СОПОСТАВЛЕНИЕ ПРОИЗВОДИТЕЛЬНОСТЕЙ GPU И CPU ДЛЯ МАТРИЧНОГО УМНОЖЕНИЯ С ДВОЙНОЙ ТОЧНОСТЬЮ

А. А. Мясищев
Украина, г. Хмельницкий, Хмельницкий национальный университет
alex@tup.km.ua

Матричные вычисления составляют основу многих научных и инженерных расчетов. Например, для решения задач расчета на прочность, исследования процессов пластического течения материалов часто используется метод конечных элементов, который сводится к решению систем линейных уравнений и матричным вычислениям. В связи с этим многие стандартные библиотеки программ содержат процедуры для различных матричных операций. Вследствие своей вычислительной трудоемкости и возможности эффективного распараллеливания, матричные вычисления представляют собой классическую область для параллельных вычислений.

Сопоставим эффективность использования для расчета произведения квадратных матриц для **двойной точности** вычислительную систему, в которой установлены 6-тиядерный процессор AMD Phenom II X6 1090T (CPU) с видеокартой NVIDIA GeForce GTX 480 и вычислительную систему с таким же процессором, но с графическим процессором (GPU) NVIDIA Tesla C2075. Расчет будет выполняться для 3-х случаев. В первом случае используется только процессор (CPU), для которого будет выполняться распараллеливание по 6-ядрам с использованием библиотек ScaLAPACK и библиотек ATLAS (автоматически настраиваемое программное обеспечение для решения задач линейной алгебры). Библиотеки ATLAS при компиляции настраиваются под конкретную архитектуру процессора вычислительной системы [1,2]. Во втором случае расчет будет выполняться на видеокарте NVIDIA GeForce GTX 480 с использованием технологии CUDA [1]. В третьем – на графическом процессоре NVIDIA Tesla C2075. Вычислительная система настроена на работу с 32-х разрядной операционной системой Linux Ubuntu ver. 10.10 desktop. В системе установлены компилятор FORTRAN F77, библиотеки MPI [3], ScaLAPACK [1] и ATLAS [2] для 6-тиядерного процессора. Для программирования на NVIDIA инсталлируются видеодрайвер nvidia и программное обеспечение с сайта http://developer.nvidia.com/cuda-toolkit-archive. Последовательность установки библиотек MPI, ScaLAPACK, ATLAS под ОС Linux Ubuntu ver. 9.04 desktop подробно представлена в работе [4] для системы с 4-х ядерным процессором CORE 2 QUAD PENTIUM Q6600 2.4GHZ, поэтому здесь не рассматривается. Рассмот-

рим более подробно установку программного обеспечения для NVIDIA CUDA.

1. Копируем видеодрайвер NVIDIA с сайта http://www.nvidia.ru/Download/index.aspx?lang=ru для 32-х разрядной ОС Linux.

2. Открываем файл /etc/modprobe.d/blacklist.conf и добавляем в него модули свободных драйверов vga16fb, nouveau, rivafb, nvidiafb, rivatv.

3. Устанавливаем необходимые пакеты

```
sudo apt-get install linux-headers-`uname -r` binutils pkg-config build-essential xserver-xorg-dev
```

4. Далее переходим в параллельно работающую консоль и вводим свои данные (логин и пароль). Остановим X-сервер командой:

```
sudo /etc/init.d/gdm stop
```

и запускаем установку драйвера

```
sudo sh ./NVIDIA-Linux-x86-290.10.run
```

После ответа на ряд вопросов драйвер должен установиться. Далее запускаем графический сервер

```
sudo service gdm start && exit
```

Выполняем команду

```
sudo apt-get --purge remove nvidia-*
```

и перегружаем компьютер

```
reboot
```

5. С сайта http://developer.nvidia.com/cuda-toolkit-40 копируем для Linux Ubuntu 10.10, 32-и разрядной версии CUDA Toolkit и GPU Computing SDK. Устанавливаем их командами:

```
sudo sh ./cudatoolkit_4.0.17_linux_32_ubuntu10.10.ru
sudo sh ./gpucomputingsdk_4.0.17_linux.run
```

