


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”

Lobachevsky State University of Nizhni Novgorod

Faculty of Computational mathematics and cybernetics

Introduction to GPU programming

06 Lecture
Optimization of CUDA applications

Bastrakov S.I.
Software department

Contents

- General recommendations
- Example: optimization of parallel reduction

General recommendations

Algorithm choice

- Choose algorithms that are appropriate for parallel computing. Typical problem decomposition schemes:
  - decomposition into thousands or more independent subproblems;
  - decomposition into tens/hundreds of independent subproblems that are decomposed into smaller problems.
- Preferably high computational density.
- Using CPU for sequential computations and GPU for parallel.
- Minimizing data exchanges or doing it asynchronously.

Asynchronous execution

- There are streams, data type cudaStream.
- cudaStreamCreate.
- cudaStreamDestroy .
- cudaStreamSynchronize.
- cudaDeviceSynchronize.

Example: cudaStreamCreate

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

Example: cudaMemcpyAsync

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
   MyKernel<<<100, 512, 0, stream[i]>>>
         (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
```

Example: cudaStreamDestroy

```
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

Memory optimization

- The main kind of optimization for GPU applications.
- Using efficient patterns for global and shared memory (refer to the previous lecture for details).
- Storing intensively used data in cache/shared memory.

Occupancy optimization

- 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.

Arithmetic operations

- 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.

Using tools and libraries

- CUDA Toolkit contains several optimized libraries: CUBLAS, CUFFT, CURAND, CUSPARSE, NPP, Thrust.
  - There are lots of 3rd party libraries as well.
- Use profiler to find bottlenecks and receive performance recommendations.

Example: optimization of parallel reduction

Problem statement

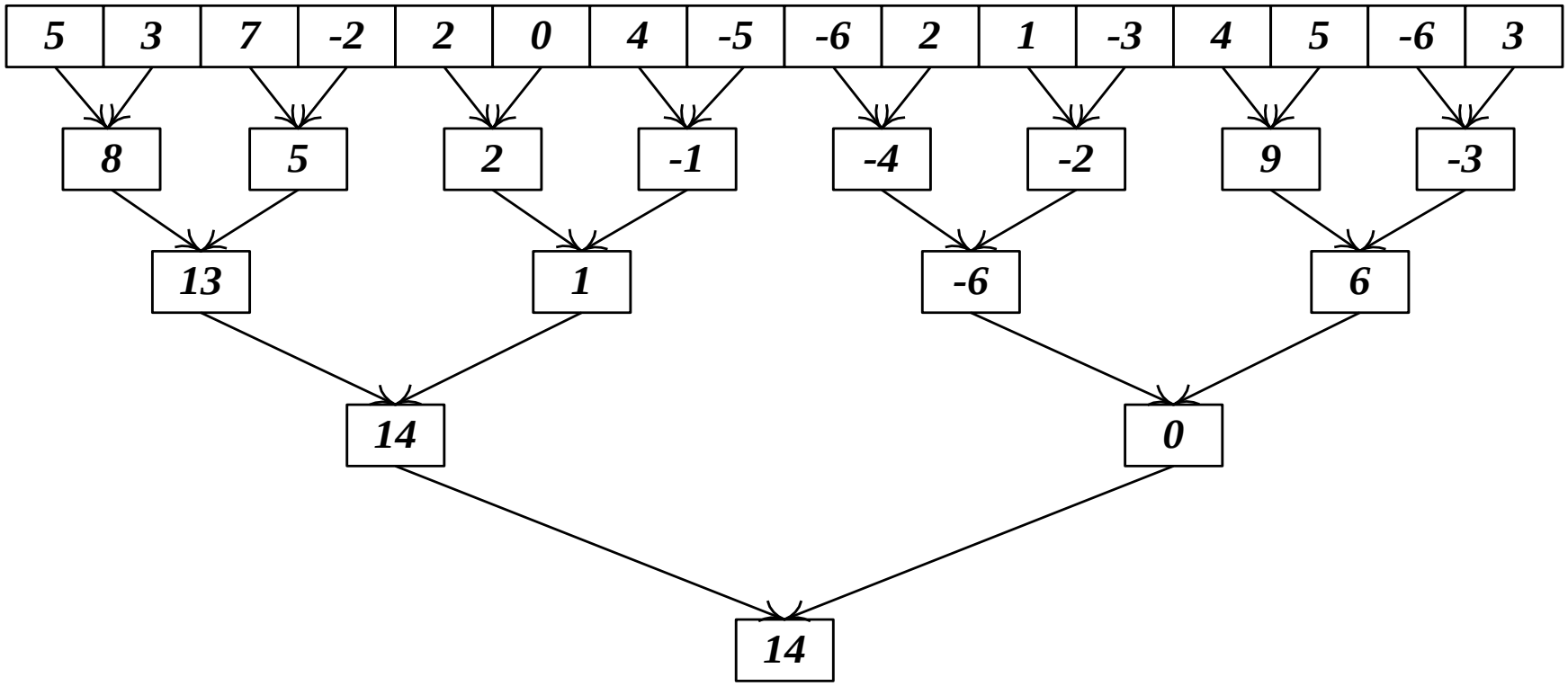
- Problem statement:
  - Array $a_0, a_1, \ldots, a_{n-1}$
  - Associative operation «+» (might be +, *, min, max)
- Result:

