矩阵及并行计算理论基础

矩阵乘法基础

C = AB A∈R（K,M） B∈R（M，P）

（1）矩阵计算的内积视角：将A视作一个行向量矩阵，将B视作一个列向量矩阵。

C(I,j) = A(I,:)B(:j)

（2）行向量视角：将B视作一个行向量矩阵，将A视作系数矩阵。

C（i:） = ∑m->M A(I,:)B(m:)

矩阵乘法基础

- 列向量视角:

将A视作一个一个列向量矩阵，将B视作稀疏矩阵。

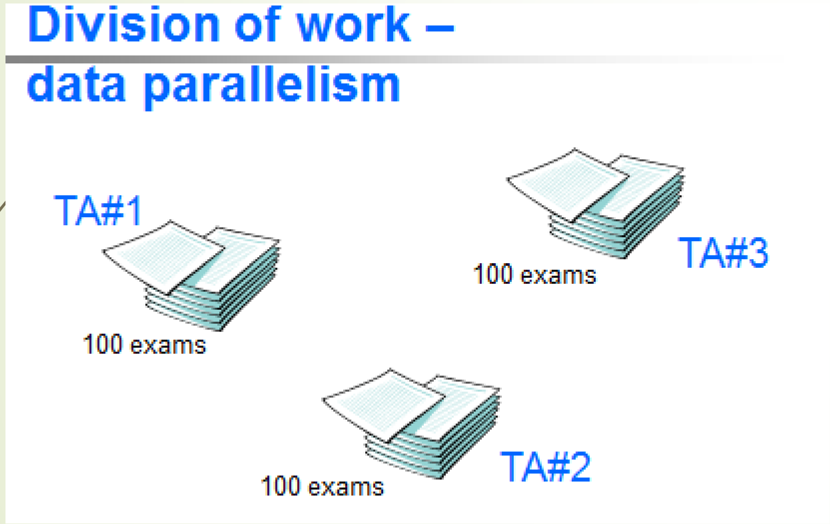
C（:j）=∑m->M B(mj)A(:m)

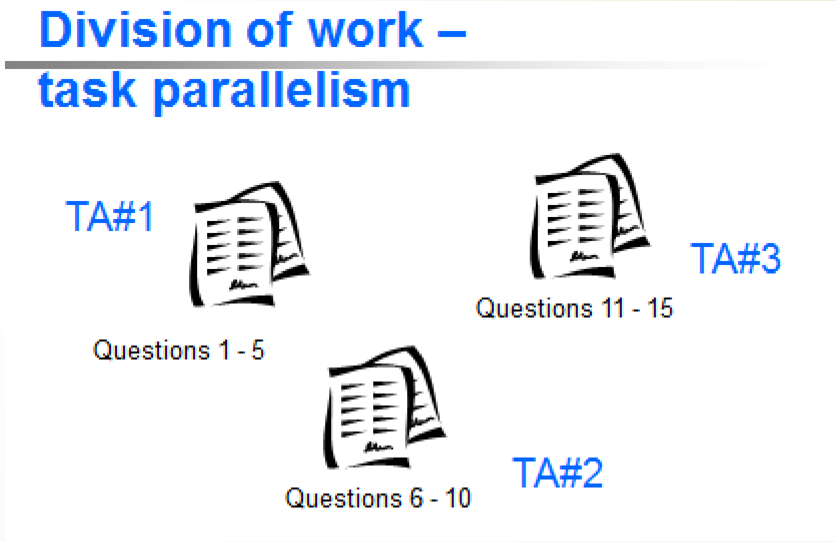
Eg :

A [[1 -2 ,2],[0 , 2 1]] B [[2 0], [-1,1],[0, 1]]

