



# 并行计算实验上机

## 第一、二、三次实验内容指导



# 主要内容

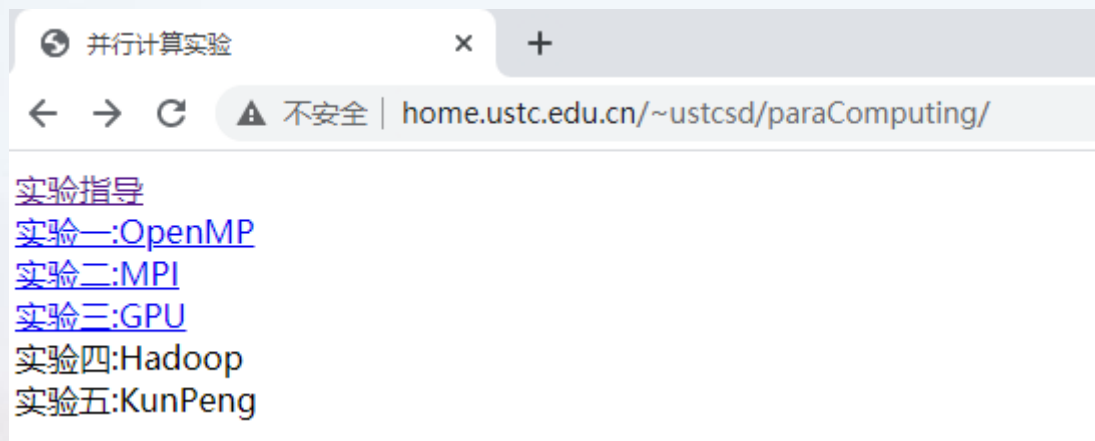
1. 资料下载
2. 关于上机
3. 上机题目(OpenMP、MPI、GPU)



# 实验资料

- 实验资料下载:

<http://home.ustc.edu.cn/~ustcsd/paraComputing/>





# 主要内容

1. 资料下载
2. 关于上机
3. 上机题目(OpenMP、MPI、GPU)



# 关于上机

- 上机地点：电三楼**5**楼机房
- 上机时间：
  - 5.15-6.12连续五周周六，分两批次：学号末位奇数9:00-12:00，学号末位偶数14:00-17:00
  - OpenMP: 5.15      9:00-12:00/14:00-17:00
  - MPI:      5.22      9:00-12:00/14:00-17:00
  - GPU:      5.29      9:00-12:00/14:00-17:00
  - Hadoop: 6.5      9:00-12:00/14:00-17:00（实验指导不在此文档内）
  - KunPeng: 6.12      9:00-12:00/14:00-17:00（实验指导不在此文档内）
- 上机要求：每次上机后，请在两周内提交你的实验报告，命名:OpenMP\_PB1401100\_张三，并发至邮箱：[ustc\\_pc2021@163.com](mailto:ustc_pc2021@163.com)



# 关于上机

## ■ 实验报告(模板)

《并行计算》上机报告					
姓名:		学号:		日期:	
上机题目:	MPI 并行编程实验				
实验环境:					
CPU: ; 内存: ; 操作系统: ; 软件平台:					
一、算法设计与分析:					
题目一:					
题目二:					
二、核心代码:					
题目一:					
题目二:					



# 主要内容

1. 资料下载
2. 关于上机
3. 上机题目(OpenMP、MPI、GPU)





# 实验1: OpenMP

## ■ 简介:

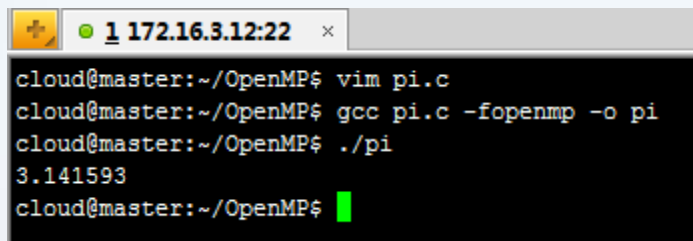
- OpenMP是一个共享存储并行系统上的应用程序接口。它规范了一系列的编译制导、运行库例程和环境变量。
- 它提供了C/C++和FORTRAN等的应用编程接口，已经应用到UNIX、Windows NT等多种平台上。
- OpenMP使用FORK-JOIN并行执行模型。所有的OpenMP程序开始于一个单独的主线程（Master Thread）。主线程会一直串行地执行，直到遇到第一个并行域（Parallel Region）才开始并行执行。
- ①FORK: 主线程创建一队并行的线程，然后，并行域中的代码在不同的线程队中并行执行；②JOIN: 当诸线程在并行域中执行完之后，它们或被同步或被中断，最后只有主线程在执行。