$$A = a_0 + a_1 + \ldots + a_{n-1}$$

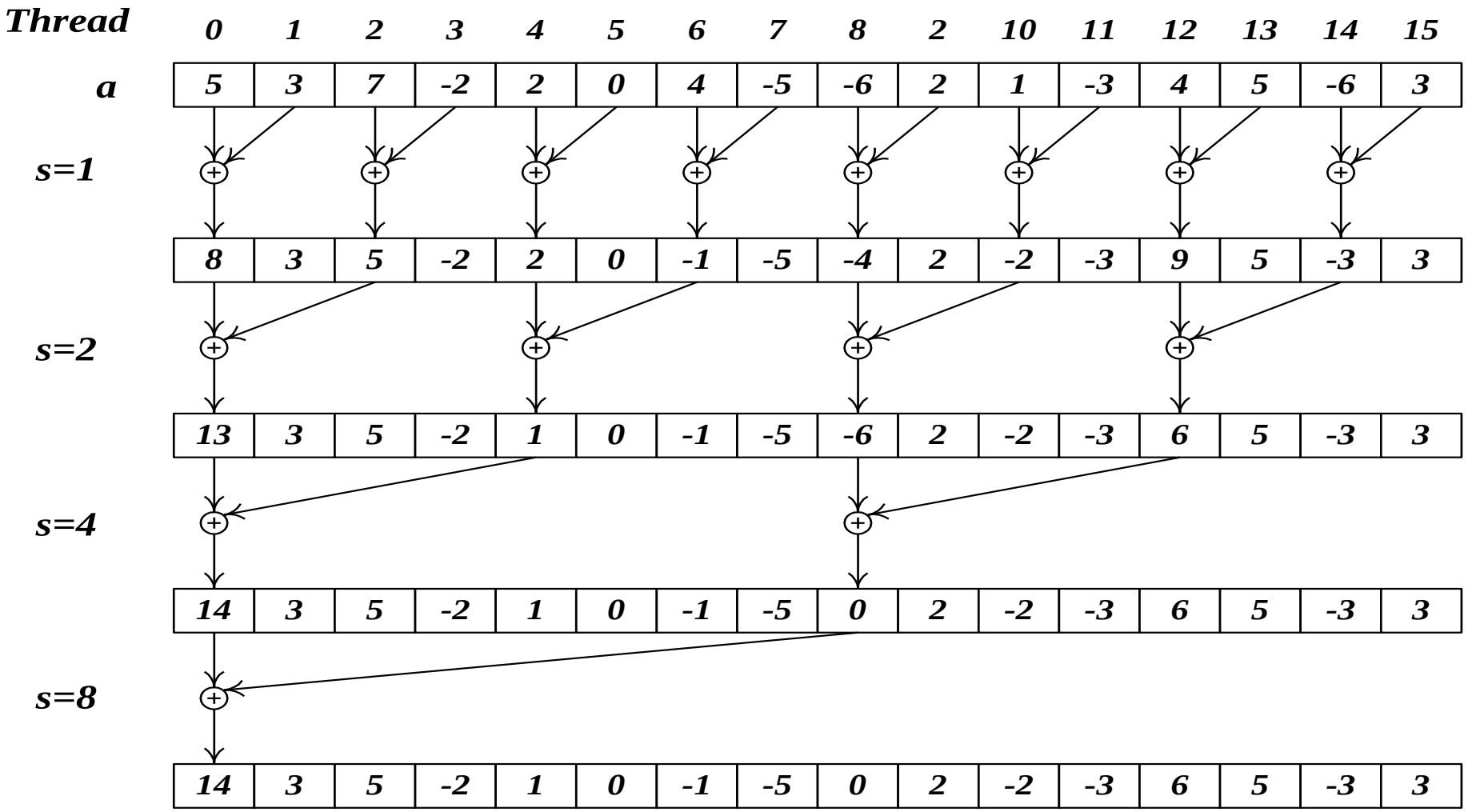
- Limiting factor is memory access.

