

课程实验

师清

2022 年 12 月 15 日

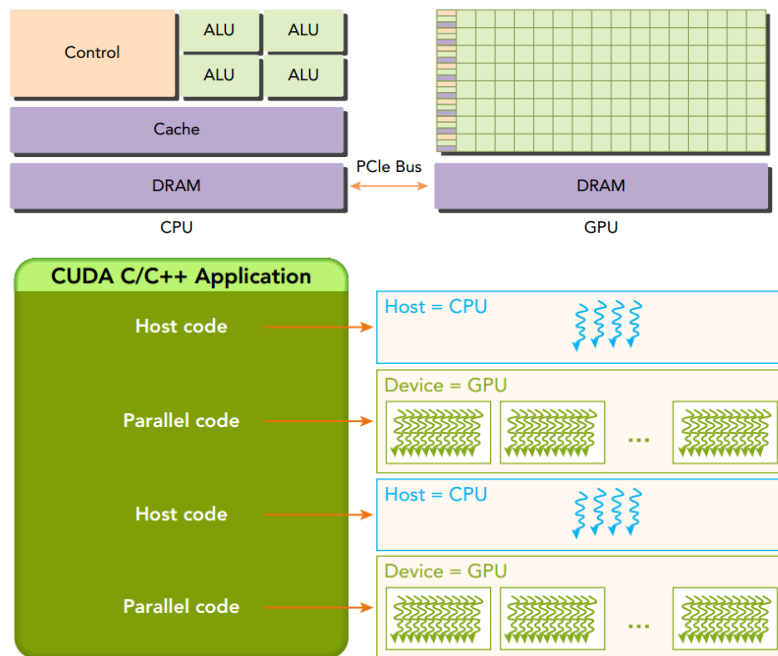
1 cuda 编程模型

1.1 异构并行计算

CPU+GPU

CPU:heavy-weight,for complex control logic

GPU:light-weight,for data-parallel tasks with simple control logic

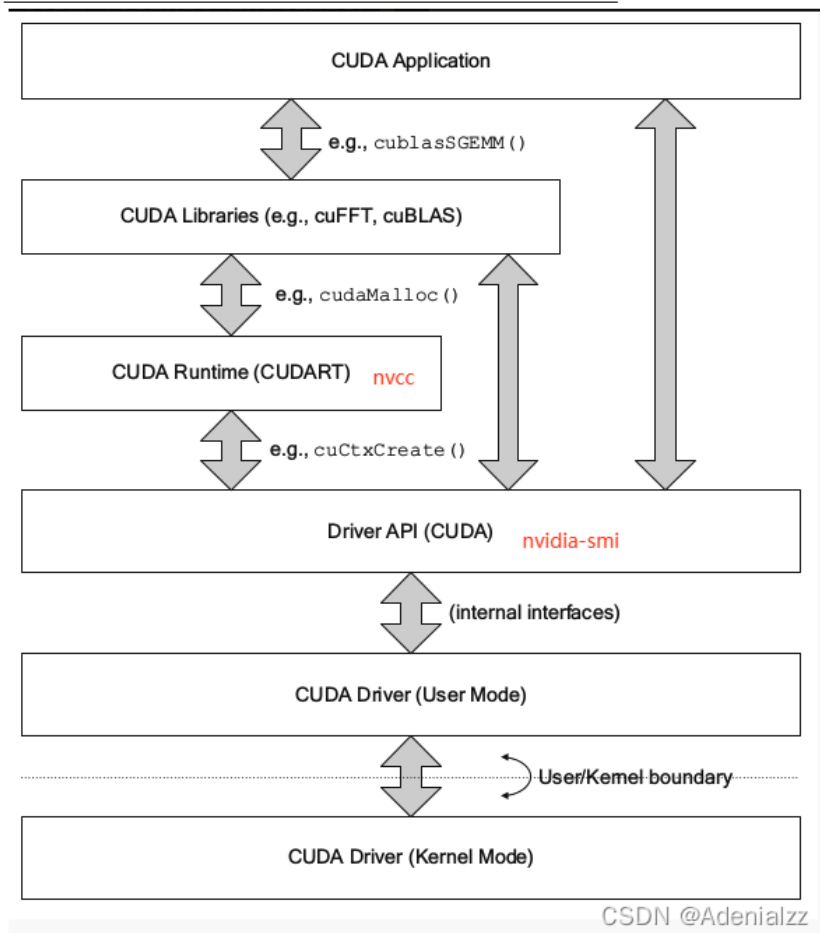


Peak computational performance:TFLOPS(每秒万亿次的浮点运算)

