Atomic Operations

Atomic operations, fast reduction

GPU Programming

http://cuda.nik.uni-obuda.hu

Szénási Sándor
szenasi.sandor@nik.uni-obuda.hu

GPU Education Center of Óbuda University

NVIDIA GPU EDUCATION CENTER

Atomic Operations

Atomic arithmetic

Other atomic functions

Fast reduction

Atomic operations

- Atomic operations are operations which are performed without interference from any other threads. Atomic operations are often used to prevent race conditions which are common problems in multithreaded applications [27].
- In case of some task we need atomic operations, for example:
  - sum/average of a data structure
  - min/max item of a data structure
  - count of some items in a data structure
  - etc.

Possible solutions

- We can implement some of these tasks in parallel environment (for example, is there any special item in the data structure?)
- But some of them is hard to parallelize (for example, find the minimum value in the data structure)

CUDA atomic instructions

CUDA atomics

- The atomic instructions of the CUDA environment can solve the race conditions mentioned before. When using atomic instructions the hardware will guarantee the serialized execution
- Operand location
  - variable in global memory
  - variable in shared memory
- Operand size
  - 32bit integer (≥CC1.1)
  - 64bit integer (≥CC1.2)
  - 32bit float (≥CC1.1)
  - 64bit double (≥CC1.3)

Performance notes

- If two threads perform an atomic operation at the same memory address at the same time, those operations will be serialized. This will slow down the kernel execution
- In case of some special tasks, we can not avoid atomic instructions. But in most cases if it is possible we would try to find another solution. The goal is to use as less atomic instructions as possible

Arithmetic operations

Basic arithmetic operations

- The first parameter of atomic instructions is usually a memory address (in global or local memory), the second parameter is an integer
- int atomicAdd(int* address, int val)
  Reads the 32-bit or 64-bit word old located at the address in global or shared memory, computes (old + val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old
- int atomicSub(int* address, int val)
  Reads the 32-bit word old located at the address in global or shared memory, computes (old - val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old
- unsigned int atomicInc(unsigned int* address, unsigned int val)
  Reads the 32-bit word old located at the address in global or shared memory, computes ((old >= val) ? 0 : (old+1)), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old
- unsigned int atomicDec(unsigned int* address, unsigned int val)
  Reads the 32-bit word old located at the address in global or shared memory, computes (((old == 0) | (old > val)) ? val : (old-1)), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old

Comparision atomic operations

Minimum-maximum operations

- int atomicMin(int* address, int val)
  Reads the 32-bit word old located at the address in global or shared memory, computes the minimum of old and val, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old
- int atomicMax(int* address, int val)
  Reads the 32-bit word old located at the address in global or shared memory, computes the maximum of old and val, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old

Comparision operations

- int atomicExch(int* address, int val)
  Reads the 32-bit or 64-bit word old located at the address in global or shared memory and stores val back to memory at the same address. These two operations are performed in one atomic transaction. The function returns old
- int atomicCAS(int* address, int compare, int val)
  Compare And Swap: reads the 32-bit or 64-bit word old located at the address in global or shared memory, computes (old == compare ? val : old), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old

Logical atomic instructions

Logical atomic operations

- int atomicAnd(int* address, int val)
  Reads the 32-bit word old located at the address in global or shared memory, computes (old & val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old
- int atomicOr(int* address, int val)
  Reads the 32-bit word old located at the address in global or shared memory, computes (old | val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old
- int atomicXor(int* address, int val)
  Reads the 32-bit word old located at the address in global or shared memory, computes (old ^ val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old