6. Выполняем настройку системы

```
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib:$LD_LIBRARY_PATH
echo 'export PATH=/usr/local/cuda/bin:$PATH' >> ~/.bashrc &&
echo 'export LD_LIBRARY_PATH=
/usr/local/cuda/lib:$LD_LIBRARY_PATH' >> ~/.bashrc
```

и устанавливаем дополнительные пакеты

```
sudo apt-get install g++ freeglut3-dev libxi-dev libxmu-dev
```

7. Для правильной компиляции всех примеров может возникнуть необходимость выполнения команды (или подобных)

```
sudo ln -s /usr/lib/libGL.so.1 /lib/libGL.so
```

8. Выполняем компиляцию примеров

```
cd ~/NVIDIA_GPU_Computing_SDK/C && make
```

9. Для тестирования правильности установки запускаем один из примеров

```
cd ~/NVIDIA_GPU_Computing_SDK/C/bin/linux/release&&./deviceQuery
```

На консоли должны появиться основные возможности GPU.

Рассмотрим программы расчета произведения квадратных запол-

ненных матриц для NVIDIA GPU. В первом случае для простоты примем, что элементы матриц хранятся в глобальной памяти GPU. Для составления даже этой простой программы необходимо подробнее рассмотреть программную модель GPU для компилятора Си [5]. Верхний уровень ядра, называемый сеткой (grid), состоит из блоков. В свою очередь блоки представляют собой либо одноразмерную, либо двухразмерную сеть нитей. Вышесказанное может быть проиллюстрировано рис. 1. Номер нити в блоке или номер блока в grid синтаксически специфицируется оператором <<<…>>> и может быть переменной типа int или dim3.

Число нитей, входящих в блок, определяется встроенной переменной blockDim. Индекс нити внутри блока определяется переменной threadIdx, а индекс блока внутри grid – переменной blockIdx. Нити в блоке являются непосредственными исполнителями вычислений. Нить является 3-х компонентным вектором, т.е. может идентифицироваться, используя одноразмерный, двухразмерный и трехразмерный индекс нити, образуя в свою очередь одноразмерный, двух размерный и трех размерный блок нитей. Существует ограничение количества нитей на один блок, которое не может превышать 1024 нити.

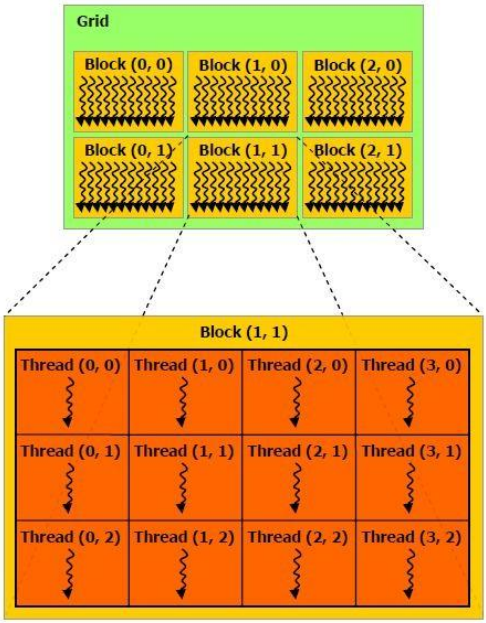

Рис. 1

Расширение Си CUDA позволяет вызвать функцию, называемую ядром так, что она будет параллельно выполнять N разных нитей CUDA. Такие функция декларируется спецификатором global . В качестве иллюстрации ниже рассмотрен пример распечатки функцией ядра трех раз слова hello=n (n-номер нити), т.к. в функции hello<<<1, 3>>>(); указан запрос на работу одного блока и трех нитей.

```
#include <stdio.h>
__global__void hello() {
  int i=threadIdx.x; printf("Hello=%d\n",i);
}
__host__int main() {  hello<<<1, 3>>>();  cudaThreadExit(); }
```