Memory bandwidth:GB/s

1.2 cuda 编程架构

CUDA: A Platform for Heterogeneous Computing



两种 api: CUDA Driver API 和 CUDA Runtime API

1.2.1 物理上

nvidia gpu 架构的演变:

架构代号	Fermi	Kepler	Maxwell	Pascal	Volta	Turing	Ampere	Hopper
中文代号	费米	开普勒	麦克斯韦	帕斯卡	伏特	图灵	安培	赫柏
时间	2010	2012	2014	2016	2017	2018	2020	2022
核心参数	16 个 SM，每个 SM 包括 32 Cuda Cores，共计 512 Cuda Cores	15 个 SMx，每个 SMx 包括 192 个单精度 + 64 个双精度的 Cuda cores；	16 个 SMM，每个 SM 包括 4 个处理块，每个处理块包括 32 个 CUDA Unit + 8 个 SFU	Pascal 架构有 GP100、GP102 GP100 有 60 个 SM 每个 SM 包括 64 个 cuda cores 32 个 DP cores	80 个 SM，每个 SM 里 32 个 FP64 64 个 INT32 64 个 FP32 8 个 Tensor core	TU102 核心 72 个 SM，SM 全新设计，每个 SM 里 64 个 INT32 64 个 FP32 32 个 FP64 8 个 Tensor core	A100 有 108 SMs 每个 SM 64 个 FP32 64 个 INT32 32 个 FP64 4 个 Tensor core	H100 132 SM 每个 SM 128 个 FP32 64 个 INT32 64 个 FP64 4 个 Tensor core
特点/优势	首个完整 GPU 计算架构，支持与共享缓存结合 Cache 部分的 GPU 架构，支持 ECC 的 GPU 架构	游戏性能大幅提升，首次支持 GPU Direct 技术	相比 Kepler 的每组 SM 单元 192 个减少到了每组 128 个，但是每个 SMM 单元拥有更多的逻辑控制电路	NVLink 一代，双向互联带宽 160GB/s P100 有 56 个 SM HBM	Nvlink 2.0 Tensor Core 1.0 满足深度学习和 AI 运算	Tensor Core 2.0 RT Core 1.0	Tensor Core 3.0 RT Core 2.0 Nvlink 3.0 结构稀疏性 MIG 1.0	Tensor Core 4.0 Nvlink 4.0 结构稀疏性矩阵 MIG 2.0
纳米制程	40/28nm 30 亿晶体管	28nm 71 亿晶体管	28nm 80 亿晶体管	16nm 153 亿晶体管	12nm 211 亿晶体管	12nm 186 亿晶体管	7nm 283 亿晶体管	4nm 800 亿晶体管
代表型号	Quadro 7000	K80 K40M	M5000 M4000	P100 GTX 1080 P6000	V100 Titan V	T4 2080TI RTX 5000	A100、A30 3090	H100

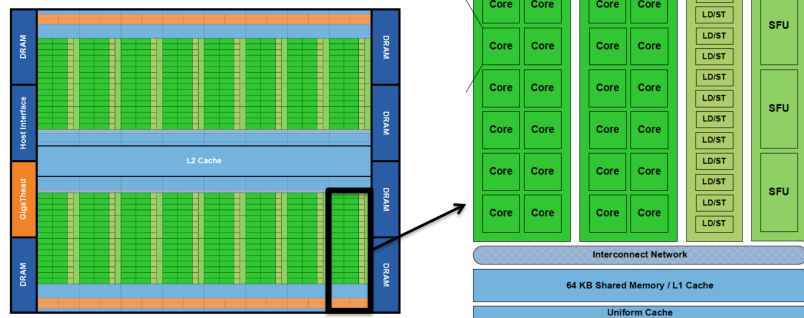
Fermi 架构: 第一个完整的 GPU 计算架构¹:

