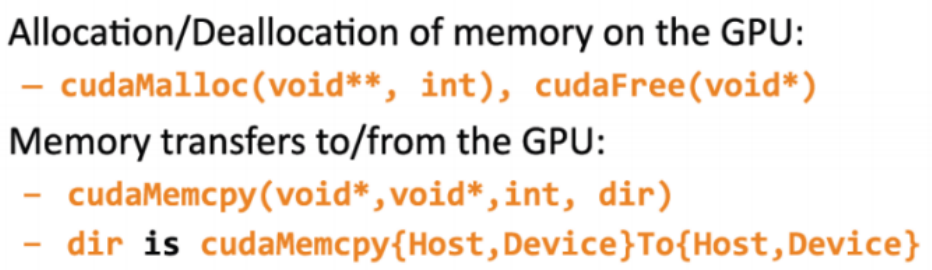
10.1 GPGPU简介，CUDA 编程模型

- 背景
  - GPU里是multi-core chip，有自己的内存
  - GPU需要和CPU配合一起使用
  - SIMD，在单核上执行（多个执行单元执行同一条指令）
  - 单核上是并发的执行多线程
- 大纲
  - GPGPU和CUDA编程模型简介
  - CUDA线程层次结构
  - CUDA内存层次结构
  - 将CUDA映射到Nvidia GPU
  - OpenCL简介
- GPGPUs和CUDA编程模型的介绍
  - GPU硬件的演变
    - CPU架构用摩尔定律来增长
      - 片上缓存数量
      - 处理器的复杂度和时钟速率
      - 遗留工作负载的单线程性能
    - GPU架构使用摩尔定律
      - 增加片上并行度和DRAM的带宽
      - 提高绘图应用程序的灵活性和性能
      - 加速通用数据并行工作负载
  - **CUDA(Compute Unified Device Architecture)编程模型的目标**
    - 为跨各种处理器的数据并行编程提供一个固有的**可扩展**环境
      - CUDA被抽象成许多可以以任何顺序运行的独立的计算块
      - CUDA编程模型的可扩展性源于“线程块”的批处理执行
      - 许多程序在多核的GPU上可以达到线性加速比
    - 让程序员可以访问SIMD硬件，否则大量可用的执行硬件将被浪费
      - 硬件架构师喜欢 SIMD，因为它允许非常节省空间和节能的实施。
      - 但是，CPU 上的标准 SIMD 指令不灵活，难以使用，编译器难以定位。
      - Cuda 线程抽象将以增加硬件为代价提供可编程性。
  - 在GPU上运行程序
    - 在GPU内存中分配一个buffer，将数据拷贝到buffer中
    - 给GPU提供一个kernel程序

    - SPMD执行kernel
- CUDA C 语言扩展
    - 在 GPU 上运行的代码是用标准 C/C++ 语法编写的，具有最少的扩展集：
        - 在 GPU 上运行的代码是用标准 C/C++ 语法编写的，具有最少的扩展集：
        - 为 SIMD 扩展提供 MIMD 线程抽象
        - 启用 CUDA 线程层次结构的规范
        - 线程块内的同步和数据共享
        - GPU 特定功能的内在函数库

```
__global__ void KernelFunc(...);  // define a kernel callable from host
__device__ void DeviceFunc(...);  // function callable only on the device
__device__ int  GlobalVar;        // variable in device memory
__shared__ int  SharedVar;        // in per-block shared memory
KernelFunc<<<500, 128>>>(...);    // 500 blocks, 128 threads each
// Thread indexing and identification
dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;
__syncthreads();   // thread block synchronization intrinsic
sinf, powf, atanf, ceil, min, sqrtf, ... // <math.h> functionality
```

- CUDA 主机运行时支持
    - CUDA 本质上是一种异构编程模型
        - 串行代码在CPU主机线程上运行，并行设备代码在有许多核的GPU上运行
        - 主机和设备之间通信通过PCI-Express link
        - PCI-E link 慢（高延迟，低带宽）



- 例子：向量加法

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays, fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