并行计算理论

- 数据并行和任务并行

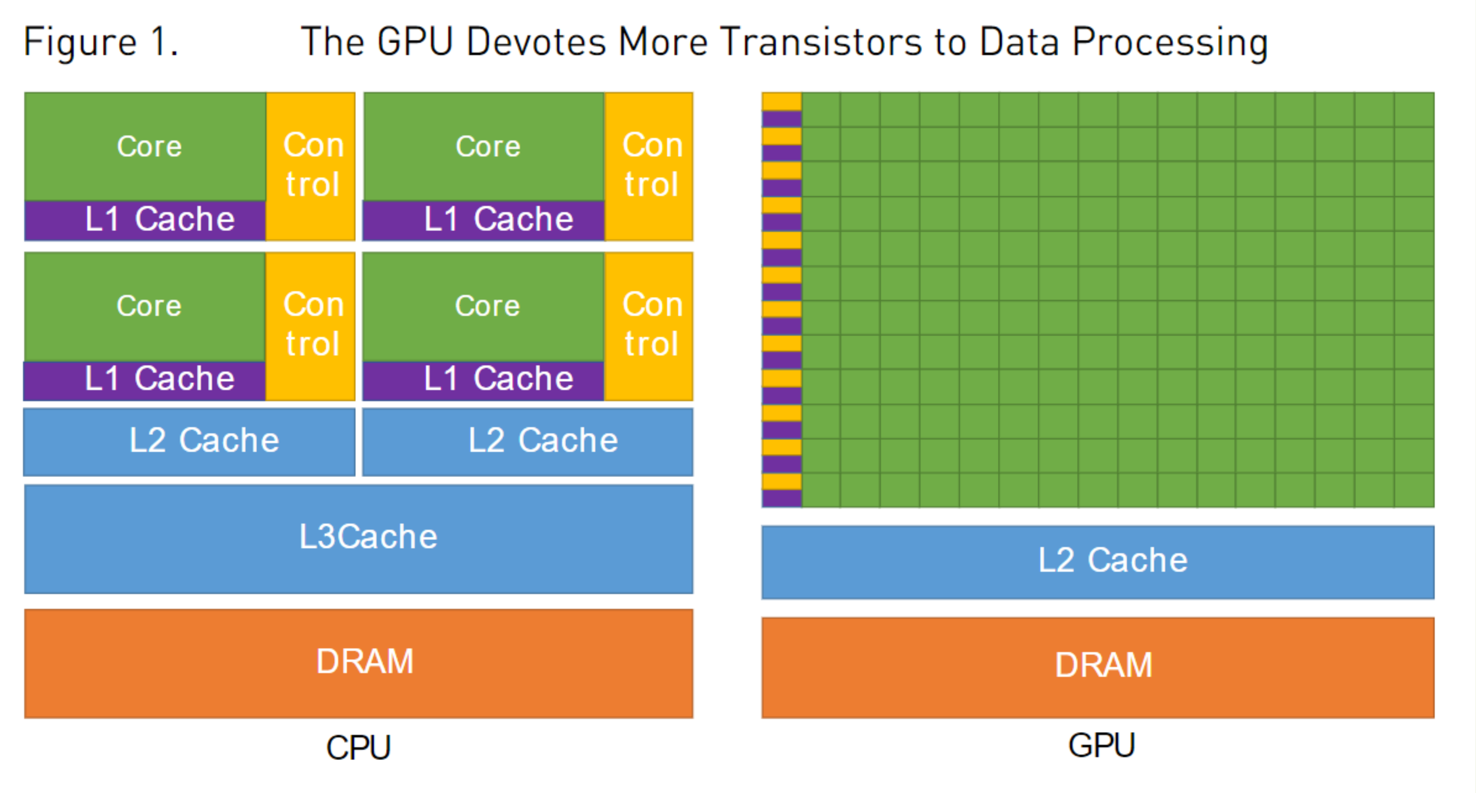




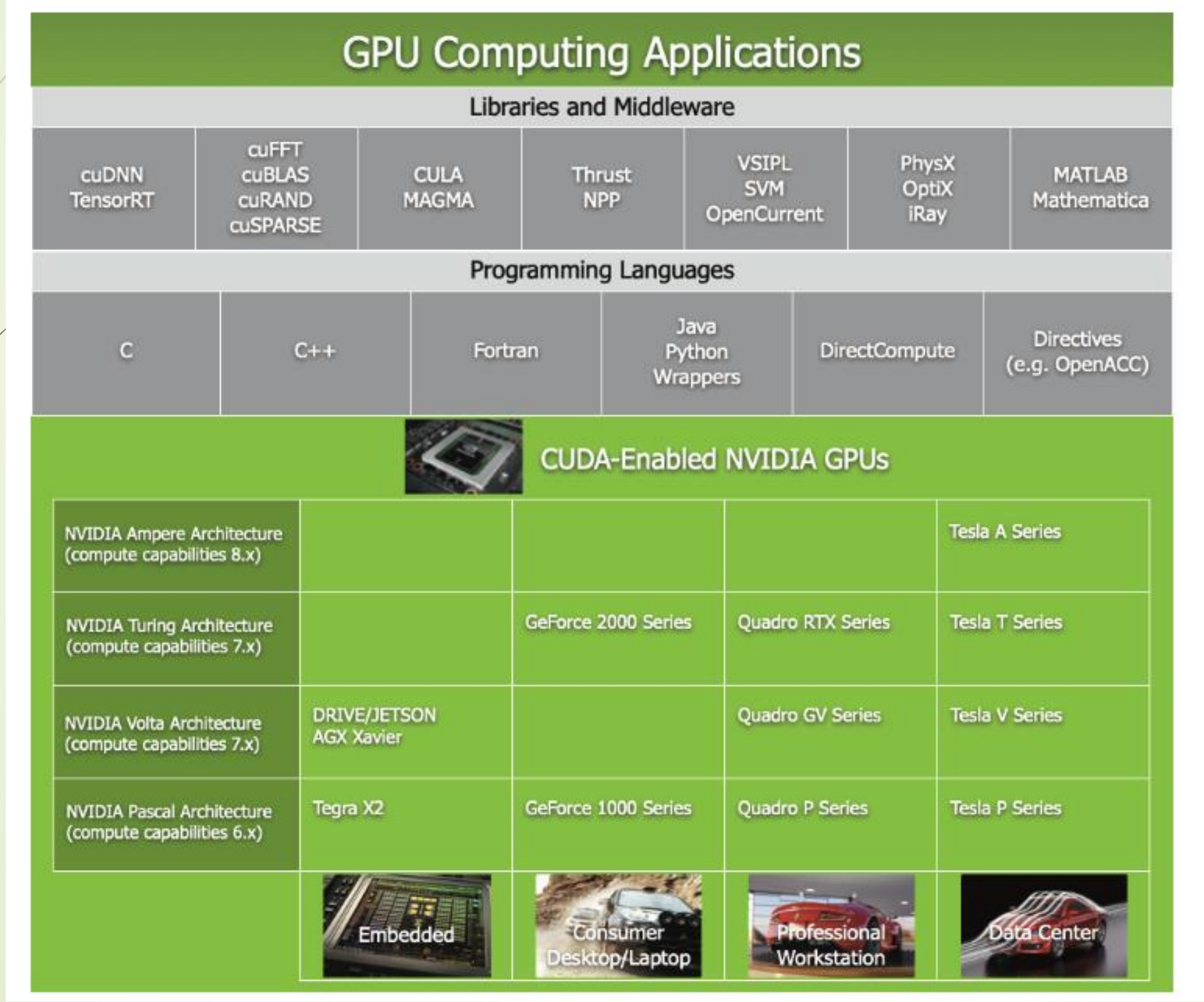
并行计算理论

Data Parallelisms	Task Parallelisms
1. Same task are performed on different subsets of same data.	1. Different task are performed on the same or different data.
2. Synchronous computation is performed.	2. Asynchronous computation is performed.
3. Amount of parallelization is proportional to the input size.	3. Amount of parallelization is proportional to the number of independent tasks is performed.
4. It is designed for optimum load balance on multiprocessor system.	4. Here, load balancing depends upon on the e availability of the hardware and scheduling algorithms like static and dynamic scheduling.

CPU和GPU体系结构差异



CUDA应用生态



GPU Computing Applications
Libraries and Middleware
cuDNN
TensorRT
cuFFT
cuBLAS
cuRAND
cuSPARSE
CULA
MAGMA
Thrust
NPP
VSIPL
SVM
OpenCurrent
PhysX
OptiX
iRay
MATLAB
Mathematica
Programming Languages
C
C++
Fortran
Java
Python
Wrappers
DirectCompute
Directives
(e.g. OpenACC)
CUDA-Enabled NVIDIA GPUs
NVIDIA Ampere Architecture
(compute capabilities 8.x)
Tesla A Series
NVIDIA Turing Architecture
(compute capabilities 7.x)
GeForce 2000 Series
Quadro RTX Series
Tesla T Series
NVIDIA Volta Architecture
(compute capabilities 7.x)
DRIVE/JETSON
AGX Xavier
Quadro GV Series
Tesla V Series
NVIDIA Pascal Architecture
(compute capabilities 6.x)
Tegra X2
GeForce 1000 Series
Quadro P Series
Tesla P Series
Embedded
Consumer
Desktop/Laptop
Professional
Workstation
Data Center

