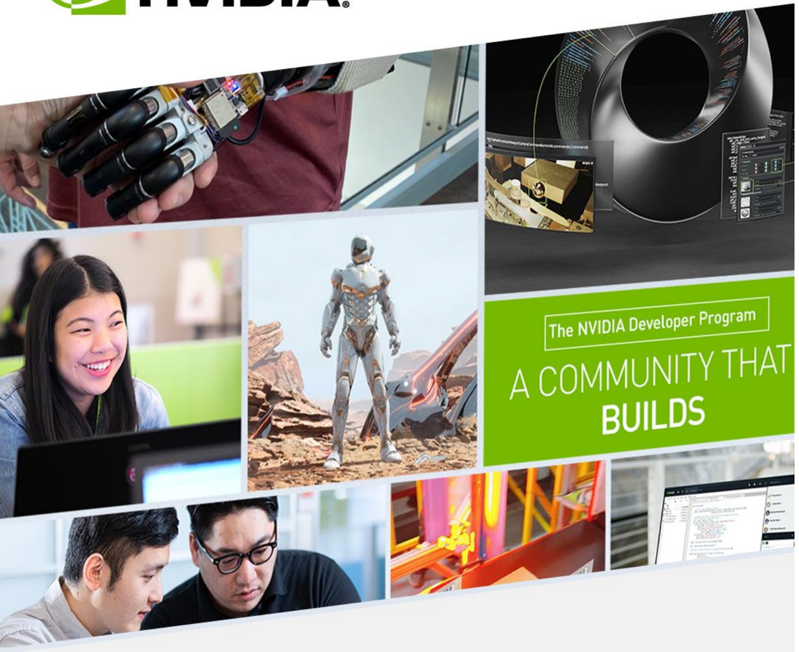
CUDA ON ARM PLATFORM—错误检测, 事件和统一内存

NVIDIA企业级开发者社区 何琨



加入 NVIDIA 开发者计划

获取最新版本软件、工具及开发信息

扫码了解详情

错误检测与事件

- CUDA应用程序运行时的错误检测
- CUDA中的事件
- 利用事件进行计时
- CUDA中的统一内存

CUDA的运行时检测函数

```
__host____device__const char*     cudaGetErrorName ( cudaError_t error )
Returns the string representation of an error code enum name.

__host____device__const char*     cudaGetErrorString ( cudaError_t error )
Returns the description string for an error code.

__host____device__cudaError_t     cudaGetLastError ( void )
Returns the last error from a runtime call.

__host____device__cudaError_t     cudaPeekAtLastError ( void )
Returns the last error from a runtime call.
```

CUDA的运行时检测函数

返回错误代码:

__host____device__cudaError_t	cudaGetLastError (void)
__host____device__cudaError_t	cudaPeekAtLastError (void)

返回错误描述:

__host____device__const char*	cudaGetErrorName (cudaError_t error)
__host____device__const char*	cudaGetErrorString (cudaError_t error)

CUDA的运行时检测函数

```
const int block_size = 128;

const int grid_size = (N + block_size - 1) / block_size;

add<<<grid_size, block_size+1024>>>(d_x, d_y, d_z, N);

printf("cuda error:%s \n", cudaGetErrorString(cudaGetLastError()));
printf("cuda error:%s \n", cudaGetErrorName(cudaPeekAtLastError()));
```

```
Output:
cuda error:invalid configuration argument
cuda error:cudaSuccess
Errors
```

CUDA的运行时检测函数

```
#pragma once
#include <stdio.h>

#define CHECK(call)                                   \
do                                                    \
{                                                     \
    const cudaError_t error_code = call;              \
    if (error_code != cudaSuccess)                    \
    {                                                 \
        printf("CUDA Error:\n");                      \
        printf("    File:       %s\n", __FILE__);     \
        printf("    Line:       %d\n", __LINE__);     \
        printf("    Error code: %d\n", error_code);   \
        printf("    Error text: %s\n",                \
            cudaGetErrorString(error_code));          \
        exit(1);                                      \
    }                                                 \
} while (0)
```

CUDA的运行时检测函数

```
#pragma once
#include <stdio.h>

#define CHECK(call)                                                     \
do                                                                      \
{                                                                       \
    const cudaError_t error_code = call;                                \
    if (error_code != cudaSuccess)                                      \
    {                                                                   \
        printf("CUDA Error:\n");                                        \
        printf("    File:       %s\n", __FILE__);                       \
        printf("    Line:       %d\n", __LINE__);                       \
        printf("    Error code: %d\n", error_code);                     \
        printf("    Error text: %s\n",                                  \
            cudaGetErrorString(error_code));                            \
        exit(1);                                                        \
    }                                                                   \
} while (0)
```

```
CHECK(cudaMemcpy(d_b, h_b, sizeof(int)*n*k, cudaMemcpyHostToDevice));
```

CUDA的事件(event)

之前的课程中，我们已经讨论过，如何利用CUDA加速矩阵相乘的例子。
那么，我们如何判断加速比，如何计时呢?