NVIDIA Fermi Architecture

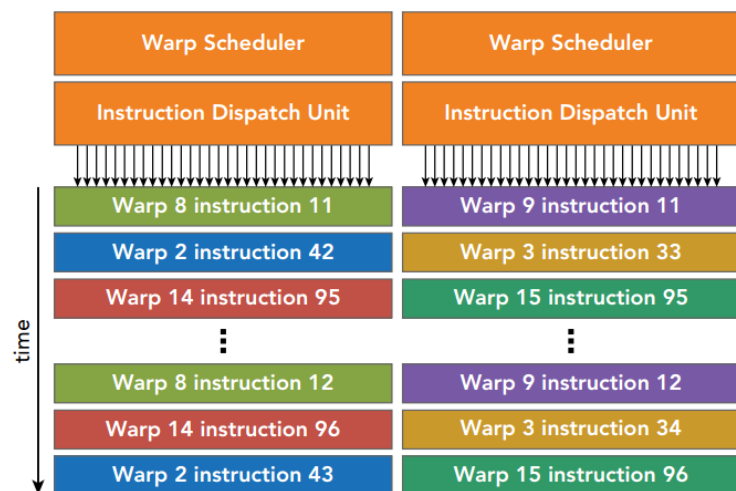
Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU).

More cores per multiprocessors and faster arithmetic operations

Memory Protection Support: Memory are protected by a Single - Error Correct Double - Error Detect (SECDED) ECC code



¹Fermi 架构



单个 SM(Stream Multiprocessor) 包括如下组成部分:

- Core, 也叫流处理器 Stream Processor
- LD/ST (load/store) 模块来加载和存储数据
- SFU (Special function units) 执行特殊数学运算 (sin、cos、log 等)
- 寄存器 (Register File)
- L1 缓存
- 全局内存缓存 (Uniform Cache)
- 纹理缓存 (Texture Cache)

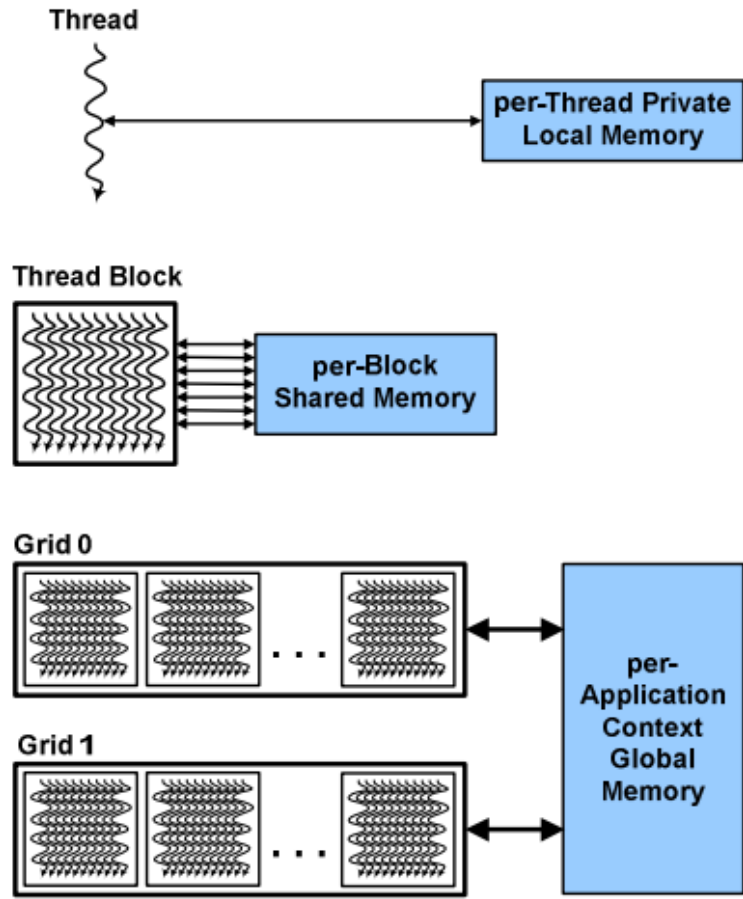
1.2.2 逻辑上

[thread hierarchy](#)²:

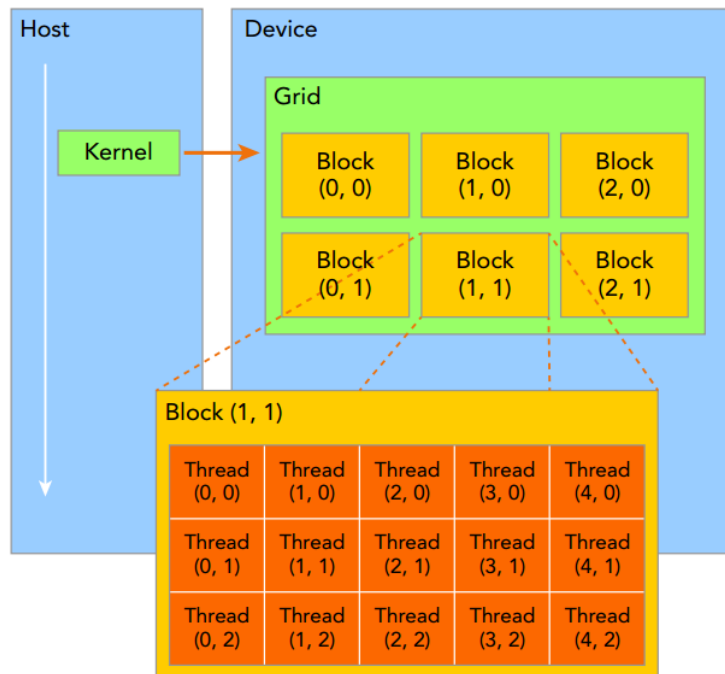
grid,block,thread,warp

gridDim(blockIdx.x,blockIdx.y,blockIdx.z),blockDim(threadIdx.x,threadIdx.y,threadIdx.z)

²cuda 核函数的并行机制



CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

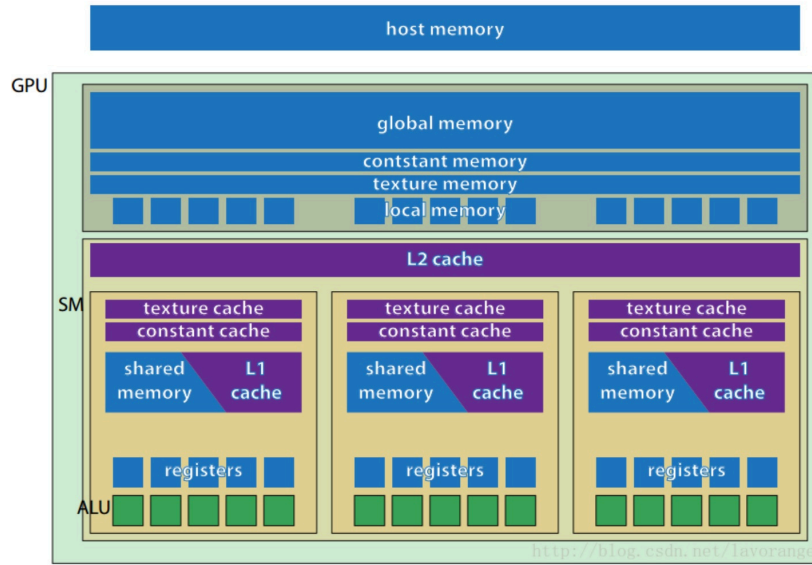


memory hierarchy³:

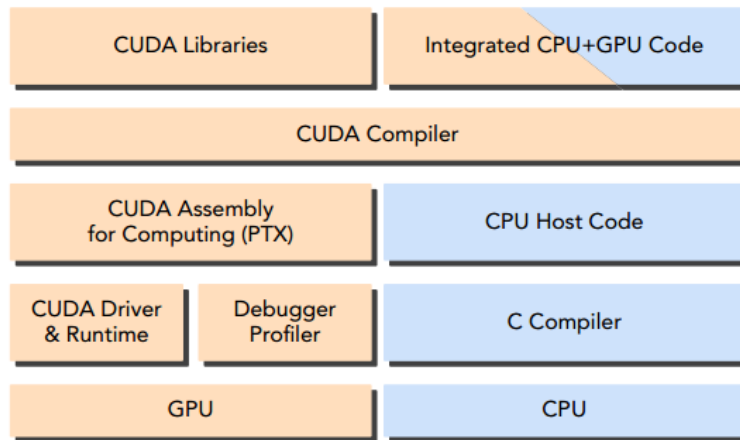
register, local memory, shared memory, constant memory, texture memory, global memory, host memory

³cuda 内存层次

Memory



1.2.3 CUDA C



[nvcc](#), 编译工具⁴

nvprof 已被弃用，使用下面两个作为 profile 工具

[NVIDIA Nsight Compute](#), [ncu](#)