GPU计算扩展性

Figure 3. Automatic Scalability

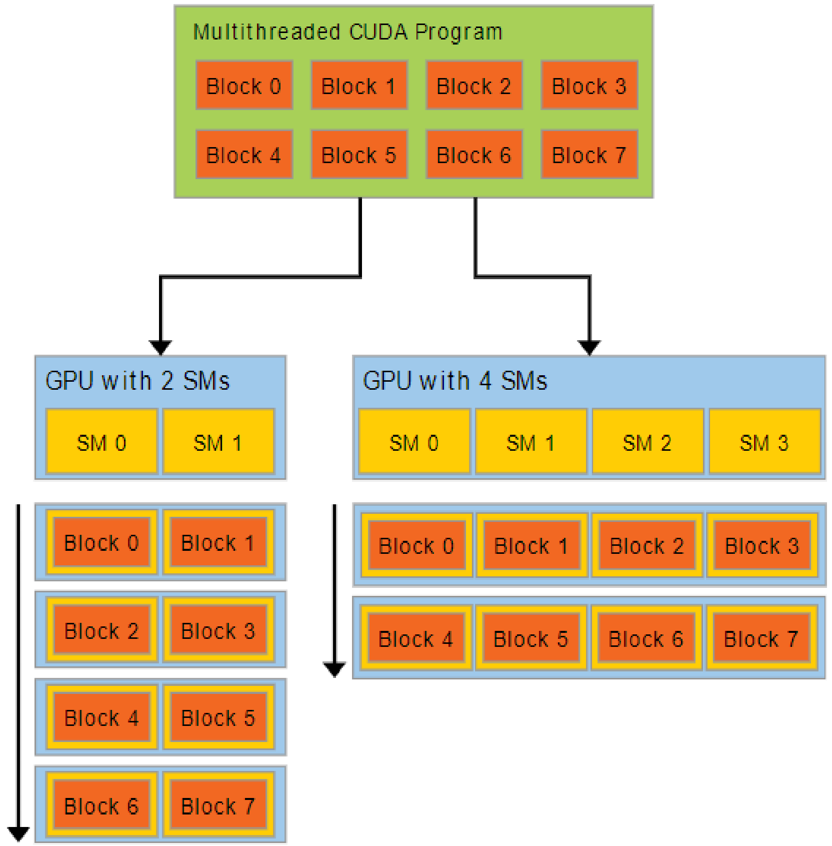
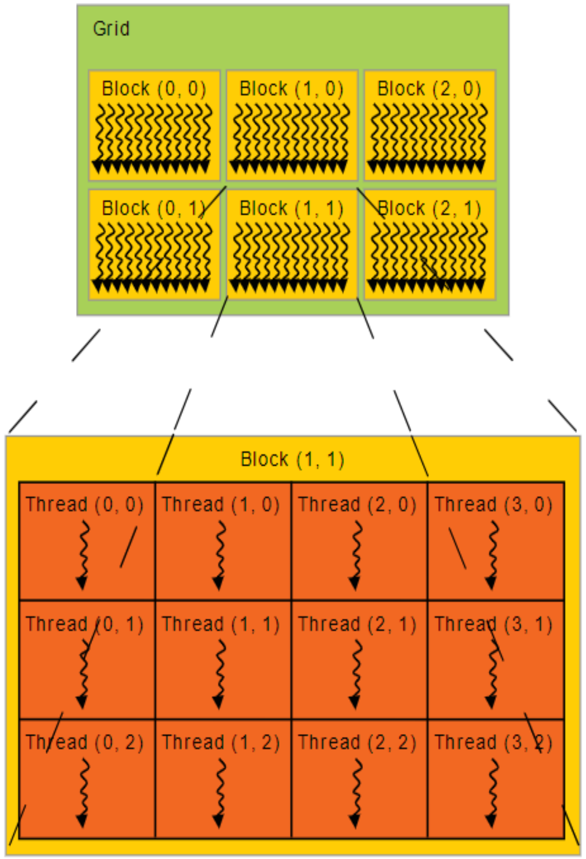
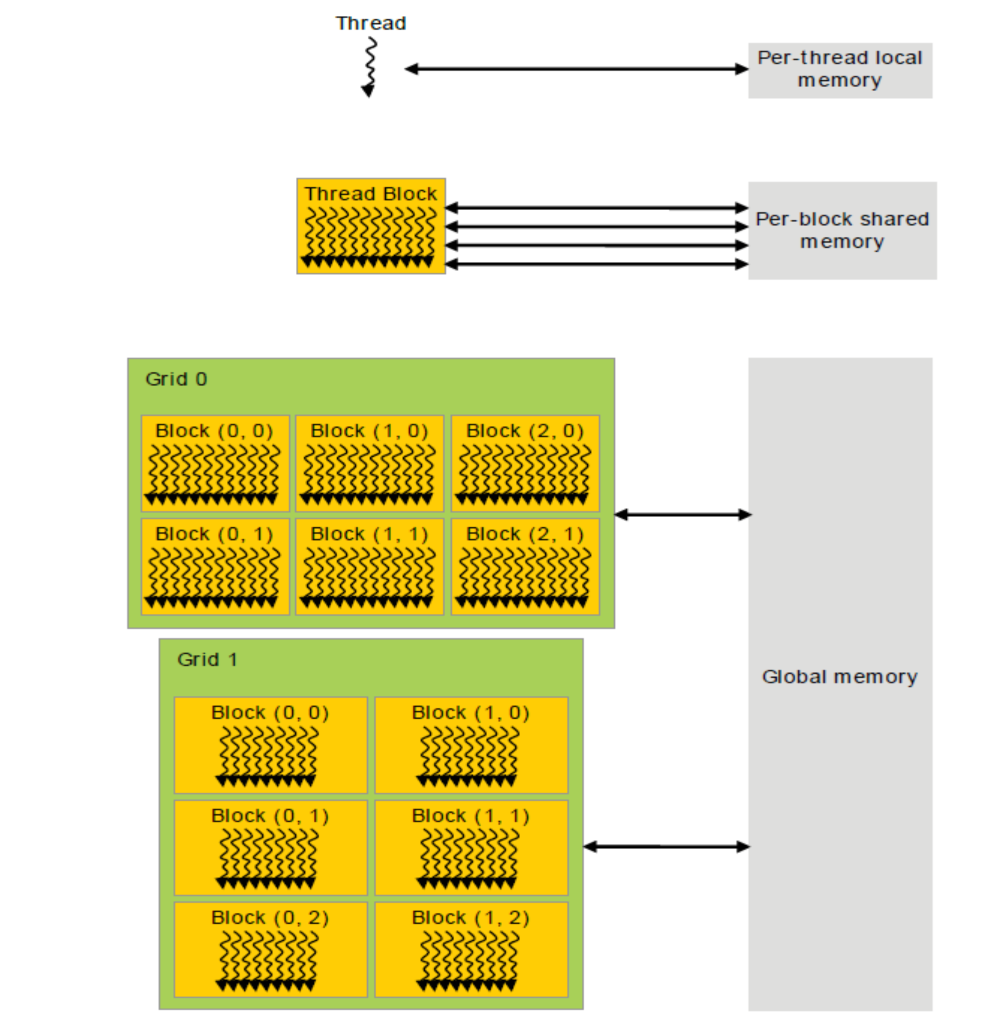


