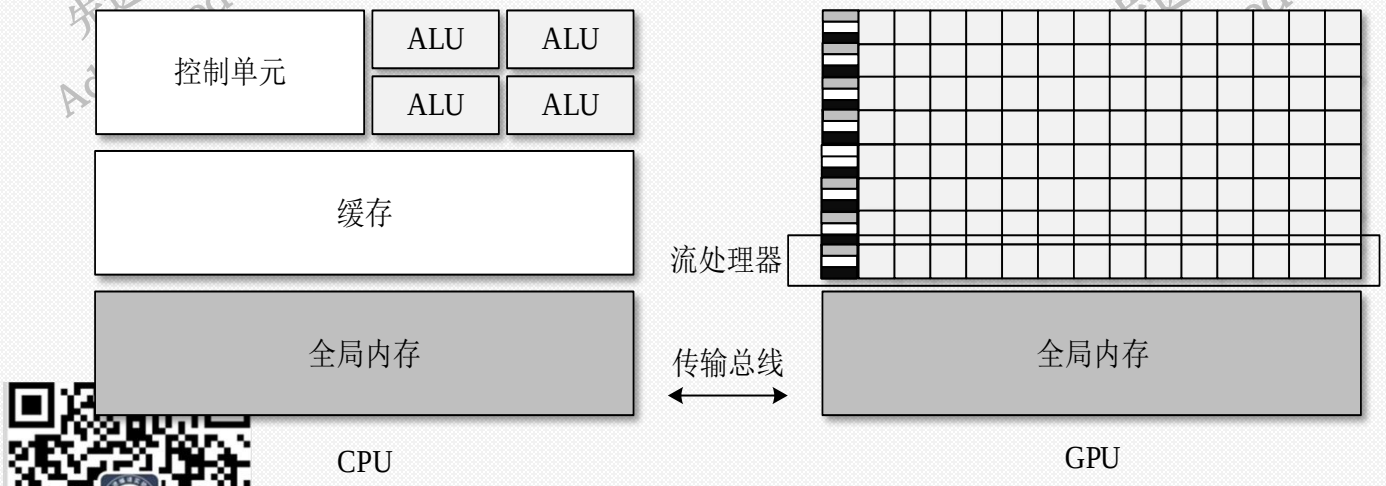
第十章 CUDA程序优化

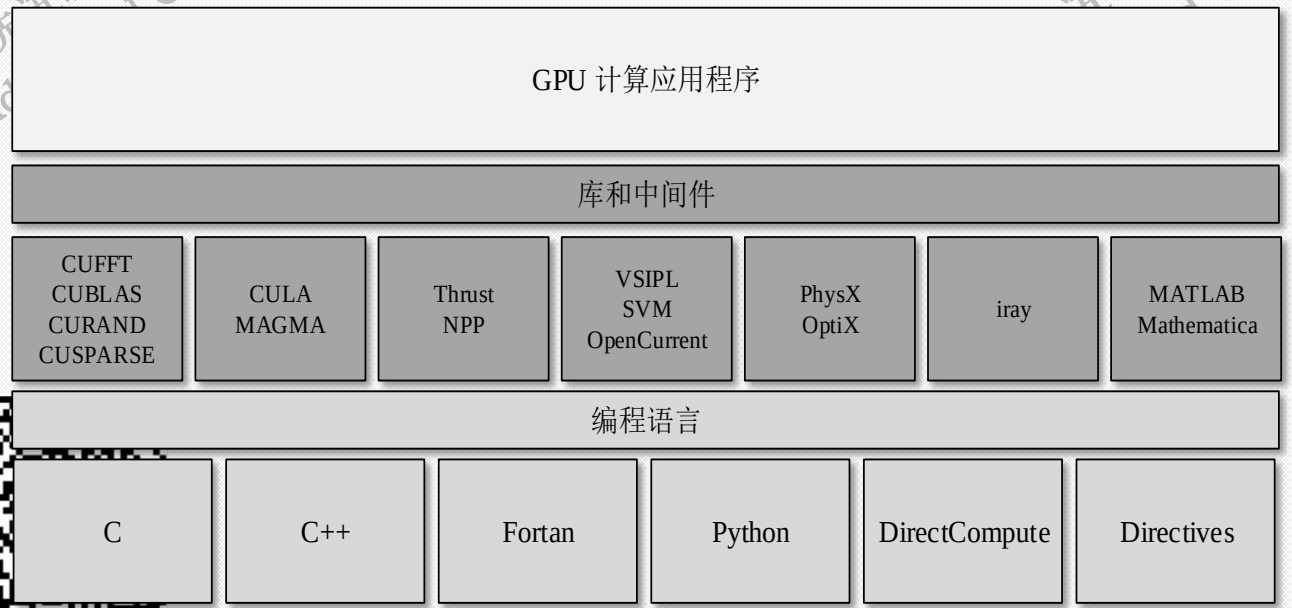
10.1 CUDA编程简介

- CUDA是什么
- CUDA编程模型
- CUDA程序编写
- CUDA版矩阵乘

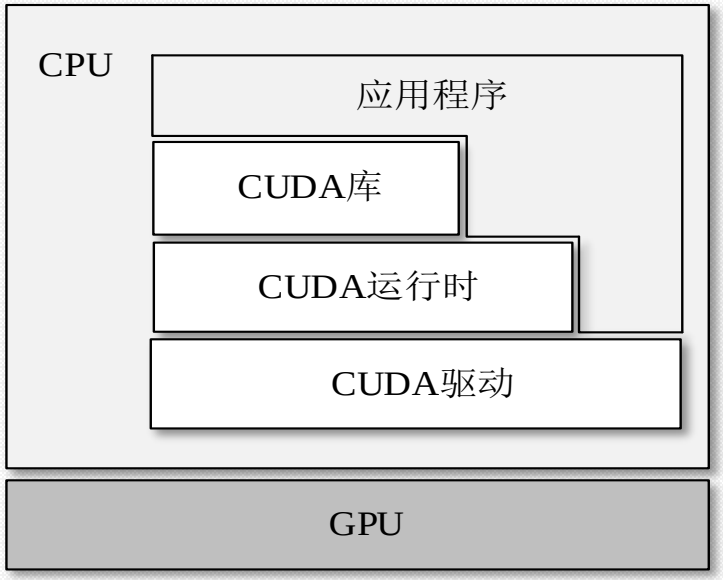
统一计算设备架构CUDA是NVIDIA提出的通用并行计算平台和编程模型，为使用GPU的异构计算开发提供了便捷高效的开发环境。异构计算采用并行或分布式计算方式，通过协调地使用性能、结构各异的计算器件以满足不同的计算需求，由CPU处理器与众核GPU可组成一个典型的异构计算架构。如图所示。



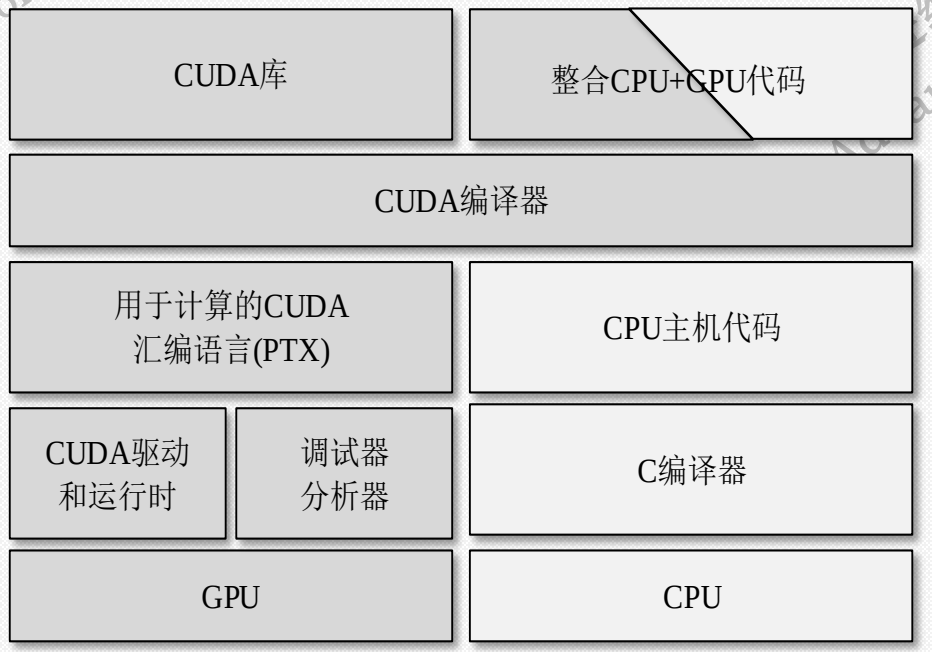
CUDA平台支持开发者使用C/C++、FORTRAN、Python等行业标准程序语言的扩展来构建CUDA程序，同时CUBLAS、Thrust等丰富的CUDA加速库也为CUDA程序的开发提供了便利，如图所示。CUDA C是标准ANSI C语言的一个扩展，被广泛应用于各领域CUDA程序的开发，本章后续范例将统一使用CUDA C进行编写。



CUDA提供了两层应用程序接口API来管理GPU设备，分别是CUDA驱动和CUDA运行时，如图所示。CUDA驱动能够细致全面的控制GPU设备的运行状态，但使用驱动API编程的难度较大。CUDA运行时作为更高级的API实现在CUDA驱动API的上层，使用运行时API能够简化管理GPU设备的操作、降低编程难度。



CUDA程序用NVCC编译器进行编译，其编译流程自上而下如图10.4所示，NVCC编译器在编译过程中会将主机代码与设备代码进行分离，经过代码分离后，使用C语言编写的主机端代码将由本地C语言编译器进行编译，使用CUDA C语言编写的设备端代码会通过NVCC编译器进行编译。



