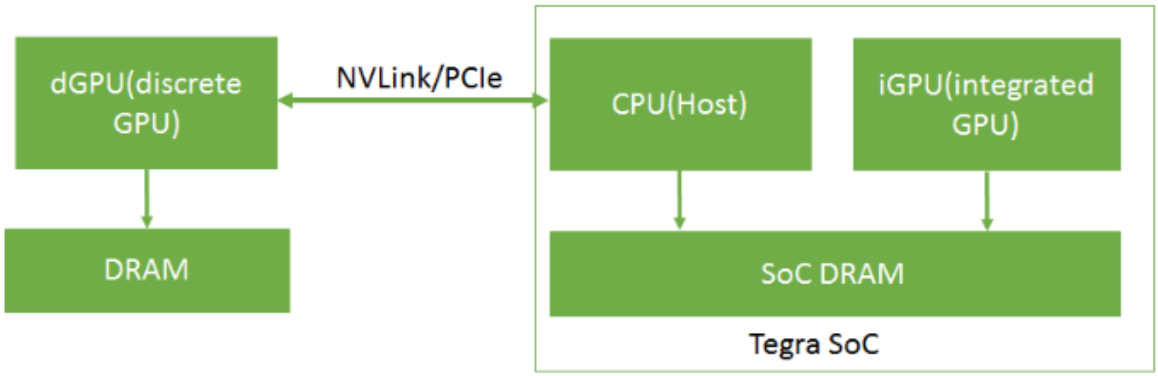
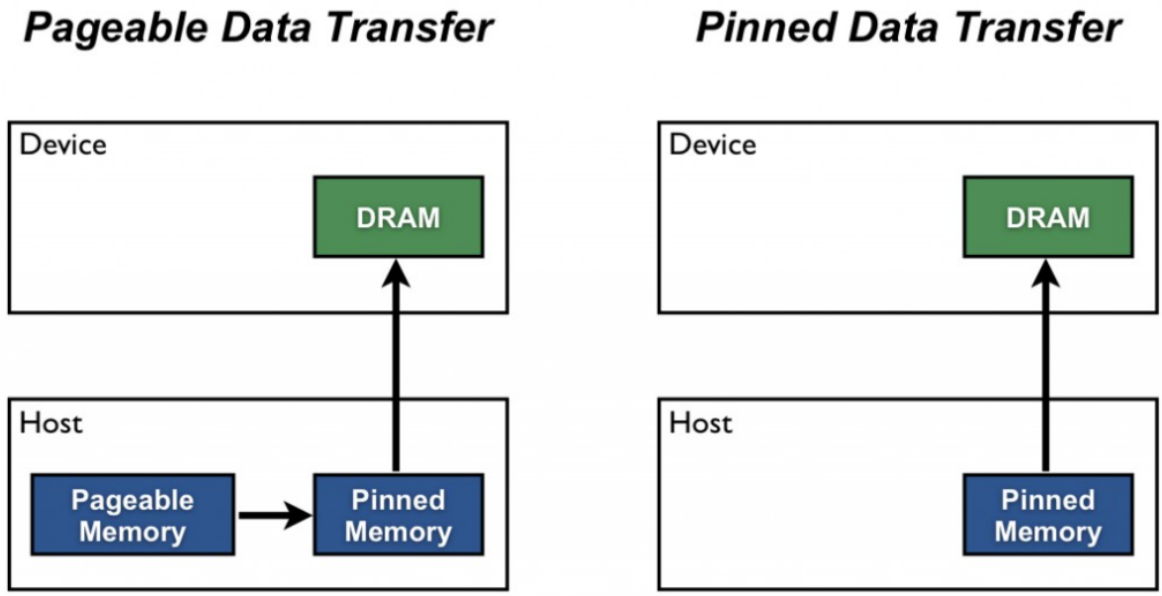
基础板载结构 和 存储体系结构

device memory, host memory, unified memory allocated on the same physical SoC DRAM