# 实验1: OpenMP

- 实验环境:
  - 在Linux上编译和运行OpenMP程序  
编译OpenMP程序: `gcc a.c -fopenmp -o a`  
运行OpenMP程序: `./a`

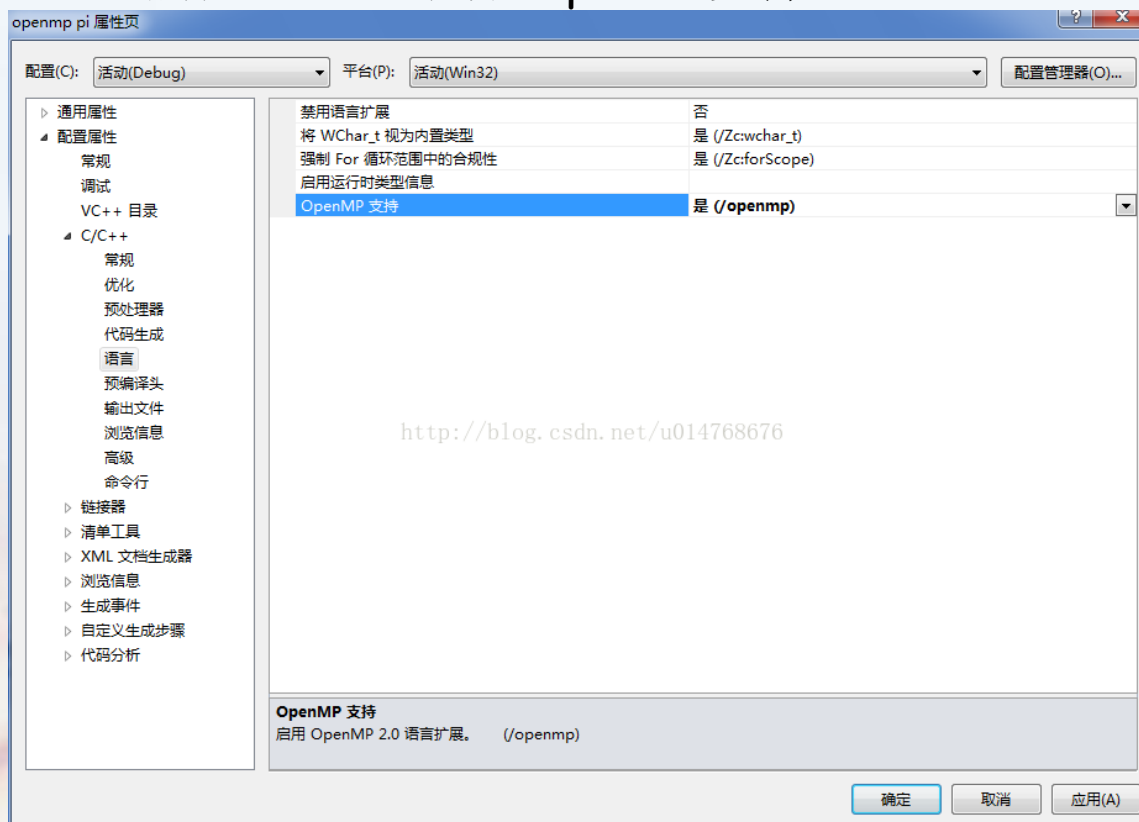


```
cloud@master:~/OpenMP$ vim pi.c
cloud@master:~/OpenMP$ gcc pi.c -fopenmp -o pi
cloud@master:~/OpenMP$ ./pi
3.141593
cloud@master:~/OpenMP$
```



# 实验1: OpenMP

- 实验环境:
  - 在Windows Microsoft Visual Studio中编写程序，可直接配置使用OpenMP。属性-C/C++-语言-OpenMP支持





