GPU编程高级优化技术杂谈

前言

数年前，我初入编程领域,一开始根据兴趣GPGPU这个方向,
那是CUDA和OpenCL还未出现，那是底层汇编找色器的时代，而我当时正是通过OpenGL使用GPU汇编指令，Cg以及GLSL编写着色器来进行GPU通用计算,直至现在一直从事基于GPU和CPU高性能异构计算的工作。
数年前以网名cyrosly经常混迹于CSDN
CUDA论坛和CUDA计算QQ群，讨论各种相关技术或是对一些网友的问题答疑解惑。现在想来那段时期，有过狂妄，有过激情，更结识了友情。是我的好友郑经维正是在学习CUDA技术的过程中结识的，当时虽只一面之缘，却成为了这个圈子中最要好的朋友,也正是因为他的劝说才有了我写此书的决定，或许一本对上感觉的书对于读者的意义远大于在论坛上回答成千上百个问题。本来经纬是想让我尽可能出版的，但是由于工作的原因，没有多余的精力继续写下去，因此打算把还远未未完成的残稿贡献出来。

本书的目的跳过众多相关书籍频繁重复的内容通过几个有趣的实例直接介绍GPU编程中的高级优化技术，读者可从本书中一窥诸如cublas,cufft那些高性能库的大概面貌和其中所使用的主要优化技术。当然,即使是初学者,也可以通过本书达到技术上跳跃式的升级，而作者也信奉一个观念：一看就懂的书不是好书，因为这代表了读者最终可以从中获取的信息量太少抑或是自己潜意识里早已知晓但并未显现，并可能引发部分读者的思考：是否物有所值。本书内容绝不雷同，力求精简，节奏很快，希望读者可以通过分析本书中的代码找到开发高质量GPU程序的感觉。作者本人没有写书的经验，，甚至自认不太擅长摆弄文字，所以本书未必是一本好书，但绝对有其独特之处，如果读者可从中或多或少的学到些其它相关书籍中没有见过的内容，那么也不枉此书了。写作过程历时约一个半月，由于写作仓促，因此不免有疏漏之处,若有发现,可联系作者更正。

作者的联系方式：

QQ : 295553381

微信 : 13710058492

目录

第一章 设备微架构

前言

第一章我们介绍CUDA设备和GCN设备的微架构做。对设备微架构的了解可以在深度优化时提供理论依据和方向指导，对微架构方面细节的掌握有时甚至是帮助某些应用达到最优性能必须要的。当然，对底层架构细节的了解并不是必须的，若读者对这些内容没有兴趣，可以跳过本章。

1.0 CUDA设备微架构

kepler架构包含了数组SMX,每个SMX有以下功能单元构成：

1 指令预取单元和微指令缓冲区

2 4个warp调度器,每个warp调度器对应两个指令分派单元

3 192个CUDA Core和64或8个双精度计算单元

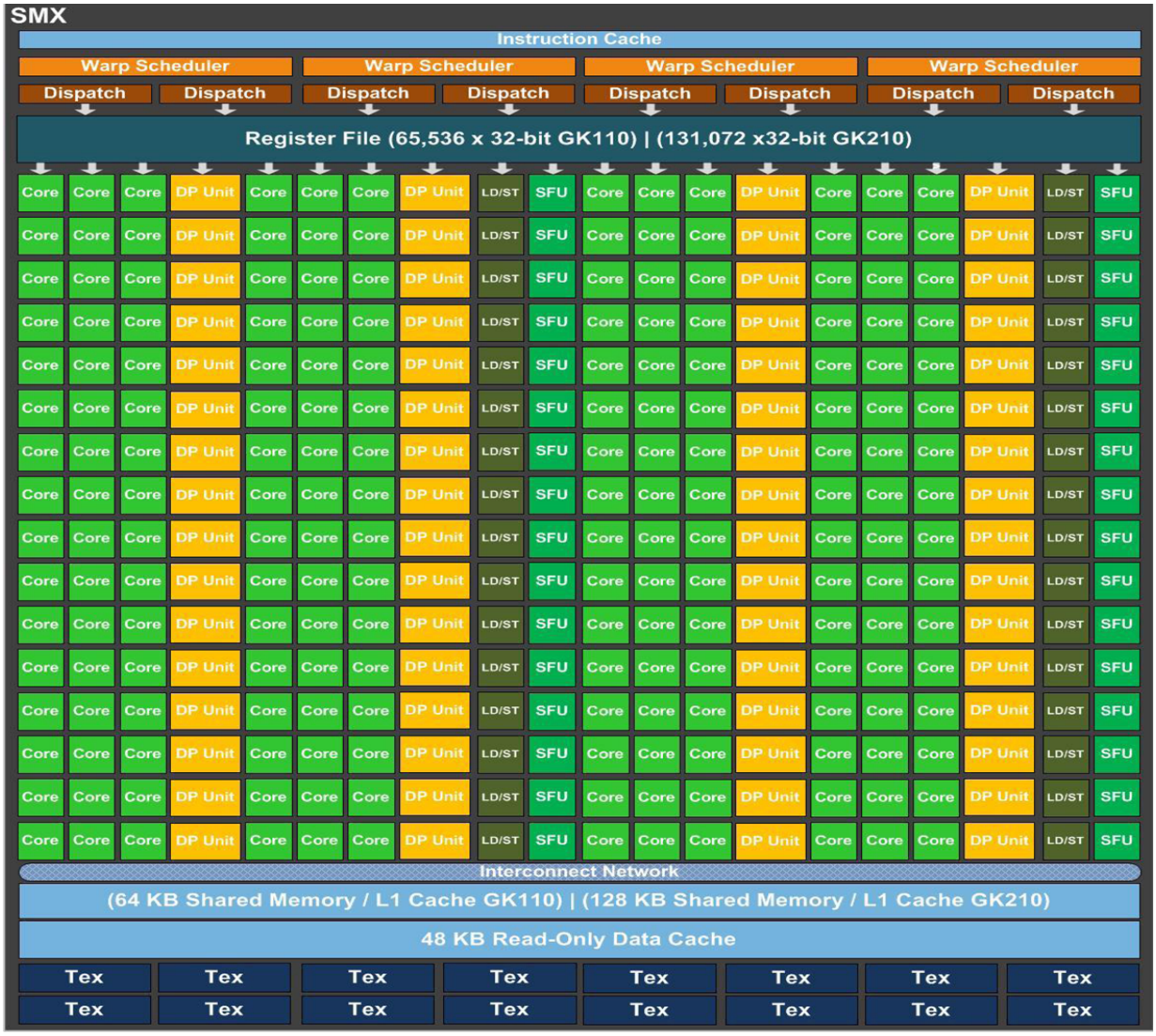
4 32个超越函数计算单元(SFU)

5 分支逻辑控制单元

6 32个LD/ST存储器访问单元

7 片上缓存结构，包括共享内存，L1缓存，纹理缓存，常量内存以及只读缓存，不同的设备大小可能不同

kepler设备SMX微架构图



各种指令的在不同的功能单元上执行，大致可分为四类：简单计算指令，复杂计算指令，分支指令和访存指令，下面是各个单元所支持的操作（仅以kepler和maxwell设备为例）

CUDA Core：32位单精度浮点加法，乘法，积和融加运算；32位整数加法；32位数据的比较操作，最小和最大操作；32位数据的位逻辑操作（and, or, xor）;8位，16位数据和32位数据之间的转换操作。

双精度计算单元：双精度浮点加法，乘法，积和融加运算。

SFU：整数乘法,除法；单精度浮点数除法以及基本的数学函数如正弦，余弦和指数等操作；popc，clz，brev，bfe和bfi操作。

分支逻辑控制单元：分支，跳转等逻辑操作。

LD/ST单元：全局内存，共享内存，局部内存，常量内存，纹理加载等存储器访问操作。

warp vote和warp shuffle操作时在是专门的组合逻辑单元上完成的。

maxwell设备和kepler类似，但是每个SMM里包含了四组独立的warp计算单元，每个warp单元包含了一个微指令缓冲区，两个指令分派单元，1个warp调度器，32个CUDA Core，1个双精度计算单元，8个单精度SFU,4个纹理单元，16k个32位寄存器组成的寄存器文件以及24k纹

理缓存。所有四个warp单元共享一个指令缓存以及64k~96k的共享/L1缓存。虽然每个SMM中的CUDA Core数量少于SMX，但是每个计算单元具有更高的性能功耗比。相对来说，maxwell具有更优秀的单精度计算效能，但为了平衡性能功耗比，所有计算能力的maxwell设备对双精度计算的支持都十分有限，而kepler更适合那些需要双精度计算的专业领域。每三个SMX或每四个SMM组成一个GPC，所有GPC共享512k~2M的二级缓存。

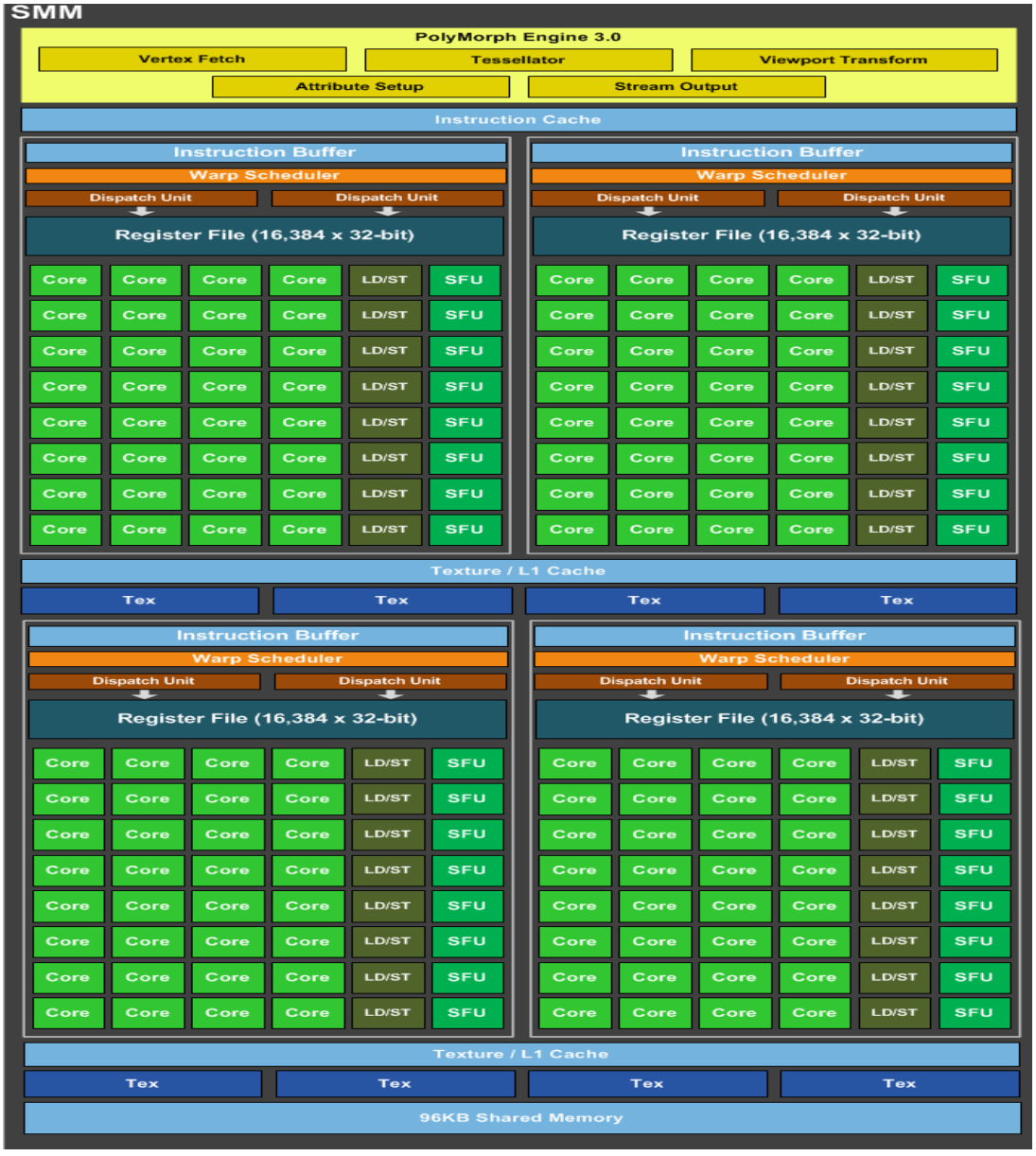
缓存小结：

对于计算能力2.x的设备，纹理管线完全独立于L1/L2缓存结构。

对于计算能力3.x的设备，纹理和L1缓存是各自独立的，但都通过L2缓存加载全局内存数据。

对于计算能力5.x的设备，共享内存，纹理缓存都属于L1缓存的一部分，所有对全局数据的访问都通过L2缓存路径，除非采用直写策略。

maxwell设备微架构图



1.0.1 寄存器文件结构

正因为寄存器索引包含在指令编码中的，且距离计算单元最近，因此其延迟比其它任何类型的内存都要低，带宽也要远高于其它类型的内存，这也正是为什么充分使用寄存器是某些应用达到峰值性能所必须且唯一的原因。在计算能力3.5+的kepler设备和maxwell设备上的每个SM中的64k个32-bit寄存器文件被划分为4个bank,每个bank中的16k个32-bit寄存器共享32个通道(lane)，每个通道32位宽。由于每个线程最多可使用255个寄存器，那么如果使用单个bank的话，每个源或目标寄存器在指令编码中需要占用8位；如果使用4个bank，那么每个目标寄存器或是源寄存器只需占用6位，剩下的位数可以留作它用。比如对与四操作数指令（比如FMA），如果采用单个bank结构的寄存器文件，那么寄存器索引在指令编码中需要占用32位；而采用4-bank结构的寄存器文件的话，寄存器索引在指令编码中只需要占用24位，节省了8位。在32位操作数的情况下，寄存器文件和4个bank的映射关系为

```
bank0 : R0, R4, R8 , R12, …

bank1 : R1, R5, R9 , R13, …

bank2 : R2, R6, R10, R14, …

bank3 : R3, R7, R11, R15, …
```

因此每个源或目标寄存器在指令编码中无需额外的2位表示寄存器的bank，否则将无任何节省。了解设备的寄存器文件结构对于性能分析以及深度优化具有至关重要的作用，因为不合理的寄存器分配会造成bank冲突（bank-conflict）,类似于共享内存的bank冲突；在没有bank冲突的情况下，指令同时从多个bank存取寄存器数据。由于源操作数最多为3个，所以寄存器的bank冲突分为1-way,2-way两种情况；目标寄存器和元寄存器之间不存在bank冲突的问题，因为取源操作数和写结果操作是在不同的时钟次序下进行的，。每增加一路冲突，就会增加一个时钟周期的延迟。比如

```
FMA    R1，R7，R3，R1
```

具有1路bank冲突（R7和R3）。如果调整寄存的分配如下

```
FMA    R1，R0，R3，R1
```

则可以消除bank冲突。

使用向量加载可以保证寄存器分配对齐到避免bank冲突的边界上，比如

```
LD.E.128 R0,  [R13]
```

R0，R1，R2，R3四个寄存器分别对应于bank0, bank1, bank2, bank3。注意到R13位于bank1中，但是这并不会和上面代码中的R1产生冲突，因为LD/ST单元会首先取出R13中的数据作为地址发送到数据总线上，然后（一定的数据传输延迟后）才将数据加载到bank0中的R0寄存器。使用向量类型时寄存器的分配需要对其到偶数编码的寄存器边界上，因此有时候使用向量类型会增加寄存器的使用数量，这有点类似于在分配对齐内存时（比如使用AVX256指令时最好将数据首地址对其到32字节边界上），实际分配的内存空间会比通常的内存分配大一点。

寄存器文件分配需要注意的另一个问题是操作数端口对齐，简单来说就是在为指令分配寄存器时，相邻的指令中位于同一个bank中的寄存器最好是具有相同的位置，例如

```
FMA  R2,  R0,  R1,  R2
FMA  R6,  R1,  R4,  R6
```