- CUDA 软件环境
  - nvcc 编译器的工作方式很像 icc 或 gcc：编译 C++ 源代码，生成二进制可执行文件
  - Nvidia CUDA OS 驱动程序管理与设备的低级交互，为 C++ 程序提供 API
  - Nvidia Cuda SDK 有许多代码示例演示各种 Cuda 功能；
  - 库支持不断增长：
    - CUBLAS for basic linear algebra
    - CUFFT for Fourier Fransforms
    - CULapack (3rd party proprietary) linear solvers, eigensolvers, ...
    - CAFÉ for deep learning
  - 操作系统便携：Linux、Windows、Mac OS
  - Cuda 的工业应用势头强劲
- Nvidia CUDA GPU 架构
  - CUDA 编程模型是一组对 C 的数据并行扩展，可在 GPU、CPU、FPGA 上实现
  - CUDA GPU 是流式多处理器的集合
    - 每个 SM 类似于 Multi-Core CPU 的一个核心
  - 每个 SM 都是共享控制逻辑、寄存器文件和 L1 缓存的 SIMD 执行管道（标量处理器）的集合。
- **CUDA线程分层结构**
  - Cuda 编程模型中的并行性表示为 4 级层次结构
    - Stream 是按顺序执行的 Grid 列表。 Fermi GPU 并行执行多个 Streams
    - 一个 Grid 是一组执行同一个内核的最多 2^{32} 个thread block
    - 一个线程块是一组多达 1024 个 [512 pre-Fermi] Cuda thread
    - 每个 Cuda 线程都是一个独立的、轻量级的标量执行上下文

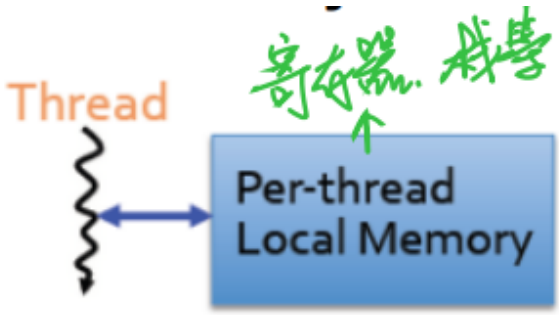
- 32 个cuda thread组形成以lockstep SIMD 执行的 Warp

- 什么是CUDA thread
  - 从逻辑上讲，每个 CUDA 线程：
    - 有自己的控制流和PC、寄存器文件、调用栈、...
    - 可以随时访问任意GPU全局内存地址
    - 可通过五个整数在网格内唯一标识：threadIdx.{x,y,z}、blockIdx.{x,y}
  - 非常精细的粒度：不要指望任何单个线程来完成昂贵计算的大部分
    - 在完全占用时，每个线程有 21 个 32 位寄存器
    - ... 1536 个线程共享一个 64 KB L1 缓存/__shared__ 内存
    - GPU 没有操作数绕过网络：功能单元延迟必须通过多线程或 ILP 隐藏（例如，来自循环展开）
  - 什么是warp
    - CUDA 处理器的逻辑 SIMD 执行宽度
    - 一组同时执行的 32 个 CUDA 线程
      - 当一个 warp 中的所有线程都执行来自同一台 PC 的指令时，执行硬件的使用效率最高。
      - 如果 warp 中的线程发散（执行不同的 PC），则某些执行管道将未使用（预测）
      - 如果warp访问中的线程对齐连续的 DRAM 块，则访问将合并为单个高带宽访问
      - 可通过将线程索引除以 32 来唯一识别
    - 从技术上讲，warp 大小可能会在未来的架构中发生变化。但是许多现有的程序会中断
  - 什么是CUDA thread block
    - 线程块是虚拟化的多线程核心
      - 标量线程、寄存器和共享内存的数量在内核调用时动态配置。
      - 由多个 (1-1024) 个 cuda 线程组成，它们都共享整数标识 blockIdx.{x,y}
    - 执行中等粒度的数据并行任务
      - 可缓存工作集应适合 128KB（64KB，per-Fermi）寄存器文件和 64KB（16KB）L1
      - 受 GPU DRAM 容量限制的不可缓存工作集
      - 块中的所有线程共享一个指令缓存
    - 块内的线程通过内部屏障同步并通过快速的片上共享内存进行通信
  - 什么是CUDA grid
    - 一组执行相关计算的线程块。
      - 单个内核调用中的所有线程都具有相同的入口点和函数参数，最初仅在 blockIdx.{x,y} 上有所不同