I/O一致性(也称为单向一致性)是GPU等I/O设备可以读取CPU缓存中的最新更新的特性。 当CPU和GPU共享相同的物理内存时，不需要进行CPU缓存管理操作。 GPU缓存管理操作仍然需要执行，一致性是单向方法。



Pinned memory(zero-copy) is recommended for small buffers because the caching effect is negligible for such buffers and also because pinned memory does not involve any additional overhead, unlike Unified Memory. With no additional overhead, pinned memory is also preferable for large buffers if the access pattern is not cache friendly on iGPU. For large buffers, when the buffer is accessed only once on iGPU in a coalescing manner, performance on iGPU can be as good as unified memory on iGPU.

GPU通过DMA访问pinned memory。

On Tegra® devices with I/O coherency (with a compute capability of 7.2 or greater) where unified memory is cached on both CPU and iGPU, for large buffers which are frequently accessed by the iGPU and the CPU and the accesses on iGPU are repetitive, unified memory is preferable since repetitive accesses can offset the cache maintenance cost.

CUDA基础

```
add<<<n, m>>>(N, x, y);
// 第二个参数表示 一个线程块中的线程数，CUDA运行时使用的线程块的大小是32的倍数
// 第一个参数表示 线程块的数量
```

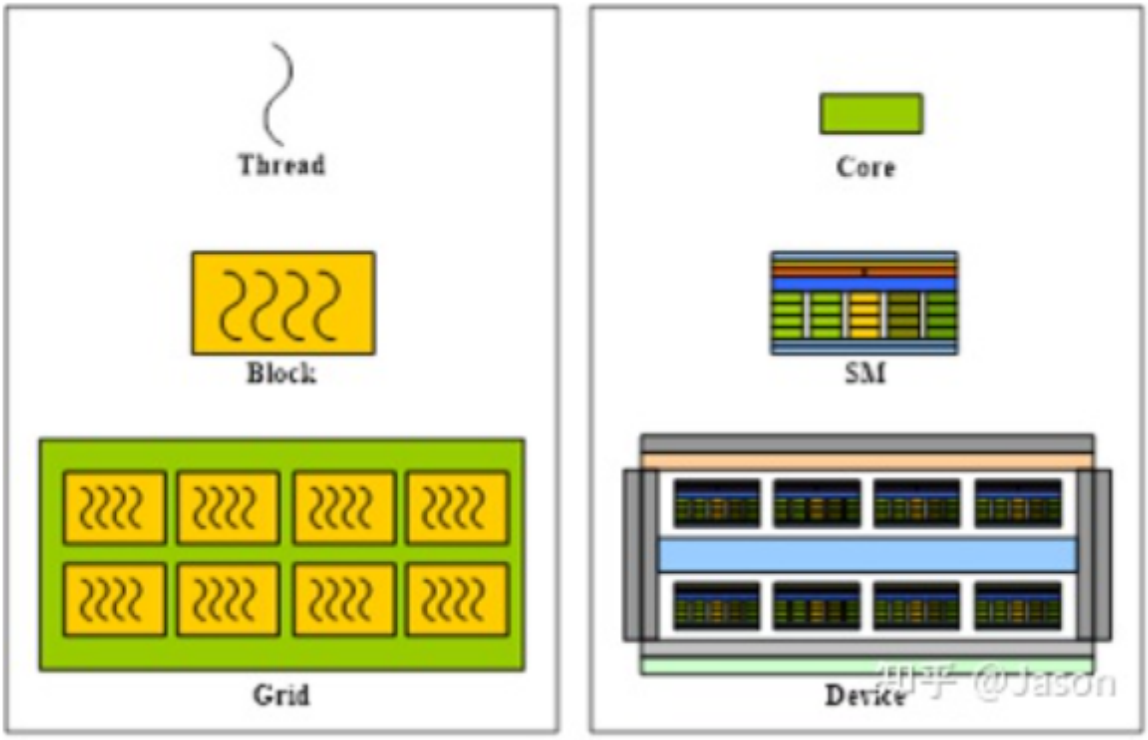
```
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}

add<<<1, 1>>>(N, x, y); // N个数相加，单核上跑单线程
```

```
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  /* threadIdx.x包含当前线程在其块中的索引，blockDim.x包含块中的线程数 */
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}

add<<<1, 256>>>(N, x, y);
```