一个典型的CUDA程序实现流程如下：

（1）获取GPU设备

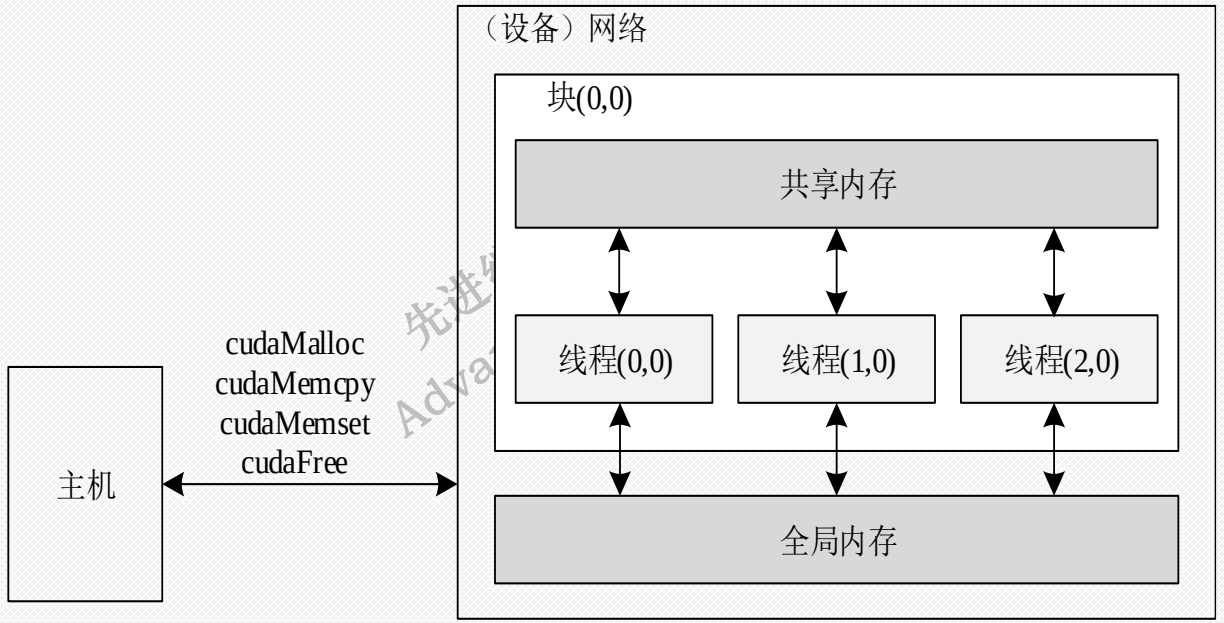
（2）开辟GPU上显存空间

（3）发起主机向设备的数据传输

（4）启动核函数

（5）发起设备向主机的数据传输

（6）释放GPU的显存空间，重置设备



与实现流程对应的CUDA程序主要代码如下。在编写CUDA程序时，通过调用CUDA运行时的cudaMalloc、cudaFree等函数能够显式地控制GPU设备进行内存开辟与内存释放；通过调用cudaMemcpy函数能够控制CUDA程序中主机端与设备端的数据传输；使用语句kernel_name <<<grid,block>>>能够实现对核函数的调用；通过调用cudaDeviceResset函数能够对GPU设备进行重置。

```
cudaSetDevice(0);
cudaMalloc((void**) &d_a, sizeof(float) * n);
cudaMemcpy(d_a, a, size_t count,
cudaMemcpyHostToDevice);
kernel<<<blocks,threads>>>;
cudaMemcpy(a, d_a, size_t count,
cudaMemcpyDeviceToHost);
cudaFree(d_a);
cudaDeviceReset();
```

下面概要地对一些常用CUDA运行时函数进行介绍，常用的设备管理类函数有：

cudaError_t cudaGetDeviceCount(int* count)//用来获取当前系统中可用GPU设备的数量
cudaSetDevice (int *device)//用来在系统中选择希望调用的GPU设备
cudaDeviceReset(void)//用来显式销毁和清理当前GPU设备上的所有资源
cudaDeviceSynchronize(void)//用来显式地阻塞主机端进程直至系统中的GPU设备完成其上的计算任务

常用的内存管理类函数有：

cudaMalloc (void** devptr, size_t size)//用来在GPU设备上分配一定字节的线性内存，并以devPtr的形式返回指向所分配内存的指针

cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)//用来实现主机端与设备端数据传输

cudaMallocPitch(void** devPtr, size_t* pitch, size_t width, size_t height)//用来在GPU设备上分配线性内存，并以*devPtr的形式返回指向所分配内存的指针

cudaMemcpy2D(void* dst, size_t* pitch, const void* src, size_t width, size_t height, cudaMemcpyKind kind)//针对cudaMallocPitch在GPU设备上分配内存后，进行主机端到设备端的数据传输

cudaFree(void *devPtr)//释放devPtr指向的由cudaMalloc()调用开辟的GPU上内存空间

设备核函数：

__global__ void kernel_name (argument list)//设备端执行的代码称为核函数，在程序中使用__global__声明定义，函数返回类型必须为void类型

错误处理函数：

const char* cudaGetErrorString (cudaError_t error)//将CUDA程序运行时产生的错误信息error进行转化为可读的错误信息

下面将以CUDA向量相加为例完整展示CUDA程序的编写过程，一个在主机端执行的向量相加函数的代码如下所示。

```
void sumArraysOnHost(float *A, float *B, float *C, const int N){
    for (int idx = 0; idx < N; idx++){
      C[idx] = A[idx] + B[idx];
    }
}
```

该函数将两个大小为N的向量A和B相加，通过N次循环实现计算操作，该函数对应的CUDA核函数代码如下。

```
__global__ void sumArraysOnGPU(float *A, float *B, float *C, const int N){
    int tx = threadIdx.x;
    C[tx] = A[tx] + B[tx];
}
```

CUDA向量相加的实现流程如下：

（1）使用cudaGetDeviceProperties 获取GPU设备

（2）完成数组A和数组B的初始化

（3）使用cudaMalloc函数开辟用于存储数组A、B、C元素的GPU内存空间d_A、d_B和d_C

（4）使用cudaMemcpy控制CPU端向GPU端的传输数组A、B的元素

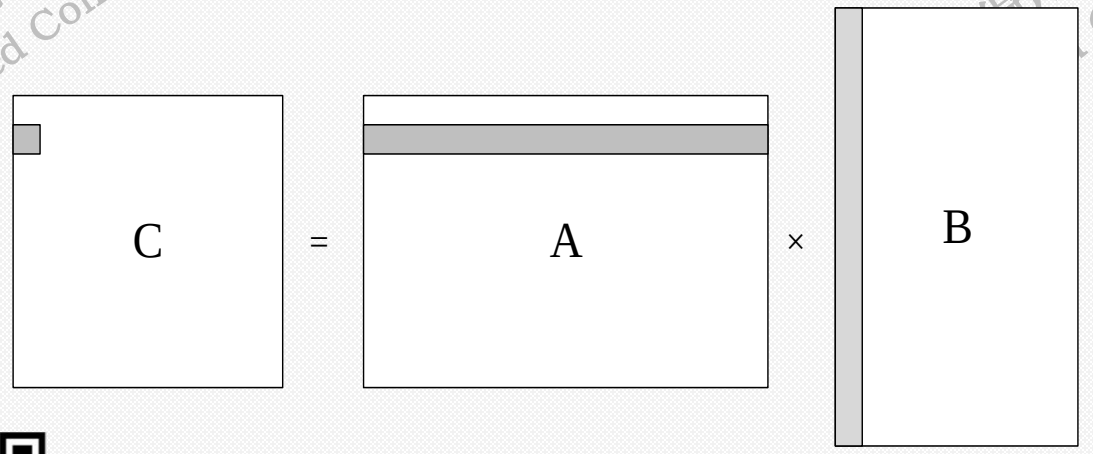
（5）启动核函数sumArraysOnGPU在GPU上进行数组相加运算

（6）使用cudaMemcpy控制GPU端向CPU端传输结果数组C的元素

（7）验证CUDA数组相加的正确性

（8）释放GPU的显存空间，重置设备

矩阵乘法作为一种基本的数学运算，在科学计算领域有着非常广泛的应用，矩阵乘法的快速算法对科学计算有着极为重要的意义。一般矩阵乘法的实现思想如图10.6所示，阵A中的一行和矩阵B中的一列进行向量内积得到矩阵C中的一个元素。



使用C语言对一般矩阵乘法进行实现，其中A、B矩阵与结果矩阵C均为长宽是width的方阵。部分核心代码如下。

```
void MatrixMulOnHost(float *A, float *B, float *C, int width){
    for(int i=0; i<width; i++){
        for(int j=0; j<width; j++){
            float sum = 0.0;
            for(int k=0; k<width; k++){
                float a = A[i*width+k];
                float b = B[k*width+j];
                sum+=a * b;
            }
            C[i*width+j]=sum;
        }
```

通过上面对CUDA编程的介绍，对于一个结果矩阵规模为width*width的一般矩阵乘法，可以启用width*width个线程，每个线程负责计算结果矩阵中的一个元素，其核函数代码如下。

```
__global__ void MatrixMulKernel(float* Ad, float* Bd, float* Cd, int width){
        int offset = threadIdx.x;
        int row = offset /width;
        int col = offset %(width-1);
        float sum = 0;
        for(int i=0;i<width;i++){
            sum += Ad[row*width+i]*Bd[i*width+col];
        }
        Cd[row*width+col] = sum;
}
```


使用NVCC对CUDA矩阵乘的代码进行编译并运行测试，编译命令为：nvcc matrixmul.cu -o matrixmul，测试环境中GPU设备为NVIDIA RTX 3090，CUDA版本为11.6，使用性能分析工具Nsight System对核函数进行计时用于性能监测，执行命令为：nsys profile --stats=true ./matrixmul，测试结果如表所示。

函数名称	矩阵规模	线程布局	运行时间(us)
MatrixMulOnHost	32*32	—	262
MatrixMulKernel	32*32	(1,1024)	5.18

10.2 线程结构优化

- 线程组织优化
- 线程布局优化

在进行CUDA程序的编写时，可以考虑通过优化线程的组织方式，在核函数的构建时将负责执行计算任务的thread划分至多个block上实现，利用多个SM处理器提高任务的并行程度，从而提升GPU设备的利用率，实现对CUDA程序的优化。

