CUDA编程培训和应用

贾伟乐
中科院超级计算中心
2012年11月8日

jiaweile@gmail.com

提纲

- 5分钟CUDA入门
- CUDA硬件和编程模型
- GPU加速实例
  - 流体GPU加速初探
  - 第一原理计算GPU加速

GP(General-purpose)GPU

- 从Graphic Processing到General-purpose Computing；
  - 众核(Many-core)
  - SIMT(Single Instruction Multiple Threads)
- 协助CPU完成计算任务；
  - CPU端
    - 整理GPU需要用到的数据；
    - 启动计算，决定GPU计算的threads拓扑结构；
    - 接收计算结果；
    - 启动完成，即可继续执行CPU端代码，与GPU异步；
  - GPU端
    - 众多Threads执行计算

5分钟CUDA入门

Just Do It!

CPU-GPU互动

Intel® Pentium® 4 Processor Extreme Edition
6.4 GB/s
82925X MCH
PCI Express* x16 Graphics
8.0 GB/s
DDR2
8.5 GB/s
DDR2
2 GB/s
DMI
ICH6RW
Intel® High Definition Audio
4 PCI Express* x1
500 MB/s
8 Hi-Speed USB 2.0 Ports
60 MB/s
150 MB/s
4 Serial ATA Ports
133 MB/s
6 PCI
Intel® Matrix Storage Technology
Intel® Wireless Connect Technology
BIOS Supports HT Technology

CPU
1.CPU开始程序
北桥芯片组
主内存
2.从PC内存拷贝数据到GPU内存上
3.CPU通过驱动加载命令到GPU
GPU
4.GPU并行执行
GPU内存
5.把运行完的数据拷贝回主存或者直接输出现实

CUDA运行过程

```
#include <stdio.h>
#include <cuda.h>
#include <assert.h>
#define SIZE  20
__global__ void hello(char *result)
{
        int tid = threadIdx.x + blockIdx.x * blockDim.x;
        char * p = "Hello, world!";
        for(int i = 0; i <SIZE; i ++)
                result[i] = p[i];

        printf("thread: %d, %s\n", tid, resul
}
int main()
{
        cudaSetDevice(0);
        char * ptr_d, *ptr_h;

        ptr_h =(char*) malloc(sizeof(char)*SIZE);
        assert( cudaMalloc((void**) &ptr_d, sizeof(char)*SIZE) ==
cudaSuccess);

        hello<<<1, 4>>>(ptr_d);
        cudaThreadSynchronize();

        assert(cudaMemcpy(ptr_h, ptr_d, SIZE*sizeof(char),
cudaMemcpyDeviceToHost)== cudaSuccess);

        printf("on Host: %s\n", ptr_h);

        cudaFree(ptr_d);
        free(ptr_h);

        printf("******END CUDA*********\n");
}
```

0.Kernel 函数

编译： nvcc -arch=“sm_20” main.cu –o main

运行： ./main

2.在GPU上分配内存

```
thread: 0, Hello, world!
thread: 1, Hello, world!
thread: 2, Hello, world!
thread: 3, Hello, world!
on Host: Hello, world!
******END CUDA**********
```

5.释放GPU上的内存

6.结束程序

Follow it!：Hello CUDA World!

- Hello Cuda World !

```
#include <stdio.h>
__global__ void helloCUDA(float *a)
{
    printf("Hello ! data=%f\n",*a);
}
}
int main()
{
    float h_a=1;
    float *d_a,;
    cudaMalloc(&d_a,sizeof(float));
    cudaMemcpy(d_a,&h_a,sizeof(float),cudaMemcpyHostToDevice);
    helloCUDA<<<4,16>>>(d_a);
    cudaMemcpy(&h_a,d_a,sizeof(float),cudaMemcpyHostToDevice);
    cudaFree(d_pa);
    printf("Hello World!\n");
    return 0;
}
```

GPU上运行的函数代码

申请设备内存

CPU源数据—>GPU

启动GPU计算

GPU->CPU结果传回

释放GPU内存

Part I: GPU硬件和编程模型

- **从GPGPU到CUDA**
- CUDA编程模型
- GPU存储器
- CUDA硬件模型
- CUDA程序设计工具
- 新一代Kepler GPU

A Big “New” Problem

If nothing is done to reduce power, Moore’s Law will be at risk

Power for a 1 cm^2 Chip (Watts)

10,000
1,000
100
10
1

‘70 ‘80 ‘90 ‘00 ‘10

4004
8008
8080
8085
8086
286
386
486
Pentium® Processors

Hot Plate
Nuclear Reactor
Rocket Nozzle
Sun’s Surface

Source: Pat Gelsinger, Intel, ISSCC 2001

Graphic Processing Unit (GPU)

- 用于个人计算机、工作站和游戏机的专用图像显示设备
  - 显示卡
    - nVidia和ATI (now AMD)是主要制造商
      - Intel准备通过Larrabee进入这一市场
  - 主板集成
    - Intel

传统GPU架构

Host / FW / VTF

Cull / Clip / Setup

Z-Cull

Shader Instruction Dispatch

2 Tex

Fragment Crossbar

Memory Partition

Memory Partition

Memory Partition

Memory Partition

DRAM(s)

DRAM(s)

DRAM(s)

DRAM(s)

GeForce 7800

Graphics program

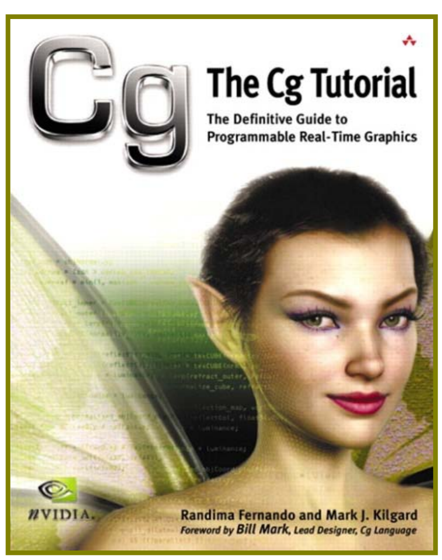
Vertex processors

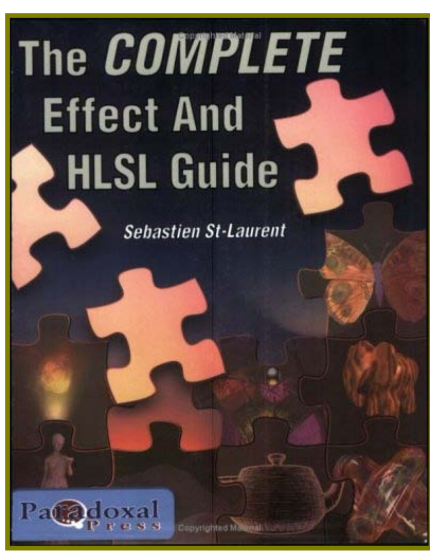
Fragment processors

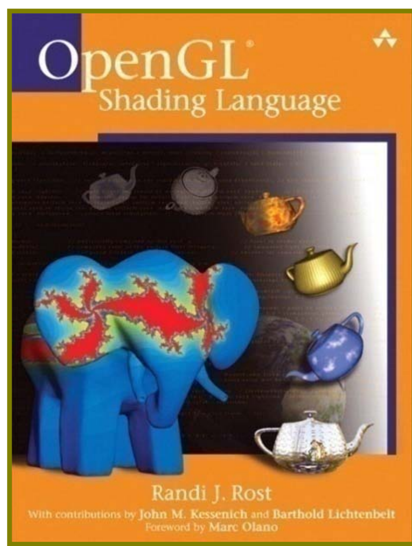
Pixel operations

Output image

High Level GPU Languages

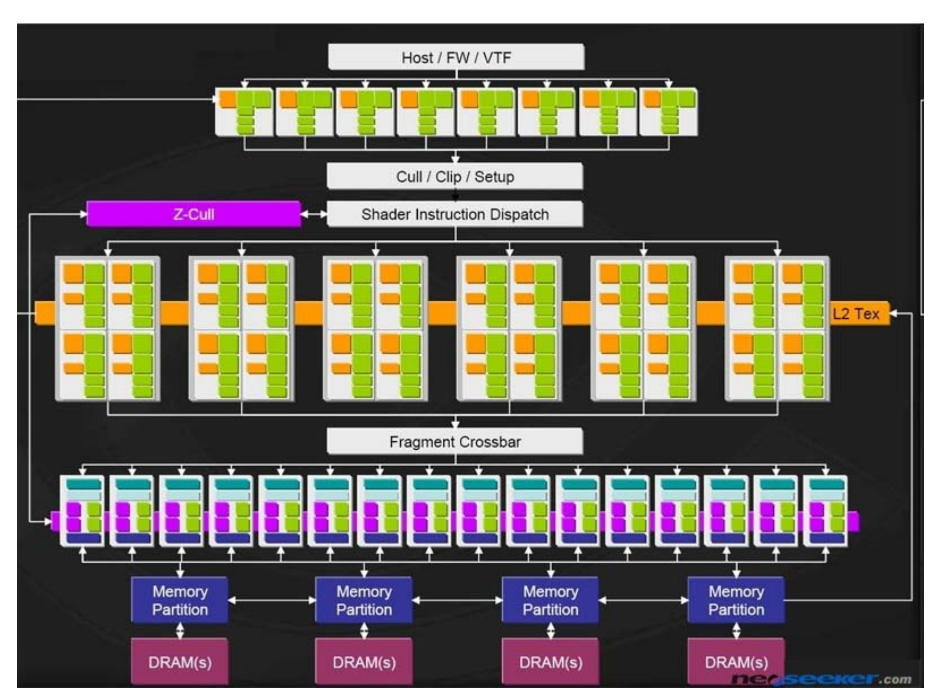






GPGPU

- 核心思想
  - 用图形语言描述通用计算问题
  - 把数据映射到vertex或者fragment处理器
- 但是
  - 硬件资源使用不充分
  - 存储器访问方式严重受限
  - 难以调试和查错
  - 高度图形处理和编程技巧



G80 GPU

Host
Input Assembler
Vtx Thread Issue
Geom Thread Issue
Setup / Rstr / ZCull
Pixel Thread Issue
Thread Processor

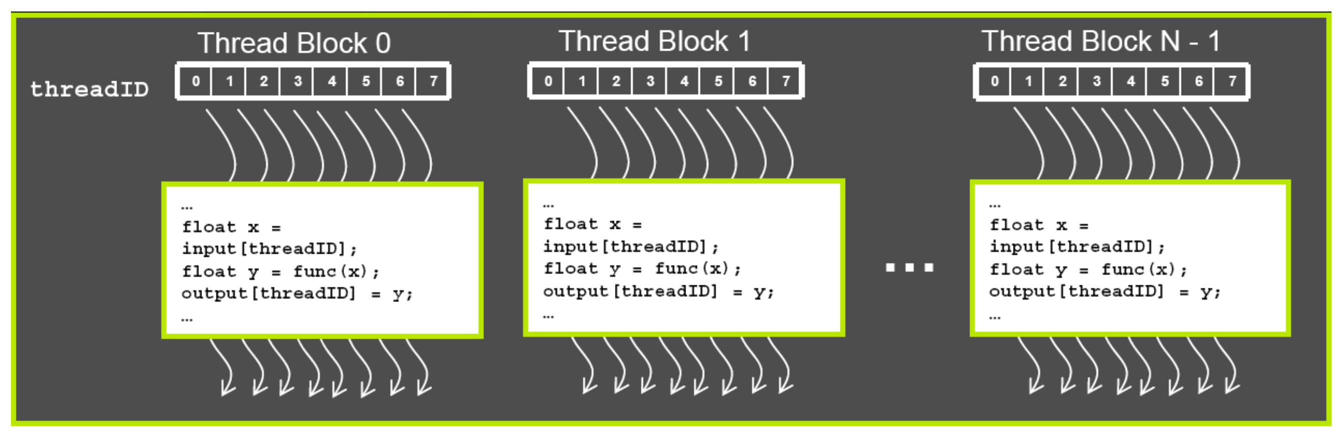
SP TF L1 (×8)
L2 FB (×6)

Streaming Processor (SP)

Streaming Multiprocessor (SM)

CUDA: Compute Unified Device Architecture

- CUDA: 集成CPU + GPU C应用程序
- 通用并行计算模型
  - 单指令、多线程执行模式 (SIMT)
    - 所有线程执行同一段代码(1000s threads on the fly)
    - 大量并行计算资源处理不同数据
  - 隐藏存储器延时
    - 提升计算 / 通信比例
    - 合并相邻地址的内存访问
    - 快速线程切换1 cycle@GPU vs. ~1000 cycles@CPU



GPU accelerated app

Speedup	Application
146X	Interactive visualization of volumetric white matter connectivity
36X	Ionic placement for molecular dynamics simulation on GPU
18X	Transcoding HD video stream to H.264
17X	Simulation in Matlab using .mex file CUDA function
100X	Astrophysics N-body simulation
149X	Financial simulation of LIBOR model with swaptions
47X	GLAME@lab: An M-script API for linear Algebra operations on GPU
20X	Ultrasound medical imaging for cancer diagnostics
24X	Highly optimized object oriented molecular dynamics
30X	Cmatch exact string matching to find similar proteins and gene sequences

Part I: GPU硬件和编程模型

- 从GPGPU到CUDA
- **CUDA编程模型**
- GPU存储器
- CUDA硬件模型
- CUDA程序设计工具
- 新一代Kepler GPU

- "Compute Unified Device Architecture"
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded co-processor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute - graphics free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management

CUDA Performance

CUDA/G80 Advantage Over Dual Core

197x

47x

20x

10x

Rigid Body Physics Solver

Matrix Numerics

BLAS1: 60+ GB/s
BLAS3: 100+ GFLOPS

Wave Equation

FDTD: 1.2 Gcells/s
FFT: 52 GFLOPS

(GFLOPS as defined by benchFFT)

Biological Sequence Match

SSEARCH: 5.2 Gcells/s

Finance

Black Scholes: 4.7 GOptions/s

CUDA Highlights: Easy and Lightweight

- The API is an extension to the ANSI C programming language
  → Low learning curve

- The hardware is designed to enable lightweight runtime and driver
  → High performance

Extended C

- **Declspecs**
  - **global, device, shared, local, constant**
- **关键字**
  - **threadIdx, blockIdx**
- **内部函数**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];
  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

Extended C

Integrated source
(foo.cu)

nvcc
EDG C/C++ frontend
Open64 Global Optimizer

GPU Assembly
foo.s

CPU Host Code
foo.cpp

OCG

gcc / cl

G80 SASS
foo.sass

CUDA Programming Model: A Highly Multithreaded Coprocessor

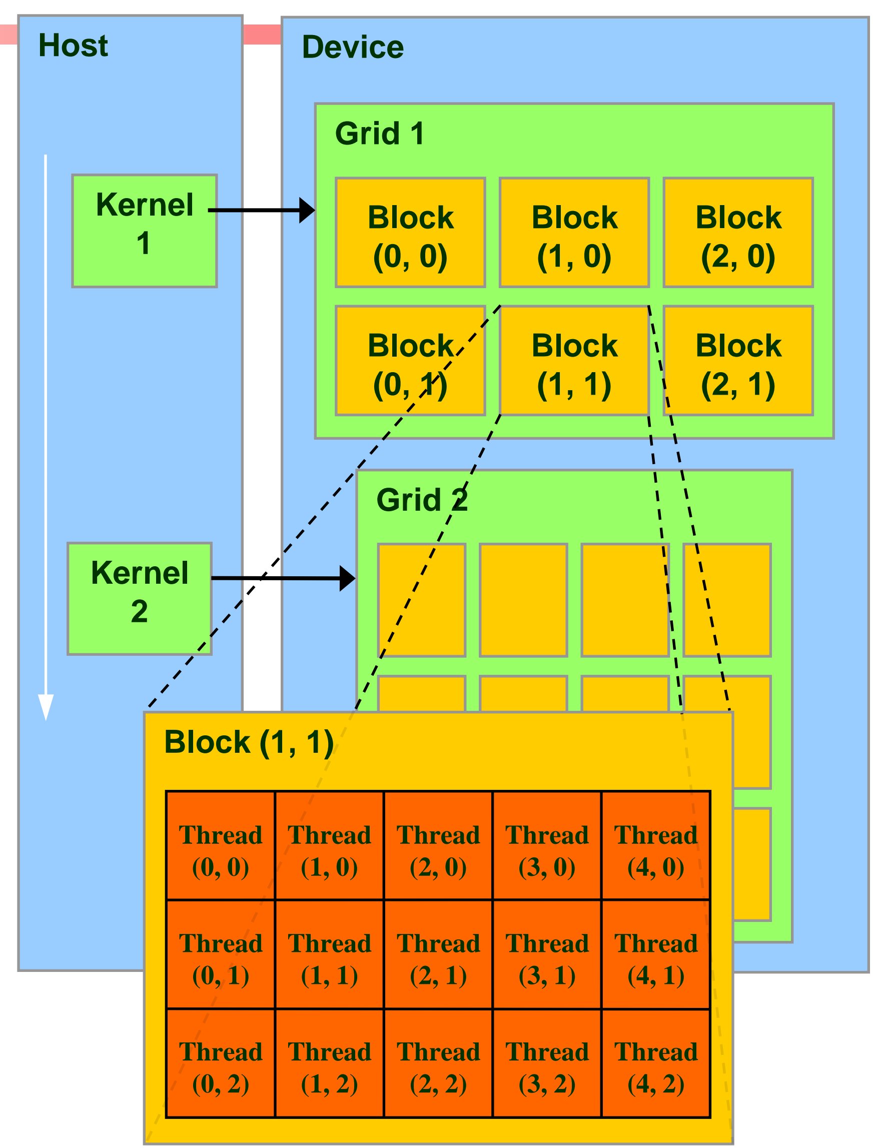
中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Thread Batching: Grids and Blocks

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

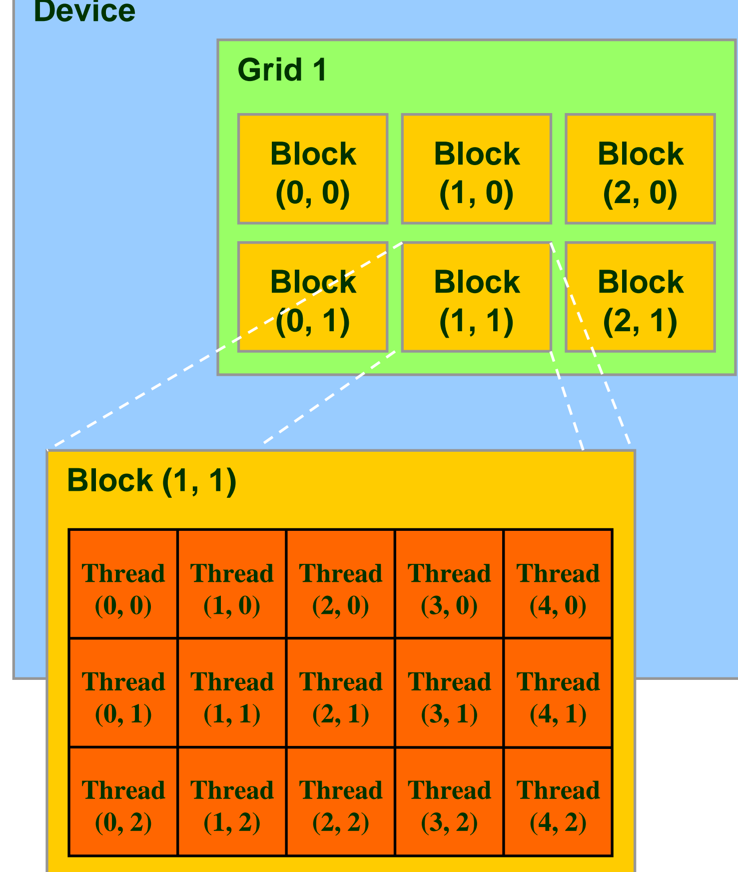
- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