{R0,R4}/R1位于同一个bank中，但是在指令中的次序却不同，如果调整为

```
FMA  R2，  R0，  R1，  R2
FMA  R6，  R4，  R1，  R6
```

则可以对齐到操作数端口上，编译器也往往在在指令间插入MOV操作来达到操作数端口对齐的目的

```
FADD R2, R0, R1
FADD R2, R1, R2
      =>
FADD R2, R0, R1
MOV R4, R2
FADD R2, R4, R1
```

在64位操作数的情况下，寄存器和4个bank的映射关系为

```
bank0 : R[0:1], R[ 8: 9], …
bank1 : R[2:3], R[10:11],…
bank2 : R[4:5], R[12:13], …
bank3 : R[6:7], R[14:15], …
```

你可能会奇怪，为什么对于32位操作数和64位操作数寄存器和bank之间的映射关系不同。这是因为bank和寄存器之间并不是隶属关系，bank仅仅相当于数据通道，寄存器文件中的任何数据都可以从任何一个bank“进出”。可以将bank设想成地铁站的多个出入口，将每个warp中的bank对应的32个通道比作地铁的各个门，而将人比作数据。那么，人们首先从地铁站的各个出入口进站，然后再从各个地铁的门进入地铁；反向的情况则是人们从地铁的各个门离开地铁，然后从各个出入口离站。注意，“进出”的不是寄存器本身，而是寄存器中所包含的数据，寄存器文件本身是一组固定在芯片上的门电路阵列，是无法移动的；而我们在编程中所说的寄存器指的仅是物理寄存器的标示符；所以为了区分，常将两种情况分称为物理寄存器和逻辑寄存器，以下若无特别说明，指的均是逻辑寄存器。

由于每个warp中的32个线程共享四个bank,每个bank有32个32位通道，因此对于64位操作数需要连续的两个通道或是根据高低32位分两次存取数据，所以每个warp调度器为对应的warp中所有的线程发射指令需要两个时钟的延迟（或者说每个warp调度器在1个时钟周期只能为16个双精度单元发射指令，这样在没有bank冲突的情况下，4个warp调度器在单个时钟周期内可以以全吞吐率为SM内的全部64个双精度单元进行指令译码）,下面以DFMA操作说明warp中32个线程通过32路通道存取寄存器文件的操作

```
DFMA  R4，  R0，  R2，  R4
```

第一个时钟周期:

lane 0 1 2 … 31

bank0 R0, R0, R0, …, R0

bank1 R2, R2, R2, …, R2

bank2 R4, R4, R4, …, R4

第二个时钟周期:

lane 0 1 2 … 31

bank0 R1, R1, R1, …, R1

bank1 R3, R3, R3, …, R3

bank2 R5, R5, R5, …, R5

L1缓存，常量内存，共享内存以及纹理缓存的信息在很多资料中都有详细的叙述，这里不再多说，有兴趣的读者可查阅本章后面的参考资料。

1.0.2 指令流水线

首先指令预取单元从指令缓存（片上的缓存，物理存储介质和L1缓存相同）中取出多条指令进行译码，然后将译码后的指令存储在隶属warp调度器的微指令缓冲区中。指令发射单元从微指令缓冲区中取出两条指令分派给warp调度器选定的warp指令管线中，两条同类指令是顺序进入管线的，指令管线通过将一条指令的发射，取操作数时钟和另一条指令的取操作数和在功能单元上的执行时钟重叠来流水线化指令的操作，这样可以有效降低整个指令流水期间所有指令的总时钟数。流水线化在多个层次都同时存在，如不同功能单元之间的流水化，warp单元和warp单元之间的流水化以及指令和指令之间的流水化。不同功能单元之间的流水化，比如将计算单元上的操作和LD/ST单元上的操作重叠进行来隐藏数据访问的延迟。通过多个warp之间的切换让处于停顿延迟中的warp和正在执行的warp的时钟重叠隐藏warp中的计算或是访存延迟，亦即将warp流水化。单条指令在管线中的过程分为发射，执行和写回。发射阶段从寄存器文件中取操作数；在执行阶段根据指令中的操作码在相应的单元上执行特定的操作；写回阶段则将计算结果写入目标寄存器。因此，指令之间的流水化正是通过将前一条指令的执行和写回与下一条指令的发射和执行的时钟重叠来减少整个管线中所有指令的操作延迟。在指令译码阶段完成数据的存取后指令的计算结果并非立即可用，而是需要数个时钟的延迟。一般来说指令管线的深度至少不应小于指令的计算延迟，才足以保证指令流过管线后已经完成计算。理想情况下，每当一条指令从指令管线末端流出（完成计算）时，管线的前端就会流入一条新的指令，这样指令管线中的多条指令通过时钟重叠的方式有效的隐藏计算延迟。不同的指令可能具有不同的指令发射周期，亦即当发射一条指令后，需要几个时钟的停顿后才会发射下一条同类指令。我们假设一个指令管线的深度和指令的执行延迟均为4个时钟周期，每周期均可发射1条指令，那么指令流水线情况如下

1 clock I0

2 clock I1 I0

3 clock I2 I1 I0

4 clock I3 I2 I1 I0

5 clock　　I3　I2　I1　I0

6 clock　　I3　I2　I1　I0

7 clock　　I3　I2　I1　I0

8 clock　　I3　I2　I1　I0

流水线化,红色部分是从管线末端流出的以完成操作

如果没有采用流水线化，那么

1~ 5 clock I0

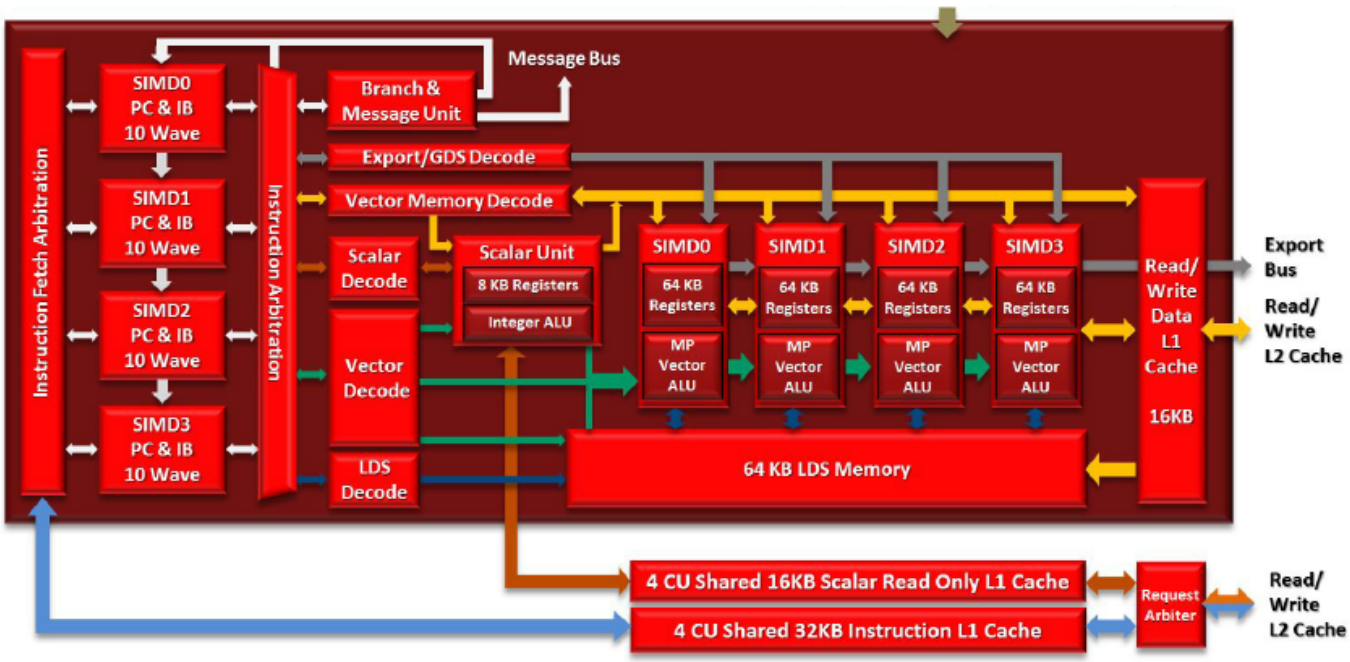
6~10 clock　　I1

11~15 clock　　I2

16~20 clock　　I3

可以看出，采用流水线化的操作完成四条指令的译码和执行只需要8个时钟周期，而不采用流水线化的操作则需要20个时钟周期。

1.1 GCN设备微架构



GCN设备由多组Compute Unit(简称CU),每组CU中包含组4个16路向量SIMD单元，一个标量单元和一个分支通信单元。每个SIMD单元配备64K的向量寄存器（寄存器粒度是32位），每个线程最多可以使用255个寄存器；向量单元主要执行简单浮点和整数计算，如32位和64位的浮点加法，浮点乘法,FMA以及32位整数加减法以及24位整数乘法和乘加操作。标量单元配备8K的标量寄存器以及一个整数计算单元，寄存器的粒度同样是32位大小。标量单元负责流控制等操作，而整数单元则负责全精度的整数运算。每个CU中的wavefront调度器每个时钟周期可以发射5条指令，包括4条向量指令（每个SIMD单元一条）和一条标量指令。多个SIMD单元可以运行同一个work

group中的不同线程，但是同一个wavefront中的64个线程智能在同一个SIMD单元上分4个时钟周期执行，即使每个work-
group中只有一个wavefront,线程调度器也不会将线程平均分配到CU中的各个SIMD单元上：GCN设备上的最小并发调度粒度是wavefront(目前是64)。关于GCN架构更详细的说明可以参考AMD的GCN架构白皮书。

1.1.1 寄存器文件结构

GCN设备上的每个Compute Unit具有256k个32bit寄存器，寄存器文件被划分成4个bank,各自分属Compute Unit内的4组向量计算单元使用。每个bank有64（不同的设备可能不同）个通道，通道宽度是4字节，一个向量计算单元中每个ALU最多可以使用4个通道（4操作数指令）。在GCN设备上没有寄存器bank冲突的问题，因为4个bank各自独立，每个向量计算单元只能使用与其对应的一个bank。当使用8字节或16字节的数据时，使用连续的2个或4个寄存器，但是寄存器的分配要满足对其要求。对于8字节数据，寄存器编号需要对齐到2的倍数；对于16字节数据，寄存器编号需要对齐到4的倍数；因此当使用向量类型时有时会增加寄存器的使用数量，这一点类似于对齐的内存分配会比普通的分配略大（比如使用AVX256指令时，通常要求存取数据的首地址按照32字节对齐）。

1.1.2 指令流水线

GCN设备上的指令流水线原理类似于CUDA设备，因此这里不再重复。

1.2 GPU设备上的条件分支

这里并不假定任何型号的设备，因此这一节所讲的同时适用于CUDA设备，GCN设备和其它宽向量设备。每个warp或wavefront拥有一组条件掩码寄存器堆栈（堆栈中包含了多个掩码寄存器，每个掩码寄存器32位（CUDA）或64位（GCN），每位对应一个线程的掩码），GCN设备上的掩码寄存器堆栈深度为6。假设我们有一个深度为为3的嵌套分支，那么warp或wavefront中线程的执行过程为

1
首先将计算得到的最外层的条件掩码写入掩码寄存器堆栈顶部的第1个寄存器(掩码堆栈指针当前指向的位置)。

2
掩码寄存器堆栈指针减1，接着将计算得到的第2层分支的条件掩码写入掩码寄存器堆栈的下一个寄存器(掩码堆栈指针当前指向的)。

3
掩码寄存器堆栈指针减1，接着将计算得到的第3层分支的条件掩码写入掩码寄存器堆栈的下一个寄存器。

4
从掩码寄存器堆栈指针指向的位置取出掩码，如果warp或wavefront中的线程号对应的位值为1，则执行第3层分支内的计算，否则忽略。

5
掩码寄存器堆栈指针加1，取出掩码寄存器中的掩码，如果warp或wavefront中的线程号对应的位值为1，则执行第2层分支内的计算，否则忽略。

6

掩码寄存器堆栈指针加1，取出掩码寄存器中的掩码，如果warp或wavefront中的线程号对应的位值为1，则执行第1层分支内的计算，否则忽略。

逻辑分支图

小结

本章我们分别介绍了CUDA设备和GCN设备的微架构，重点分析了寄存器文件结构和指令管线以及GPU设备上实现条件分支的原理。深入了解这些内容不仅可以让你知道指令在硬件中具体的行为以及为什么会这样，同时还会在优化过程中的某个角落为你说明性能无法进一步提升的障碍所在。通常理解和使用所有这些内容需要一个漫长的过程，建议经常到设备商家官方网站上下载最新的技术资料，同时一些编译原理和计算机结构方面的书籍也会帮到你，因为很多技术原理无论在CPU还是GPU上都是通用的。

参考资料

1《CUDA Programming Guide》

2《CUDA Binary Utilities》

3《NVIDIA Kepler GK110 Whitepaper》

4《NVIDIA Geforce GTX980 Whitepaper》

5《AMD GCN Architecture Whitepaper》

6《AMD Southern Island Series Instruction Set Architecture》

7《Graphics Core Next Architecture，Generation 3》

8《Compute Systems A programmer's Perspective》，Randal E.Bryant, David R.O'Hallaron

9 Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs，Junjie Lai, Andr_e Seznec

第二章 GPU矩阵乘法的高效实现

前言

本章通过介绍开发GPU上的高效矩阵乘法的各种优化方法来展示一些GPU高级优化技术，其中所用到的大多数优化技术不仅在支持CUDA的设备上有效，对于GCN设备,MIC设备和CPU上同样有效。如非特别说明，在所有版本的实现中，数据都是列序存储,。

2.0 指令级并行和数据预取

对于类似矩阵乘法这样的高计算密度类型的应用，努力提高每个线程中的ILP(instruction level parallism,指令级并行)比单纯使TLP(thread level parallism,线程级并行)更加有效,而且是达到接近峰值性能唯一的途径。ILP的真髓在于通过充分发掘单一线程内的指令流水线中的并行和并发性（如果看过第一章就会明白，单一线程内的并行存在于被双发的指令在不同功能单元上的操作，比如在LD/ST上的访存操作和计算单元上的FMA操作可以同时进行；而并发则存在于相同功能单元上的不同阶段中分时时钟序列中）以及利用寄存器的最低延迟和超高带宽的优势（在kepler和GCN设备上，寄存器的带宽大约是共享内存的6倍）。循环展开通常是提高ILP的有效方法，原因在于通过循环展开编译器可以在更长的无数据依赖的指令序列中对指令进行排序从而最大化指令吞吐率并最小化流水线中的停顿，同时减少循环索引和部分存储器地址的计算。

dgemm, loop unrolling&prefetch

```
__device__ __forceinline__ void d_rank8x8( double* C, const double* A, const double* B )
{
	double a[8], b;
	a[0]=A[0*16];
	a[1]=A[1*16];
	a[2]=A[2*16];
	a[3]=A[3*16];
	a[4]=A[4*16];
	a[5]=A[5*16];
	a[6]=A[6*16];
	a[7]=A[7*16];
#pragma unroll
	for( int i=0; i<8; ++i ){
		b=B[i*16];
		C[i*8+0]+=a[0]*b;
		C[i*8+1]+=a[1]*b;
		C[i*8+2]+=a[2]*b;
		C[i*8+3]+=a[3]*b;
		C[i*8+4]+=a[4]*b;
		C[i*8+5]+=a[5]*b;
		C[i*8+6]+=a[6]*b;
		C[i*8+7]+=a[7]*b;
	}
}
__global__ void cuk_dgemm_unroll( double* d_C, const double* d_A, const double* __restrict__ d_B, int n, int lda, int ldb, int ldc )
{
	__shared__ double smem[2048];
	double p0, p1, p2, p3, q0, q1, q2, q3, c[64]={0.f};
	int k, lane, slot;

	lane=threadIdx.x&15;
	slot=(threadIdx.y<<1)+(threadIdx.x>>4);
	d_C+=(((blockIdx.y<<7)+slot)*ldc+(blockIdx.x<<7)+lane);
	d_A+=(threadIdx.y*lda+(blockIdx.x<<7)+threadIdx.x);
	d_B+=(((blockIdx.y<<7)+((threadIdx.x&1)<<4)+(threadIdx.x>>1))*ldb+threadIdx.y);
```

```
double* St=&smem[(threadIdx.y<<7)+threadIdx.x];
double* At=&smem[lane];
double* Bt=&smem[1024+slot];

if(threadIdx.y<n)
{
        p0=d_A[0*32];
        p1=d_A[1*32];
        p2=d_A[2*32];
        p3=d_A[3*32];
        q0=d_B[0*32*ldb];
        q1=d_B[1*32*ldb];
        q2=d_B[2*32*ldb];
        q3=d_B[3*32*ldb];
}
for( k=n-8; k>=0; k-=8 )
{
        *(St+0*32)=p0;
        *(St+1*32)=p1;
        *(St+2*32)=p2;
        *(St+3*32)=p3;
        *(St+0*32)=q0;
        *(St+1*32)=q1;
        *(St+2*32)=q2;
        *(St+3*32)=q3;
        __syncthreads();
        if(threadIdx.y<k){
                d_A+=(lda<<3); d_B+=8;
                p0=d_A[0*32];
                p1=d_A[1*32];
                p2=d_A[2*32];
                p3=d_A[3*32];
                q0=d_B[0*32*ldb];
                q1=d_B[1*32*ldb];
                q2=d_B[2*32*ldb];
                q3=d_B[3*32*ldb];
        }
#pragma unroll
        for( int i=0; i<8; ++i ){
                d_rank8x8( c, &At[i*128], &Bt[i*128] );
        } __syncthreads();
}
if(k!=-8)
{
        *(St+0*32)=p0;
        *(St+1*32)=p1;
        *(St+2*32)=p2;
        *(St+3*32)=p3;
        *(St+0*32)=q0;
        *(St+1*32)=q1;
        *(St+2*32)=q2;
        *(St+3*32)=q3;
        __syncthreads();
        do{
```