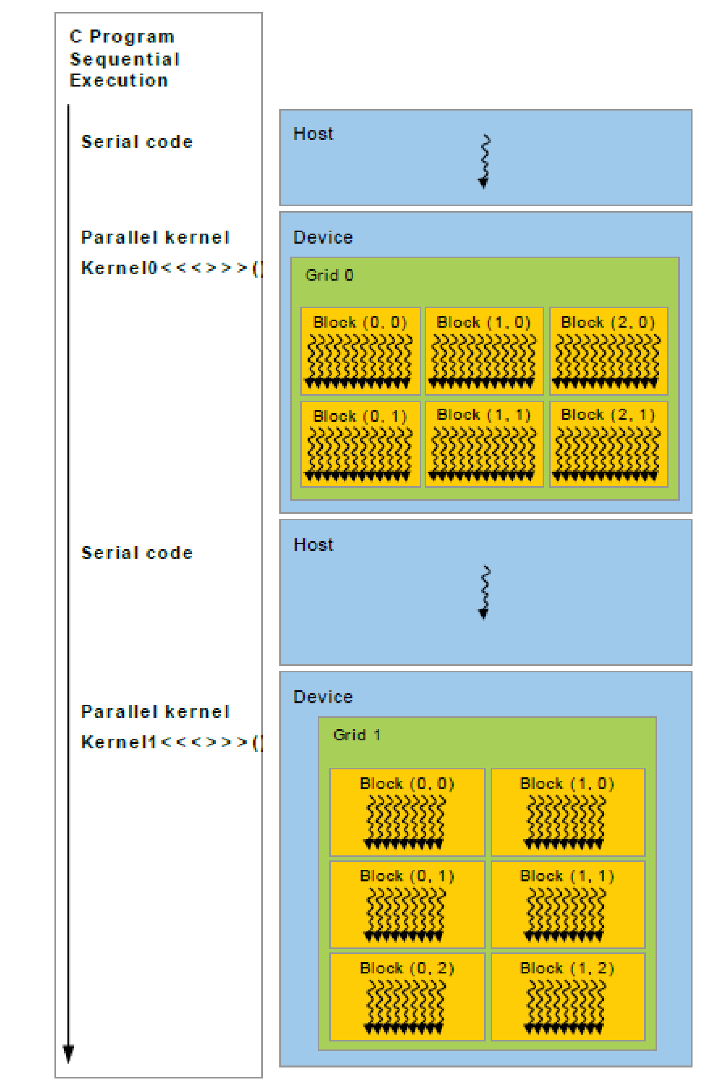
Figure 4. Grid of Thread Blocks



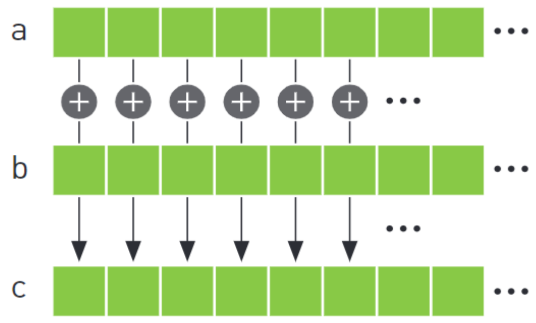
CUDA存储层次及编程模型

Figure 5. Memory Hierarchy





CUDA计算基础




```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    // handle the data at this index
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

```
void add( int *a, int *b, int *c ) {
    int tid = 0;    // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;    // we have one CPU, so we increment by one
    }
}
```

```
__device__ int thread_sum(int *input, int n)
{
     int sum = 0;

    for(int i = blockIdx.x * blockDim.x + threadIdx.x;
        i < n / 4;
        i += blockDim.x * gridDim.x)
    {
        int4 in = ((int4*)input)[i];
        sum += in.x + in.y + in.z + in.w;
    }
    return sum;
}

__global__ void sum_kernel_block(int *sum, int *input, int n)
{
    int my_sum = thread_sum(input, n);

    extern __shared__ int temp[];
    auto g = this_thread_block();
    int block_sum = reduce_sum(g, temp, my_sum);
```

CUDA计算基础

```
// 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);

    // 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);
```

显存分配

内存->显存数据拷贝

内核Kernel调用计算流程

CUDA多设备系统

设备遍历

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device);
    printf("Device %d has compute capability %d.%d.\n",
           device, deviceProp.major, deviceProp.minor);
}
```

设备选择

```
size_t size = 1024 * sizeof(float);
cudaSetDevice(0);            // Set device 0 as current
float* p0;
cudaMalloc(&p0, size);       // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0
cudaSetDevice(1);            // Set device 1 as current
float* p1;
cudaMalloc(&p1, size);       // Allocate memory on device 1
MyKernel<<<1000, 128>>>(p1); // Launch kernel on device 1
```