⁴nvcc 编译过程

[NVIDIA Nsight Systems](#), nsys
CUDA C syntax

1.3 本机实验环境

GPU: NVIDIA GeForce RTX 3050 Laptop GPU

Driver Version: 470.141.03

CUDA Version: 11.4

nvcc:release 11.0, V11.0.194

2 cuda 编程练习

(APOD 开发模型, 即: Assess, Parallelize, Optimize, Deploy)

[代码地址](#)

2.1 矩阵加法

2.1.1 代码

```
#include <cuda_runtime.h>
#include <stdio.h>
#include "../include/myhelp.h"

void sumArrays(float* a, float* b, float* res, const int size){
    for(int i=0; i<size; i++){
        res[i]=a[i]+b[i];
    }
}

__global__ void sumArraysGPU(float* a, float* b, float* res){
    int i=threadIdx.x;
    res[i]=a[i]+b[i];
}

int main(int argc, char** argv){
    int dev=0;
    cudaSetDevice(dev);

    int n=32;
    printf("Vector size:%d\n", n);
    int nByte=sizeof(float)*n;
```



```

float* a_h=(float*) malloc(nByte);
float* b_h=(float*) malloc(nByte);
float* res_h=(float*) malloc(nByte);
float* res_from_gpu_h=(float*) malloc(nByte);
memset(res_h,0,nByte);
memset(res_from_gpu_h,0,nByte);

float* a_d,* b_d,* res_d;
CHECK(cudaMalloc((float**)&a_d,nByte));
CHECK(cudaMalloc((float**)&b_d,nByte));
CHECK(cudaMalloc((float**)&res_d,nByte));

initialData(a_h,n);
initialData(b_h,n);

CHECK(cudaMemcpy(a_d,a_h,nByte,cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(b_d,b_h,nByte,cudaMemcpyHostToDevice));

dim3 block(n);
dim3 grid(n/block.x);
sumArraysGPU<<<grid,block>>>(a_d,b_d,res_d);
printf("Execution configuration<<<%d,%d>>>",block.x,grid.x);

CHECK(cudaMemcpy(res_from_gpu_h,res_d,nByte,cudaMemcpyDeviceToHost));
sumArrays(a_h,b_h,res_h,n);

checkResult(res_h,res_from_gpu_h,n);
cudaFree(a_d);
cudaFree(b_d);
cudaFree(res_d);

free(a_h);
free(b_h);
free(res_h);
free(res_from_gpu_h);

return 0;
}

```

2.1.2 实验结果

```
starting...
Using device 0: NVIDIA GeForce RTX 3050 Laptop GPU
CPU execution time:0.218834 sec
CPU execution configuration<<<(512,512),(32,32)>>> Time elapsed 0.074313 sec
Check result success!
GPU Execution configuration<<<(8388608,1),(32,1)>>> Time elapsed 0.068178 sec
Check result success!
GPU Execution configuration<<<(512,16384),(32,1)>>> Time elapsed 0.248608 sec
Check result success!
```

2.1.3 分析总结

不同的 execution configuration 会影响执行性能, 因此尝试不同的 grid 和 block dimensions 可能产生更好的性能。

2.2 warp divergence

2.2.1 实验结果

```
./divergence using device 0:NVIDIA GeForce RTX 3050 Laptop GPU
Data size:64
warmup <<<1,64>>> elapsed 0.000021 sec
mathKernel1<<<1, 64>>>elapsed 0.000008 sec
mathKernel2<<<1, 64>>>elapsed 0.000008 sec
mathKernel<<<1, 64>>>elapsed 0.000007 sec
```

2.2.2 分析总结

同一个 warp 内的线程必须执行同一条指令, 当线程内存在控制流时, 不同线程可能有不同的执行路径, 这会降低核函数效率, 所以要尽量避免同一个 warp 内的线程分化。

2.3 数组求和

2.3.1 实验结果

```
Using device 0: NVIDIA GeForce RTX 3050 Laptop GPU
with array size 16777216 grid 16384 block 1024
cpu sum:2139334732
cpu reduce elapsed 0.005951 ms cpu_sum: 2139334732
gpu reduceNeighbored elapsed 0.015273 ms gpu_sum: 2139334732<<<grid 16384 block 1024>>>
gpu reduceNeighboredLess elapsed 0.012469 ms gpu_sum: 2139334732<<<grid 16384 block 1024>>>
gpu reduceInterleaved elapsed 0.010630 ms gpu_sum: 2139334732<<<grid 16384 block 1024>>>
Test success!
```