# 实验1: OpenMP

- 题目:
  - 用4种不同并行方式的OpenMP实现 $\pi$ 值的计算
  - 用OpenMP实现PSRS排序(附1)
  
- 示例: Pi



# 实验1: OpenMP

## ■ Pi (串行):

```
#include <stdio.h>
static long num_steps = 100000; // 越大值越精确
double step;
void main(){
    int i;
    double x, pi, sum = 0.0;
    step = 1.0/(double)num_steps;
    for(i=1; i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step*sum;
    printf("%lf\n", pi);
}
```

在这里简单说明一下求 $\pi$ 的积分方法，使用公式 $\arctan(1)=\pi/4$ 以及 $(\arctan(x))'=1/(1+x^2)$ 。  
在求解 $\arctan(1)$ 时使用矩形法求解：  
求解 $\arctan(1)$ 是取 $a=0$ ,  $b=1$ 。

$$\int_a^b f(x)dx = y_0\Delta x + y_1\Delta x + \dots + y_{n-1}\Delta x$$

$$\Delta x = (b - a)/n$$

$$y = f(x)$$

$$y_i = f(a + i * (b - a)/n) \quad i = 0, 1, 2, \dots, n$$



# 实验1: OpenMP

## ■ Pi (使用并行域并行化的程序):

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
    int i;
    double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS); //设置2线程
    #pragma omp parallel private(i) //并行域开始, 每个线程(0和1)都会执行该代码
    {
        double x;
        int id;
        id = omp_get_thread_num();
        for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREADS){
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
        for(i=0, pi=0.0; i<NUM_THREADS; i++) pi += sum[i] * step;
        printf("%lf\n", pi);
    }
    //共2个线程参加计算, 其中线程0进行迭代步0,2,4,...线程1进行迭代步1,3,5,....
```



# 实验1: OpenMP

## ■ Pi (使用共享任务结构并行化的程序):

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
    int i;
    double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS); //设置2线程
    #pragma omp parallel //并行域开始，每个线程(0和1)都会执行该代码
    {
        double x;
        int id;
        id = omp_get_thread_num();
        sum[id]=0;
        #pragma omp for //未指定chunk，迭代平均分配给各线程（0和1），连续划分
        for (i=0;i< num_steps; i++){
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
        for(i=0, pi=0.0;i<NUM_THREADS;i++) pi += sum[i] * step;
        printf("%lf\n",pi);
    }
} //共2个线程参加计算，其中线程0进行迭代步0~49999，线程1进行迭代步50000~99999.
```





# 实验1: OpenMP

- Pi (使用private子句和critical部分并行化的程序):

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
    int i;
    double pi=0.0;
    double sum=0.0;
    double x=0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS); //设置2线程
    #pragma omp parallel private(x,sum) //该子句表示x,sum变量对于每个线程是私有的
    {
        int id;
        id = omp_get_thread_num();
        for (i=id, sum=0.0; i< num_steps; i=i+NUM_THREADS){
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
        #pragma omp critical //指定代码段在同一时刻只能由一个线程进行执行
        pi += sum*step;
    }
    printf("%lf\n",pi);
}
//共2个线程参加计算，其中线程0进行迭代步0,2,4,...线程1进行迭代步1,3,5,...当被指定为critical的代码段正在被0线程执行时，1线程的执行也到达该代码段，则它将被阻塞知道0线程退出临界区。
```



# 实验1: OpenMP

## ■ Pi (使用并行规约的并程序):

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
    int i;
    double pi=0.0;
    double sum=0.0;
    double x=0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS); //设置2线程
    #pragma omp parallel for reduction(+:sum) private(x) //每个线程保留一份私有拷贝sum，x为线程私有，最后对
    线程中所以sum进行+规约，并更新sum的全局值
    for(i=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    pi = sum * step;
    printf("%lf\n",pi);
} //共2个线程参加计算，其中线程0进行迭代步0~49999，线程1进行迭代步50000~99999.
```



# 主要内容

1. 资料下载
2. 关于上机
3. 上机题目(OpenMP、MPI、GPU)