Courtesy: NDVIA

Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



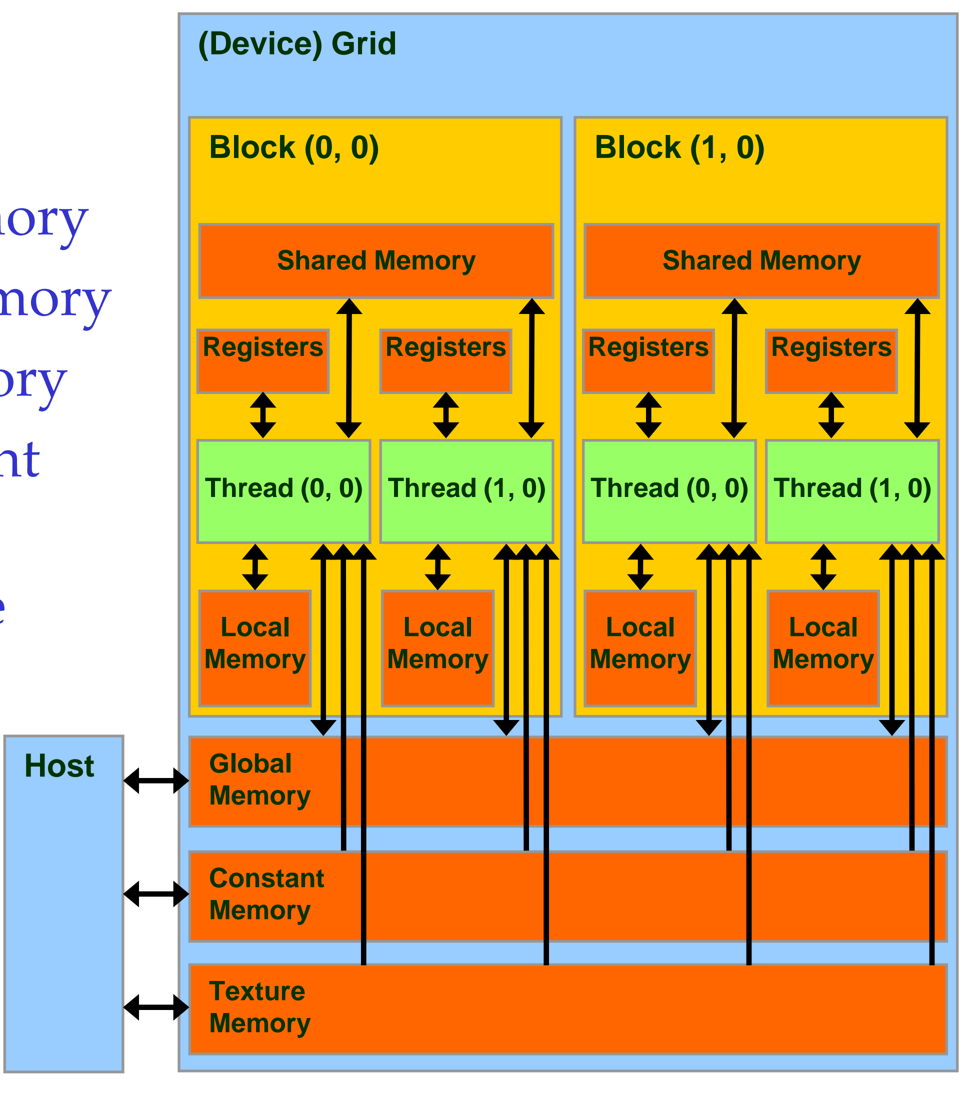
Courtesy: NDVIA

CUDA Device Memory Space Overview

中国科学院
计算机网络信息中心
Computer Network Information Center,
Chinese Academy of Sciences

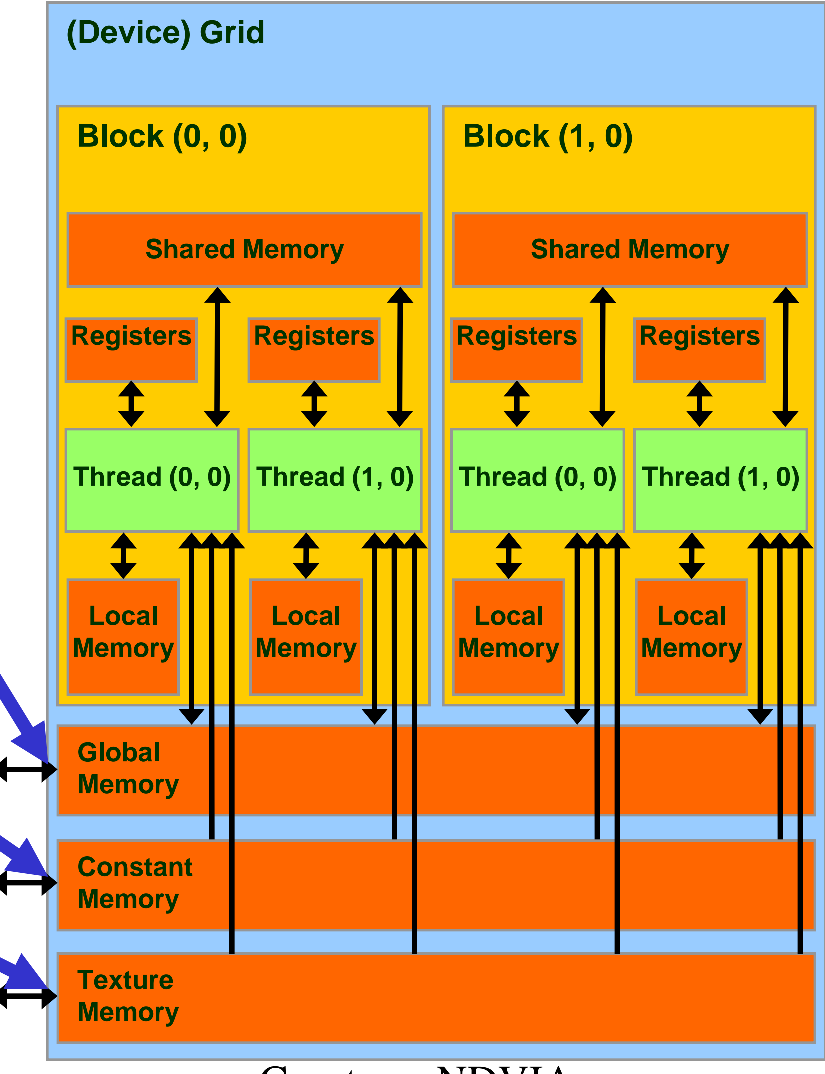
- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memories



Global, Constant, and Texture Memories (Long Latency Accesses)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads

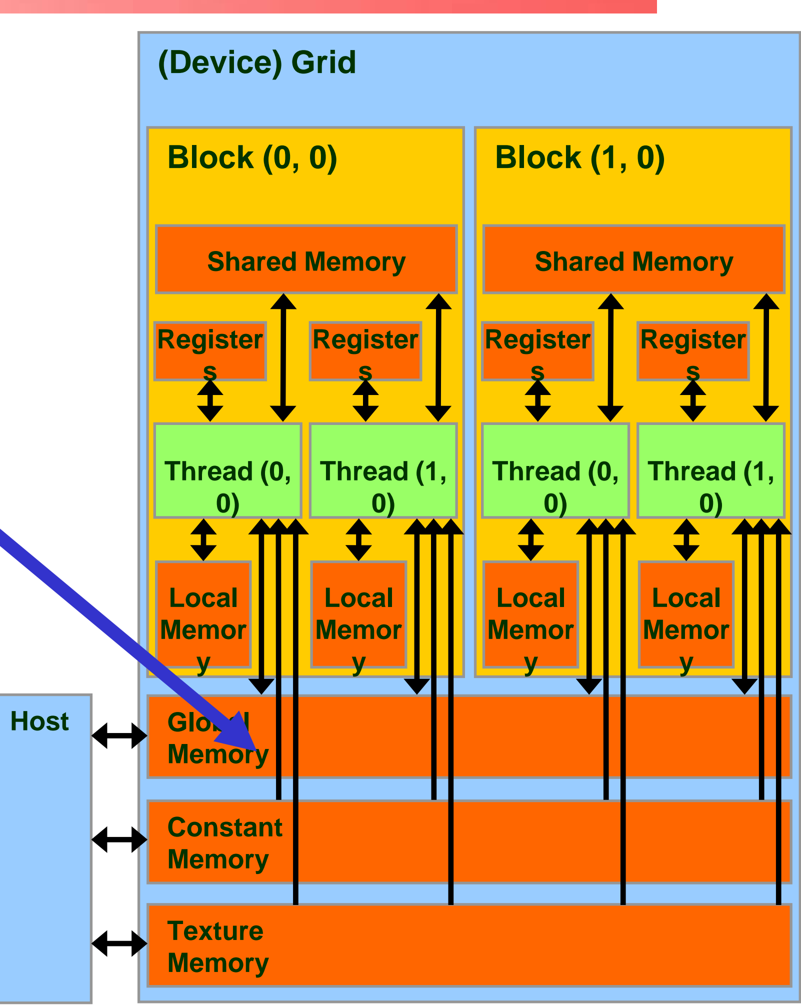


Courtesy: NDVIA

CUDA Device Memory Allocation

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

- cudaMalloc()
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object



CUDA Device Memory Allocation (cont.)

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to **Md.elements**
  - "d" is often used to indicate a device data structure

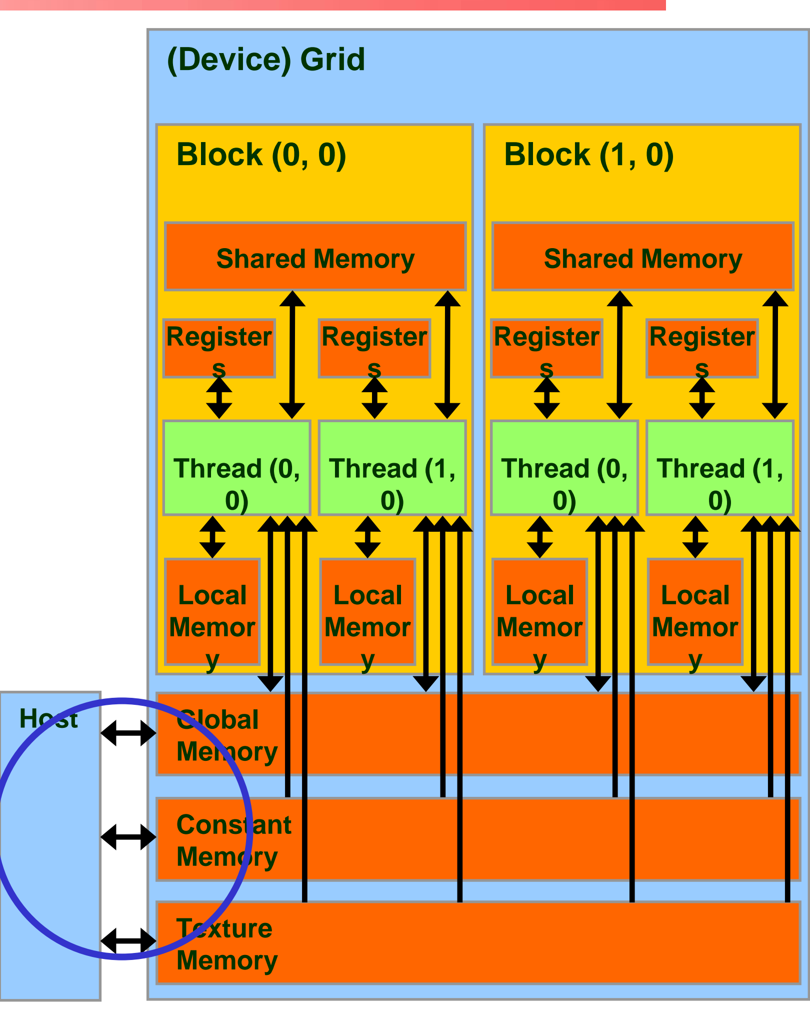
```
MATRIX_SIZE = 64;
float *Md.elements;
int size = MATRIX_SIZE * MATRIX_SIZE * sizeof(float);
```

```
cudaMalloc((void**)&Md.elements, size);
cudaFree(ptr);
```

CUDA Host-Device Data Transfer

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device



CUDA Host-Device Data Transfer (cont.)

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

```
cudaMemcpy(Md.elements, M.elements, size,
    cudaMemcpyHostToDevice);

cudaMemcpy(M.elements, Md.elements, size,
    cudaMemcpyDeviceToHost);
```

CUDA Function Declarations

中国科学院
计算机网络信息中心
Computer Network Information Center,
Chinese Academy of Sciences

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together

CUDA Function Declarations (cont.)

中国科学院
计算机网络信息中心
Computer Network Information Center,
Chinese Academy of Sciences

- `__device__` functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

Calling a Kernel Function – Thread Creation

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3    DimGrid(100, 50);     // 5000 thread blocks
dim3    DimBlock(4, 8, 8);    // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared
   memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes
   >>>(...);
```

- Any call to a kernel function is synchronous
  - Blocks until completion

A Simple Running Example Matrix Multiplication

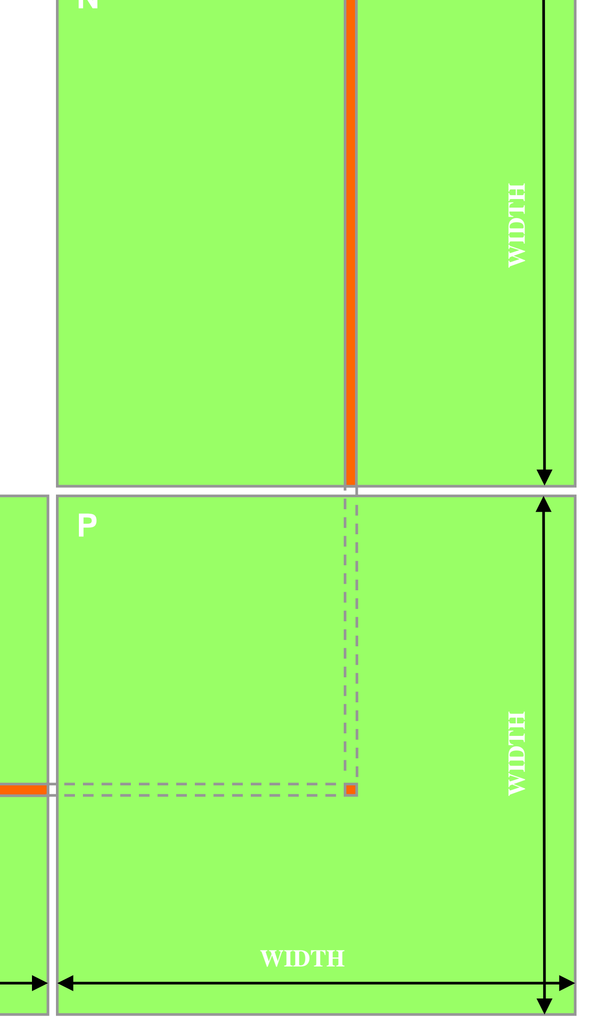
中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

- A straightforward matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Leave shared memory usage until later
  - Local, register usage
  - Thread ID usage
  - Memory data transfer API between host and device

Square Matrix Multiplication Example

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

- P = M * N of size WIDTH x WIDTH
- Without blocking:
  - One thread handles one element of P



Step 1: Matrix Data Transfers

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

```
// Allocate the host memory M where we will copy to device
Matrix AllocateMatrix(const int height, const int width, float initVal)
{
    Matrix M;
    M.width  = MATRIX_SIZE;
    M.height = MATRIX_SIZE;
    M.pitch  = MATRIX_SIZE;
    int size = MATRIX_SIZE * MATRIX_SIZE * sizeof(float);
    M.elements = (float*) malloc(size);

    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            M.elements[i*width + j] = initVal;
        }
    }
    return M;
}
```

Step 2: Matrix Multiplication A Simple Host Code in C

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

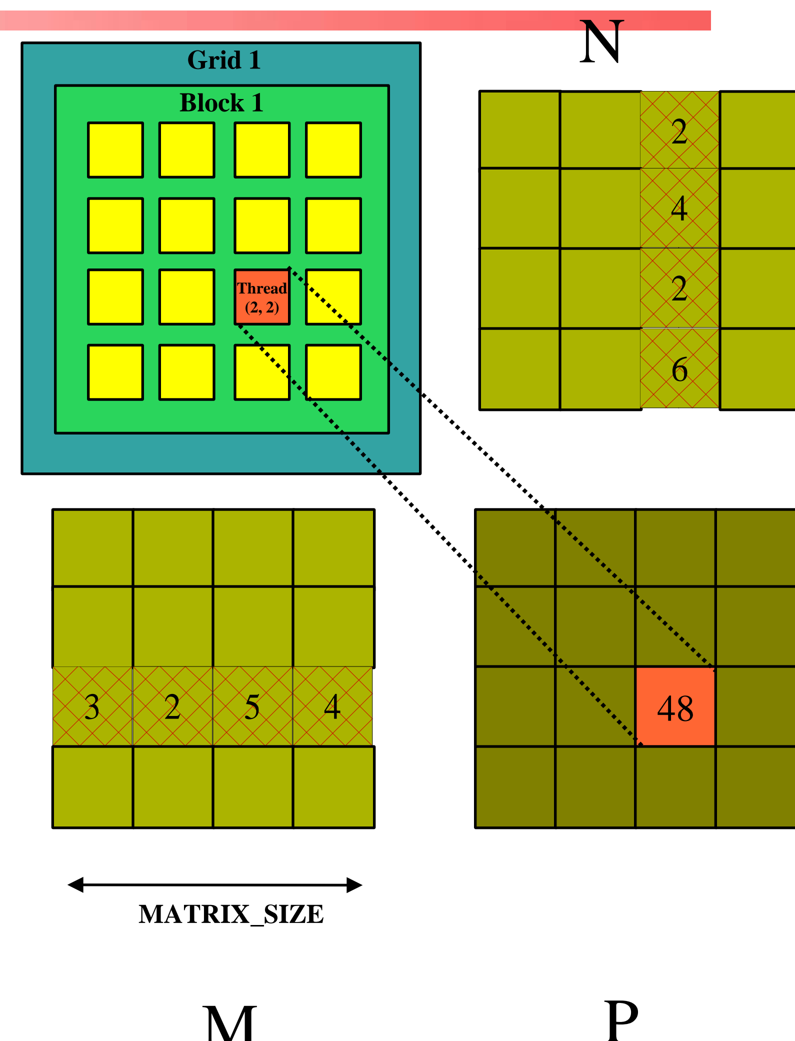
// Matrix multiplication on the (CPU) host in double precision
// for simplicity, we will assume that all dimensions are equal

```
void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i)
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k];
                double b = N.elements[k * N.width + j];
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
}
```

Multiply Using One Thread Block

中国科学院
计算机网络信息中心
Computer Network Information Center,
Chinese Academy of Sciences

- One Block of threads compute matrix P
  - Each thread computes one element of P
- Each thread
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block



Step 2: Matrix Multiplication Host-side Main Program Code

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

```
int main(void) {
// Allocate and initialize the matrices
    Matrix  M  = AllocateMatrix(MATRIX_SIZE, MATRIX_SIZE, 1);
    Matrix  N  = AllocateMatrix(MATRIX_SIZE, MATRIX_SIZE, 1);
    Matrix  P  = AllocateMatrix(MATRIX_SIZE, MATRIX_SIZE, 0);

// M * N on the device
    MatrixMulOnDevice(M, N, P);

// Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);
return 0;
}
```

Step 2: Matrix Multiplication Host-side code

```
// Matrix multiplication on the device
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
   // Load M and N to the device
   Matrix Md = AllocateDeviceMatrix(M);
   CopyToDeviceMatrix(Md, M);
   Matrix Nd = AllocateDeviceMatrix(N);
   CopyToDeviceMatrix(Nd, N);

   // Allocate P on the device
   Matrix Pd = AllocateDeviceMatrix(P);
   CopyToDeviceMatrix(Pd, P); // Clear memory
```

Step 2: Matrix Multiplication Host-side Code (cont.)

```
    // Setup the execution configuration
    dim3 dimBlock(MATRIX_SIZE, MATRIX_SIZE);
    dim3 dimGrid(1, 1);

    // Launch the device computation threads!
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

    // Read P from the device
    CopyFromDeviceMatrix(P, Pd);

    // Free device matrices
    FreeDeviceMatrix(Md);
    FreeDeviceMatrix(Nd);
    FreeDeviceMatrix(Pd);
}
```

Step 2: Matrix Multiplication Device Device-side Kernel Function

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

Step 2: Device Kernel Function
Matrix Multiplication (cont.)

```
    for (int k = 0; k < MATRIX_SIZE; ++k)
    {
        float Melement = M.elements[ty * M.pitch + k];
        float Nelement = N.elements[k * N.pitch + tx];
        Pvalue += Melement * Nelement;
    }
    // Write the matrix to device memory;
    // each thread writes one element
    P.elements[ty * P.pitch + tx] = Pvalue;
}
```

Step 2: Some Loose Ends

```
// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M)
{
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMalloc((void**)&Mdevice.elements, size);
    return Mdevice;
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}

void FreeMatrix(Matrix M) {
    free(M.elements);
}
```

Step 2: Some Loose Ends (cont.)