CUDA GPU有许多并行处理器，它们被分组成 流多处理器（Streaming Multiprocessors, or SMs）。每个SM可以运行多个并发线程块。




```
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x; //blockIDx.x 表示当前线程块的编号
  int stride = blockDim.x * gridDim.x; //gridDim.x表示线程块的数量
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}

int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);
```

```
    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

Pinned Memory

pinned memory即page-locked host memory，有几个优点：

- pinned memory和device memory之间的拷贝 与 内核计算某些情况下可以并行
- 某些设备上pinned memory可以被编址到device memory的地址空间，免去了数据拷贝。因此，这样的内存块通常有两个地址:一个在主机内存中，由cudaHostAlloc()或malloc()返回，另一个在设备内存中，可以通过cudaHostGetDevicePointer()获取，然后用于从内核中访问块。 这里还需要注意内存同步问题，避免写后读等hazard。

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);
```

```
    // Allocate input vectors h_A and h_B in pinned memory
    float* h_A = (float*)cudaHostAlloc(size);
    float* h_B = (float*)cudaHostAlloc(size);
    float* h_C = (float*)cudaHostAlloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free pinned memory
    cudaFreeHost(h_A);
    cudaFreeHost(h_B);
    cudaFreeHost(h_C);

    return 0;
}
```

Unified Memory

```
#include <iostream>
#include <math.h>

// CUDA kernel to add elements of two arrays
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
```

```
    y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y;

  // Allocate Unified Memory -- accessible from CPU or GPU
  // cudaError_t cudaMallocManaged(void** ptr, size_t size);
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Launch kernel on 1M elements on the GPU
  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  add<<<numBlocks, blockSize>>>(N, x, y);

  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  cudaFree(x);
  cudaFree(y);

  return 0;
}
```

当运行在CPU或GPU上的代码访问以这种方式分配的数据(通常称为CUDA管理数据)时，CUDA系统软件和/或硬件负责将内存页迁移到**访问处理器**的内存中。

Pascal及以上版本的GPU架构通过其**页面迁移引擎**支持虚拟内存页面故障和迁移，添加49位虚拟内存地址和on-demand page migration的特性。49位虚拟地址足以使GPU访问整个系统内存以及系统中所有GPU的内存。**页面迁移引擎**允许GPU线程在非常驻内存访问时出现缺页，因此系统可以按需将页面从系统中的任何地方迁移到GPU内存中，从而实现高效处理。

一旦达到GPU内存限制，driver开始踢出旧page。这是按需发生的，page fault会产生显著的时间开销。

分配内存cudaMallocManaged()具体发生了什么？（pascal架构以上）

在函数返回时并不会被物理的分配，而是在访问（或预抓取，prefetching）时创建。页面可以在任何时候迁移到任何处理器的内存中，驱动程序使用启发式方法来维护数据局部性和防止过多的页面错误。（应用程序可以使用cudaMemAdvise()来指导驱动程序，并使用cudaMemPrefetchAsync()显式地迁移内存。）