## 实验2: MPI

- 简介:
  - **MPI (Message Passing Interface)** 是目前最重要的一个基于消息传递的并行编程工具，它具有移植性好、功能强大、效率高等许多优点，而且有多种不同的免费、高效、实用的实现版本，几乎所有的并行计算机厂商都提供对它的支持，成为了事实上的并行编程标准。
  - **MPI**是一个库，而不是一门语言，因此对**MPI**的使用必须和特定的语言结合起来进行。**MPI**不是一个独立的自包含系统，而是建立在本地并行程序设计环境之上，其进程管理和**I/O**均由本地并行程序设计环境提供。例如，**MPI**可以建立在**IBM SP2**的**POE/MPL**之上，也可以建立在**Intel Paragon**的**OSF/NX**。除了这些商业版本的**MPI**实现，还有一些免费版的**MPI**实现，主要有**MPICH**，**LAM**和**CHIMP**。



## 实验2: MPI

### ■ 实验环境:

- 本次实验, 要求自己在Linux环境下搭建单结点的MPI环境, 有条件的同学, 可以尝试多节点。
- 用的版本是MPICH-3.0.4, 下载后, 解压, 安装配置如下:

```
./configure --enable-fc --enable-cxx --enable-romio --enable-threads=multiple --  
prefix=${HOME}/soft/mpich2/3.0.4 --with-pm=mpd  
make  
make install
```

设置环境变量:

编辑~/.bashrc, 在文件的末尾, 添加如下几行

```
export PATH=${HOME}/soft/mpich2/3.0.4/bin:${PATH}  
export LD_LIBRARY_PATH=${HOME}/soft/mpich2/3.0.4/lib:${LD_LIBRARY_PATH}  
export MANPATH=${HOME}/soft/mpich2/3.0.4/share/man:${MANPATH}  
更新环境变量, source
```

编辑\${HOME}/.mpd.conf文件, 添加一行: MPD\_SECRETWORD=mypasswd

修改该文件权限, chmod 600

启动进程管理器: mpdboot

查看: mpdtrace





## 实验2: MPI

- 编译运行:
  - 编译mpi程序: `mpicc demo.c -o demo.o`
  - 运行mpi程序: `mpirun -np 4 ./demo.o` (-np选项指定需要运行的进程数, 大家可以自由设置, 而非固定使用此处的4)





