СТРОИТЕЛЬСТВО

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ
СТРОИТЕЛЬНЫЙ УНИВЕРСИТЕТ

Кафедра прикладной математики

ОСНОВЫ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Методические указания к практическим занятиям
для обучающихся бакалавриата по всем УГСН 01.00.00, реализуемым НИУ МГСУ

Составители:
Ю.П. Галагуз, М.С. Максютов

Москва
Издательство МИСИ – МГСУ
2020

УДК 519.6
ББК 22.19
О-75

Рецензент — кандидат технических наук, доцент *В.И. Прокопьев*,
доцент кафедры прикладной математики НИУ МГСУ

О-75 **Основы параллельных вычислений** [Электронный ресурс] : методические указания к практическим занятиям для обучающихся бакалавриата по всем УГСН 01.00.00, реализуемым НИУ МГСУ / сост.: Ю.П. Галагуз, М.С. Максютов ; Министерство науки и высшего образования Российской Федерации ; Национальный исследовательский Московский государственный строительный университет, кафедра прикладной математики. — Электрон. дан. и прогр. (0,8 Мб). — Москва : Издательство МИСИ – МГСУ, 2020. — Режим доступа : http://lib.mgsu.ru/Scripts/irbis64r_91/cgiirbis_64.exe?C21COM=F&I21DBN=IBIS&P21DBN=IBIS. — Загл. с титул. экрана.

В методических указаниях приведены варинты заданий, примеры решения задач.
Для обучающихся бакалавриата по всем УГСН 01.00.00, реализуемым НИУ МГСУ.

Учебное электронное издание

Редактор, корректор *А.А. Космина*
Компьютерная верстка *О.Г. Горюновой*
Дизайн первого титульного экрана *Д.Л. Разумного*

Для создания электронного издания использовано:
Microsoft Word 2010, Adobe InDesignCS5.5, ПО Adobe Acrobat

Подписано к использованию 15.02.2020 г. Объём данных 0,8 Мб.

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Национальный исследовательский
Московский государственный строительный университет».
129337, Москва, Ярославское ш., 26.

Издательство МИСИ – МГСУ.
Тел.: (495) 287-49-14, вн. 13-71, (499) 188-29-75, (499) 183-97-95.
E-mail: ric@mgsu.ru, rio@mgsu.ru

Оглавление

1. OPENMP

Параллельное программирование

Параллельное программирование применяется тогда, когда для последовательной программы требуется уменьшить время её выполнения или когда последовательная программа ввиду большого объёма данных перестаёт помещаться в память одного компьютера. Направление развития в области высокопроизводительных вычислений как раз направлено на решение этих двух задач: создание мощных вычислительных комплексов с большим объёмом оперативной памяти, с одной стороны, и разработка соответствующего программного обеспечения (ПО) — с другой. Однако большая часть известных алгоритмов являются последовательными и довольно трудно распараллеливаются. Причиной возникающих сложностей является зависимость результата вычислений от предыдущих значений. Тем не менее, многие задачи можно распараллелить и можно часто получить огромный выигрыш во времени.

Введение в OpenMP

OpenMP (Open Multi-Processing) — открытый стандарт для распараллеливания программ на языках C/C++ и Fortran. Первая версия появилась в 1997 г. (Fortran) и 1998 г. (C/C++). OpenMP — механизм написания параллельных программ для систем с общей памятью. Состоит из набора директив компилятора и библиотечных функций. Позволяет достаточно легко создавать многопоточные приложения на C/C++, Fortran. Находит применение в многопроцессорных или многоядерных системах с разделяемой памятью.

Программная модель OpenMP

Основная нить порождает дочерние нити по мере необходимости. Программирование ведётся путём вставки директив компилятора в ключевые места исходного кода программы. Компилятор интерпретирует эти директивы и вставляет в соответствующие места программы библиотечные вызовы для распараллеливания участков кода. Вставки в код начинаются с директивы *#pragma omp*. Например, директива *#pragma omp parallel for* указывает на то, что следуемый сразу за ней оператор цикла *for* будет разделён по итерациям между нитями.

OpenMP позволяет программно контролировать число нитей. В среде Visual Studio, чтобы иметь возможность работать с OpenMP, нужно подключит опцию */openmp*. В Visual Studio в свойствах проекта выбираем C/C++, затем Language и включаем опцию Open MP Support.

В модели с разделяемой памятью взаимодействие нитей происходит через разделяемые переменные. При неаккуратном обращении с такими переменными в программе могут возникнуть т.н. ошибки соревнования. Это происходит из-за того, что нити выполняются параллельно и, соответственно, последовательность доступа к разделяемым переменным может быть различна при каждом запуске программы.

Для контроля ошибок соревнования работу нитей необходимо синхронизировать. Для этого используются такие примитивы синхронизации как критические секции, барьеры, атомарные операции и блокировки. Однако синхронизация может потребовать от программы дополнительных накладных расходов, и её часто стараются минимизировать.

Так как конструкции OpenMP являются директивами, то если компилятор их не понимает, он их пропустит, и программа будет работать последовательно.

Параллельные регионы

Параллельные регионы являются основным понятием в OpenMP. Именно там, где задан этот регион, программа исполняется параллельно. Как только компилятор встречает *#pragma omp parallel*, он вставляет инструкции для создания параллельных нитей.

Каждая порождённая нить исполняет код в структурном блоке. По умолчанию синхронизация между нитями отсутствует, и поэтому последовательность выполнения конкретного оператора различными нитями не определена.

После выполнения параллельного участка кода все нити, кроме основной, завершаются, и только основная нить затем продолжает исполняться.