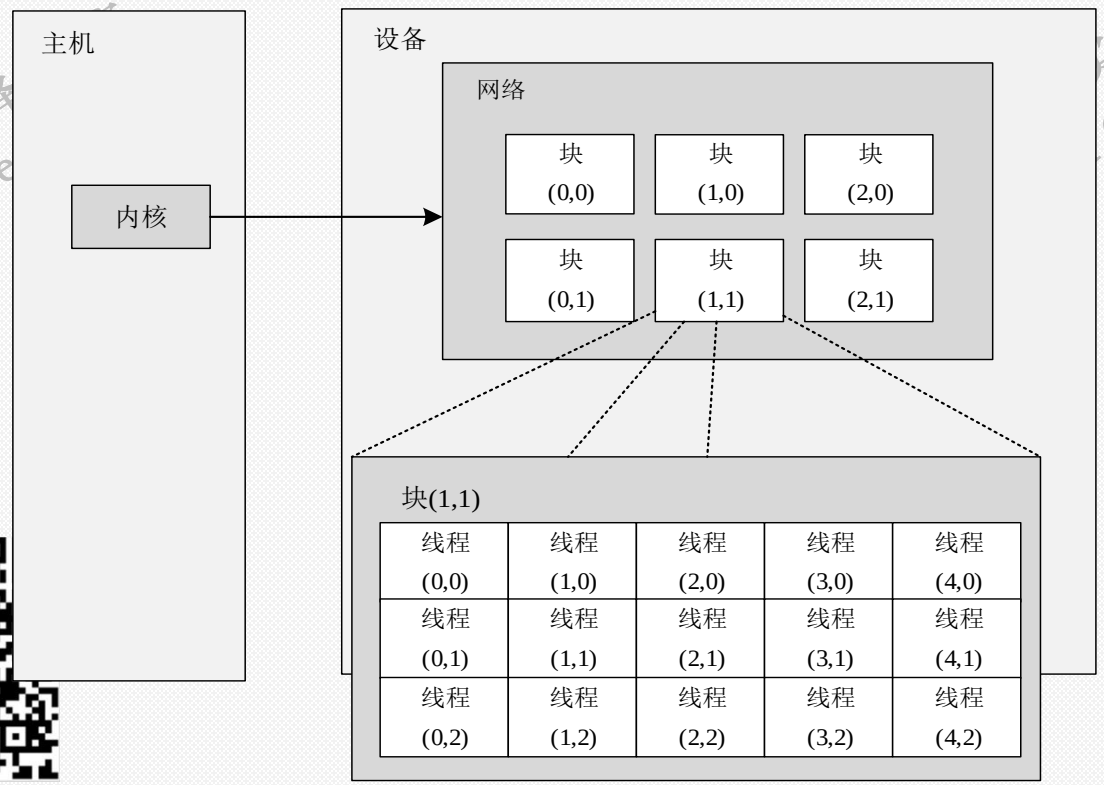
例如CUDA矩阵乘的实现中，核函数仅开启了一个block用于计算结果矩阵上的全部元素，多数SM器件处于闲置状态，GPU利用率较低。可以对其进行线程组织的优化，通过开启多个block，每个block中的线程负责计算结果矩阵上的部分元素达到提升效率的目的。

```
__global__ void MatrixMulKernel_multiblock(float* Ad, float* Bd, float* Cd, int width){
    int tx = threadIdx.x + blockIdx.x * blockDim.x;
    int row = tx / width;
    int col = tx % (width - 1);
    float sum = 0;
    for (int k = 0; k < width; k++){
            sum += Ad[row * width + k] * Bd[k * width + col];
    }
    Cd[row * width + col] = sum;
}
```

在主机端函数中通过修改变量grid，可以调整CUDA矩阵乘实现中开启的线程块数目。在矩阵规模为32*32以及64*64的情况下，对经过线程组织优化的CUDA矩阵乘代码进行测试，编译使用命令：nvcc multi_block.cu -o multi_block。并利用性能分析工具Nsight System监测性能变化情况，使用命令nsys profile –stats=true ./ multi_block，测试结果如下表所示。

	函数名称	线程布局	时间
32*32	MatrixMulOnHost	—	70
	Kernel_MultiBlock	(1,1024)	5.12us
		(2,512)	3.90us
		(4,256)	3.74us
		(8,128)	3.48us
		(16,64)	3.36us
		(32,32)	3.65us
64*64	MatrixMulOnHost	—	639us
	Kernel_MultiBlock	(4,1024)	7.68us
		(8,512)	5.47us
		(16,256)	4.73us
		(32,128)	4.70us
		(64,64)	5.05us
1024*1024	MatrixMulOnHost	—	5.32s
	Kernel_MultiBlock	(1024,1024)	1.86s
		(2048,512)	1.73s

blockDim和gridDim是描述线程块和网格各自维度的内置变量，其中blockDim表示线程块的维度，其大小为线程块中的线程的数量；gridDim表示网格的维度，其大小为网格中的线程的数量。下图展示了一个二维线程块和二维网格的线程层次结构。



通过核函数代码可以发现，相较于线程组织优化的核函数MatrixMulKernel_multiblock，该核函数中使用内置的二维坐标变量blockIdx.x、blockIdx.y、threadIdx.y、threadIdx.y、blockDim.x、blockDim.y来建立线程至目标元素的映射是更为直观高效方法。主机端代码基本与10.2.1节保持一致，只需将线程布局由一维变成二维，更改后线程布局如下。

```
__global__ void MatrixMulKernel_2DGrid2DBlock(float* Ad, float* Bd, float* Cd, int width){
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0;
    for (int k = 0; k < width; k++){
        sum += Ad[row * width + k] * Bd[k * width + col];
    }
    Cd[row * width + col] = sum;
}
```

通过核函数代码可以发现，相较于线程组织优化的核函数MatrixMulKernel_multiblock，该核函数中使用内置的二维坐标变量blockIdx.x、blockIdx.y、threadIdx.y、threadIdx.y、blockDim.x、blockDim.y来建立线程至目标元素的映射。主机端代码基本与10.2.1节保持一致，只需将线程布局由一维变成二维，更改后线程布局如下。

```
/*--------------------设置线程布局----------------------------------*/
dim3 block(32,32);
dim3 grid((Width+block.x-1)/block.x,(Width+block.y-1)/block.y);
```

进一步增大矩阵乘的数据规模至1024*1024，对使用二维线程块的CUDA矩阵乘进行测试，通过在主机端代码中调整block与grid的维度观察线程布局对CUDA矩阵乘性能的影响，调整过程中不改变grid内block数目以及block内thread的数目，测试结果如下表所示。

矩阵规模	函数名称	线程布局	时间(s)
1024*1024	MatrixMulOnHost	—	5.32
	Kernel_MultiBlock	(1024,1024)	1.86
	Kernel_2Dgrid2Dblock	((1,1024),(1024,1))	1.8564
		((2,512),(512,2))	1.0251
		((4,256),(256,4))	1.0201
		((8,128),(128,8))	1.0202
		((16,64),(64,16))	1.0196
		((32,32),(32,32))	1.0254
		((64,16),(16,64))	1.0308
		((128,8),(8,128))	1.3301
		((256,4),(4,256))	2.2937
		((512,2),(2,512))	4.3188
		((1024,1),(1,1024))	8.5023

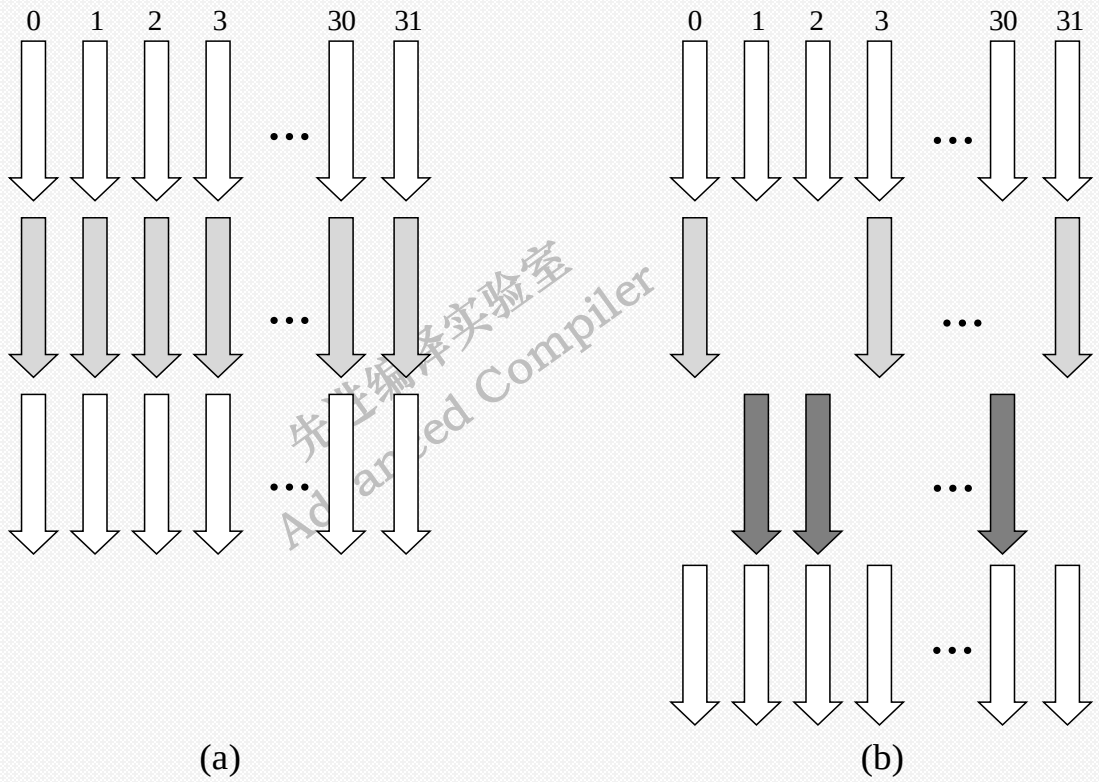
10.3 分支优化

- 基本原理
- 代码实现
- 性能分析

先进编译实验室
Advanced Compiler

GPU内硬件调度的最小并行单位是线程束（warp），它由32个线程组成，流式多处理器SM一般由一个或者多个线程束组成，由于GPU上没有复杂的分支预测单元，线程束内的线程以单指令流多线程SIMT（Single Instruction Multiple Threads）的方式执行，即一个线程束中的32个线程同时执行相同的指令，若核函数执行过程中存在条件分支语句，线程束中的线程按顺序串行通过多条分支路径。

如图中(a)所示，核函数执行过程中不存在分支路径时，较好的利用了多线程资源。但图(b)中，线程束内线程执行过程中存在2条分支路径，若任意线程进入到首个分支路径，线程束中其余的线程都处于等待状态，直到该线程执行完分支程序，当所有分支路径都执行完之后，线程束中的所有线程才会回到同一条执行路径上，该现象被称为线程束分化，线程束分化的出现会削弱线程束执行过程中的并行性，降低线程的活跃度，从而影响CUDA的程序性能。