Компилируем командой [5]

```
nvcc -arch=sm_20 aa1.cu -o aa1
```

Запускаем

```
./aa1
Hello=0
Hello=1
Hello=2
```

Если задействовать работу еще и 2-х блоков, то программа будет иметь вид:

```
#include <stdio.h>
__global__void hello() {
  int i=threadIdx.x, b=blockIdx.x;
  printf("Hello block=%d, thread=%d\n",b,i);
}
__host__int main() {
  hello<<<2, 3>>>();
  cudaThreadExit();
}
```

Результатом ее работы будет следующий листинг:

```
Hello block=0, thread=0
Hello block=0, thread=1
Hello block=0, thread=2
Hello block=1, thread=0
Hello block=1, thread=1
Hello block=1, thread=2
```

Рассмотренные программы являются простейшими, т.к. передачи данных между памятью CPU и GPU не происходит.

Рассмотрим более сложные программы произведения квадратных заполненных матриц, которые и будут использованы для сопоставления производительности CPU и GPU. Программы разбиты на три класса в соответствии с типом используемой памяти на GPU и библиотеки расчета произведения [5]:

1. При расчете произведения матрицы размещаются в глобальной памяти GPU.

2. Матрицы размещаются в разделяемой памяти GPU.

3. Расчет произведения выполняется подпрограммой cublasDgemm библиотеки cuBLAS.

Первая программа. Предполагается, что матрицы a, b и c размещаются в одномерных массивах. Ниже представлен текст программы с пояснениями:

```
#include <stdio.h>
#define BLOCK 16
// Функция умножения двух матриц
__global  void mulMatr(double* a,double* b,double* c,int n) {
  //Получаем id текущей нити.
```

```
  int i = threadIdx.y+blockIdx.y*blockDim.y;
  int j = threadIdx.x+blockIdx.x*blockDim.x;
  //Расчитываем результат.
  double sum=0.0;
  for(int p = 0; p < n; p++) sum+=a[i*n + p] * b[p*n + j];
  c[i*n+j] =sum;
}
__host__int main() {
  int N, M;
  double mf=0.0;
  printf ( "Input N->");
  scanf ( "%d",&N);
  printf ( "Matrix = %dx%d elements\n", N,N );
  M=N*N;
#define shorter() (double*)malloc(sizeof(double)*M)
  //Выделяем память под вектора
  double* a = shorter(), * b = shorter(), * c = shorter();
  //Инициализируем значения векторов
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      a[i*N+j] = 1.0*((i+1)+2*(j+1)), b[i*N+j] = 1.0/a[i*N+j];
  //Указатели на память в видеокарте
  double* deva, * devb, * devc;
  //Выделяем память для векторов на видеокарте
  cudaMalloc((void**)&deva, sizeof(double) * M);
  cudaMalloc((void**)&devb, sizeof(double) * M);
  cudaMalloc((void**)&devc, sizeof(double) * M);
  cudaEvent_t start, stop;
  float gpuTime=0.0;
  cudaEventCreate ( &start );  cudaEventCreate ( &stop );
  //Копируем данные в память видеокарты
  cudaMemcpy(deva,a,sizeof(double)*M, cudaMemcpyHostToDevice);
  cudaMemcpy(devb, b, sizeof(double)*M, cudaMemcpyHostToDevice);
  //Выполняем вызов функции ядра
  dim3 threads = dim3(BLOCK,BLOCK);
  dim3  blocks = dim3(N/BLOCK,N/BLOCK);
  cudaEventRecord(start, 0); // привязываем событие к началу
                             // выполнения ядра
  mulMatr<<<blocks, threads>>>(deva, devb, devc,N);
  cudaEventRecord(stop, 0); // привязываем событие к концу
                            //выполнения ядра
  //Получаем результат расчета
  cudaMemcpy(c, devc, sizeof(double)*M, cudaMemcpyDeviceToHost);
  cudaEventSynchronize(stop); //Дожидаемся выполнение
                           //ядра, синхронизируя по событию stop
  cudaEventElapsedTime (&gpuTime,start,stop);//Запрашиваем время
                                             //между start, stop
  //Результаты расчета
  gpuTime=gpuTime/1000.0;
```

```
  mf=((2.0*N-1)*N*N)/(gpuTime*1000000.0);
  printf("time=%.14fsec\nspeed=%0.2fMFlops\n",gpuTime,mf);
  printf("i=%d\tj=%d\tC=%.15f\n",
              N/256,N/128,c[(N/256)*N+(N/128)]);
  printf("i=%d\tj=%d\tC=%.15f\n",
              3*N/4,5*N/16,c[(3*N/4)*N+(5*N/16)]);
  printf("i=%d\tj=%d\tC=%.15f\n",N-4,N-2,c[(N-4)*N+(N-2)]);
  // Высвобождаем ресурсы
  cudaEventDestroy(start);   cudaEventDestroy(stop);
  cudaFree(deva);  cudaFree(devb);  cudaFree(devc);
  cudaThreadExit();
  delete[] a; a = 0;  delete[] b; b = 0;  delete[] c; c = 0;
}
```

