CPU+GPU异构并行计算

林灯
lind@lsec.cc.ac.cn
中科院计算数学所

授人以渔

- Books:
  - CUDA C Programming Guide: http://docs.nvidia.com/cuda/
  - CUDA C Best Practices Guide: http://docs.nvidia.com/cuda/
  - Programming Massively Parallel Processors: A Hands-on Approach / 大规模并行处理器编程实战
- Q&A:
  - NVIDIA Developer Zone: https://devtalk.nvidia.com/
  - Stack Overflow: http://stackoverflow.com/

授人以渔

- Courses:
  - 胡文美老师在coursera上开设的"异构并行程序设计":
  - https://www.coursera.org/course/hetero

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

异构并行程序设计

This course introduces concepts, languages, techniques, and patterns for programming heterogeneous, massively parallel processors. Its contents and structure have been significantly revised based on the experience gained from its initial offering in 2012. It covers heterogeneous computing architectures, data-parallel programming models, techniques for memory bandwidth management, and parallel algorithm patterns.

概要

1. GPU
   1.1. 什么是GPU
   1.2. 为什么是GPU
   1.3. GPU发展
2. CUDA
   2.1. 什么是CUDA
   2.2. CPU与GPU协作
   2.3. 线程组织结构
   2.4. 线程调度
   2.5. 存储器
   2.6. 线程协作
   2.7. 例子: 稠密矩阵乘法

1. GPU

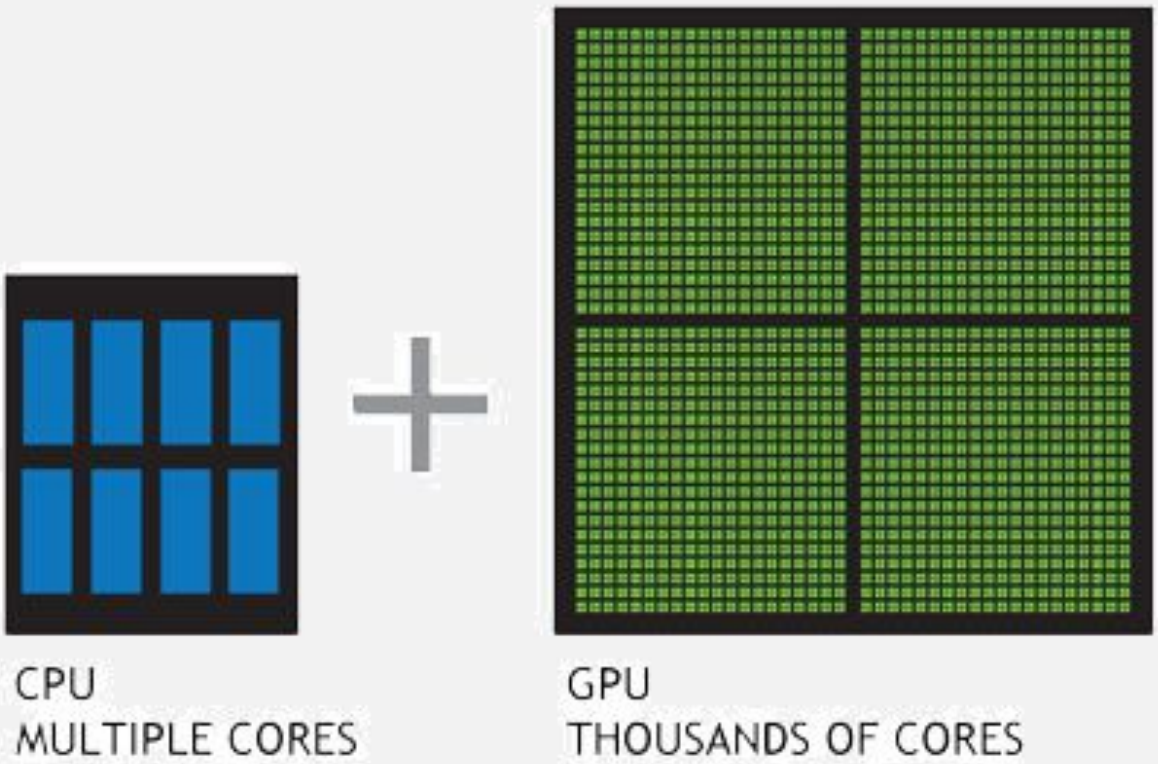
1.1. 什么是GPU

- Graphic Processing Unit / 图形处理单元

- 显卡的处理核心

- 从可视化专用计算转向通用并行计算
  - 06年前: 专用(可视化)
  - 06年起: 专用(可视化) + 通用(并行计算)