## 实验2: MPI

- 题目:
  - 用MPI编程实现PI的计算。
  - 用MPI实现PSRS排序(附1)。



## 附 (1)

### ■ 划分方法

$n$ 个元素 $A[1..n]$ 分成 $p$ 组, 每组 $A[(i-1)n/p+1..in/p]$ ,  $i=1\sim p$

### ■ 示例: MIMD-SM模型上的PSRS排序

begin

(1)均匀划分: 将 $n$ 个元素 $A[1..n]$ 均匀划分成 $p$ 段, 每个 $p_i$ 处理

$A[(i-1)n/p+1..in/p]$

(2)局部排序:  $p_i$ 调用串行排序算法对 $A[(i-1)n/p+1..in/p]$ 排序

(3)选取样本:  $p_i$ 从其有序子序列 $A[(i-1)n/p+1..in/p]$ 中选取 $p$ 个样本元素

(4)样本排序: 用一台处理器对 $p^2$ 个样本元素进行串行排序

(5)选择主元: 用一台处理器从排好序的样本序列中选取 $p-1$ 个主元, 并播送给其他 $p_i$

(6)主元划分:  $p_i$ 按主元将有序段 $A[(i-1)n/p+1..in/p]$ 划分成 $p$ 段

(7)全局交换: 各处理器将其有序段按段号交换到对应的处理器中

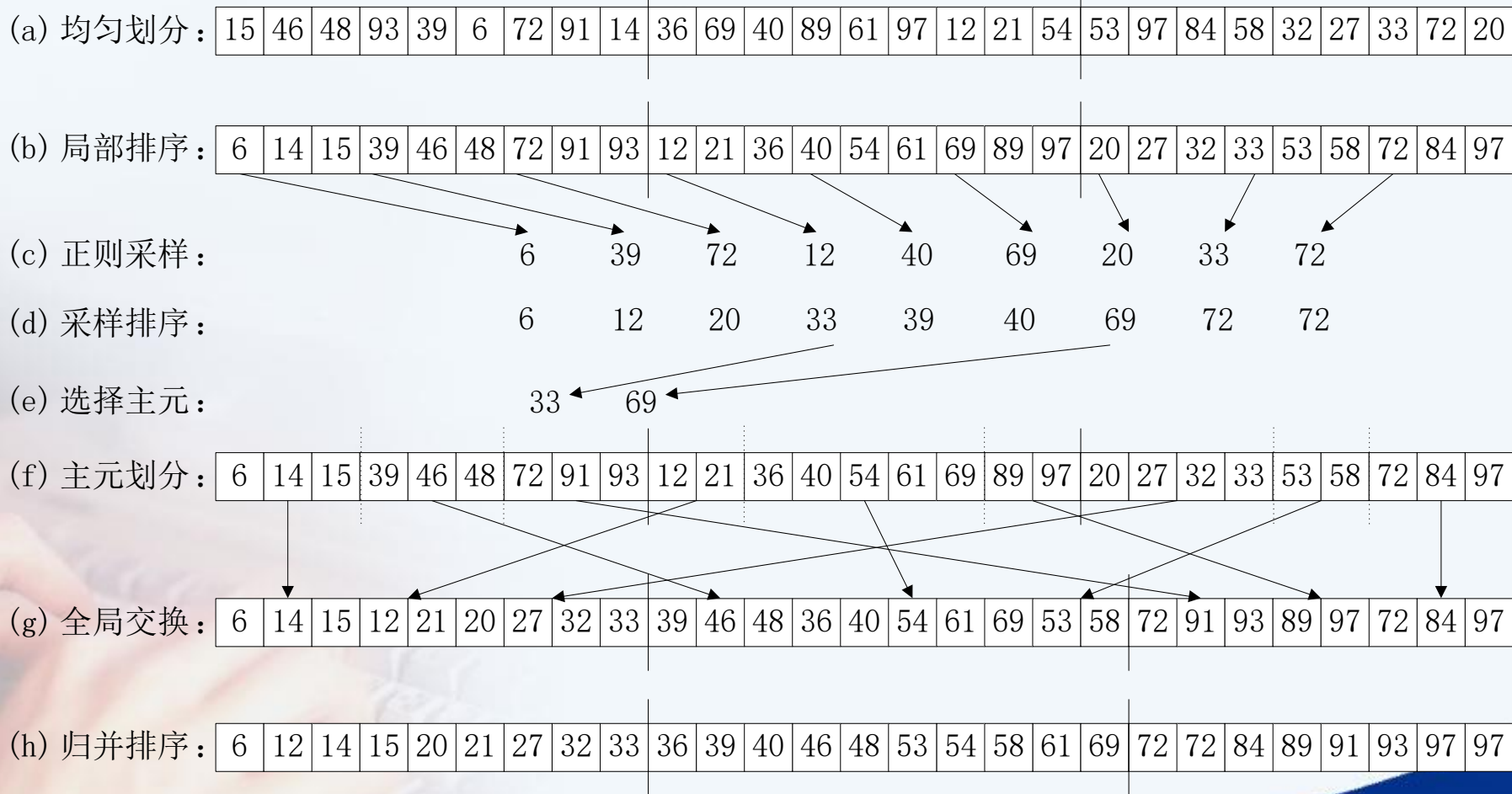
(8)归并排序: 各处理器对接收到的元素进行归并排序

end.



# 附 (1)

## ■ 例6.1 PSRS排序过程。N=27, p=3, PSRS排序如下:





# 主要内容

1. 资料下载
2. 关于上机
3. 上机题目(OpenMP、MPI、GPU)



## 实验3: GPU

- 简介:
  - **CUDA™**是一种由**NVIDIA**推出的通用并行计算架构，该架构使**GPU**能够解决通用的计算问题。
  - 它提供了**C/C++**和**FORTRAN**等的应用编程接口，已经应用到**UNIX**、**Windows NT**等多种平台上。
- 编译:
  - **Windows**下可直接使用**Windows Microsoft Visual Studio**等集成开发环境。
  - **Linux**下编译: `nvcc cuda.cu`。