Каждая нить имеет свой уникальный номер, который изменяется от 0 (для основной нити) до числа нитей минус 1. Идентификатор нити может быть определён с помощью функции *omp_get_thread_num()*.

OpenMP предполагает, что программа выполняется в системе с разделяемой памятью (*shared*), в которой хранятся переменные программы, доступные всем её нитям. Кроме того, каждая нить имеет доступ к памяти, недоступной (*private*) для всех других нитей.

Полезная функция *double omp_get_wtime(void)* возвращает затраченное время в секундах.

Условие ***reduction.*** Это условие позволяет производить безопасное глобальное вычисление. Приватная копия каждой перечисленной переменной инициализируется при входе в параллельную секцию в соответствии с указанным оператором, например, при подсчёте суммы, которая будет записана в переменную *S* в цикле *for*. Вид директивы следующий: *#pragma omp parallel for reduction(+:S)*. При выходе из параллельной секции из частично вычисленных значений вычисляется результирующее и передаётся в основной поток.

Пример 1. Нахождение определённого интеграла $\int_0^1 x^2 dx$ методом трапеций

```
#include<iostream>
#include<time.h>
#include<omp.h>
#ifndef _OPENMP
int pp = printf("NO OPENMP\n");
#else
int pp = printf("%d\n",_OPENMP);
#endif

typedef double Type;
const unsigned int n=5000000+1;
Type *f=new Type[n];
Type fun(Type x){return x*x;}
Type sumOpenMP(Type*f, unsigned int n){Type s=0;
#pragma omp parallel for reduction(+:s) if(1)
for(int i=1;i<n-1;i++)s+=f[i];
return s+(f[0]+f[n-1])/2;}
```

```
Type sum(Type *f, unsigned int n){Type s=0;
#pragma omp parallel for reduction(+:s) if(0)
for(int i=1;i<n-1;i++)s+=f[i];
return s+(f[0]+f[n-1])/2;}

void main(){
double t1,t2;
Type s, dn=Type(n);
Type xa=0, xb=1, h=(xb-xa)/(dn-1);
for(int i=0;i<=n-1;i++)f[i]=fun(xa+i*h);
t1=omp_get_wtime(); s=h*sumOpenMP(f,n);
t2=omp_get_wtime();
printf("Time OpenMP    = %f\n",t2-t1);
printf("Exact S = %25.20lf  Trapz S OpenMP    = "
"%25.20lf\n",1/3.0,s);
t1=omp_get_wtime(); s=h*sum(f,n); t2=omp_get_wtime();
printf("Time nonOpenMP = %f\n",t2-t1);
printf("Exact S = %25.20lf  Trapz S nonOpenMP = "
"%25.20lf\n",1/3.0,s);}
```

Пример 2. Решение уравнения Пуассона $\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=f(x,y)$, $0\le x\le 1$, $0\le y\le 1$, $u_\Gamma=0$, $u(x, y)$ — ?

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include <math.h>
#include<time.h>
#include<omp.h>
#include<windows.h>
#include<direct.h>
using namespace std;
#ifndef _OPENMP
int pp=printf("NO OPENMP\n\n");
#else
int pp=printf("%d\n\n",_OPENMP);
#endif

typedef double T;
const unsigned int n=50+1;
const unsigned int N=n*n;
const unsigned int CYC=20000;
T U[N],f[N],x[n],y[n];
T fun(T x, T y){return x+y;}
void main(){
T dN =T(N), h=T(1)/(n-1);
```

```
double t1,t2;
for (int j=0;j<n;j++)for(int i=0;i<n;i++)
{f[j*n+i]=fun(i*h,j*h);U[j*n+i]=0;}
for(int j=0;j<n;j++)x[j]=j*h;
for(int i=0;i<n;i++)y[i]=i*h;
t1=omp_get_wtime();
#pragma omp parallel for num_threads(12) if(1)
for(int cyc=1;cyc<=CYC;cyc++)
//#pragma omp parallel for num_threads(12) if(1)
for(int i=2;i<=n-1;i++) for(int j=2;j<=n-1;j++)
U[n*(i-1)+j-1]=(U[n*(i-1)+j-1-1]+U[n*(i-1)+j]+
U[n*(i-2)+j-1]+U[n*i+j-1])/4-h*h/4*f[n*(i-1)+j-1];
t2=omp_get_wtime();

printf("t1=%f t2=%f Time=%f\n",t1,t2,t2-t1);
printf("N/2 = %d Umkr = %f\n",N/2,U[N/2]);