```
// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost)
{
   int size = Mhost.width * Mhost.height * sizeof(float);
   cudaMemcpy(Mdevice.elements, Mhost.elements, size,
              cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice)
{
   int size = Mdevice.width * Mdevice.height * sizeof(float);
   cudaMemcpy(Mhost.elements, Mdevice.elements, size,
              cudaMemcpyDeviceToHost);
}
```

Part I: GPU硬件和编程模型

- 从GPGPU到CUDA
- CUDA编程模型
- **GPU存储器**
- CUDA硬件模型
- CUDA程序设计工具
- 新一代Kepler GPU

GeForce 8800

- Maximum number of threads per block: 512
- Maximum size of each dimension of a grid: 65,535
- Number of streaming multiprocessors (SM):
  - GeForce 8800 GTX: 16 @ 675 MHz
  - GeForce 8800 GTS: 12 @ 600 MHz
- Device memory:
  - GeForce 8800 GTX: 768 MB
  - GeForce 8800 GTS: 640 MB
- Shared memory per multiprocessor: 16KB divided in 16 banks
- Constant memory: 64 KB
- Warp size: 32 threads (16 Warps/Block)

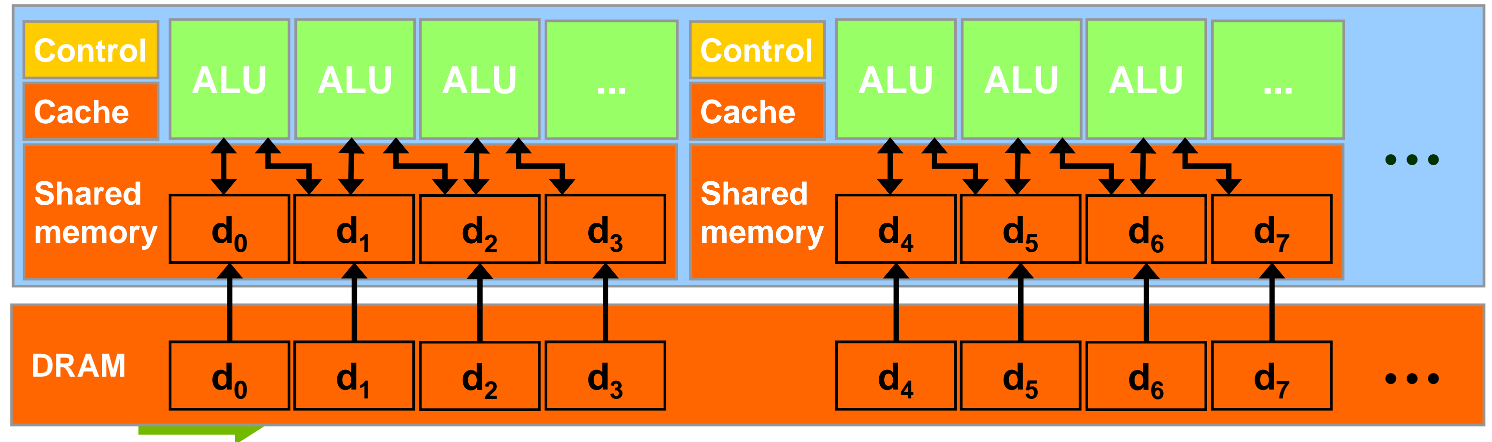
What is the GPU Good at?

- The GPU is good at
  data-parallel processing
  - The same computation executed on many data elements in parallel – low control flow overhead

  with high SP floating point arithmetic intensity
  - Many calculations per memory access
  - Currently also need high floating point to integer ratio
- High floating-point arithmetic intensity and many data elements mean that memory access latency can be hidden with calculations instead of big data caches – Still need to avoid bandwidth saturation!

CUDA Highlights: On-Chip Shared Memory

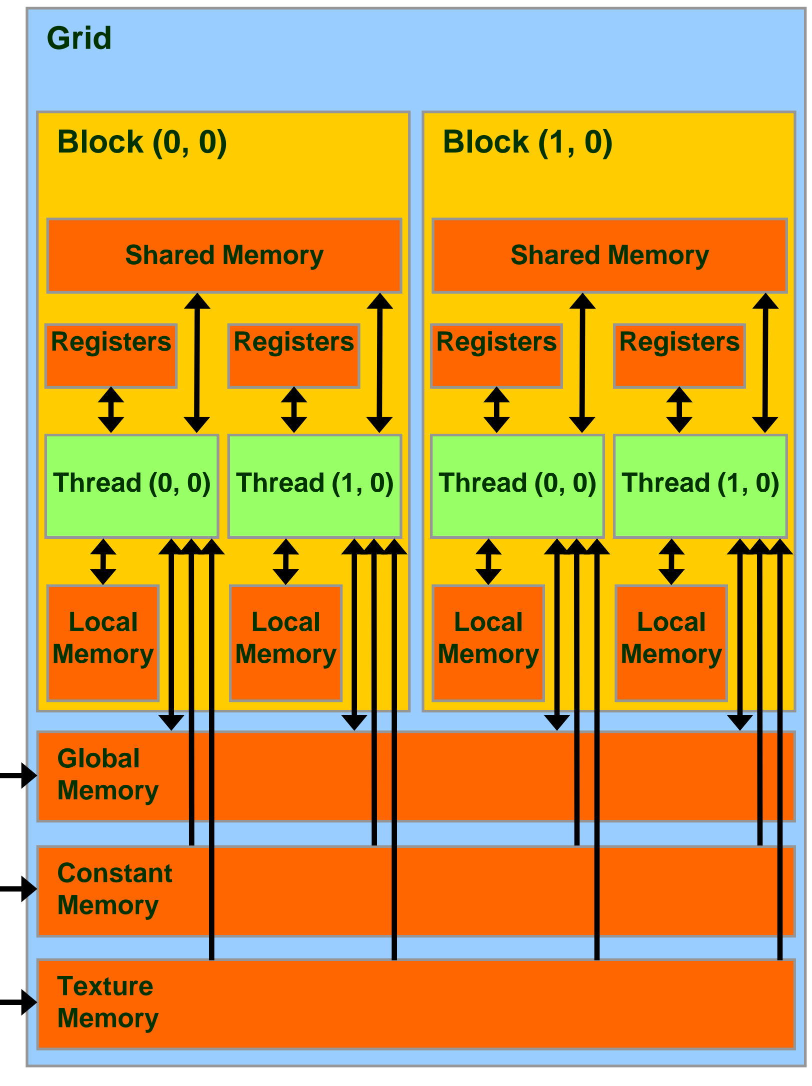
- CUDA enables access to a parallel on-chip shared memory for efficient inter-thread data sharing



Big memory bandwidth savings

Programming Model: Memory Spaces

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory



- The host can read/write global, constant, and texture memory

A Common Programming Pattern

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

- Local and global memory reside in device memory (DRAM) - much slower access than shared memory
- So, a profitable way of performing computation on the device is to block data and computation to take advantage of fast shared memory:
  - Partition data into data subsets that fit into shared memory
  - Handle each data subset with one thread block by:
    - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory

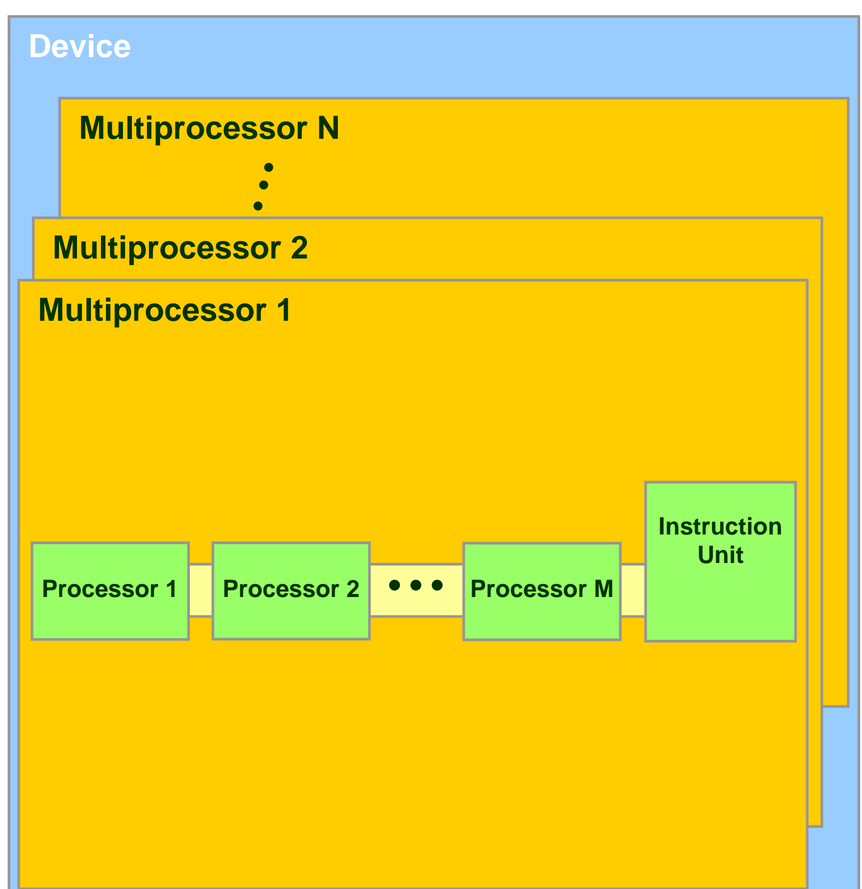
A Common Programming Pattern (cont.)

中国科学院 计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

- Texture and Constant memory also reside in device memory (DRAM) - much slower access than shared memory
  - But… cached!
  - Highly efficient access for read-only data
- Carefully divide data according to access patterns
  - R/O no structure → constant memory
  - R/O array structured → texture memory
  - R/W shared within Block → shared memory
  - R/W registers spill to local memory
  - R/W inputs/results → global memory

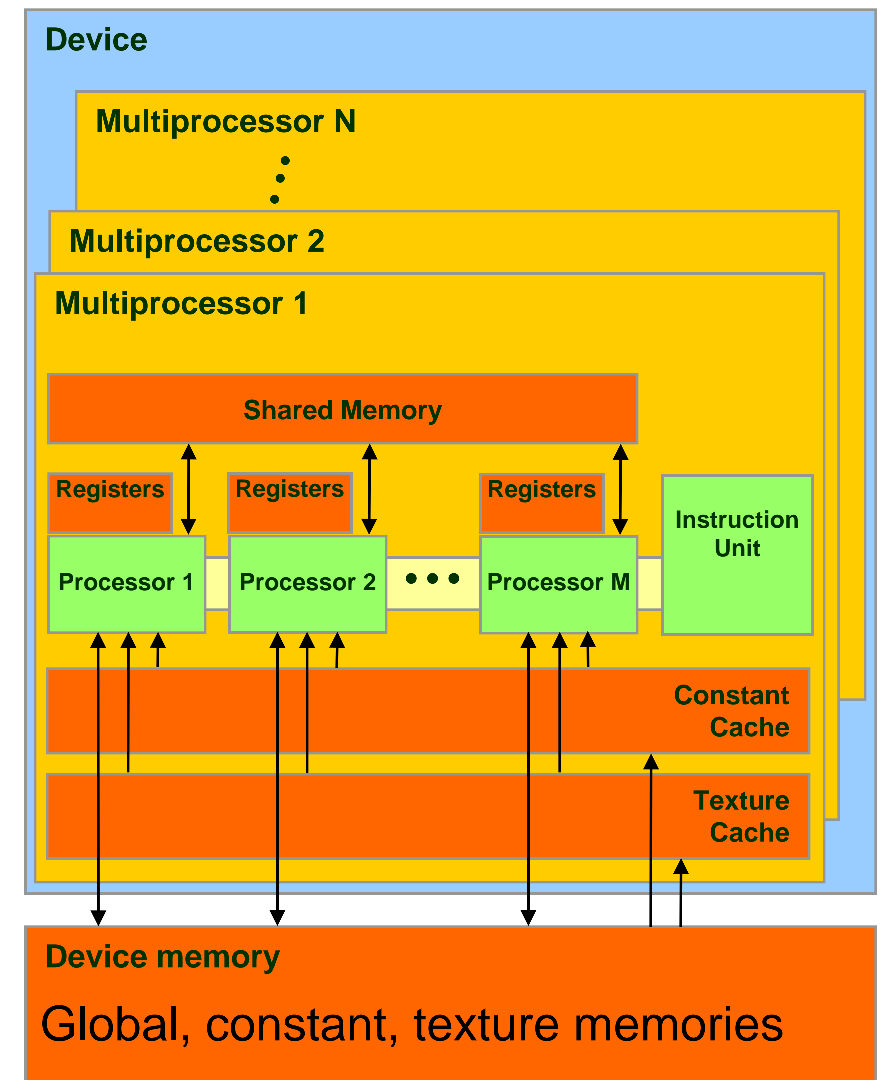
G80 Hardware Implementation: A Set of SIMD Multiprocessors

- The device is a set of 16 multiprocessors
- Each multiprocessor is a set of 32-bit processors with a Single Instruction Multiple Data architecture – shared instruction unit
- At each clock cycle, a multiprocessor executes the same instruction on a group of threads called a warp
- The number of threads in a warp is the warp size



Hardware Implementation: Memory Architecture

- The local, global, constant, and texture spaces are regions of device memory
- Each multiprocessor has:
  - A set of 32-bit registers per processor
  - On-chip shared memory
    - Where the shared memory space resides
  - A read-only constant cache
    - To speed up access to the constant memory space
  - A read-only texture cache
    - To speed up access to the texture memory space



Hardware Implementation: Execution Model (review)

- Each thread block of a grid is split into warps, each gets executed by one multiprocessor (SM)
  - The device processes only one grid at a time
- Each thread block is executed by one multiprocessor
  - So that the shared memory space resides in the on-chip shared memory
- A multiprocessor can execute multiple blocks concurrently
  - Shared memory and registers are partitioned among the threads of all concurrent blocks
  - So, decreasing shared memory usage (per block) and register usage (per thread) increases number of blocks that can run concurrently

Threads, Warps, Blocks

- There are (up to) 32 threads in a Warp
  - Only <32 when there are fewer than 32 **total** threads
- There are (up to) 16 Warps in a Block
- Each Block (and thus, each Warp) executes on a single SM
- G80 has 16 SMs
- At least 16 Blocks required to “fill” the device
- More is better
  - If resources (registers, thread space, shared memory) allow, more than 1 Block can occupy each SM

More Terminology Review

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

- device = GPU = set of multiprocessors
- Multiprocessor = set of processors & shared memory
- Kernel = GPU program
- Grid = array of thread blocks that execute a kernel
- Thread block = group of SIMD threads that execute a kernel and can communicate via shared memory

Memory	Location	Cached	Access	Who
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A - resident	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

Access Times

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached

Application Programming Interface

- The API is an extension to the C programming language
- It consists of:
  - Language extensions
    - To target portions of the code for execution on the device
  - A runtime library split into:
    - A common component providing built-in vector types and a subset of the C runtime library in both host and device codes
    - A host component to control and access one or more devices from the host
    - A device component providing device-specific functions

Language Extensions: Variable Type Qualifiers

	Memory	Scope	Lifetime
`__device__ __local__ int LocalVar;`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`

- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory

Variable Type Restrictions

- Pointers can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:

```
__global__ void KernelFunc(float* ptr)
```

  - Obtained as the address of a global variable:

```
float* ptr = &GlobalVar;
```

Language Extensions: Built-in Variables

- `dim3 gridDim;`
  - Dimensions of the grid in blocks (`gridDim.z` unused)
- `dim3 blockDim;`
  - Dimensions of the block in threads
- `dim3 blockIdx;`
  - Block index within the grid
- `dim3 threadIdx;`
  - Thread index within the block

Common Runtime Component

- Provides:
  - Built-in vector types
  - A subset of the C runtime library supported in both host and device codes

Common Runtime Component: Built-in Vector Types

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

- `[u]char[1..4]`, `[u]short[1..4]`, `[u]int[1..4]`, `[u]long[1..4]`, `float[1..4]`
  - Structures accessed with `x`, `y`, `z`, `w` fields:

```
uint4 param;
int y = param.y;
```

- `dim3`
  - Based on `uint3`
  - Used to specify dimensions

Common Runtime Component: Mathematical Functions

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`
- Etc.
  - When executed on the host, a given function uses the C runtime implementation if available
  - These functions are only supported for scalar types, not vector types

Host Runtime Component

- Provides functions to deal with:
  - Device management (including multi-device systems)
  - Memory management
  - Error handling

- Initializes the first time a runtime function is called

- A host thread can invoke device code on only one device
  - Multiple host threads required to run on multiple devices

Host Runtime Component: Memory Management

- Device memory allocation
  - `cudaMalloc(), cudaFree()`
- Memory copy from host to device, device to host, device to device
  - `cudaMemcpy(), cudaMemcpy2D(), cudaMemcpyToSymbol(), cudaMemcpyFromSymbol()`
- Memory addressing
  - `cudaGetSymbolAddress()`

Device Runtime Component: Mathematical Functions

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

- Some mathematical functions (e.g. `sin(x)`) have a less accurate, but faster device-only version (e.g. `__sin(x)`)
  - `__pow`
  - `__log`, `__log2`, `__log10`
  - `__exp`
  - `__sin`, `__cos`, `__tan`

Device Runtime Component: Synchronization Function

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

- `void __syncthreads();`
- Synchronizes all threads in a block
- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW/WAR/WAW hazards when accessing shared or global memory
- Allowed in conditional constructs only if the conditional is uniform across the entire thread block

Compilation

- Any source file containing CUDA language extensions must be compiled with nvcc
- nvcc is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- nvcc can output:
  - Either C code
    - That must then be compiled with the rest of the application using another tool
  - Or object code directly

Linking

- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (`cudart`)
  - The CUDA core library (`cuda`)

Debugging Using the Device Emulation Mode

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread
- When running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. `printf`) and vice-versa
  - Detect deadlock situations caused by improper usage of `__syncthreads`

Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads could produce different results.
- Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode
- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs, instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host

Part I: GPU硬件和编程模型

- 从GPGPU到CUDA
- CUDA编程模型
- GPU存储器
- **CUDA硬件模型**
- CUDA程序设计工具
- 新一代Kepler GPU

Work Distribution for CUDA

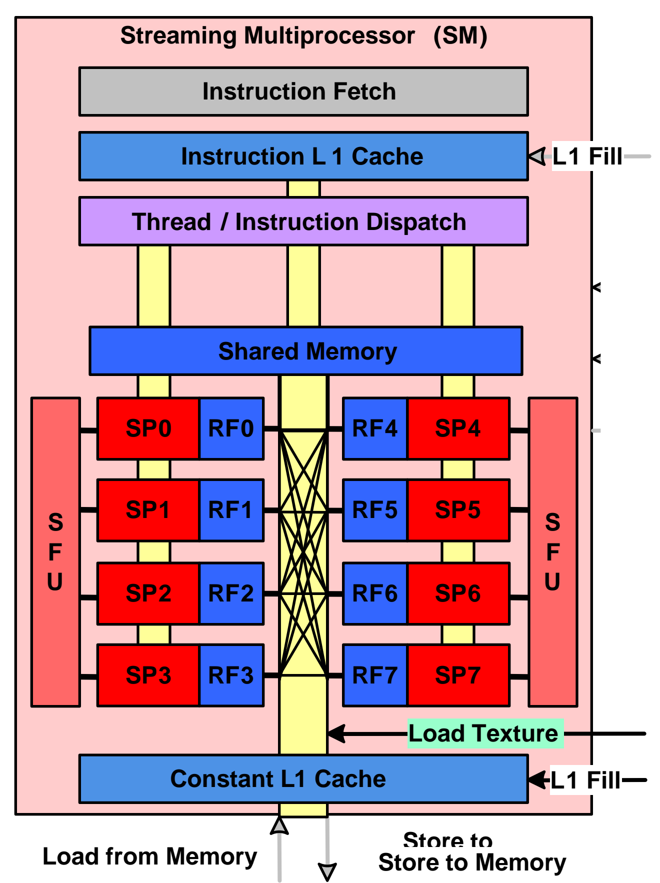
- Grid is launched on the SPA
- Thread Blocks are serially distributed to all the SM's
  - Potentially >1 Thread Block per SM
- Each SM launches Warps of Threads
  - 2 levels of parallelism
- SM schedules and executes Warps that are ready to run
- As Warps and Thread Blocks complete, resources are freed
  - SPA can distribute more Thread Blocks
- Load/Store to/from Memory can occur at any time in program

Host
Kernel 1
Kernel 2
Device
Grid 1
Block (0, 0) | Block (1, 0) | Block (2, 0)
Block (0, 1) | Block (1, 1) | Block (2, 1)
Grid 2
Block (1, 1)

Thread (0, 0)	Thread (1, 0)	Thread (2, 0)	Thread (3, 0)	Thread (4, 0)
Thread (0, 1)	Thread (1, 1)	Thread (2, 1)	Thread (3, 1)	Thread (4, 1)
Thread (0, 2)	Thread (1, 2)	Thread (2, 2)	Thread (3, 2)	Thread (4, 2)

Streaming Multiprocessor (SM)

- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 768 threads active
  - SIMD instruction per 16/32 threads
  - Cover latency of texture/memory loads
- Hot clock 1.35 GHz
  - 20+ GFLOPS
- local register file (RF*n*)
- 16 KB shared memory
- DRAM texture and memory access

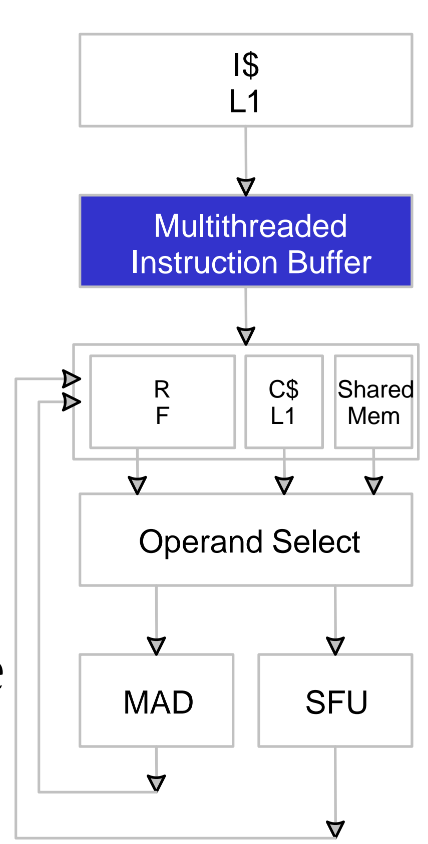


Streaming Processor (SP)

- One scalar ALU
  - Serves as datapath for 1 thread of a warp
  - Each SM has 8 SP
  - Each SM has 2 SFU
- Threads
  - A warp instruction can issue every clock
  - Need ~8 warps to typically saturate the MAD/SFU pipes

SM Instruction Buffer

- Fetch one warp instruction/cycle
  - from instruction L1 cache
  - into any instruction buffer slot
- Issue one "ready-to-go" warp instruction/cycle
  - from any warp - instruction buffer slot
  - operand scoreboarding used to prevent hazards
- Issue selection based on round-robin/age of warp
- SM broadcasts SIMD instruction to 32 Threads of a Warp



Scoreboarding

- All operands of all instructions in the Instruction Buffer are scoreboarded
  - prevents hazards
  - cleared instructions are eligible for issue
- Decoupled Memory/Processor pipelines
  - any thread can continue to issue instructions until scoreboarding prevents issue
  - allows Memory/Processor ops to proceed in shadow of Memory/Processor ops

Branching

- Conditional branch to label, subroutine call
- SM schedules each Warp independently
- SM executes 32 threads of a Warp as a SIMD instruction
  - SM enables/disables sets of threads when branches diverge
- Synchronization
  - Re-converge diverged threads in a Warp
- Barrier Synchronization
  - CUDA uses barrier instruction to synchronize multiple Warps in a Thread Block

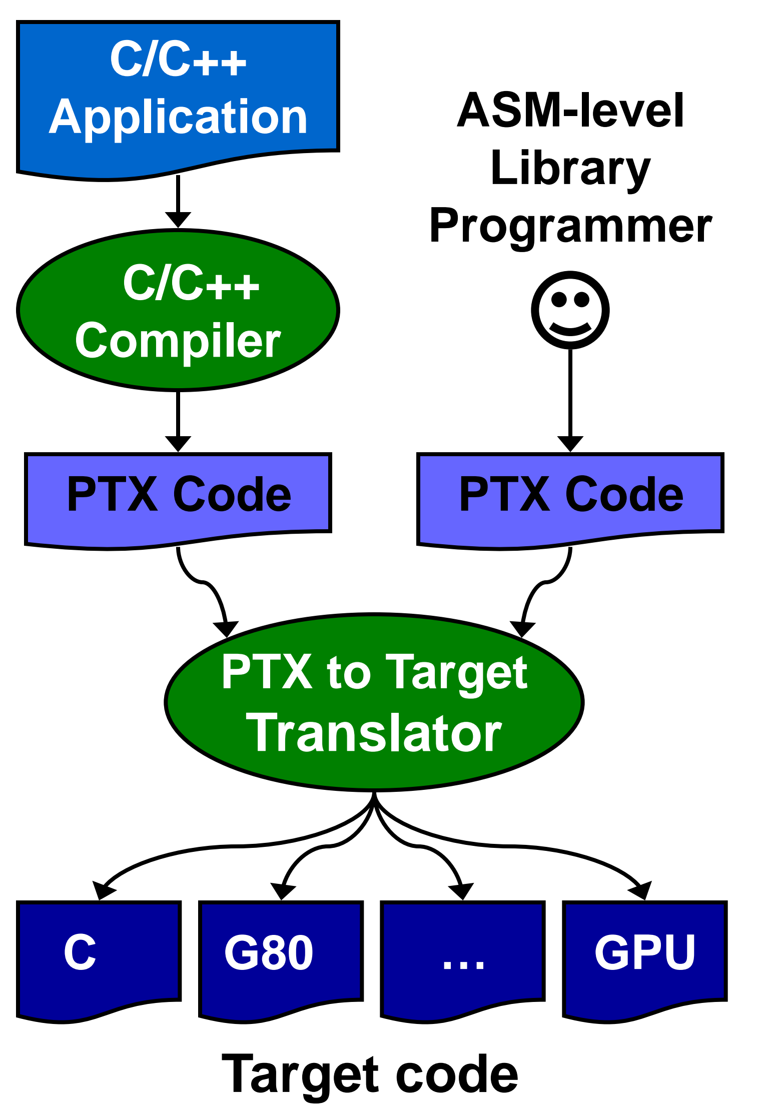
Shared Memory

- Each SM has 16 KB of Shared Memory
  - 16 banks of 32bit words
- CUDA uses Shared Memory as shared storage visible to all threads in a thread block
  - read and write access
- Not used explicitly for pixel shader programs
  - we dislike pixels talking to each other ☺

I$
L1
Multithreaded
Instruction Buffer
R
F
C$
L1
Shared
Mem
Operand Select
MAD
SFU

Sample CUDA / PTX Programs

PTX Virtual Machine and ISA



- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state
- ISA – Instruction Set Architecture
  - Variable declarations
  - Instructions and operands
- Translator is an optimizing compiler
  - Translates PTX to Target code
  - Program install time
- Driver implements VM runtime
  - Coupled with Translator

Compiling CUDA to PTX

CUDA

```
float4 me = gx[gtid];
me.x += me.y * me.z;
```

PTX

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
# 174       me.x += me.y * me.z;
mad.f32               $f1, $f5, $f3, $f1;
```

CUDA Function

- CUDA

```
__device__ void interaction(
            float4 b0, float4 b1, float3 *accel)
{
   r.x = b1.x - b0.x;
   r.y = b1.y - b0.y;
   r.z = b1.z - b0.z;
   float distSqr = r.x * r.x + r.y * r.y + r.z * r.z;
   float s = 1.0f/sqrt(distSqr);
   accel->x += r.x * s;
   accel->y += r.y * s;
   accel->z += r.z * s;
}
```

- PTX

```
sub.f32      $f18, $f1, $f15;
sub.f32      $f19, $f3, $f16;
sub.f32      $f20, $f5, $f17;
mul.f32      $f21, $f18, $f18;
mul.f32      $f22, $f19, $f19;
mul.f32      $f23, $f20, $f20;
add.f32      $f24, $f21, $f22;
add.f32      $f25, $f23, $f24;
rsqrt.f32    $f26, $f25;
mad.f32      $f13, $f18, $f26, $f13;
mov.f32      $f14, $f13;
mad.f32      $f11, $f19, $f26, $f11;
mov.f32      $f12, $f11;
mad.f32      $f9, $f20, $f26, $f9;
mov.f32      $f10, $f9;
```

Performance

Performance Variables

- Shorter programs are overhead limited
- Longer programs are instruction-rate limited
  - Must have enough threads per thread block – at least 192, more is better
  - Must have enough thread blocks – at least 32, more is better
- RF load balancing
  - RF space commonly in high demand

Performance Variables (2)

- Compiler quality important for good performance
  - Minimize register usage in CUDA programs
    - Reduces spilling to memory
  - Interleave non-dependent FP/DATA ops
    - maximizes issue rate
  - Cluster non-dependent texture and memory reads
    - decreases program latency

Summary

- G80 SPA is a revolutionary GPU architecture
  - CUDA – non-graphics programming interface
  - Shared Memory, Barriers
  - Unified, scalar
  - Excellent compiler target
  - Scalable (SP per SM, SM per TPC, TPC count)
  - Scalable programming model
- Did not do everything we wanted
  - Limit scope of changes in order to ship product
  - Gives us a chance to do another one ☺

Part I: GPU硬件和编程模型

- 从GPGPU到CUDA
- CUDA编程模型
- GPU存储器
- CUDA硬件模型
- **CUDA程序设计工具**
- 新一代Kepler GPU

Debuggers & Profilers

GPU Compilers

Parallelizing Compilers

Numerical Packages

- MATLAB
- Mathematica
- NI LabView
- pyCUDA

Debuggers & Profilers:

- cuda-gdb
- NV Visual Profiler
- Parallel Nsight
- Visual Studio
- Allinea
- TotalView

GPU Compilers:

- C
- C++
- Fortran
- OpenCL
- DirectCompute
- Java
- Python

Parallelizing Compilers:

- PGI Accelerator
- CAPS HMPP
- mCUDA
- OpenMP

Libraries

- BLAS
- FFT
- LAPACK
- NPP
- Video
- Imaging
- GPULib

GPGPU Consultants & Training

OEM Solution Providers

CUDA文档

http://docs.nvidia.com/cuda/index.html

DEVELOPER ZONE

CUDA TOOLKIT DOCUMENTATION

- CUDA Toolkit
  - CUDA Toolkit v5.0 Release Notes
- Getting Started Guides
  - CUDA Getting Started Guide for Linux
  - CUDA Getting Started Guide for Mac OS X
  - CUDA Getting Started Guide for Microsoft Windows
- Programming Guides
  - CUDA C Programming Guide
  - CUDA C Best Practices Guide
  - Kepler Compatibility Guide for CUDA Applications
  - Tuning CUDA Applications for Kepler
  - CUDA Dynamic Parallelism
  - PTX ISA Version 3.1
- Reference Manuals
  - CUDA Runtime API
  - CUDA Driver API
  - CUDA Math API
  - CUBLAS
  - CUFFT
  - CURAND
  - CUSPARSE
  - Thrust
- CUDA Samples
  - CUDA Samples
  - Release Notes
  - Guide to New Features

CUDA Getting Started Guide for Microsoft Windows
This guide discusses how to install and check for correct operation of the CUDA Development Tools on Microsoft Windows systems.

Programming Guides

CUDA C Programming Guide
This guide provides a detailed discussion of the CUDA programming model and programming interface. It then describes the hardware implementation, and provides guidance on how to achieve maximum performance. The Appendixes include a list of all CUDA-enabled devices, detailed description of all extensions to the C language, listings of supported mathematical functions, C++ features supported in host and device code, details on texture fetching, technical specifications of various devices, and concludes by introducing the low-level driver API.

CUDA C Best Practices Guide
This guide presents established parallelization and optimization techniques and explains coding metaphors and idioms that can greatly simplify programming for CUDA-capable GPU architectures. The intent is to provide guidelines for obtaining the best performance from NVIDIA GPUs using the CUDA Toolkit.

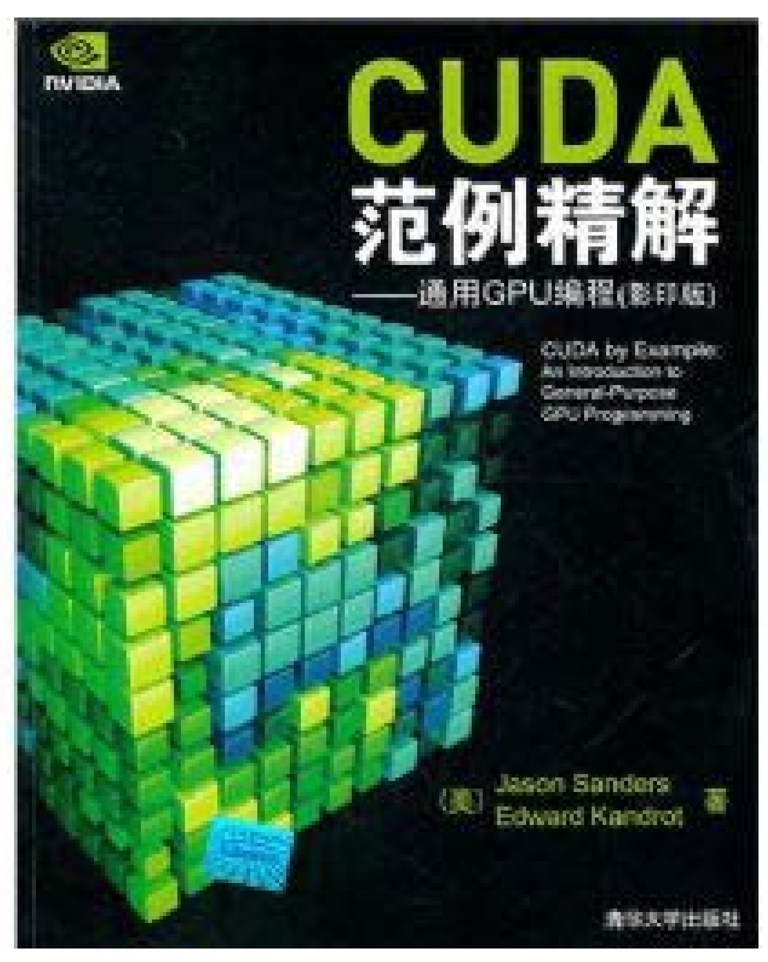
Kepler Compatibility Guide for CUDA Applications
This application note is intended to help developers ensure that their NVIDIA CUDA applications will run effectively on GPUs based on the NVIDIA Kepler Architecture. This document provides guidance to ensure that your software applications are compatible with Kepler.

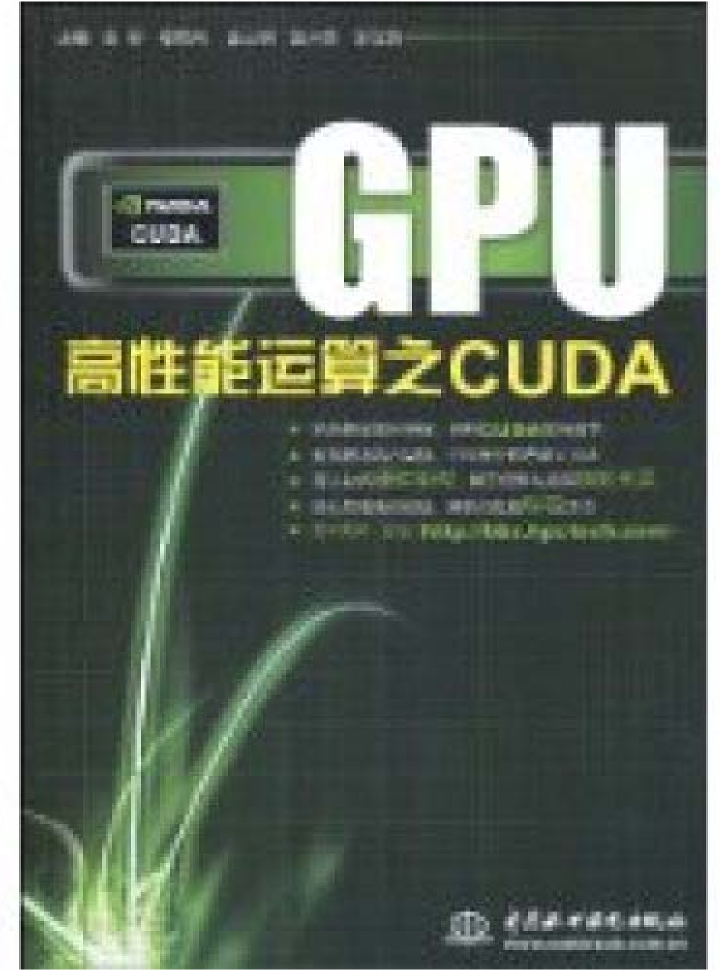
Tuning CUDA Applications for Kepler
Kepler is NVIDIA's next-generation architecture for CUDA compute applications. Applications that follow the best practices for the Fermi architecture should typically see speedups on the Kepler architecture without any code changes. This guide summarizes the ways that an application can be fine-tuned to gain additional speedups by leveraging Kepler architectural features.

CUDA Dynamic Parallelism
This document provides guidance on how to design and develop software that takes advantage of the new Dynamic Parallelism capabilities introduced with CUDA 5.0.

PTX ISA Version 3.1
This guide provides detailed instructions on the use of PTX, a low-level parallel thread execution virtual machine and instruction set architecture (ISA). PTX exposes the GPU as a data-parallel computing device.

Books on CUDA





集成开发环境parallel Nsight

http://www.nvidia.cn/object/parallel-nsight-cn.html

OpenACC: New Open Standard for GPU Computing

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

Faster, Easier, Portability

http://www.openacc-standard.org

The OpenACC™ API
QUICK REFERENCE GUIDE

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.

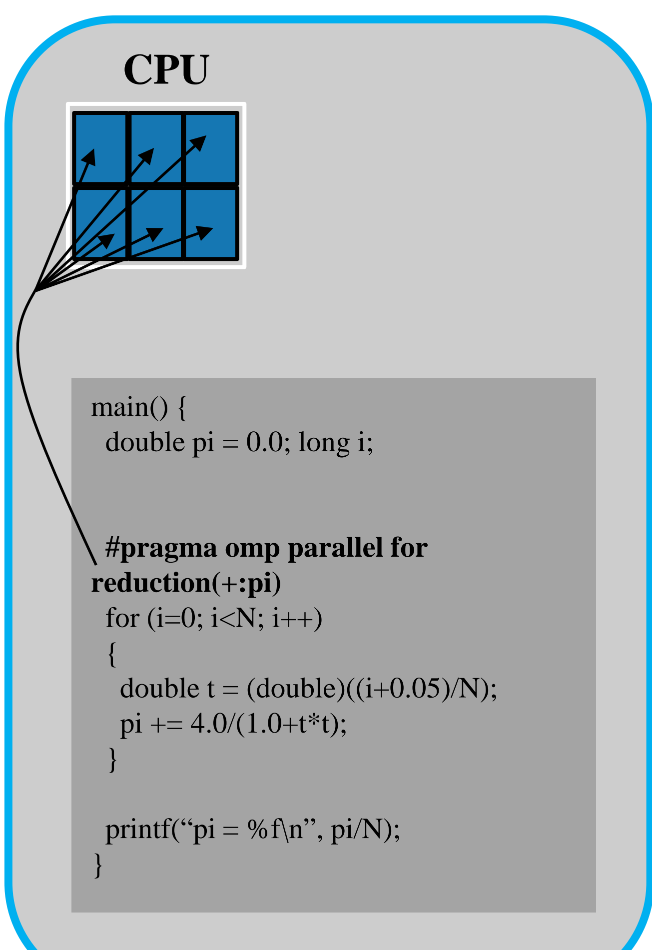
Version 1.0, November 2011

© 2011 OpenACC-standard.org all rights reserved.