1.2. 为什么是GPU



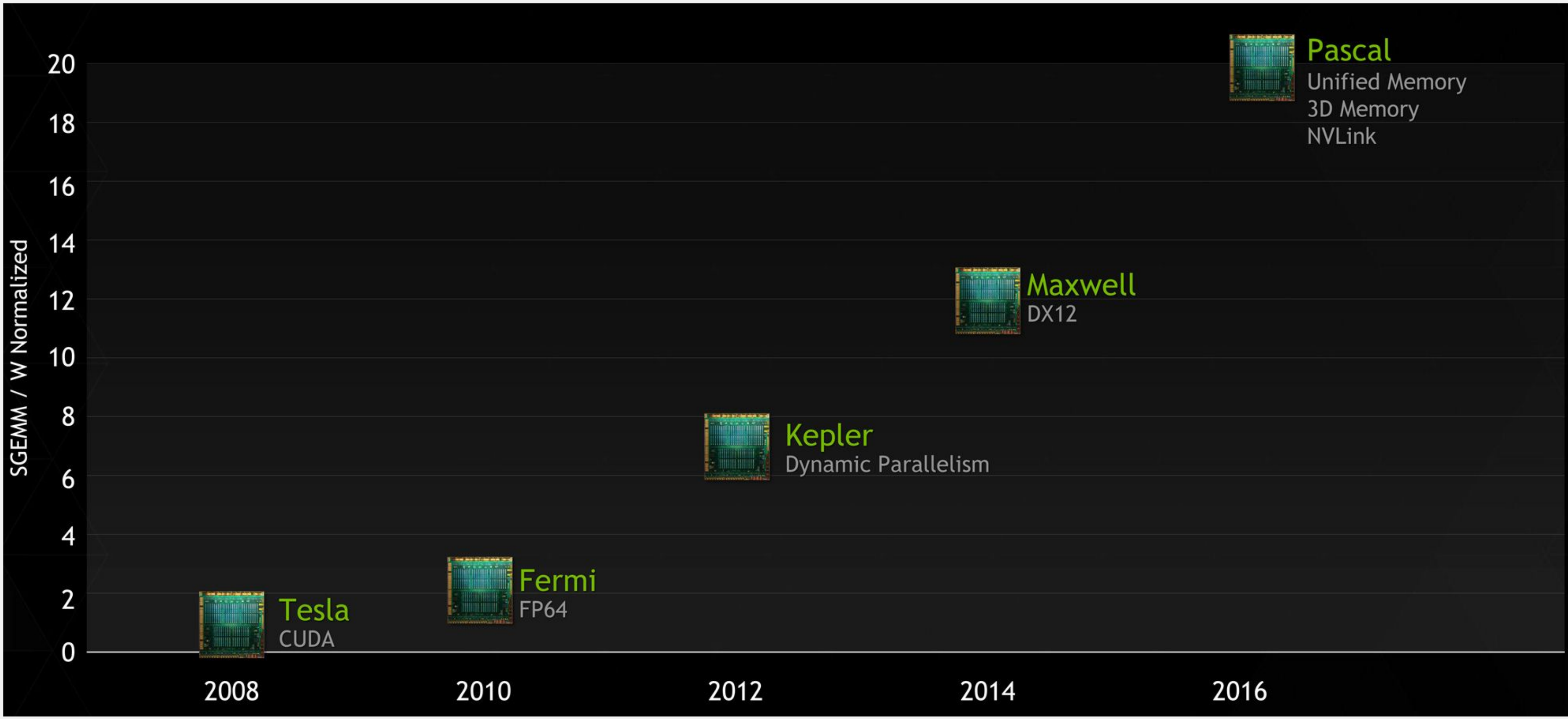
Seymour Cray : If you were plowing a field, which would you rather use: 2 strong oxen or 1024 chickens?

1.2. 为什么是GPU

- GPU与CPU有着完全不同的计算架构:
  - GPU把更多空间给了计算部件, 自然弱化了缓存、流控制等部件
  - CPU把更多空间给了缓存、流控制等部件, 自然弱化了计算部件

- 大量的计算部件使得GPU具有天然的强大的并行处理能力

1.3. GPU发展

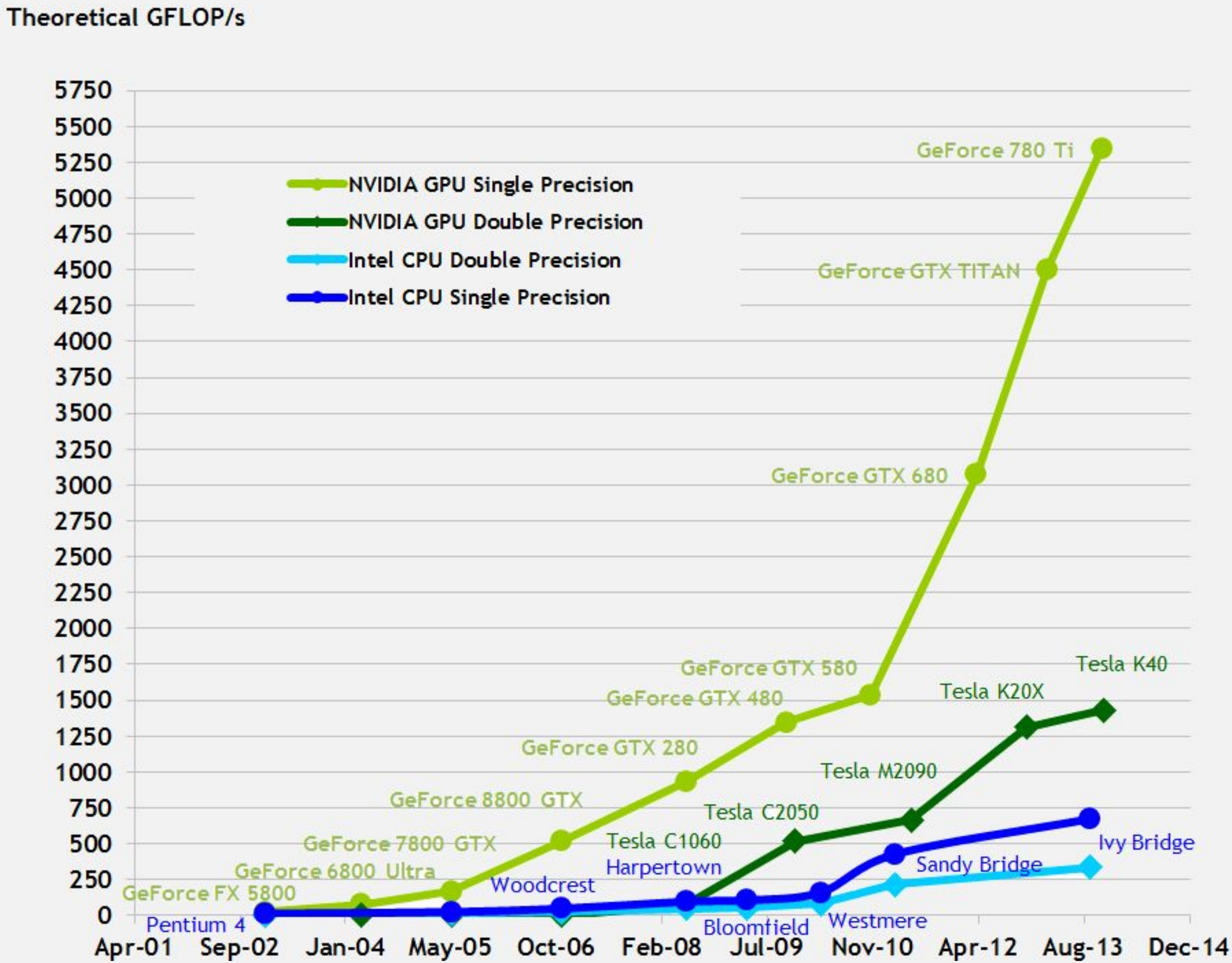

SGEMM / W Normalized
20
18
16
14
12
10
8
6
4
2
0
2008
2010
2012
2014
2016
Tesla
CUDA
Fermi
FP64
Kepler
Dynamic Parallelism
Maxwell
DX12
Pascal
Unified Memory
3D Memory
NVLink