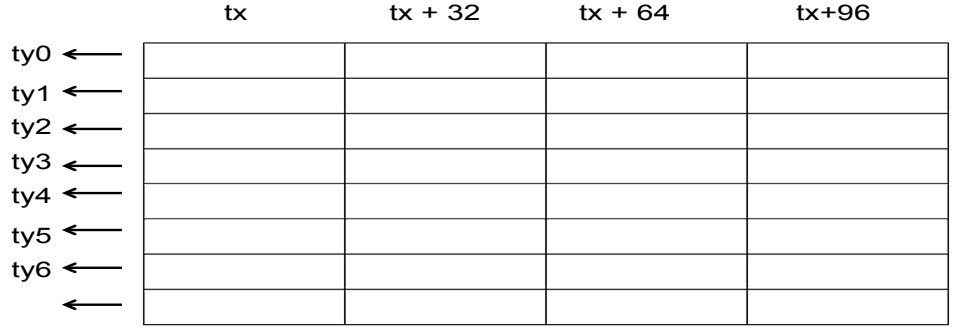
```
                d_rank8x8(c, At, Bt);
                At+=128; Bt+=128;
            }while((++k)<=0);
        }

#pragma unroll
        for( int i=0; i<64; i+=8 ){
            d_C[0*16]=c[i+0];
            d_C[1*16]=c[i+1];
            d_C[2*16]=c[i+2];
            d_C[3*16]=c[i+3];
            d_C[4*16]=c[i+4];
            d_C[5*16]=c[i+5];
            d_C[6*16]=c[i+6];
            d_C[7*16]=c[i+7];
            d_C+=(ldc<<4);
        }
}
```

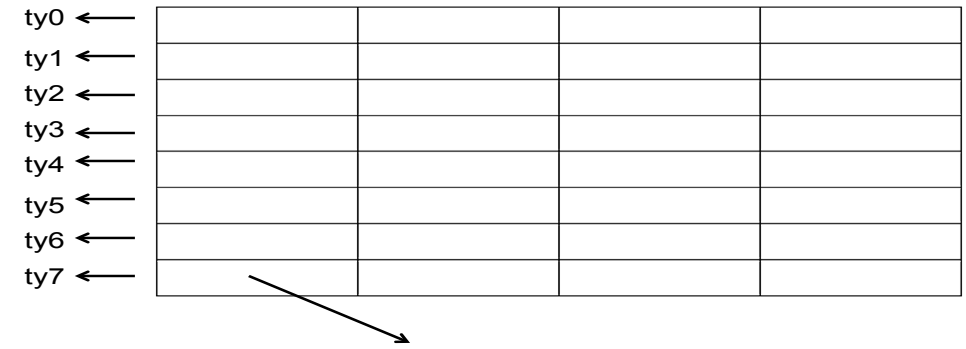
第一个版本中我们主要的优化方法是通过循环展开获得更多的ILP以及通过数据预取和计算的重叠隐藏访存延迟。（注：循环展开并非在任何情况下都能提升指令级并行，待展开的计算中还需要尽可能的无数据依赖关系，待展开的循环体中的计算也应尽可能的简单；而且过度的循环展开还可能会造成寄存器溢出或是设备资源占用率过低从而导致性能下降）。

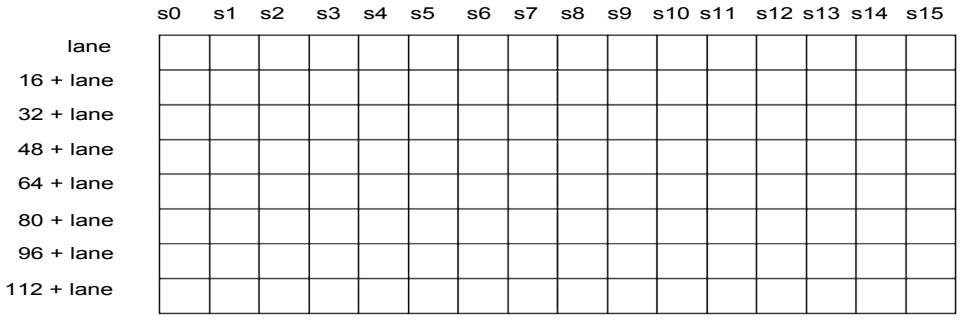
注意内核参数“const double* __restrict__ d_B”,对于计算能力3.5+的设备，编译器会自动使用LDG指令通过纹理管线加载指令加载数据，从而可以利用纹理缓存较少非合并访问的影响。对于计算能力2.x或3.x的设备,显式的使用纹理通常比普通的缓存效果更好，虽然很轻微(受限于L2缓存有限的带宽和纹理操作的长延迟)。这是因为纹理管线独立于普通的存储路径,可以提升可用的缓存带宽(纹理缓存的带宽+L1缓存的带宽)，从而更加平衡负载,并减小不同数据之间的L1缓存污染(通过纹理缓存加载的数据不经过L1缓存)。对于计算能力5.x的设备，由于统一了L1缓存和纹理缓存，因此使用线性纹理和普通L1缓存已经没有本质的区别了。

此外所选择的的网格布局也会直接影响性能和代码的可读性。根据不同的架构选择每线程计算的元素数量，太少的话不能充分利用寄存器文件高带宽低延迟的优势；太多的话会造成寄存器溢出。对于计算能力2.x和3.0的设备，每个线程计算16个元素(1x16或者4x4),线程块的大小选择为256(计算能力2.x，block布局可设置为64x4或32x8,每个block计算64x64的子矩阵)或1024(计算能力3.0，block布局可设置为128x8或32x32,每个block计算128x128的子矩阵)。对于计算能力3.5+,5.x的设备，比较好的选择是每线程计算8x8个元素(对于比较小的规模可以每个线程计算4x8个元素，每个block计算128x64大小的子矩阵)，线程块的大小设置为256(block布局可设置为16x16,32x8,128x2或者256；每个block计算128x128的子矩阵,一个SM最多可支持一个block并发(对于计算能力3.7的设备,每个SMX上的活跃block队列最多可部署2个)。

我们采用32x8的block布局，每个线程计算64个元素，每个block计算128x128的子矩阵。







2.1 双缓冲区技术

我们知道GPU网格调度器通过block内的warp(在AMD的GPU中称之为wavefront)切换隐藏内存访问和计算延迟；而通过block之间的切换隐藏块内同步延迟，但这里由于寄存器和共享内存资源的限制，一个SMX上最多只能部署一个活跃block（对于计算能力3.7的设备，由于具有双倍的共享内存和寄存器文件，所以每个SMX可以支持两个block并发），因而无法再通过增加每个SMX上可以并发的block数量来进一步提升效率。类似单个线程内的并行和并发，这时我们就需要考虑如何在单个block内寻找更多的并行和并发机制。容易看出主循环体内的并行和并发受限于块内同步，因此如果消除下一个循环步预取的数据和当前计算中所用数据的依赖关系，则第二个同步原语是可以消除的，为此我们通过双倍的共享内存作为双缓冲来消除这种影响：

dgemm,loop unrolling+double buffering

```
__global__ void cuk_dgemm_unroll_db( double* d_C, const double* d_A, const double* __restrict__ d_B, int n, int lda, int ldb, int ldc )
{
	__shared__ double smem[4096];
	…
	if(threadIdx.y<n)
	{
		smem[0*32+sidx]=d_A[0*32];
		smem[1*32+sidx]=d_A[1*32];
		smem[2*32+sidx]=d_A[2*32];
		smem[3*32+sidx]=d_A[3*32];
		smem[1024+0*32+sidx]=d_B[0*32*ldb];
		smem[1024+1*32+sidx]=d_B[1*32*ldb];
		smem[1024+2*32+sidx]=d_B[2*32*ldb];
		smem[1024+3*32+sidx]=d_B[3*32*ldb];
	}
	for( k=n-8; k>=0; k-=8 )
	{
```

```
            __syncthreads();
            if(threadIdx.y<k){
                d_A+=(lda<<3); d_B+=8;
                sidx^=0x800;
                smem[0*32+sidx]=d_A[0*32];
                smem[1*32+sidx]=d_A[1*32];
                smem[2*32+sidx]=d_A[2*32];
                smem[3*32+sidx]=d_A[3*32];
                smem[1024+0*32+sidx]=d_B[0*32*ldb];
                smem[1024+1*32+sidx]=d_B[1*32*ldb];
                smem[1024+2*32+sidx]=d_B[2*32*ldb];
                smem[1024+3*32+sidx]=d_B[3*32*ldb];
            }
        #pragma unroll
            for( int i=0; i<8; ++i ){
                d_rank8x8( c, &smem[i*128+lane], &smem[i*128+slot] );
            }
            lane^=0x800; slot^=0x800;
        }
        ……
}
```

使用双缓冲技术将当前计算的读取缓冲区和负责装载下一步数据的缓冲区相分离，从而加载下一步数据到共享内存的操作可以和当前的计算重叠执行，减少了同步开销从而减少流水线的中断。但是使用双缓冲时需要注意当前的设备上具体的效果，此外不同的计算规模效果也会不同。对于计算能力2.x的设备，由于双精度指令和存储指令不能双发(因此LD/ST指令和双精度浮点计算指令是顺序发射的，但在指令完成操作的这段延迟期间，其执行仍是重叠的)，在其上使用双缓冲区的效果相比kepler和maxwell设备会小些。

2.2 宽内存事务

我们还可以做得更好,对内层循环(d_rank8x8)进行分析可知，每个循环步共有512条FMA指令和128条共享内存加载指令，浮点计算指令所占比例(这里我们不考虑数据预取和缓冲区地址计算的相关指令,这样并不影响分析的有效性)是0.8。如果能够进一步提升这个比例，就有可能进一步提升性能。因为FMA操作的数量是固定的，因此可以通过使用128bit存储操作来缩减每个线程的内存事务的数量。首先需要对d_rank8x8进行改写：

```
#define mRank8x8(i0,i1,i2,i3){          \
        c[0*8+0]+=a[i0].x*b[i0].x;\
        c[0*8+1]+=a[i0].y*b[i0].x;\
        c[0*8+2]+=a[i1].x*b[i0].x;\
        c[0*8+3]+=a[i1].y*b[i0].x;\
        c[0*8+4]+=a[i2].x*b[i0].x;\
        c[0*8+5]+=a[i2].y*b[i0].x;\
        c[0*8+6]+=a[i3].x*b[i0].x;\
        c[0*8+7]+=a[i3].y*b[i0].x;\
        c[1*8+0]+=a[i0].x*b[i0].y;\
        c[1*8+1]+=a[i0].y*b[i0].y;\
        c[1*8+2]+=a[i1].x*b[i0].y;\
        c[1*8+3]+=a[i1].y*b[i0].y;\
        c[1*8+4]+=a[i2].x*b[i0].y;\
        c[1*8+5]+=a[i2].y*b[i0].y;\
        c[1*8+6]+=a[i3].x*b[i0].y;\
```

```
        c[1*8+7]+=a[i3].y*b[i0].y;\
        c[2*8+0]+=a[i0].x*b[i1].x;\
        c[2*8+1]+=a[i0].y*b[i1].x;\
        c[2*8+2]+=a[i1].x*b[i1].x;\
        c[2*8+3]+=a[i1].y*b[i1].x;\
        c[2*8+4]+=a[i2].x*b[i1].x;\
        c[2*8+5]+=a[i2].y*b[i1].x;\
        c[2*8+6]+=a[i3].x*b[i1].x;\
        c[2*8+7]+=a[i3].y*b[i1].x;\
        c[3*8+0]+=a[i0].x*b[i1].y;\
        c[3*8+1]+=a[i0].y*b[i1].y;\
        c[3*8+2]+=a[i1].x*b[i1].y;\
        c[3*8+3]+=a[i1].y*b[i1].y;\
        c[3*8+4]+=a[i2].x*b[i1].y;\
        c[3*8+5]+=a[i2].y*b[i1].y;\
        c[3*8+6]+=a[i3].x*b[i1].y;\
        c[3*8+7]+=a[i3].y*b[i1].y;\
        c[4*8+0]+=a[i0].x*b[i2].x;\
        c[4*8+1]+=a[i0].y*b[i2].x;\
        c[4*8+2]+=a[i1].x*b[i2].x;\
        c[4*8+3]+=a[i1].y*b[i2].x;\
        c[4*8+4]+=a[i2].x*b[i2].x;\
        c[4*8+5]+=a[i2].y*b[i2].x;\
        c[4*8+6]+=a[i3].x*b[i2].x;\
        c[4*8+7]+=a[i3].y*b[i2].x;\
        c[5*8+0]+=a[i0].x*b[i2].y;\
        c[5*8+1]+=a[i0].y*b[i2].y;\
        c[5*8+2]+=a[i1].x*b[i2].y;\
        c[5*8+3]+=a[i1].y*b[i2].y;\
        c[5*8+4]+=a[i2].x*b[i2].y;\
        c[5*8+5]+=a[i2].y*b[i2].y;\
        c[5*8+6]+=a[i3].x*b[i2].y;\
        c[5*8+7]+=a[i3].y*b[i2].y;\
        c[6*8+0]+=a[i0].x*b[i3].x;\
        c[6*8+1]+=a[i0].y*b[i3].x;\
        c[6*8+2]+=a[i1].x*b[i3].x;\
        c[6*8+3]+=a[i1].y*b[i3].x;\
        c[6*8+4]+=a[i2].x*b[i3].x;\
        c[6*8+5]+=a[i2].y*b[i3].x;\
        c[6*8+6]+=a[i3].x*b[i3].x;\
        c[6*8+7]+=a[i3].y*b[i3].x;\
        c[7*8+0]+=a[i0].x*b[i3].y;\
        c[7*8+1]+=a[i0].y*b[i3].y;\
        c[7*8+2]+=a[i1].x*b[i3].y;\
        c[7*8+3]+=a[i1].y*b[i3].y;\
        c[7*8+4]+=a[i2].x*b[i3].y;\
        c[7*8+5]+=a[i2].y*b[i3].y;\
        c[7*8+6]+=a[i3].x*b[i3].y;\
        c[7*8+7]+=a[i3].y*b[i3].y;\
}

#define mFetchSmem(i0,i1,i2,i3,k){              \
        a[i0]=smem[k*64+ 0+lane];               \
        a[i1]=smem[k*64+16+lane];               \
```

```
        a[i2]=smem[k*64+32+lane];       \
        a[i3]=smem[k*64+48+lane];       \
        b[i0]=smem[512+k*64+ 0+slot];   \
        b[i1]=smem[512+k*64+16+slot];   \
        b[i2]=smem[512+k*64+32+slot];   \
        b[i3]=smem[512+k*64+48+slot];   \
}
```