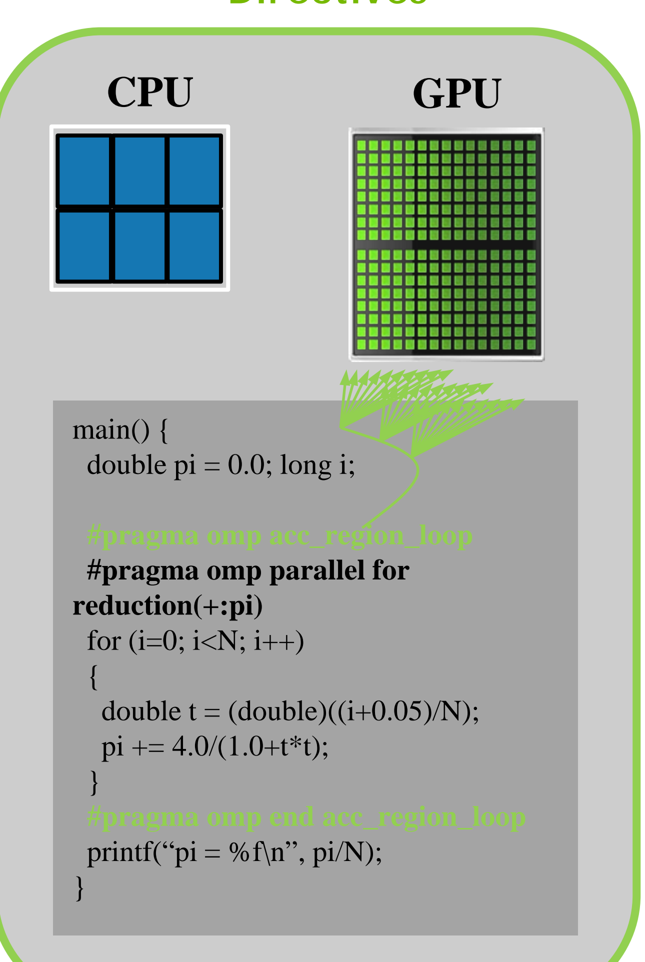
Directives: Add One Line of Code

中国科学院
计算机网络信息中心
Computer Network Information Center,
Chinese Academy of Sciences

OpenMP



Directives



Part I: GPU硬件和编程模型

- 从GPGPU到CUDA
- CUDA编程模型
- GPU存储器
- CUDA硬件模型
- CUDA程序设计工具
- **新一代Kepler GPU**

SMX: Efficient Performance

- Power-Aware SMX Architecture
- Clocks & Feature Size
- SMX result -

  Performance up

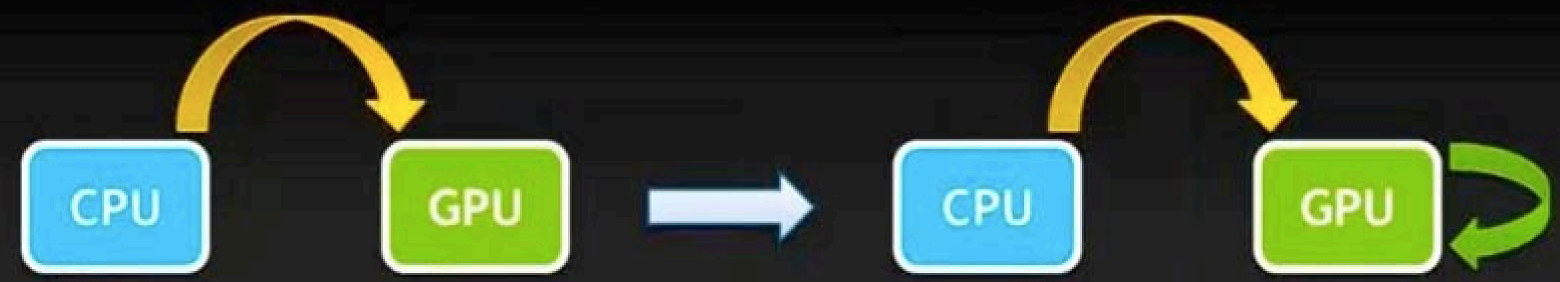
  Power down

Dynamic Parallelism

What is Dynamic Parallelism?

The ability to launch new kernels from the GPU

- Dynamically - based on run-time data
- Simultaneously - from multiple threads at once
- Independently - each thread can launch a different grid



Fermi: Only CPU can generate GPU work

Kepler: GPU can generate work for itself

Dynamic Parallelism

Familiar Syntax and Programming Model

```
int main() {
    float *data;
    setup(data);

    A <<< ... >>> (data);
    B <<< ... >>> (data);
    C <<< ... >>> (data);

    cudaDeviceSynchronize();
    return 0;
}
```

```
__global__ void B(float *data)
{
    do_stuff(data);

    X <<< ... >>> (data);
    Y <<< ... >>> (data);
    Z <<< ... >>> (data);
    cudaDeviceSynchronize();

    do_more_stuff(data);
}
```

CPU

main

GPU

A

B

X

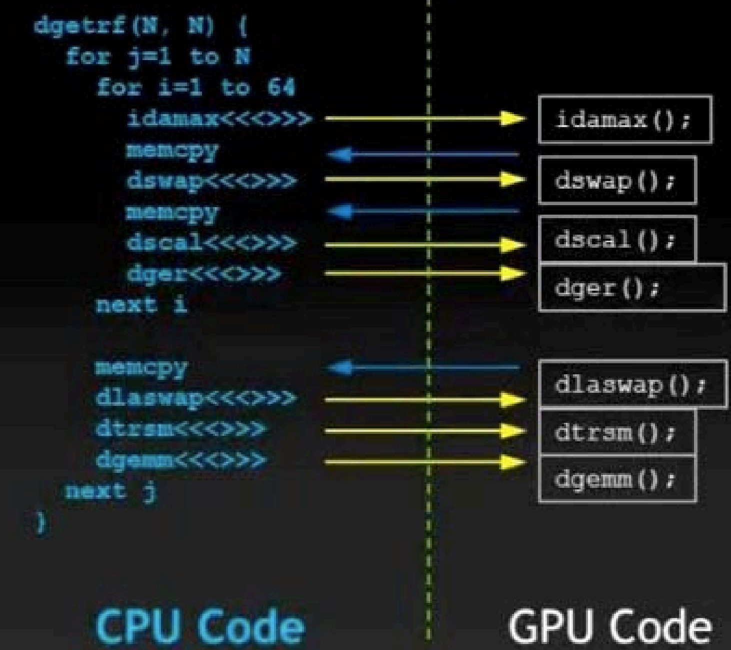
Y

Z

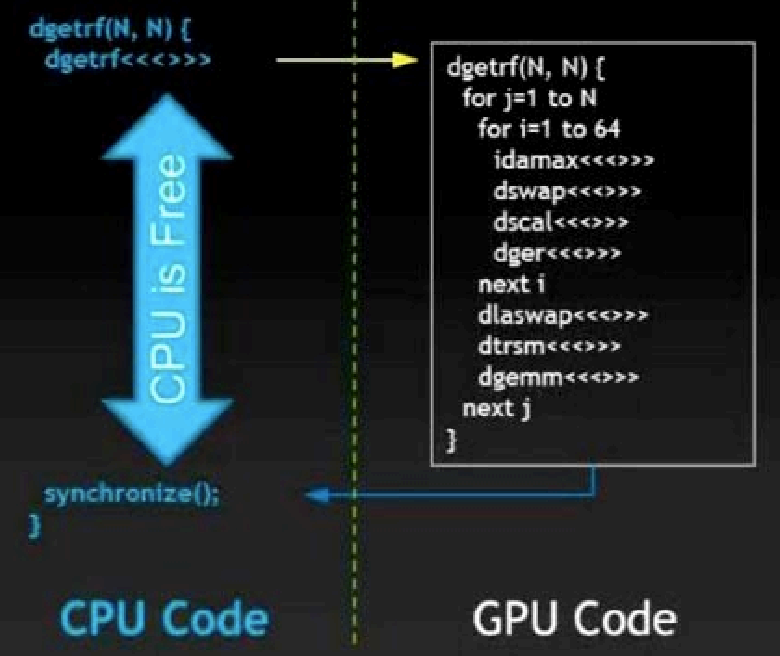
Dynamic Parallelism

Simpler Code: LU Example

LU decomposition (Fermi)



LU decomposition (Kepler)



Dynamic Parallelism

Dynamic Work Generation

Fixed Grid

Statically assign conservative worst-case grid

Initial Grid

Dynamically assign performance where accuracy is required

Dynamic Grid

Hyper-Q

Hyper-Q: Simultaneous Multiprocess

A B C D E F

CPU Processes

Shared GPU

Without Hyper-Q

GPU Utilization %

100

50

0

A B C D E F

Time

With Hyper-Q

GPU Utilization %

100
50
0

C B A | E D | A F | D C B | D C F E | B A | F E

Time

中国科学院
计算机网络信息中心
Computer Network Information Center,
Chinese Academy of Sciences

提纲

- 5分钟CUDA入门
- ***粒子物理与流体力学耦合系统***
- 流体GPU加速初探
- 第一原理计算一些体会

天文-粒子与流体耦合系统

- 按现有理论，宇宙中物质是重子物质（流体模型）和无碰撞的暗物质（粒子模型）相互作用的耦合系统。
- 在宇宙学流体/N体模拟中重子物质可视为无粘可压缩流体，求解方法一般有欧拉网格方法，SPH粒子方法。
- 暗物质通常用N粒-体子模拟。

天文-流体模型

- 宇宙学流体欧拉方程式：

$$\frac{\partial U}{\partial t}+\frac{\partial F(U)}{\partial x}+\frac{\partial G(U)}{\partial y}+\frac{\partial H(U)}{\partial z}=S$$

$$\begin{pmatrix}\rho\\ \rho u\\ \rho v\\ \rho w\\ E\end{pmatrix},\quad \begin{pmatrix}\rho u\\ \rho u^2+P\\ \rho uv\\ \rho uw\\ u(E+P)\end{pmatrix},\quad \begin{pmatrix}\rho v\\ \rho uv\\ \rho v^2+P\\ \rho vw\\ v(E+P)\end{pmatrix},\quad \begin{pmatrix}\rho w\\ \rho uw\\ \rho vw\\ \rho w^2+P\\ w(E+P)\end{pmatrix}.$$

天文-粒子模型

- 暗物质粒子方程:

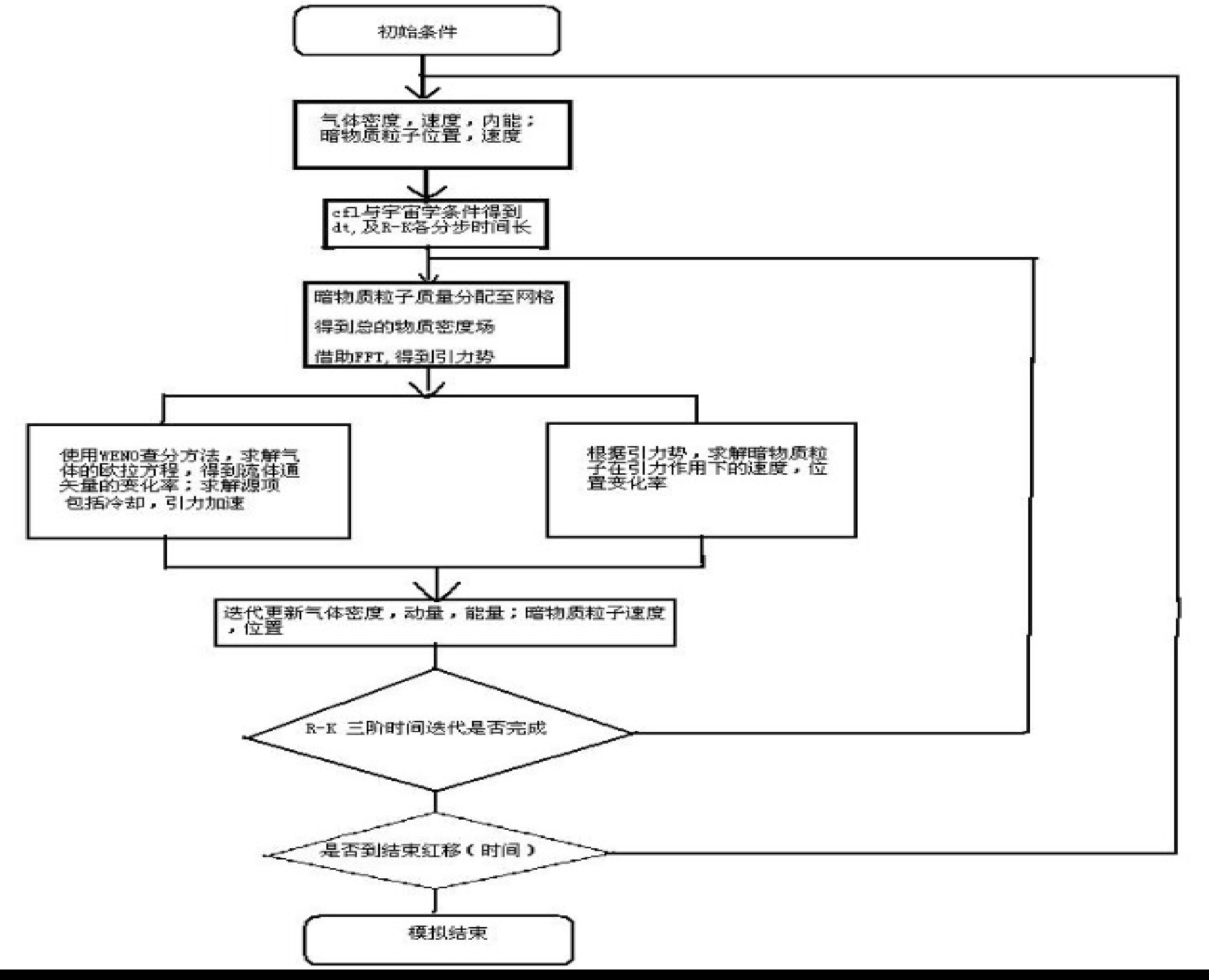
$$\frac{d\mathbf{r}_{\mathrm{DM}}}{dt} = \frac{1}{a}\mathbf{v}_{\mathrm{DM}}$$
$$\frac{d\mathbf{v}_{\mathrm{DM}}}{dt} = -\frac{\dot{a}}{a}\mathbf{v}_{\mathrm{DM}} + \mathbf{G}$$

- 引力势方程:

$$\nabla^2\phi(r,t) = 4\pi G[\rho_{tot}(r,t) - \rho_0(t)]/a$$

- FFT

$$\hat{\phi}(\vec{k},t) = -4\pi G a^2 \frac{\hat{\rho}(\vec{k},t)}{k^2}$$


初始条件
气体密度，速度，内能；
暗物质粒子位置，速度
cfl与宇宙学条件得到
dt, 及R-K各分步时间长
暗物质粒子质量分配至网格
得到总的物质密度场
借助FFT, 得到引力势
使用WENO差分方法，求解气
体的欧拉方程，得到随体通
矢量的变化率；求解源项
包括冷却，引力加速
根据引力势，求解暗物质粒
子在引力作用下的速度，位
置变化率
迭代更新气体密度，动量，能量；暗物质粒子速度
，位置
R-K 三阶时间迭代是否完成
是否到结束红移（时间）
模拟结束

Wigeon简介

- WIGEON(Feng et al. 2004)使用三维五阶WENO空间有限差分方法求解重子物质的欧拉方程，时间差分为三阶Runge-Kutta，N-体粒子模拟暗物质，Particle Mesh方法求解引力。
- 使用Particle Mesh求解N-体问题，与基于网格的weno相结合，计算最基本单元为网格中的各个点，实现了重子与暗物质耦合计算。

粒子算法-Particle-Mesh

粒子插值到网格

FFT

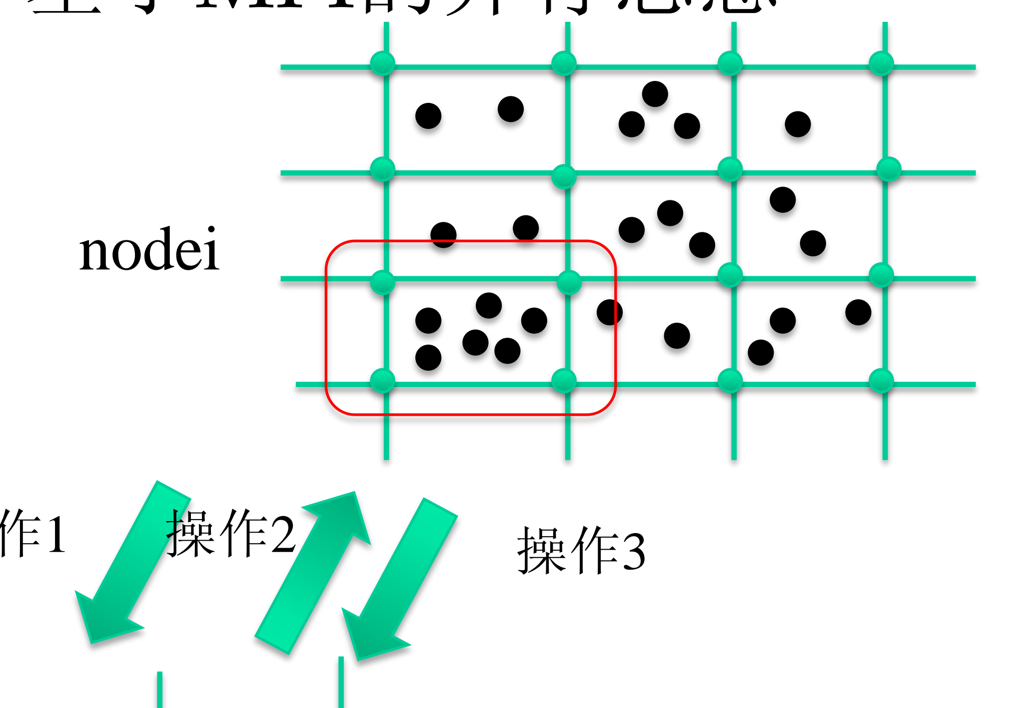
格点反作用到粒子

粒子并行策略（Particle-Mesh）

- 并行计算难点
  - 粒子的分布高度无序性
  - 粒子动态迁移，随时而变
  - 每个粒子在更新状态时既需要本地数据，也需要全局数据
  - 粒子在某些时刻存在严重的聚团效应
    - 内存问题
    - 负载平衡挑战

粒子并行策略（PM）

- 基于MPI的并行思想



以网格单元为单位组织粒子。粒子数目超过一定数量的粒子，将本地的一部分粒子以及这些粒子所贡献密度的部分网格点发送到空闲节点上，二者进行协作，通过通信完成计算。

操作一：发送粒子的速度、位置、临近网格点；
操作二：返回网格点的密度值
操作三：发送下一时间步的网格点的引力

提纲

- 5分钟CUDA入门
- 粒子物理与流体力学耦合系统
- ***流体GPU加速初探***
- 第一原理计算一些体会

天文流体模型-WENO

- 数学模型

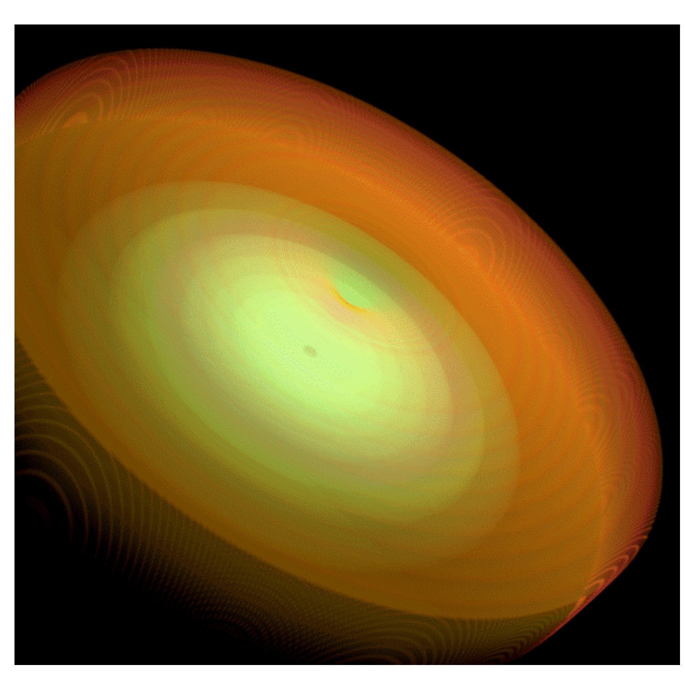
$$F(U)_{x,x=x_i} \simeq \frac{\hat{F}_{i+1/2} - \hat{F}_{i-1/2}}{\Delta x} \qquad \hat{F}_{i+1/2} = w_1 \hat{F}^{(1)}_{i+1/2} + w_2 \hat{F}^{(2)}_{i+1/2} + w_3 \hat{F}^{(3)}_{i+1/2}$$

