降低，那么性能就优化到了极限，执行时间就会最短。

在大多数情况下，深度学习训练中的GPU计算资源是被充分利用的，因为大多数对计算需求很高的算子都使用cuDNN或者cublas提供的高性能接口。而OneFlow单卡性能最快的秘诀就是尽可能充分利用了显存带宽资源。

如何评估一个CUDA Kernel是否充分利用了显存带宽资源？

对于显存带宽资源来说，“充分利用“指的是Kernel的有效显存读写带宽达到了设备显存带宽的上限，其中设备显存带宽可以通过执行 cuda中的的bandwidthTest得到。Kernel的有效显存带宽通过Kernel读写数据量和Kernel执行时间进行评估，读写数据量 / 执行时间 = 当前Kernel的有效显存带宽。

一:减少全局内存的访问:

(OEW)借助算子融合,减少kernel启动次数和访存量. 其实算子融合做不好照样访存量大,具体看怎么做.

(SX1)(LN1)基于ShareMem或寄存器做Reduce

‘减少实际需要的访存大小’:???里面在relu做前后向时用bitset取代activation ckpt的做法没看懂,要看下源码

二: 确保全局内存访问合并

在<GPU架构/Memory>和<调用流程>中有提内存访问合并,但By和bi有点混乱,需要结合这篇(OFO)整理一下

‘注意的是，如果一个warp中的32个thread访问连续的32字节的内存数据时，会触发内存访问合并，此时仅需要一次内存事务（即一条指令）就能完成访存操作。所以“确保全局内存访问合并”这个准则就是希望一个warp内对global memory的读写操作合并到尽可能少的内存事务中，使得内存带宽被尽可能充分利用，最终Kernel的执行速度更快。对于capability 6.0及以上版本的GPU，当warp中内存访问地址对齐32字节的整数倍且thread间访问内存连续时，内存访问效率最高。’(OFO)

三: 优化kernel计算量

访存计算比要何时,取一次数据需要足够的计算量取掩盖. 我想到的. 旷视的博客也有写.

‘对于像Transpose这样的数据重排列的Kernel，我们并不关心里面数据具体的值，而是仅涉及到数据的 搬运，此时将小数据类型（如2字节的half类型）合并成一个大的数据类型（如8字节的int64），可以使 得计算量减少到原来的1/4。在通常的Transpose Kernel实现中，坐标变换涉及到整数除法，速度很慢，性能瓶颈主要来自于整数除法，而通过合并数据类型，可以减少坐标变换的次数，从而加速Kernel 的执行时间’(但这在(PT)中貌似没提)

‘根据Tensor的Shape优化计算量’??? 没看懂

当Tensor中的元素个数较少时，使用int32类型表示坐标会比int64类型表示坐标计算更快。

在坐标变换时，通常会遇到先除后取模的情形，此时如果把取模运算改写成乘法运算可以提升性能。如下所示：

row = offset / row\_size;

col = offset - row \* row\_size;

四：延迟隐藏

TODO

--end--

1. **NCCL**

[Collective Operations — NCCL 2.10.3 英伟达分布式通信库集合操作原语](https://app.yinxiang.com/shard/s30/nl/5421460/cf304451-5bb0-4ec3-88a7-b11882525f3b)

[A Gentle Introduction to Multi GPU and Multi Node Distributed Training (lambdalabs.com)](https://lambdalabs.com/blog/introduction-multi-gpu-multi-node-distributed-training-nccl-2-0/)

???

multi-node nccl without mpi: [Multiple node running nccl tests failed. · Issue #92 · NVIDIA/nccl-tests (github.com)](https://github.com/NVIDIA/nccl-tests/issues/92)

1. **推理**

??? TensorRT is an SDK for high-performance deep learning inference. It includes a deep learning inference optimizer and runtime that delivers low latency and high-throughput for deep learning inference applications. The core of NVIDIA TensorRT is a C++ library that facilitates high-performance inference on NVIDIA GPUs. TensorRT takes a trained network, which consists of a network definition and a set of trained parameters, and produces a highly optimized runtime engine which performs inference for that network.