CPU TIMER?

CUDA的事件(event)

CUDA event 本质是一个GPU时间戳，这个时间戳是在用户指定的时间点上记录的。由于GPU本身支持记录时间戳，因此就避免了当使用CPU定时器来统计GPU执行时间时可能遇到的诸多问题。

CUDA的事件(event)

__host__cudaError_t cudaEventCreate (cudaEvent_t* event)
Creates an event object.

__host____device__cudaError_t cudaEventDestroy (cudaEvent_t event)
Destroys an event object.

__host__cudaError_t cudaEventElapsedTime (float* ms, cudaEvent_t start, cudaEvent_t end)
Computes the elapsed time between events.

__host____device__cudaError_t cudaEventRecord (cudaEvent_t event, cudaStream_t stream = 0)
Records an event.

__host__cudaError_t cudaEventSynchronize (cudaEvent_t event)
Waits for an event to complete.

CUDA的事件(event)

声明:
cudaEvent_t event;

创建：
cudaError_t cudaEventCreate(cudaEvent_t* event);

销毁：
cudaError_t cudaEventDestroy(cudaEvent_t event);

添加事件到当前执行流：
cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream = 0);

等待事件完成，设立flag：
cudaError_t cudaEventSynchronize(cudaEvent_t event);//阻塞
cudaError_t cudaEventQuery(cudaEvent_t event);//非阻塞

当然，我们也可以用它来记录执行的事件：
cudaError_t cudaEventElapsedTime(float* ms, cudaEvent_t start, cudaEvent_t stop);

cudaEventRecord()视为一条记录当前时间的语句，并且把这条语句放入GPU的未完成队列中。因为直到GPU执行完了在调用cudaEventRecord()之前的所有语句时，事件才会被记录下来。且仅当GPU完成了之前的工作并且记录了stop事件后，才能安全地读取stop时间值。

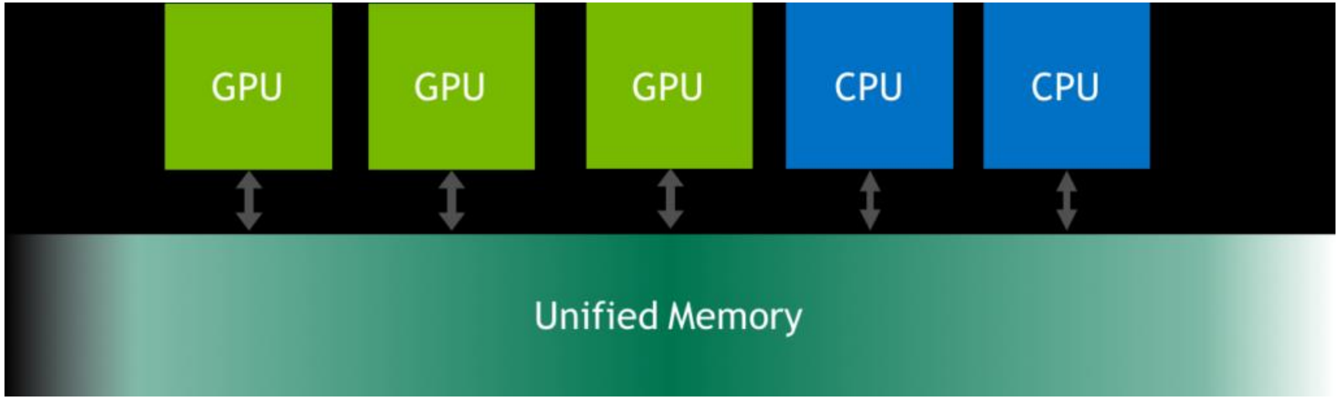
CUDA的事件(event)

```
cudaEvent_t     start, stop;

cudaEventCreate( &start );
cudaEventCreate( &stop ) ;
cudaEventRecord( start) ;

//在GPU上执行的一些操作
//
//.........................

cudaEventRecord( stop)
cudaEventSynchronize( stop );

float   elapsedTime;
cudaEventElapsedTime( &elapsedTime,start, stop ) );

printf( "Time to generate:  %.2f ms\n", elapsedTime );

cudaEventDestroy( start );
cudaEventDestroy( stop );
```