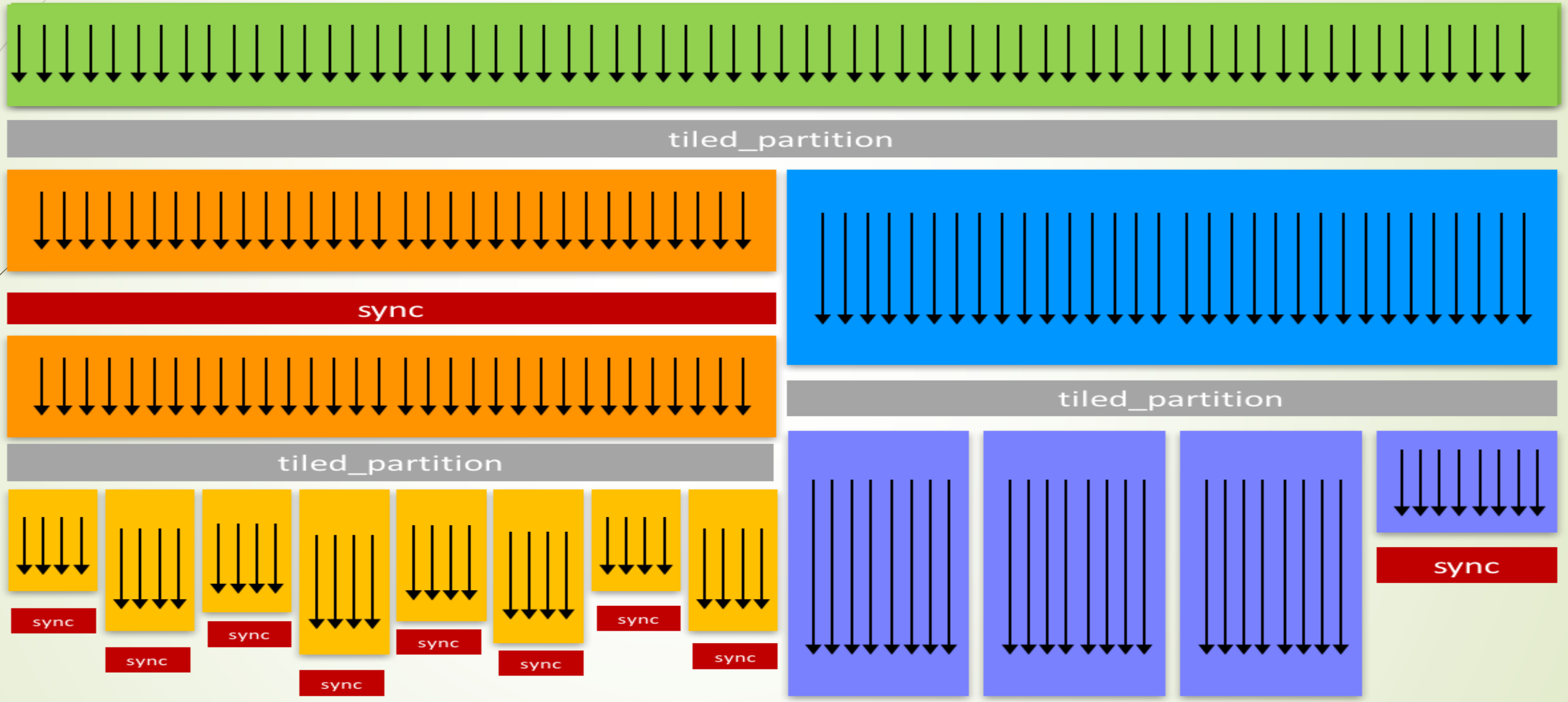
CUDA事件和CUDA流

```
cudaSetDevice(0);               // Set device 0 as current
cudaStream_t s0;
cudaStreamCreate(&s0);          // Create stream s0 on device 0
MyKernel<<<100, 64, 0, s0>>>(); // Launch kernel on device 0 in s0
cudaSetDevice(1);               // Set device 1 as current
cudaStream_t s1;
cudaStreamCreate(&s1);          // Create stream s1 on device 1
MyKernel<<<100, 64, 0, s1>>>(); // Launch kernel on device 1 in s1

// This kernel launch will fail:
MyKernel<<<100, 64, 0, s0>>>(); // Launch kernel on device 1 in s0
```

设备间互访

```
cudaSetDevice(0);                   // Set device 0 as current
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);              // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0);        // Launch kernel on device 0
cudaSetDevice(1);                   // Set device 1 as current
cudaDeviceEnablePeerAccess(0, 0);   // Enable peer-to-peer access
                                    // with device 0

// Launch kernel on device 1
// This kernel launch can access memory on device 0 at address p0
MyKernel<<<1000, 128>>>(p0);
```

Cooperative Groups



Thrust：CUDA编程体系的STL

- Vector相关操作

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

#include <iostream>

int main(void)
{
    // H has storage for 4 integers
    thrust::host_vector<int> H(4);

    // initialize individual elements
    H[0] = 14;
    H[1] = 20;
    H[2] = 38;
    H[3] = 46;

    // H.size() returns the size of vector H
    std::cout << "H has size " << H.size() << std::endl;

    // print contents of H
    for(int i = 0; i < H.size(); i++)
        std::cout << "H[" << i << "] = " << H[i] << std::endl;

    // resize H
    H.resize(2);

    std::cout << "H now has size " << H.size() << std::endl;

    // Copy host_vector H to device_vector D
    thrust::device_vector<int> D = H;

    // elements of D can be modified
    D[0] = 99;
    D[1] = 88;

    // print contents of D
    for(int i = 0; i < D.size(); i++)
        std::cout << "D[" << i << "] = " << D[i] << std::endl;