并行归约求解是一个极为常见的问题，它通过计算把多个数据结果归约为一个最终结果，使用CUDA实现并行归约基本流程如下：

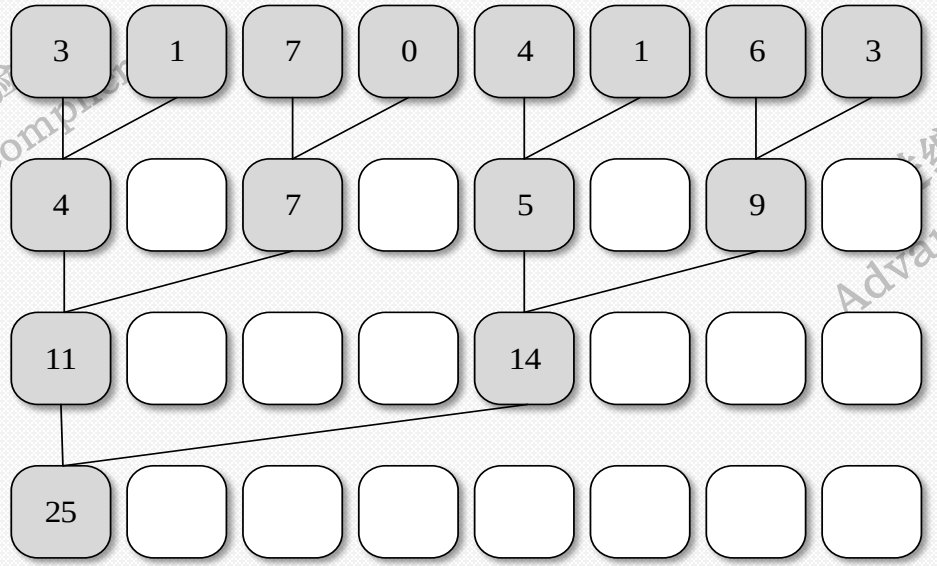
（1）把数据集合划分为较大的数据块，完成线程块与较大数据块的一一对应。

（2）把较大的数据块再划分为更小的数据块，完成线程块内每个线程与更小数据块的一一对应。

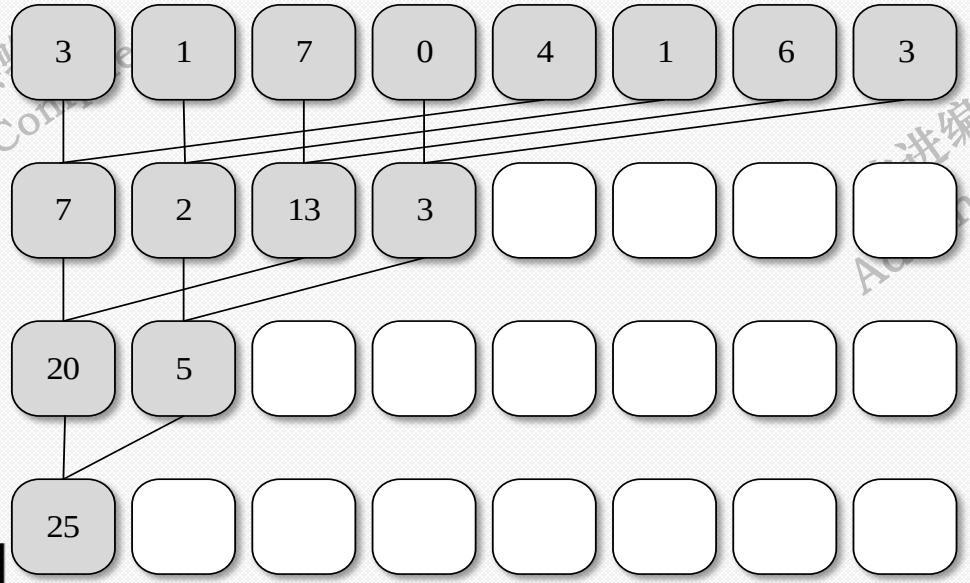
（3）完成对整个数据集合的处理，求出最终结果。

在进行并行归约计算时，常使用迭代成对实现的方法，即一个线程对两个元素求和产生一个局部结果，并将计算得到的局部结果作为下一次迭代的输入值，根据线程所取元素位置的不同，有相邻配对和交错配对两种方法。

相邻配对法如图所示，使用该配对方法的CUDA并行归约计算中，一个线程对相邻的两个元素进行求和操作。



交错配对法如图所示，使用该配对方法的CUDA并行归约计算中，一个线程对具有固定跨度的两个元素进行求和操作。



使用相邻配对法的CUDA并行归约核函数代码如下所示。

```
__global__ void reduce_GPU (int * g_idata, int * g_odata, unsigned int n){
        //set thread ID
        unsigned int tid = threadIdx.x;
        if (tid >= n) return;
        int *idata = g_idata + blockIdx.x*blockDim.x;
        for (int stride = 1; stride < blockDim.x; stride *= 2){
                if ((tid % (2 * stride)) == 0){
                        idata[tid] += idata[tid + stride];
                }
                __syncthreads();
        }
        if (tid == 0)
                g_odata[blockIdx.x] = idata[0];
}
```

对使用相邻配对的CUDA并行归约与CPU串行归约的性能进行初步测试比较，CPU串行归约实现的代码与调用核函数reduce_GPU的代码如下所示。

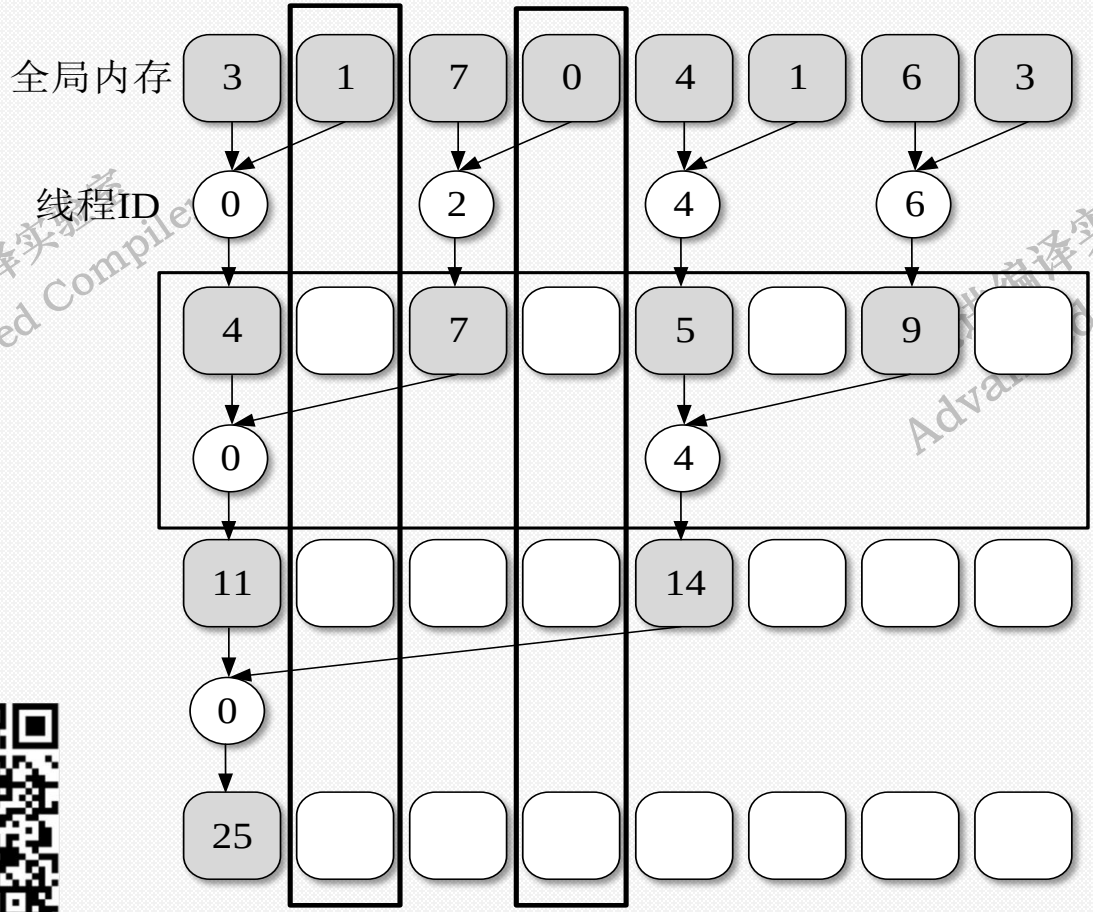
```
int reduce_CPU(int *data, int const size){
    int cpu_sum = 0;
    for (int i = 0; i < size; i++)
        cpu_sum += data[i];return cpu_sum;
}
                int blocksize = 1024;
                dim3 block(blocksize, 1);
                dim3 grid((size - 1) / block.x + 1, 1);
                printf("grid %d block %d \n", grid.x, block.x);
                iStart = cpuSecond();
                reduce_GPU <<<grid, block >>>(idata_dev, odata_dev, size);
                cudaDeviceSynchronize();
                iElaps = cpuSecond() - iStart;
                cudaMemcpy(odata_host, odata_dev, grid.x * sizeof(int), cudaMemcpyDeviceToHost);
                gpu_Sum = 0;
                for (int i = 0; i < grid.x; i++)
                    gpu_Sum += odata_host[i];
                printf("reduce_GPU elapsed %lf ms gpu_Sum:%d<<<grid %d block %d>>>\n",iElaps, gpu_Sum,
                grid.x, block.x);
```

程序中输入数组的数据规模为int size = 1 << 24，内核配置为一维网格和一维块，线程块内线程数为1024，CPU版串行归约和CUDA版并行归约程序的运行时间如表所示。

函数名称	时间(us)
reduce_CPU	2154.11
reduce_GPU	393.23

从表中的测试结果可以看出，CUDA并行归约相较于CPU串行归约性能提升明显，但通过观察核函数reduce_GPU代码可以发现，由于采取相邻配对的方法，核函数中使用了条件执行语句if ((tid % (2 * stride)) == 0)，该语句使得CUDA并行归约的执行过程中出现了不同计算分支，进而导致了线程束分化现象。

