The Ministry of Education and Science of the Russian Federation

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program

of the Lobachevsky State University of Nizhni Novgorod

among the world's research and education centers

Strategic initiative

“Achieving leading positions in the field of supercomputer technology

and high-performance computing”

Introduction to GPU programming

Lecture 6. Optimization of CUDA applications

Nizhni Novgorod

2014

Author: S.I. Bastrakov

OBJECTIVES

The objective of this lecture is to discuss optimization of CUDA applications and demonstrate optimization of parallel reduction implementation.

ABSTRACT

This lecture discusses various techniques and tools that can be used for optimization of CUDA application. We illustrate it by an example of optimization of parallel reduction implementation.

BRIEF OVERVIEW

The first and sometimes most important principle of optimization is to choose an algorithm that is appropriate for parallel computing. Typical problem decomposition schemes for GPU applications are: decomposition into thousands or more independent subproblems, or decomposition into tens/hundreds of independent subproblems that are decomposed into smaller problems. It is preferable to have high computational density. Sometimes a part of an algorithm is highly parallelizable, while another part is sequential, in this case it is recommended to use CPU for sequential computations and GPU for parallel. Another important factor is minimizing data exchanges or doing it asynchronously using cudaStream data type and functions cudaStreamCreate, cudaStreamDestroy, cudaStreamSynchronize, and cudaDeviceSynchronize.

The main kind of optimization for GPU applications is memory optimization. It is extremely important to use efficient patterns for global and shared memory (refer to the previous lecture for details). Intensively used data should be stored in cache/shared memory.

Number of threads per multiprocessor is usually much higher than number of CUDA cores. Large number of threads helps to hide memory latency. Occupancy is a ratio of number of active threads per multiprocessor to maximum possible number of threads per multipricessor. Limiting factor for increasing number of threads is size of shared memory and registers. If there are enough resources, multiple blocks can run on the same multiprocessor concurrently.

Use single precision floating point arithmetic when possible. GPUs support very fast but less precise versions if math routines: __sinf(x), __cosf(x), __expf(x), etc. Compiler option -use_fast_math replaces all math routines with faster and less precise ones.

CUDA Toolkit contains several optimized libraries: CUBLAS, CUFFT, CURAND, CUSPARSE, NPP, Thrust. Some libraries will be discussed in the next lecture. There are lots of 3rd

party libraries as well. Use profiler to find bottlenecks and receive performance recommendations.

We demonstrate the main optimization principles on a parallel reduction implementation. Here is a basic reduction implementation:

```
__global__ void reduce1 ( int * inData, int * outData )
{
  __shared__ int data [BLOCK_SIZE];
  int tid = threadIdx.x;
  int i   = blockIdx.x * blockDim.x + threadIdx.x;
  data [tid] = inData [i];    // load into shared memory
  __syncthreads ();
  for ( int s = 1; s < blockDim.x; s *= 2 ) {
    if ( tid % (2*s) == 0 )   // heavy branching !!!
      data [tid] += data [tid + s];
    __syncthreads ();
  }
  if ( tid == 0 )             // write result of block reduction
    outData[blockIdx.x] = data [0];
}
```

We sequentially demonstrate application of the following optimization steps: eliminating conditional statements, avoiding bank conflicts, load balancing, loop unrolling, and warp-level parallelism. The version with all the optimizations applied is as follows:

```
__global__ void reduce4 ( int * inData, int * outData )
{
  __shared__ int data [BLOCK_SIZE];
  int tid = threadIdx.x;
  int i   = 2 * blockIdx.x * blockDim.x + threadIdx.x;
  data [tid] = inData [i] + inData [i+blockDim.x]; // sum
  __syncthreads ();
  for ( int s = blockDim.x / 2; s > 32; s >>= 1 ) {
    if ( tid < s )
      data [tid] += data [tid + s];
    __syncthreads ();
  }
  if ( tid < 32 ) { // unroll last iterations
    data [tid] += data [tid + 32];
    data [tid] += data [tid + 16];
    data [tid] += data [tid + 8];
    data [tid] += data [tid + 4];
    data [tid] += data [tid + 2];
    data [tid] += data [tid + 1];
  }
  if ( tid == 0 )
    outData [blockIdx.x] = data [0];
}
```

FOR STUDENTS

Empirical evaluation of various performance optimization techniques is presented in [1]. Advanced optimization topics are covered in [2, 3].

REFERENCES

1. Farber R. CUDA Application Design and Development. – Morgan Kaufmann, 2011. – 336 p.

2. GPU Computing Gems Emerald Edition, ed. Wen-mei W. Hwu. – Morgan Kaufmann, 2011. – 886 p.
3. NVIDIA CUDA C Best Practices Guide [http://docs.nvidia.com/cuda/cuda-c-best-practices-guide#axzz3JRcPurfI]

INDIVIDUAL WORK

1. Describe main optimization techniques for CUDA applications.
2. Which factors are important in evaluation how suitable is an algorithm for GPU?