Вторая программа. В функции ядра используется разделяемая память. Здесь принимается, что из-за малого размера разделяемой памяти, исходные квадратные матрицы разбиваются на блочные матрицы размером 16x16 элементов (BLOCK_SIZE 16), как в работе [5].

```
#include <stdio.h>
#define BLOCK_SIZE 16
__global void matrixMul( double* C, double* A, double* B, int
wA, int wB) {
Текст представленной программы, которая выполняется на    GPU
( global   void matrixMul), соответствует тексту  программы-
примера,       которая       поставляется       с       сайта
http://developer.nvidia.com/cuda-toolkit-32-downloads под именем
matrixMul_kernel.cu.
}
__host__int main() {
  int N, M;
  double mf=0.0f;
  printf ( "Input N->");
  scanf ( "%d",&N);
  printf ( "Matrix = %dx%d elements\n", N,N );
  M=N*N;
  //Выделяем память под вектора
  double* a = new double[M], * b = new double[M];
  double* c = new double[M];
  //Инициализируем значения векторов
  for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      a[i*N+j] = 1.0*((i+1)+2*(j+1)), b[i*N+j] = 1.0/a[i*N+j];
  //Указатели на память в видеокарте
  double* deva, * devb, * devc;
  //Выделяем память для векторов на видеокарте
  cudaMalloc((void**)&deva, sizeof(double) * M);
  cudaMalloc((void**)&devb, sizeof(double) * M);
  cudaMalloc((void**)&devc, sizeof(double) * M);
  cudaEvent_t start, stop;
```

```
  float gpuTime=0.0f;
  cudaEventCreate ( &start ); cudaEventCreate ( &stop );
  //Копируем данные в память видеокарты
  cudaMemcpy(deva, a, sizeof(double)*M, cudaMemcpyHostToDevice);
  cudaMemcpy(devb, b, sizeof(double)*M, cudaMemcpyHostToDevice);
  //Выполняем вызов функции ядра
  dim3 threads = dim3(BLOCK_SIZE,BLOCK_SIZE);
  dim3  blocks = dim3(N/BLOCK_SIZE,N/BLOCK_SIZE);
  cudaEventRecord(start, 0); // привязываем событие
                             //к началу выполнения ядра
  matrixMul<<<blocks, threads>>>(devc, deva, devb,N,N);
  cudaEventRecord(stop, 0); // привязываем событие
                            //к концу выполнения ядра
  //Получаем результат расчета
  cudaMemcpy(c, devc, sizeof(double)*M, cudaMemcpyDeviceToHost);
  cudaEventSynchronize(stop); //Дожидаемся выполнение
                          // ядра, синхронизируя по событию stop
  cudaEventElapsedTime (&gpuTime,start,stop);//Запрашиваем время
                                             //между start, stop
  //Результаты расчета
  gpuTime=gpuTime/1000;
  mf=((2.0*N-1)*N*N)/(gpuTime*1000000.0);
  printf("time=%.4fsec\nspeed=%0.2fMFlops\n",gpuTime,mf);
  printf("i=%d\tj=%d\tC=%.5f\ni=%d\tj=%d\tC=%.5f\n",
              N/256,N/128,c[(N/256)*N+(N/128)],
              3*N/4,5*N/16,c[(3*N/4)*N+(5*N/16)]);
  printf("i=%d\tj=%d\tC=%.5f\n",N-4,N-2,c[(N-4)*N+(N-2)]);
  // Высвобождаем ресурсы
  cudaEventDestroy(start);  cudaEventDestroy(stop);
  cudaFree(deva); cudaFree(devb); cudaFree(devc);
  delete[] a; a = 0;  delete[] b; b = 0;  delete[] c; c = 0;
}
```