2.3.2 分析总结

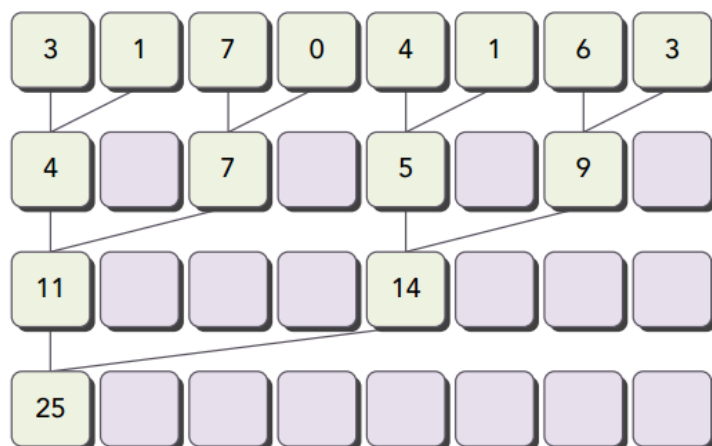


FIGURE 3-19

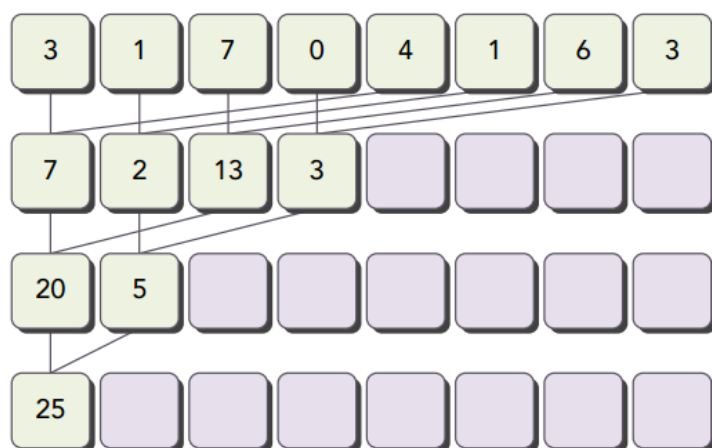


FIGURE 3-20

- The Parallel Reduction Problem(并行规约问题),避免分支分化 (branch divergence)
- 使用 interleaved pair approach 替代 neighbored approach, 性能提升
- Synchronization: system-level 和 block-level, 分别使用 `cudaDeviceSynchronize()` 和 `__syncthreads()`

2.4 循环展开

正常循环

```
1 int x;  
2 for (x = 0; x < 100; x++)  
3 {  
4     delete(x);  
5 }
```

循环展开后

```
1 int x;  
2 for (x = 0; x < 100; x += 5 )  
3 {  
4     delete(x);  
5     delete(x + 1);  
6     delete(x + 2);  
7     delete(x + 3);  
8     delete(x + 4);  
9 }
```

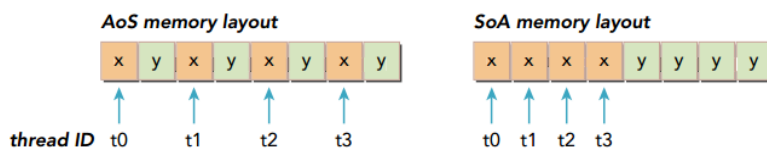
2.4.1 实验结果

```
Using device 0: NVIDIA GeForce RTX 3050 Laptop GPU  
with array size 16777216 grid 16384 block 1024  
cpu sum:2138989603  
cpu reduce elapsed 0.003168 ms cpu_sum: 2138989603  
gpu warmup elapsed 0.007500 ms  
reduceUnrolling2 elapsed 0.005445 ms gpu_sum: 2138989603<<<grid 8192 block 1024>>>  
reduceUnrolling4 elapsed 0.003656 ms gpu_sum: 2138989603<<<grid 4096 block 1024>>>  
reduceUnrolling8 elapsed 0.002102 ms gpu_sum: 2138989603<<<grid 2048 block 1024>>>  
reduceUnrollingWarp8 elapsed 0.001901 ms gpu_sum: 2138989603<<<grid 2048 block 1024>>>  
reduceCompleteUnrollWarp8 elapsed 0.001744 ms gpu_sum: 2138989603<<<grid 2048 block 1024>>>  
reduceCompleteUnroll elapsed 0.001765 ms gpu_sum: 2138989603<<<grid 2048 block 1024>>>  
Test success!
```