Device component – atomic functions

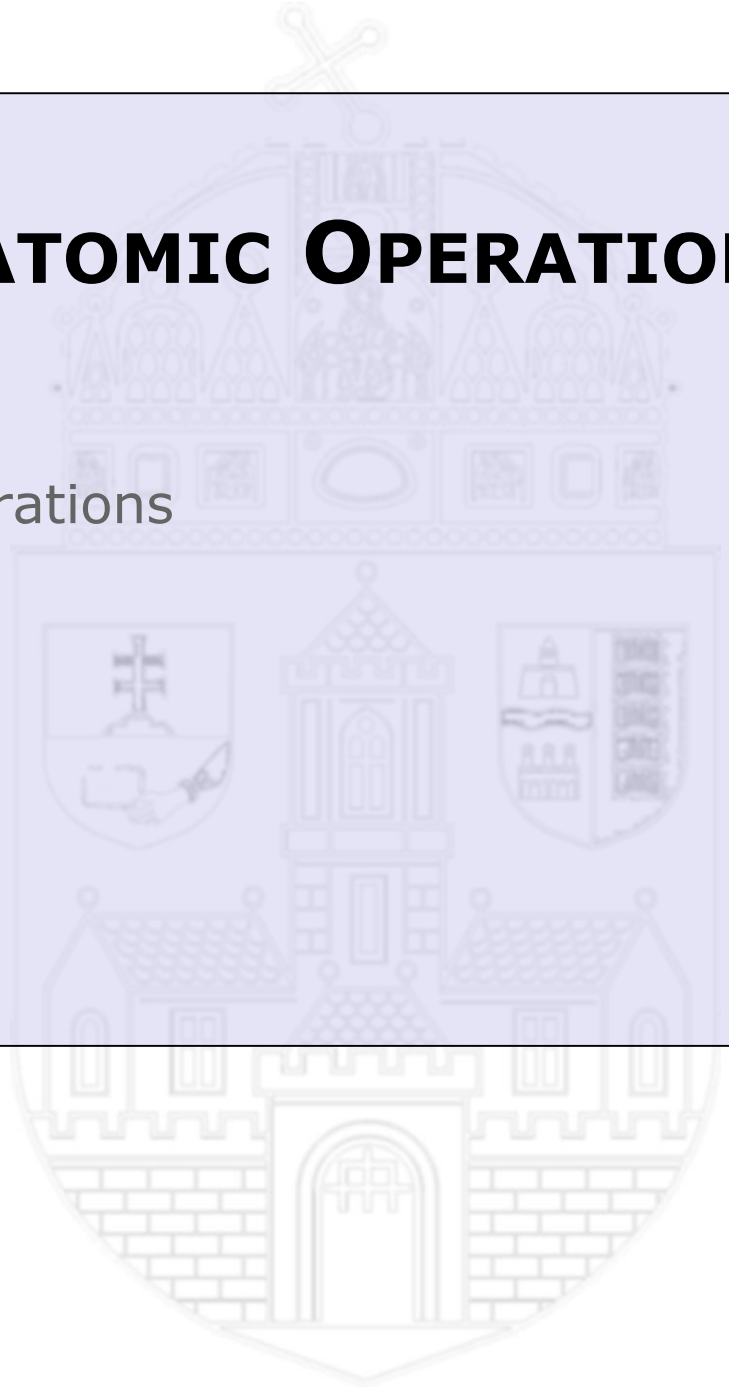
Warp vote functions (≥CC1.2)

- int __all(int condition)
  Evaluates predicate for all threads of the warp and returns non-zero if and only if predicate evaluates to non-zero for all of them
- int __any(int condition)
  Evaluates predicate for all threads of the warp and returns non-zero if and only if predicate evaluates to non-zero for any of them