## 实验3: GPU

- Windows下安装:
  - 在设备管理器中查看GPU是否支持CUDA。
  - 在CUDA下载页面选择合适的系统平台，下载对应的开发包。
  - 安装开发包，需要预先安装Visual Studio 2010 或者更高版本。
  - 验证安装，打开命令提示框，输入命令nvcc - V。
- Linux下安装:
  - 使用lspci |grep nvidia -I 命令查看GPU型号。
  - 在CUDA下载页面选择合适的系统平台，下载对应的开发包(\*.run)。
  - 安装：使用：

```
chmod a+x cuda_7.0.28_linux.run sudo  
./cuda_7.0.28_linux.run。
```
  - 设置环境变量：

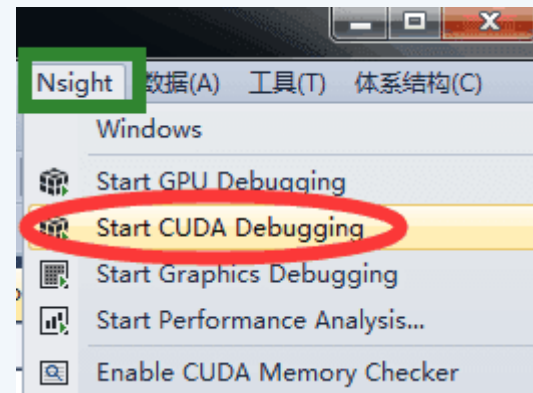
```
PATH=/usr/local/cuda/bin:$PATH export PATH  
source /etc/profile
```





## 实验3: GPU

- Windows下创建及调试:
  - 新建项目-CUDA 7.0 Runtime。
  - 调试: 使用Nsight 进行调试:  
Nsight->start CUDA debugging



- Linux下创建及调试:
  - 创建\*.cu以及\*.cuh文件, 需包含<cuda\_runtime.h>头文件。
  - 调试: 使用cuda-gdb进行调试:  
nvcc-g -G \*.cu -o binary
    - nvcc为cuda程序编译器。
    - g 表示可调试。
    - \*.cu 为cuda源程序。
    - o 生成可执行文件。