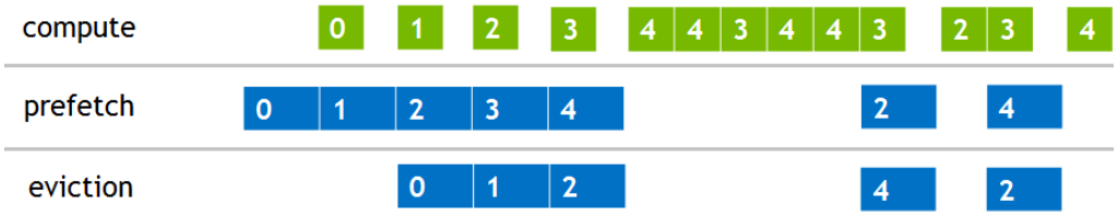
Tesla P100支持hardware page faullting 和 migration。 所以在这种情况下，运行时在运行内核之前不会自动将所有页面复制回GPU。 因此内核启动时没有任何迁移开销，当它访问任何缺失的页面时，GPU暂停访问线程的执行，页面迁移引擎在恢复线程之前将页面迁移到设备上。 这意味着page migration包括在了内核执行时间内。

这会导致host to device产生大量page fault。解决方案有三种：

- 将数据初始化转移到另一个CUDA内核的GPU上，避免page fault

```
__global__ void init(int n, float *x, float *y) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
}
```

- 运行内核多次，查看平均运行时间和最小运行时间。这是因为运行几次后基本不会出现page fault
- 运行内核前，需要预取数据到GPU内存中 (prefetching)。预取操作可以在单独的CUDA流（非阻塞流？）中执行，并与一些在GPU上执行的计算工作重叠。 cudaMemPrefetchAsync() 对GPU来说时一个异步操作，CPU仍然可能参与其中，因为它需要启动页面迁移和更新页面映射。 因此，为了更好地与GPU内核重叠，我们需要确保提前安排了足够的GPU工作。




```
// Prefetch the data to the GPU after initializing
int device = -1;
cudaGetDevice(&device);
cudaMemPrefetchAsync(x, N*sizeof(float), device, NULL);
cudaMemPrefetchAsync(y, N*sizeof(float), device, NULL);

// Run kernel on 1M elements on the GPU
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
saxpy<<<numBlocks, blockSize>>>(N, 1.0f, x, y);
```

有时，创建多个数据副本或pin page到系统内存并启用零拷贝访问是有用的。CUDA 8提供了新的cudaMemAdvise() API，它提供了一组内存使用提示，允许对托管分配进行更细粒度的控制。

TODO

In the **hybrid multigrid code** there are inter-communications between CPU and GPU levels, such as **restriction and interpolation kernels**. The same data is accessed by two processors many times within a single multigrid cycle.

一旦数据被移动，就不存在重用，而且处理页面错误的开销超过了将数据保留在处理器本地的任何好处。

为了优化这一阶段，可以将区域固定到CPU内存中，并通过分别使用cudaMemAdviseSetPreferredLocation 和 cudaMemAdviseSetAccessedBy 的使用提示来建立从GPU的直接映射:

```
cudaMemAdvise(ptr, size, cudaMemAdviseSetPreferredLocation,
cudaCpuDeviceId);
cudaMemAdvise(ptr, size, cudaMemAdviseSetAccessedBy, myGpuId);
```

CUDA stream

流，是一系列顺序执行的指令。不同的流执行可能是乱序的。当执行没有指定流的指令时，系统使用默认流。分配给同一个流的指令将顺序执行。CUDA 7版本可以为每个host thread使用一个独立的默认流(per-thread default stream)，让不同线程发起的流可以并行。

创建、设置、销毁 stream

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);

// kernel<<< blocks, threads, bytes >>>();    // default stream
// kernel<<< blocks, threads, bytes, 0 >>>(); // stream 0

for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}

for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

使用per-thread default stream

调用nvcc时加上 --default-stream per-thread 且 在cuda.h or cuda_runtime.h之前定义#define CUDA_API_PER_THREAD_DEFAULT_STREAM 宏

non-blocking thread

```
__host__•__device__•cudaError_t cudaStreamCreateWithFlags ( cudaStream_t*
pStream, unsigned int  flags )
// The flags argument determines the behaviors of the stream. Valid values for
flags are
// cudaStreamDefault: Default stream creation flag.
// cudaStreamNonBlocking: Specifies that work running in the created stream may
run concurrently with work in stream 0 (the NULL stream), and that the created
stream should perform no implicit synchronization with stream 0.
```

```
// default stream
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a) //不阻塞host thread
myCpuFunction(b) //myCpuFunction 与 kernel 并行执行
// cpu运算与kernel运算谁先完成不重要，kernel先完成会回到cpu这继续执行
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