1.3. GPU发展

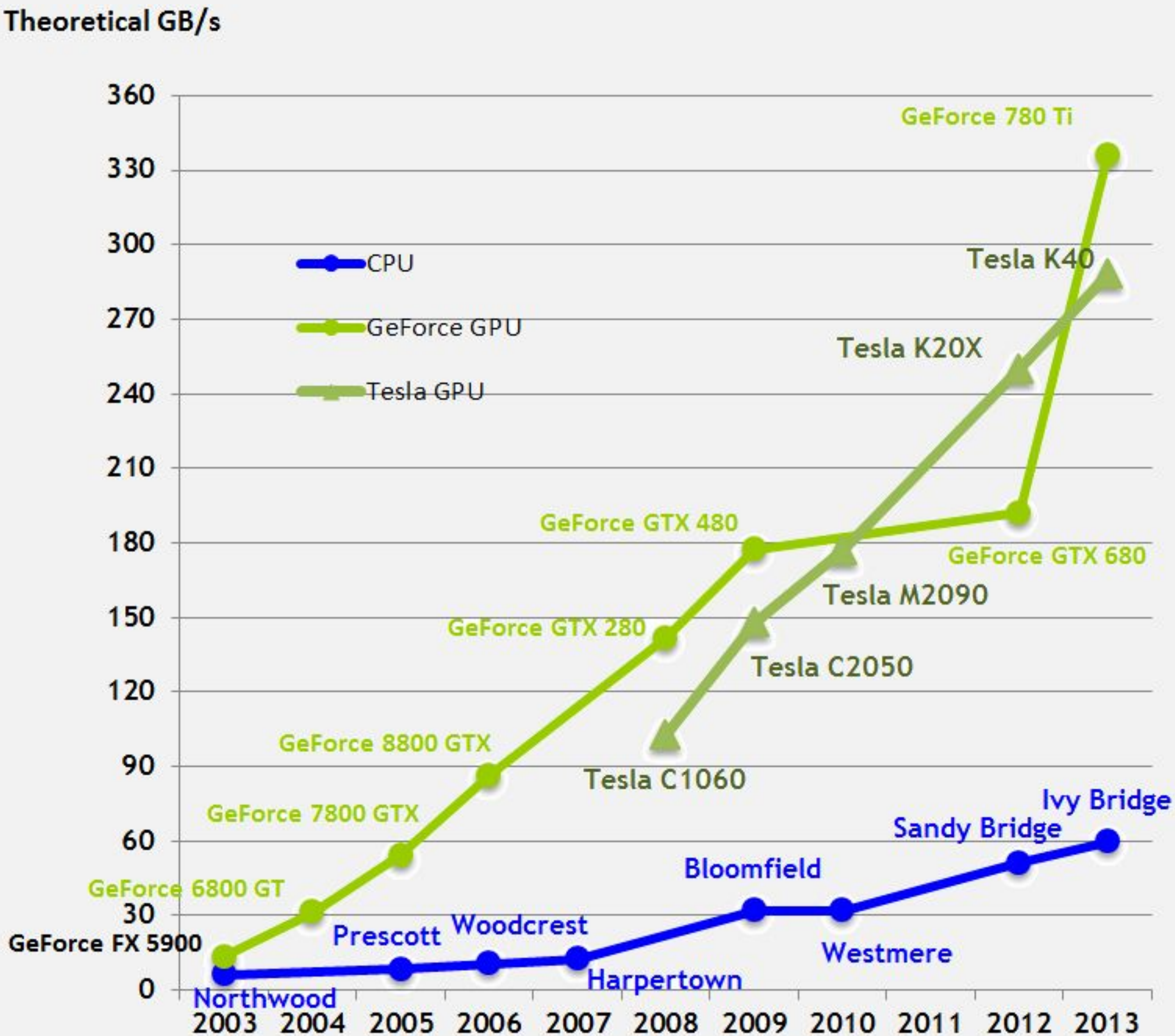
- 不同代的GPU计算架构差别较大
- 不同的GPU计算架构对应不同的计算能力(Compute Capability/C.C.)

