GPGPU

Паралелно програмиране

Основни термини

- Host

Компютърната система съдържаща GPU-то

Отнася се до CPU-то + RAM паметта към дънната платка

- Device

Видео картата/Ускорителя

- Global memory/Device memory

Паметта на видео картата

- Multiprocessor

SIMD елемент

- Compute capability

"Хардуерна" версия на графичния процесор (Tesla архитектурата 2.0)

CUDA програма

- CUDA програмата обикновено е организирана като
  - Host програма (PC системата, съдържаща видео картата, на която се стартира програмата се нарича Host). При този модел Host-а контролира последователността на изпълнение на програмата и управлява обмена на информация от/към GPU.
  - Едно или някокло ядра (kernels) стартирани на GPU. Програмата съдържа кода който може да се изпълни паралелно, с коет о да се ускорят изчисленията.

CUDA програма

- Програмен модел, използващ ускорители (accelerators)

  При този модел CPU и GPU са в Master-Slave режим

  - CPU=master
  - GPU=slave.
  - CPU и GPU могат да работят независимо един от друг, но CPU има контрол. Осъществява се паралелизъм между CPU и GPU , за което е необходима синхронизация.

Създаване на GPU код

- ## Създаване на ядро

  Чрез добавяне на спецификатора __global__ в декларацията на функцията , се създава CUDA ядро (kernel), което може да бъде извикано за изпълнение от host-а. Характерно за ядрата е, че типът на връщана стойност е void :

  ```
  void __global__ eval(float* a,float b,float* c,int size);
  ```

- ## Създаване на функция за устройство (device function)

  Спецификаторът __device__ указва , че функцията може да бъде извикана от устройство. Особеност при тези функции е, че не могат да бъдат рекурсивни.

```
__device__ float Max(float x, float y);
```

- ## Обикновени функции

  Всички останали функции (които нямат горните спецификатори) се компилират като код за изпълнение от host-а.

Извикване на CUDA kernel

Ядро може да бъде извикано като всяка една функция в кода, изпълняван от host-а. Изисква се конфигуриране на изпълнението, което се задава в <<<>>> .

```
VectorSum<<<gridSize, blockSize>>>(InputA, inputB, output, size);
```

Извикването на ядра е асинхронно, извикването на функция връща управлението веднага, преди изпълнението на устройството да е завършило. cudaDeviceSynchronize() може да се използва за изчакване на останалите ядра да приключат своето изпълнение.

Конфигуриране на изпълнението на ядро

<<<gridSize, blockSize>>>

Block of threads - блок от логически нишки, които се изпълняват едновременно (SIMT) и се разпределят на различните скаларни процесори в рамките на един мултипроцесор (MP –Multi Processor).

Grid of blocks – масив от блокове, всеки от които се изпълнява независимоедин от друг, в последователен(серийно) или паралелен режим. Разпределят се между различни мултипроцесори (MPs).

При извикване на ядро (извикване на функция) всички нишки, във всички бокове, паралелно изпълняват една и съща функция.

Разширен синтаксис за конфигуриране

```
dim3 blockSize(16,8,4);    // общо 512 нишки за един блок
dim3 gridSize(64,32);   //общо 2048 блока, съдържащи логически нишки
MyKernel<<<gridSize, blockSize>>>();
```

Нишките в блоковете могат да бъдат организирани в едно ,дву или три измерения. По подразбиране размерът на конкретното измерение се инициализира с 1.

Извличане на информация за нишка/блок

threadIdx.x(threadIdx.y, threadIdx.z) // индекс на атази нишка в съответния блока

blockDim.x(blockDim.y,blockDim.z) // брой на нишките в един блок

blockIdx.x(blockIdx.y,blockIdx.z) // индекс на блока, от който е нишката

gridDim.x(gridDim.y,gridDim.z) // брой на блоковете в грида

Ограничения

Максимален брой нишки в блок – 1024

blockDim.x*blockDim.y*blockDim.z ≤1024

· Максимален брой нишки в z- измерение – 64

blockDim.z ≤64

· Максимална големина на измеренията на блоковете:

-Fermi: 65535 във всички измерения :

gridDim.x ≤ 65535, gridDim.y ≤ 65535, gridDim.z ≤ 65535

-Kepler: 2147483647 за x- измерение, 65535 за y,z- измерения

gridDim.x ≤ 2147483647, gridDim.y ≤ 65535, gridDim.z ≤ 65535

Компилиране на кода

- На клъстера има инсталирани различни компилатори за CUDA, C, C++, Java и Fortran.
  - nvcc – компилатор за CUDA.

  [students@FKSU1207-2 ~]$nvcc <Сорс кодове> –o <име на изходния файл>
  - g++ - GNU GCC C++
  - gcc - GNU GCC C
- Изпълнение на CUDA приложения

  [students@FKSU120711s ~]$ ./<executable> <input parameters>

Организация на упражненията

- Предоставени са кодове на програми за CPU, например exercise01_cpu.c
- Необходимо е тези кодове да се модифицират, така че да използват GPU.
- В папката solutions са решенията (кодовете модифицирани за да използват GPU, например exercise01_solution.cu) до които студентите се опитват да достигнат самостоятелно.

Упражнение 1- серийно изпълнение на функция от GPU

exercise01_solution.cu

Задача- скаларно умножение на вектор. Изчисленията да се извършат от GPU, използвайки само една нишка и един блок.