```
// non-default stream
cudaStream_t stream1;
cudaError_t result;
result = cudaStreamCreate(&stream1);
result = cudaStreamDestroy(stream1);
result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1)
increment<<<1,N,0,stream1>>>(d_a)
```

stream 同步

- cudaStreamSynchronize()；block the host thread until all previously issued operations in the specified stream have completed
- cudaStreamQuery()；tests whether all operations issued to the specified stream have completed, without blocking host execution
- cudaStreamWaitEvent();

注：当多个内核指令在不同的(非默认的)流中**连续**发出时，调度器会尝试启用这些内核的并发执行，结果会在每次内核完成后(负责启动设备到主机的传输)延迟一个信号，直到所有的内核完成。

CUDA graph

CUDA图提出了一种新的CUDA工作提交模型。 图是一系列操作，比如内核启动，通过依赖项连接，这些依赖项是独立于执行定义的。 这允许一个图被定义一次，然后重复启动。 将图的定义与执行分离可以实现许多优化:首先，与流相比，CPU启动成本降低了，因为很多设置都是提前完成的; 第二，将整个工作流呈现给CUDA可以实现优化，这可能是用流的分段工作提交机制无法实现的。

要查看图形的优化，考虑流中发生的情况:当你将内核放入流中时，主机驱动程序执行一系列操作，为GPU上的内核执行做准备。 这些操作是设置和启动内核所必需的，是必须为每个发布的内核支付的开销。 对于执行时间较短的GPU内核，这种开销成本可能占到整个端到端执行时间的很大一部分。

使用图的工作提交分为三个不同的阶段:定义、实例化和执行：

- 在定义阶段，程序创建图中操作的描述以及它们之间的依赖关系。
- 实例化获取图形模板的快照，验证它，并执行大量的设置和初始化工作，目的是将启动时需要做的事情最小化。 产生的实例称为可执行图。
- 一个可执行图可以被启动到流中，类似于任何其他CUDA工作。 它可以被启动任意次而不重复实例化。

operation形成图中的节点。operation之间的依赖关系是边。这些依赖关系约束操作的执行顺序。一旦操作所依赖的节点完成，就可以随时计划操作。调度由CUDA系统决定。

节点可以是：内核、CPU函数call、内存复制、memset、empty、等待event、记录event、子图、等待external semaphore、发送external semaphore

图可以通过两种机制创建:显式API和流捕获。

```
// 显式API创建图
// Create the graph - it starts out empty
```

```
cudaGraphCreate(&graph, 0);

// For the purpose of this example, we'll create
// the nodes separately from the dependencies to
// demonstrate that it can be done in two stages.
// Note that dependencies can also be specified
// at node creation.
cudaGraphAddKernelNode(&a, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&b, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&c, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&d, graph, NULL, 0, &nodeParams);

// Now set up dependencies on each node
cudaGraphAddDependencies(graph, &a, &b, 1);     // A->B
cudaGraphAddDependencies(graph, &a, &c, 1);     // A->C
cudaGraphAddDependencies(graph, &b, &d, 1);     // B->D
cudaGraphAddDependencies(graph, &c, &d, 1);     // C->D
```

```
// 流捕获创建图
cudaGraph_t graph;

// 调用cudaStreamBeginCapture()将流置于捕获模式。 当一个流被捕获时，启动到流中的工作不会排队等待执行。 相反，它被追加到逐步构建的内部图中。
cudaStreamBeginCapture(stream);

kernel_A<<< ..., stream >>>(...);
kernel_B<<< ..., stream >>>(...);
libraryCall(stream);
kernel_C<<< ..., stream >>>(...);

