异构加速计算平台常见的组合为CPU+GPU，其中CPU负责负责逻辑运算（串行部分），GPU负责密集计算（并行部分）。GPU凭借其超多核心（轻量级线程）的架构设计在单纯的浮点运算中拔得头筹，如今在深度学习领域应用最为广泛（矩阵计算）。

目前主流GPU生产商有Nvidia和AMD，其中Nvidia占主要市场份额，而AMD凭借其良好的性价比和不断进步的技术得到越来越多的关注。这里主要使用Nvidia的GPU。

对于用户来说，最关注如何通过编程充分发挥GPU算力。对此，Nvidia提供了CUDA（通用并行计算平台和编程模型）来帮助用户快速开发应用程序。为此开发了一整套的CUDA Toolkit，包含对应编译器，性能分析，调试器，运行库，常用数学库和函数库等。

# Hello World

CUDA支持的编程语言包括C/C++，Python和Fortran，本文主要使用C/C++进行学习和说明。NVCC是对应的编译器，它本质是提供一个C/C++的语法扩展子集和独立的运行库支持（runtime.h）。一个最简单的Hello World程序如下：

```
#include <stdio.h>
int main() {
        printf("Hello World!\n");
        return 0;
}
```

它完全是用C语言编写的，用nvcc可直接编译运行，可见nvcc对C/C++是完全兼容的。

```
> nvcc hello_world.cu
> ./a.out
Hello World!
```

但以上的程序仅用了CPU，并未使用GPU，那么我们写一个最简单的使用GPU的函数

```
#include <stdio.h>
// 核函数，用__global修饰
// 将被device调用，即在GPU上执行并行执行的函数
__global__ void mykernel() {
        printf("mykernel\n");
}

int main() {
        // host调用device代码，2个grid，1个block（进程的逻辑划分单位）
        mykernel<<<2,1>>>();
        // 让device上的printf能打印出来，它与标准库实现不同
        cudaDeviceSynchronize();
        printf("Hello World!\n");
        return 0;
}
```

```
> nvcc hello_world.cu
> ./a.out
mykernel
mykernel
Hello World!
```

```
# 在slurm调度系统的超算上的提交脚本，网格12区的提交脚本
srun -p gpu -N 1 -n 1 --gres=gpu:1 a.out
# 用nvprof时一定在可执行文件前加 ./
srun -p gpu -N 1 -n 1 --gres=gpu:1 nvprof ./a.out
```

nvprof是一个极好针对CUDA的性能探测分析工具。nvidia profilers是nvidia开发的一系列性能分析工具。

## 获取GPU设备的关键属性

```
#include <iostream>

int main() {
        int dev = 0;
        cudaDeviceProp devProp;
        cudaGetDeviceProperties(&devProp, dev);
        std::cout << "使用GPU device " << dev << ": " << devProp.name <<
std::endl;
        std::cout << "SM的数量：" << devProp.multiProcessorCount << std::endl;
        std::cout << "每个线程块的共享内存大小：" << devProp.sharedMemPerBlock /
1024.0 << " KB" << std::endl;
        std::cout << "每个线程块的最大线程数：" << devProp.maxThreadsPerBlock <<
std::endl;
        std::cout << "每个EM的最大线程数：" << devProp.maxThreadsPerMultiProcessor
<< std::endl;
        std::cout << "每个EM的最大线程束数：" << devProp.maxThreadsPerMultiProcessor
/ 32 << std::endl;
        return 0;
}
```

## 一维向量加法

常说程序=数据+算法，在CUDA编程中也不例外。

- 数据本质为内存的存储管理，在Host（CPU）用正常的`malloc`/ `free`等库函数可实现内存动态申请和释放，而在Device（GPU）提供了`cudaMalloc`/`cudaFree`等接口进行内存的申请和释放；
- 算法本质为解决问题的步骤集合，在并行编程中，主要考虑负载均衡算法设计，即如何给每个任务分配合理的计算负载。

## 手动内存管理

以下实例为一维向量加法的并行实现，每个线程负责一个元素的加法。

```
#include <stdio.h>

__global__ void add(float *x, float *y, float *z, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = index; i < n; i += stride)
        z[i] = x[i] + y[i];
}

int main() {
    int N = (1 << 20);
    int nBytes = N * sizeof(float);
    // Define and allocate memory in host
    float *x, *y, *z;
    x = (float*)malloc(nBytes);
    y = (float*)malloc(nBytes);
    z = (float*)malloc(nBytes);

    // Assign initial values
    for (int i = 0; i < N; i++) {
        x[i] = 10.0;
        y[i] = 20.0;
    }

    // Define and allocate memory in device
    float *d_x, *d_y, *d_z;
    cudaMalloc((void**)&d_x, nBytes);
    cudaMalloc((void**)&d_y, nBytes);
    cudaMalloc((void**)&d_z, nBytes);

    // Copy memory from host to device
    cudaMemcpy((void*)d_x, (void*)x, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy((void*)d_y, (void*)y, nBytes, cudaMemcpyHostToDevice);

    // Call gpu/device code
    dim3 blockSize(256);
    dim3 gridSize((N + blockSize.x - 1) / blockSize.x);

    add<<< gridSize, blockSize>>>(d_x, d_y, d_z, N);

    // Copy memory from device to host
    cudaMemcpy((void*)z, (void*)d_z, nBytes, cudaMemcpyDeviceToHost);

    // Validation of Result
    float maxError = 0.0;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, (float)(fabs(z[i] - 30.0)));
    printf("Result Validation: max Error = %.5f\n", maxError);

    // Free device and host memory
    cudaFree(d_x); cudaFree(d_y); cudaFree(d_z);
    free(x); free(y); free(z);
```

```
    return 0;
}
```