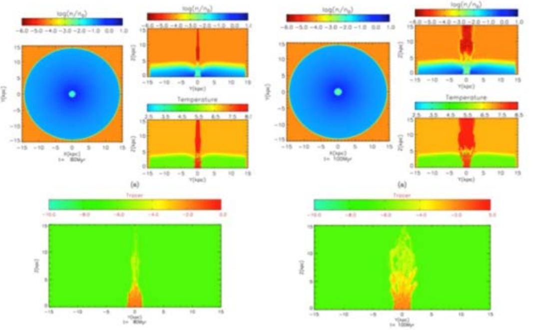
$$\hat{F}^{(1)}_{i+1/2} = \frac{1}{3} F(U_{i-2}) - \frac{7}{6} F(U_{i-1}) + \frac{11}{6} F(U_i)$$

$$\hat{F}^{(2)}_{i+1/2} = -\frac{1}{6} F(U_{i-1}) + \frac{5}{6} F(U_i) + \frac{1}{3} F(U_{j+1})$$

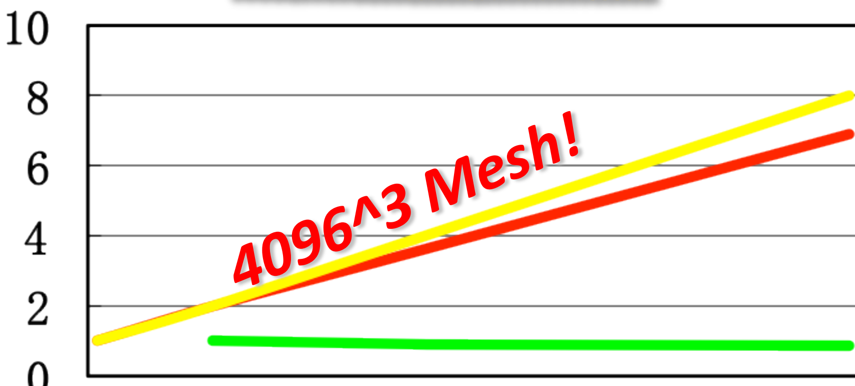
$$\hat{F}^{(3)}_{i+1/2} = \frac{1}{3} F(U_j) + \frac{5}{6} F(U_{j+1}) - \frac{1}{6} F(U_{j+2})$$

万核级天文模拟软件：P-Wigeon





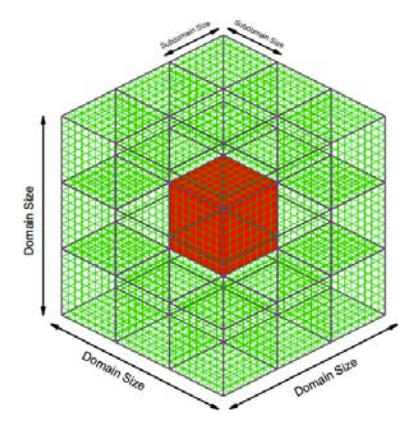
软件计算规模与速度
已达国际先进水平
紫台 冯珑珑研究员

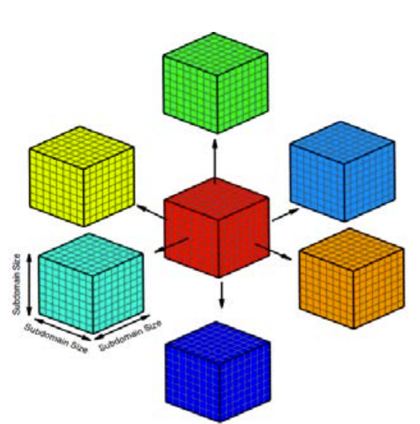


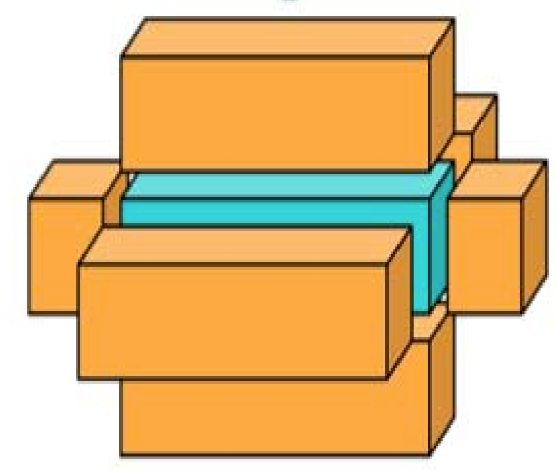
	2048	4096	8192	16384
Speedup	1	1.996	3.62	6.9
Linear	1	2	4	8
Efficiency		99.79%	90.54%	86.20%

WENO格式的数据依赖

- 计算特点
  - 分别计算三维空间中x、y、z三个方向的五阶有限差分，Ghost Cell单方向为5*2=10；
  - 差分阶数越强，每个方向的数据相关性越强；
  - MPI并行化时将数据切割成若干数据块，每个块内部必须尽可能大，否则存在一个与差分阶数相当厚度的外壳，如图所示。







WENO函数GPU加速

- GPU并行化策略
  - 以x方向的计算为例

Bsize.x

z

Block

Bsize.y

One Thread

y

WENO函数GPU加速

- 数据结构

Z

y

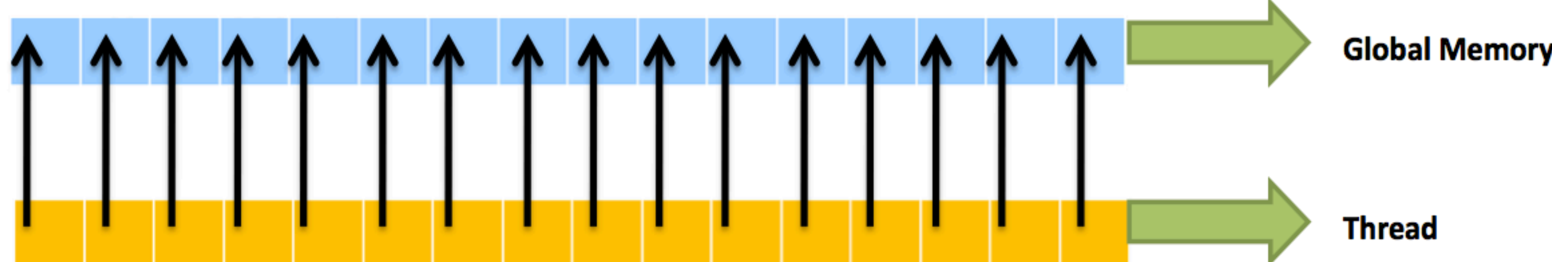
(x0,y0,z0)

$$\begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{pmatrix}$$

x

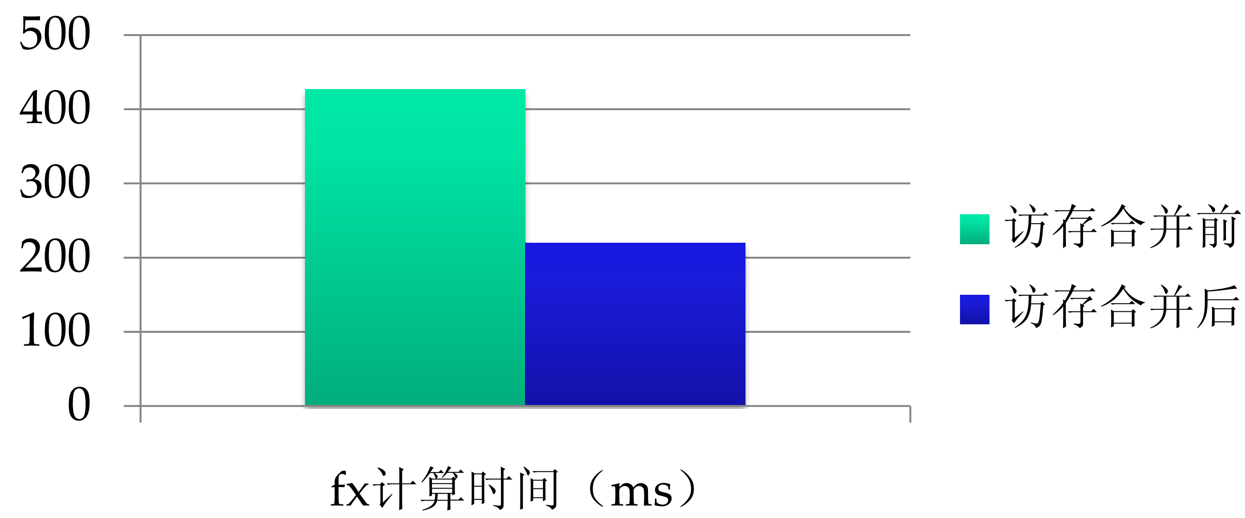
WENO函数GPU加速

- 优化一
  - 优化访存模式——合并访问：连续索引号的thread访问连续的GPU内存效率更高。
    - 三维的五元素向量数据块-->五个连续存储的单元素三维数据块，AOS->SOA
    - 交换x、y方向数据，确保x方向计算访存合并
    - 对于大数据计算，在数据局部性较好的情况下，Global读写效率良好。



WENO函数GPU加速

- 优化一
  - 优化访存模式——连续索引号的thread访问连续的GPU内存效率更高。
  - 使用前后对比

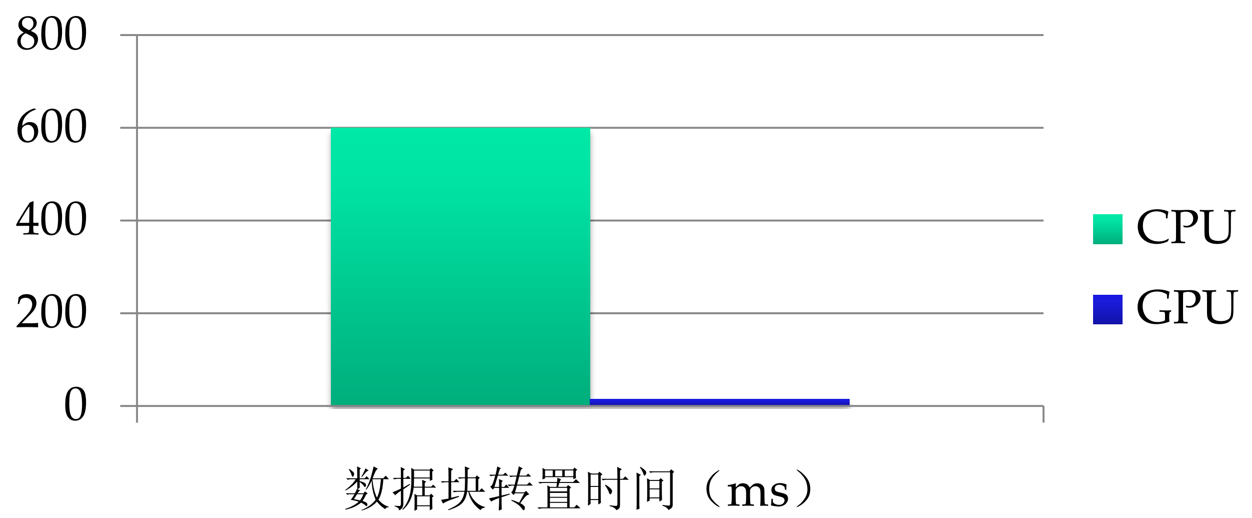


WENO函数GPU加速

- 优化二
  - 把数据块的变换处理移植到GPU上，最大化并行；
    - 数据块的变换本质上就是一个四维矩阵的坐标轴转置，并行性很好；
    - 使用shared memory作为桥梁，充分利用访存合并的优势；
    - AOS->SOA数据格式转化：30～40倍的加速比；

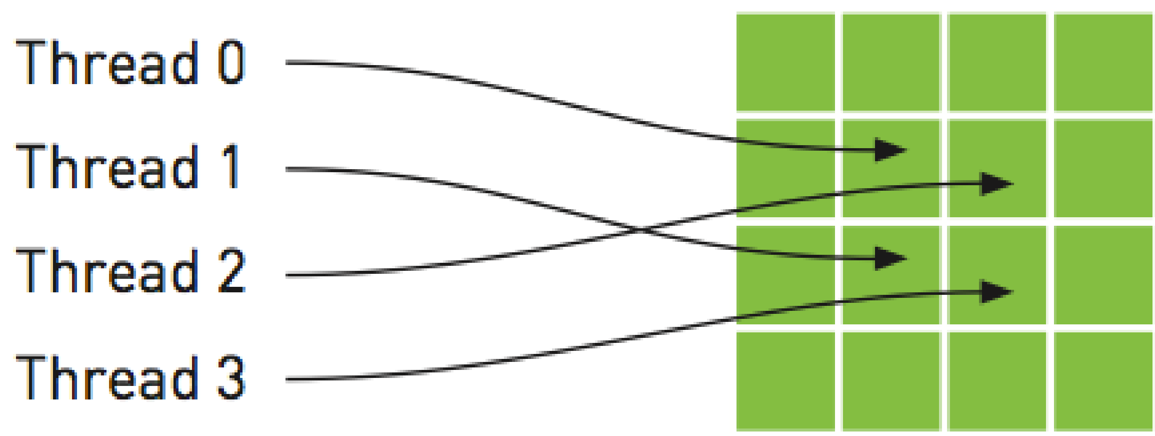
WENO函数GPU加速

- 优化二
  - 把数据块的变换处理移植到GPU上，最大化并行；



WENO函数GPU加速

- 优化三
  - 使用Texture Memory加速访存只读数据；
    - Texture Memory支持二维、三维空间的访存局部性，解决了三维数据必然有一个方向的计算无法达到访存合并的问题；



WENO函数GPU加速

- 优化三
  - 使用Texture Memory加速访存只读数据；
    - Texture Memory强制内存对齐，解决高阶差分的外壳数据导致的内存不对齐问题；
    - Texture Memory拥有独立于Glocal Memory的cache，加速Global Memory的读取且无附加代价。

WENO函数GPU加速

- 优化三
  - 使用Texture Memory加速访存只读数据;

    //声明一个2D的Texture Mem

    - texture<int2,2> texro;

    //将某个在kernel开始后不再修改的内存绑定到此Texture Mem

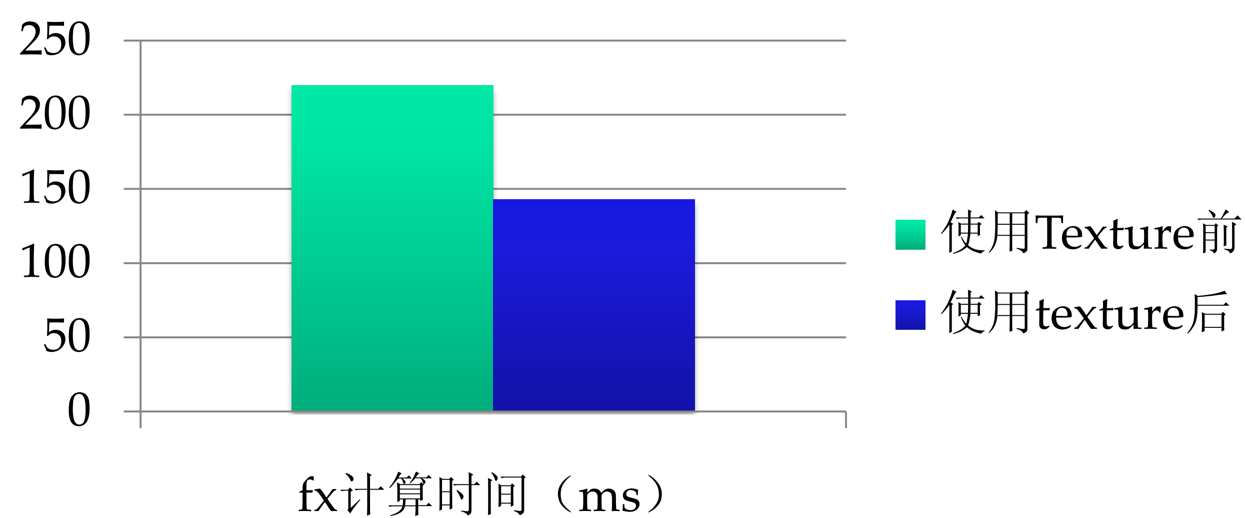
    - cudaBindTexture2D(&offset,texrPtr,d_ucnew,&channelDesc,width,height,pitch);

    //通过纹理机制读取数据，转化成需要的类型

    - int2 data=tex2D(texro,i,j);
    - value＝ __hiloint2double(data.y,data.x);

WENO函数GPU加速

- 优化三
  - 使用Texture Memory加速访存只读数据；
  - 使用前后对比



WENO函数GPU加速

- 优化四
  - 短数组变多个单独变量
    - GPU kernel中申请的数组，系统默认将其放在Glocal Memory中；
    - Glocal Memory读写取需要200～300 cycles；
    - 单个变量存储在register中，读写只需1cycles；
    - 需要注意的是，Register的使用过多，同样会导致访存延迟的增大；

WENO函数GPU加速

- 优化五
  - 精简分支判断
  - 部分for循环手动展开
- 优化六
  - 选择最优的BlockSize
    - 充分利用register、share Memoroy等资源
    - GPU的并行优势在于thread的无延迟切换，大小合适的BlockSize可以使得thread轻松切换，全速运行。
    - 可通过多次试运行获得最佳值

WENO函数GPU加速

- 优化七
  - 选择合适的编译参数
    - -use_fast_math
    - -maxrregcount
    - -Xptxas –dlcm=cg or ca
    - 等

WENO函数GPU加速

16
14
12
10
8
6
4
2
0

1 2 3 4 5 6 7

CPU
GPU

1、移植到GPU
2、调解指令顺序
 打开小循环
 精简分支
3、数据块转置
 Global访存合并
4、使用share
5、使用texture
6、选择blocksize
 削减register
7、选择合适的编
 译选项

WENO函数GPU加速

- 结果分析
  - 从优化曲线中可以看到，在使用了share memory后，性能急速下降，由于之前考虑到大数据计算对Global访存频率较高，因此使用share memory来做缓存，为了找到性能下降的原因，我们使用了cuda profiler工具。

Share Mem	Active warp/cycle	Glob throughput
使用前	7.8	103.9
使用后	0.99	5.44

  - 由于share mem的资源有限，因此在使用了大量的share mem后，同一时间处于活动状态的thread变少，大量的访存需求都在等待，share mem起到的缓存作用有限，而活动thread数目的减少影响了计算对访存延迟的掩盖。

WENO函数GPU加速

- 最终结果
  - Weno计算部分整体达到15加速比（初步）

20
15
10
5
0

Fx Gy Hz Kernel weno

CPU时间
GPU时间
SpeedUp

WENO函数GPU加速

- 存在的问题
  - 个别数组的访存不合并
  - 由于数据的线性存储，只使用了Texture的2D模式，Texture在3D空间中的优势未利用；
  - 拷备与计算未重叠执行；
- 可能的进一步优化
  - 修改数据结构，充分利用Texture3D
  - 使用muti-stream实现内存拷贝和计算的重叠执行
  - 使用muti-GPU线性扩充加速比

提纲

- 5分钟CUDA入门
- 粒子物理与流体力学耦合系统
- 流体GPU加速初探
- ***第一原理计算一些体会***

Density Function Theory (DFT) calculations is widely used

A survery of computational material science algorithm in NERSC community (2007)

3.10% 2.90%
6.40%
6.70%
6.90%
74%

- DFT
- Beyond DFT
- QMC
- CMD
- CMC
- PDE

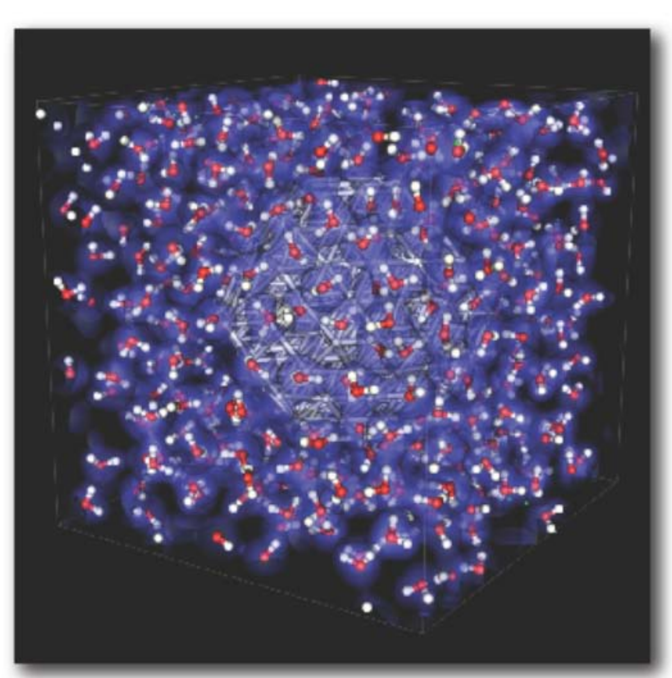
Challenges for DFT calculations

- 100 to 1000 atoms
- *ab initio* MD for a few ns
- search for structures

State-of-the-art: 1-2 min per MD step(so can only calculate a few ps, But want: ns!)