表1. 不同的GPU计算架构对应不同的计算能力

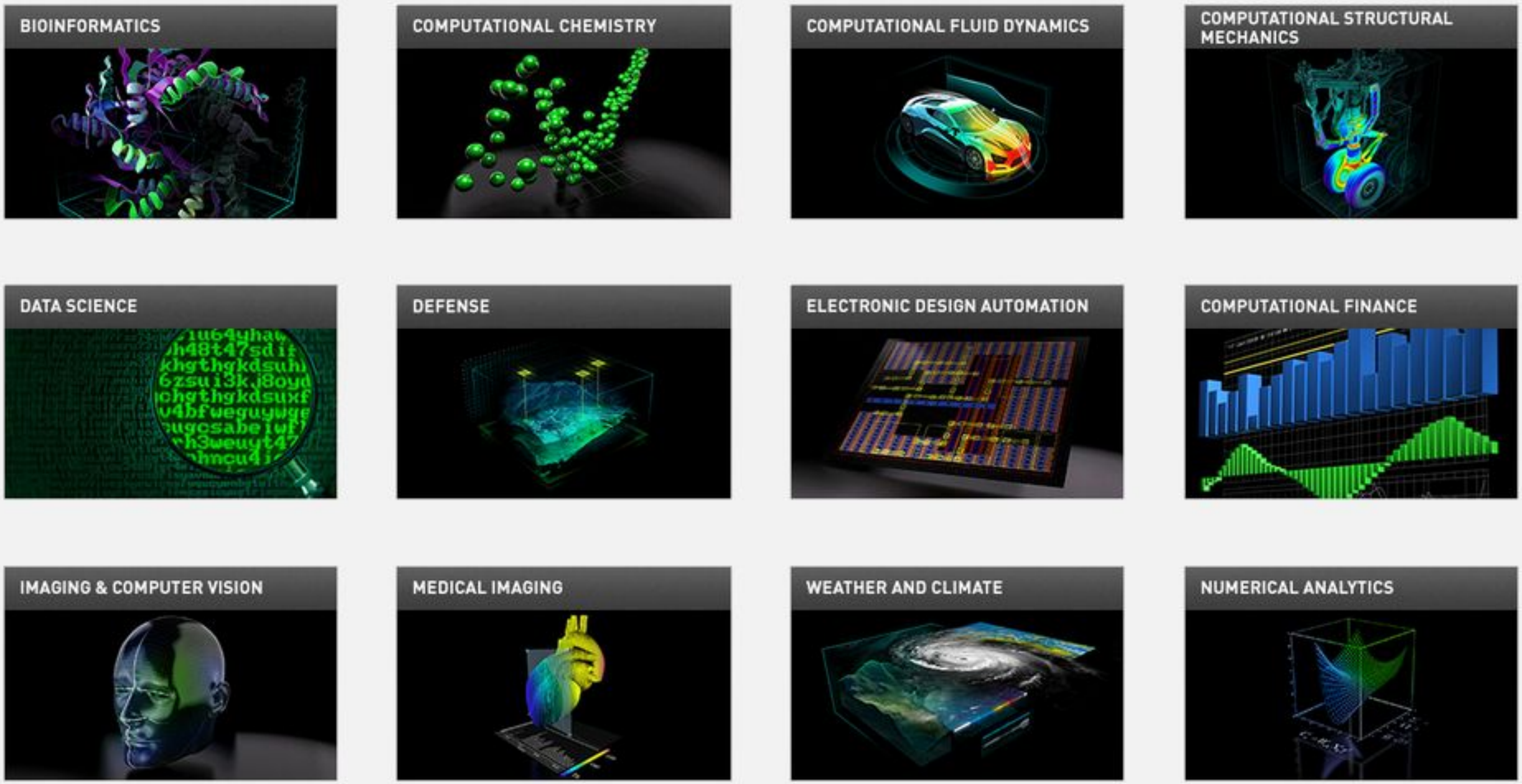
Arch	Tesla	Fermi	Kelper	Maxwell
C.C.	1.x	2.x	3.x	5.x



Theoretical GFLOP/s
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Double Precision
Intel CPU Single Precision
GeForce 780 Ti
GeForce GTX TITAN
GeForce GTX 680
GeForce GTX 580
GeForce GTX 480
GeForce GTX 280
GeForce 8800 GTX
GeForce 7800 GTX
GeForce 6800 Ultra
GeForce FX 5800
Tesla K40
Tesla K20X
Tesla M2090
Tesla C2050
Tesla C1060
Harpertown
Woodcrest
Pentium 4
Bloomfield
Westmere
Sandy Bridge
Ivy Bridge
5750
5500
5250
5000
4750
4500
4250
4000
3750
3500
3250
3000
2750
2500
2250
2000
1750
1500
1250
1000
750
500
250
0
Apr-01
Sep-02
Jan-04
May-05
Oct-06
Feb-08
Jul-09
Nov-10
Apr-12
Aug-13
Dec-14


Theoretical GB/s
360
330
300
270
240
210
180
150
120
90
60
30
0
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
CPU
GeForce GPU
Tesla GPU
GeForce 780 Ti
Tesla K40
Tesla K20X
GeForce GTX 480
GeForce GTX 680
Tesla M2090
GeForce GTX 280
Tesla C2050
GeForce 8800 GTX
Tesla C1060
GeForce 7800 GTX
Ivy Bridge
Sandy Bridge
Bloomfield
GeForce 6800 GT
Prescott
Woodcrest
GeForce FX 5900
Westmere
Harpertown
Northwood

1.3. GPU发展


BIOINFORMATICS
COMPUTATIONAL CHEMISTRY
COMPUTATIONAL FLUID DYNAMICS
COMPUTATIONAL STRUCTURAL MECHANICS
DATA SCIENCE
DEFENSE
ELECTRONIC DESIGN AUTOMATION
COMPUTATIONAL FINANCE
IMAGING & COMPUTER VISION
MEDICAL IMAGING
WEATHER AND CLIMATE
NUMERICAL ANALYTICS

1.3. GPU发展

表2. 一些相关领域中使用GPU加速的应用软件

Fields	Applications
Numerical Analytics	Mathematica Wolfram, MATLAB-Mathworks
Molecular Dynamics	AMBER, CHARMM, ESPResSo, LAMMPS
Materials Science	Abinit, CASTEP, CP2K, GAMESS, Gaussian
Quantum Chemistry	Altair-OptiStruct, ANSYS-Mechanical
Structural Mechanics	ANSYS-Fluent, Altair-AcuSolve, FluiDyna
Fluid Dynamics	GTC, GTS, PIConGPU
Geophysics	AWP-ODC, SPECFEM

1.3. GPU发展

表3. 世界最快超级计算机排行 (前6, 截止15年7月)

Rank	Name	Country	Processors
1	Tianhe 2	China	Xeon E5C2692 + Xeon Phi 31S1P
2	Titan	USA	Opteron 6274 + Tesla K20X
3	Sequoia	USA	PowerPC A2
4	K computer	Japan	SPARC64 VIIIfx
5	Mira	USA	PowerPC A2
6	Piz Daint	USA	Xeon E5C2670 + Tesla K20X