Atomic Operations

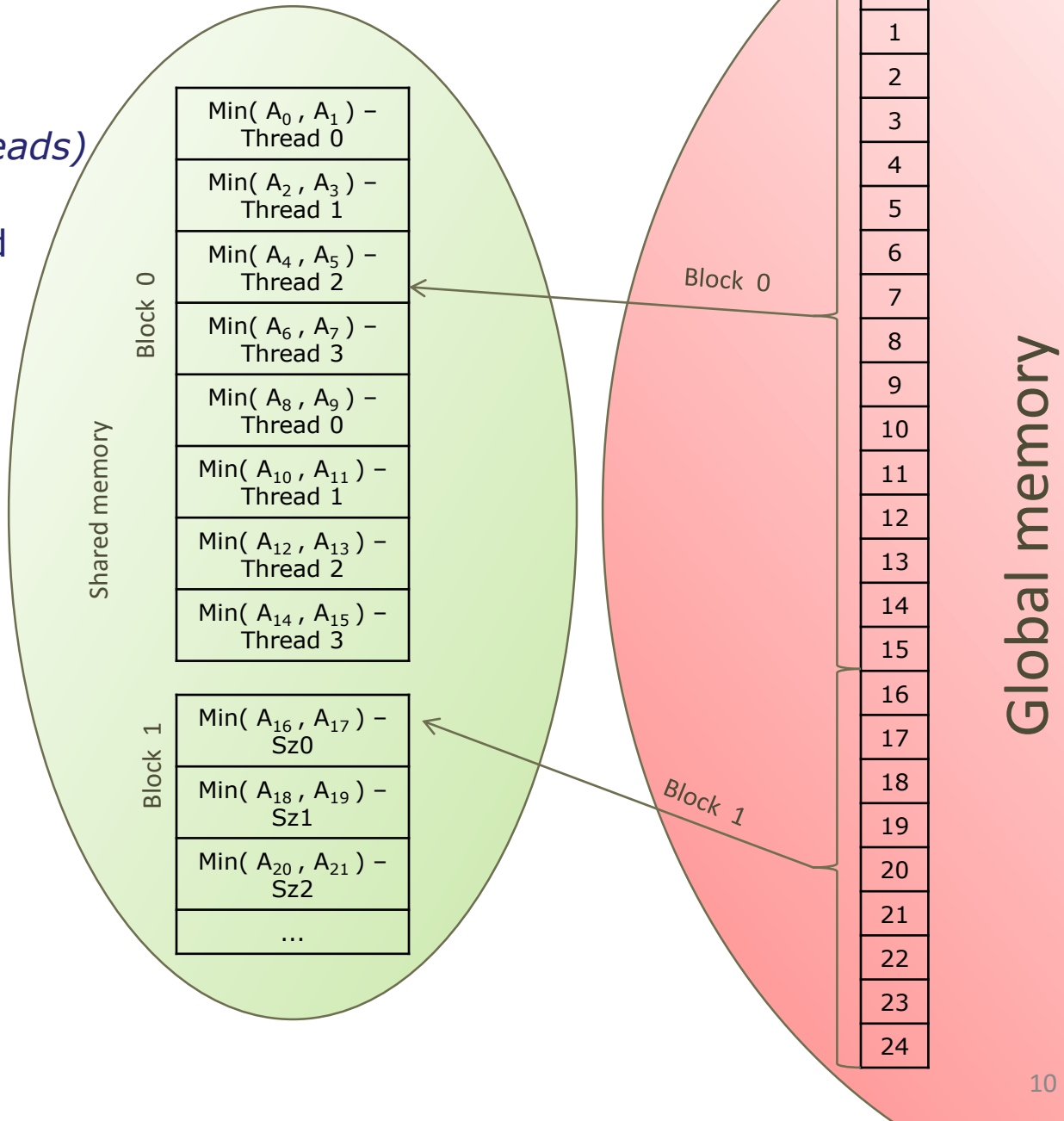
Kernel level operations

Fast reduction

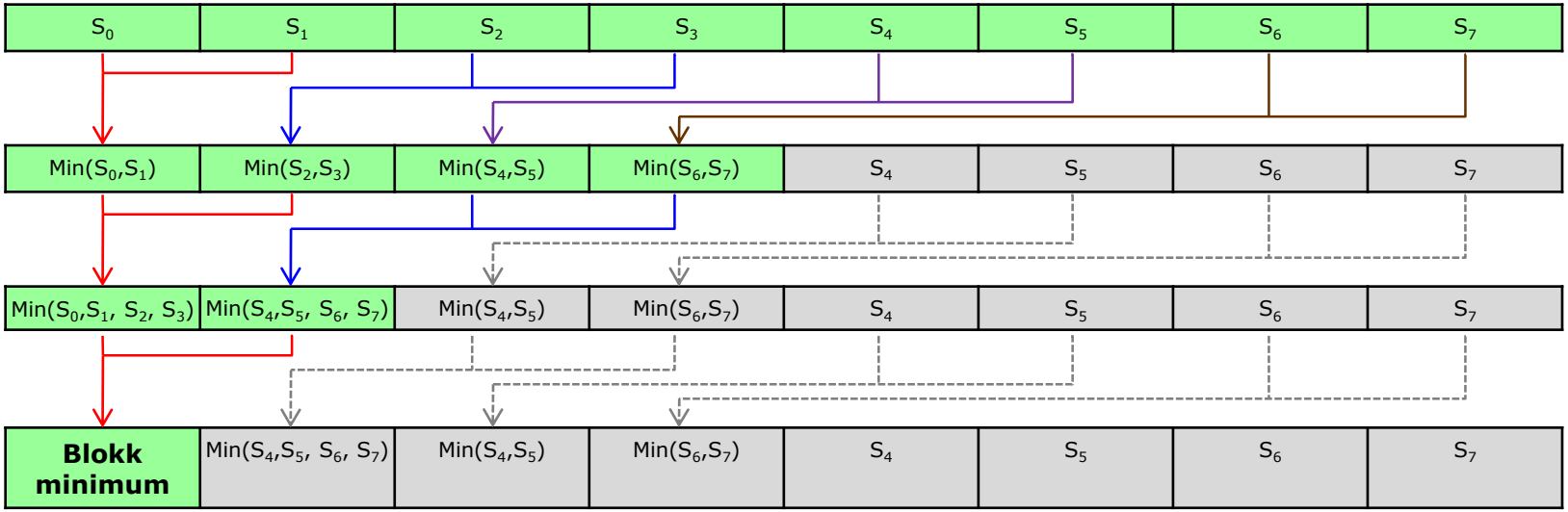


Parallel minimum search - loading

- One example
  N = 24
  BlockN = 4 (nbr of threads)
- Every block allocate one array in the shared memory (size is BlockN*2)
- Every thread in every blocks load 2 values from the global memory and stores the smaller one in shared memory
- If we have empty spaces we have to fill them with some values
- Synchronization



Parallel minimum search – find minimum of block



- Every thread do $log_2 BlockN$ number of iterations. In every iteration threads do the following operation:
  $S_x = Min(S_{2x} , S_{2x+1})$
- At the end of the last iteration, the first value of the array will be the smallest one
- After that we find the globally minimum
  - using atomic instructions
  - we store the minimum values of blocks into another vector and redo the minimum search to this vector

Parallel minimum search - kernel

```
1  __global__ static void MinSearch(int *devA) {
2      __shared__ int localMin[BlockN*2];
3      int blockSize = BlockN;
4      int itemc1 = threadIdx.x * 2;
5      int itemc2 = threadIdx.x * 2 + 1;
6
7      for(int k = 0; k <= 1; k++) {
8        int blockStart = blockIdx.x * blockDim.x * 4 + k * blockDim.x * 2;
9        int loadIndx = threadIdx.x + blockDim.x * k;
10       if (blockStart + itemc2 < N) {
11         int value1 = devA[blockStart + itemc1];
12         int value2 = devA[blockStart + itemc2];
13         localMin[loadIndx] = value1 < value2 ? value1 : value2;
14       } else
15         if (blockStart + itemc1 < N)
16                 localMin[loadIndx] = devA[blockStart + itemc1];
17         else
18                 localMin[loadIndx] = devA[0];
19     }
20     __syncthreads();
```

