Optimization of a CUDA code

Presentation

Performance measurement

Optimization of data transfers between host and device

Cache memory

Shared memory

Presentation

- we must generate enough parallelism to consistently occupy (overflow) all the cores

→ a *big enough data set* is required to generate enough data-based parallelism

→ matrix size of at least a few thousands

- we must optimize the use of the *memory system* of the GPU

presence of *bottlenecks* in the memory system:

- between host memory and device global memory
- between device global memory and each core

Performance measurement

- measure the *running time* in function of the *problem size* (matrix size)

  → use the C function **clock()** of return type **clock_t** (unsigned integer on 64 bits) before and after a task, then subtract the times.

  - **clock()** can be used in *host code*
  - **clock()** can also be used in *kernel code* (only for 2.x)

- calculate the performance as *billions of floating point operations per second* (*Gflops*) in function of the problem size

  → analytical expression of the number of floating point operations in function of the problem size

- *speed up*, performance as *Gflops per core*, ...

Optimization of data transfers between host and device

the transfer rate between host and device is limited = *bottleneck*

→ as much as possible, data transfers between host and device should be used only to:

- load the *input data* on the GPU
- repatriate on the host the *final result data*

→ as much as possible, the main data structures should be *duplicated* from the host memory onto the device global memory

but the device global memory contains only 3 GB

→ use the device global memory as a *cache memory*

= exploit *locality* and *repetition of calculations* within the data structures...

Cache memory and shared memory of each multiprocessor

- *cache system and multiprocessor shared memory*

  = *partial solution* to *bottleneck* between device global memory and cores

  → exploit *locality* and *repetition of calculations*

  → try to increase the number of times an element loaded in the cache and / or shared memory will be accessed

  = try to introduce a local loop (see matrix multiplication)

- *shared memory of a multiprocessor*

  = extra layer of *manually handled cache memory*

  → expect performance to be increased by a factor 5 or so (see matrix multiplication)

- *variable in shared memory*

  - is declared inside the kernel code with modifier **__shared__**
  - is accessible by all the threads of the block which is executed on this multiprocessor

- *explicit initialization of the elements of an array in shared memory*

  step 1: each thread of the block is used to load one element

  step 2: meeting point between all the threads of the block → use **_syncthreads();**

  step 3: perform calculations on this array (only after all elements have been loaded...)

- *configure ratio L1 cache - shared memory*

shared memory = 16KB ; L1 cache = 48KB :

```
CUDA_CALL(".." , cudaFuncSetCacheConfig(my_kernel , cudaFuncCachePreferL1));
```

shared memory = 48KB ; L1 cache = 16KB :

```
CUDA_CALL(".." , cudaFuncSetCacheConfig(my_kernel , cudaFuncCachePreferShared));
```

example: matrix mat optimized

→ **WITHOUT SHARED MEMORY:**

```
__global__ void kernel_mul_mat(  double *C , double *A , double *B , int n , int m , int p
                                      , int nB, int mB , int k0 , int k1)
{
   int iB = blockIdx.x;    int i0 = iB * n  / nB  , i1 = (iB + 1) * n  / nB;      int i = i0 + threadIdx.x;
   int jB = blockIdx.y;    int j0 = jB * m / mB , j1 = (jB + 1) * m / mB;    int j = j0 + threadIdx.y;
   int k;

   if (i < i1  &&  j < j1)
   {
      for (k = k0 ; k < k1 ; k++)
         Mij(C , m , i , j) += Mij(A , p , i , k) * Mij(B , m , k , j);
   }
}
```

→ **WITH SHARED MEMORY:**

```
__global__ void kernel_mul_mat(  double *C , double *A , double *B , int n , int m , int p
                                      , int nB, int mB , int k0 , int k1)
{
   int   iB = blockIdx.x   ,   i0 = iB * n /nB    ,   i1 = (iB + 1) * n /nB    ,   di = threadIdx.x   ,   i = i0 + di;
   int   jB = blockIdx.y   ,   j0 = jB * m/mB   ,   j1 = (jB + 1) * m/mB   ,   dj = threadIdx.y   ,   j = j0 + dj;
   int dk , k1_k0 = k1 - k0;

   double Cij;
   __shared__ double  __A__[SB][SB] , __B__[SB][SB];

   if (i < i1  &&  j < j1)
      Cij = Mij(C , m , i , j);

   if (i < i1  &&  dj < k1_k0)                    //// thread  (i , j) loads A(i , k)
      __A__[di][dj] = Mij(A , p , i , k0 + dj);

   if (di < k1_k0  &&  j < j1)                    //// thread  (i , j) loads B(k , j)
      __B__[di][dj] = Mij(B , m , k0 + di , j);

   __syncthreads();

   if (i < i1  &&  j < j1)
   {
      for (dk = 0 ; dk < k1_k0 ; dk++)
         Cij += __A__[di][dk] * __B__[dk][dj];

      Mij(C , m , i , j) = Cij;
   }
}
```

up to 40 Gflops