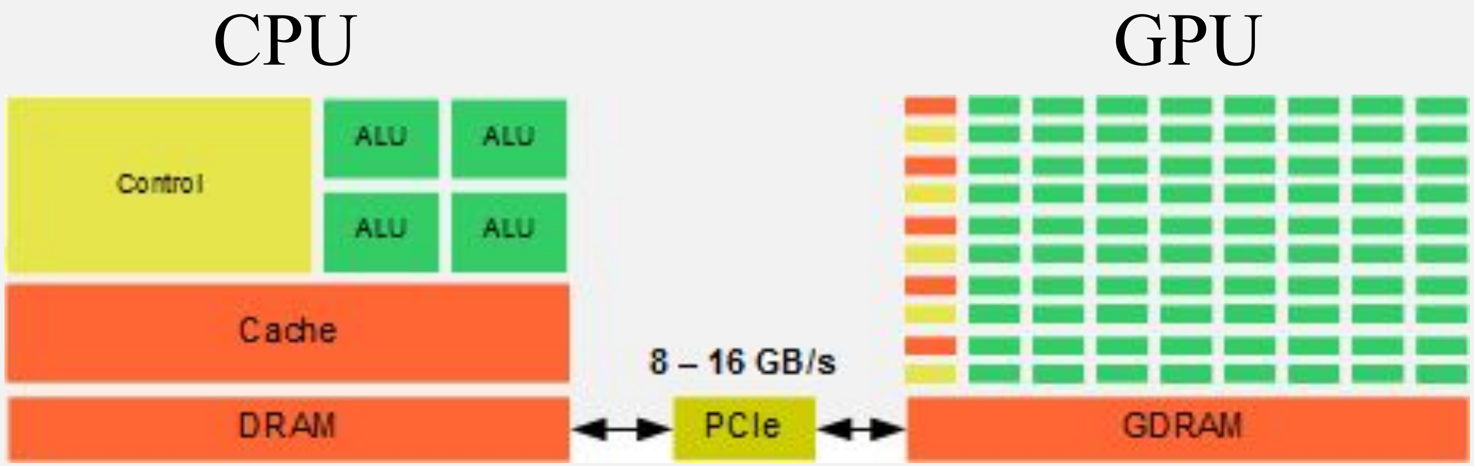
数据来源: http://en.wikipedia.org/wiki/TOP500

2. CUDA

2.1. 什么是CUDA

- Compute Unified Device Architecture
- CPU+GPU异构并行计算架构和应用程序接口
- 由NVIDIA在2006年提出
- 充分地发挥GPU强大的并行处理能力

2.2. CPU与GPU协作



例1. 计算 $f(x) = \sin x \cdot \cos 7x \cdot e^x, x \in [0,1]$

- 其实未必需要使用CPU+GPU异构并行计算
- 有助于理解CPU与GPU协作及CUDA基本概念

2.2. CPU与GPU协作

```
/* CPU codes solving
 * f(x)=sin(x)*cos(7*x)*exp(x)
 * x \in [0,1] */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define nh 10000000

inline double
func(double x){
   return sin(x)*cos(7.*x)*exp(x);
}
```

```
void
evaluate(double *vals){
   for(int i = 0; i < nh; i++)
      vals[i] = func((i+1.0)/nh);
}
int
main(){
   double *vals = (double *)
      malloc(nh*sizeof(double));
   evaluate(vals);
   free(vals);
   return 0;
}
```

2.2. CPU与GPU协作

- CPU V.S. GPU
  - host V.S. device
  - host codes V.S. device codes
- Function Qualifier:
  - __global__: called from host && device codes
  - __device__: called from device && device codes
  - __host__: host codes
- Kernel:
  - qualified with __global__
  - invocation: kernel<<<......>>>(......)
- Execution Configuration Parameters: <<<......>>>

2.2. CPU与GPU协作

```
/* CPU+GPU codes solving
 * f(x)=sin(x)*cos(7*x)*exp(x),
 * x \in [0,1] */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda.h>
#include <cuda_runtime.h>
#define nh 10000000
#define tpb 512
```

```
__inline__ __device__ double
func(double x){
   return sin(x)*cos(7.*x)*exp(x);
}

__global__ void
evaluate(double *vals){
   int i = blockIdx.x * blockDim.x
         + threadIdx.x;
   if(i < nh)
      vals[i] = func((i+1.0)/nh);
}
```

2.2. CPU与GPU协作

```
int
main(){
   double *h_vals, *d_vals;
   int size = nh*sizeof(double);
   h_vals = malloc(size);
   cudaMalloc(&d_vals, size);
   dim3 gridsize(nh/tpb+1,1,1);
   dim3 blocksize(tpb,1,1);
   evaluate<<<gridsize,
     blocksize>>>(d_vals);
```

```
   cudaMemcpy(
     h_vals, d_vals, size,
     cudaMemcpyDeviceToHost);
   free(h_vals);
   cudaDeviceReset();
   return 0;
}
```

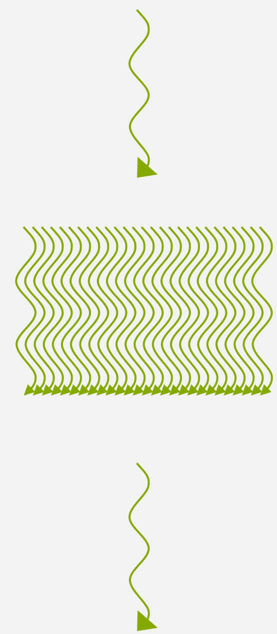
2.2. CPU与GPU协作

```
int
main(){
    double *h_vals, *d_vals;
    int size = nh*sizeof(double);
    h_vals = malloc(size);
    cudaMalloc(&d_vals, size);
    dim3 gridsize(nh/tpb+1,1,1);
    dim3 blocksize(tpb,1,1);
    evaluate<<<gridsize,
      blocksize>>>(d_vals);
cudaMemcpy(
      h_vals, d_vals, size,
      cudaMemcpyDeviceToHost);
    free(h_vals);
    cudaDeviceReset();
    return 0;
}
```