TensorRT黑客松比赛: 冠亚军

[trt-samples-for-hackathon-cn/hackathon at master · NVIDIA/trt-samples-for-hackathon-cn (github.com)](https://github.com/NVIDIA/trt-samples-for-hackathon-cn/tree/master/hackathon)

[Archermmt/yolov3\_dcn\_nv\_hackthon2021 (github.com)](https://github.com/Archermmt/yolov3_dcn_nv_hackthon2021), [Ice\_boxing/CornerNet-Lite-TRT - Gitee.com](https://gitee.com/crouchggj/corner-net-lite-trt/tree/TRT2021)

??? Triton Inference Server (formerly TensorRT Inference Server) provides a cloud inferencing solution optimized for NVIDIA GPUs. The server provides an inference service via an HTTP or GRPC endpoint, allowing remote clients to request inferencing for any model being managed by the server.

??? Nvidia多进程服务 [CUDA 多进程服务工具MPS为啥这么有用？](https://cloud.tencent.com/developer/article/1531537)

香侬科技也搞了个类似Triton的 [ShannonAI/service-streamer: Boosting your Web Services of Deep Learning Applications. (github.com)](https://github.com/ShannonAI/service-streamer)

1. **参考**

<CUDA\_C\_Programming\_Guide\_11.4> 简称\_CUPG

<CUDA\_C\_Best\_Practices\_Guide.11.4> 简称\_CUBP

<KERNEL PROFILING GUIDE User Manual v2021.2.0> 简称\_KPG

<NVIDIA TESLA V100 GPU volta ARCHITECTURE> 简称\_V100

[[支持Transformer全流程训练加速，最高加速3倍！字节跳动LightSeq上新](https://app.yinxiang.com/shard/s30/nl/5421460/9cacb381-1980-46eb-a5fc-dc81cddce42b)]和开源代码 简称 \_LS

< MLSys-2021-data-movement-is-all-you-need-a-case-study-on-optimizing-transformers-Paper> 简称\_p1

[[Fast Dynamic Indexing of Private Arrays in CUDA](https://app.yinxiang.com/Home.action?login=true#n=94d9e4d6-86a8-43e6-9dc7-e441f9f8ca3e&s=s30&ses=4&sh=2&sds=5&)] 简称\_b1

[CUDA优化之LayerNorm性能优化实践--oneflow性能优化分享](https://app.yinxiang.com/shard/s30/nl/5421460/d4521ce5-dae5-45c9-8176-f289be78025f) 简称LN1

[如何实现一个高效的Softmax CUDA kernel？——OneFlow 性能优化分享](https://app.yinxiang.com/shard/s30/nl/5421460/22398148-f538-43c5-b985-9ca5b04c019e) 简称SX1

[开源100天，OneFlow送上“百天大礼包”：深度学习框架如何进行性能优化？](https://app.yinxiang.com/shard/s30/nl/5421460/92ba4b1b-0b70-44ec-ac01-61f2a9b8bce0) OFO

[如何设置CUDA Kernel中的grid\_size和block\_size？](https://app.yinxiang.com/shard/s30/nl/5421460/3f3911ce-096c-43cf-8f78-64de70f84757) GBSZ

[如何实现GPU上比PyTorch快6倍的Permute/Transpose算子？](https://app.yinxiang.com/shard/s30/nl/5421460/d0a2548f-6cc7-42c3-bf6f-3f3edde19f14) (PT)

[高效、易用、可拓展我全都要：OneFlow CUDA Elementwise模板库的设计优化思路](https://app.yinxiang.com/shard/s30/nl/5421460/e0824074-f613-472a-853b-3bdad902e35a) OEW

[CUDA Pro Tip: Increase Performance with Vectorized Memory Access](https://app.yinxiang.com/shard/s30/nl/5421460/65dbdaa5-66cf-4309-b60a-e4f983e6fe51) NVT

1. **TODO:**

[Deep Learning Documentation (nvidia.com)](https://docs.nvidia.com/deeplearning/)

\*\*\*

thread不要用太多register和share mem,一是两者数量有限,二是会影响并发度.

kernel占的资源不要太多,这样才能在一个SM内并发多个block,使得其中一些block被\_\_syncthreads时有其他block可执行

这篇文章的B和b, Byte和bit有点混乱,需要整理. 一般B代表Byte, b代表bit, 1B=8b (跟GlbMem,L1相关)

L1是每次启动blk就占一块么,不同blk也无法共享L1中的同一数据?

1. **TODO**
   1. **大体思路**
      1. **temp one**
      2. **temp two**
         1. **temp two one**
            1. **temp two one one**
            2. **temp two one two**

**temp two one two one**

* + - 1. **temp tow two**
         1. **temp two two one**
      2. **temp two three**
  1. **目前状态**
  2. **实际计划**