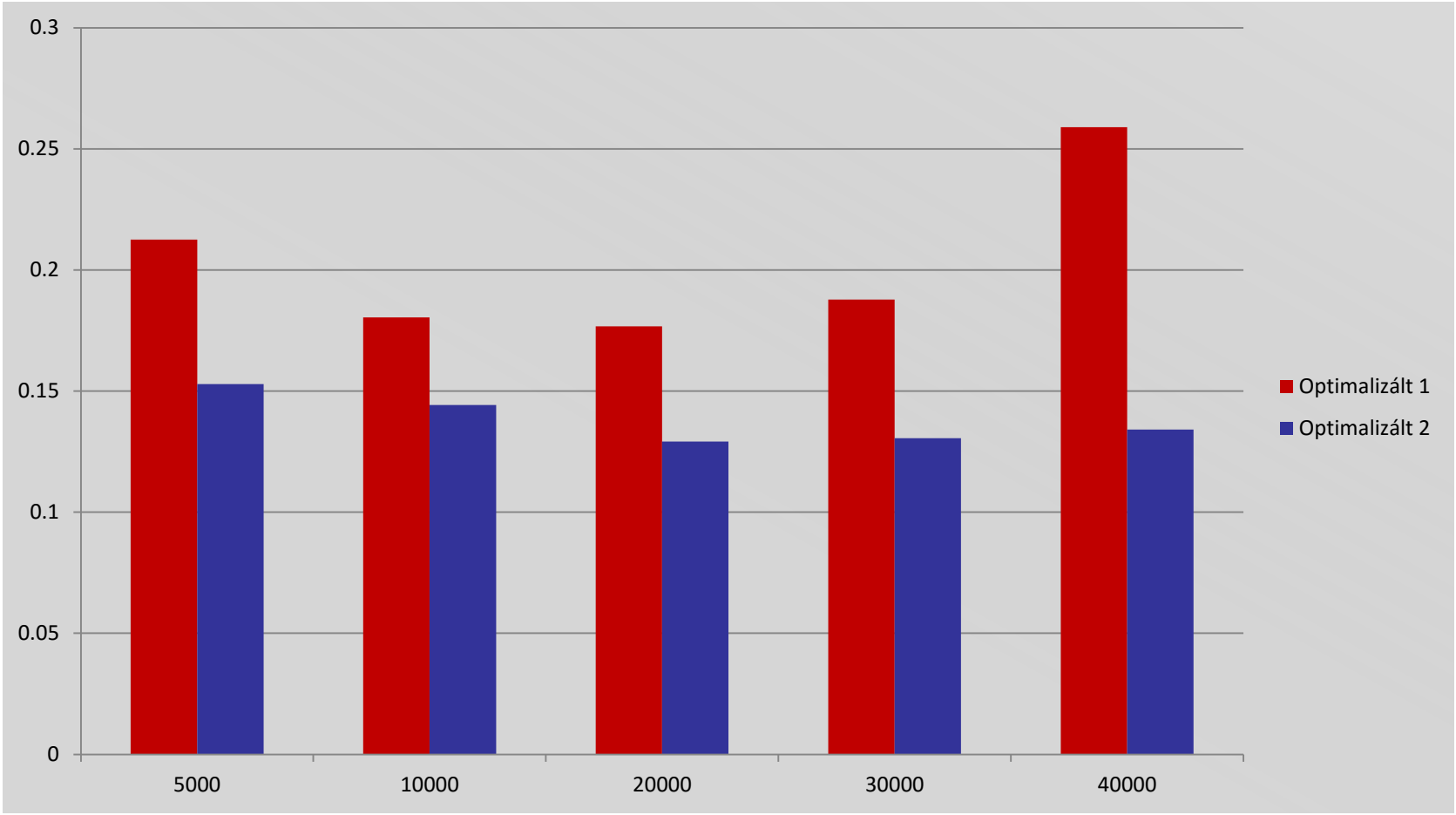
Parallel minimum search – kernel (2)

```
21      while (blockSize > 0) {
22         int locMin = localMin[itemc1] < localMin[itemc2] ? localMin[itemc1] : localMin[itemc2];
23         __syncthreads();
24         localMin[threadIdx.x] = locMin;
25         __syncthreads();
26         blockSize = blockSize / 2;
27      }
28      if (threadIdx.x == 0) atomicMin(devA, localMin[0]);
29   }
```

- A more optimized version is available at *http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf*
- Block size must be 2^N