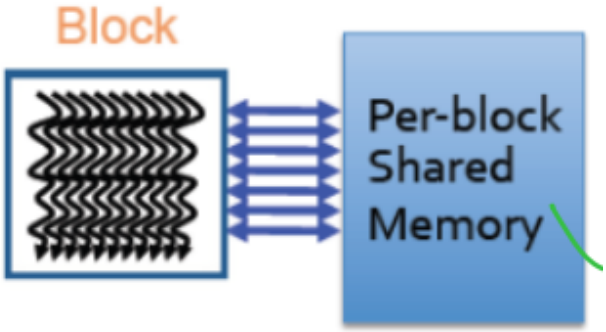
        - 网格中的 therad 块可以执行他们想要的任何代码
    - 性能可移植性/可扩展性每个网格需要许多块：每个 SM 上执行 1-8 个块
    - 内核调用的线程块必须是并行子任务
        - 程序必须对任何块执行的交错有效
        - 内存系统的灵活性在技术上允许线程块以任意方式进行通信和同步
        - block之间没什么依赖
- 什么是CUDA stream
    - 按顺序执行的一系列命令（内核调用、内存传输）。
    - 要同时执行多个内核调用或内存传输，应用程序必须指定多个流。
    - 并发内核执行只会发生在 Fermi 和更高版本上
    - 在 pre-Fermi 设备上，内存传输将与内核同时执行

```
cudaStream_t s0, s1;
cudaStreamCreate (&s0);  cudaStreamCreate (&s1);

cudaMemcpyAsync (a0, cpu_a0, N0*sizeof(float),
                 cudaMemcpyHostToDevice, s0);
vecAdd <<<N0/256, 256, 0, s0>>> (a0, b0, c0, N0);

cudaMemcpyAsync (a1, cpu_a1, N1*sizeof(float),
                 cudaMemcpyHostToDevice, s1);
vecAdd <<<N1/256, 256, 0, s1>>> (a1, b1, c1, N1);
```

- **CUDA内存分层结构**
    - 每个 CUDA 线程都可以私有访问可配置数量的寄存器
        - 128 KB (64 KB) SM 寄存器文件在所有常驻线程之间进行分区
        - 如有必要，寄存器、堆栈溢出到（缓存在 Fermi 上）“local”DRAM



    - 每个线程块都可以私有访问可配置数量的暂存器内存
        - Fermi SM 的 64 KB SRAM 可配置为 16 KB L1 缓存 + 48 KB 暂存器，反之亦然*
        - Pre-Fermi SM 只有 16 KB 暂存器
        - 可用的暂存器空间在常驻线程块之间进行分区，提供了另一种并发状态权衡



- 所有网格中的线程块共享对大型“global”内存池的访问，与主机 CPU 的内存分开。
  - 全局内存保存应用程序的持久状态，而线程本地和块本地内存是临时的
  - 全局内存比片上内存代价贵得多：O(100)x 延迟，O(1/50)x（聚合）带宽
  - 在 Fermi 上，全局内存缓存在 768KB 共享 L2 中
- 以下了解——其他内存
  - 由于 Cuda 的图形遗产，存在内存层次结构的其他只读组件
  - 64 KB Cuda 常量内存与全局内存位于同一 DRAM 中，但通过特殊的只读 8 KB per-SM 缓存访问
  - Cuda 纹理内存也驻留在 DRAM 中，并通过小型每 SM 只读缓存访问，但还包括插值硬件 该硬件对图形性能至关重要，但仅偶尔对通用工作负载有用
  - 这些缓存的行为针对它们在图形工作负载中的作用进行了高度优化
- 系统中的每个 CUDA 设备都有自己的全局内存，与主机 CPU 内存分开
- 通过 cudaMalloc()/cudaFree() 分配和释放
- 主机 <-> 设备内存传输是通过 PCI-E 上的 cudaMemcpy() 进行的，并且非常昂贵
  - 微秒延迟，~GB/s 带宽
- 通过多个 CPU 线程管理的多个设备
- 线程块同步
  - 块内屏障指令__syncthreads() 同步共享内存和全局内存
    - 保证正确性，在读取其他线程写的值之前一定要__syncthreads()
    - 在一个块内的所有线程必须执行相同的__syncthreads()，否则GPU将会中断
    - 其他内在函数

```
- int __syncthreads_count(int), int __syncthreads_and(int),
  int __syncthreads_or(int)

extern __shared__ float T[];
__device__ void
transpose (float* a, int lda){
    int i = threadIdx.x, j = threadIdx.y;
    T[i + lda*j] = a[i + lda*j];
    __syncthreads();
    a[i + lda*j] = T[j + lda*i];
}
```