通过使用128bit存储器操作，我们将共享内存加载指令数目减少了一半，现在只有64条共享内存加载指令，浮点计算指令所占比例约是0.89。虽然128bit的共享内存操作以及其引发的bank conflicts会带来更高的延迟，但是通过合理的安排计算足以将这些延迟抵消掉。

dgemm, loop unrolling+double buffering+128bit LD/ST

```
__global__ void cuk_dgemm_unroll_db_128b( double* d_C, const double* d_A, const double* __restrict__ d_B, int n, int lda, int ldb, int ldc )
{
        __shared__ double2 smem[2048];
        double2 a[4], b[4];
        ……

        if(threadIdx.y<n){
                ((double*)smem)[0*32+sidx]=d_A[0*32];
                ((double*)smem)[1*32+sidx]=d_A[1*32];
                ((double*)smem)[2*32+sidx]=d_A[2*32];
                ((double*)smem)[3*32+sidx]=d_A[3*32];
                ((double*)smem)[1024+0*32+sidx]=d_B[0*32*ldb];
                ((double*)smem)[1024+1*32+sidx]=d_B[1*32*ldb];
                ((double*)smem)[1024+2*32+sidx]=d_B[2*32*ldb];
                ((double*)smem)[1024+3*32+sidx]=d_B[3*32*ldb];
        }
        for( k=n-8; k>=0; k-=8 )
        {
                __syncthreads();
                if(threadIdx.y<k){
                        d_A+=(lda<<3); d_B+=8;
                        sidx^=0x800;
                        ((double*)smem)[0*32+sidx]=d_A[0*32];
                        ((double*)smem)[1*32+sidx]=d_A[1*32];
                        ((double*)smem)[2*32+sidx]=d_A[2*32];
                        ((double*)smem)[3*32+sidx]=d_A[3*32];
                        ((double*)smem)[1024+0*32+sidx]=d_B[0*32*ldb];
                        ((double*)smem)[1024+1*32+sidx]=d_B[1*32*ldb];
                        ((double*)smem)[1024+2*32+sidx]=d_B[2*32*ldb];
                        ((double*)smem)[1024+3*32+sidx]=d_B[3*32*ldb];
                }
        #pragma unroll
                for( int i=0; i<8; ++i ){
                        mFetchSmem(0,1,2,3,i)
                        mRank8x8(0,1,2,3)
                }
                lane^=0x400; slot^=0x400;
        }
        ……
```

}

合理使用128bit存储操作可以减少单个线程内的访存事务的数量(也会因此提升计算指令的整体比例)从而提升性能。对于双字宽数据,提升效果相对较小;因为相对于单字宽数据,双字宽内存事务的数量减少相对较低,因此计算指令所占比例的提升也较小(不过对于矩阵乘法，这仍然是进一步提升效率的有效且必须的技术)，在单精度的情况下会得到更好的效果。

2.3 二级数据预取

除了对全局内存的数据进行预取外，我们可以进一步加入共享内存的预取流水,从而通过将计算和共享内存操作重叠来进一步隐藏访问共享内存的延迟，同时减少计算中等待数据的流水线停顿。另外，对于双精度矩阵乘法在某些设备上使用双缓冲反而会降低性能,尤其是规模比较小的矩阵不建议使用；而加入对共享内存数据的预取在多数设备和规模上均能性能提升(作者在GTX680，GTX780，GTX970上测试中都获得了提升)。

dgemm, loop unrolling + double buffering + 128bit LD/ST + smem_prefetch

```
__global__ void cuk_dgemm_unroll_db_128b_prefsmem( double* d_C, const double* d_A, const double* __restrict__ d_B, int n, int lda, int ldb, int ldc )
{
	……

	((double*)smem)[0*32+sidx]=d_A[0*32];
	((double*)smem)[1*32+sidx]=d_A[1*32];
	((double*)smem)[2*32+sidx]=d_A[2*32];
	((double*)smem)[3*32+sidx]=d_A[3*32];
	((double*)smem)[1024+0*32+sidx]=d_B[0*32*ldb];
	((double*)smem)[1024+1*32+sidx]=d_B[1*32*ldb];
	((double*)smem)[1024+2*32+sidx]=d_B[2*32*ldb];
	((double*)smem)[1024+3*32+sidx]=d_B[3*32*ldb];
	__syncthreads();
	mFetchSmem(0,2,4,6,0)
	for( k=n-8; k>=0; k-=8 )
	{
		if(threadIdx.y<k)
		{
			d_A+=(lda<<3); d_B+=8;
			sidx^=0x800;
			((double*)smem)[0*32+sidx]=d_A[0*32];
			((double*)smem)[1*32+sidx]=d_A[1*32];
			((double*)smem)[2*32+sidx]=d_A[2*32];
			((double*)smem)[3*32+sidx]=d_A[3*32];
			((double*)smem)[1024+0*32+sidx]=d_B[0*32*ldb];
			((double*)smem)[1024+1*32+sidx]=d_B[1*32*ldb];
			((double*)smem)[1024+2*32+sidx]=d_B[2*32*ldb];
			((double*)smem)[1024+3*32+sidx]=d_B[3*32*ldb];
		}
		mFetchSmem(1,3,5,7,1)
		mRank8x8(0,2,4,6)
		mFetchSmem(0,2,4,6,2)
		mRank8x8(1,3,5,7)
		mFetchSmem(1,3,5,7,3)
```

```
            mRank8x8(0,2,4,6)
            mFetchSmem(0,2,4,6,4)
            mRank8x8(1,3,5,7)
            mFetchSmem(1,3,5,7,5)
            mRank8x8(0,2,4,6)
            mFetchSmem(0,2,4,6,6)
            mRank8x8(1,3,5,7)
            mFetchSmem(1,3,5,7,7)
            mRank8x8(0,2,4,6)
            __syncthreads();
            lane^=0x400, slot^=0x400;
            mFetchSmem(0,2,4,6,0)
            mRank8x8(1,3,5,7)
        }
        ……
}
```

2.4 细节调优

以上代码都假设A的行数，B的列数以及C的行列数是128的倍数。所以如果矩阵大小不匹配block的计算尺寸，那么更好的实现是添加一个边界处理的内核专门处理边界。如果边界大小和block的计算尺寸相差不大(因此不会浪费太多显存空间),则可以通过填充零将矩阵补齐到匹配block的计算尺寸,这样就不再需要单独的边界处理内核，效率也会更高些。还有一点需要说明：将数据存取操作和浮点计算操作混合排列可以获得更高的效率（更高的IPC）,这里为了更清晰的表达所以没有这样做。上述代码的性能仍有较大的提升空间，如果想获得接近峰值的效率，需要手工使用PTX或SASS对细节进行调优，否则仅仅使用高层的C/C++语言进行内核开发是无法获得接近于峰值性能的。如果开发人员想通过使用PTX显式的控制诸如寄存器分配，指令排序等问题，可能得不到想要的结果。原因是PTX并非真正的面相本地机器指令的汇编语言，而是一种为了兼容不同计算能力的设备而设计的一种类似寄存器传输语言（RTL）的中间层伪汇编语言；从源代码到PTX的编译过程只会做一些初级的优化，多数优化，比如寄存器分配，指令重排，基本块融合等都是在从PTX代码到SASS原生汇编代码的过程中进行的(PTX和SASS的关系类似AMD GPU编程中的IL和GCN原生ISA的关系，这里所说的多数情况在IL和GCN ISA之间也同样存在)。因此若要显式的控制指令的执行顺序和寄存器分配等就需要直接使用SASS,不过NVIDIA现在并未给出直接使用汇编编写GPU内核程序的开发工具，也未开放任何关于指令集编码信息的文档，因此开发者需要自己想办法绕过这个限制。至少对于矩阵乘法，ptxas做的并不是太好（使用C/C++只能得到最高约75%峰值的效率，使用PTX可以超过80%的峰值，但是离最好的效果还有差距），主要有两个原因：

1 指令的排序效果不理想，也就是LD/ST指令，FMA，IADD，LOP，ISETP等指令之间的排列位置不够理想，最好的指令之间的排列距离要综合考虑指令管线的深度，dual issue,以及各种操作的发射延迟和计算延迟；对指令放置在不同位置进行测试选优也往往是必不可少的。对于这些情况人工通常可以比编译器做的更好。

2 寄存器bank冲突(bank-conflict)和功能单元的操作数端口对齐问题对性能的影响也是非常大的。寄存器bank冲突分为2-way和3-way两种情况，对于2-way,每条FMA指令发射到流水线会增加一个时钟的延迟，3-way则会增加2个时钟周期，这可能会严重影响效率（每个循环步增加的时钟周期=存在端口冲突的指令数 x

1或是2）,ptxas为了避免寄存器bank冲突以及强制将寄存器对齐到指令的操作数端口而增加了很多冗余的MOV指令（循环体的追尾效应，使用cuobjdump查看SASS代码会看到在循环体的末尾处多出很多MOV操作）,虽然MOV操作是直接在寄存器文件内部移动数据（意味着无需像其它操作那样先从数据端口读取寄存器中的数据，然后再将数据通过数据端口写入寄存器），但仍然不是免费的,哪怕其延迟很低。即使这样仍然不能保证完全消除寄存的端口冲突。循环体的追尾效应虽然可以通过在循环体前端移除数据预取代码中的条件代码以及通过循环剥离移除循环体内的条件代码来消除；但由于第一因素仍然存在，所以性能的提升仍然受限。

最后需要说明的是由于每个线程的输出是8字长，所以每16个线程输出一列不会有非合并写出的问题(对于同一个warp对应的8字节数据，会分两个批次进行传输，每次传输连续的128字节序列，对应16个线程)，因此是否通过共享内存对数据进行重新打包对性能影响不大，因此我们并没有使用，但是当使用单精度实现时需要这样做。以下是使用共享内存对数据进行重打包的示例代码（d_C的地址计算也需要做相应的改变）：

```
        double2* Cs=&smem[(slot<<4)+lane];
        volatile double* Ct=&((double*)smem)[(threadIdx.y<<8)+threadIdx.x];
#pragma unroll
        for( int i=0; i<64; i+=8 )
        {
                Cs[0*16]=make_double2(c[i+0],c[i+1]);
                Cs[1*16]=make_double2(c[i+2],c[i+3]);
                Cs[2*16]=make_double2(c[i+4],c[i+5]);
                Cs[3*16]=make_double2(c[i+6],c[i+7]);
                d_C[0*32]=Ct[0*32];
                d_C[1*32]=Ct[1*32];
                d_C[2*32]=Ct[2*32];
                d_C[3*32]=Ct[3*32];
                d_C[ldc+0*32]=Ct[4*32];
                d_C[ldc+1*32]=Ct[5*32];
                d_C[ldc+2*32]=Ct[6*32];
                d_C[ldc+3*32]=Ct[7*32];
                d_C+=(ldc<<4);
        }
```

下面给出各个版本的测试结果(每个测例循环100次取平均，单位ms)

version	GTX970		
size	1024x1024	2048x2048	4096x4096
cublas	0.019500	0.141330	1.109940
unroll	0.017320	0.136030	1.081080
unroll+128b	0.016690	0.132290	1.047700
unroll+128b+prefsmem	0.016670	0.131820	1.042550
unroll+db+128b	0.017000	0.135720	1.070310
unroll+db+128b+prefsmem	0.016850	0.134790	1.068920

本章小结

通过合理组合使用循环展开，128bit存储操作以及对全局内存和共享内存的两级数据预取可以有效提升矩阵乘法的效率。在这些优化技术中，循环展开，数据预取通常是有效的，而128bit内存操作和双缓冲技术则需视情况而定。此外，对一些细节的处理也必不可少，比如将固定的索引计算移除循环并加到指针变量上以减少地址计算和指令数量。当矩阵规模比较小时(比如512x512)，可以每个线程计算32个元素，每个block计算128x64的输出，从而充分利用硬件资源平衡计算负载。当然，这些优化技术并不仅限于矩阵乘法。

小结

一开始我们就从一个性能很高的版本开始更进一步的优化，之所以没有从一个更加简单的版本(比如CUDA programming
guide上的例子)开始逐步走完全程，并不是因为作者太懒(好吧，我承认是有这个原因)。其一是因为各种相关书籍都有介绍，除了占用篇幅，作者实在找不到把那些内容放在本书的理由。更主要的原因是作者更建议一开始就从尽可能高效的方法去实现，这个建议对任何项目都适用。假设你正在做一个比较大的项目，可能会想着先让项目运行起来，至于效率以后可以逐步改进。这种想法对有些项目可能适用。但是对于更多的项目这样做可能会大幅度拖延开发进度，因为当你准备优化时，可能会发现为了改进性能，不得不改变数据结构
(因为有时候大幅度的性能提升必须依赖特定的数据结构和算法)和大范围的改写代码逻辑。有些情况下一个简单的数据转换接口即可在时间和效率上得到折中（通常意味着不是最好)；但很多时候对数据结构或是算法的改变却会牵一发而动全身甚至从架构到代码都需要重新设计开发。所以强烈建议从一开始就用尽可能优化的方法作为起点，用尽可能优质的实现构建你的项目，这样前期可能会需要多些时间，但是之后会发现这样可以大大提高整体的开发进度，同时项目的质量也会得到保障。总之，对一个低质量的程序通过渐进的方法不断优化的工作效率远低于直接使用优化的方法；正所谓快刀斩乱麻，有时候屏蔽已有的东西，可以避免被潜意识中经验性或现有的代码规则所限制。

参考资料

1《cuda programming guide》

2《kepler tuning guide》

3《maxwell tuning guide》

4《PTX ISA version 4.2》

5《nvidia-kepler-gk110-architecture-whitepaper》

6 Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs，by Junjie Lai, Andr_e Seznec。

7 Anatomy of High-Performance Matrix Multiplication，KAZUSHIGE GOTO,The University of Texas at Austin and ROBERT A. VAN DE GEIJN The University of Texas at Austin。

8 On Reducing TLB misses in Matrix Multiplication,kazushige goto, Robert van de geijn, november 1, 2002。

第三章 基于GPU的稀疏直接求解器

前言

本章可能是所有章节中最难得了，如果节奏过快，对于即使有一定相关开发经验的人来说要彻底理解其中的内容也会相当困难，所以作者会给出相比其它章节更多的说明。第一章中类似的技术将会在本章得到应用，通过阅读本章，读者不仅可以进一步巩固第一章中学到的GPU计算的优化技术，还能了解到一些有趣的算法。在这个过程中读者还会更深的体会到从算法的伪代码到实际的可执行代码之间的难度。

3.0 稀疏矩阵直接解法概述

很多科学计算问题中都涉及到求解线性方程组，而这些方程组不仅规模很大且通常是稀疏的。解决这类问题通常使用迭代解法或直接解法。迭代解法一般内存空间需求小，程序实现比较简单；但是迭代解法的收敛速度依赖于矩阵的性态，如果矩阵过于病态，即使采用复杂的预处理技术(预处理越复杂，也意味着求解效率越低，内存空间需求越大，从而迭代解法优势也会逐渐丧失)也很难收敛甚至不收敛。直接解法的缺点是内存空间的需求比迭代解法大很多，但是其具有解的精度高，计算时间稳定，没有收敛性问题等优点；尤其是当矩阵不变，而具有多个右端项或右端项不断变化时，效率比迭代法更高。但是直接法程序实现的难度非常的大，其中所用到的算法通常深度抽象，逻辑交错庞杂，即使是作者本人，在事隔几年后读自己的代码也废了不少精力，所以希望读者若对这一领域感兴趣一定要有足够的耐心和毅力；若工作中并不涉及这一领域则建议跳过本章以免浪费太多精力。稀疏直接求解方法繁多纷杂，但整体步骤大体一致：