统一内存的基本概念

Unified Memory:

- 统一内存是可从系统中的任何处理器访问的单个内存地址空间。这种硬件/软件技术允许应用程序分配可以从CPU s 或 GPUs 上运行的代码读取或写入的数据。分配统一内存非常简单，只需将对 malloc() 或 new 的调用替换为对 cudaMallocManaged() 的调用，这是一个分配函数，返回可从任何处理器访问的指针。



统一内存的基本概念

Unified Memory:

CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

统一内存的基本概念

Unified Memory: 两种实现方法

1. cudaError_t cudaMallocManaged(void **devPtr, size_t size, unsigned int flags=0);

2. __managed__

统一内存的基本概念

Unified Memory: 两种实现方法

1. cudaError_t cudaMallocManaged(void **devPtr, size_t size, unsigned int flags=0);

```
__global__ void printme(char *str) {
    printf(str);
}
int main() {

    char *s;
    cudaMallocManaged(&s, 100);
    strncpy(s, "Hello Unified Memory\n", 99);
    printme<<< 1, 1 >>>(s);
    cudaDeviceSynchronize();
    cudaFree(s);
    return  0;
}
```

统一内存的基本概念

Unified Memory: 两种实现方法

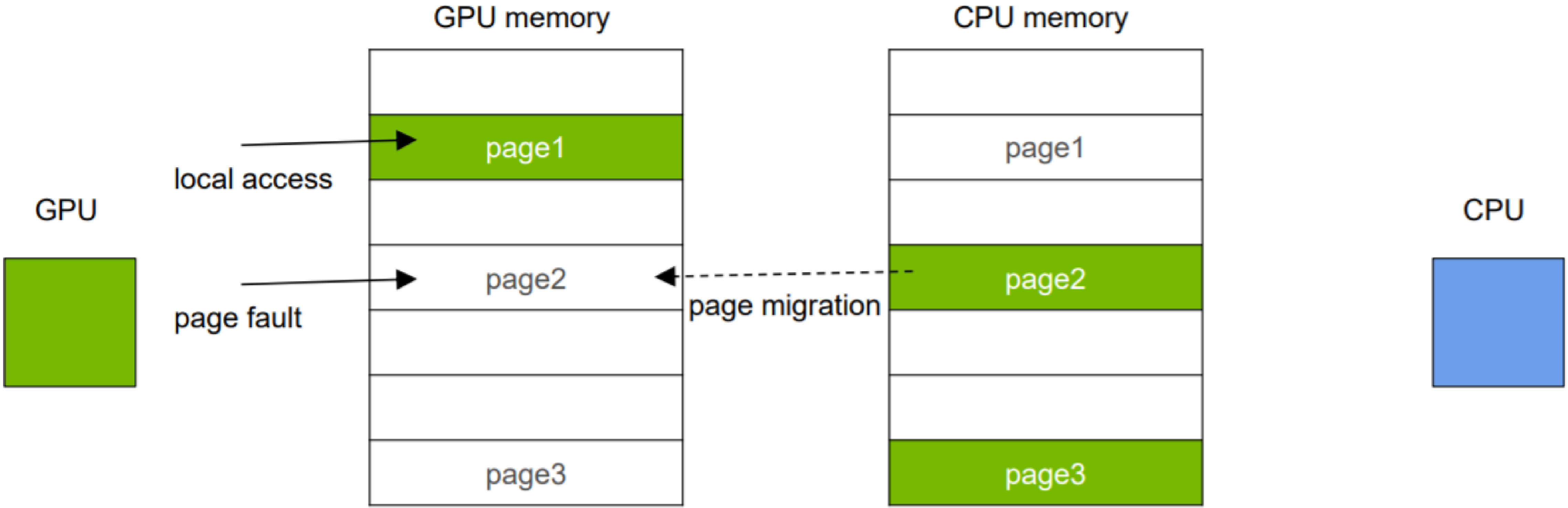
2. __managed__

```
__device__ __managed__ int x[2];
__device__ __managed__ int y;
__global__ void kernel() {
    x[1] = x[0] + y;
}
int main() {
    x[0] = 3;
    y = 5;
    kernel<<< 1, 1 >>>();
    cudaDeviceSynchronize();
    printf("result = %d\n", x[1]);
    return  0;
}
```