- *This section is based on reduction Example from GPU Computing SDK and the corresponding whitepaper, and materials of A.V. Boreskov, A.A. Harlamov*

Hierarchic reduction



Version 1


Thread
0 1 2 3 4 5 6 7 8 2 10 11 12 13 14 15
a
5 3 7 -2 2 0 4 -5 -6 2 1 -3 4 5 -6 3
s=1
8 3 5 -2 2 0 -1 -5 -4 2 -2 -3 9 5 -3 3
s=2
13 3 5 -2 1 0 -1 -5 -6 2 -2 -3 6 5 -3 3
s=4
14 3 5 -2 1 0 -1 -5 0 2 -2 -3 6 5 -3 3
s=8
14 3 5 -2 1 0 -1 -5 0 2 -2 -3 6 5 -3 3

Version 1

```
__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];
}
```

Version 2

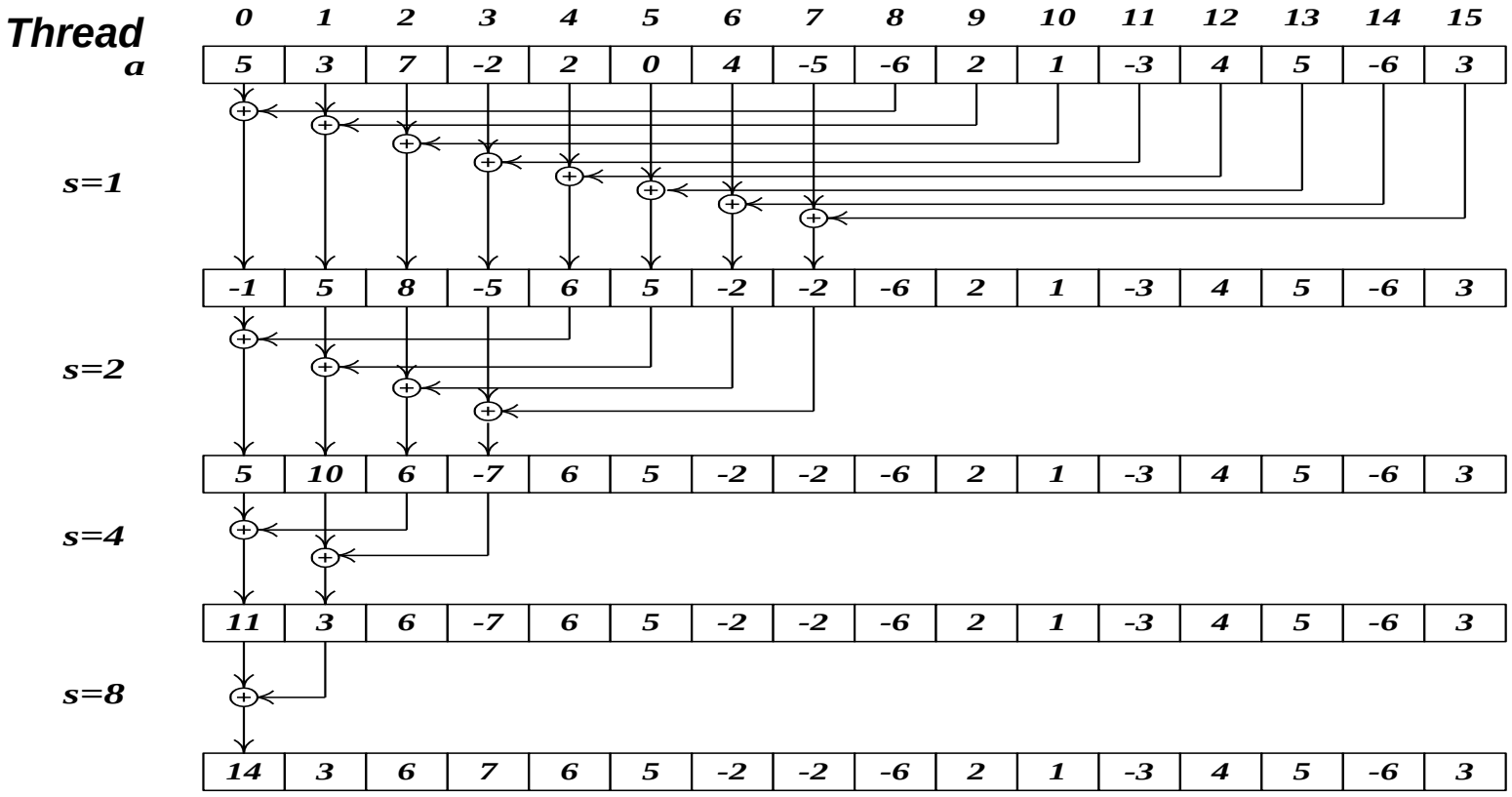
```
__global__ void reduce2 ( int * inData, int * outData )
{
  __shared__ int data [BLOCK_SIZE];
  int tid = threadIdx.x;
  int i   = blockIdx.x * blockDim.x + threadIdx.x;
  data [tid] = inData [i];     // load into shared memory
  __syncthreads ();
  for ( int s = 1; s < blockDim.x; s <<= 1 )
  {
    int index = 2 * s * tid;
    if ( index < blockDim.x )
      data [index] += data [index + s];
    __syncthreads ();
  }
  if ( tid == 0 )              // write result of block reduction
    outData [blockIdx.x] = data [0];
}
```

Version 2

- It differs from version 1 only in which threads do the same work.
- Now we do not have conditional statements.
- But on each iterations there are twice as many bank conflicts

Version 3

- Change summation order



Version 3

```