执行过程中分支的具体情况如下图所示。

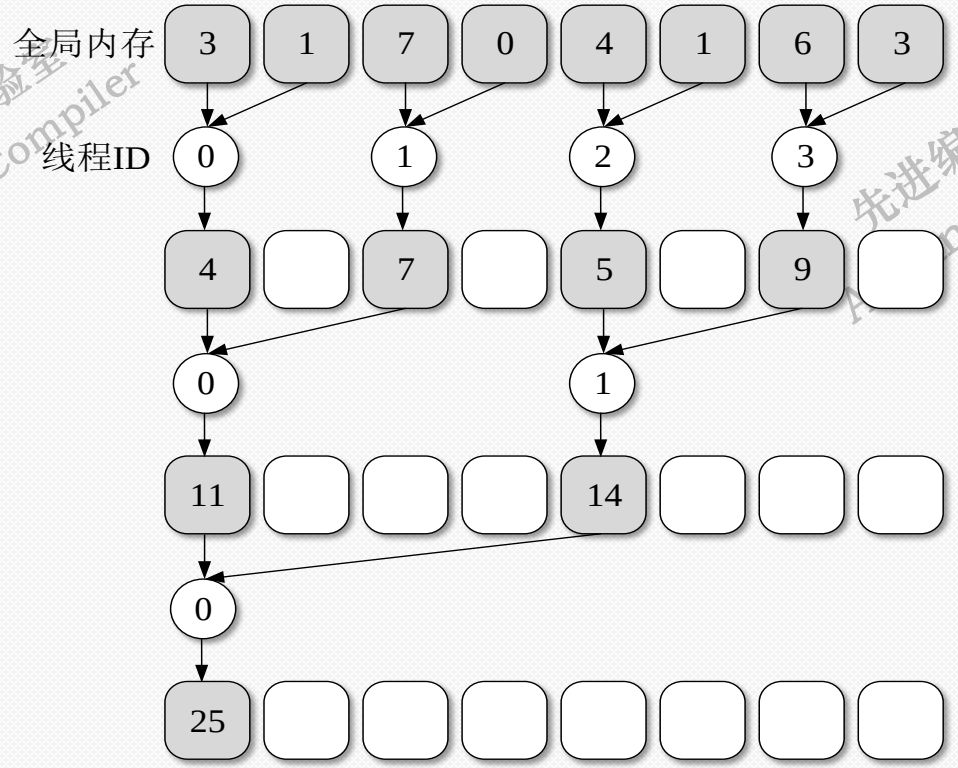


对核函数reduce_GPU执行过程进行分析发现，随着迭代次数的增加，线程束分化现象愈发严重，分支的存在对程序性能会造成不良的影响。此时可以通过采用交错配对的方法解决核函数reduce_GPU中因分支造成的性能损耗，使用交错配对的CUDA并行归约实现核函数reduceNeighboredLess代码如下。

```
__global__ void reduceNeighboredLess(int * g_idata,int *g_odata,unsigned int n){
        unsigned int tid = threadIdx.x;
        unsigned idx = blockIdx.x*blockDim.x + threadIdx.x;
        int *idata = g_idata + blockIdx.x*blockDim.x;
        if (idx > n)return;
        for (int stride = 1; stride < blockDim.x; stride *= 2){
                int index = 2 * stride *tid;
                if (index < blockDim.x){
                        idata[index] += idata[index + stride];
                }
                __syncthreads();

        if (tid == 0)
                g_odata[blockIdx.x] = idata[0];
}
```

通过采用交错配对的方法，从而使得CUDA并行归约实现时线程获取成对元素的方式如下图所示，交错配对的方法保证了线程束中线程在多轮迭代后依然保持着较高的线程利用率，改善了使用相邻配对CUDA并行归约中的线程束分化情况。



对采用相邻配对的CUDA并行归约reduce_GPU和采用交错配对的CUDA并行归约reduceNeighboredLess进行测试，验证分支优化的效果。测试过程中，使用NVCC编译器进行编译的命令为nvcc reduce.cu -o reduce，使用Nsight System工具监测核函数运行时间的命令为nsys profile –stats=true ./reduce，运行结果如下表所示。可以看出改善并行归约分支的代码核函数reduceNeighboredLess比ArraySum_GPU01运行时间快了1.7倍。

函数名称	时间(us)
reduce_CPU	2154.11
reduce_GPU	393.23
reduceNeighboredLess	227.13

借助Nsight Compute工具对核函数进行性能分析，通过ncu –metrics l1tex__t_bytes_pipe_lsu_mem_global_op_ld.sum.per_second参数选项对核函数的运行内存加载吞吐量进行检测，使用命令ncu –metrics l1tex__t_bytes_pipe_lsu_mem_global_op_ld.sum.per_second ./reduce，测试结果如下表所示。

Type	指标参数	函数名称	内存吞吐量(GB/s)
GPU	l1tex__t_bytes_pipe_lsu_mem_global_op_ld.sum.per_second	reduce_GPU	583.35
	l1tex__t_bytes_pipe_lsu_mem_global_op_ld.sum.per_second	reduceNeighboredLess	1050.01

在Nsight Compute工具中可以使用参数smsp__average_inst_executed_per_warp.ratio检测核函数中线程束的指令数目，检测结果如下表所示。

Type	参数指标	函数名称	指令数
GPU	smsp__average_inst_executed_per_warp.ratio	reduce_GPU	317.94
	smsp__average_inst_executed_per_warp.ratio	reduceNeighboredLess	124.19

核函数reduceNeighboredLess通过分支优化消除了大量的分支判断指令，进而减少了因分支判断带来的大量时延。CUDA并行归约的示例说明了通过对CUDA程序进行分支优化，能够避免或大幅减少线程束分化对程序带来的性能损耗，从而实现对CUDA程序的性能优化。

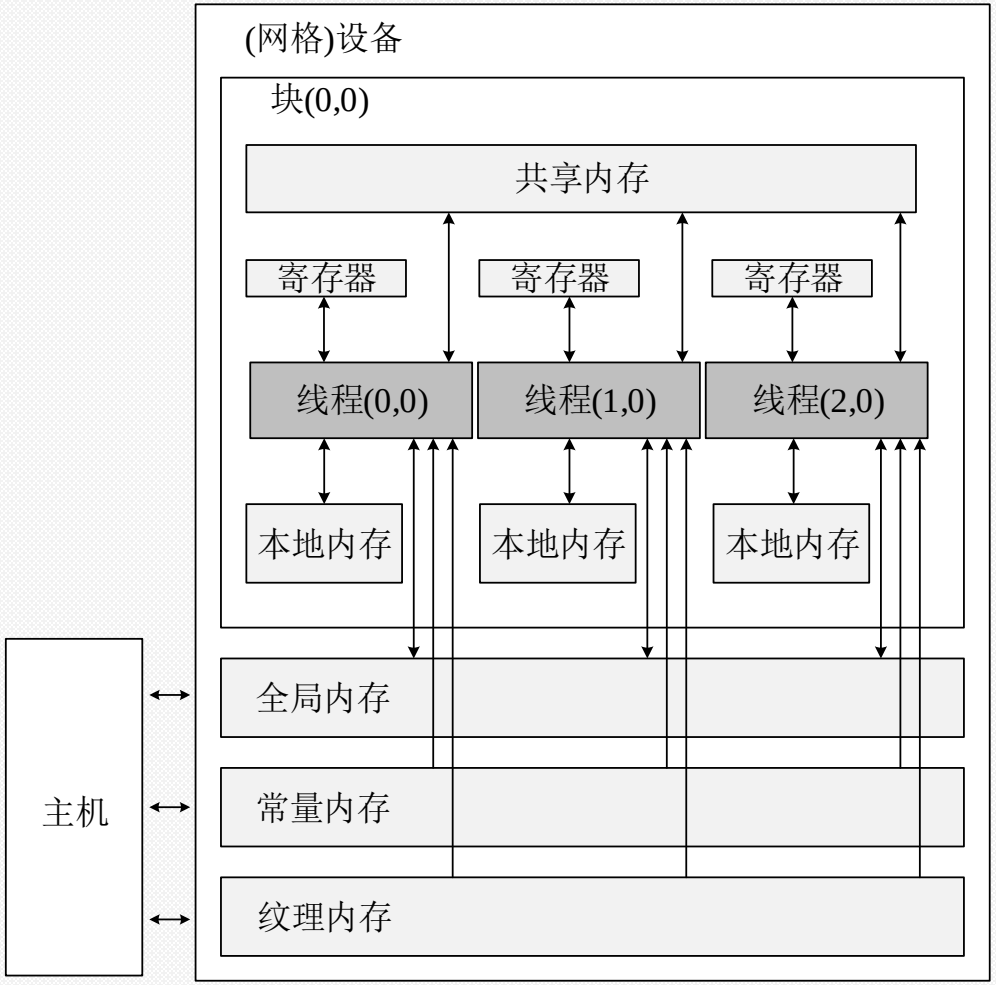
10.4 访存优化

- 全局内存优化
- 共享内存优化
- 避免bank冲突
- 高速缓存优化

先进编译实验室
Advanced Compiler

10.4 访存优化

CUDA的存储层次包括寄存器、共享内存、本地内存、常量内存、纹理内存、全局内存等，图中描述了CUDA内存空间的层次结构。不同的存储层次存在不同的作用域、生命周期和缓存行为。本节将从全局内存、共享内存、bank冲突以及高速缓存四个方面描述如何对CUDA程序进行访存优化。



在CUDA执行模型中，对全局内存的访存指令以线程束为单位，并通过缓存来实现加载或存储执行，为了提高CUDA程序对全局内存的访存效率，需要关注两个特性：

（1）合并内存访问，当一个线程束中全部32线程访问一个连续的内存块时，即可达成合并内存访问。

（2）对齐内存访问，当线程束执行内存事务的目标首地址为设备缓存粒度（32字节的二级缓存或128字节的一级缓存）的整数倍时，即可达成对齐内存合并。

在GPU设备上，对全局内存的访问通常需要几百个时钟周期，而执行一次计算操作只需要几个时钟周期，因此除了通过合并对齐提高对全局内存的访问效率之外，提升核函数的计算访存比，复用取自全局内存的数据对提升CUDA程序的性能同样至关重要。