1 顶点重排

2 符号分解

3 数值分解

4 三角回代

5 迭代改善

有时也将第1步和第2步合并在一起统称为符号分析，最后一步的迭代改善则是根据具体需求决定是否使用，不是必须的。本章以求解大规模对称正定矩阵为例讲解开发快速求解大规模稀疏矩阵的主要算法和相关优化技术。对称正定矩阵的求解技术已经发展的非常成熟，通常使用cholesky三角分解方法，该方法数值稳定性好，且分解过程中无需选主元。在开始之前我们对一些符号进行约定：

adj(v)　　：顶点v的邻接节点的集合。

deg(v)　　：顶点v的度，也就是顶点为邻接节点的个数。

reach(v)　　：顶点v的可达集，即通过顶点v可以到达的顶点的集合。

snode(v0, v1, v2, …, vn)：表示组成超节点的所有顶点编号的集合，简写为snode。

innz(v)　　：顶点v对应的波前中的非零元的行号或列号。

3.1 基于quotient graph的符号分析

在进行真正的数值三角分解之前，我们需要事先确定存储分解因子所需要的内存大小，以避免在分解的过程中动态内存分配。这一过程称之为符号分解或符号消元，这不仅可以提高程序的运行效率，还可以节省很多内存并避免更多的内存碎片。假设矩阵A具有n个非零元，分解后的分解因子L具有m个非零元(因此有m-n个非零元填入，通常m-n>>n)，那么按照定义进行符号分解需要O(m)的存储空间。但是[1]中证明了，只需要O(n)大小的内存即可完成符号分解，且这一大小是可以事先确定的。实现这一方法的模型就是quotient-graph。quotient-graph的执行流图如下：

3.1.1 顶点重排

我们知道，三角分解过程中会产生很多填入元，如果直接进行分解，其填入元数量一般非常多。当矩阵达到一定规模时，对存储空间的需求会大幅度暴涨，求解就会非常耗时和困难甚至无法完成。所以我们需要事先对原矩阵的稀疏结构图中的顶点进行重排序处理。重排序的过程并不改变原始图的拓扑结构，因而具有等价关系，下面两幅图展示了重排序的作用，左边是排序前的原始矩阵，右边是排序后的矩阵：

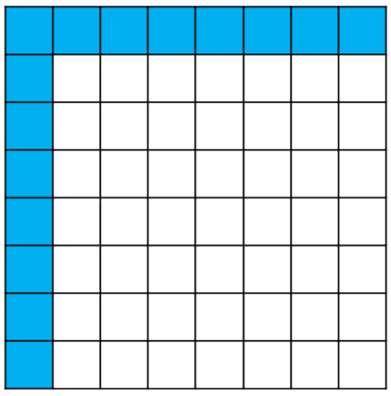


fig3.1

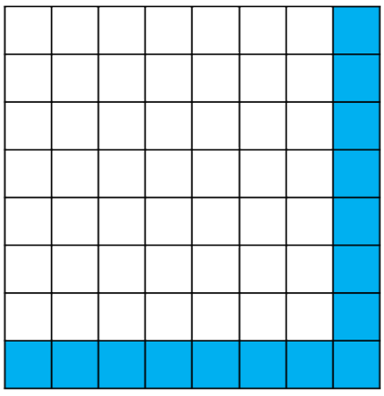


fig3.2

可以看到，如果直接进行分解，矩阵的所有元素都编程了非零元素；而重排序后进行分解非零元素的数量不变，这虽然只是最理想的情况，但实际上通过顶点的重排序仍然能大幅度减少分解过程中新产生的非零元素。稀疏矩阵顶点图的重排序算法通常都是NP问题，从而只能采用启发式算法。目前最常用的算法有以缩减外形(或带宽)为目的算法，的如RCM算法；
以减少填入元为目的的类最小度算法(minimum degree)和用于分而治之的嵌套剖分(nested dissection)算法。虽然更窄的带宽通常意味着更少的填入元，但是其效果一般不如类最小度算法。嵌套剖分目前比较成熟的技术是通过multilevel方法(类似多重网格方法中的思想)找到极小顶点分割子将整个图剖分成多个独立的子图，然后再采用类最小度算法对各个子图分别进行局部优化处理(fig3.3)。类最

小度算法常用的有最小度算法(MD),多重最小度算法(MMD)和近似最小度算法(AMD)。多数情况下性能最高效果最好的是近似最小度算法，本书中我们仅采用最小度算法。

最消度算法的思想是当一个顶点的度比较小时，当消去该顶点时引入新的边的概率会更小或是引入的新边的数量会更少，因此所有最小度算法都是从一个具有最小度的顶点开始。第一个具有最小度的顶点可以随机的选择，也可以选择所有具有相同最小度的顶点中具有最大伪直径的顶点。为了简单，我们遍历当前未消元的顶点并选择第一个顶点度最小的顶点。

最小度算法 pesudo-code

```
do{
        v=one vertex of graph with minimum degree
        elimination:
                delete all edges adjacency to 'v'
                add new edges between the adjacency verties of 'v'(if non-existent edges between them)
        update graph: remove 'v' from graph
}while(graph.num_verties>0)
```

最小度算法的实现有很多种方法，这里作者仅给出自己的实现（基于quotient graph模型）。以下代码都经过数百次针对不同布局和大小的矩阵的测试；但是按照严格的标准来说，数百次的测试远远不够，因此作者不保证代码中没有BUG，请读者慎用。

首先，需要选择一个具有最小度的顶点,这里我们称之为主元顶点：

```
int search_pivot( const int* deg, const char* flag, int n )
{
        int i, d, p, q;
        for( i=0; i<n; ++i ){ if( flag[i]==0 ) break; }
        d=deg[i]; p=i;
        while( (++i)<n ){ if( flag[i]==0 ){ q=deg[i]; if( q<d ){ d=q; p=i; } } }
        return p;
}
```

然后需要得到这个顶点的可达集:

```
int gather_reaches( aux_param_t* params, const int64* ptr, const int* nbor, int e )
{
        int64 p, q;
        int i, id, nen, n=0;
        p=ptr[e]; q=p+params->len[e];
        params->flag[e]=1;
        while( p<q ){
                id=nbor[p++];
                if( params->flag[id]==0 ){
                        if( params->mark[id]==0 ){ params->reach[n++]=id; params->mark[id]=1; }
                } else {
                        nen=gather_enbors( params, ptr, nbor, id );
                        for( i=0; i<nen; ++i ){
                                id=params->nbn[i];
```

```
                                        if( params->mark[id]==0 ){ params->reach[n++]=id; }
                                        params->mark[id]=2;
                                }
                        }
                }
                return n;
        }
```

gather_reaches函数通过遍历主元顶点所有未消去的邻接节点以及所有以消去的邻接节点的邻接节点来获得主元顶点的可达集;gather_enbors用来获取某个以消去顶点的邻接顶点：

```
        int gather_enbors( aux_param_t* params, const int64* ptr, const int* nbor, int e )
        {
                int k, id, v, n=0;
                int64 p, q;
                if( params->esize[e]>0 )
                {
                        v=e;
                        for( k=0; k<params->esize[e]+1; ++k ){
                                p=ptr[v]; q=p+params->len[v];
                                while( p<q ){
                                        id=nbor[p++];
                                        if( params->flag[id]==0 ){ params->nbn[n++]=id; }
                                }
                                v=params->elink[v];
                        }
                }
                else
                {
                        p=ptr[e]; q=p+params->len[e];
                        while( p<q ){
                                id=nbor[p++];
                                if( params->flag[id]==0 ){ params->nbn[n++]=id; }
                        }
                }
                return n;
        }
```

在消元过程中，可能有多个具有相同邻接顶点的以消去顶点，为了避免重复计算，需要对顶点进行"吸收",亦即将多个具有相同邻接节点的以消去顶点合并到其中一个超顶点。顶点吸收可以减少冗余计算，同时也是基于quotient-graph模型计算时所必须的，否则内存空间将无处容纳顶点在消元过程中增加的边:

```
        void absorb_elements( int* nbor, const int64* ptr, aux_param_t* params, int nr, int e )
        {
                int i, id, enb, n, n_deads, n_slots, sign, v;
                int64 ep, p=ptr[e], q=p+params->len[e];
                enb=0;
                for( enb=0; p<q; ++p ){
                        if( params->flag[( id=nbor[p] )]!=0 ){ params->nbn[enb++]=id; }
                }
                if( enb==0 ){ for( id=0; id<nr; ++id ){ params->mark[params->reach[id]]=0; } return; }

                for( p=0, i=0; i<enb; ++i ){
                        id=params->nbn[i];
                        params->spot[p++]=id; n=params->esize[id]; params->flag[id]=2;
                        for( q=0; q<n; ++q ){ params->spot[p++]=( id=params->elink[id] ); }
```

```
            }
            v=e; params->esize[e]=(int)p;
            for( i=0; i<p; ++i ){ id=params->spot[i]; params->elink[v]=id; v=id; }

            i=n_slots=sign=0; v=e;
            for(;;)
            {
                ++n_slots; ep=ptr[v]; p=ep; q=ptr[v+1];
                while( p<q ){
                    if( i>=nr ){ sign=1; break; }
                    nbor[p++]=params->reach[i++];
                }
                params->len[v]=(int)(p-ep);
                if( sign!=0 ) break;
                v=params->elink[v];
            }
            params->esize[e]=--n_slots;

            for( params->flag[e]=2, v=0; v<nr; ++v )
            {
                id=params->reach[v];
                if( params->mark[id]!=2 ){ params->mark[id]=0; continue; }
                params->mark[id]=0; p=ptr[id]; q=p+params->len[id];
                while( p<q ){ if( params->flag[nbor[p]]==2 ){ nbor[p]=e; break; } ++p; }
                ep=p; i=0;
                while( (++p)<q ){ if( params->flag[nbor[p]]!=2 ){ params->spot[i++]=nbor[p]; }; }
                n_deads=(int)(q-(++ep)-i);
                if( n_deads>0 ){
                    for( p=0; p<i; ++p, ++ep ){ nbor[ep]=params->spot[p]; }
                    params->len[id]-=n_deads;
                }
            }
            params->flag[e]=1;
        }
```

最后我们需要更新顶点的度，首先我们需要计算出当前消元顶点的可达集（是对可达集中的顶点的度进行更新），为此遍历消元顶点的邻接节点，如果其邻接节点是以消去的顶点，则遍历该顶点的邻接顶点，如果其邻接顶点未被标记，则加入可达集；如果消元顶点的邻接顶点是未消去顶点，则将其加入可达集（如果未标记的话）。顶点v的可达集可表示为

$$Reach(v) = adj(s) \cup \sum adj(e)$$

接着我们遍历可达集中的顶点，对于可达集中的每个顶点，遍历其邻接顶点，如果其邻接顶点为以消去顶点，则将该顶点的顶点度加1，否则不变。如果其邻接顶点为非消去顶点，则将其顶点度加1，如此进行，直至可达集中的所有顶点处理完成，下面是具体的实现代码：

```
    void update_degrees( aux_param_t* params, const int64* ptr, const int* nbor, int nr )
    {
        int64 p, q;
        int k, d, s, n, i, id;
        for( k=0; k<nr; ++k )
        {
            s=params->reach[k];
            if( params->deg[s]==0 ) continue;
            d=0; p=ptr[s]; q=p+params->len[s]; params->mark[s]=1;
            do{
```

```
                                id=nbor[p];
                                if( params->flag[id]==0 ){
                                        if( params->mark[id]==0 ){ params->spot[d++]=id; params->mark[id]=1; }
                                } else {
                                        n=gather_enbors( params, ptr, nbor, id );
                                        for( i=0; i<n; ++i ){
                                                id=params->nbn[i];
                                                if( params->mark[id]==0 ){ params->spot[d++]=id; params->mark[id]=1;
}
                                        }
                                }
                        }while( (++p)<q );
                        params->deg[s]=d; params->mark[s]=0;
                        for( i=0; i<d; ++i ){ params->mark[params->spot[i]]=0; }
                }
        }
```

现在可以组装成完整的基于quotient-graph的最小度重排序函数了：

```
        void mdo( int* iperm, char* aux, int64* Ap, int* Ai, int n )
        {
                aux_param_t params;
                int64 start, end;
                int d, i, e, m;
                params.flag     =(char*)aux;
                params.mark     =params.flag+n;
                params.deg      =(int*)( params.mark+n );
                params.len      =params.deg+n;
                params.esize    =params.len+n;
                params.elink    =params.esize+n;
                params.reach    =params.elink+n;
                params.nbn      =params.reach+n;
                params.spot     =params.nbn+n;

                for( start=0, i=0; i<n; ++i )
                {
                        params.flag [i]=0;
                        params.mark[i]=0;
                        params.esize[i]=0;
                        end=Ap[i+1];
                        d=(int)(end-start);
                        params.deg[i]=d;
                        params.len[i]=d;
                        start=end;
                }

                for( i=0; i<n-1; ++i )
                {
                        e=search_pivot( params.deg, params.flag, n );
                        iperm[i]=e;
                        m=gather_reaches( &params, Ap, Ai, e );
                        if( m==0 ) continue;
                        absorb_elements( Ai, Ap, &params, m, e );
                        update_degrees( &params, Ap, Ai, m );
                }
                for( i=0; i<n; ++i ){ if( params.flag[i]==0 ) break; }
```

```
        iperm[n-1]=i;
    }
```

aux是一个辅助内存数组，之所以没有在内部分配，是因为在求解大规模矩阵时，通常会对多个子矩阵的顶点图进行重排序，这时候可以分配与参与计算的线程数量对等数量的aux数组，每个线程中的aux数组的大小是该线程中所要计算的图中尺寸最大的图所对应的大小，这样aux数组便可被各个线程中的多个顶点图复用，而不用频繁的开辟和释放操作。通过重排序得到了新的顶点编号后，就可以建立消去树了。需要说明的是，在我们的最小度算法中使用的是外度更新，而非标准的顶点度更新。顶点v的外度定义为

由于考虑了额外更深层次顶点的影响，因此使用顶点外度的效果通常都要好于使用标准顶点度。

最小度算法有很多改进措施，第一个改进方法是将多个无区别顶点同时消去，这一方法也就是MMD算法。所谓无区别顶点就是当两个不相邻顶点的邻接顶点集合与自身的集合相同的顶点，定义为

$$u \cup adj(u) \iff v \cup adj(v),$$

用可达集可表示为

$$reach(v, S) \cup v \iff reach(u,S) \cup u$$

第二个改进方法是进行不完全的度更新，也就是不需要每次都对完整的邻接顶点集合的所有顶点进行更新，而只更新满足一定条件的具有最小度的顶点，这一方法称为近似最小度(AMD)算法，AMD算法多数情况下的排序效果比MD和MMD算法好，同时算法的时间复杂度也大幅降低。

3.1.2 构建消去树

```
void build_etree( char* aux, int* parents, const int64* Cp, const int* Ci, int n )
{
        int64 a, b, p, q, *Rp, *ptr;
        int *Ri, *ancestor, i, k, next;
        Rp=(int64*)aux;
        ptr=Rp+n+1;
        Ri=(int*)(ptr+n);
        ancestors=Ri+Cp[n];
        memset( Rp, 0, n*sizeof(int64) );

        p=Cp[0];
        for( k=0; k<n; ++k ){
                q=Cp[k+1];
                while( p<q ){ ++Rp[Ci[p++]]; }
        }
        a=Rp[0]; Rp[0]=0;
        for( k=0; k<n; ++k ){
                b=Rp[k+1]; Rp[k+1]=a; a+=b;
        }
        memcpy( ptr, Rp, n*sizeof(int64) );
        p=Cp[0];
        for(k=0; k<n; ++k ){
                q=Cp[k+1];
```

```
            while( p<q ){ Ri[ptr[Ci[p++]]++]=k; }
        }

        for( k=0; k<n; ++k )
        {
            parents[k]=-1; ancestors[k]=-1;
            for( p=Rp[k]; p<Rp[k+1]; ++p )
            {
                i=Ri[p];
                while((i>=0)&(i<k)){
                    next=ancestors[i];
                    ancestors[i]=k;
                    if( next<0 ){ parents[i]=k; }
                    i=next;
                }
            }
        }
}
```