FILE *fg,*ft; char *dir="c:/Figures/"; _mkdir(dir);
char fileg[40]; strcat(strcpy(fileg,dir),
"poissong.gnu");
char filet[40]; strcat(strcpy(filet,dir),
"poissont.txt");
char gnuplot[80]="\"c:/Program Files/gnuplot/bin/"
"gnuplot.exe\" "; strcat(gnuplot, fileg);
fg=fopen(fileg,"w+"); ft=fopen(filet,"w+");
int I=n/2; for(int j=0;j<n;j++)fprintf(ft,
"%10.5f %10.5f\n",x[j],U[I*n+j]);
fprintf(fg,"set xlabel 'x';plot '%s' u 1:2 title"
" \"mkr\" with lines lt rgb \"#0000FF\" lw 2;"
"pause mouse keypress",filet,filet);
fclose(fg); fclose(ft); system(gnuplot);
fg=fopen(fileg,"w+"); ft=fopen(filet,"w+");
int J=n/2; for(int i=0;i<n;i++)fprintf(ft,
"%10.5f %10.5f\n",y[i],U[J*n+i]);
fprintf(fg,"set xlabel 'y';plot '%s' u 1:2 title"
" \"mkr\" with lines lt rgb \"#00FF00\" lw 2;"
"pause mouse keypress",filet,filet);
fclose(fg); fclose(ft); system(gnuplot);
fg=fopen(fileg,"w+"); ft=fopen(filet,"w+");
for(int i=0;i<n;i++){for(int j=0;j<n;j++)
fprintf(ft,"%10.5f %10.5f %10.5f\n",x[j],y[i],
U[i*n+j]); fprintf(ft,"\n");}
fprintf(fg,"set xlabel 'x';set ylabel 'y';set "
"palette model HSV defined ( 0 0 1 1, 1 1 1 1);splot"
" '%s' u 1:2:3 with pm3d;pause mouse keypress",filet);
fclose(fg); fclose(ft);    system(gnuplot);}
```

2. CUDA

CUDA (Compute Unified Device Architecture, 2007 год) — программно-аппаратная архитектура параллельных вычислений. Работает только на видеокартах Nvidia. Архитектура CUDA даёт разработчику возможность по своему усмотрению организовывать доступ к набору инструкций видеокарте (графическом процессоре GPU — graphics processing unit) и управлять её памятью. Функции, ускоренные при помощи CUDA, можно также вызывать из различных языков, например, Python и MATLAB.

Средство разработки NVIDIA SDK позволяет создавать программы, обеспечивающие выполнение вычислений общего назначения на GPU. Такие программы позволяют повысить производительность без привлечения дорогостоящих суперкомпьютеров. CUDA, наряду с OpenCL — это очень эффективный подход к построению высокоскоростных вычислительных систем.

Для работы в CUDA C/C++ в среде Visual Studio необходима видеокарта Nvidia с поддержкой ядер-CUDA (это верно для большинства даже недорогих видеокарт). Осталось лишь инсталлировать драйвера видеокарты и CUDA Toolkit. После установки CUDA Toolkit в Visual Studio появляется шаблон для создания проектов NVIDIA. Создание проекта Nvidia в Visual Studio очень просто и мало чем отличается от создания проекта C/C++, C# и т.д.

Что же касается выбора конкретной видеокарты Nvidia или AMD для вычислений в CUDA, OpenACC, OpenCL, то в прикладной математике и механике преобладает или, во всяком случае, часто нужна работа именно с числами с плавающей точкой (запятой) двойной точности (FP64, double), а не одинарной (FP32, float). Но большинство современных видеокарт для настольных компьютеров, часто обладая выдающейся производительностью с FP32, имеют в разы (в 16, в 32 и т.п.) более слабую производительность в FP64. В то же время высокопроизводительные специализированные видеокарты с отличным соотношением FP64 : FP32 очень дорогие.

Программа, выполняемая на GPU, но вызываемая с центрального процессора (CPU — central processing unit), называется ядром (kernel). При написании нового или переносе старого кода на CUDA обычно в подобном kernel преобразуются важные или критические участки кода, поддающиеся распараллеливанию. Если эти участки изолированы, то потребуется передавать данные с хоста (основного компьютера) на устройство (видеокарту) и обратно, а это затратная операция, которую, по возможности, стараются как можно реже применять. На этой стадии программирования нам могут оказать помощь профилировщики (nvprof). При программировании в CUDA необходимо думать (чаще, чем в OpenMP) о синхронизации между хостом и устройством. Например, для синхронизации всех нитей в блоке вызывают функцию *__syncthreads()*, для синхронизации всех потоков kernel *cudaDeviceSynchronize ()*.

Технология CUDA вводит ряд дополнительных расширений для языка C/C++, которые необходимы для написания кода для GPU: Это спецификаторы функций, которые показывают, как и откуда будут выполняться функции, спецификаторы переменных, которые служат для указания типа используемой памяти GPU, спецификаторы запуска ядра GPU, встроенные переменные для идентификации нитей, блоков и др. параметров при исполнении кода в ядре GPU и также дополнительные типы переменных.

Всего в CUDA три таких спецификатора функций:

1) *__host__* — выполняется на CPU, вызывается с CPU;

2) *__global__* — выполняется на GPU, вызывается с CPU;
3) *__device__* — выполняется на GPU, вызывается с GPU.

Спецификаторы запуска ядра служат для описания количества блоков, нитей и памяти, которые мы хотим выделить при расчёте на GPU. Синтаксис запуска ядра имеет следующий вид:

KernelFunction<<<gridSize,blockSize>>>(Type param1, Type param2,...), где *myKernelFunc* — функция ядра (спецификатор *__global__*), *gridSize* — размерность сетки блоков (dim3 — тип, встроенный в CUDA), выделенную для расчётов, *blockSize* — размер блока (тип dim3), т.е. число нитей, выделенных для расчётов. В язык добавлены следующие специальные переменные: *gridDim* — размер сетки grid (имеет тип dim3), *blockDim* — размер блока (имеет тип dim3), *blockIdx* — индекс текущего блока в grid (имеет тип uint3), *threadIdx* — индекс текущей нити в блоке (имеет тип uint3).

Для работы с блоками и нитями необходимо знать возможности видеокарты, например, число нитей в блоке или блоков в сетке и т.д. Для этого можно воспользоваться утилитами типа CUDA-Z и GPU-Z.

Решение уравнения теплопроводности — явная схема (CUDA C) $\frac{\partial u}{\partial t} = a\frac{\partial^2 u}{\partial x^2}$, $u(0, t) = 2$, $u(1, t) = 1$, $u(x, 0) = \sin(\pi x) - x + 2$, $0 \le x \le 1$, $0 \le t \le 0{,}5$, $u(x, t)$ — ?