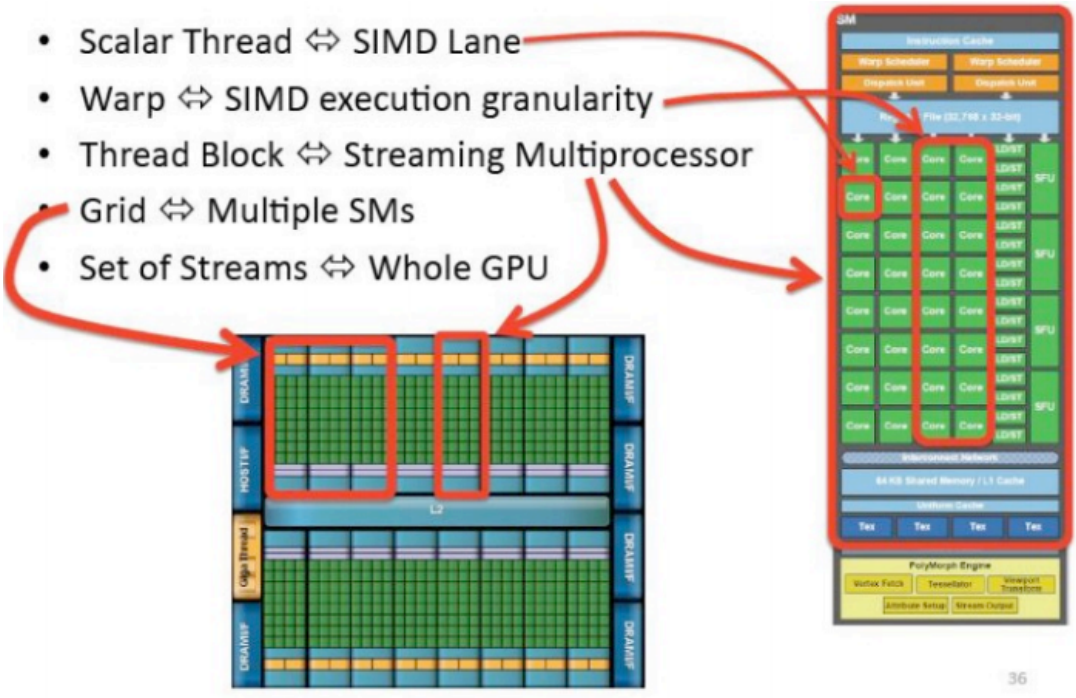
- 使用每块的共享内存

- 每块共享内存/L1 缓存是一种关键资源：没有它，大多数 CUDA 程序的性能将无可救药地受限于 DRAM
  - Block-shared variables can be declared statically:
    ```
    __shared__ int begin, end;
    ```
  - Software-managed scratchpad is allocated statically:
    ```
    __shared__ int scratch[128];
    scratch[threadIdx.x] = ... ;
    ```
  - ... or dynamically:
    ```
    extern __shared__ int scratch[];