CPU

GPU

CPU

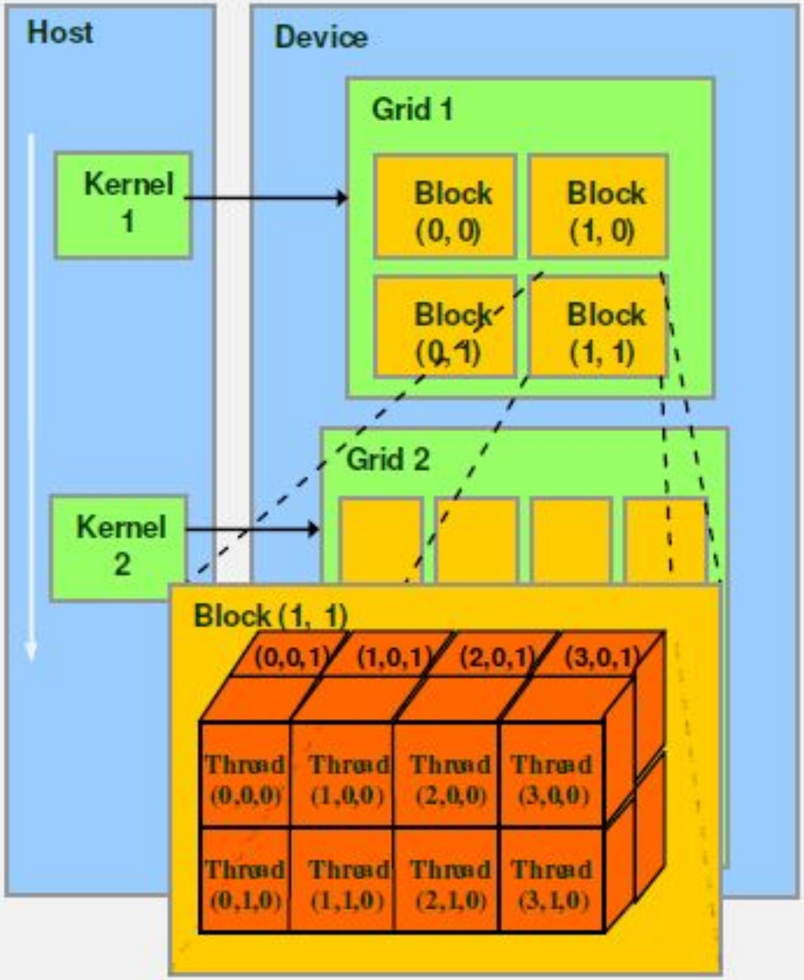


kernel launching creates a grid of threads each of which would execute the body of kernel.

2.3. 线程组织结构

➢ 线程网格(grid):
- 1/2/3维线程块数组
- 每一维大小均有限制
- 线程块<==>线程块索引

➢ 线程块(block):
- 1/2/3维线程数组
- 每一维大小均有限制
- 容纳线程总数有限制
- 所有线程块尺寸都相同
- 线程<==>线程索引



2.3. 线程组织结构

- 线程网格/线程块尺寸都由执行配置参数决定

```
// dim3 is a CUDA's built-in type
dim3 gridsize(nh/tpb+1,1,1);
dim3 blocksize(tpb,1,1);
evaluate<<<gridsize, blocksize>>>(d_vals);
```

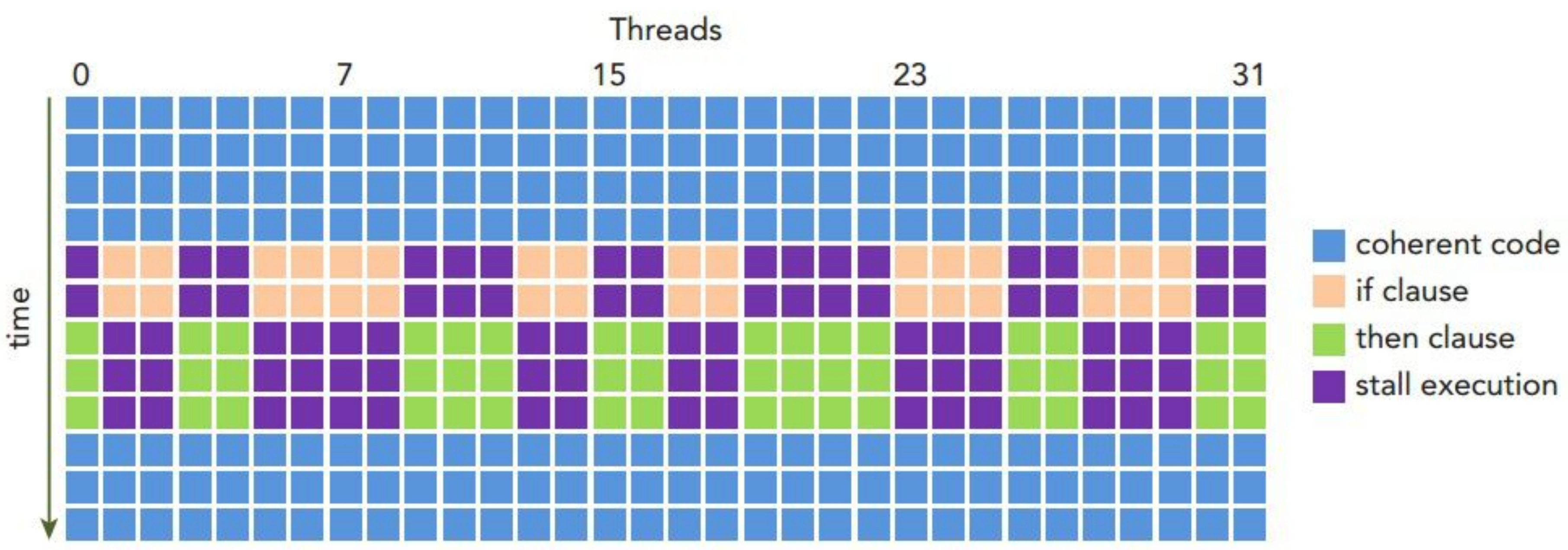
- 在执行核函数过程中，线程有时需要使用线程网格尺寸/线程块尺寸/线程所在线程块在线程网格中的索引/线程在线程块中的索引，这些信息存放在CUDA内建变量gridDim/blockDim/blockIdx/threadIdx之中。

2.4. 线程调度

- 线程束(warp):
  - 线程调度的基本单位
  - 线程块最终划分成若干线程束
  - 32个线程组成1个线程束
  - 不足32个线程将补充空闲线程
- 单指令多线程架构(SIMT):
  - Single Instruction, Multiple Threads
  - 线程束中线程总是同时执行同一指令
  - 显著减少硬件设计成本和功耗

2.4. 线程调度

➢ 线程束分叉(warp divergence):
- 线程束中线程执行不同代码路径
- 线程束中线程串行执行所有代码路径
- 性能显著下降



2.5. 存储器

Memory	Function
register	与CPU端register功能相同。
shared memory	与CPU端L1 Cache功能相似。可编程，潜在带宽非常大，但需要避免bank conflict或者采用广播或多播。
texture memory	GPU独有。缓存对全局内存的随机访问。
constant memory	GPU独有。存储小规模只读数据。访问模式需要充分使用广播并充分考虑带宽限制。
global memory	与CPU端内存功能相似。存储大规模数据。顺序访问可以合并访存。随机访问需要用texture或read-only data cache缓存。
local memory	GPU独有。当register不足或者存在较大数组或结构体时，用local memory替代register。