__global__ void reduce3 ( 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];
  __syncthreads ();
  for ( int s = blockDim.x / 2; s > 0; s >>= 1 )
  {
    if ( tid < s )
      data [tid] += data [tid + s];
    __syncthreads ();
  }
  if ( tid == 0 )
    outData [blockIdx.x] = data [0];
}
```

Version 4

- In version 3 we got rid of bank conflicts.
- But on the first iteration half threads are doing nothing. We fix it in version 4.

Version 4

```
__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 > 0; s >>= 1 )
  {
    if ( tid < s )
      data [tid] += data [tid + s];
    __syncthreads ();
  }
  if ( tid == 0 )
    outData [blockIdx.x] = data [0];
}
```

Version 5

- For s <= 32 there is only one active warp.
- In this case we can unroll the loop and avoid synchronization

```
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];
}
```

Results

Version	Run time (ms)
1	19.09
2	11.91
3	10.62
4	9.10
5	8.67

- Further optimization is possible

References

- Farber R. CUDA Application Design and Development. – Morgan Kaufmann, 2011. – 336 p.
- GPU Computing Gems Emerald Edition, ed. Wen-mei W. Hwu. – Morgan Kaufmann, 2011. – 886 p.
- NVIDIA CUDA C Best Practices Guide [http://docs.nvidia.com/cuda/cuda-c-best-practices-guide#axzz3JRcPurfl]

Authors

- Bastrakov S.I.,
  Assistant of the Software department of CMC faculty.
  bastrakov@vmk.unn.ru