    kernel_call <<< grid_dim, block_dim, scratch_size >>> ( ... );
    ```
  - Most intra-block communication is via shared scratchpad:
    ```
    scratch[threadIdx.x] = ...;
    __syncthreads();
    int left = scratch[threadIdx.x - 1];
    ```
  - 每个 SM 有 64 KB 的私有内存，将 16KB/48KB（或 48KB/16KB）分为硬件管理的暂存器和硬件管理的非连贯缓存
  - Pre-Fermi，SM 内存只有 16 KB，并且只能用作硬件管理的暂存器
  - 除非数据将在块中的线程之间共享，否则它应该驻留在寄存器中
  - 在 Fermi 上，128 KB 的寄存器文件是两倍大，并且可以在更高的带宽和更低的延迟下访问
  - Pre-Fermi，寄存器文件为 64 KB，与暂存器一样快
- **共享内存块冲突**
  - 共享内存被分块：它由 32 个（16 个，pre-Fermi）独立可寻址的 4 字节宽内存组成
    - 地址交错：float *p 指向 bank k 中的浮点数，p+1 指向 bank (k+1) mod 32 中的浮点数
  - 每个 bank 可以满足每个周期的单个 4 字节访问
    - 当两个线程（在同一个 warp 中）尝试在给定周期内访问同一个 bank 时，就会发生 bank 冲突
    - GPU 硬件会串行执行这两个访问，warp 的指令将需要一个额外的周期来执行
  - 存储区冲突是二阶性能影响：即使是对片上共享内存的串行访问也比对片外 DRAM 的访问要快
  - 步长的影响
    - Unit-Stride 访问是无冲突的
    - Stride-2 访问：线程 n 与线程 16+n 冲突
    - Stride-3 访问是无冲突的
    - 三个无冲突案例
    - 32-float 块内的排列是可以的
    - 多个线程读取一个bank相同的内存地址

        - 所有线程读取相同内存地址会产生一次广播
    - 原子的内存操作

```
int atomicAdd (int*,int), float atomicAdd (float*, float), ...
...
int atomicMin (int*,int),
...
int atomicExch (int*,int), float atomicExch
(float*,float), ...
int atomicCAS (int*, int compare, int val), ...
```

        - CUDA 提供了一组相互原子执行的指令
        - 允许对共享或全局内存中线程之间共享的变量进行非只读访问
        - 比标准加载/存储贵得多
        - 具有一致性，可以实现例如自旋锁!
    - 向量内存一致性
        - 默认情况下，您不能假设内存访问以程序指定的相同顺序发生。尽管线程自己的访问对该线程似乎是按程序顺序发生的；
        - 要强制排序，请使用*memory fence*指令
            - __threadfence_block()：使所有以前的内存访问对线程块内的所有其他线程可见
            - __threadfence()：使设备上的所有其他线程可以看到以前的全局内存访问
        - 经常还必须使用 volatile 类型限定符
            - 具有与 CPU C/C++ 相同的行为：禁止编译器在易失性内存中进行寄存器提升值
        - 确保指针取消引用产生加载/存储指令
            - 声明为 volatile float *p; *p 必须产生一个内存引用。
- CUDA映射到Nvidia GPUs



- Cuda 被设计为“功能上的宽容”：很容易让正确的程序运行。 您在优化代码上投入的时间越多，您获得的性能就越高
- 使用简单的“同质 SPMD”方法来编写 Cuda 程序可以加快速度
- 实现性能需要了解 Cuda 的硬件实现
- 每一个GPU层级都对应一个内存资源
  - Scalar Threads / Warps: Subset of register file
  - Thread Block / SM: shared memory (l1 Cache)
  - Multiple SMs / Whole GPU: Global DRAM
- 大规模多线程用于隐藏延迟：DRAM 访问、功能单元执行、PCI-E 传输
- 一个高性能的 CUDA 程序必须谨慎地用资源使用来换取并发性

  More registers per thread $\Leftrightarrow$ fewer threads

  More shared memory ber block $\Leftrightarrow$ fewer blocks
- 多核处理器 = 将计算限制问题转化为内存限制问题的设备
- 内存也是 SIMD！ CPU 和 GPU 的内存系统都需要在对齐的块中访问内存

- CUDA 总结
  - CUDA 编程模型提供了一种为异构、分层平台组织数据并行程序的通用方法
  - 目前，唯一的生产质量实现是 Nvidia GPU 上 C/C++ 的 CUDA
  - 但是“标量线程”、“warp”、“块”和“网格”的 CUDA 概念也可以映射到其他平台
  - CUDA 编程的简单“同质 SPMD”方法很有用，尤其是在实施和调试的早期阶段
  - 但要实现高效率，需要仔细考虑从计算到处理器、数据到内存以及数据访问模式的映射

以上内容整理于 幕布文档