File-scope and global-scope CUDA __device__ variables may also opt-in to Unified Memory management by adding a new __managed__ annotation to the declaration. These may then be referenced directly from either host or device code

统一内存的基本概念

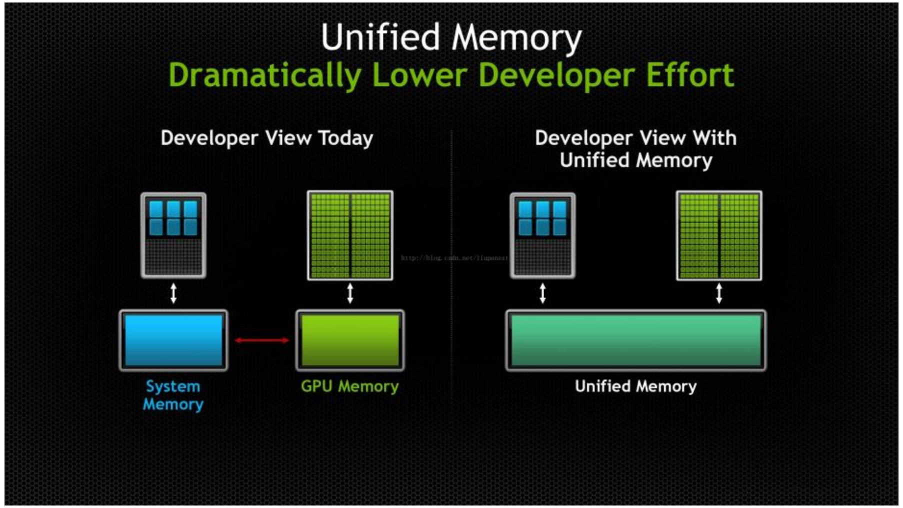
Unified Memory:

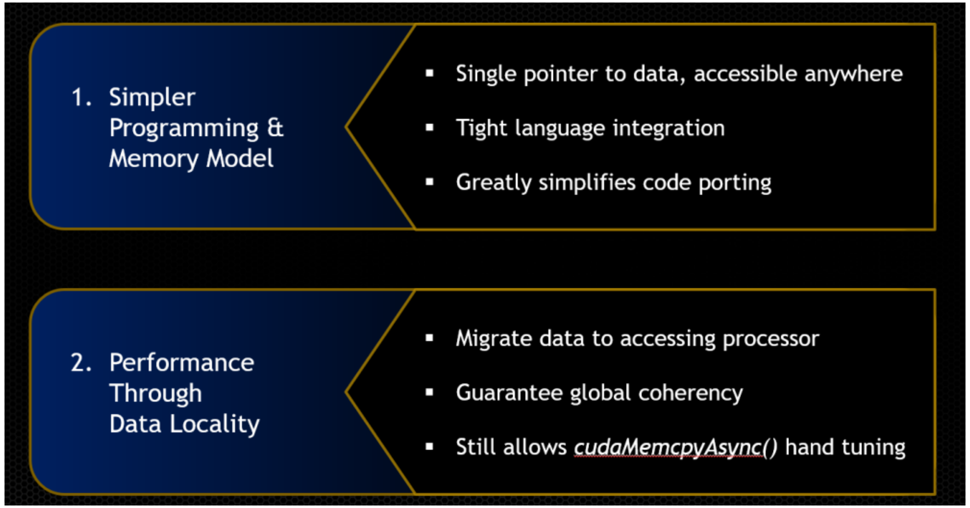


统一内存的基本概念

Unified Memory:

- 可直接访问CPU内存、GPU显存，不需要手动拷贝数据。
- CUDA 在现有的内存池结构上增加了一个统一内存系统，程序员可以直接访问任何内存/显存资源，或者在合法的内存空间内寻址，而不用管涉及到的到底是内存还是显存。
- CUDA 的数据拷贝由程序员的手动转移，变成自动执行，因此，它仍然受制于PCI-E的带宽和延迟。