对经过线程结构优化的CUDA矩阵乘的核函数Kernel_2Dgrid2Dblock进行分析，矩阵的数据规模和最优配置线程块的维度均为2的整次幂时，能够实现对全局内存的对齐合并访存。但观察核函数中的核心计算代码sum += Ad[row * width + k] * Bd[k * width + col]可知，块内线程对全局内存进行两次读取操作对应一次乘累加计算操作，计算指令只占计算主体的三分之一，核函数执行中存在大量访问全局内存带来的时延。

为了解决这一问题，重新构建CUDA矩阵乘的核函数，每个线程负责计算一个大小为4×4的矩阵块。经过对CUDA矩阵乘核函数的重新设计，MatrixMul_4x4内计算主体的计算访存比变为了16/8，有利于隐藏访问全局内存时导致的时延。

对经过全局内存优化的CUDA矩阵乘进行测试，在矩阵规模为1024的情况下与经过线程结构优化的CUDA矩阵乘进行对比，编译使用命令为：nvcc global.cu -o global，使用Nsight System工具进行监测核函数运行时间，使用命令：nsys profile –stats=true ./global，测试结果如下表所示。

矩阵规模	函数名称	线程组织布局	时间
1024*1024	MatrixMul_2grid2block	((16,64),(64,16))	1019.6ms
	MatrixMul_4x4	((16,16),(16,16))	455.45us
		((8,8),(32,32))	417.06us

共享内存是GPU上的关键内存部件，与全局内局相比共享内存具有更高的带宽和更低的延迟，其作用类似于一个可编程管理的缓存，在SM上执行的线程块中的所有线程共享该部分内存空间，因此过度使用共享内存空间会限制SM上活跃线程块的数量。

在CUDA内存模型中，每个线程块在开始执行任务时会被分配一定数量的共享内存空间，该共享内存空间具有与线程块相同的生命周期，且地址空间被线程块中所有的线程共享，因此，共享内存常被用作线程块内线程通信的通道，实现块内线程的相互协同。通过最大化利用共享内存这一高速片上内存资源，可以优化核函数对全局内存访问模式的，提升CUDA程序的性能。

CUDA开发者可以对共享内存变量进行静态或动态的分配，例如一个共享内存的二维浮点数组：__shared__ float tile [size_y][size_x]，使用__shared__修饰符对共享内存变量进行声明，若该共享内存变量在核函数中被声明，则变量的作用域仅为核函数内；若该共享内存变量在CUDA程序中所有核函数外被声明，则变量的作用域应为CUDA程序的全局。

通过进行全局内存优化，CUDA矩阵乘法的性能获得了大幅提升，但全局内存高延迟的物理特性限制了其性能的进一步提升，在此基础上选择grid(8,8)block(32,32)的线程布局，通过使用共享内存资源继续对CUDA矩阵乘进行优化。

首先核函数内使用修饰符__shared__静态开辟了数据规模为1024的共享内存空间ldsa与ldsb，接下来线程根据指令进行数据从全局内存到共享内存的转移，线程块内的1024个线程将矩阵A和矩阵B中的1024（128*8）个元素分别转移至ldsa与ldsb中，线程对结果矩阵块中元素部分和进行计算的核心代码未发生变化，但进行乘累加运算时只需以较低的通信开销到共享内存上获取目标元素，从而大大减少了对全局内存频繁访问带来的时延。

对使用共享内存的CUDA矩阵乘进行测试，编译命令为：nvcc MaMul_shared.cu -o shared，使用Nsight System工具进行监测核函数运行时间，使用命令：nsys profile –stats=true ./shared，测试结果如下表所示。由测试得到的数据可知，核函数MatrixMul_Shared的执行时间仅为255.67us，证明了共享内存的使用成功减少了CUDA矩阵乘中面向全局内存访问带来的时延，CUDA矩阵乘的性能得到了进一步的提升。

矩阵规模	函数名称	线程组织布局	时间(us)
1024*1024	MatrixMul_4x4	((8,8),(32,32))	417.06
	MatrixMul_Shared	((8,8),(32,32))	255.67

在以CUDA并行归约例对分支优化进行了说明，CUDA并行归约同样可以利用共享内存提升程序性能，使用共享内存的CUDA并行归约核函数代码如下。

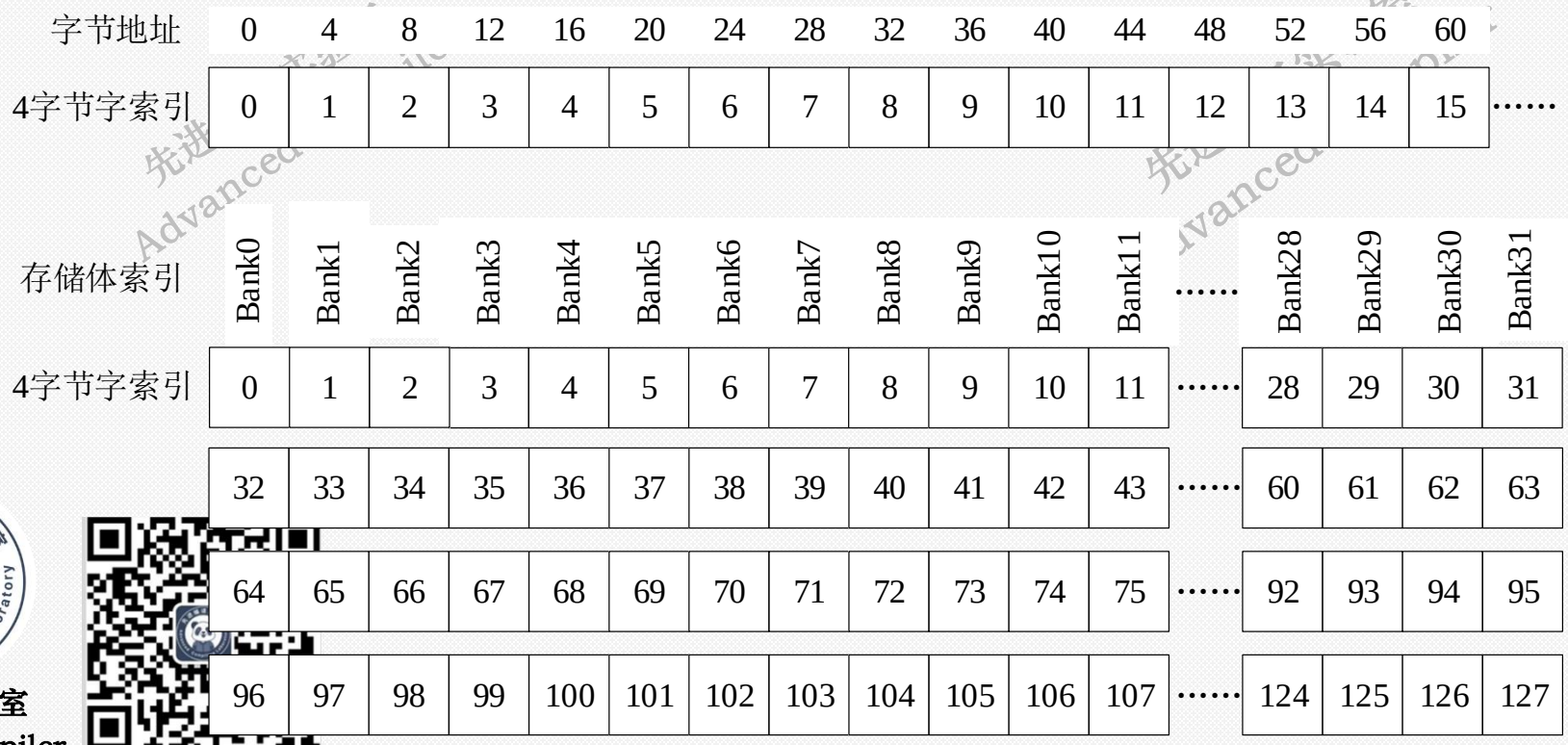
```
__global__ void reduce_shared(int * g_idata,int *g_odata,){
    __shared__ int s_data[1024];
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int tx = threadIdx.x;
    s_data[cacheIndex] = g_idata[tid];
    __syncthreads();
    for (int stride = 1; stride < blockDim.x; stride *= 2){
        int index = 2 * stride * tx;
        if (index < blockDim.x){
            s_data[index] += s_data[index + stride];
        }
            __syncthreads();

    if(cacheIndex == 0)
        g_odata[blockIdx.x] = s_data[tx];
```

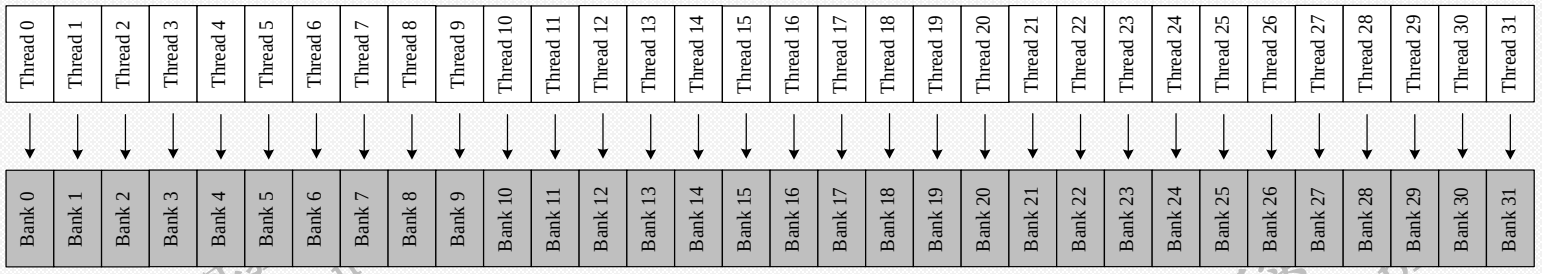
观察上述代码可知，核函数内开辟了数据规模为1024的共享内存空间s_data，线程块内线程将进行归约计算需要用到的数据块首先从全局内存转移至共享内存上，在进行累加计算时从s_data内获取目标元素，对使用共享内存的CUDA归约进行测试，编译命令为：nvcc Reduce_shared.cu -o Reduce_shared，使用Nsight System工具进行监测核函数运行时间，使用命令：nsys profile –stats=true ./Reduce_shared，测试结果如下表所示。

函数名称	□ □ (us)
reduce_GPU	393.02
reduce_NeighboredLess	227.14
reduce_shared	247.32