Третья программа. Для перемножения используется б-ка CUBLAS.

```
#include <stdio.h>
#include <cublas.h>
int main ( int argc, char** argv ) {
  float time_seconds=0.0f, mf=0.0f;
  int N;
  cudaEvent_t start, stop;
  cudaEventCreate ( &start ); cudaEventCreate ( &stop );
  printf ( "Input N->");
  scanf ( "%d",&N);
  printf ( "Matrix = %dx%d elements\n", N,N );
  int M=N*N;
double *d_A=new double[M],*d_B=new double[M],*d_C=new double[M];
  for (int j = 0; j < N; j++)
    for (int i = 0; i < N; i++)
      A[i+j*N] = 1.0*((i+1)+2*(j+1)), B[i+j*N] = 1.0/A[i+j*N];
  cublasInit();
```

```
  cublasAlloc ( N * N, sizeof(double), (void**)&d_A);
  cublasAlloc ( N * N, sizeof(double), (void**)&d_B);
  cublasAlloc ( N * N, sizeof(double), (void**)&d_C);
  cublasSetMatrix(N,N,sizeof(double),(void*)A,N,(void*)d_A, N);
  cublasSetMatrix(N,N,sizeof(double),(void*)B,N,(void*)d_B, N);
  cudaEventRecord(start, 0);
  cublasDgemm('n','n',N,N,N,1.0,d_A, N, d_B, N, 0.0f, d_C, N );
  cudaEventRecord(stop, 0); cudaEventSynchronize(stop);
  cudaEventElapsedTime (&time_seconds,start,stop);
  cublasGetMatrix(N,N,sizeof(double),(void*)d_C,N,(void*)C,N);
  cublasFree (d_A); cublasFree (d_B); cublasFree (d_C);
  cublasShutdown();
  time_seconds=time_seconds/1000;
  mf=((2.0*N-1)*N*N)/(time_seconds*1000000.0);
  printf("time=%.4fsec\nspeed=%0.2fMFlops\n",time_seconds,mf);
  printf("i=%d\tj=%d\tC=%.5f\n",
       N/256,N/128,C[(N/256)+(N/128)*N]);
  printf("i=%d\tj=%d\tC=%.5f\n",
       3*N/4,5*N/16,C[(3*N/4)+(5*N/16)*N]);
  printf("i=%d\tj=%d\tC=%.5f\n",N-4,N-2,C[(N-4)+(N-2)*N]);
}
```

В таблице 1 сведены результаты расчетов по представленным выше программам для GPU при двойной точности вычислений. Также представлены результаты и для CPU с использованием библиотек ScaLAPACK и ATLAS по методу, изложенному в работе [4]. Результаты представлены в виде дроби: числитель – время счета в секундах, знаменатель – производительность в Гигафлопсах в секунду. Производительность определялась как отношение числа операций с плавающей точкой при матричном произведении к затраченному времени. Число операций в программах определяется по выражению: op=N*N(2*N-1), где N – число строк или столбцов квадратной матрицы.