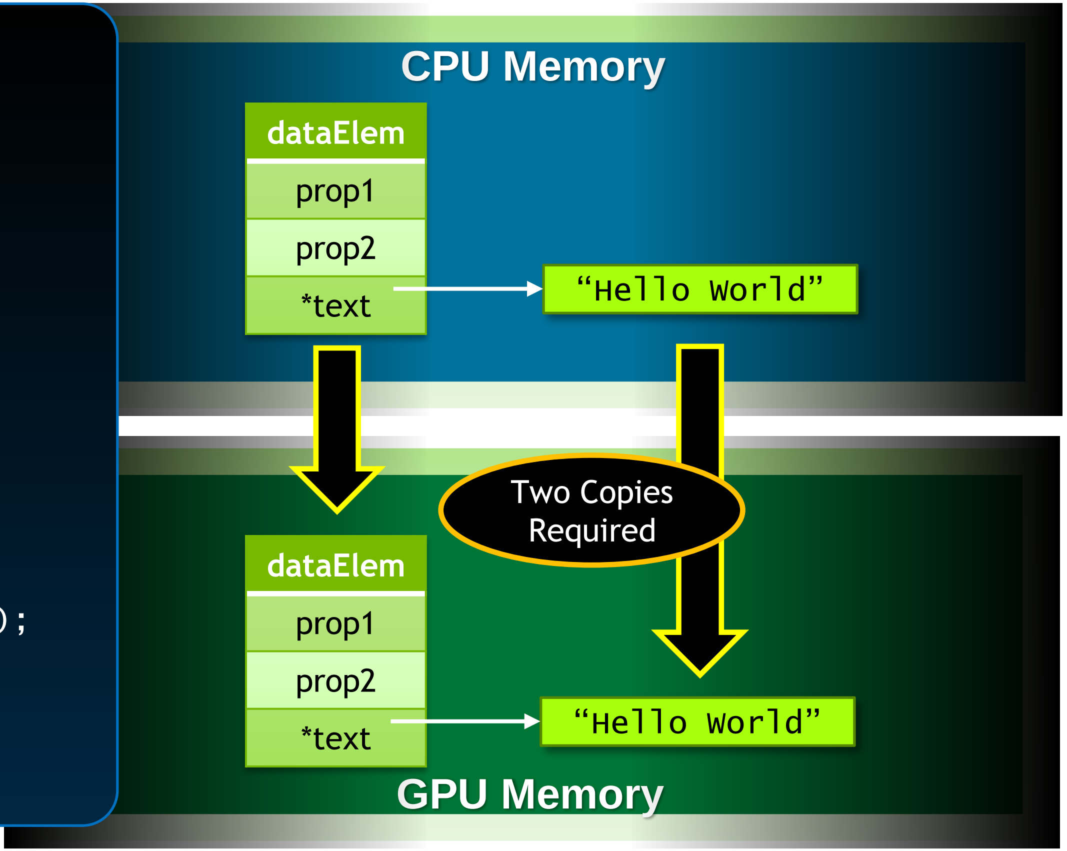


统一内存的基本概念

Unified Memory :