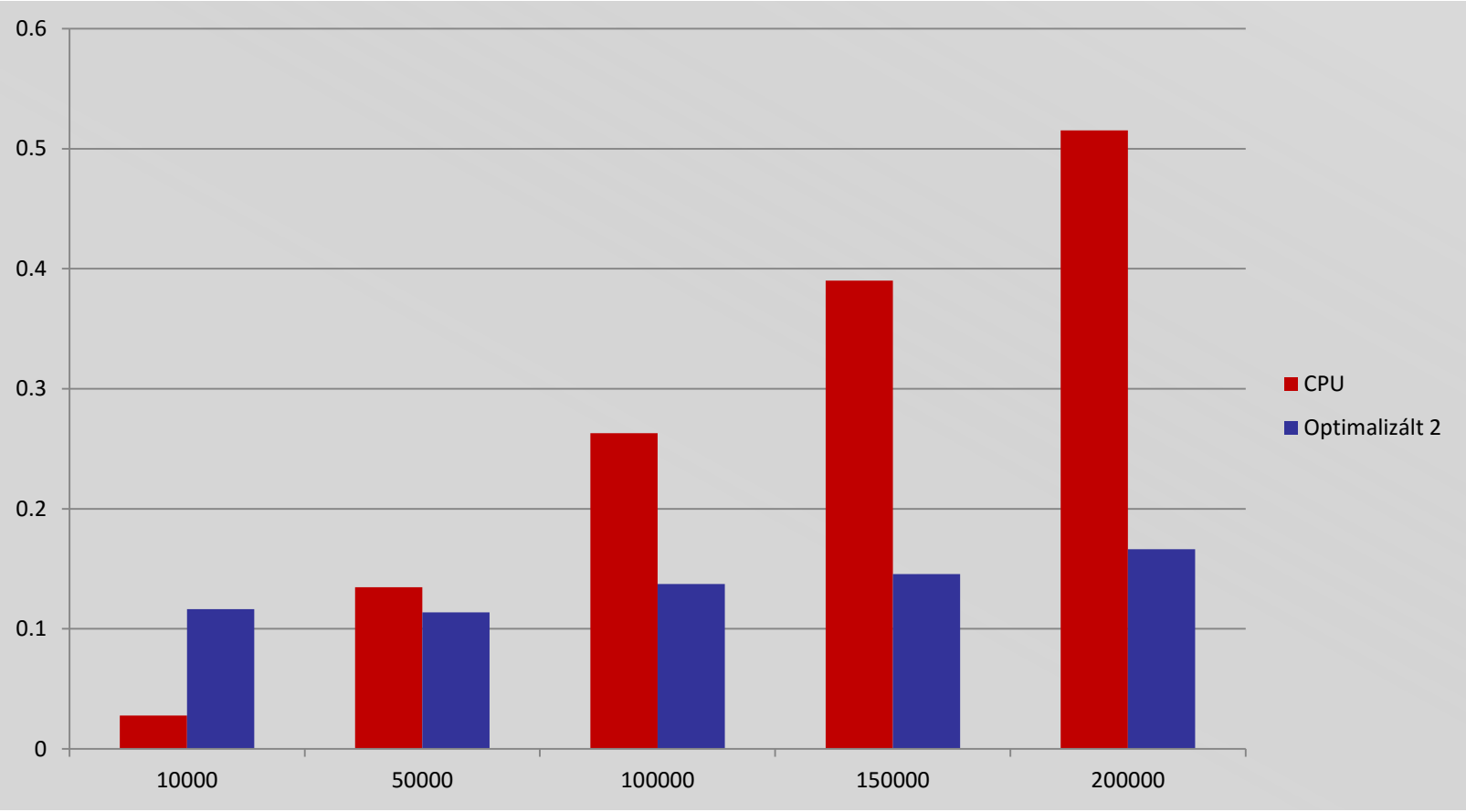
Comparing runtime of atomic and parallel version

- Horizontal axis: size of vector (N)
- Vertical axis: runtime (second)



Comparing of CPU and GPU implementation

- Horizontal axis: size of vector (N)
- Vertical axis: runtime (second)



Values do not contain transfer time from CPU to GPU!