```
#include<stdio.h>
#include<string.h>
#include<malloc.h>
#include<math.h>
#include<time.h>
#include<direct.h>
#include<cuda.h>
#include"cuda_runtime.h"
#include"device_launch_parameters.h"

const int METHOD=1;
typedef double Ty;
#define pi Ty(acos(-1))

#define a 1.0
#define xa 0.0
#define xb 1.0
Ty uexact(Ty x,Ty t){
return exp(-a*t*pi*pi)*sin(pi*x)-x+2;}

__host__ __device__ Ty u0(Ty x){
return sin(pi*x)-x + 2;}
__host__ __device__ Ty ua(Ty t){return 2;}
__host__ __device__ Ty ub(Ty t){return 1;}

#define n 50
#define N (n+1)
#define h ((xb-xa)/n)
#define tau (h*h/2/a)
```

```
#define tbeg 0
#define tend 0.5
#define K (floor((tend-tbeg)/tau)+1)
Ty *t=(Ty*)malloc(K*sizeof(Ty));
Ty *x=(Ty*)malloc(N*sizeof(Ty));
Ty *uk=(Ty*)malloc(N*sizeof(Ty));
Ty *ukp1=(Ty*)malloc(N*sizeof(Ty));

void mkrCPU(Ty*uk,Ty*ukp1,Ty*x,Ty*t){
int k,j;
for(k=0;k<=K-2;k++){for(j=1;j<=N-2;j++)
ukp1[j]=uk[j]+tau*a*(uk[j-1]-2*uk[j]+uk[j+1])/h/h;
ukp1[0]=ua(t[k+1]);   ukp1[N-1]=ub(t[k+1]);
for(j=0;j<=N-1;j++)uk[j]=ukp1[j]; }}

__global__ void mkrGPU1(Ty*uk,Ty*ukp1){
int j=threadIdx.x; if (j>0 && j<N-1)ukp1[j]=
uk[j]+tau*a*(uk[j-1]-2*uk[j]+uk[j+1])/h/h;
else if(j==0)ukp1[j]=ua(0); else ukp1[j]=ub(0);}

int main(){
double t1,t2;
for(int i=0;i<N;i++){
x[i]=xa+h*i; ukp1[i]=uk[i]=u0(x[i]);}
for(int i=0;i<K;i++)t[i]=tbeg+tau*i;

Ty *ukd,*ukp1d;
cudaMalloc(&ukd, N*sizeof(Ty));
cudaMalloc(&ukp1d, N*sizeof(Ty));
cudaMemcpy(ukd,uk,N*sizeof(Ty),
cudaMemcpyHostToDevice);
cudaMemcpy(ukp1d,ukp1,N*sizeof(Ty),
cudaMemcpyHostToDevice);
t1=double(clock())/CLOCKS_PER_SEC;
if(METHOD==0)mkrCPU(uk,ukp1,x,t);
else if(METHOD==1){
for(int i=1;i<=K/2;i++){
mkrGPU1<<<1,N>>>(ukd,ukp1d);
mkrGPU1<<<1,N>>>(ukp1d,ukd);}
cudaDeviceSynchronize();}
t2=double(clock())/CLOCKS_PER_SEC;
if(METHOD>0){cudaDeviceSynchronize();
cudaMemcpy(uk,ukd,N*sizeof(Ty),
cudaMemcpyDeviceToHost);}

printf("t1 = %lf, t2 = %lf Time = %lf\n",t1,t2,t2-t1);
printf("Umkr = %f  Uexact = %f\n",uk[N/2],
uexact(x[N/2],t[(int)(K-1)]));
```

```
FILE *fg,*ft;
char *dir = "c:/Figures/"; _mkdir(dir);
char fileg[40]; strcat(strcpy(fileg,dir),
"perenosg.gnu");
char filet[40]; strcat(strcpy(filet,dir),
"perenost.txt");
char gnuplot[80] ="\"c:/Program Files/gnuplot/bin/"
"gnuplot.exe\" "; strcat(gnuplot,fileg);

fg=fopen(fileg,"w+"); ft=fopen(filet,"w+");
for(int j=0;j<N-1;j++)fprintf(ft,"%10.5f %10.5f "
"%10.5f\n",x[j],uexact(x[j],tend),uk[j]);
fprintf(fg,"plot '%s' u 1:2 title \"exact\" with "
"lines lt rgb \"#FF0000\" lw 6, '%s' u 1:3 title "
"\"calc\" with points lt rgb \"#0000FF\" ps 0.3; "
"pause mouse keypress",filet,filet);
fclose(fg); fclose(ft);    system(gnuplot);
return 0;}
```

Решение уравнения переноса разностной противопоточной схемой (CUDA C)

$$\frac{\partial u}{\partial t}+\frac{\partial u}{\partial x}=f(x,t),\ f(x,t)=\frac{xt}{11},\ u(0,t)=1,\ u(x,0)=1-x,\ 0\le x\le 1,\ 0\le t\le 0{,}5,\ u(x,t)-?$$