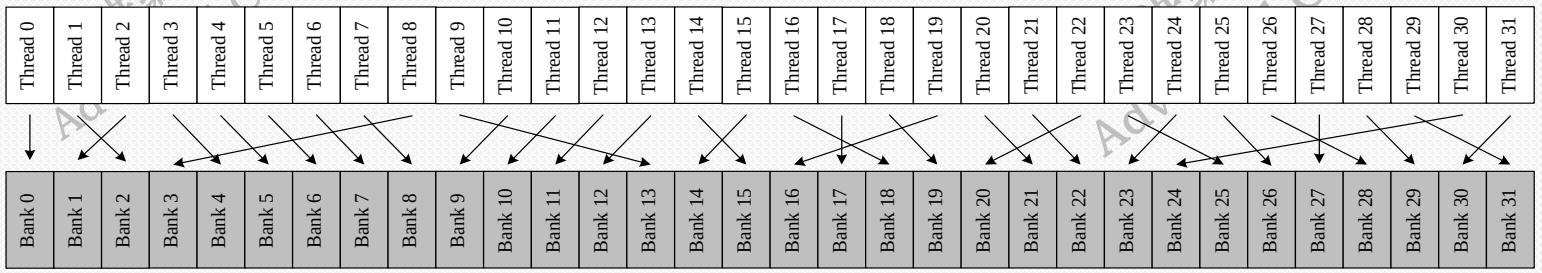
内存带宽是衡量存储设备性能的重要指标，为了获得较高的内存带宽，GPU上的共享内存设备被分32个大小相等的存储器模块，这些存储模块被称为存储体bank，可以被一个线程束内的32个线程同时访问。如下图所示，在费米架构的GPU设备上，连续的4字节数据被分配到连续的32个存储体中。



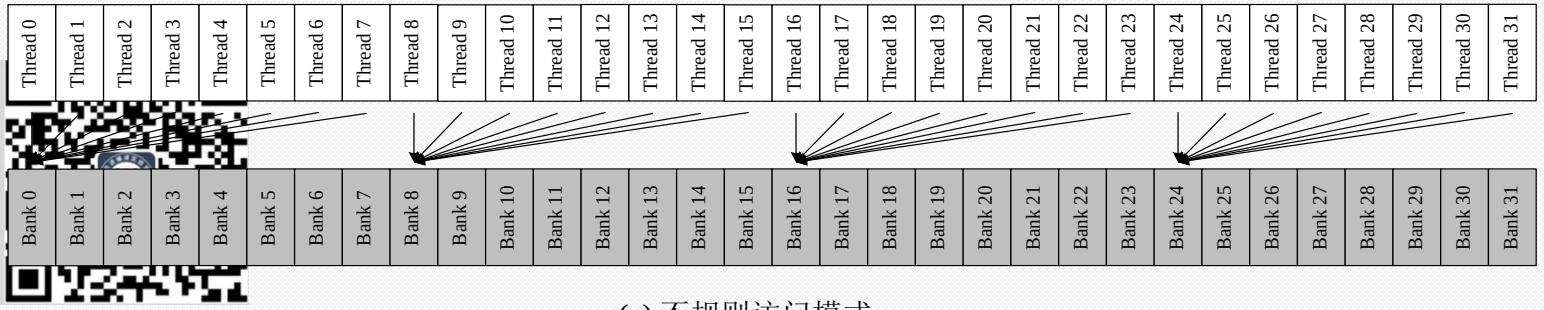
当一个线程束中的不同线程访问一个bank中的不同的字地址时，就会发生bank冲突。图中展示了三种不同的共享内存访问模式。



(a) 线性访问模式



(b) 交叉访问模式



(c) 不规则访问模式

对bank冲突的相关概念进行了解后，重新分析10.4.3节中使用共享内存的CUDA归约核函数代码，发现执行累加操作的s_data[index] += s_data[index + stride]语句会在读取s_data内目标元素时导致bank冲突，当步长变量stride为1时，一个线程束对s_data的访问会产生两路bank冲突，随着迭代中步长的变量stride的增长，bank冲突现象更加严重。通过重新构建累加操作的执行方式来避免bank冲突。

对经过bank冲突优化的共享内存CUDA归约进行测试，编译使用命令：nvcc bankconflict.cu -o bankconflict，使用Nsight System工具进行监测核函数运行时间，使用命令：nsys profile –stats=true ./ bankconflict，测试结果如下表所示。

函数名称	时间(us)
reduce_NeighboredLess	227.14
reduce_shared	247.32
reduce_nobankconflict	217.11

使用Nsight Compute工具对共享内存事务进行监测，其中l1tex__data_pipe_lsu_wavefronts_mem_shared_op_ld.sum与l1tex__data_pipe_lsu_wavefronts_mem_shared_op_st.sum选项分别表示核函数执行过程中对共享内存进行读写所需内存事务的总和，smsp__sass_average_data_bytes_per_wavefront_mem_shared.pct参数表示核函数对共享内存的利用效率，使用命令ncu –metrics smsp__sass_average_data_bytes_per_wavefront_mem_shared.pct ./bankconflict，测试结果如下表所示。

函数名称	参数指标	指标
reduce_shared	smsp__sass_average_data_bytes_per_wavefront_mem_shared.pct	21.11%
reduce_nobankconflict		90.74%
reduce_shared	l1tex__data_pipe_lsu_wavefronts_mem_shared_op_ld.sum	3137536
reduce_nobankconflict		598016
reduce_shared	l1tex__data_pipe_lsu_wavefronts_mem_shared_op_st.sum	1896866
reduce_nobankconflict		625972

与CPU缓存类似，GPU缓存是不可被编程的内存空间。在GPU上有4种缓存分别为一级缓存、二级缓存、只读常量缓存以及只读纹理缓存。在每SM中有一个只读常量缓存和只读纹理缓存，它们用于进一步提高GPU设备的读取性能。在CPU上，内存数据的加载和存储都可以被缓存，但是GPU上的内存存储操作不会被缓存，只有内存加载操作会被缓存。

下面以CUDA矩阵转置为例说明利用高速缓存的优化，将矩阵A、矩阵B均存在全局内存中，其中核函数transpose1中按行对矩阵A进行合并的读操作，而对矩阵B的写操作是非合并的。在核函数transpose2中按列对矩阵A进行非合并的读操作，对矩阵B进行合并的写操作。

对两种CUDA矩阵转置实现进行测试，编译使用命令：nvcc cache.cu -o cache，使用Nsight System工具进行监测核函数运行时间，使用命令：nsys profile –stats=true ./cache。测试结果如下表所示。

函数名称	时间(us)
transpose1	41.34
Transpose 2	19.55

由测试结果得到的数据可知，transpose2的执行时间远小于transpose1，出现性能差距的原因是transpose2对矩阵A的非合并读操作会经由高速缓存，而transpose1中对矩阵B的非合并写操作并不能被缓存，transpose2利用高速缓存优化了面向全局内存的不合并访问，从而获得了更优的性能。

一级缓存和共享内存共享SM上的内存资源，可以通过cudaFuncSetCacheConfig API动态的分配二者的资源占比，其函数原型如下。

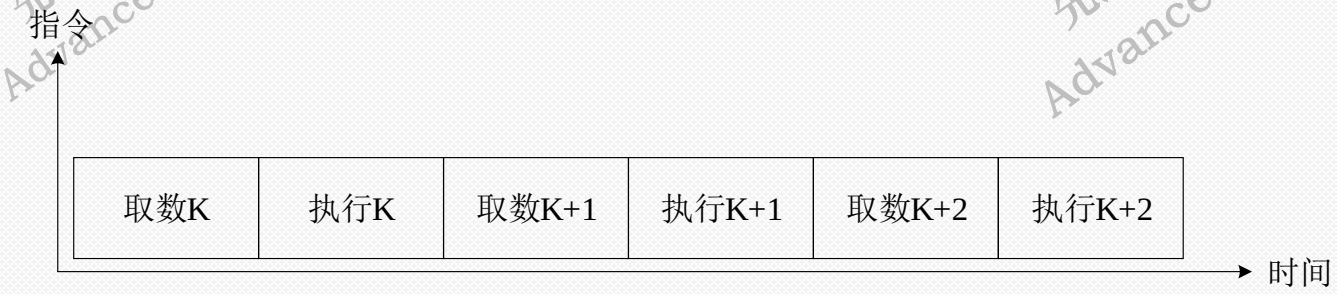
```
cudaError_t cudaFuncSetCacheConfig(const void* func, enum
cudaFuncCachecacheConfig);
func表示分配策略：
cudaFuncCachePreferNone: no preference (default)
cudaFuncCachePreferShared: prefer 48KB shared memory and 16KB L1 cache
cudaFuncCachePreferL1: prefer 48KB L1 cache and 16KB shared memory
cudaFuncCachePreferEqual: Prefer equal size of L1 cache and shared memory, both 32KB
```

优化人员可以通过调用cudaFuncSetCacheConfig函数并选用适当的分配策略对GPU设备上一级缓存与共享内存资源的比例进行调整，从而实现对CUDA程序的优化。

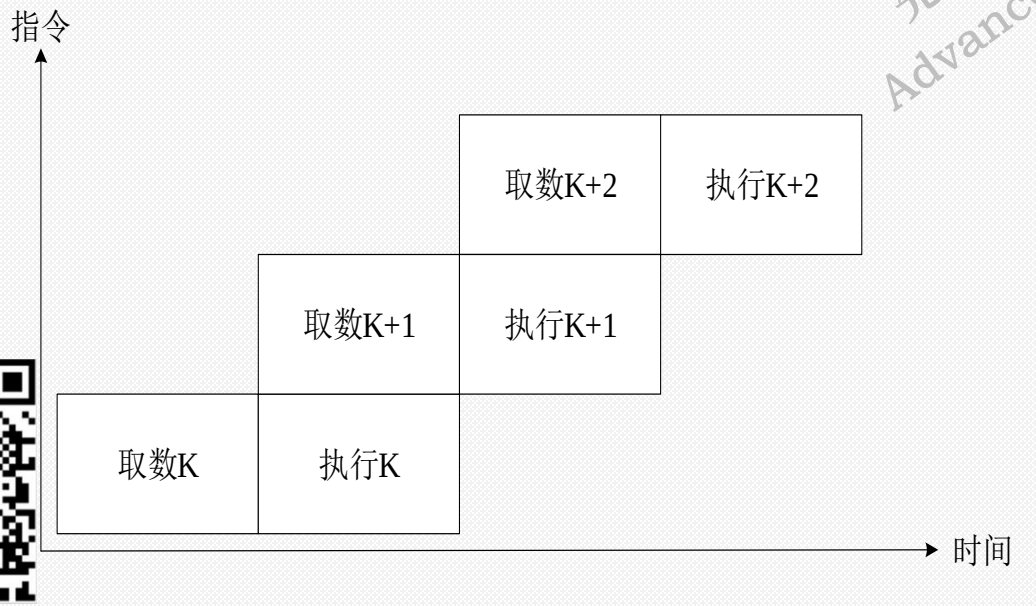
10.5 数据预取

- 基本原理
- 代码实现
- 性能分析

本节提到的数据预取就是希望在执行第k次计算时，同时读取K+1次迭代的数据，这样能够在计算和内存读取之间形成时间重叠而提升程序的性能。这样的数据预取方法是与第八章中提到的数据预取方法是有所不同的。未进行数据预取时的指令执行过程如下图所示。