案例

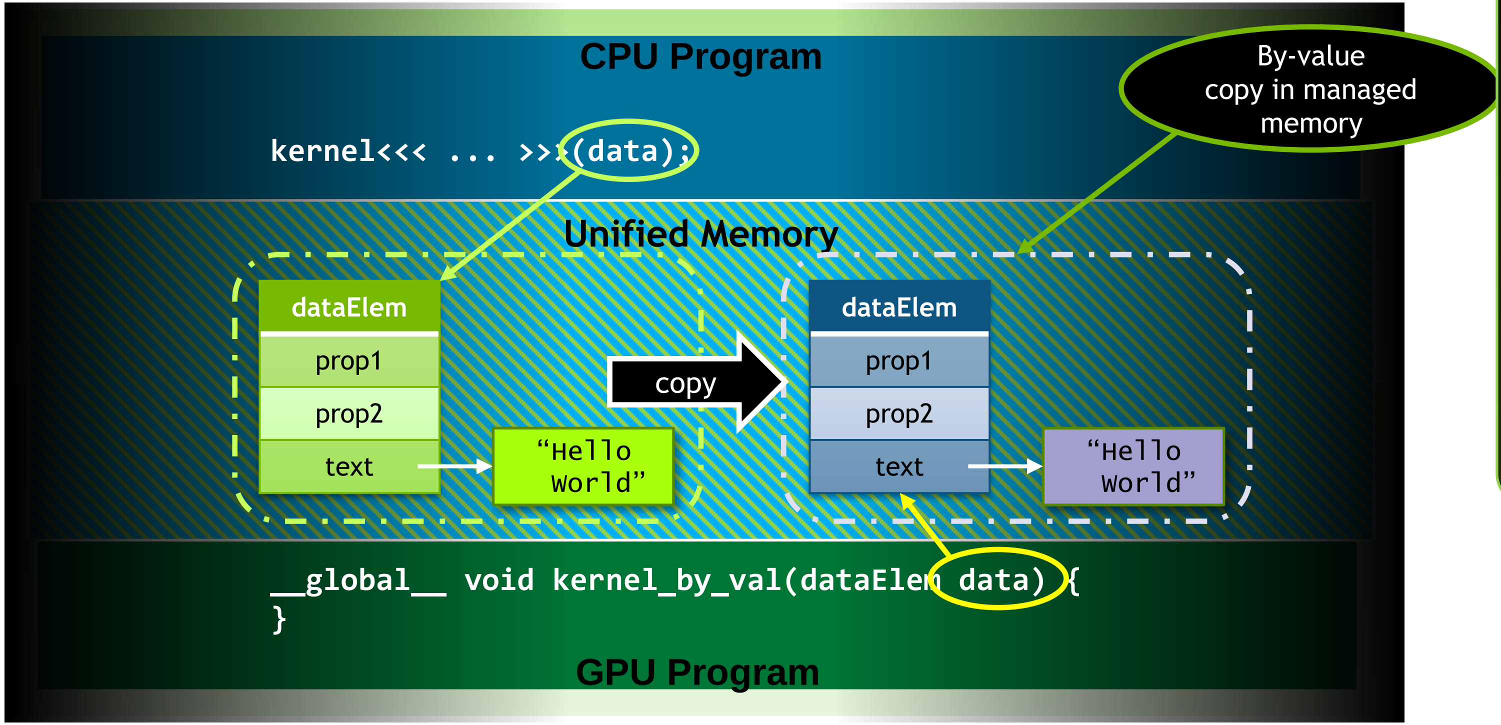
```
void launch(dataElem *elem) {
    dataElem *g_elem;
    char *g_text;

    int textlen = strlen(elem->text);

    // Allocate storage for struct and text
    cudaMalloc(&g_elem, sizeof(dataElem));
    cudaMalloc(&g_text, textlen);

    // Copy up each piece separately, including
    // new “text” pointer value
    cudaMemcpy(g_elem, elem, sizeof(dataElem));
    cudaMemcpy(g_text, elem->text, textlen);
    cudaMemcpy(&(g_elem->text), &g_text,
                                 sizeof(g_text));

    // Finally we can launch our kernel, but
    // CPU & GPU use different copies of “elem”
    kernel<<< ... >>>(g_elem);
}
```



统一内存的基本概念

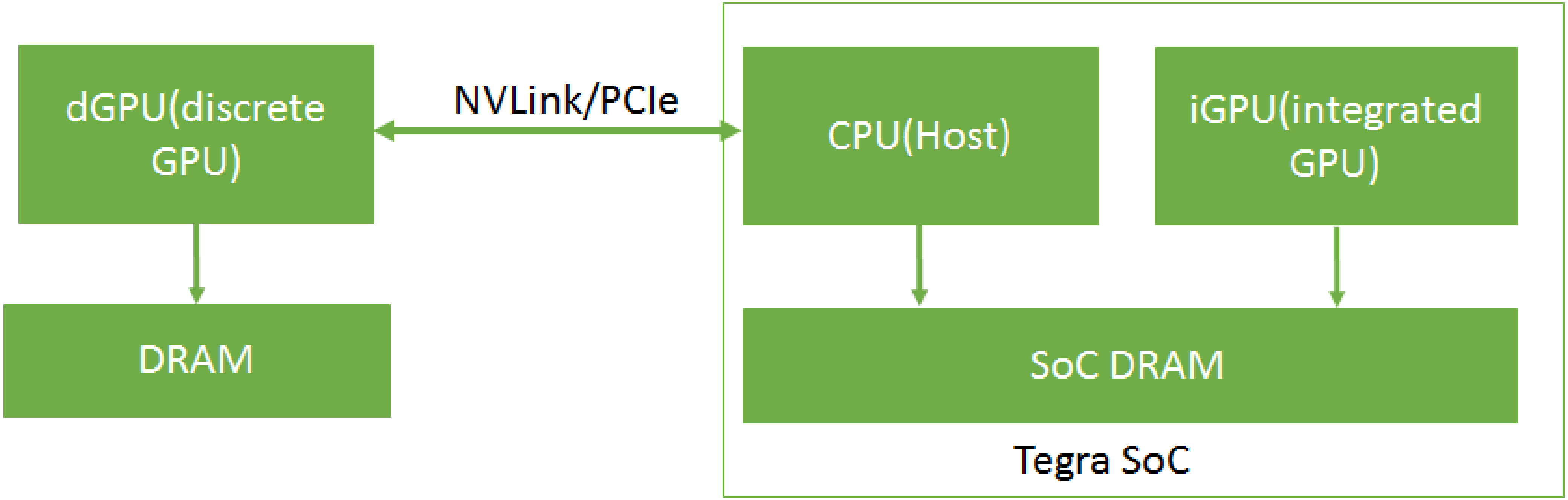
Unified Memory :