```
#include<stdio.h>
#include<string.h>
#include<malloc.h>
#include<math.h>
#include<time.h>
#include<direct.h>
#include<cuda.h>
#include"cuda_runtime.h"
#include"device_launch_parameters.h"
const int METHOD=1;
typedef double Ty;
#define s 1.0/11
const Ty a=1;
const Ty xa=0,xb=1;
Ty uexact(Ty x,Ty t){if(x>=a*t)
return s*(x-a*t)*t*t/2+s*a*pow(t,3)/3+1-x+a*t;
else return s*pow(x,3)/3/a/a+s*(t-x/a)*x*x/(2*a)+1;}
__host__ __device__ Ty u0(Ty x){return 1-x;}
__host__ __device__ Ty ua(Ty t){return 1;}
__host__ __device__ Ty ub(Ty t){return 0;}
__host__ __device__ Ty f(Ty x,Ty t){return s*x*t;}

const int n=200, nplus=10, N=n+1+nplus;
const Ty c=1; // c<=1 Courant
const Ty h=(xb-xa)/n, tau=c*h/a, tbeg=0, tend=0.5;
```

```
const int K=floor((tend-tbeg)/tau)+1;
Ty *t=(Ty*)malloc(K*sizeof(Ty));
Ty *x=(Ty*)malloc(N*sizeof(Ty));
Ty *ukm1=(Ty*)malloc(N*sizeof(Ty));
Ty *uk=(Ty*)malloc(N*sizeof(Ty));
Ty *ukp1=(Ty*)malloc(N*sizeof(Ty));

__constant__ Ty H[1], TAU[1], A[1];
__constant__ int Nc[1], Kc[1];

void mkrCPU(Ty*uk,Ty*ukp1,Ty*x,Ty*t){
for(int k=1; k<K; k++) {for(int j=1;j<N;j++)
ukp1[j]=uk[j]-tau*a/h*(uk[j]-uk[j-1])+
tau*f(x[j-1],t[k-1]);
ukp1[0]=ua(0); for(int j=0;j<N;j++)uk[j]=ukp1[j];}}

__global__ void mkrGPU1(Ty*uk,Ty*ukp1,Ty*x,Ty*t){
for (int k=1;k<Kc[0];k++){
{int j=threadIdx.x;
if(j>0 && j<Nc[0]-1)
ukp1[j]=uk[j]-TAU[0]*A[0]/H[0]*(uk[j]-uk[j-1])+
TAU[0]*f(x[j-1],t[k-1]);
else if(j==0)ukp1[j]=ua(0);}
__syncthreads(); ukp1[Nc[0]-1]=0; int j=threadIdx.x;
if (j<Nc[0])uk[j]=ukp1[j];}}
int main(){
double t1,t2;
for(int i=0; i<N; i++){x[i]=xa+h*i; ukp1[i]=
ukm1[i]=uk[i]=u0(x[i]);}
for(int i=0; i<K; i++)t[i]=tbeg+tau*i;

Ty*ukd,*ukp1d,*xd,*td;
cudaMalloc(&ukd,N*sizeof(Ty));
cudaMalloc(&ukp1d,N*sizeof(Ty));
cudaMalloc(&xd,N*sizeof(Ty));
cudaMalloc(&td,K*sizeof(Ty));
cudaMemcpy(ukd,uk,N*sizeof(Ty),
cudaMemcpyHostToDevice);
cudaMemcpy(ukp1d,ukp1,N*sizeof(Ty),
cudaMemcpyHostToDevice);
cudaMemcpy(xd,x,N*sizeof(Ty),
cudaMemcpyHostToDevice);
cudaMemcpy(td,t,K*sizeof(Ty),
cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(H,&h,sizeof(Ty));
cudaMemcpyToSymbol(TAU,&tau,sizeof(Ty));
cudaMemcpyToSymbol(A,&a,sizeof(Ty));
```

```
cudaMemcpyToSymbol(Nc,&N,sizeof(int));
cudaMemcpyToSymbol(Kc,&K,sizeof(int));

t1=double(clock())/CLOCKS_PER_SEC;
if(METHOD==0)mkrCPU(uk,ukp1,x,t);
else if(METHOD==1)mkrGPU1<<<1,N>>>(ukd,ukp1d,xd,td);
t2=double(clock())/CLOCKS_PER_SEC;
printf("Time = %lf\n",t2-t1);
if(METHOD>0)cudaMemcpy(uk,ukd,N*sizeof(Ty),
cudaMemcpyDeviceToHost);
printf("Umkr = %f, Uexact = %f",uk[N/2],
uexact(x[N/2],t[K-1]));

FILE *fg,*ft;
char *dir="c:/Figures/";
_mkdir(dir);
char fileg[40]; strcat(strcpy(fileg,dir),
"perenosg.gnu");
char filet[40]; strcat(strcpy(filet,dir),
"perenost.txt");
char gnuplot[80]="\"c:/Program Files/gnuplot/bin/"
"gnuplot.exe\" "; strcat(gnuplot,fileg);
fg=fopen(fileg,"w+");ft=fopen(filet,"w+");
for(int j=0;j<N-1;j++)fprintf(ft,"%10.5f %10.5f "
"%10.5f\n",x[j],uexact(x[j],tend),uk[j]);
fprintf(fg,"plot '%s' u 1:2 title \"exact\" with "
"lines lt rgb \"#FF0000\" lw 6, '%s' u 1:3 title "
"\"calc\" with points lt rgb \"#0000FF\" ps 0.3; "
"pause mouse keypress",filet,filet);
fclose(fg); fclose(ft);    system(gnuplot);
return 0;}
```

Пример сложения двух векторов в CUDA Fortran. Чтобы иметь возможность программировать в CUDA Fortran, необходима установка Visual Studio 2017 с Visual C++ и Windows 10 SDK. Затем устанавливаем PGI Community. После этого создаём через Проводник файл C:\Files\ test_cuda_fortran.cuf с текстом программы. Файл создаётся текстовый, но расширение меняем на «.cuf». Данный файл можно открывать и редактировать Блокнотом или, например, бесплатным редактором-блокнотом Notepad++ и т.п. Далее для запуска компиляторов PGI можно создать ярлык на Рабочем столе и указать расположение объекта:

```
C:\Windows\System32\cmd.exe /c "C:\Program Files\PGI\win64\18.10\
pgi_dos.bat"
```

Запускаем компилятор, вводим команду

cd c:\Files, затем, pgf90 -Mcuda=cc35 -o test_cuda_fortran.exe test_cuda_fortran.cuf & test_cuda_fortran.exe
где вместо cc35 нужно подставить номер, который зависит от вычислительных возможностей конкретной видеокарты, например, cc30.

Текст программы test_cuda_fortran.cuf (сумма двух векторов)

```
module modsumGPU
contains
attributes(global) subroutine sumGPU(a, b, c)
integer, intent(inout) :: a(:), b(:), c(:)
integer :: i, n
i = blockDim%x*(blockIdx%x-1) + threadIdx%x
n = size(a);
if (i <= n) c(i) = a(i) + b(i)
end subroutine sumGPU
end module modsumGPU
program MAIN
use cudafor
use modsumGPU
use omp_lib
integer, parameter :: n = 10*1000
integer, allocatable :: a(:), b(:), c(:)
integer, device, allocatable :: ad(:), bd(:), cd(:)
real :: time,t1,t2,tt1,tt2
integer:: S = 0
allocate(a(n), b(n), c(n), ad(n), bd(n), cd(n))
a = 1; b = 2
ad = a; bd = b;
t1=omp_get_wtime()
call CPU_TIME(tt1)
call sumGPU<<<10,1000>>>(ad, bd, cd)
call CPU_TIME(tt2)
t2=omp_get_wtime()
c = cd
write(*,*) <omp_get_wtime()>,t1,t2,t2-t1
write(*,*) <call cpu_time()>,tt1,tt2,tt2-tt1
do i=1,n
 S=S+c(i)s
enddo
print*,3*n,S
deallocate(a, b, c, ad, bd, cd)
END program MAIN
```

3. OPENACC

Стандарт OpenACC (**Open Acc**elerators) был анонсирован ноябре 2011 года — это совместная разработка суперкомпьютерных гигантов CRAY, CAPS, PGI, NVIDIA. Сам стандарт призван значительно упростить работу программиста и создать высокоуровневую прослойку над уже известными CUDA и OpenCL.

Данный стандарт, например, поддерживается бесплатной версией компилятора PGI (версия Community). Написание программы, выполняемой параллельно на тысячах ядер современных GPU, не требует больших усилий и практически полностью перекладывается на компилятор. Всё что нужно сделать — расставить директивы в коде программы аналогично OpenMP.

OpenACC поддерживает языки C/C++ и Fortran. Все директивы в C/C++ версии стандарта начинаются обычно с *#pragma acc*, далее ставится спецификатор "acc" и одна из основных директив, дополненная одним или несколькими условиями. Чаще всего используются директивы *parallel, kernels* и *data*. Для Fortran это *!$acc*.

Рассмотрим некоторые директивы:

• Директива *parallel* указывает на необходимость распараллеливания. Компилятор, проводя анализ кода, определяет необходимость исполнения различных его частей на GPU или на хосте.

• Директива *kernels* — аналог *parallel*, указывает на то, что для каждого нового цикла необходимо создать отдельную *__device__* функцию.

• Директива *loop* предшествует оператору цикла и используется для спецификации его свойств. Современные компиляторы не требуют её явного указания.

Несмотря на всю мощь компилятора, который старается всё взять на себя, иногда нужно подсказывать, какие данные необходимо передать с хоста на устройство и обратно, а так как копирование данных — часто самое узкое место в производительности, то нужно заранее продумать, где и как оптимизировать доступ к данным. Условия передачи данных требуют входные данные, например, OpenACC C/C++ это *a[start:length]*, где *a* — массив, или указатель на него; *start* — номер стартового элемента для копирования, а *length* –длина региона данных, копируемого на GPU или с него; *start* и *length* указываются в элементах массива. Эти условия можно использовать только с директивами *kernels, parallel* и *data region*. Наиболее часто используемые из них: *copyin* — указывает, что данные на GPU используются только для чтения, и нет необходимости копировать их обратно на хост, *copyout* — данные появятся только в результате выполнения ядра на GPU и никак не зависят от предыдущих значений по этому адресу, их нужно скопировать на хост после выполнения кернела, *create* — выделяет в памяти устройства место для данных, не требующих какого-либо копирования, например, массив для хранения промежуточных результатов.

Некоторые стандартные функции библиотек C/C++ и Fortran имеют свою реализацию в OpenACC: *acc_get_device_num()*, *acc_async_wait()*, *acc_async_wait_all()*, *acc_on_device()*, *acc_malloc()*, *acc_free()* и т.д.

OpenACC прост и неприхотлив в использовании, он очень сильно напоминает OpenMP и задумывался как ответвление OpenMP. К достоинствам OpenACC можно отнести также высокую степень абстракции и кроссплатформенность — сразу после выхода новых архитектур необязательно переписывать весь код, большую часть компилятор сделает за нас.

Итак, OpenACC даёт возможность быстро переписать свои проекты под использование GPU и практически не требует навыков их программирования. С его помощью уже ускорены десятки проектов в областях изучения и прогнозирования поведения атмосферы, аэро- и гидродинамики, и статистических расчётов. На сегодня OpenACC — один из лучших способ получить высокую производительность, не вникая во все тонкости программирования в CUDA или OpenCL.

Пример умножения двух матриц (OpenACC C)

Необходима установка Visual Studio 2017 Community, Professional, или Enterprise edition с Visual C++ и the Windows 10 SDK. Затем PGI Community, бесплатная лицензия на который даётся на 1 год. На данный момент PGI Community в операционной системе Windows не поддерживает C++, а только C. Необходимые шаги можно узнать на сайте https://www.pgroup.com/

Итак, создаём через Проводник файл C:\Files\test_openacc_c.c с текстом программы. Файл создаётся текстовый, но расширение меняем на «.c». Данный файл можно открывать и редактировать Блокнотом или, например, бесплатным редактором Notepad++ и т.п.

Далее для запуска компиляторов PGI можно создать ярлык на Рабочем столе и указать расположение объекта:

```
C:\Windows\System32\cmd.exe /c "C:\Program Files\PGI\win64\18.10\
pgi_dos.bat"
```

Запускаем компилятор, вводим команду cd c:\Files, затем, если хотим использовать OpenACC, вводим

```
pgcc -Minfo=accel -acc -ta=nvidia,cc35 -o c:\Files\test_openacc_c.
exe c:\Files\ test_openacc_c.c & c:\ Files\test_openacc_c.exe
```

или без OpenACC, а только PGI C, то (сокращая длинные пути)

```
pgcc -o test_openacc_c.exe test_openacc_c.c & test_openacc_c.
exe
```

Текст программы test_openacc_c.c (произведение двух матриц)

```
#include <openacc.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
typedef double T;
void main(){
const int n=1000;
T dn=(T)n, S=0;
T a[n][n], b[n][n], c[n][n];float t1,t2;
t1=(float)clock()/CLOCKS_PER_SEC;
#pragma acc kernels loop independent if(1)for(int i=0;i<n;i++)
for(int j=0;j<n;j++)
{a[i][j]=b[i][j]=(i+1+j+1)/dn/dn; c[i][j]=0;}
t2=(float)clock()/CLOCKS_PER_SEC;
printf("%f %f %f\n",t1,t2,t2-t1);t1=(float)clock()/CLOCKS_PER_
SEC;
```

```
  #pragma acc kernels loop independent if(1)
  for(int i=0;i<n;i++) for (int j=0;j<n;j++)
  for(int k=0;k<n;k++) c[i][j]+=a[i][k]*b[k][j];
  t2=(float)clock()/CLOCKS_PER_SEC;printf("%f  %f
%f\n",t1,t2,t2-t1);t1=(float)clock()/CLOCKS_PER_SEC;#pragma acc
kernels loop independent if(1)
  for(int i=0;i<n;i++) for(int j=0;j<n;j++) S+=c[i][j];
  t2=(float)clock()/CLOCKS_PER_SEC;printf("%f %f %f\n",t1,t2,t2-t1);
  printf("%f\n", S); printf("%f\n",dn); }
```

Пример сложения двух больших векторов (OpenACC Fortran)

Компиляция и запуск программы осуществляется аналогично OpenACC C

```
  pgf90 -Minfo=accel -acc -ta=nvidia,cc35 -o test_openacc_fortran.
exe test_openacc_fortran.f90 & test_openacc_fortran.exe
```

Код файла test_openacc_fortran.f90