2.4.2 分析总结

- 循环展开是一种通过减少分支和循环指令来优化程序性能的方法，利用手动重复执行某个操作来代替一个循环体
- 循环展开为什么能提升性能？编译器可以进行 low-level 的指令优化（指令流水的充分调度）

2.5 结构体数组 vs 数组结构体



2.5.1 实验结果

```
(py38) buyizhiyou@buyizhiyou-Lenovo-Y70002021:~/workspace/cuda/19_aos/build$ ./aos
Vector size:16777216
Execution configuration<<<16384,1024>>> Time elapsed 0.002810 sec
result check success!
(py38) buyizhiyou@buyizhiyou-Lenovo-Y70002021:~/workspace/cuda/19_aos/build$ cd ../../20_soa/build/
(py38) buyizhiyou@buyizhiyou-Lenovo-Y70002021:~/workspace/cuda/20_soa/build$ ./soa
Vector size:16777216
Execution configuration<<<16384,1024>>> Time elapsed 0.002578 sec
result check success!
(py38) buyizhiyou@buyizhiyou-Lenovo-Y70002021:~/workspace/cuda/20_soa/build$ |
```

2.5.2 分析总结

- 内存访问模式: 理想的是对齐联合访问 (aligned and coalesced access pattern)
- 并行编程范式, 尤其是 SIMD (单指令多数据) 对 SoA 更友好。CUDA 中普遍倾向于 SoA 因为这种内存访问可以有效地合并。

2.6 矩阵转置

2.6.1 实验结果

```
(py38) buyizhiyou@buyizhiyou-Lenovo-Y70002021:~/workspace/cuda/22_transform_matrix/build$ ./transform_matrix 2
strating...
Using device 0: NVIDIA GeForce RTX 3050 Laptop GPU
CPU Execution Time elapsed 0.086112 sec
Time elapsed 0.005158 sec
Check result success!
(py38) buyizhiyou@buyizhiyou-Lenovo-Y70002021:~/workspace/cuda/22_transform_matrix/build$ ./transform_matrix 3
strating...
Using device 0: NVIDIA GeForce RTX 3050 Laptop GPU
CPU Execution Time elapsed 0.095592 sec
Time elapsed 0.002936 sec
Check result success!
(py38) buyizhiyou@buyizhiyou-Lenovo-Y70002021:~/workspace/cuda/22_transform_matrix/build$ ./transform_matrix 4
strating...
Using device 0: NVIDIA GeForce RTX 3050 Laptop GPU
CPU Execution Time elapsed 0.104322 sec
Time elapsed 0.001631 sec
Check result success!
(py38) buyizhiyou@buyizhiyou-Lenovo-Y70002021:~/workspace/cuda/22_transform_matrix/build$ ./transform_matrix 5
strating...
Using device 0: NVIDIA GeForce RTX 3050 Laptop GPU
CPU Execution Time elapsed 0.109104 sec
Time elapsed 0.001638 sec
Check result success!
(py38) buyizhiyou@buyizhiyou-Lenovo-Y70002021:~/workspace/cuda/22_transform_matrix/build$ ./transform_matrix 6
strating...
Using device 0: NVIDIA GeForce RTX 3050 Laptop GPU
CPU Execution Time elapsed 0.084095 sec
Time elapsed 0.005553 sec
Check result success!
(py38) buyizhiyou@buyizhiyou-Lenovo-Y70002021:~/workspace/cuda/22_transform_matrix/build$ ./transform_matrix 7
strating...
Using device 0: NVIDIA GeForce RTX 3050 Laptop GPU
CPU Execution Time elapsed 0.083944 sec
Time elapsed 0.003414 sec
Check result success!
(py38) buyizhiyou@buyizhiyou-Lenovo-Y70002021:~/workspace/cuda/22_transform_matrix/build$ |
```

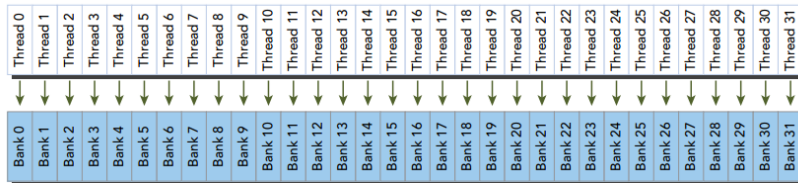
2.6.2 分析总结

- 当分析一个核函数性能时, 注意分析 memory latency(memory bandwidth), 一个好的内存访问模式能提高存储带宽

- 2,3 分别是按行读取和按列读取，按列读取的吞吐量大于按行读取的原因是缓存命中
- 4,5 分别是利用循环展开提升性能
- 6,7 是使用对角化的方式，提高对存储块的均匀访问