2.5. 存储器

Memory	on/off chip	Cache	Latency	B.W.	Scope/ Lifetime	size
register	on	\	lowest	highest	thread/ kernel	32-64 K/SM
shared memory	on	\	very low	very high	block/ kernel	48-96 KB/SM
texture memory	off	texture cache	medium	high	grid/ app.	10s KB
constant memory	off	constant cache	very low	low	grid/ app.	64 KB
global memory	off	L2 cache	very high	high	grid/ app.	100s MB - 1s GB
local memory	off	L1/2 cache	very high	high	thread/ kernel	\

2.6. 线程协作

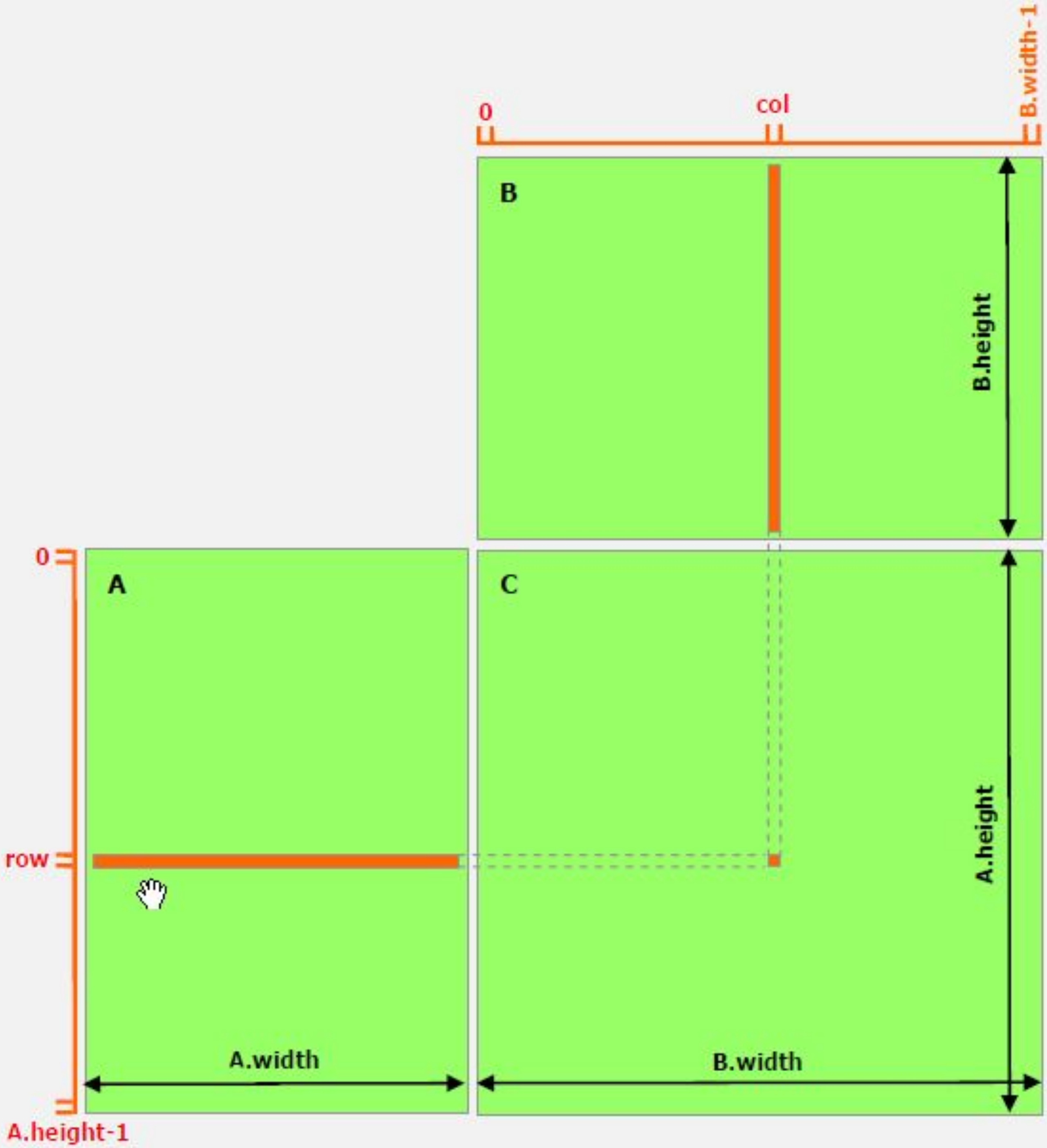
- 栅栏同步:
  - 线程块内线程: __syncthreads()
  - 线程块间线程: 多个核函数
- 存储器访问: shared/global memory
- 原子操作:
  - 共享内存并行编程中不可回避的问题: 竞争条件
  - 线程串行地访问陷入竞争条件的内存位置: 原子操作
  - 性能显著下降 ==> 尽量规避
  - atomic****()

2.7. 例子: 稠密矩阵乘法

Naive: $C = A \cdot B \Leftrightarrow C_{ij} = \sum_k A_{ik} \cdot B_{kj}$

```
__global__ void
mat_mul_naive(float *d_A,
   float *d_B, float *d_C)
{
  // C_{ij} is calculated
  int i = blockIdx.y * blockDim.y
         + threadIdx.y;
  int j = blockIdx.x * blockDim.x
         + threadIdx.x;
```

```
  float accu = 0.0;
  int k;
  for(k=0; k<wA; k++)
    accu +=
        d_A[i*wA+k]*d_B[k*wB+j];
  d_C[i*wB+j] = accu;
}
```