```
  use omp_lib
  real :: t1, t2
  integer:: n=100000000
  integer, allocatable:: a(:), b(:), c(:)
  allocate(a(n), b(n),c(n))
  !$acc data create(a,b,c) copyout(c)
  !$acc parallel
  a(:)=1; b(:)=2;
  !$acc end parallel
  t1 = omp_get_wtime()
  !$acc parallel loop
  do i=1,n
  c(i)=a(i)+b(i)
  enddo
  !$acc end parallel loop
  t2 = omp_get_wtime()
  !$acc end data
  print*, n*3, sum(c), c(1)
  print '(a,f15.6,a)', 'Time: ', t2 - t1, 'secs'
  END
```

4. OPENCL

OpenCL — фреймворк для написания компьютерных программ, связанных с параллельными вычислениями на различных графических и центральных процессорах. В OpenCL входят язык программирования, который основан на стандарте языка программирования C99, C++11, и интерфейс API. OpenCL обеспечивает параллелизм на уровне инструкций и на уровне данных и является осуществлением техники GPGPU. OpenCL является полностью открытым стандартом, его использование не облагается лицензионными отчислениями. OpenCL разрабатывается и поддерживается некоммерческим консорциумом Khronos Group, в который входят многие крупные компании, включая AMD, Apple, ARM, Intel, Nvidia и др.

OpenCL — это технология, связанная с параллельными компьютерными вычислениями на различных типах графических и центральных процессоров. OpenCL позволяет работать как с CPU, так и с GPU, но обычно его используют в работе с GPU. Для использования данной технологии понадобится почти любая относительно современная видеокарта.

При помощи API получаем доступ к устройствам, которые поддерживают OpenCL. Это часть приложения называется хостом. Затем пишем код, который будет выполняться на устройстве. Такой код называется *kernel*. При помощи API прогружаем код *kernel* и запускаем его выполнение на выбранном устройстве.

Для работы в Visual Studio с OpenCL необходимо установить (обновить) драйвера видеокарт (AMD, Nvidia), а затем AMD SDK или всё та же CUDA Toolkit. После этого создаём обычный консольный проект для C/C++. В свойствах проекта выбираем (Если Платформа решений — x86), далее:

1) C/C++ > General в Additional Include Directories добавляем C:\Program Files (x86)\AMD APP SDK\3.0\include;

2) Linker > General в Additional Library Directories добавляем C:\Program Files (x86)\AMD APP SDK\3.0\lib\x86;

3) Linker > Input в Additional dependencies добавляем opencl.lib;

или:

1) C/C++ > General в Additional Include Directories добавляем C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.2\include;

2) Linker > General в Additional Library Directories добавляем C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.2\lib\Win32;

3) Linker > Input в Additional dependencies добавляем opencl.lib;

Пример сложения двух одномерных массивов в OpenCL

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <math.h>
#include <time.h>
#define __CL_ENABLE_EXCEPTIONS
#include <CL/cl.hpp>
#define T int
using namespace std;
```

```
void sumCPU(T *a, T *b, T *c, int n){
for (int i=0; i<n; i++) c[i]=a[i]+b[i];}

const char *kernelSource="                    \n"\
"#define T int                                \n"\
"#define GT __global T                        \n"\
"__kernel void sumGPU1(GT*a,GT*b,GT*c,int n){\n"\
"int id=get_global_id(0);                     \n"\
"if(id<n) c[id]=a[id]+b[id];                  \n"\
"}";
int main(){
double t1, t2;
T sum;
unsigned int n=50000000;
T *ah, *bh, *ch;
cl::Buffer ad, bd, dc;
size_t bytes=n*sizeof(T);
ah=new T[n]; bh=new T[n]; ch=new T[n];
for(int i=0; i<n; i++) {ah[i]=1; bh[i]=2;}
cl_int err=CL_SUCCESS;
try{
std::vector<cl::Platform> platforms;
cl::Platform::get(&platforms);
if (platforms.size()==0){
std::cout<<"Platform size 0\n"; return -1;}
cl_context_properties properties[]=
{CL_CONTEXT_PLATFORM,
(cl_context_properties)(platforms[0])(),0};
cl::Context context(CL_DEVICE_TYPE_GPU, properties);
std::vector<cl::Device> devices=context.getInfo
<CL_CONTEXT_DEVICES>();
cl::CommandQueue queue(context, devices[0], 0, &err);
ad=cl::Buffer(context, CL_MEM_READ_ONLY, bytes);
bd=cl::Buffer(context, CL_MEM_READ_ONLY, bytes);
dc=cl::Buffer(context, CL_MEM_WRITE_ONLY, bytes);
queue.enqueueWriteBuffer(ad, CL_TRUE, 0, bytes, ah);
queue.enqueueWriteBuffer(bd, CL_TRUE, 0, bytes, bh);
cl::Program::Sources source(1,
std::make_pair(kernelSource,strlen(kernelSource)));
cl::Program program=cl::Program(context, source);
program.build(devices);
cl::Kernel kernel(program, "sumGPU1", &err);
kernel.setArg(0, ad); kernel.setArg(1, bd);
kernel.setArg(2, dc); kernel.setArg(3, n);
cl::NDRange localSize(64),globalSize(n);
cl::Event event;
t1=double(clock())/CLOCKS_PER_SEC;
```

```
  queue.enqueueNDRangeKernel(kernel,cl::NullRange,
  globalSize,localSize,NULL,&event);
  event.wait();
  t2=double(clock())/CLOCKS_PER_SEC;queue.enqueueReadBuffer(dc,
CL_TRUE, 0, bytes, ch);}
  catch (cl::Error e){std::cerr<<"error "<<e.what()
  <<"("<<e.err()<<")"<<std::endl;}

  sum=0; for(int i=0; i<n; i++) sum+=ch[i];
  cout <<"Sum(c) GPU = "<<sum <<" Time GPU t1 = "<<
  t1<<" t2 = " <<t2<<" t2-t1= "<<t2-t1<<endl;t1=double(clock())/
CLOCKS_PER_SEC;
  sumCPU(ah,bh,ch,n);
  t2=double(clock())/CLOCKS_PER_SEC;
  sum=0; for(int i=0; i<n; i++) sum+=ch[i];
  cout<<"Sum(c) CPU = "<<sum<<" Time CPU t1 = "<<
  t1<<" t2 = "<<t2<<" t2-t1= "<<t2-t1<<endl;
  delete(ah,bh,ch);
  cout<<"Sum(c) Exc = "<<(T)3*n<<endl;
  cout<<kernelSource<<endl; return 0; }
```

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Строительная информатика. — 3-е изд. / П.А. Акимов, Т.Б. Кайтуков, М.Л. Мозгалева, В.Н. Сидоров. — Москва : АСВ, 2018. — 432 с. — ISBN 978-5-4323-0066-9.

2. Информатика: учебник для вузов / А.Б. Золотов, П.А. Акимов, В.Н. Сидоров, М.Л. Мозгалева. — Москва : АСВ, 2013. — 404 с. — ISBN 978-5-93093-982-8.

3. Дискретно-континуальный метод конечных элементов. Приложения в строительстве / А.Б. Золотов, П.А. Акимов, В.Н. Сидоров, М.Л. Мозгалева. — Москва : АСВ, 2010. — 336 с. — ISBN 978-5-93093-753-4.

4. Орлов С.А. Организация ЭВМ и систем: учебник для вузов : 2-е изд. / С.А. Орлов, Б.Я. Цилькер. — Санкт-Петербург : Питер, 2011. — 688 с. — ISBN 978-5-49807-862-5.

5. Мелехин В.Ф. Вычислительные машины, системы и сети: учебник для вузов. 3-е изд., стер. — Москва : Академия, 2010. — 560 с. — ISBN 978-5-7695-5840-5.