## 统一内存管理

从上面的手动管理程序可以明显感受到定义和申请Host和Device的数据并在二者间进行Memcpy传输带来的繁琐编程，特别遇到复杂庞大的数据结构，数据的定义，申请和传输的代码量会急剧上升。为此，CUDA 6.0推出统一内存管理（Unified Memory）的编程模型，将CPU和GPU的内存在逻辑上统一起来，即对编程人员透明，由系统去实现复杂的内存拷贝等操作，极大的简化的编程工作量。

使用方法和普通的 malloc和free方法基本一致，即`cudaMallocManged`替代malloc，`cudaFree`替代free，唯一注意的是在访问由GPU计算的数据时需要先用`cudaDeviceSynchronize`进行同步。

该编程模式主要优点为简化编程，易维护移植。**缺点在于性能比手动管理略低，追求高性能编程需考虑到这一点**（一维向量加法中，手动 vs. 自动 = 6.409s vs. 6.897s）。

优化统一内存管理

```
#include <stdio.h>

__global__ void add(float *x, float *y, float *z, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = index; i < n; i += stride)
        z[i] = x[i] + y[i];
}

int main() {
    int N = (1 << 20);
    int nBytes = N * sizeof(float);
    // Define and allocate memory in unified memory
    float *x, *y, *z;
    cudaMallocManaged((void**)&x, nBytes);
    cudaMallocManaged((void**)&y, nBytes);
    cudaMallocManaged((void**)&z, nBytes);

    // Assign initial values
    for (int i = 0; i < N; i++) {
        x[i] = 10.0;
        y[i] = 20.0;
    }

    // Call gpu/device code
    dim3 blockSize(256);
    dim3 gridSize((N + blockSize.x - 1) / blockSize.x);

    add<<< gridSize, blockSize>>>(x, y, z, N);
    // Synchronzie device memory
    cudaDeviceSynchronize();

    // Validation of Result
    float maxError = 0.0;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, (float)(fabs(z[i] - 30.0)));
    printf("Result Validation: max Error = %.5f\n", maxError);
```

```
    // Free unified memory
    cudaFree(x); cudaFree(y); cudaFree(z);

    return 0;
}
```

# 循环并行化算法

[CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops]

在并行编程中，热点常常集中在循环内部，因此循环的并行化设计成为关键。以两个向量加法为例，其串行代码如下：

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

让每个线程负责一个元素的计算，调用时必须让总线程数大于或等于元素总数n。以下模式也称一体化内核（*monolithic kernel*）。很明显的缺点为当数据量增长到一定程度后，线程数量不可能无限增长，同时每个线程仅算一个元素会带来反复创建销毁的开销，并且利用率不高带来性能损失。

```
void add(float *x, float *y, float *z, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n)
        z[tid] = x[tid] + y[tid];
}
```

因此引入网格跨步（grid-stride）分配法，其实本质与round-robin分配一样。线程号为`tid`的线程负责满足`taskId%N=tid`的任务，其中`taskId`为任务编号，`N`为任务总数。

```
__global__ void add(float *x, float *y, float *z, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;  // 等价于tid
    int stride = blockDim.x * gridDim.x;    // 等价于N

    /*
    // Round-Robin常见写法
    for (int i = 0; i < N; i++) {
        if (i % N == tid)
            z[i] = x[i] + y[i];
    }
    */

    // 减少循环次数的写法，等价于Round-Robin方法
    for (int i = index; i < n; i += stride)
        z[i] = x[i] + y[i];
}
```

网格跨步方法有四个优点：

- 提高线程复用性：减少线程的创建和销毁开销，如共享变量和私有变量初始化操作等；
- 可扩展性强：无论增加多少数据，每个线程都可均摊负载计算；

- 便于调试：其本质是串行代码和一体化内核的一种泛化，调用kernel时用`add<<<1,1>>>`就切换为串行代码，便于验证正确性；若用`add<<<m,n>>>，m*n>=N`，则等价于一体化内核的方法；
- 可读性好：和串行循环的模式基本一致。

__ldg未定义问题：gpu芯片计算能力须在3.5或以上，编译时加入`-arch=sm_35`即可。

cuda极简教程

A Even Easier Introduction to CUDA

gpu加速会导致精度误差

# HIP

CUDA移植到HIP

https://github.com/ROCm-Developer-Tools/HIP/blob/master/docs/markdown/CUBLAS_API_supported_by_HIP.md

修改模块publication

- 将html转为markdown格式
- 双次跳转渲染转为单击跳转

增加相册模块Album

- 按照地点和时间划分子版块
- 选择，裁剪，布局有意义的照片

修改模块Blog

- 将所有具体内容转到Blog的index中
- Blog支持中文书写