    // H and D are automatically deleted when the function returns
    return 0;
}
```

Thrust：CUDA编程体系的STL

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/sequence.h>

#include <iostream>

int main(void)
{
    // initialize all ten integers of a device_vector to 1
    thrust::device_vector<int> D(10, 1);

    // set the first seven elements of a vector to 9
    thrust::fill(D.begin(), D.begin() + 7, 9);

    // initialize a host_vector with the first five elements of D
    thrust::host_vector<int> H(D.begin(), D.begin() + 5);

    // set the elements of H to 0, 1, 2, 3, ...
    thrust::sequence(H.begin(), H.end());

    // copy all of H back to the beginning of D
    thrust::copy(H.begin(), H.end(), D.begin());

    // print D
    for(int i = 0; i < D.size(); i++)
        std::cout << "D[" << i << "] = " << D[i] << std::endl;

    return 0;
}
```

设备显存分配和初始化

一致的数据操作体验

数据交换

Thrust：CUDA编程体系的STL

```
size_t N = 10;

// raw pointer to device memory
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof(int));

// wrap raw pointer with a device_ptr
thrust::device_ptr<int> dev_ptr(raw_ptr);

// use device_ptr in thrust algorithms
thrust::fill(dev_ptr, dev_ptr + N, (int) 0);
```

Thrust和CUDA raw ptr交互

```
int main(void)
{
    // create an STL list with 4 values
    std::list<int> stl_list;

    stl_list.push_back(10);
    stl_list.push_back(20);
    stl_list.push_back(30);
    stl_list.push_back(40);

    // initialize a device_vector with the list
    thrust::device_vector<int> D(stl_list.begin(), stl_list.end());

    // copy a device_vector into an STL vector
    std::vector<int> stl_vector(D.size());
    thrust::copy(D.begin(), D.end(), stl_vector.begin());

    return 0;
}
```

和标准STL交互

Thrust：CUDA编程体系的STL

```
struct saxpy_functor
{
    const float a;

    saxpy_functor(float _a) : a(_a) {}

    __host__ __device__
        float operator()(const float& x, const float& y) const {
            return a * x + y;
        }
};

void saxpy_fast(float A, thrust::device_vector<float>& X,
 thrust::device_vector<float>& Y)
{
    // Y <- A * X + Y
    thrust::transform(X.begin(), X.end(), Y.begin(), Y.begin(), saxpy_functor(A));
}

void saxpy_slow(float A, thrust::device_vector<float>& X,
 thrust::device_vector<float>& Y)
{
    thrust::device_vector<float> temp(X.size());

    // temp <- A
    thrust::fill(temp.begin(), temp.end(), A);

    // temp <- A * X
    thrust::transform(X.begin(), X.end(), temp.begin(), temp.begin(),
 thrust::multiplies<float>());

    // Y <- A * X + Y
    thrust::transform(temp.begin(), temp.end(), Y.begin(), Y.begin(),
 thrust::plus<float>());
}
```

算法与数据变换支持

Thrust：CUDA编程体系的STL

```
Void sequential_scan(float *x, float *y, int Max_i){
      y[0] = x[0]
      for (int i = 1; I < max_i; i++){
            y[i] = y[i-1] + x[i];
      }
}
```

	[3 1 7 0 4 1 6 3]
Exclusive	[0 3 4 11 11 15 16 22]
Inclusive	[3 4 11 11 15 16 22 25]

```
#include <thrust/scan.h>
int data[6] = {1, 0, 2, 2, 1, 3};
thrust::inclusive_scan(data, data + 6, data); // in-place scan
// data is now {1, 1, 3, 5, 6, 9}
```

闭扫描算法

```
#include <thrust/scan.h>
int data[6] = {1, 0, 2, 2, 1, 3};
thrust::exclusive_scan(data, data + 6, data); // in-place scan
// data is now {0, 1, 1, 3, 5, 6}
```

开扫描算法

CuPy： NumPy-like API accelerated with CUDA。CUDA GPU 库在Nvidia GPU 上的实现Numpy相应功能的函数库，CUPY使用 CuBLAS、CUDNN、Curand、CuoSver、CuPaSeSE 、NCCL等CUDA库，以充分利用GPU架构。

特点：

- Multi CUDA Core并行加速。
- Numpy 的一个镜像，替换简单。支持 Numpy 的大多数数组运算，包括索引、广播、数组数学以及各种矩阵变换。
- 支持编写自定义 代码，实现加速。

CuPy

CuPy 安装 :

pip install cupy

CuPy导入 :

```
import numpy as np
import cupy as cp
```

CuPy使用注意事项

- 矩阵维度和整体尺寸足够大，计算密集。

  尺寸过小，数据拷贝和内核加载等初始化耗时引入额外开销成为主要因素。

- 尽量避免 CPU和GPU混合编程。

```
import cupy as cp
Import numpy as np

gpu_data = cp.ones((1024,1024,8,8))
cpu_data = cp.ones((1024,1024,8,8))

for item in range(2048):
    gpu = gpu_data + gpu_data
```

CuPy

切换设备

```
>>> x_on_gpu0 = cp.array([1, 2, 3, 4, 5])
>>> cp.cuda.Device(1).use()
>>> x_on_gpu1 = cp.array([1, 2, 3, 4, 5])
```