案例



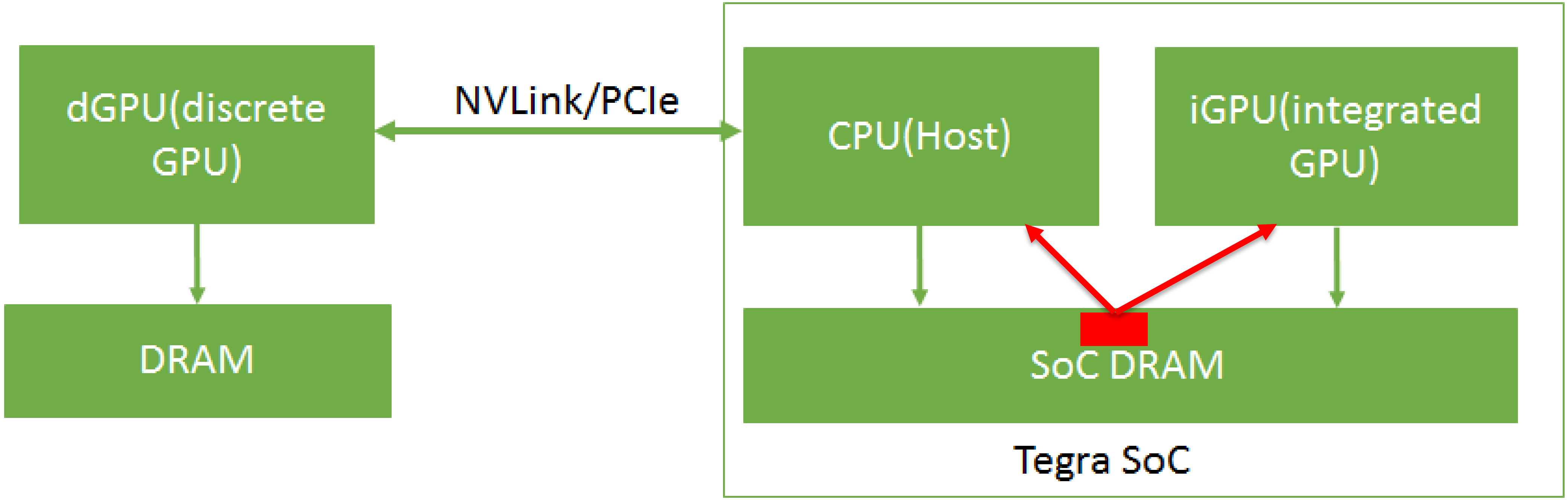

```
// Deriving from "Managed" allows
pass-by-reference
class String : public Managed {
    int length;
    char *data;

    // Unified memory copy
constructor allows pass-by-value
    String (const String &s) {
        length = s.length;
        cudaMallocManaged(&data,
length);
        memcpy(data, s.data,
length);
    }
};
```

基于ARM平台的JETSON NANO的存储单元特点

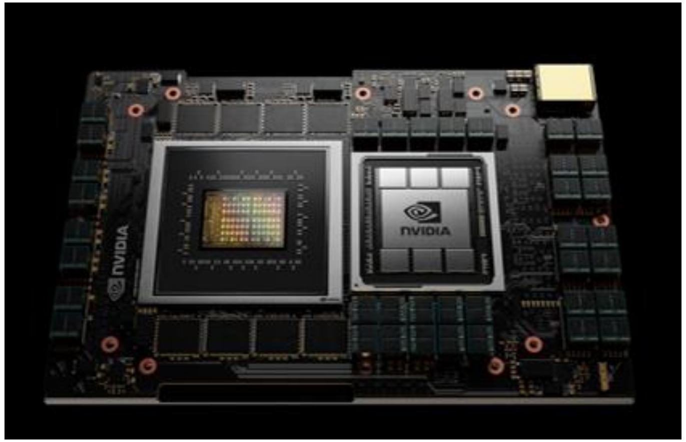


基于ARM平台的JETSON NANO的存储单元特点

Because device memory, host memory, and unified memory are allocated on the same physical SoC DRAM, duplicate memory allocations and data transfers can be avoided