对于GPU设备而言，进行面向全局内存的取数操作需要约400~800个时钟周期，而进行算数操作则只需要约0~20个时钟周期，若CUDA程序中存在串行执行的计算和内存读取过程，则取数操作会带来大量的时间损耗，不利于提升CUDA程序的性能。使用数据预取优化该问题时，优化人员可以在对数据k进行计算的同时预取数据k+1，在执行操作k+1时可直接进行计算而无需等待取数操作，一定程度上掩藏读取数据k+1的延时，并且在计算的过程中同时预取数据k+2，其流程如下图所示。



10.5 数据预取 10.5.2 代码实现

按照上节提到的数据预取原理，对使用共享内存CUDA矩阵的核函数进行数据预取，根据数据预取的原理，考虑在CUDA矩阵乘核函数中开辟2块共享内存空间，在核心计算的循环体外完成全局内存向第一块共享内存的数据搬运操作，循环体内面向第二块共享内存的数据搬运操作和面向第一块共享内存的乘累加操作得以交叉执行，从而更好地控制时延，提升CUDA矩阵乘的性能。

```
__global__ void MatrixMulShared_4x4(float* Ad, float* Bd, float* Cd, int width){
    for(int j=0; j<width; j+=8 ){
        ldsa[offset_inner] = Ad[ldsm_row*width+ldsm_col+j];
        ldsb[offset_inner] = Bd[(ldsn_row+j)*width+ldsn_col];
        __syncthreads();
      for(int i = 0;i < 8;i++){
            rA[0] = ldsa[threadIdx.y*4+(i*128)+0];
            rA[1] = ldsa[threadIdx.y*4+(i*128)+1];
            rA[2] = ldsa[threadIdx.y*4+(i*128)+2];
            rA[3] = ldsa[threadIdx.y*4+(i*128)+3];

            rB[0] = ldsa[threadIdx.x*4+(i*128)+0];
            rB[1] = ldsa[threadIdx.x*4+(i*128)+1];
            rB[2] = ldsa[threadIdx.x*4+(i*128)+2];
            rB[3] = ldsa[threadIdx.x*4+(i*128)+3];

            rC[0] =rC[0] + rA[0] * rB[0];
            rC[1] =rC[1] + rA[0] * rB[1];
            rC[2] =rC[2] + rA[0] * rB[2];
            rC[3] =rC[3] + rA[0] * rB[3];

           rC[4] =rC[4] + rA[1] * rB[0];
            rC[5] =rC[5] + rA[1] * rB[1];
            rC[6] =rC[6] + rA[1] * rB[2];
            rC[7] =rC[7] + rA[1] * rB[3];

           rC[8] =rC[8] + rA[2] * rB[0];
            rC[9] =rC[9] + rA[2] * rB[1];
            rC[10] =rC[10] + rA[2] * rB[2];
            rC[11] =rC[11] + rA[2] * rB[3];

           rC[12] =rC[12] + rA[3] * rB[0];
            rC[13] =rC[13] + rA[3] * rB[1];
            rC[14] =rC[14] + rA[3] * rB[2];
            rC[15] =rC[15] + rA[3] * rB[3];
    }
}
```

在共享内存优化核函数的基础上添加数据预取操作，在核函数内开辟共享内存空间ldsa、ldsb的数据规模由1024增大至2048，首先在核心计算的循环体外将矩阵A、B上的1024个元素从全局内存搬运至ldsa[0~1023]、ldsb[0~1023]上，在核心计算的循环体的首次迭代中，线程取ldsa[0~1023]、ldsb[0~1023]内的元素进行乘累加操作，同时将下次乘累加操作的目标元素从矩阵A、B搬运至ldsa[1024~2047]、ldsb[1024~2047]上，通过数据搬运与运算指令的交叉执行，一定程度上掩藏了CUDA矩阵乘中进行全局内存向共享内存数据搬运的耗时。

对经过数据预取优化的共享内存CUDA矩阵乘进行测试，编译使用命令nvcc preload.cu -o preload，使用Nsight System工具进行监测核函数运行时间，使用命令nsys profile –stats=true ./preload。测试结果如下表所示。

矩阵规模	线程布局	核函数名称	时间(us)
1024*1024	(8,8) (32,32)	MatrixMulShared_4x4	256.58
		MatrixMulShared_preload	237.28

由测试的得到数据可知，经过数据预取优化的CUDA矩阵乘相较于未经优化的共享内存CUDA执行时间进一步减小，性能得到了提升。对共享内存CUDA矩阵乘进行数据预取优化时，每个线程块使用了更大的共享内存空间，由于GPU上共享内存空间存在资源限制，消耗过多存储资源会一定程度上影响GPU设备的性能，无法反映数据预取的真实优化效果，因此调整CUDA矩阵乘结果矩阵的数据规模和线程布局再次进行测试，测试结果如下表所示。

矩阵规模	线程布局	核函数名称	时间（us）
512*512	(8,8) (16,16)	MatrixMulShared_4x4	86.52
		MatrixMulShared_preload	63.10

10.6 循环展开

- 基本原理
- 代码实现
- 性能分析

先进编译实验室
Advanced Compiler

循环展开通过增加每次迭代计算的元素的数量，从而减少循环的迭代次数。循环展开通过消除分支和管理归纳变量，让更多的并发操作被添加到流水线上，有效提升程序性能，以数组求和的循环为例。

```
for (int i=0;i<100;i++){
    a[i]=b[i]+c[i];
}
```

```
for (int i=0;i<100;i+=4){
    a[i+0]=b[i+0]+c[i+0];
    a[i+1]=b[i+1]+c[i+1];
    a[i+2]=b[i+2]+c[i+2];
    a[i+3]=b[i+3]+c[i+3];
}
```

GPU设备通过线程束间切换实现计算的高效并发，充足的运算指令有利于提升CUDA程序的性能，因此使用循环展开增加核函数内的指令数有利于提升CUDA程序的并行性。此外，由于GPU设备缺少复杂的分支预测单元，因此消除使用循环展开循环迭代中的分支判断语句有利于减少CUDA程序执行时进行判断和分支预测造成的耗时。

下面代码是CUDA归约核函数redue_nobankconflict中的最后一次累加计算操作，通过观察代码可以发现，该循环体实现的归约操作由线程块内的一个线程束实现，经由一次分支判断线程束仅执行一次计算操作，且随着迭代的进行，循环条件使得线程束内的活跃线程数目不断减少。

```
for (int stride = 16; stride >0; stride=stride>>1){
    if (tx < stride)
   data[0][col] += data[0][col+ stride];
   __syncthreads();
}
```

在redue_nobankconflict核函数为基础使用循环展开对CUDA归约实现进行优化，经过优化的代码如下所示。

```
__global__ void reduce_unroll(int *a, int *r)
{
    __shared__ int data[32][32];
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int tx = threadIdx.x;
    int row = tx / 32;
    int col = tx % 32;
    data[row][col] = a[tid];
    __syncthreads();
    for(int stride = 16; stride>0; stride=stride>>1){
        if (row < stride)
            data[row][col] += data[row+stride][col];
        __syncthreads();
    }
```

```
    if(tx < 32){
        data[0][col] += data[0][col+ 16];
        __syncthreads();
        data[0][col] += data[0][col+ 8];
        __syncthreads();
        data[0][col] += data[0][col+ 4];
        __syncthreads();
        data[0][col] += data[0][col+ 2];
        __syncthreads();
        data[0][col] += data[0][col+ 1];
        __syncthreads();
    }
    __syncthreads();
    if(tx==0)
    r[blockIdx.x] = data[0][0];
}
```

对经由循环展开优化的CUDA矩阵归约进行测试，相同数据规模下对CUDA归约的不同实现进行测试，编译使用命令：nvcc reduce_unroll.cu -o reduce_unroll，使用Nsight System工具进行监测核函数运行时间，指令为nsys profile –stats=true ./reduce_unroll，测试结果如下表所示。

函数名称	时间(us)
reduce_NeighboredLess	227.14
reduce_shared	247.32
reduce_nobankconflict	217.11
reduce_unroll	168.24

由测试结果可知，经过循环展开优化的CUDA归约核函数reduce_unroll 执行时间为168.24us，相较于消除bank冲突的归约核函数reduce_nobankconflict性能取得了进一步提升。对CUDA程序进行循环展开操作会消耗GPU上更多的寄存器资源，在reduce_unroll中选择进行循环展开的循环体内迭代次数有限，因此在寄存器资源许可的范围内进行了完全展开的操作，优化人员CUDA程序中迭代次数较多的循环体进行展开操作时，需要注意避免过度展开，因为避免过度展开反而降低程序性能。

10.7 小结

本章从CUDA编程模型、多层次存储结构、数据预取以及循环展开等角度对CUDA程序优化方法进行了介绍，并结合矩阵乘法和归约等示例对优化方法进行了实现。

首先介绍CUDA的基础概念以及CUDA程序的编写方法，并以矩阵乘为例说明如何改写CUDA程序 。在CUDA线程结构理论基础之上，通过构建合理的线程布局挖掘线程并行性实现对CUDA程序的优化。之后对如何根据GPU设备的SIMT执行模式消除程序中的分歧以提升CUDA程序的性能进行了分析。依据CUDA的多层次存储结构特点，介绍了如何利用全局内存、共享内存、高速缓存三个不同层次的存储空间构建CUDA程序中高效内存访问模式。最后说明了如何使用数据预取、循环展开等优化方法对使用CUDA编程模型的程序进行优化。