For >>1000 atoms, linear scaling method (divide and conquer)

Nanocatalysis: Pt



Molecular dynamics
Pt_{201}+427H_2O 1482 atoms

P. Kent, ORNL
M. Neurock, U. Virginia

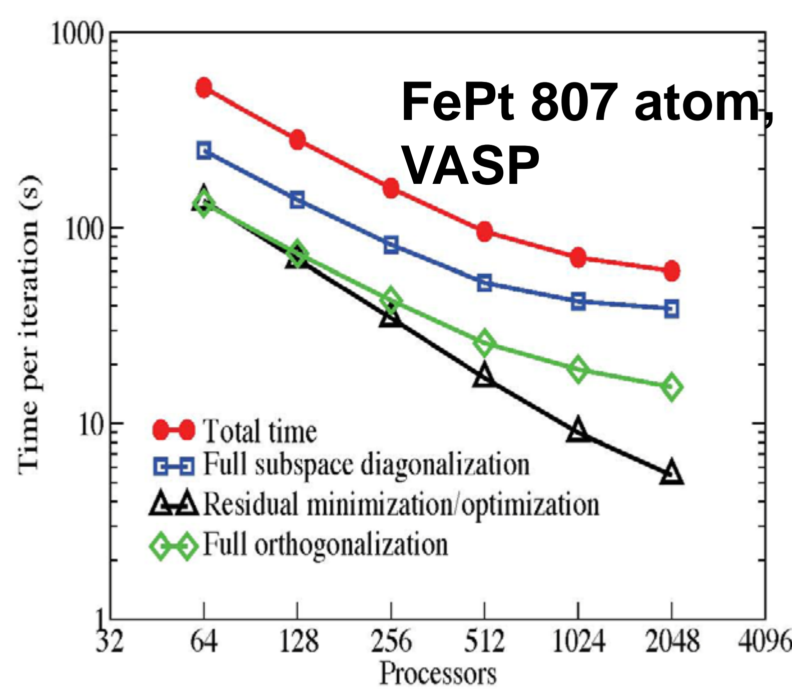
PWP-DFT codes

- **Most mature, and widely used**
- **Dozens of them:**

  VASP, CASTEP, CPMD, ABINIT, PWSCF, DACAPO, SOCORRO, DFT++, PARATEC, DOD-PW, CP2K, SPHINX, QBOX, PEtot

- **CPU codes do not scale > 1000 cores**
- **1 or 2 minutes per MD step**

P. Kent, ORNL



Idea: use GPU to speed up the absolute speed !!!

PEtot code

- Developed in Lawrence Berkeley National Lab
- Free: https//hpcrd.lbl.gov/~linwang/PEtot/PEtot.html
- 3 levels parallelization: G-space, state index, k-point
- norm conserving pseudopotential and ultra-soft psd.
- parallel FFT (by Andrew Canning)
- Can calculate 10,000 states on a few thousand cores

PEtot code

What is PEtot ?

PEtot stands for parallel total Energy (Etot). It is a parallel plane wave pseudopotential program for atomistic total energy calculation based on density functional theory. It is designed for large system simulations to be run on large parallel computers like IBM SP machines at NERSC, and linux cluster machines. It is developed under U.S. Department of Energy fundings and it is a freely distributed public source code. It has a LBNL BSD license, which means that you can use it and change it for noncommercial purposes. However, we will not be responsible for any potential problems it might cause directly or indirectly due to the running of this code.

There are two versions of the code. In the lastest version3, there are three levels of parallelizations (G-vector, band index, k-points), and there are all-band algorithms. We urge the users to use the new version3 of this code, which has more features and is faster, especially for large systems. The explanation on this web page is for version2. The documentation for the newest version3 is in its tar file.

More detail documentations for different versions are in the tar.gz file.

Download the PEtot Package (source files)

Double click to download PAR.ETOT_version1.tar.gz (0.5MB, the first version)

Double click to download PAR.ETOT_version2.tar.gz (4MB, the second version)

Large scale calculations using PEtot

Related work

- VASP on GPU:

  Speedup the computational-intensive part

  4—7x of speedup

- PWscf:

  Part of PWSCF speedup

  2—4x of speedup

DFT calculation is time-consuming

$$[-\frac{1}{2}\nabla^2 + V_{tot}(r)]\psi_i(r) = \varepsilon_i \psi_i(r)$$

- **If the size of the system is *N*:** $\psi_i(r)$
- ***N* coefficients to describe one wavefunction**
- ***i* = 1,…, *M* wavefunctions** $\psi_i(r)$**, *M* is proportional to *N*.**
- **Orthogonalization:** $\int \psi_i(r)\psi_j^*(r)d^3r$ **, *M*^{2} wave function pairs, each with *N* coefficients: *N**M*^{2}, i.e *N*^{3} scaling.**

The repeated calculation of these orthogonal wave functions make the computation expensive, *O(N*^{3}*).*

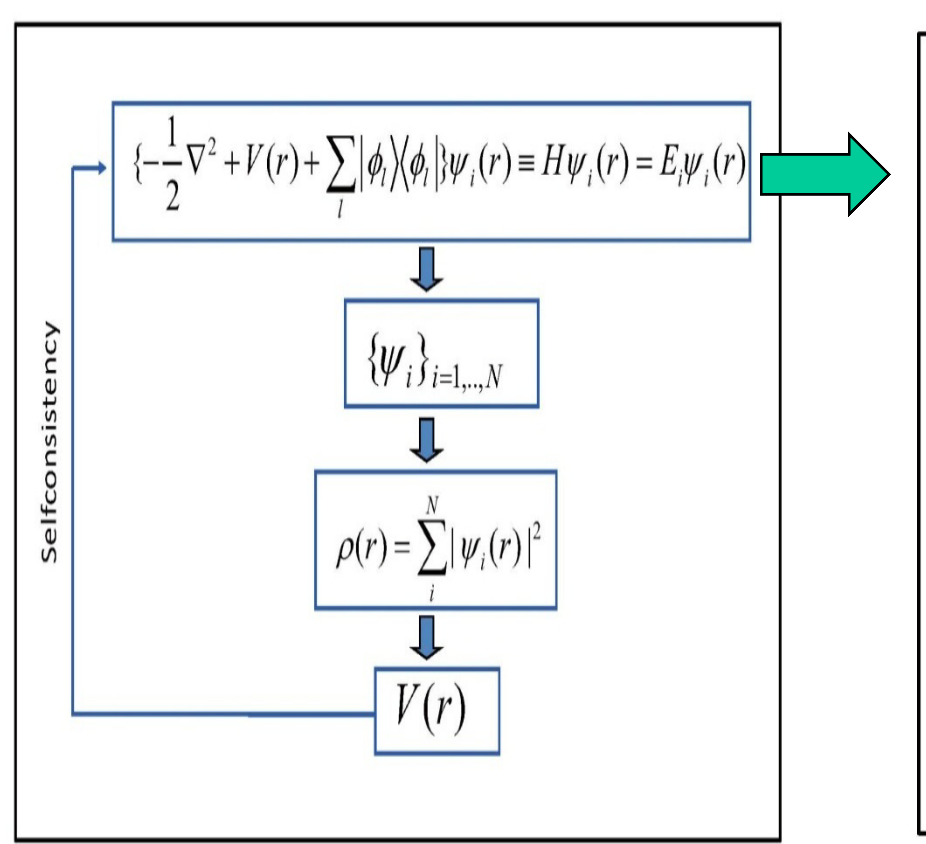
PWP-DFT on GPU

- Optimal CPU parallelization scheme has been worked out in past 10-15 years
- But the same scheme might not be optimal for GPU
- It might be necessary to redesign the scheme, instead of following the old scheme

The PWP-DFT calculation flow chart

中国科学院
计算机网络信息中心
Computer Network Information Center,
Chinese Academy of Sciences

The overall flow chart of SCF iterations



The conjugate-gradient (CG) to solve the Schrodinger’s eq.

$h(i,j) = \langle\psi_i|H|\psi_j\rangle$ Sub_diag, *

$P_i = H\psi_i - \varepsilon_i\psi_i$ Hpsi, *

$P_i = A(P_i - \frac{\lambda_i}{\lambda_i^o} P_i^o)$ Precond. CG step

$P_i = P_i - \sum_{j=1,i} \langle P_i|\psi_j\rangle$ Projection, *

$\psi_i = \psi_i \cos\theta_i + P_i \sin\theta_i$ Line minimiz.

$\psi_i = \psi_i - \sum_{j<i} \langle\psi_i|\psi_j\rangle$ Orth., *

$h(i,j) = \langle\psi_i|H|\psi_j\rangle$ Sub_diag, *

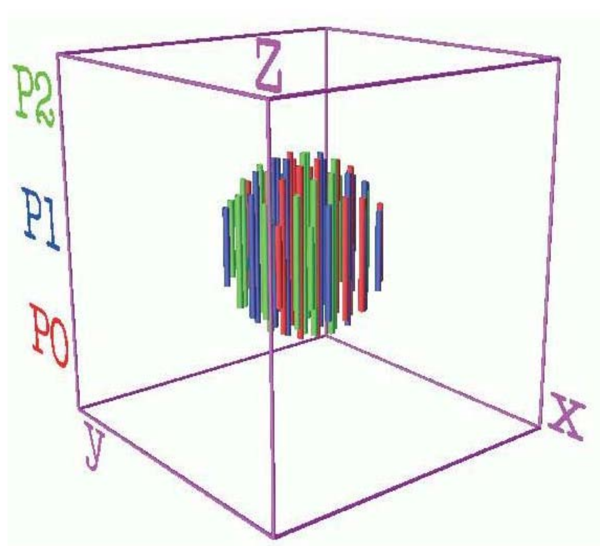
nline

The kernels in the H*ψ (Hpsi) (CPU)

中国科学院
计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

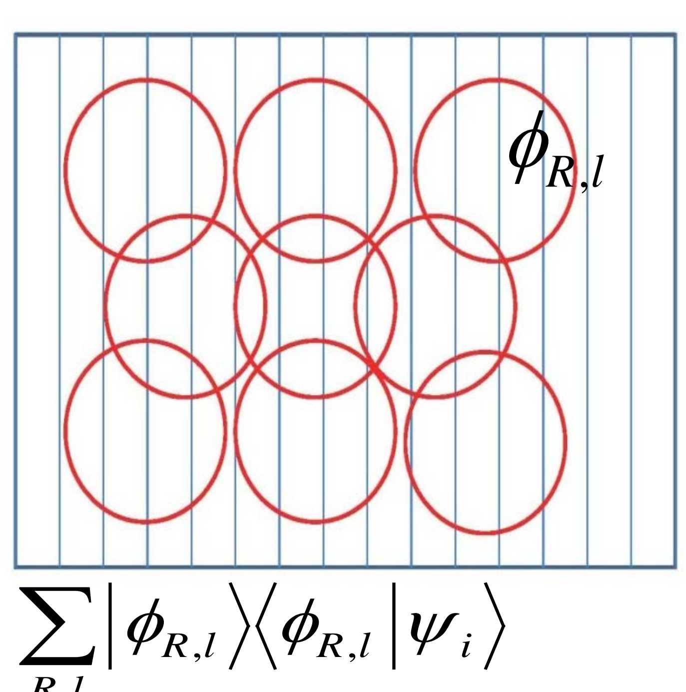
$$\{-\frac{1}{2}\nabla^2 + V(r) + \sum_l |\phi_l\rangle\langle\phi_l|\}\psi_i(r)$$

FFT (by A. Canning)



FFT takes about 20% time in PWP-DFT !

Real sace
Nonlocal pseudopotential



$$\sum_{R,l} |\phi_{R,l}\rangle\langle\phi_{R,l}|\psi_i\rangle$$

CPU parallelization scheme

3D division of processors $P_{j,g}$

$$\psi_{i,k}(r) = \sum_G C_{i,k}(G)\exp(-i(G+k)\bullet r)$$

k_1

P_{00}	P_{01}	P_{02}	P_{03}	G_1
P_{10}	P_{11}	P_{12}	P_{13}	G_2
P_{20}	P_{21}	P_{22}	P_{23}	G_3
P_{30}	P_{31}	P_{32}	P_{33}	G_4
Ψ_1	Ψ_2	Ψ_3	Ψ_4	

....

k_n

P_{00}	P_{01}	P_{02}	P_{03}	G_1
P_{10}	P_{11}	P_{12}	P_{13}	G_2
P_{20}	P_{21}	P_{22}	P_{23}	G_3
P_{30}	P_{31}	P_{32}	P_{33}	G_4
Ψ_1	Ψ_2	Ψ_3	Ψ_4	

Parallel FFT

(each CPU has many 1D FFTs)

P2 P1 P0 Z x y

G_1, G_2, G_3 (G-space)

P2 P1 P0 Z x y

Real space

But it does not work for GPU

- Parallel FFT is too fragmented to be scalable
- Nonlocal projectors have been fragmented on each core
- In general data chunk is too small
  (cannot fully realize the power of GPU, and CPU-GPU data copy takes time).

We need parallelization schemes with larger chunk of data

Hybrid parallelization scheme for GPU

G-parallel

G_0	P_0
	.
	.
G_{14}	P_{14}
G_{15}	P_{15}

$\{\Psi_i\}$

Diag rotation $\langle \psi_i | \psi_j \rangle$

CUBLAS
MPI_allreduce

Wave function transpose

MPI_alltoall

Index parallel

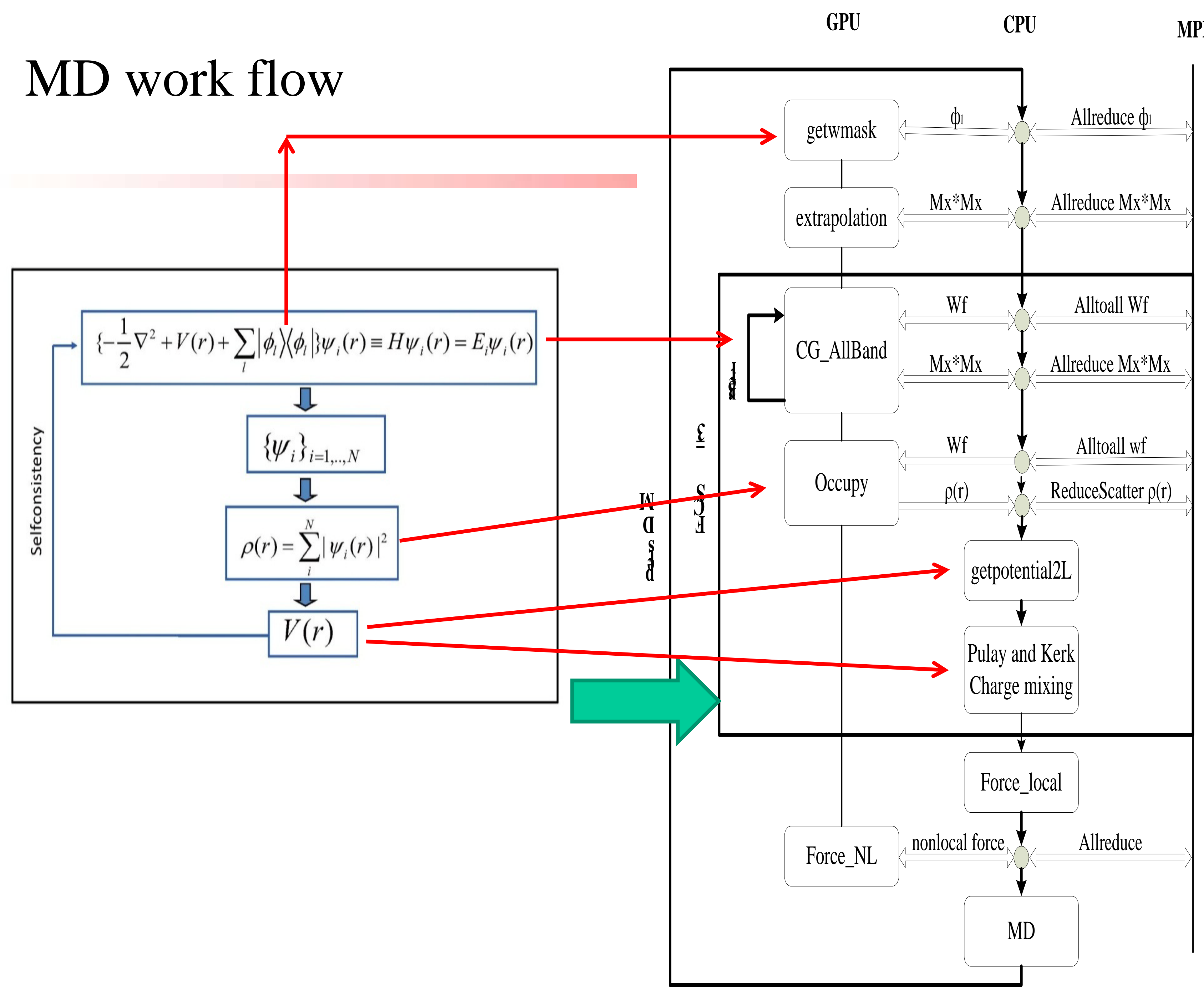
P_0 . . . . P_{14} P_{15} $\{G\}$

Ψ_0 Ψ_{14} Ψ_{15}

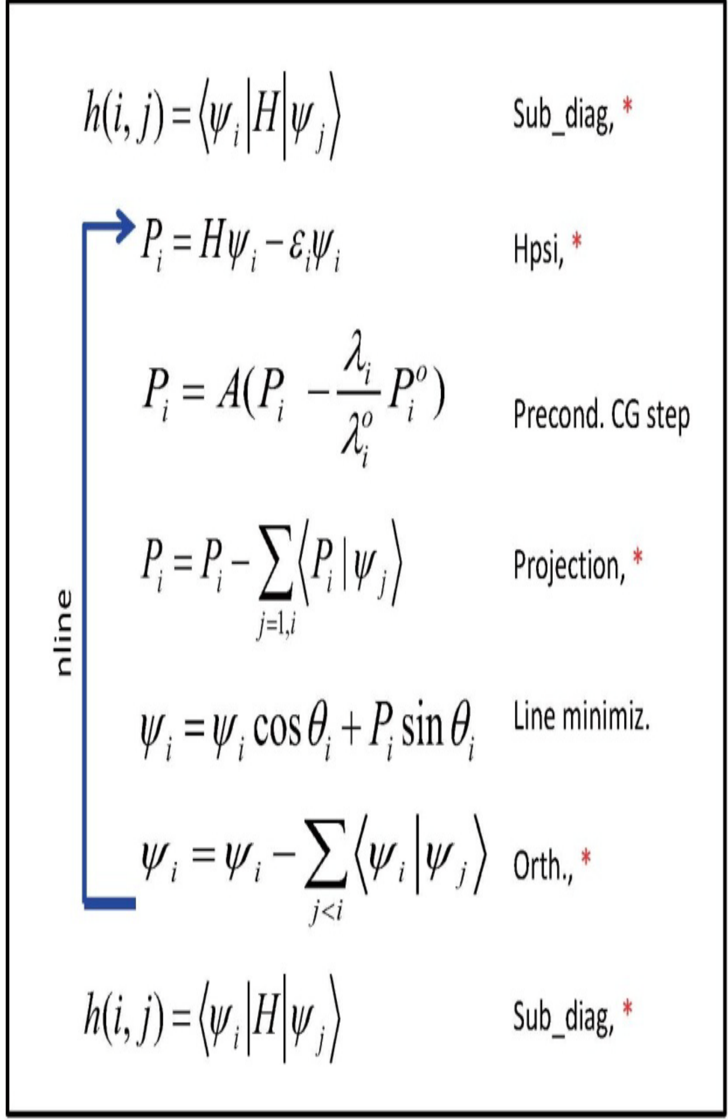
Hpsi
FFT
nonlocal

CUFFT

- The FFT is within a single GPU (no multi-GPU FFT)
- memory limitation to the size: a few thousand atoms