Таблица 1

МатNxN	CPU 6 ядер	GPU GeForce GTX 480			GPU Tesla C2075		
		Глоб.	Разд.	cublas	Глоб.	Разд.	cublas
1024	0.062 34.6	0.034 61.9	0.018 122.1	0.014 155.7	0.045 47.3	0.022 96.0	0.008 284.4
2048	0.407 42.2	0.282 60.8	0.140 123.0	0.105 163.0	0.375 45.9	0.178 96.4	0.058 295.8
3072	1.234 47.0	0.919 63.1	0.471 123.1	0.352 164.6	1.233 47.0	0.600 96.7	0.194 299.2
4096	2.870 47.9	2.213 62.1	1.141 120.5	0.835 164.5	2.963 46.4	1.455 94.5	0.458 300.1
5120	5.687 47.2	4.253 63.1	2.181 123.0	1.626 165.1	5.715 47.0	2.781 96.5	0.893 300.5
6144	-	7.482 62.0	3.771 123.0	2.810 165.1	10.025 46.3	4.805 96.5	1.543 300.6

Выводы: 1. Производительность GPU GeForce GTX 480 в 3.5 раза выше производительности CPU AMD Phenom II X6 1090T, а производительность GPU Tesla C2075 в 6.3 раза выше CPU при использовании библиотеки CUBLAS.

2. Использование библиотеки CUBLAS значительно повышает производительность GPU по сравнению с традиционными методами программирования, которые используют большинство пользователей. Использование CUBLAS существенно уменьшает сложность написания программ для задач линейной алгебры.

3. Производительность GPU при матричном умножении с использованием CUBLAS для GeForce GTX 480 практически равна пиковой производительности при двойной точности вычислений (пиковая – 168.1, полученная 165.1 Гигафлопс). Для GPU Tesla C2075 пиковая производительность почти в два раза выше полученной (пиковая – 515.2, полученная 300.6 Гигафлопс). Это указывает на недостаточную эффективность библиотеки CUBLAS для GPU Tesla.

4. Стоимость GPU GeForce GTX 480 в момент написания статьи была равна 2500 грн., а GPU Tesla C2075 – 20000 грн. Поэтому, исходя из соотношения стоимость – производительность выгоднее использование для небольших расчетов, т.е. требующих памяти до 1.5 Гбайт, GPU GeForce GTX 480. Особенно выгодно его использование для расчетов с одинарной точностью, т.к. GPU GeForce GTX 480 работает в 8.4 раза быстрее CPU AMD Phenom II X6 1090T, а GPU Tesla C2075 лишь в 6.5 раза (это показали расчеты по этим же программам, которые были переписаны для чисел с одинарной точностью).

Литература

1. The ScaLAPACK Project [Electronic resource]. – Mode of access : http://www.netlib.org
2. Automatically Tuned Linear Algebra Software (ATLAS) [Electronic resource]. – Mode of access : http://math-atlas.sourceforge.net/
3. Антонов А. С. Параллельное программирование с использованием технологии MPI : учебное пособие / А. С. Антонов. – М. : Изд-во МГУ, 2004. – 71 с.
4. Мясищев А. А. Достижение наибольшей производительности перемножения матриц на системах с многоядерными процессорами / А. А. Мясищев // Теорія та методика навчання математики, фізики, інформатики : збірник наукових праць. Випуск VIII : в 3-х томах. – Кривий Ріг : Видавничий відділ НМетАУ, 2010. – Т. 3. – С. 174-186.
5. Боресков А. В. Основы работы с технологией CUDA / А. В. Боресков, А. А. Харламов. – М. : ДМК Пресс, 2010. – 232 с.