0
col
B.width-1
B
B.height
0
A
C
row
A.height
A.width
B.width
A.height-1

2.7. 例子: 稠密矩阵乘法

Block: $C = A \cdot B \Leftrightarrow C_{ij} = \sum_k A_{ik} \cdot B_{kj}$

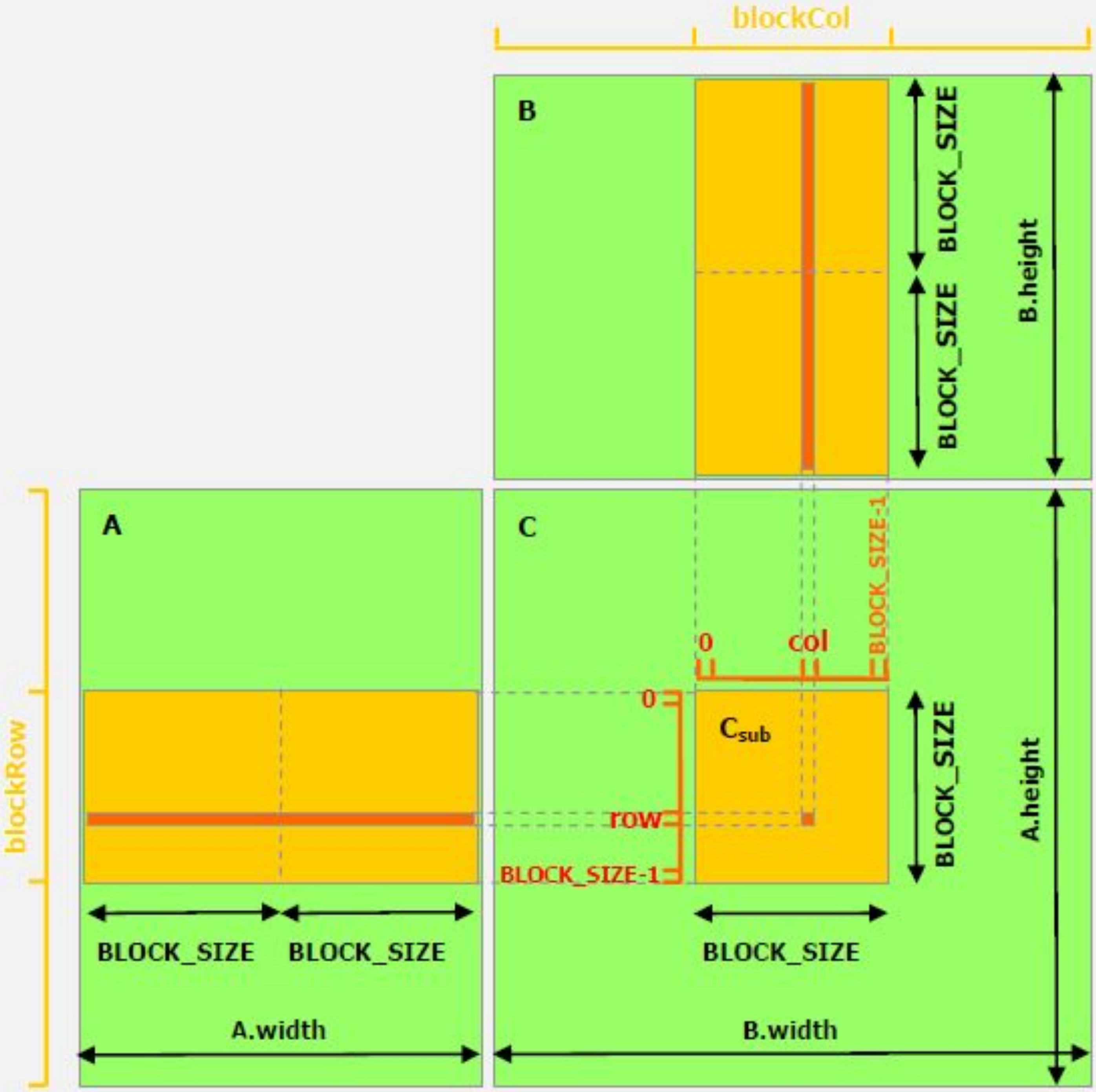
```
__global__ void
mat_mul_block(float *d_A, float
*d_B, float *d_C) {
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  __shared__ float
     As[BLOCK_SIZE][BLOCK_SIZE];
  __shared__ float
     Bs[BLOCK_SIZE][BLOCK_SIZE];
```

```
int aBegin = wA*BLOCK_SIZE*by;
int aEnd   = aBegin+wA-1;
int aStep  = BLOCK_SIZE;
int bBegin = BLOCK_SIZE*bx;
int bStep  = BLOCK_SIZE*wB;
float Csub = 0;
int a, b, k;
for(a = aBegin, b = bBegin;
   a <= aEnd;
   a += aStep, b += bStep) {
```

2.7. 例子: 稠密矩阵乘法

Block: $C = A \cdot B \Leftrightarrow C_{ij} = \sum_{k} A_{ik} \cdot B_{kj}$

```
  As[ty][tx] = d_A[a+wA*ty+tx];
  Bs[ty][tx] = d_B[b+wB*ty+tx];
  __syncthreads();
  for(k = 0; k < BLOCK_SIZE; ++k)
   Csub += AS[ty][k] * Bs[k][tx];
  __syncthreads();
 }
 d_C[wB * (BLOCK_SIZE * by + ty)
  + BLOCK_SIZE * bx + tx] = Csub;
}
```


blockCol
B
BLOCK_SIZE
BLOCK_SIZE
B.height
A
C
BLOCK_SIZE-1
0
col
0
C_sub
row
BLOCK_SIZE-1
BLOCK_SIZE
A.height
blockRow
BLOCK_SIZE
BLOCK_SIZE
BLOCK_SIZE
A.width
B.width

2.7. 例子: 稠密矩阵乘法

为什么要块算法？
提高Compute to Global Memory Access ratio (CGMA)

为了计算出一个元素的访存次数：
naive: 2*wA
block: 2*wA/BLOCK_SIZE

2.7. 例子: 稠密矩阵乘法

"drop-in": cuBLAS

```
......
#include <cublas_v2.h>
......
void
mat_mul_opt(float *d_A,
   float *d_B, float *d_C)
{
  float alpha = 1.0, beta = 0.0;
  cublasHandle_t handle;
  cublasCreate(&handle);
  cublasSgemm(handle,
   CUBLAS_OP_N, CUBLAS_OP_N,
  N, N, N, &alpha, d_A, N, d_B, N,
   &beta, d_C, N);
  cublasDestroy(handle);
}
```

2.7. 例子: 稠密矩阵乘法

表4. naive VS block VS drop-in

method	performance	speedup
naive	80.479812	1
block	33.382137	2.4109
drop-in	3.4565598	23.2832

Conclusion: use libraries that CUDA provides rather than coding by yourself

Thank You！ Any Question?