MD work flow
GPU
CPU
MPI
getwmask
ϕl
Allreduce ϕl
extrapolation
Mx*Mx
Allreduce Mx*Mx
CG_AllBand
Wf
Alltoall Wf
Mx*Mx
Allreduce Mx*Mx
Occupy
Wf
Alltoall wf
ρ(r)
ReduceScatter ρ(r)
getpotential2L
Pulay and Kerk
Charge mixing
Force_local
Force_NL
nonlocal force
Allreduce
MD
Selfconsistency
$\{-\frac{1}{2}\nabla^2 + V(r) + \sum_l |\phi_l\rangle\langle\phi_l|\}\psi_i(r) \equiv H\psi_i(r) = E_i\psi_i(r)$
$\{\psi_i\}_{i=1,..,N}$
$\rho(r) = \sum_i^N |\psi_i(r)|^2$
$V(r)$

CG_Allband Kernel



GPU

CPU

MPI

Wave function input

MPI_Alltoall(wf)

*

$H\Psi_i$: FFT + Nonlocal

Memcpy(wf)

Memcpy(wf)

Memcpy(wf)

MPI_Alltoall(wf)

*

Sub_diag: $<\Psi_i|H|\Psi_j>$

Memcpy(mx)

MPI_Allreduce(mx)

dsyev

Memcpy(mx)

iteration = 3

$P_i = A(H\Psi_i - \varepsilon_i\Psi_i)$

Proj: $P_i = P_i - \sum_j <P_i|\Psi_j>\Psi_j$

Memcpy(mx)

MPI_Allreduce(mx)

Memcpy(mx)

HP_i: FFT + Nonlocal

Memcpy(P)

MPI_Alltoall(P)

*

Memcpy(P)

Memcpy(P)

MPI_Alltoall(P)

*

Memcpy(P)

$\Psi_i = \Psi_i\cos\theta_i + P_i\sin\theta_i$

Orth: $\Psi_i = \Psi_i - \sum_j <\Psi_i|\Psi_j>\Psi_j$

Memcpy(mx)

MPI_Allreduce(mx)

Memcpy(mx)

Sub_diag: $<\Psi_i|H|\Psi_j>$

Memcpy(mx)

MPI_Allreduce(mx)

dsyev

Memcpy(mx)

Data compression on residual P

Error of Lowest Eigenstate (CPU)
Error of Lowest Eigenstate (GPU)
Error of Lowest Eigenstate (ELPA)
Maximum Error of 1024 Eigensates (CPU)
Maximum Error of 1024 Eigenstates (GPU)
Maximum Error of 1024 Eigenstates (ELPA)

Error

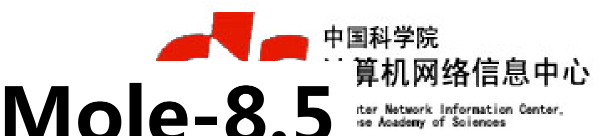
10^{2} 10^{0} 10^{2} 10^{4} 10^{6} 10^{8} 10^{10} 10^{12} 10^{14} 10^{16}

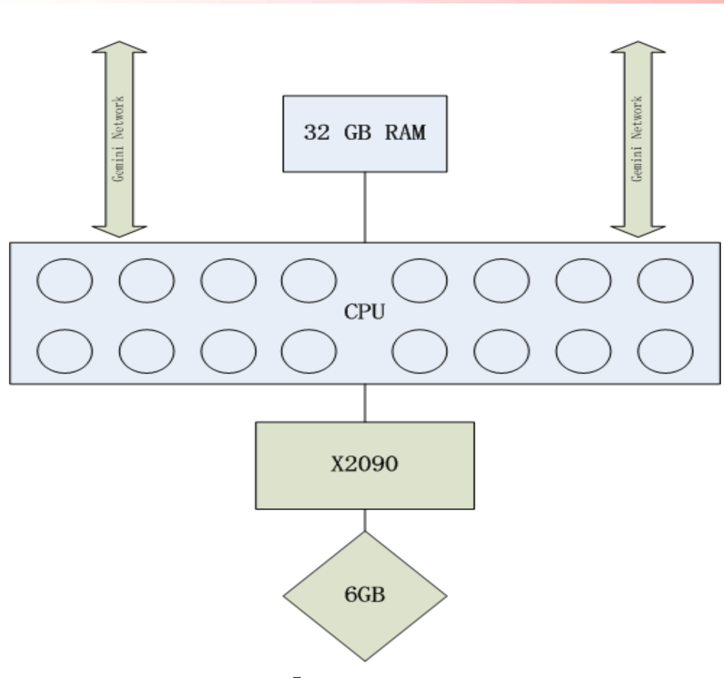
0 2 4 6 8 10 12 14 16 18 20

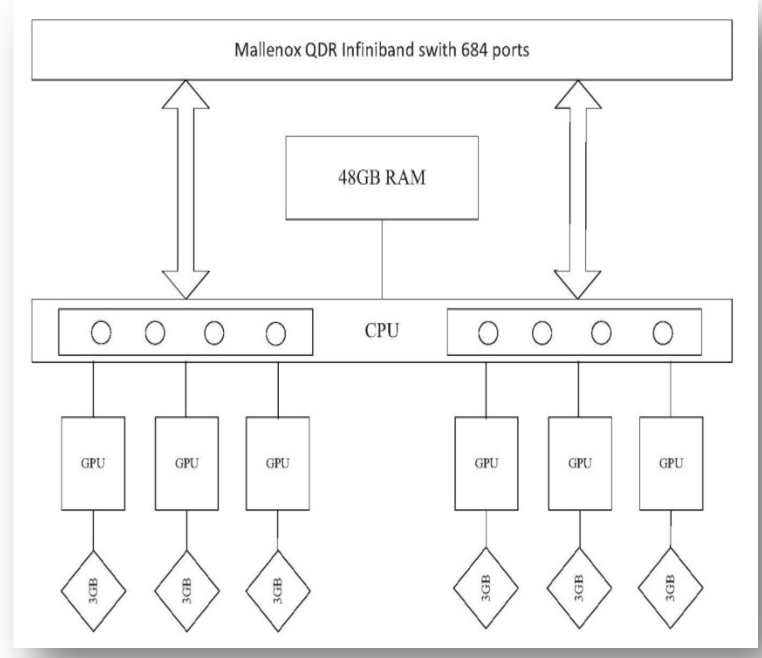
Iteration

Titan

Mole-8.5







Titan : 960 nodes
16-core 2.2 GHz Opteron 6274 CPU
1 Fermi X2090 GPU/node
(Oak Ridge Leadership Computing Facility)

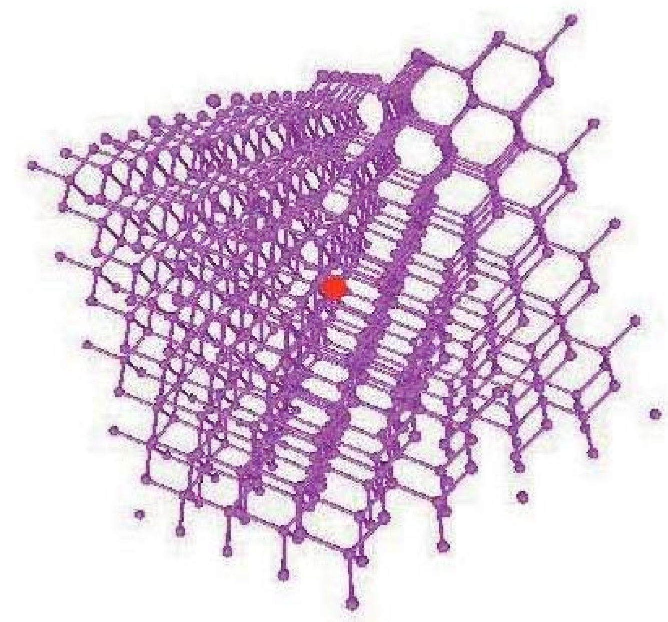
Mole-8.5 : 360 nodes
2 Xeon 5520 quad-core CPU
6 Fermi C2050 GPU cards/node
(Institute of Processing Engineering, CAS)

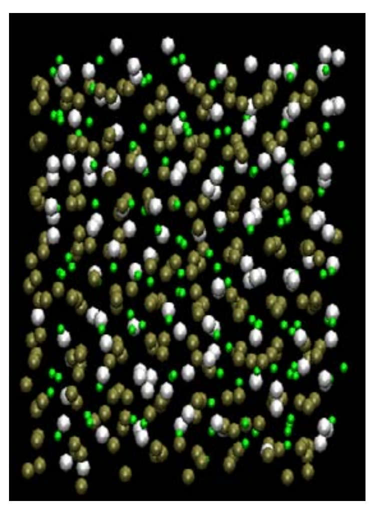
Strategy: one CPU core controls one GPU card, CPU/GPU unit

Testing systems

GaAs:N (512 atoms)
2048 electrons
128^3 FFT grid
40 Ryd Ecut
3.3 x10^5 PW coeff

Ga-In-P (512 atoms)
1800 steps of MD
128^3 FFT grid
Temperature is 1600K





CG_AllBand results

No. of computing units	32	64	128	256
Titan CPU(1core/node)	493s	274s	162s	106s
Titan CPU(16 core/node)	--	543s	323s	215s
Titan GPU	15.7	10.5	9.3	6.8
Titan Speedup	31x	26x	17x	15x
Mole-8.5 CPU	496s	284s	178s	125s
Mole-8.5 GPU	25.2s	15.4s	9.4s	7.7s
Mole-8.5 Speedup	20x	19x	19x	16x

The computational time and overall speed of the CG_AllBand comparing the CPU and GPU for the 512 atom GaAs:N test system. Each CG_AllBand has 4 CG steps. This is for the non-Γ point version of the CG_AllBand code

Scaling of different tasks in CG_Allband

CUBLAS zgemm and CUBLAS ztrsm
FFT
Nonlocal
Precond, line-minimization
MPI Alltoall
MPI Allreduce
Memcpy wave functions
Memcpy Mx*Mx
Zpotrf and Zheev

Time(s)
0 5 10 15 20 25 30 35
0 50 100 150 200 250
Computing units

① ② ③ ④

CUBLAS zgemm & ztrsm
Nonlocal
Compression
FFT
Precond and line minimize
Zheev
Zpotrf

Scalability
0 5 10 15
0 50 100 150 200 250
Computing units

① Num. Comp. ② MPI commun.
③ CPU-GPU memcpy ④ Matrix diag. lib

Different steps and speedups

speedup

35
30
25
20
15
10
5
0

x1.0
x4.3
x12.1
x24.5
x31

CPU time
CUBLAS
FFT inside GPU
AB-CG inside GPU
MPI data compression

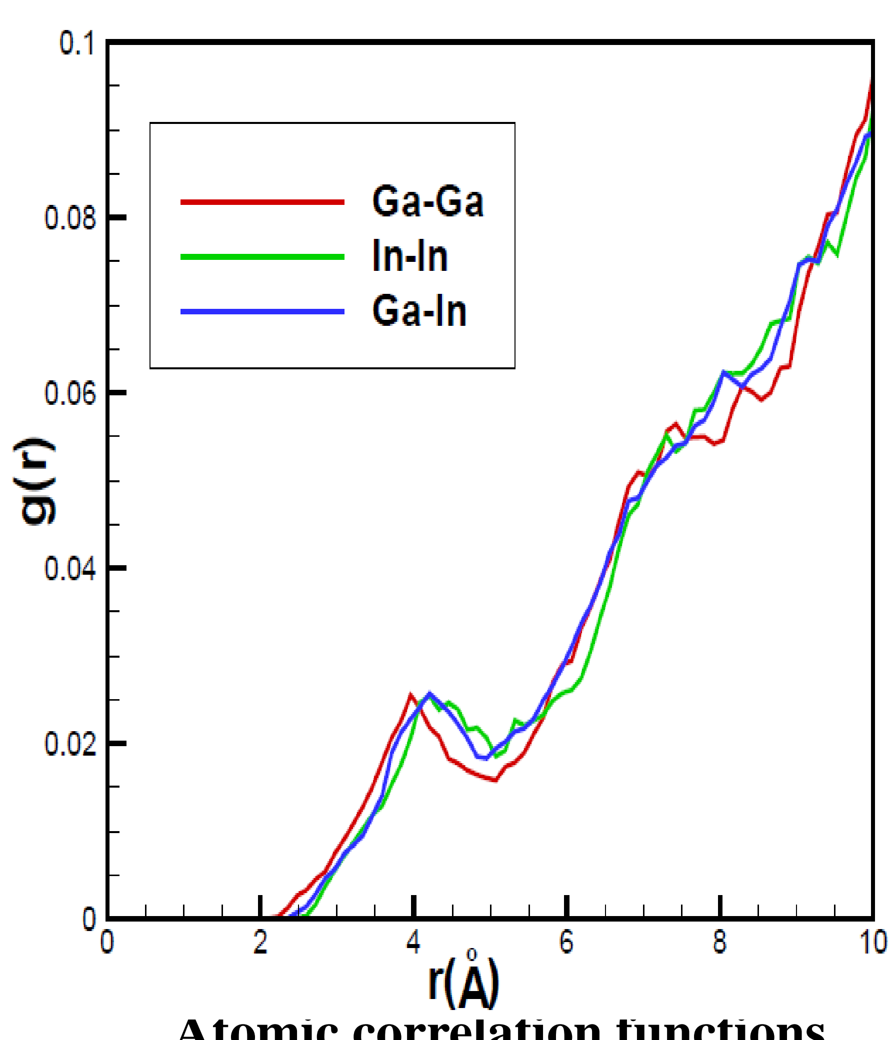
The speedup of GPU CG_AllBand over CPU PEtot code on Titan.

Time for one MD step

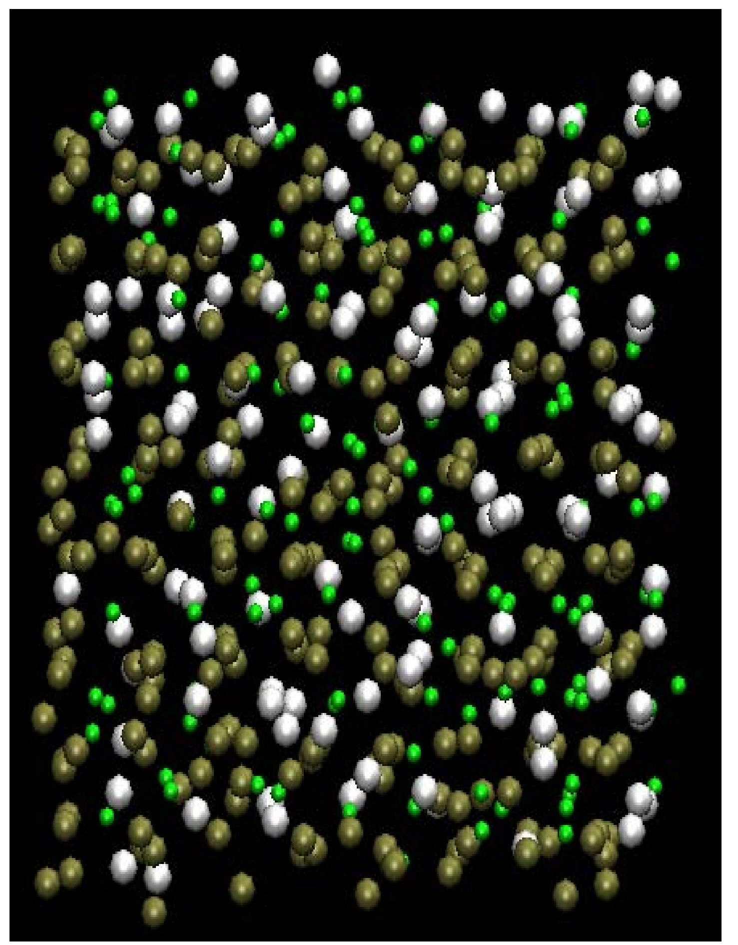
No. of CPU core	3216	6416	12816	25616
*PEtot_CPU(NP)	277s(8)	223s(8)	203s(8)	216s(8)
No. of GPU	32	64	128	256
PEtot_GPU(Titan)	31.6s	20.8s	13.2s	**11.4s**

The computational time and overall speedup compared with CPU of one MD step for runs with different CPU/GPU computing units for the 512 atom GaAs:N test system.

1800 MD steps of GaInP



Atomic correlation functions



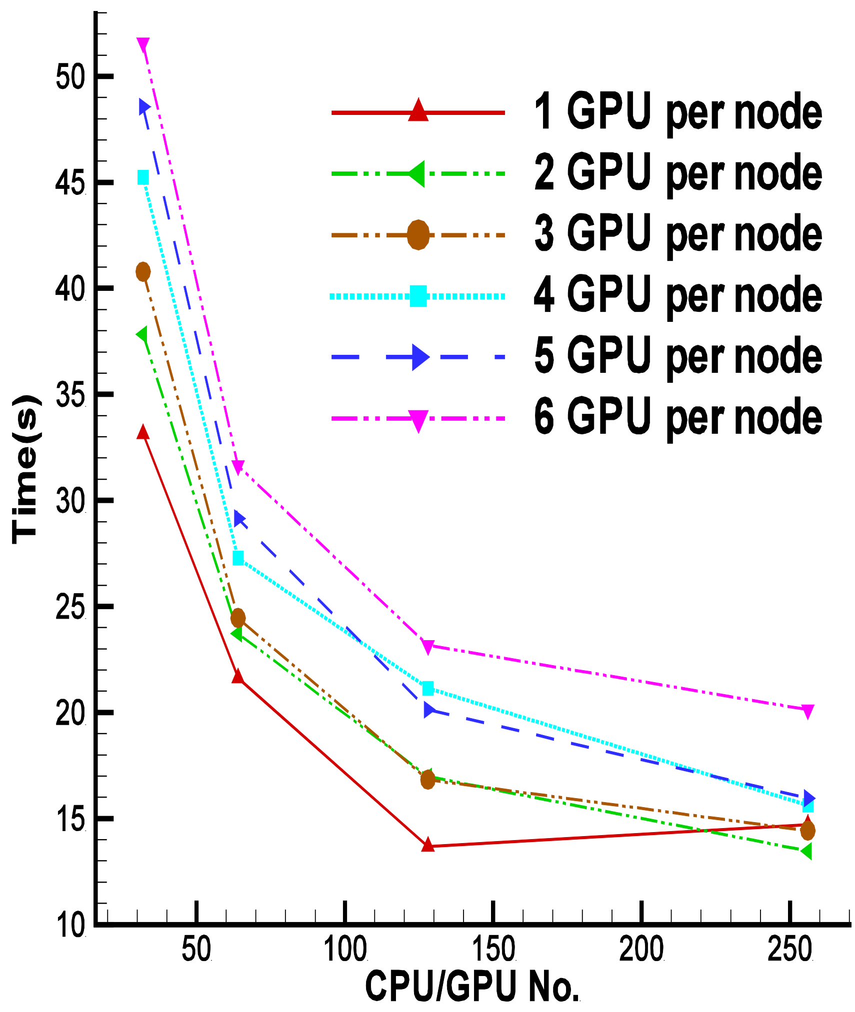
89年至今SC大陆地区一作文章

- **SC06**
- *Locality and parallelism optimization for dynamic programming algorithm in bioinformatics*
- **SC09**
- *Adaptive and scalable metadata management to support a trillion files*
- *FACT: fast communication trace collection for parallel applications through program slicing*
- *SmartStore: a new metadata organization paradigm with semantic-awareness for next-generation file systems*
- **SC11：352篇取74篇，录取率21%**
- *Fast implementation of DGEMM on Fermi GPU*
- *Large scale plane wave pseudopotential density function theory calculations on GPU clusters*

Different configurations

Testing results on Mole-8.5:

- **Generally, more GPU means faster speed**
- **Economically, 3 GPUs per node is the optimal way. (price*computation_time is the lowest)**



总结&反馈-CoDesign

- 计算核心31倍加速，相比领域最快的第一原理分子动力学模拟有7倍加速，相比原始CPU代码有18倍加速
- 提出了全新、系统的异构计算算法

The paper, in particular the GPU implementation is a great Achievement.
VASP Georg Kresse

希望在材料计算的众核软件开发方面进行合作。
JÜLICH Stefan Blügel

谢谢！