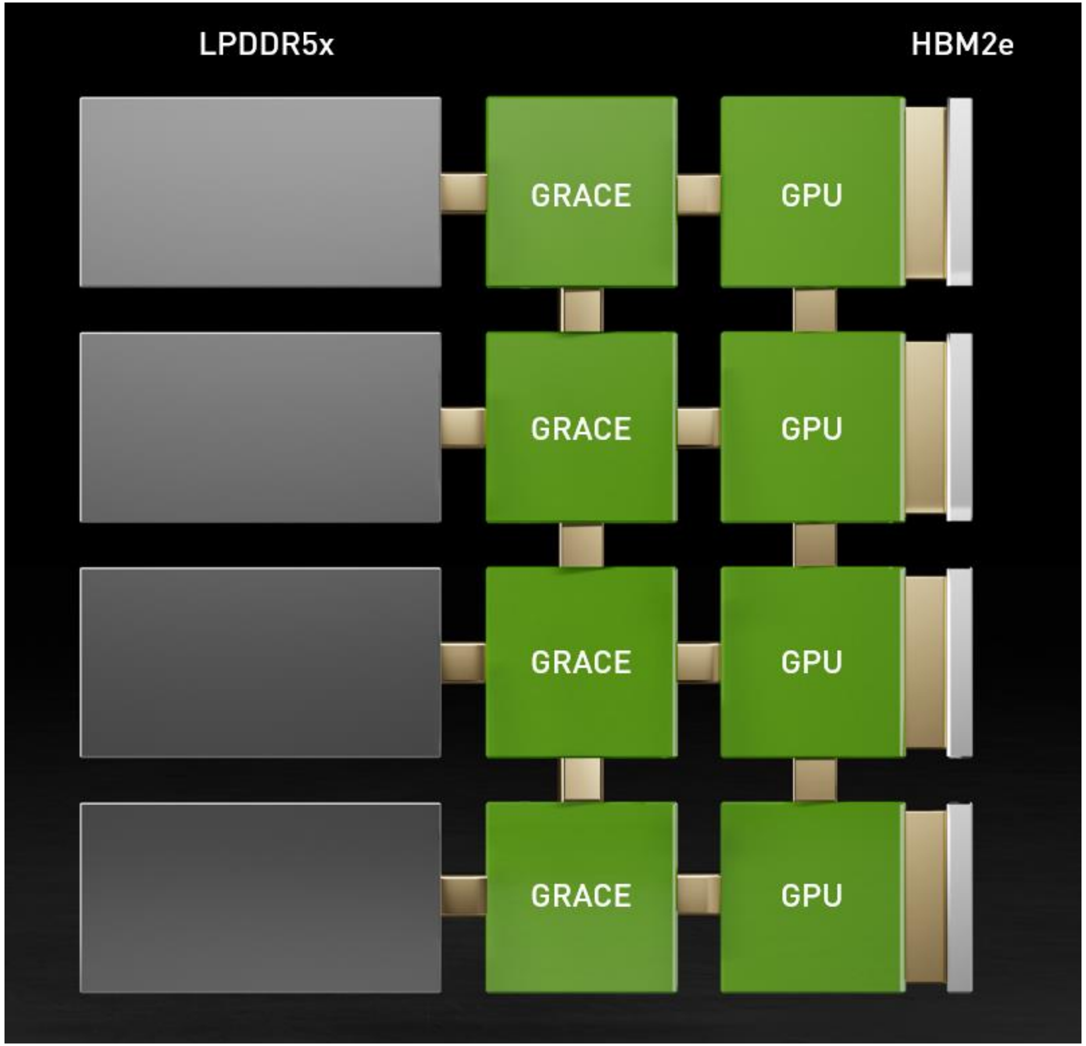
将来！



A NEW COMPUTING ARCHITECTURE FOR AI AND DATA SCIENCE

30X Increase System Memory to GPU

GPU	8,000	GB/sec	
CPU	500	GB/sec	
NVLINK	500	GB/sec	
Mem-to-GPU	2,000	GB/sec	30X



Overview: A CUDA HMM Program

Before: CPU-only

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);

  use_data(data);
  free(data);
}
```

After: CUDA with UVM + HMM

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort<<<…>>>(data, N, 1, compare);
  cudaDeviceSynchronize();
  use_data(data);
  free(data);
}
```

更多资源:

https://developer.nvidia-china.com

https://www.nvidia.cn/developer/community-training/


NVIDIA