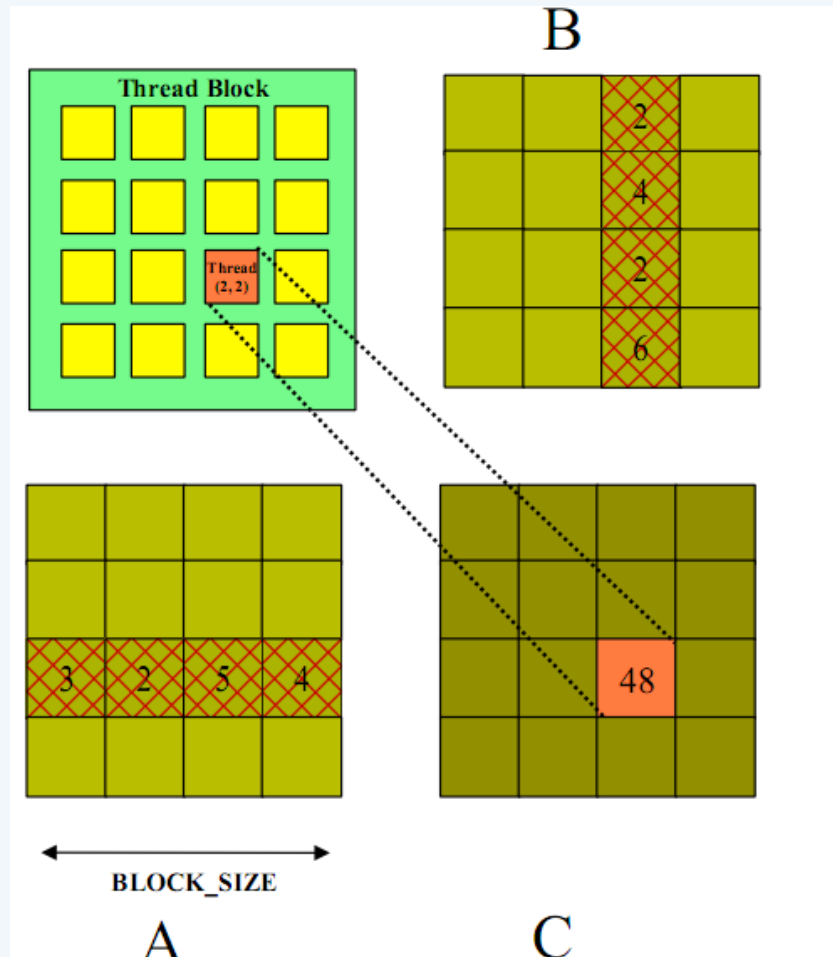
## 实验3: GPU

- 题目:
  - 编程实现GPU上的矩阵乘法。
  - 编程实现GPU上的矩阵向量乘。
- 示例:
  - GPU上的矩阵乘法。



## 实例：GPU上矩阵乘法

- GPU上矩阵乘法：
  - 每一个线程计算**C**矩阵中的一个元素。
  - 每一个线程从全局存储器读入**A**矩阵的一行和**B**矩阵的一列。
  - A**矩阵和**B**矩阵中每个元素都被方位  $N = \text{BLOCK\_SIZE}$  次。





## 实例：GPU上矩阵乘法

### ■ GPU上矩阵乘法(主机端函数):

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

//CUDA RunTime API
#include <cuda_runtime.h>
//单个block大小
#define THREAD_NUM 256
///矩阵大小
#define MATRIX_SIZE 1000
///block个数
int blocks_num = (MATRIX_SIZE + THREAD_NUM - 1) / THREAD_NUM;

int main() {
    //定义矩阵
    float *a, *b, *c, *d;
    int n = MATRIX_SIZE;
    //分配主机端内存
    a = (float*)malloc(sizeof(float)* n * n);
    b = (float*)malloc(sizeof(float)* n * n);
    c = (float*)malloc(sizeof(float)* n * n);
    d = (float*)malloc(sizeof(float)* n * n);
    float *cuda_a, *cuda_b, *cuda_c;
    ...
}
```



## 实例：GPU上矩阵乘法

### ■ GPU上矩阵乘法(主机端函数):

```
...  
//分配设备端显存  
cudaMalloc((void**)&cuda_a, sizeof(float)* n * n);  
cudaMalloc((void**)&cuda_b, sizeof(float)* n * n);  
cudaMalloc((void**)&cuda_c, sizeof(float)* n * n);  
///生成矩阵a, b  
generateMatrix(a, b);  
  
//cudaMemcpyHostToDevice - 从内存复制到显存  
//cudaMemcpyDeviceToHost - 从显存复制到内存  
cudaMemcpy(cuda_a, a, sizeof(float)* n * n, cudaMemcpyHostToDevice);  
cudaMemcpy(cuda_b, b, sizeof(float)* n * n, cudaMemcpyHostToDevice);  
  
///设备端函数  
CUDataKernel << < blocks_num, THREAD_NUM, 0 >> >(cuda_a , cuda_b , cuda_c , n , time);  
  
//cudaMemcpy 将结果从显存中复制回内存  
cudaMemcpy(c, cuda_c, sizeof(float)* n * n, cudaMemcpyDeviceToHost);  
//Free  
cudaFree(cuda_a);  
cudaFree(cuda_b);  
cudaFree(cuda_c);  
}
```





## 实例：GPU上矩阵乘法

- GPU上矩阵乘法(设备端函数):

```
__global__ static void CUDAKernal (const float* a, const float* b, float* c, int n)
{

    //block内的threadID
    const int tid = threadIdx.x;

    //blockID
    const int bid = blockIdx.x;

    //全局threadID
    const int idx = bid * THREAD_NUM + tid;

    const int row = idx / n;

    const int column = idx % n;

    ....
}
```





## 实例：GPU上矩阵乘法

- GPU上矩阵乘法(设备端函数):

```
....  
  
//计算矩阵乘法  
if (row < n && column < n)  
{  
    float t = 0;  
  
    for (i = 0; i < n; i++)  
    {  
        t += a[row * n + i] * b[i * n + column];  
    }  
    c[row * n + column] = t;  
}  
}
```



## 实例：GPU上矩阵乘法

### ■ GPU上矩阵乘法(shared memory):

```
__global__ static void CUDAKernal (const float* a, const float* b, float* c, int n)
{
    ///静态分配shared memory
    __shared__ int s[64];
    ....
}

...

///动态分配shared memory
CUDAKernal << < blocks_num, THREAD_NUM, N >> >(cuda_a , cuda_b , cuda_c , n , time);

....
```