// 调用cudaStreamEndCapture()返回这个图，它也结束了流的捕获模式。
cudaStreamEndCapture(stream, &graph);
```

流捕获可以处理cudaEventRecord()和cudaStreamWaitEvent()表示的跨流依赖关系，前提是正在等待的事件被记录到同一个捕获图中。

When an event is recorded in a stream that is in capture mode, it results in a *captured event.* A captured event represents a set of nodes in a capture graph.

当流等待捕获的事件时，它会将流置于捕获模式(如果还没有的话)，流中的下一项将对捕获事件中的节点具有额外的依赖关系。 然后将这两个流捕获到相同的捕获图中。

A graph is a snapshot of a workflow, including kernels, parameters, and dependencies, in order to replay it as rapidly and efficiently as possible.

CUDA并行

CUDA的一些独立的进程的操作可以并行执行：

- host计算
- device计算
- host2device的内存传输
- device2host的内存传输
- device间的内存传输
- 给定device间的内存传输

以下device操作对host来说是异步的：

- 内核启动
- 内存在单个device的内存中复制
- 以Async为后缀的内存复制函数
- 内存设置函数调用

Async memory copies will also be synchronous if they involve host memory that is not page-locked.

CUDA三种并行方式：

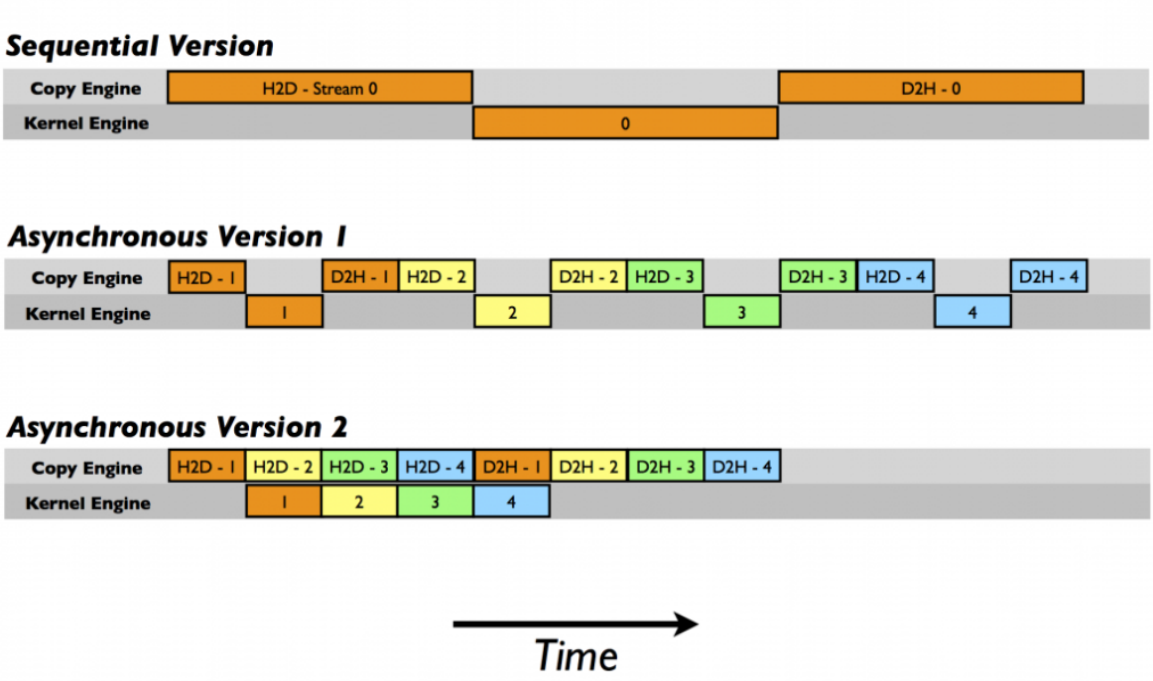
- 内核并行：CUDA内核可以并行，来自一个CUDA上下文的内核不能与来自另一个CUDA上下文的内核并发执行。
