Программирование для систем с несколькими GPU

Романенко А.А.
arom@ccfit.nsu.ru

Новосибирский государственный университет

Количество GPUs

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
   cudaDeviceProp deviceProp;
   cudaGetDeviceProperties(&deviceProp, device);
   printf("Device %d has compute capability %d.%d.\n",
        device, deviceProp.major, deviceProp.minor);
}
```

GPU и поток исполнения CUDA 3.2

- Поток ассоциирован с одним GPU *
- Выбор GPU или явно (cudaSetDevice()) или неявно - по-умолчанию.
- По умолчанию выбирается GPU с номером «0»
- Если в потоке выполнена какая-либо операций над GPU, то попытка сменить GPU на другой приведет к ошибке.

* - на уровне драйвера это не так.

GPU и поток исполнения CUDA 4.0

- Любой поток имеет доступ ко всем GPU
  - Выбор активного устройства через вызов cudaSetDevice()
- Возможность запуска параллельных ядер из разных потоков.

Копирование данных между GPU

- **CUDA 3.2**

cudaMemcpy(Host, GPU1);
cudaMemcpy(GPU2, Host);

CUDA 4.0

cudaMemcpy(GPU1, GPU2);

Можно как читать так и писать в память.

Поддерживается только на Tesla 20xx (Fermi)

64-битные приложения

Unified Virtual Addressing CUDA 4.0

- Память центрального процессора и всех GPU объединена в единое виртуальное адресное пространство.
- Один параметр (cudaMemcpyDefault) вместо 4-х (cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice)
- Поддерживается только на Tesla 20xx (Fermi)
- 64-битные приложения

Многопоточное программирование

- POSIX Threads
- WinThreads
- OpenMP
- MPI
- IPC
- пр.

OpenMP

```
#pragma omp parallel sections
  {
  #pragma omp section
   {
    cudaSetDevice(0);
    ...
   }
  #pragma omp section
   {
    cudaSetDevice(1);
    ...
   }
  }
```

OpenMP

```
int nElem = 1024;
   cudaGetDeviceCount(&nGPUs);
   if(nGPUs >= 1){
        omp_set_num_threads(nGPUs);
#pragma omp parallel
        {
           unsigned int cpu_thread_id = omp_get_thread_num();
           unsigned int num_cpu_threads = omp_get_num_threads();
           cudaSetDevice(cpu_thread_id % nGPUSs); //set device

           dim3 BS(128);
           dim3 GS(nElem / (gpu_threads.x * num_cpu_threads));

           // memory allocation and initialization
           int startIdx = cpu_thread_id * nElem / num_cpu_threads;
           int threadNum = nElem / num_cpu_threads;
           kernelAddConstant<<<GS, BS>>>(pData, startIdx, threadNum);
           // memory copying
        }
```

OpenMP. Сборка программ

- gcc 4.3
- Command line
  - ```
    $ nvcc -Xcompiler \
    -fopenmp -Xlinker\
    -lgomp cudaOpenMP.cu
    ```
- Makefile
  - ```
    EXECUTABLE := cudaOpenMP
    CUFILES := cudaOpenMP.cu
    CUDACCFLAGS := -Xcompiler -fopenmp
    LIB := -Xlinker -lgomp
    include ../../common/common.mk
    ```

CUDA Utillity Library

```
static CUT_THREADPROC solverThread(SomeType *plan){
        // Init GPU
        cutilSafeCall( cudaSetDevice(plan->device) );
        // start kernel
        SomeKernel<<<GS, BS>>>(some parameters);
        cudaThreadSynchronize();

        cudaThreadExit();
        CUT_THREADEND;
    }
```

- Макросы используются для переносимости программы с Unix на Windows и обратно.

CUDA Utillity Library

```
SomeType solverOpt[MAX_GPU_COUNT];
   CUTThread threadID[MAX_GPU_COUNT];

   for(i = 0; i < GPU_N; i++){
      solverOpt[i].device = i; …
   }

//Start CPU thread for each GPU
   for(gpuIndex = 0; gpuIndex < GPU_N; gpuIndex++){
       threadID[gpuIndex] =
              cutStartThread((CUT_THREADROUTINE)solverThread,
                             &SolverOpt[gpuIndex]);
   }


//waiting for GPU results
   cutWaitForThreads(threadID, GPU_N);
```

Работа с драйвером

- Для каждого устройства явно создается контекст (**cuCtxCreate**)
- Перед выполнением операций с устройством соответствующий контекст делается текущим (**cuCtxPushCurrent**), а после операции – снимается (**cuCtxPopCurrent**)
- В конце контексты удаляются (**cuCtxDestroy**)

- (!!) Если контекст создан до вызова fork(), то после него работа с контекстом может быть некорректна

Создание контекстов

```
for(int i=0; i<nGPUS; i++){
   CUdevice dev;
   CUresult cu_status = cuDeviceGet(&dev, i);
   if (cu_status != CUDA_SUCCESS) {/* обработка ошибки */ }

   device_t *device = &devices[i];
   cu_status = cuCtxCreate(divice->ctx, 0, dev);
   if (cu_status != CUDA_SUCCESS) {/* обработка ошибки */ }

   CUresult cu_status = cuCtxPopCurrent(divice->ctx);
   if (cu_status != CUDA_SUCCESS) { /* обработка ошибки */ }
}
```

Работа с контекстами

```
for(int i=0; i<nGPUS; i++){
   device_t *device = &devices[i];
   // сделать контекст активным для текущего потока\процесса
   CUresult cu_status = cuCtxPushCurrent(device->ctx);
   if (cu_status != CUDA_SUCCESS) {/* обработка ошибки */ }

   // инициализация памяти, запуск ядра …

   // отключить контекст от потока\процесса
   cu_status = cuCtxPopCurrent(device->ctx);
   if (cu_status != CUDA_SUCCESS) {/* обработка ошибки */ }
}
```

Завершение контекста

```
for(int i=0; i<nGPUS; i++){
   device_t *device = &devices[i];
   // сделать контекст активным для текущего потока\процесса
   CUresult cu_status = cuCtxPushCurrent(device->ctx);
   if (cu_status != CUDA_SUCCESS) {/* обработка ошибки */ }
   // дождаться завершения ядра
   cuda_status = cudaThreadSynchronize();

   // сохранение результата, освобождение памяти


   // удалить контекст
   cu_status = cuCtxDestroy (device->ctx);
   if (cu_status != CUDA_SUCCESS) {/* обработка ошибки */ }
}
```