CUDA Programming

Piotr Luszczek

Structure of CUDA Code

```
// parallel function (GPU)
__global__ void sum(double x, double y, double *z) {
  *z = x + y;
}

// sequential function (CPU)
void sum_cpu(double x, double y, double *z) {
  *z = x + y;
}

// sequential function (CPU)
int main(void) {
  double *dev_z, hst_z;

  cudaMalloc( &device_z, sizeof(double) );

  // launch parallel code (CPU → GPU)
  sum<<<1,1>>>(2, 3, dev_z);

  cudaMemcpy( &hst_z, dev_z, sizeof(double), cudaMemcpyDeviceToHost );

  printf("%g\n", hst_z[i]);

  cudaFree(dev_z);

  return 0;
}
```

Introducing Parallelism to CUDA Code

- Two points where parallelism enters the code
  - Kernel invocation
    - `sum<<< 1,1>>>( a, b, c )`
    - `sum<<<10,1>>>( a, b, c )`
  - Kernel execution
    - `__global__ void sum(double *a, double *b, double*c)`
    - `c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x]`
- CUDA makes the connection between:
  - invocation `sum<<<10,1>>>`
    with
  - execution and its index blockIdx.x
- Recall GPU massive parallelism
  - Many CUDA cores
  - Many CUDA threads
  - Many GPU SM (or SMX) units

CUDA Parallelism with Blocks

```
int N = 100, SN = N * sizeof(double);

__global__ void sum(double *a, double *b, double *c) {
  c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x]; // no loop!
}

int main(void) {
  double *dev_a, *dev_b, *dev_c, *hst_a, *hst_b, *hst_c;

  cudaMalloc( &dev_a, SN ); hst_a = calloc(N, sizeof(double));
  cudaMalloc( &dev_b, SN ); hst_b = calloc(N, sizeof(double));
  cudaMalloc( &dev_c, SN ); hst_c = malloc(N, sizeof(double));

  cudaMemcpy( dev_a, hst_a, SN, cudaMemcpyHostToDevice );
  cudaMemcpy( dev_b, hst_b, SN, cudaMemcpyHostToDevice );

  sum<<<10,1>>>(dev_a, dev_b, dev_c); // only 10 elements will be used

  cudaMemcpy( &hst_c, dev_c, SN, cudaMemcpyDeviceToHost );

  for (int i=0; i<10; ++i) printf("%g\n", hst_c[i]);

  cudaFree(dev_a); free(hst_a);
  cudaFree(dev_b); free(hst_b);
  cudaFree(dev_c); free(hst_c);

  return 0;
}
```

Details on Execution of Blocks on GPU

- Blocks is a level of parallelism
  - There are other levels
- Blocks execute in parallel
  - Synchronization is
    - Explicit (special function calls, etc.)
    - Implicit (memory access, etc.)
    - Mixed (atomics, etc.)
- Total number of available blocks is hardware specific
  - CUDA offers inquiry functions to get the maximum block count

`// BLOCK 0` `c[0]=a[0]+b[0];`	`// BLOCK 1` `c[1]=a[1]+b[1];`
`// BLOCK 2` `c[2]=a[2]+b[2];`	`// BLOCK 3` `c[3]=a[3]+b[3];`
`// BLOCK 4` `c[4]=a[4]+b[4];`	`// BLOCK 5` `c[5]=a[5]+b[5];`
`// BLOCK 6` `c[6]=a[6]+b[6];`	`// BLOCK 7` `c[7]=a[7]+b[7];`
`// BLOCK 8` `c[8]=a[8]+b[8];`	`// BLOCK 9` `c[9]=a[9]+b[9];`

Introducing Thread Parallelism to CUDA Code

- Kernel invocation
  - `sum<<<10, 1>>>( x, y, z )  // block-parallel`
  - `sum<<< 1,10>>>( x, y, z )  // thread-parallel`
- Kernel execution
  - `z[threadIdx.x] = x[threadIdx.x] + y[threadIdx.x]`
- Consistency of syntax
  - Minimum changes to switch from blocks to threads
  - Similar naming for blocks and threads

CUDA Parallelism with Threads

```
int N = 100, SN = N * sizeof(double);

__global__ void sum(double *a, double *b, double *c) {
  c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x]; // no loop!
}

// sequential function (CPU)
int main(void) {
  double *dev_a, *dev_b, *dev_c, *hst_a, *hst_b, *hst_c;

  cudaMalloc( &dev_a, SN ); hst_a = calloc(SN);
  cudaMalloc( &dev_b, SN ); hst_b = calloc(SN);
  cudaMalloc( &dev_c, SN ); hst_c = malloc(SN);

  cudaMemcpy( dev_a, hst_a, SN, cudaMemcpyHostToDevice );
  cudaMemcpy( dev_b, hst_b, SN, cudaMemcpyHostToDevice );

  sum<<<1,10>>>(dev_a, dev_b, dev_c);

  cudaMemcpy( &hst_c, dev_c, SN, cudaMemcpyDeviceToHost );

  for (int i=0; i<10; ++i) printf("%g\n", hst_c[i]);

  cudaFree(dev_a); free(hst_a);
  cudaFree(dev_b); free(hst_b);
  cudaFree(dev_c); free(hst_c);

  return 0;
}
```

More on Block and Thread Parallelism

- When to use blocks and when to use threads?
  - Synchronization between threads is cheaper
  - Blocks have higher scheduling overhead
- Block and thread parallelism can be combined
  - Often it is hard to get good balance between both
  - Exact combination depends on
    - GPU generation
      - Tesla, Fermi, Kepler, Maxwell, Pascal, Volta, ...
    - SM/SMX configuration
    - Memory size