2.7 共享内存的读写

memory bank:



padding:

Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	padding
0	1	2	3	4	
0	1	2	3	4	
0	1	2	3	4	
0	1	2	3	4	
0	1	2	3	4	

Bank 0	Bank 1	Bank 2	Bank 3	Bank 4
0	1	2	3	4
	0	1	2	3
4		0	1	2
3	4		0	1
2	3	4		0
1	2	3	4	

2.7.1 实验结果

Time (s)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridV2	BlockV2	Name
17.1	4,865	1	4,865.0	4,865.0	4,865	4,865	0.0	1	1	setColHeadCol(int *)
15.7	4,440	1	4,440.0	4,440.0	4,440	4,440	0.0	1	1	setRowHeadRow(int *)
14.9	4,224	1	4,224.0	4,224.0	4,224	4,224	0.0	1	1	setRowHeadCol(int *)
14.0	3,968	1	3,968.0	3,968.0	3,968	3,968	0.0	1	1	setRowHeadColipad(int *)
13.9	3,936	1	3,936.0	3,936.0	3,936	3,936	0.0	1	1	setColHeadRow(int *)
13.1	3,712	1	3,712.0	3,712.0	3,712	3,712	0.0	1	1	setRowHeadColDyn(int *)
11.4	3,232	1	3,232.0	3,232.0	3,232	3,232	0.0	1	1	setRowHeadColDynipad(int *)

2.7.2 分析总结

- shared memory 的行主序读写和列主序读写, 按照行主序读和写, 减少共享内存冲突 (bank conflict)
- shared memory 的动态分配, 在 `kernel_name <<< ... >>>` 里面可以指定分配的 shared memory
- shared memory 的 padding, 减少 bank conflicts

2.8 利用共享内存做 reduce 求和任务

2.8.1 实验结果

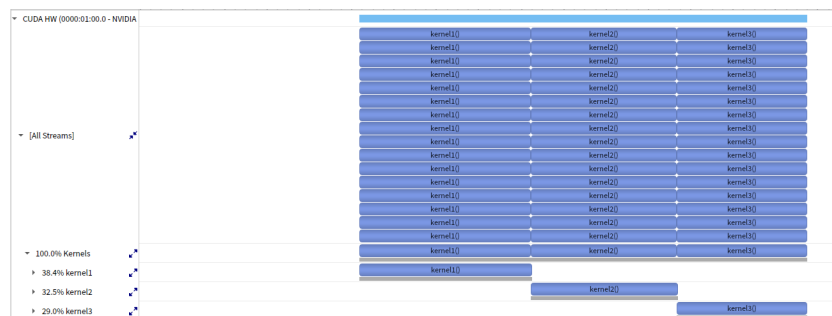
```
(py38) buytzhlyou@buytzhlyou-Lenovo-Y70002021:~/workspace/cuda/25_reduce_integer_shared_memory
shared_memory
Using device 0: NVIDIA GeForce RTX 3050 Laptop GPU
with array size 16777216
grid 16384 block 1024
gpu warmup      elapsed 0.007554 ms
reduceGmem      elapsed 0.010032 ms gpu_sum: 2139151230
reduceSmem      elapsed 0.010522 ms gpu_sum: 2139151230
reduceUnroll4Smem elapsed 0.002732 ms gpu_sum: 2139151230
```

2.8.2 分析总结

- 可以看到使用 shared memory 相比使用 global memory 带来的性能提升。因为 shared memory 是片上 (on-chip), 可以减少访问全局内存的时间。

2.9 cuda stream

2.9.1 实验结果



2.9.2 分析总结

- 最开始性能测试命令 `ncu` 报错，未做更精细的性能分析（后重装 `cuda` 可以使用）
- 因为使用不同的 GPU 架构和 CUDA 版本，有些程序没有产生像书中分析那样的结果
- 还有更深入的优化并行算法的方法有待研究

4 参考资料

1. 《Professional CUDA C Programming》