- 内核与内存复制并行：一些设备可以在内核执行的同时以GPU为原点或目标进行异步内存复制，如果复制过程涉及到host端，host memory必须是pinned memory。也可以在内核执行的同时和/或在设备之间进行复制 。
- 内存复制并行：一些设备可以device为原点或目标进行重叠内存复制，如果复制过程涉及到host端，host memory必须是pinned memory。

并行示例：

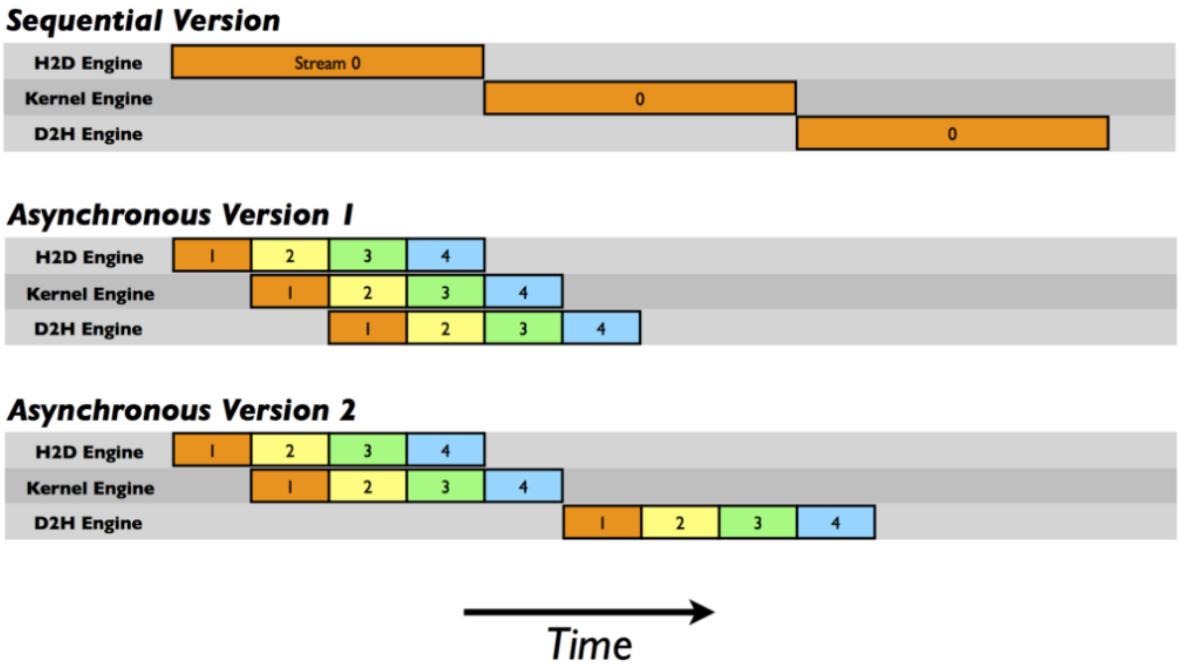
```
//Asynchronous Version 1
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
```

```
//Asynchronous Version 1
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    MyKernel<<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
```

不支持内存复制并行的设备：



支持内存复制并行的设备：



提高并行能力的指导思想：

- 所有独立操作都应该在有依赖的操作之前发出
- 任何类型的同步都应该尽可能地延迟

stream可以设置优先级。At runtime, pending work in higher-priority streams takes preference over pending work in low-priority streams. （？）

当调用同步函数时，在设备完成请求的任务之前，控制不会返回给主机线程。 在host线程执行任何其他CUDA调用之前，可以通过调用带有特定标志的cudaSetDeviceFlags()来指定host线程是yield、block还是spin。

测试样例

- baseline
- pinned memory
- unified memory
  - baseline
  - prefetch
- blocking vs non-blocking thread
- cudasetDeviceFlags：yield、spin、block