其中parents存储了消元过程中每个顶点之间的依赖关系，这个信息将在后面符号分解和寻找超节点的过程中被用到。

3.1.4 符号分解

我们队矩阵A的稀疏结构X进行重新编码得到具有新的稀疏结构Y的矩阵B，Y与X中顶点之间的拓扑关系并未改变，但是却具有更少的填入元。然后通过稀疏结构Y进行符号分解。符号分解的目的是获取分解过程中分解因子中各个非零元素的位置信息以及整个分解因子所需要的存储空间大小，从而避免进行实际的数值分解的过程中动态的计算索引以及动态的内存分配，这样将具有更高的效率并避免内存碎片的积累。符号分解之所以有效是因为很多顶点对应的波前(后续章节会讲到)可能进行多次更新，若果直接进行数值分解，就需要多次重复的计算索引以及很多低效的内存开辟释放操作，严重影响效率。符号分解的代码同样基于quotient-graph,前面的一些函数会得到复用。

首先我们需要先计算得到分解因子中每列的非零元素数量，并最终得到整个分解因子所需要的内存大小：

```
int64 symbolic_eliminate( char* aux, int* Ln, const int64* Yp, const int* Yi, int n )
{
        int64 nA=Hp[n]<<1;
        int* Ai=(int*)(aux+ELIMINATION_BUFFER_BYTES(n));
        int64* counter=(int64*)(Ai+nA);
        int64* Ap=counter+n+1;
        int64 p, q, c, nnz;
        int k, m;
        for( p=0, k=0; k<n; ++k ){
            q=Yp[k+1]; counter[k]=q-p; p=q;
        }
        for( p=0, k=1; k<=n; ++k ){
            q=Yp[k];
            while( p<q ){ ++counter[Yi[p++]]; }
        }
        for( Ap[0]=0, k=0; k<n; ++k ){
            Ap[k+1]=Ap[k]+counter[k]; counter[k]=Ap[k];
```

```
        }
        for( p=0, k=0; k<n; ++k ){
                q=Yp[k+1];
                while( p<q ){ Ai[counter[Yi[p++]]++]=k; }
        }
        for( p=0, k=0; k<n; ++k ){
                q=Yp[k+1];
                c=counter[k];
                while( p<q ){ Ai[c++]=Yi[p++]; }
        }
        aux_param_t     params;
        params.flag     =(char*)aux;
        params.mark     =params.flag+n;
        params.len      =(int*)( params.mark+n );
        params.esize    =params.len+n;
        params.elink    =params.esize+n;
        params.reach    =params.elink+n;
        params.nbn      =params.reach+n;
        params.spot     =params.nbn+n;
        for( p=0, k=0; k<n; ++k ){
                params.flag [k]=0;
                params.mark [k]=0;
                params.esize[k]=0;
                q=Ap[k+1];
                params.len[k]=(int)(q-p);
                p=q;
        }
        nnz=0;
        for( k=0; k<n; ++k ){
                m=gather_reaches( &params, Ap, Ai, k );
                if( m==0 ) continue;
                Ln[k]=m; nnz+=m;
                absorb_elements( Ai, Ap, &params, n, k );
        }
        return nnz;
}
```

symbolic_eliminate的返回值即是分解因子非零元的数量，而分解因子每列非零元的数量保存在Ln中,然后我们可以在分解前一次性的分配索引数组的内存，并利用Ln中的信息计算每列非零元素的行号以及存储位置：

```
void symbolic_decompose( char* aux, int* Li, const int64* Lp, const int* Sp, const int* nodes, const int* superIDs, const int64* Ap,
const int* Ai, int n, int n_snodes )
{
        int64 ii, ii0, ii1, pp, qq, *pos;
        char* mark;
        int *spot, *a;
        int i k, v, d, sid, c;

        mark=aux;
        pos=(int64*)(aux+n);
        spot=(int*)(pos+n_snodes);
        memset( mark, 0, n );
        for( k=0; k<n_snodes; ++k )
```

```
        {
                v=nodes[Sp[k+1]-1];
                ii0=Ap[v];
                ii1=Ap[v+1];
                pp=Lp[k];
                while(ii0<ii1){
                        i=Ai[ii0++];
                        if(i>v){ Li[pp++]=Ai[ii0++]; }
                }
                pos[k]=pp;
        }

        for( k=0; k<n_snodes; ++k )
        {
                ii0=Lp[k];
                ii1=Lp[k+1];
                d=(int)(ii1-ii0);
                if((k>0)&(d>1)){
                        a=&Li[ii1];
                        sort( a, a+d );
                }
                for( ii=ii0; ii<ii1; ++ii )
                {
                        v=Li[ii];
                        sid=superIDs[v];
                        if( sid==k ) continue;
                        pp=Lp[sid]+1;
                        qq=pos[sid];
                        c=0;
                        while( pp<qq ){
                                v=Li[pp++]; mark[v]=1; spot[c++]=v;
                        }
                        for( pp=ii+1; pp<ii1; ++pp ){
                                v=Li[pp];
                                if( mark[v]==0 ){ Li[qq++]=v; }
                        }
                        for( i=0; i<c; ++i ){ mark[spot[i]]=0; }
                        pos[sid]=qq;
                }
        }
}
```

这里我们是基于超节点进行的符号分解，因此在执行**symbolic_decompose**之前还需要构建超节点，超节点方法的相关内容将在**2.3**节进行介绍。注意红色部分的代码，由于符号分解的过程中我们只记录了正在进行分解的列的行号，但是为了以正确的顺序进行后续的分解，需要在此之前对其中的行号（或列号）从小到大排序。现在整个符号分析已经完成，将其过程总结如下：

1 对顶点进行重排序。

2 利用重排序得到的新顶点编码构建消去树，得到消元过程中结点依赖关系的数组**parents**。

3

利用新的顶点编码和和第2步中得到的parents信息寻找并构造超节点，并根据超节点内顶点编码的连续性把超节点标记为静态或动态的。

4

利用新的顶点编码和第3步中得到的超节点信息进行符号分解确定整个分解过程所需要的内存大小和非零元素的索引信息。

3.2 多波前法

多波前(multifrontal)法发展于有限元中的波前法，其目的是提高稀疏矩阵分解过程中的并行性，是目前求解大规模及超大规模稀疏矩阵普遍使用的高效且成熟的方法。下面我们以易于理解的方式通过多个图的展示来说明多波前法的思想。

图4

第一步：波前{0,1,2,4}可以同时进行分解，波前0的结果对{2,6,7}进行更新，

波前1的结果对波前3进行更新，波前2和波前4的结果对波前5进行更新。

第二步：波前{3,5}可以同时进行分解，波前3的结果更新波前6。

第三步：对波前6进行分解，并用分解结果更新波前7。

第四步：对波前7进行分解。

3.3 超结点法

超结点(supernodal)法也是用来提高计算效率的一种方法，但不同于多波前法同时对多个独立的波前进行并行计算， 而是通过提高计算密度和缓存友好来提升效率。假设一个连续的顶点序列{ vi, vi+1, vi+2, …, vi+n, … }，如果

E1 parent(vi)=vi+1, parent(vi+1)=vi+2,…； 且

E2 num_adj(vi)+1=num_adg(vi+1), num_adj(vi+1)+1=num_adj(vi+2), …

那么将该顶点序列称之为一个超结点，如图所示：

图5

图中有5个超节点，分别为{0,1}，{2}，{3}，{4}，{5,6,7}。对超节点进行分解可以利用高效的稠密矩阵计算提高计算效率。一般来说，获得超节点越少，表示消去树的高度越矮，计算密度越大，从而效率也越高，因此可以通过一些方法来减少超节点的数量。第一种方法是超节点融合，亦即把超节点当做普通的顶点，那么又可以对超节点进一步融合成更大的超节点；但是超节点融合会占用更多的内存。第二种方法是在建立超节点时有限度的容忍，超节点中顶点的邻接节点的数量无需严格满足E2，只要相差小于阀值即可加入超节点。第三种方法是扩展超节点的概念，超节点中的顶点无需满足连续性，但E1条件仍然要满足，可将其称之为动态超节点。现实中更有效的方法是：将超节

点分为静态超节点和动态超节点，同时配合第二种方法，这样不仅可以避免不必要的内存移动(因为动态超节点在计算之前必须要将其移入一块连续的内存中)，还可以最大限度的提高计算的密度，且内存的整体需求也一般会比较低。将超节点方法引入符号分解中还可以大幅降低存储索引所需的内存以及减少符号分解所需的计算量，如图：

图6

图中包括两个动态超节点{0,2},{1,3}和一个静态超节点{4,5,6,7}。

容易看出，对于某个超节点snode{v0,v1,v2,…,vn},只需要存储其具有最小波前的顶点vn对应的那列的非零元的行号，而其它顶点对应的波前中非零元的行号可以表示为:

innz(v0) = { v1, v2, …, vn } + innz(vn)

innz(v1) = { v2, v3, …, vn } + innz(vn)

innz(v2) = { v3, v4, …, vn } + innz(vn)

……

从而整个超节点的非零元的行号（或列号）信息可以通过超节点中的包含的顶点的标号以及最后一个顶点的非零元行号（或列号）完整的表示，亦即snode{ v0, v1, v2, …, vn } + innz(vn)，容易看出这样将使索引存储具有最小的空间需求。

建立超节点可以从正向寻找(从第一列开始向后寻找)，也可以反向寻找(从最后一列开始向前寻找)，不过反向方法通常具有更好的效果，这里分别给出两种实现，在使用中可以从前向和后向方法中选择结果最好的。

寻找超结点的反向方法：

```
int find_supernodals_backward( char* aux, int* Sp, int* verties, char* flags, const int64* Lp, const int* Li, int n )
{
	int64 ii0, ii1, ii2, ii3;
	int* child=(int*)aux;
	int* id=child+n;
	int i, k, n_snodes, snode_size, p, q, parent;
	char b;
	for( k=0; k<n; ++k ){ child[k]=-1; }
	for( k=0; k<n-1; ++k )
	{
		ii0=Lp[k];
		ii1=Lp[k+1];
		parent=Li[ii0+1];
		ii2=Lp[parent];
		ii3=Lp[parent+1];
		if(((ii1-ii0)-(ii3-ii2))==1){
			child[parent]=k;
		}
	}
	for( i=0, n_snodes=0, k=n-1; k>=0; --k )
	{
		if(child[k]!=-2)
		{
```

```
                        snode_size=1;
                        id[i++]=k;
                        p=child[k];
                        child[k]=-2;
                        while(p>=0){
                                ++snode_size;
                                id[i++]=p;
                                q=child[p];
                                child[p]=-2;
                                p=q;
                        }
                        Sp[n_snodes++]=snode_size;;
                }
        }
        for( i=0; i<n; ++i ){ verties[i]=id[n-i-1]; }
        for( i=0; i<n_snodes; ++i ){ child[i]=Sp[n_snodes-i-1]; }
        for( Sp[0]=0, i=0; i<n_snodes; ++i ){ Sp[i+1]=Sp[i]+child[i]; }
        for( k=0; k<n_snodes; ++k ){
                char b=0;
                for( i=Sp[k]; i<Sp[k+1]; ++i ){
                        if((verties[i]+1)!=verties[i+1]){ b=1; break; }
                }
                flags[k]=b;
        }
        return n_snodes;
}
```

寻找超结点的正向方法：

```
int find_supernodals_forward( char* mark, int* Sp, int* verties, char* flags, const __int64* Lp, const int* Li, int n )
{
        memset( mark, 0, n );
        int64 ii0, ii1, ii2, ii3;
        int i, k, parent, n_snodes, snode_size;
        char b;
        const int imax=n-1;
        for( Sp[0]=0, i=0, n_snodes=0, k=0; k<n; ++k )
        {
                if( mark[k]!=0) continue;
                snode_size=1;
                mark[k]=1;
                verties[i++]=k;
                ii0=Lp[k]; ii1=Lp[k+1];
                parent=(k<imax)?Li[ii0+1]:n;
                while(parent<n){
                        ii2=Lp[parent]; ii3=Lp[parent+1];
                        if(((ii1-ii0-ii3+ii2)!=1) | (mark[parent]!=0)) break;
                        ++snode_size;
                        mark[parent]=1;
                        verties[i++]=parent;
                        parent=(parent<imax)?Li[ii2+1]:n;
                        ii0=ii2; ii1=ii3;
                }
                Sp[n_snodes+1]=Sp[n_snodes]+snode_size; ++n_snodes;
        }
        for( k=0; k<n_snodes; ++k ){
```

```
            b=0;
            for( i=Sp[k]; i<Sp[k+1]; ++i ){
                if((verties[i]+1)!=verties[i+1]){ b=1; break; }
            }
            flags[k]=b;
        }
        return n_snodes;
}
```

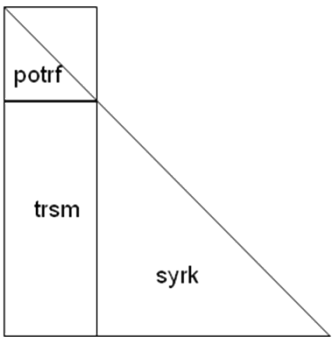
find_supernodal_*函数的返回值时超节点的数量；verties的大小是顶点的数量，每个超节点所包含的顶点编号在verties中是连续存储的；Sp中包含了超节点的第一个顶点在数组中的索引位置以及超节点的大小信息，第i个超节点中的第k个顶点为verties[Sp[i]+k],
其中k<Sp[i+1]。可以如下遍历第i个超节点中的顶点：

```
p=Sp[i]；q=Sp[i+1];

do{

    v=verties[p];

    …

}while((++p)<q);
```

flags中则标记了超节点的静态和动态属性，值为0表示静态超节点，值为1表示动态超节点。

3.4 超结点的分解

对超结点的分解可以方便的利用高效的稠密矩阵算法，下图是超结点分解的示意图



由于大部分的计算都集中在syrk,因此我们重点对syrk的实现进行优化，第一章中的矩阵乘法所用到的技术可以得到复用，只不过矩阵B变成了矩阵A的转置。虽然可以通过直接使用*gemm来实现syrk操作，但是这种方法还是比较低效，因为每一步的计算量通常并不足够大，可能无法充分利用硬件的计算资源；而且syrk比gemm具有更少的全局内存访问量。另一个需要自己开发这些操作而不使用cublas等通用库的原因是它们的实现中syrk操作都是基于完整尺寸的矩阵，对于稀疏矩阵求解这种内存非常宝贵的计算来说过于浪费。因此我们实现自己的syrk操作通过对称存储且只需要大约一半的存储空间，其数据布局如下图所示：

3.5 基于多波前+超结点方法的并行分解

可以将多波前法和超节点方法结合使用以最大化并行度，每个超节点作为一个超波前(如无特别说明，我们将超节点波前统一简称为波前)，多个波前可以同时进行计算。在安排超节点的计算顺序时，除了根据已经建立的消去树外，还要考虑当前的计算资源，包括GPU的数量，每个GPU上得设备内存的大小以及主机内存的大小。我们需要在求解性能和存储空间之间取得一个最佳的平衡。因此每个设备上可以允许同时进行计算的波前需要设置一个上限以避免在消去树的某一层中的波前对存储空间的需求超出已有的内存资源。同时当有多个GPU设备时还需要综合考虑如何将计算和所需要的设备内存平均到每个设备上。通常对消去树的深度遍历可以最小化存储资源使用量，而宽度遍历则可最大化并行度。另一个需要考虑的就是当矩阵规模很大时，内存和显存往往只能容下小部分数据，这时候就需要考虑核外求解，也就是将已经参与完计算的波前数据写入磁盘。但是后续的计算可能会要求载入已经写入磁盘的波前数据，那么在确定波前的消去顺序时又需要考虑如何最小化磁盘读写操作。确定各个波前消去顺序的方法很多，但是波前消去的顺序的多选择性仅限于消去树同一层的波前之间，各个层之间的波前的消去仍需按顺序进行，因此可以很容易的根据自己的设想或需求设计相关算法，这里不再叙述。下面给出基于多波前+超节点方法的分解算法：

idev = current_cpu_tid : 当前设备的id

A : 待分解的矩阵

U[idev] : 第idev设备上得波前更新矩阵，用来存储每个波前的syrk的计算结果

```
for( level = 0; level<etree.n_levels; ++level )
{
        start_snode = etree.snode_list[level][idev]
        end_snode =etree.snode_list[level][idev+1]
        get_front( front[idev], snode, A )
        for( snode=start_snode+1; snode<end_snode; ++snode ){
                if( next_snode.flag == dynamic ){
                        async{
                                assembly_front( front[idev].hFactor,  next_snode, A )
                                copy( front[idev], next_snode )
                        }
                } else {
                        front.[idev].hFactor=A[snode.Ap[next_snode.id] : snode.Ap[next_snode.id+1]];
                        async_copy( front, next_snode )
                }
                front[idev].dU = snode.factor( front[idev].dF )
                locked_extend_add( A, front[idev], snode )
```

```
            if( (compute_mode==out_core)&&(snode.have_no_parents)&&(snode!=last) ){
                async_output_to_disk( snode )
            }
        }
}
```