Последователност от стъпки:

- Заделяна на памет за масив(вектор) в паметта на устройството с **cudaMalloc()** и заделяна на памет за масива(вектора) от паметта на host-а с **cudaMallocHost()**
- Копиране на входния вектор в паметта на GPU с **cudaMemcpy**
- Извикване на ядрото с gridSize=1 и blockSize=1 – като аргумент подаваме адреса на паметта на устройството
- Копиране на резултата обратно в паметта на host-а, чрез **cudaMemcpy**

Упражнение 2 – паралелно изпълнение на функция от GPU

exercise02_a_solution.cu

Задача- скаларно умножение на вектор. Изчисленията да се извършат от GPU, използвайки множество нишки и множество блокове, необходими за парелелни изчисления.

Използване на **threadIdx.x**, **blockDim.x**, **blockIdx.x** и **gridDim.x** , за да се определи за всяка нишка, кой елемент от масива да обработва.

```
void __global__ eval(float* a,float b,float* c,int size)
{
 for(int i=threadIdx.x+blockIdx.x*blockDim.x; i<size; i+=blockDim.x*gridDim.x)
 {
    c[i]=a[i]*b;
 }
}
```

Упражнение 2 Допълнително условие

exercise02_b_advanced_solution.cu

В ядрото не се използва for цикъл, а вместо това се задават големи размери на блоковете в зависимост от размера на вектора.

```
void __global__ eval(float* a,float b,float* c,int size)
{
 int index=threadIdx.x+blockIdx.x*blockDim.x;
 if(index<size)
 {
     c[index]=a[index]*b;
 }
}
```

Упражнение 3 – сумиране на вектори

exercise03_solution.cu

Компилирайте примера за сумиране на вектори и изследвайте разликите в производителността(бързодействието) при различни размери на block/grid. Коя е оптималната конфигурация? В какъв обхват на размерите на block/grid се постига добра производителност (70 GB/s за T10 и 80 GB/s при Fermi)?

```
void __global__ VectorSum(float* a,float* b,float* c,int size)
{
        for(int i=threadIdx.x+blockIdx.x*blockDim.x; i<size;
    i+=blockDim.x*gridDim.x)
        {
          c[i]=a[i]+b[i];
        }
}
```

Упражнение 4 – модификации на упражнение 3

exercise04_a_solution.cu

Добавео е отместване на входния вектор , с което не се обвързва с начина по който cudaMalloc() връща регионите на паметта.

```
cudaVerify(cudaMalloc((void**)&vector_a_gpu_alloc,(vector_size+offset_a)*sizeof(float)));
cudaVerify(cudaMalloc((void**)&vector_b_gpu_alloc,(vector_size+offset_b)*sizeof(float)));
cudaVerify(cudaMalloc((void**)&vector_c_gpu_alloc,(vector_size+offset_c)*sizeof(float)));
vector_a_gpu=vector_a_gpu_alloc+offset_a;
vector_b_gpu=vector_b_gpu_alloc+offset_b;
vector_c_gpu=vector_c_gpu_alloc+offset_c;
```

Какво влияние върху производителността очаквате да имат направените промени ? При Fermi виждате ли разлика между отместването на vector_c спрямо отместването на vector_a и vector_b ?

exercise04_b_solution.cu

Ядрото е пренаписано така, че всяка нишка обработва един продължителен регион от регионите на които е разделен вектора (стъпката е 1 вместо blockDim.x*gridDim.x).

```
for(int i=my_start; i<my_end;i++)
 {
   c[i]=a[i]+b[i];
}
```

Как ще се отрази тази промяна на преоизводителността- повишаване или влошаване ? Колко голяма ще бъде разликата в производителността ?

Упражнение 5 –дифузионно уравнение в 1D

exercise05_solution.cu

За изчисление на всяка точка от полето са необходими стойностите на съседните й точки.

$$fit+1 = fit + \alpha(f\ ti-1 + f\ ti+1 - 2fit)$$

CPU имплементация :

```
void DiffusionReference(float* dataIn,float* dataOut,int size)
{
        for(int i=0; i<size; i++)
    {
        dataOut[i] = 0.5f * dataIn[i] + 0.25f * ( dataIn[i-1] + dataIn[i+1] );
    }
}
```

Отбележете, че всяка входна стойност се чете 3 , но се очаква кеширане в L1

Задача:

- Да се модифицира ядрото от упражниение 2 или 3 като се имплементира времева стъпка на дифузионната схема на едномерен масив (или просто вземете решението от **exercise05_solution.cu**
- Променете настройките за рзмерите на grid и block за оптимална производителност
- Сравнете производителността на текущото упражнение и на упражнение 3.

Отбележете, че входния масив трябва да е по-голям с един елемент в началото и в края – общо 2 елемента повече от броя на елементите които ще се изчисляват. Не забравяйте да добавите отместване където е необходимо.

Упражнение 6 – скаларно произведение, използвайки един блок памет

exercise06_a_simple_solution.cu

Имплементация на CUDA програма, която изчислява скаларното произведение на два вектора, използвайки един мултипроцесор.

Последователност от стъпки:

- Сумиране на всички елементи- едно нишков процес
- Копиране на разултата от всяка нишка в споделената памет
- Редуциране на информацията в споделента памет до един краен резултат

CPU имплементация:

```
float ScalarProduct(float* a,float* b,int size)
{
        double sum=0;
        for( int i=0;i<size;i++)
        {
         sum += a[i] * b[i];
        }
        return sum;
}
```

Упражнение 7 – скаларно произведение, използвайки множество блокове

exercise07_solution.cu

Модифицирана програма от упражнение 6 , така че да използва множество блокове за изчисления. Създадено е отделно ядро за редукция(намаляване) на междинните резултати на отделните блокове до намиране на крайния резултат, с което се осигурява мултипроцесорна синхронизация . В приложение от реалния свят (особено при използване на Fermi) тази стратегия работи дори без използване на споделената памет.