设备间数据迁移

```
>>> x_cpu = np.array([1, 2, 3])
>>> x_gpu = cp.asarray(x_cpu)  # move the data to the current device.
```

```
>>> x_gpu = cp.array([1, 2, 3])  # create an array in the current device
>>> x_cpu = cp.asnumpy(x_gpu)  # move the array to the host.
```

设备推断

```
>>> # Stable implementation of log(1 + exp(x))
>>> def softplus(x):
...     xp = cp.get_array_module(x)
...     return xp.maximum(0, x) + xp.log1p(xp.exp(-abs(x)))
```

常规计算

```
>>> x_gpu = cp.array([1, 2, 3])
>>> l2_gpu = cp.linalg.norm(x_gpu)
```

CUPY自定义内核

- f(x,y) =(x-y)^2

```
>>> squared_diff = cp.ElementwiseKernel(
...     'float32 x, float32 y',
...     'float32 z',
...     'z = (x - y) * (x - y)',
...     'squared_diff')
```

```
>>> x = cp.arange(10, dtype=np.float32).reshape(2, 5)
>>> y = cp.arange(5, dtype=np.float32)
>>> squared_diff(x, y)
array([[ 0.,  0.,  0.,  0.,  0.],
       [25., 25., 25., 25., 25.]], dtype=float32)
>>> squared_diff(x, 5)
array([[25., 16.,  9.,  4.,  1.],
       [ 0.,  1.,  4.,  9., 16.]], dtype=float32)
```

Numba

- Numba 是一款可以将 python 函数编译为机器代码的JIT编译器，经过 Numba 编译的python 代码（仅限数组运算），其运行速度可以接近 C 或 FORTRAN 语言。Numba是一个python的即时编译器，其使用Numpy的arrays，functions和loops。当调用Numba修饰函数时，它被编译为机器代码即时执行，并且全部或部分代码随后可以以本机机器代码速度运行！我们常见的Cython解释器是用c语言的方式来解释字节码的，而Numba则是使用LLVM编译技术来解释字节码的。
- 安装Numba的推荐方法是使用conda包管理

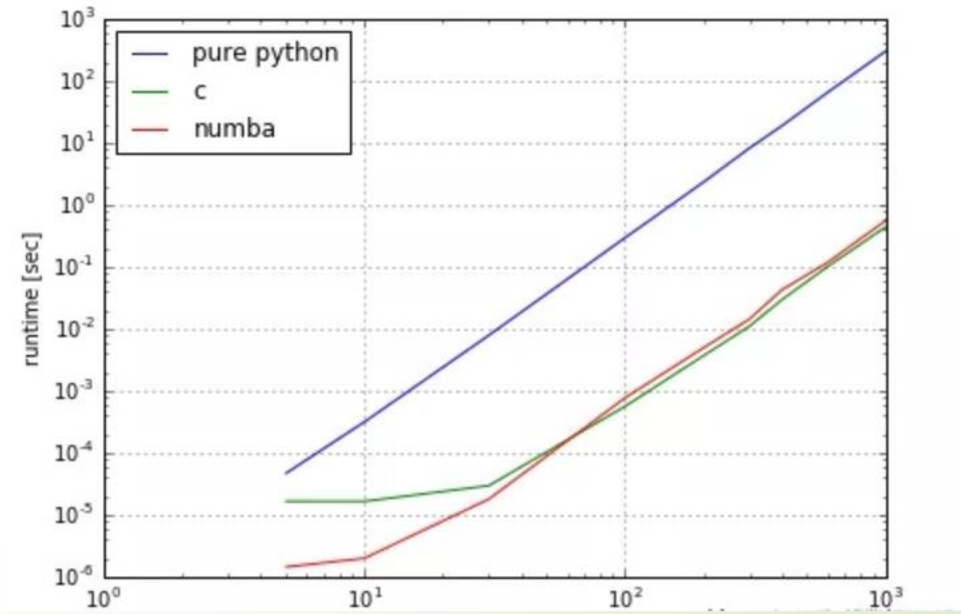
  conda install numba

- 使用pip库进行安装

  pip install numba

Numba使用范例

```python
from numba import jit
from numpy import arange
# jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when function is called.
@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return resulta = arange(9).reshape(3,3)
print(sum2d(a))
```



Numba CUDA支持

内核定义

```python
@cuda.jit
def increment_by_one(an_array):
    # Thread id in a 1D block
    tx = cuda.threadIdx.x
    # Block id in a 1D grid
    ty = cuda.blockIdx.x
    # Block width, i.e. number of threads per block
    bw = cuda.blockDim.x
    # Compute flattened index inside the array
    pos = tx + ty * bw
    if pos < an_array.size:  # Check array boundaries
        an_array[pos] += 1
```

内核调用

```python
threadsperblock = 32
blockspergrid = (an_array.size + (threadsperblock - 1)) // threadsperblock
increment_by_one[blockspergrid, threadsperblock](an_array)
```

Pycuda

- PyCUDA是Python语言用来访问Nvidia的CUDA并行计算API的安装包，其基础层是由C++语言进行编写的工具包。

安装：pip install pycuda==xxx

PyCUDA使用范例

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

print(dest-a*b)
```

内核定义

获取函数句柄

执行内核