上述算法是基于多GPU的多波前+超节点方法的实现，使用了静态+动态超节点分类技术。根据消去树的指导，逐层求解各个层内的波前(超节点)。开始分解时整个待分解矩阵存储在内存或磁盘中，然后在每一层计算的开始处使用异步传输将下一个要计算的波前数据从主机内存复制到设备内存,同时进行当前波前的计算，这样可以通过计算隐藏部分或全部下一个待分解的波前的数据传输开销。然后判断当前超节点的标志，如果是静态超节点，只需直接进行原地计算；如果是动态超节点，则要先组装超节点再进行计算。计算的结果分为两部分，一个是波前本身的分解因子F，另一个是当前波前对其它波前的更新矩阵U。由于可能有多个波前对同一个波前的计算有贡献，因此我们需要将extend_add操作锁定以保证计算不发生写冲突。最后需要根据一定的条件决定是否将以完成计算的波前写入磁盘，如果设定的计算模式是核外求解模式并且没有后续的波前计算需要用到当前的波前，则将数据写入磁盘。同时还要注意的是第三个判别条件，如果该波前是整个消去树中的最后一个波前，那么考虑到后续需要进行回代求解，因此从效率上考虑，就无需再写回磁盘了。

本章小结

这一章我们介绍并给出了稀疏直接求解器的具体实现以及先关的优化技术，这其中的很多技术都可以直接或是稍加变通的用在很多数值算法中。特别是超节点方法特别适合用GPU进行计算，但是也需要通过各种优化尽量减少CPU和GPU之间的数据通信带来的开销，建议的优化技术总结如下：

通过对超节点的静态和动态分类减少数据移动的开销，最大化原地分解。

通过动态并行对完整的三角型超节点进行分解以减少和主机通信的开销。

通过使用内存映射和点对点传输降低CPU和GPU之间以及GPU和GPU之间的数据复制开销。

通过对超节点波前的消去顺序进行调度排序使得每个波前进出内存的次数最小化。

使用基于双全局内存和双主机内存的双缓冲技术将待计算节点/波前的传输和当前节点/波前的计算重叠进行，从而充分利用CPU和GPU之间的异步并发模式。

参考资料

1《Direct Methods for Sparse Linear System》， Timothy A. Davis

2 An Approximate Minimum Degree Ordering Algorithm， Patrick R. Amestoy Timothy A. Davisy Iain S. Du_z SIAM J, 1996

3 The Computational Complexity of the Minimum Degree Algorithm， P. Heggernesy S. C. Eisenstatz G. Kumfertx A. Pothen， 2000

4 *The evolution of the minimum degree ordering algorithm*, SIAM Review， 1989

5 J. A. GEORGE AND J. W. H. LIU, *A quotient graph model for symmetric factorization*, in Sparse Matrix Proceedings 1978, I. S. Duff and G. W. Stewart, eds., SIAM

Publications, 1978
6 A NESTED DISSECTION APPROACH TO SPARSE MATRIX PARTITIONING FOR PARALLEL COMPUTATIONS, ERIK G. BOMAN_ AND MICHAEL M. WOLF
7 Minimum Degree Reordering Algorithms: A Tutorial Stephen Ingram University of British Columbia

第四章 多设备编程建议

多设备编程建议1

尽量在计算之前为所有得设备分配好内存，因为内存分配时同步的，对于包含在循环中的计算，内存分配会打断CPU和GPU之间的并发。

多设备编程建议2

如果单个CUDA设备上只需要使用一个流，那么尽量使用默认的NULL流，NULL流比起通过cuStreamCreate/cudaStreamCreate创建的流具有轻微的性能优势。但也不尽然，比如当你开发自己的库时，考虑到与外部环境的异步特性，应尽量自己创建流。如果使用OpenCL,那么你将总要自己创建默认的命令队列。

多设备编程建议3

单个节点内，如果主机端的逻辑控制和其它计算代价很小，
GPU端上下文的切换消耗在整体计算中所占比例足够的小且需要频繁的多设备间的通信，那么建议使用单线程多GPU代替多线程多GPU方法。如果多个设备间在整个计算过程中不需要或是只需很少的通信，那么建议使用多线程多GPU模式代码单线程多GPU模式，这样可以避免频繁的上下文切换导致的驱动开销。

多设备编程建议4

尽量减少设备上下文的切换次数，尤其是应该对循环中上下文切换代码进行调整，如

```
for( k=0; k<n; ++k )
{
        for( i=0; i<n_dev; ++i ){
                cuCtxSetCurrent( p_core->p_ctx[i].cuCtx );
                …
                cuEventRecord(p_core->p_ctx[i].cuEvent, NULL );
        }
        cuCtxSetCurrent( p_core->p_ctx[0].cuCtx );
        for( i=0; i<n_dev; ++i ){
                cuStreamWaitEvent( NULL, p_core->p_ctx[i].cuEvent, 0 );
        }
        …
        cuEventRecord( p_core->p_ctx[0].cuEvent, NULL );
        for( i=0; i<n_dev; ++i ){
                        cuStreamWaitEvent( NULL, p_core->p_ctx[0].cuEvent, 0 );
                cuCtxSetCurrent(p_core->p_ctx[i].cuCtx );
        }
}
```

改写成如下形式：

```
cuCtxSetCurrent( p_core->p_ctx[0].cuCtx );
for( k=0; k<n; ++k )
{
        for( i=0; i<n_dev; ++i ){
                …
                cuEventRecord(p_core->p_ctx[i].cuEvent, NULL );
                cuCtxSetCurrent( p_core->p_ctx[i].cuCtx );
        }
        for( i=0; i<n_dev; ++i ){
                cuStreamWaitEvent( NULL, p_core->p_ctx[i].cuEvent, 0 );
        }
        …
        cuEventRecord( p_core->p_ctx[0].cuEvent, NULL );
        for( i=0; i<n_dev; ++i ){
                cuStreamWaitEvent( NULL, p_core->p_ctx[0].cuEvent, 0 );
                cuCtxSetCurrent(p_core->p_ctx[i].cuCtx );
        }
 }
```

可以减少n-1次设备上下文的切换。

多设备编程建议5

使用OpenCL时，由于内存对象关联的是上下文而不是具体的设备，因此明确为为每个设备分配内存对象具有更高的效率（但是会失去单个设备上下文模式中的多设备同步的简单方便）。

多设备编程建议6

目前不管是GCN设备还是CUDA设备，统一内存的性能并不足够好，因此如果性能是首要目标，那么应该慎用并且摒弃为了抽象或是编码形式上的美观抑或是为偷懒而把那些形而上学的所谓编程艺术作为借口。毕竟新的与好的并不等价，新的形式上的功能只能说明它“有，并且可用”，但未必是最合适的。

第六章 GPU编程优化技术总结

我们在两个章节分别讲述针对CUDA和GCN这两大目前主流的GPU并行计算的设备。但是诸如合并访问,如何避免共享内存的bank conflicts以及简单的指令优化等基本内容这里不再叙述，有需要的可以参考

<<CUDA Programming Guide>>和<<AMD Accelerated Parallel Processing OpenCL Programming Guide>>

这里仅给出一些不常见的优化技巧。

6.1.0 CUDA设备的优化技术

6.1.1 访存优化

1

在计算能力为2.0或以上的CUDA设备上，当一个warp内的所有线程访问同一个地址时，可以使用统一加载操作将一个数据通过缓存广播到warp内的所有线程中，从而提升性能。虽然CUDA programming guide上提到当访问数据的地址和线程号无关且是只读数据时，编译器会自动使用LDU加载指令，但有时编译器并不能得到我们想要的结果。比如：

```
__global__ void add( float* d_a, const float* d_b, int n )
{
    int warpid=(blockDim.x>>5)*blockIdx.x+(threadIdx.x>>5);
    if(warpid>=n) return;
    d_a[threadIdx.x]+=d_b[warpid];
}
```

查看PTX代码，我们发现编译器并未使用LDU，因此我们就需要显示的使用内联PTX汇编来达到我们的目的：

```
#if defined(_WIN64)|| defined(__x86_64)|| defined(_M_X64)|| defined(_M_IA64)||defined(_M_AMD64)
    #define PTX_PTR "l"
#else
    #define PTX_PTR "r"
#endif
__device__ __forceinline__ float __ldu(const float* p )
{
    float val;
    asm volatile ("ldu.global.f32 {%0}, [%1];" : "=f"(val) : PTX_PTR(p));
    return val;
}
```

根据测试，在满足使用LDU的情况下的所有设备中均能获得性能提升，即使是在计算能力3.5+的设备上其效果也要略好于使用LDG(使用纹理缓存)。例如卷积神经网络的计算中，当每个通道中对应的是一个标量的偏置值，那么在卷积计算后对通道施加偏置的操作就可以通过LDU操作高效的完成（虽然也可以通过共享内存，但是使用LDU具有更简洁的实现，并具有轻微的性能优势）。

2

不同的block布局对内存访问的性能也是不同的，配置block的布局时应尽量使得X维度是warp大小的倍数，比如对于下面的程序，128x1比16x8具有微小的性能优势

```
__global__ void kcopy( float* b, const float* a ){
        unsigned int tidx = threadIdx.y*blockDim.x+threadIdx.x;
        b[tidx]=a[tidx];
}
```

6.1 指令优化

1

对于可以完成相同计算的指令集合，应尽可能选择具有更低延迟以及更高混合比例的指令集合，比如在某些设备上双精度可以和内存加载存储指令双发，但是却无法和单精度以及整数指令双发。

2

同时对同一个数组进行多次等距寻址时尽量将不变的索引在开始处加到数组的基址上，这样可以减少地址的计算或是便于基址+常量寻址，从而减少指令数量。

3

对于存在大量计算的循环中如果某些指令，如数据存取指令无需复杂的寻址计算，那么考虑对每个存储操作使用断定，便于编译器将计算和存储指令混合排列从而利用指令的双发(dual issue)机制。

4

使用某些特定的常量，可以将数据融入指令码中，从而具有更小的代码体积。在kepler和maxwell设备上的32位浮点数和整数的双操作数（输入）指令支持全精度的常量，比如

c=a+128.f　对应的SASS指令为 FADD　R2，R0，128,

c=a+10007.f 对应的SASS指令为 FADD32I R2，R0，10007，

这些立即数会被嵌入指令的编码序列中，但是对于三操作数指令（如FMA）则会将常数放入常量内存的第2个bank中（猜测原因是受限于指令编码的长度，因为多出的一个操作数需要额外的位数表示寄存器索引）,因此当一个计算序列中使用FMA不能减少指令数量时（亦即和使用FMUL,FADD数量相同），如果涉及到立即数，则尽量不要使用FMA代替FMUL和FADD，因为操作数直接嵌入指令编码具有更高的效率，也会具有更小的代码尺寸，除非出于精度考虑。如将

```
temp.x=c*b.x+(-s)*b.y;
temp.y=c*b.y+s*b.x;
b.x=a.x-temp.x;
b.y=a.y-temp.y;
a.x+=temp.x;
a.y+=temp.y;
```

转换为

```
temp.x=b.x;
temp.y=b.y;
b.x=a.x+(-c)*temp.x+s*temp.y;
b.y=a.y+(-c)*temp.y+(-s)*temp.x;
a.x+=c*temp.x+(-s)*temp.y;
```

a.y+=c*temp.y+s*temp.x;

并不能减少指令数量，也不会带来性能提升，除非是出于精度考虑。对于双精度数据，有规律的常量也可被嵌入到指令码中，比如0.5,0.25,0.125,0.0625,0.03125,…,1.0, 1.5, 1.25,…, 64.0, 128.0, 65536.0, …;但是无规则的常量会被放入常量内存的第2个bank中，比如

c=a+128.0　　对应的SASS指令为 DADD R2, R7, 128

c=a+790045.7 对应的SASS指令为 DADD R2, R7, c[0x2][0x0]

可以包含在双精度指令码中的常量的具体规则为：

…

+-512.0

+-256.0

+-128.0, +-128.5

+-64.0, +-64.5, 64.25

+-32.0, +-32.5, +-32.25, +-32.125

+-16.0, +-16.5, +-16.25, +-16.125, +-16.0625

…

6.1.2 分支优化

1 使用小的局部数组消除多分支或是简化复杂的条件代码计算。

2
巧妙的利用位操作和局部数组消除分支，例如第三章中通过局部数组简化了主分割面的选择，而通过巧妙的位操作减少了确定下个待遍历节点的分支。

3
分析算法看是否能将不同的路径分配到不同的warp中或block中，同时保证warp或block中的指令路径相同；或是将问题进行拆分成多个内核进行处理。

4 在多分支结构中将判据按照命中的概率从高到低进行排列。

5
某些情况下使用对函数指针列表的寻址代替switch逻辑已消除对大批量分支判断的遍历，同时可以生成更小的代码。

6.1.3 内核调用优化

通常不是限制性能的地方，但是当很多内核在一个循环中被比较长时间的跨距调用时（因此设备驱动的热身会被过长的间隔抹消或是被其它内核的调用覆盖掉内核参数缓存），尤其是当内核具有很多参数时，每次内核参数都需要从内存到设备上的内核参数缓存的复制过程，有时这也会给效率带来较大影响，这里根据作者经验总结了几个方法来优化内核的启动时间：

1

如果内核参数很多，对于指针类型的参数，考虑合并多个指针变量，并在内核内部解引用，这样做有时也会减轻寄存器压力（但也不要想当然，任何时候都应该试着查看编译后的寄存器使用情况）。例如，假设四个长度均为1024的数组(指针合并并不要求每个指针指向的数组大小一样)：

```
__global__ void …( …, const int * d_a, const int * d_b, const int * d_c, const int * d_d, … )
{
        …
        d_a+=tidx;
        d_b+=tidx;
        d_c+=tidx;
        d_d+=tidx;
}
```

可以改成如下形式：

```
__global__ void … ( …, const int * d_a, … )
{
        …
        d_a+=tidx;
        const int * d_b=d_a+1024;
        const int * d_c=d_b+1024;
        const int * d_d=d_c+1024;
        /*
        或者通过对d_a的常量偏移分别访问各个数组：
        d_a,  d_a+102,  d_a+2048,  d_a+3072
        */
        …
}
```

2

动态并行不要滥用，因为动态并行虽然将更多内核启动的控制权放在了设备端，避免了设备和主机的通信开销，但仍无法避免多次的内核启动（即使是设备端的内核启动也不是免费的）；所以很多时候将多个内核合并为一个内核比动态并行具有更高的效率，因为不但减少了内核启动的次数，同时还有助于寄存器或共享内存中的数据复用从而减少访问全局内存的次数（事实上，这也是是否选择内核合并的主要因素之一），但是需要考虑算法，寄存器和共享内存等资源限制。

3

对于追求极致性能的情况下，直接使用PTX和SASS，比如在使用C/C++进行CUDA内核开发时，查看ptx代码可以看到，内核参数中指针变量的地址是作为统一地址传入的，因此需要使用cvta.to.global指令将通用地址转化为全局内存地址的格式，而通过直接手工ptx可以显式的控制内核指针参数所指向的地址空间,从而避免多余的地址空间转换操作。例如：

```
.version 4.2

.target sm_35

.address_size 64

.visible .entry kset( .param .u64 .ptr .global .align 8 param_p, .param .s32 param_val )
{
        .reg .s64 p;
        .reg .s32 tx;
        .reg .s32 val;

        ld.param.u64 p, [param_p];
        ld.param.s32 out, [param_val];
        mov.u32 tx, %tid.x;
        mad.wide.s32 p, tx, 4, p;
        st.global.s32 [p], val;
        ret;
}
```

从PTX指令到本地SASS汇编指令并不是严格一一对应的，在这个翻译的过程中ptxas会进行实际的寄存器分配，指令的替换和重排等优化，因此很多时候你无法通过使用PTX达到控制指令执行顺序和寄存器分配的目的（ptxas做的并不够好，一个实际的例子就是对于矩阵乘法，如果想要达到接近峰值的效率，必须直接对SASS指令进行重拍以及对寄存器进行细致的分配以最小化指令计算延迟和寄存器bank conflicts引起的指令流水线停顿。但是很可惜，NVIDIA并未开放本地汇编的编程环境，甚至连SASS ISA的指令编码格式都未公开，因此需要程序员自己绕开种种限制开发自己的第三方GPU汇编器），但是PTX仍然能在一定程度上影响最终得到的SASS结果，这需要开发者耐心的对指令的顺序和逻辑进行调整并观察最终编译出来的SASS代码。

5

创建CUDA上下文时使用CU_CTX_LMEM_RESIZE_TO_MAX标志，以避免那些具有寄存器溢出的内核在下次启动时重新在设备内存上为寄存器溢出分配局部内存，这样会造成当前线程中的CUDA上下文中所包含的的所有流上的数据传输和内核计算操作中断（即使操作是异步的）。

5.2.0 GCN设备上的优化技术

CUDA设备上的分支优化和内核调用优化方法同样可以用在GCN设备上，所以本节不再做重复的叙述。

6.2.1 访存优化

1

虽然GCN设备上一个wavefront对应的连续256字节对齐数据具有最高的传输效率（每个线程4字节），但

是当遇到计算密集型的问题时，如第一章中所讲的那样使用宽向量加载和存储操作可以具有更高的效率。

2
GCN设备上的缓存结构并不具备在wavefront线程间的广播机制，因此如果多个线程访问同一个或少数几个数据，更好的方式是通过局部内存，例如：

```
#if(get_local_id(0)<4){
    l_data=g_data[get_local_id(0)];
} barrier(CLK_LOCAL_MEM_FENCE);
```

而不是

```
data=g_data[get_local_id(0)&3]
```

2 将不同block内的全局数据访问尽量分散到不同的全局内存channel和bank中，如果多个同时进行全局内存数据访问的不同block访问的数据位于同一个channel或bank中，则内存操作会串行执行，对效率的影响很大，必要时显式的对block进行调度。

3
GCN设备上的共享内存可以不经过寄存器直接访问（有点类似fermi之前的CUDA设备），因此可以省去volatile关键字。

6.2.2 指令优化

1
由于GCN设备具有独立的标量计算单元，所以支持整数计算和浮点计算指令的双发，合理调度指令的顺序可以更好的隐藏指令的发射和计算延迟，比如将浮点计算指令和预取数据的地址计算指令交叉排列。

2
当指令中包含了一些特定的常量值时，编译器可以生成更小的代码，因为这些特殊的常量对应了指令的二进制编码中特定的几个比特位。的这些值是

0，1~64,， -1~-16， +-0.5， +-1.0，+-2.0， +-4.0，1.0/(2*PI)

对1.0/(2*PI)内嵌常量的支持更多的是考虑到诸如FFT等图像计算方面的应用，但是只有矢量指令才支持在指令码中内嵌1.0/(2*PI)常量。同时自定义的PI值可能无法匹配指令支持的值，因此最好通过使用OpenCL中的内置的定义。

3
和比fermi更早期的CUDA设备类似，目前所有GCN（1.0~1.3）设备上对24位整数乘法提供原生支持，因此使用24位整数乘法具有更高的效率。

4
对于GCN1.1， GCN1.2的设备,在诸如归约和扫描的应用中尽量使用OpenCL内置的归约和扫面函数，这样可以帮助编译器生成DPP指令从而可以使用硬件上的数据并行引擎执行跨通道计算（无需通过LDS中转）。

6.3 构建性能可移植的程序

OpenCL是为跨平台的高性能并发程序设计而制定的开放式规范，虽然理论上使用OpenCL开发的程序可以在任何支持OpenCL的平台上运行，但是实际上受限于不同平台对OpenCL支持的力度以及不同硬件架构上得差异，使得同一个OpenCL程序在两个不同的设备上的性能表现可能差别很大(甚至这两个设备在理论上的技术指标很接近)。很多实际的应用，仅仅拥有代码的可移植性是不够的，因此本章主要讨论如何利用OpenCL的运行时编译系统构建性能可移植的程序。为了写出性能可移植的程序，不仅仅需要对同一厂商的不同架构的设备做针对性的优化，同时还要针对不同厂商的设备给出不同的优化实现；这一过程虽然增加了开发的时间和难度，但是从给与用户更佳体验的角度来说是完全值得的。下面我们以并行规约为例讲解如何开发性能可移植的OpenCL程序。

6.3.1 CUDA设备上的并行规约

```
#ifdef CUDA_DEVICE
#if CUDA_SM<30
#define SMEM_SIZE 264
inline void warp_reduce_add( double& s, __local volatile double* sptr, int lane )
{
        if(lane<16)
        {
                *sptr=s; s+=*(sptr+16);
                *sptr=s; s+=*(sptr+ 8);
                *sptr=s; s+=*(sptr+ 4);
                *sptr=s; s+=*(sptr+ 2);
                *sptr=s; s+=*(sptr+ 1);
        }
}
#else
#define SMEM_SIZE 8
inline double __shfl( double val, int mask )
{
        double out;
        asm volatile ("{                                \n\t"
                ".reg.b32 slo, shi, dlo, dhi;           \n\t"
                "mov.b64 { slo, shi }, %1;              \n\t"
                "shfl.down.b32 dlo, slo, %2, 0x1f;      \n\t"
                "shfl.down.b32 dhi, shi, %2, 0x1f;      \n\t"
                "mov.b64 %0, { dlo, dhi };              \n\t"
                "}" : "=d"(out) : "d"(val), "r"(mask) );
        return out;
}
inline void warp_reduce_add( double& s )
{
        s+=__shfl(s,16);
        s+=__shfl(s, 8);
        s+=__shfl(s, 4);
        s+=__shfl(s, 2);
        s+=__shfl(s, 1);
}
```

```
#endif

#else defined(GCN_DEVICE)

#define SMEM_SIZE 260

#endif

inline void block_reduce_add( double& s, __local double* smem, int lane, int warpid )
{
#if CUDA_SM<30
	__local volatile double* sptr=&smem[get_local_id(0)];
	warp_reduce_add( s, sptr, lane );
	if( lane==0 ){
		smem[256+warpid]=s;
	} barrier(CLK_LOCAL_MEM_FENCE);
	sptr+=256;
	if(get_local_id(0)<4){
		s=*sptr; s+=*(sptr+4);
		*sptr=s; s+=*(sptr+2);
		*sptr=s; s+=*(sptr+1);
	}
#else
	warp_reduce_add( s );
	if(lane==0){ smem[warpid]=s; }
	barrier(CLK_LOCAL_MEM_FENCE);
	if(get_local_id(0)<8)
	{
		s=smem[get_local_id(0)];
		s+=__shfl(s,4);
		s+=__shfl(s,2);
		s+=__shfl(s,1);
	}
#endif
}


__kernel void kReduceAdd( __global double		* g_mapped,
			  __global double		* g_temp,
			  __global unsigned int		* g_mutex,
			  __global const double		* g_a, int n )
{
	__local unsigned int l_mutex;
	__local double l_temp[SMEM_SIZE];

	double c=0;
	int i=(get_group_id(0)<<8)+get_local_id(0);
	unsigned int stride=get_num_groups(0)<<8;

	while(i<n){ c+=g_a[i]; i+=stride; }
	const int lane=get_local_id(0)&31;
	const int warpid=get_local_id(0)>>5;
	block_reduce_add( c, l_temp, lane, warpid );

	if(get_local_id(0)==0){
```

```
            __global double* g_out=(get_num_groups(0)>1)?&g_temp[get_group_id(0)]:g_mapped;
            *g_out=c;
        }

        if(get_num_groups(0)>1)
        {
            barrier(CLK_GLOBAL_MEM_FENCE);
            if(get_local_id(0)==0){
                    l_mutex=atom_add( &g_mutex, 1 );
            } barrier(CLK_LOCAL_MEM_FENCE);
            if(l_mutex==(get_num_groups(0)-1))
            {
                    c=(get_local_id(0)<get_num_groups(0))?g_temp[get_local_id(0)]:0;
                    block_reduce_add( c, l_temp, lane, warpid );
                    if(get_local_id(0)==0){
                            g_mapped[0]=c; g_mutex=0;
                        }
                }
        }
}
```

首先看warp_reduce_add函数，我们使用指针操作，并把访问共享内存的次数降到了最小，如过在主函数内改成 smem[threadIdx.x]=c然后将warp_reduce_add和block_reduce_add改成如下形式

```
        inline void warp_reduce_add( __local volatile double* sptr )
        {
                if(lane<16)
                {
                        sptr[0]+=sptr[16];
                        sptr[0]+=sptr[ 8];
                        sptr[0]+=sptr[ 4];
                        sptr[0]+=sptr[ 2];
                        sptr[0]+=sptr[ 1];
                }
        }
        inline double block_reduce_add( double* smem, int lane, int warpid )
        {
                volatile double* sptr=&smem[threadIdx.x];
                warp_reduce_add( s, sptr, lane );
                if( lane==0 ){
                        smem[256+warpid]=s;
                } barrier(CLK_LOCAL_MEM_FENCE);
                sptr+=256;
                if(get_local_id(0)<4){
                        sptr[0]+=sptr[ 4];
                        sptr[0]+=sptr[ 2];
                        sptr[0]+=sptr[ 1];
                }
                return smem[0];
        }
```

那么会多出9次共享内存的访问操作。同时注意到在不支持warp shuffle操作的设备上每个block我们多分配了64字节（SMEM_SIZE=256+8）的共享内存，这是为了减少快内同步而做的优化，如果不加上这额外的共享内存，那么我们必须像下面这样多加一次同步：

```
        warp_reduce_add( s, sptr, lane );
        barrier(CLK_LOCAL_MEM_FENCE);
        if( lane==0 ){
                smem[warpid]=s;
        } barrier(CLK_LOCAL_MEM_FENCE);
```

如果warp_reduce_add之后不加同步的话，那么就无法保证来自其他warp的第一个线程对共享内存写入之前第一个warp的计算已经完成，就会有写冲突的问题。这一优化策略对于GCN设备上的实现同样适用，这其实也是对双缓冲技术的一个变相应用。

6.3.2 GCN设备上的并行规约

GCN设备上的实现和CUDA设备上的实现很相似，稍微的不同在CUDA设备上锁步计算的粒度是32，而GCN设备上的锁步粒度则是64(和CUDA设备上基于显式的warp并发一样，单个wavefront内的线程是以锁步的方式并行执行的，因此无需同步)，因此为了在GCN架构上具有更好的亲和性，我们需要将锁步并发粒度改成64。

```
inline void wavefront_reduce_add( double& s, __local double* sptr, int lane )
{
        if(lane<32)
        {
                *sptr=s; s+=*(sptr+32);
                *sptr=s; s+=*(sptr+16);
                *sptr=s; s+=*(sptr+ 8);
                *sptr=s; s+=*(sptr+ 4);
                *sptr=s; s+=*(sptr+ 2);
                *sptr=s; s+=*(sptr+ 1);
        }
}
inline void block_reduce_add( double& s, __local double* smem, int lane, int wavefront_id )
{
        __local double* sptr=&smem[get_local_id(0)];
        wavefront_reduce_add( s, sptr, lane );
        if( lane==0 ){
                smem[256+wavefront_id]=s;
        } barrier(CLK_LOCAL_MEM_FENCE);
        sptr+=256;
        if(get_local_id(0)<2){
                *sptr=s; s+=*(sptr+2);
                *sptr=s; s+=*(sptr+1);
        }
}
```

读者可能注意到了在针对GCN设备的实现中，并没有volatile关键字。这是因为GCN设备上的LDS(Local Data Share，相当于CUDA设备上的共享内存)可以不经过寄存器直接存取数据，这一点和fermi架构之前的CUDA设备类似；这样每次访问LDS中的数据时都会从LDS中读或写，编译器也不会自作聪明去猜测程序员的意图以免造成不必要的麻烦。

```
__kernel void kReduceAdd( __global double          * g_mapped,
                          __global double          * g_temp,
                          __global unsigned int    * g_mutex,
                          __global const double    * g_a, int n )
```

```
{
	__local unsigned int l_mutex;
	__local double l_temp[SMEM_SIZE];

	double c=0;
	int i=(get_group_id(0)<<8)+get_local_id(0);
	unsigned int stride=get_num_groups(0)<<8;

	while(i<n){ c+=g_a[i]; i+=stride; }
	const int lane=get_local_id(0)&63;
	const int wavefront_id=get_local_id(0)>>6;
	block_reduce_add( c, l_temp, lane, wavefront_id );

	if(get_local_id(0)==0){
		__global double* g_out=(get_num_groups(0)>1)?&g_temp[get_group_id(0)]:g_mapped;
		*g_out=c;
	}

	if(get_num_groups(0)>1)
	{
	    barrier(CLK_GLOBAL_MEM_FENCE);
	    if(get_local_id(0)==0){
			l_mutex=atom_add( &g_mutex, 1 );
	    } barrier(CLK_LOCAL_MEM_FENCE);
	    if(l_mutex==(get_num_groups(0)-1))
	    {
			c=(get_local_id(0)<get_num_groups(0))?g_temp[get_local_id(0)]:0;
			block_reduce_add( c, l_temp, lane, wavefront_id );
			if(get_local_id(0)==0){
				g_mapped[0]=c; g_mutex=0;
			    }
		}
	}
}
```

现在，我们需要将所有的版本整合进同一个**OpenCL**内核程序中，这个可以简单的通过预处理指令来实现

```
#define CUDA_GPU		0

#define GCN_GPU		1

#pragma OPENCL EXTENSION cl_khr_fp64:enable

#if DEVICE == CUDA_DEVICE

	包含CUDA设备版本的代码

#elif DEVICE == GCN_GPU

	包含GCN设备版本的代码

#elif DEVICE == ...

	包含针对其它设备优化的版本
```

…

#endif

'DEVICE'以及'CUDA_SM'并没有在设备代码端定义，所以我们需要将它们作为命令行参数传递给OpenCL运行时编译系统，这样就可以让OpenCL驱动程序在运行时根据当前的设备选择合适的版本进行编译，从而在多个不同设备上都可以获得很高的性能

sprintf (options, "-DDEVICE=dev_type –DCUDA_SM=*cc*", …);

clBuildProgram(prog, options, …);

小结

GPU计算优化技术和方法种类繁多，每个人也可能习惯与自己的方法。但总体来说不会有太多不同，即使对于不同架构的设备很多优化技术也依然是通用的。开发性能可移植的程序从某种意义上来说不仅仅是一种挑战，也是对程序员技能的一种考验和磨练；从用户的角度考虑，他们是性能可移植性的程序的最终受益者，因此这样做也就更具有现实意义。有些优化技术在一些情况下可能适用，而另一些情况则可能适得其反，因此实际的验证必不可少。

参考资料

1《CUDA Programming Guide》

2《AMD GCN Architecture Whitepaper》

3《AMD Southern Island Series Instruction Set Architecture》

4《Graphics Core Next Architecture，Generation 3》

5《AMD Parallel Processing OpenCL Programming Guide》