Katholieke
Universiteit
Leuven

Department of
Computer Science

GEAVANCEERDE COMPUTERARCHITECTUUR

Lab 2

Thibaut Beck
Wannes Paesschesoone

Academic year 2025–2026

Contents

1 Report Overview

This report compares three methods for finding the maximum number in an array. The first method is a simple CPU-based iterative approach, the second uses CUDA's atomic operations, and the third applies a parallel reduction approach to compute the maximum.

2 Algorithm 1: CPU Iterative Method

2.1 Is the result correct?

The algorithm produces the correct result, it iterates through each element of the array, compares it with the current maximum, and updates the maximum when a larger value is found. Because the process is sequential and deterministic, the same correct result is obtained each time.

2.2 How does time increase as a function of input size?

Array Size	**Tijd (microseconds)**
128	53
256	85
512	163
1024	361
2048	719
4096	1113

Table 1: Tijd (in microseconds) als functie van array size

2.3 What happens if the input is larger than 2048?

Increasing N beyond 2048 poses no issues, as the CPU implementation does not depend on CUDA thread or block dimensions. Its only constraints are system memory capacity and the linear growth of computation time.

2.4 Calling the kernel with fewer threads

Not applicable, as this algorithm runs entirely on the CPU without any kernel or thread-based parallelism. All operations are executed sequentially.

2.5 Tackling a 2D array

For a two-dimensional array, the algorithm can either flatten the array into a single 1D representation or use nested loops over rows and columns. Both approaches ensure that every element is compared, yielding the same global maximum.

2.6 Code

```cpp
#include <iostream>
#include <chrono>

int main() {
    //const int N = 10;
    //int arr[N] = {5, 23, 23, 234, 54, 233, 23, 54, 65, 25};
    int max = 0;

    const int N = 4096;
    int arr[N];
     for (int i = 0; i < N; i++) {
        arr[i] =i;
    }

    // Start measuring time
    auto start = std::chrono::high_resolution_clock::now();

    std::cout << "Array:";
    for (int i = 0; i < N; i++) {
        std::cout << "␣" << arr[i];
        max = std::max(max, arr[i]);
    }

    // Stop measuring time
    auto end = std::chrono::high_resolution_clock::now();

    std::cout << std::endl;
    std::cout << "The␣max␣is:␣" << max << std::endl;

    // Calculate the duration and print it
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    std::cout << "Time␣taken:␣" << duration.count() << "␣microseconds" << std::endl;

    return 0;
}
```

Listing 1: CPU Iterative Maximum

3 Algorithm 2: Atomic Maximum

3.1 Is the result correct?

The result of the atomicMax algorithm is correct as long as each thread processes a unique index of the array and the atomicMax operation is used. This function ensures that updates to the global maximum are performed safely, preventing race conditions even when multiple threads attempt to modify the same variable simultaneously. In essence, the atomicMax operation acts like a lock on the variable being written to, guaranteeing that only one thread can update it at a time.

3.2 How does time increase as a function of input size?

The execution time remains stable for input sizes up to 2048 elements. When using more elements, the execution time increases sharply because the kernel configuration exceeds the maximum number of 1024 threads per block. This limit results in invalid or inefficient kernel launches, leading to poor performance and serialization effects.

Array Size	Tijd (microseconds)
128	88
256	88
512	85
1024	89
2048	4540
4096	31745

Table 2: Tijd (in microseconds) als functie van array size (tweede dataset)

3.3 What happens if the input is larger than 2048?

When the input size exceeds 2048 elements, the execution time increases dramatically, as shown in Table 2. As explained above, this is caused by exceeding the hardware limit of 1024 threads per block, resulting in inefficient kernel launches. Additionally, multiple thread blocks must perform atomic updates on the same global variable, leading to severe contention and further performance degradation.

For 4096 elements, the kernel experiences a significant slowdown, and the correct maximum value is not found. This issue typically arises because the number of launched threads is insufficient to cover all elements, causing some data to remain unprocessed. To handle larger inputs correctly, either the grid and block dimensions must be increased to match the input size, or each thread must process multiple elements through striding (e.g., using a loop over `i += totalThreads`).

3.4 Calling the kernel with fewer threads

When the kernel is launched with fewer threads than elements, each thread must handle multiple data points to ensure full coverage. This is commonly implemented using a stride loop:

```
for (int i = idx; i < N; i += gridDim.x * blockDim.x)
    atomicMax(p_max, list[i]);
```

3.5 Tackling a 2D array

For two-dimensional data, each thread can be mapped to a (x, y) coordinate using `threadIdx.x`, `threadIdx.y`, and their corresponding block indices. Each thread can then compute a local maximum for its region, and `atomicMax` combines these local results into one global maximum value. Another solution is flatten the array, so we execute the function again on a 1D array.

3.6 Code

```
#include <iostream>
#include <cuda_runtime.h>
#include <chrono>
__global__ void atomic_max(int* list, int* p_max, int N){
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < N)
        atomicMax(p_max, list[idx]);
}

int main() {
    const int N = 128;
    int arr[N];
     for (int i = 0; i < N; i++) {
        arr[i] =i;
    }

    int *p_arr, *p_max, maxNum = 0;

    cudaMalloc((void**)&p_arr, sizeof(int)*N);
    cudaMalloc((void**)&p_max, sizeof(int));
    cudaMemcpy(p_arr, arr, sizeof(int)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(p_max, &maxNum, sizeof(int), cudaMemcpyHostToDevice);

    int numBlocks = 2;
    int threadsBlock = N / numBlocks;

    auto start = std::chrono::high_resolution_clock::now();

    atomic_max<<<numBlocks, threadsBlock>>>(p_arr, p_max, N);

    cudaDeviceSynchronize(); // ensure all gpu work is done

    // Stop measuring time
    auto end = std::chrono::high_resolution_clock::now();

    cudaMemcpy(&maxNum, p_max, sizeof(int), cudaMemcpyDeviceToHost);
    std::cout << "The␣max␣number␣in␣the␣array␣is:␣" << maxNum << std::endl;

    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    std::cout << "Time␣taken:␣" << duration.count() << "␣microseconds" << std::endl;

    cudaFree(p_arr);
    cudaFree(p_max);

    return 0;
}
```

Listing 2: CUDA Atomic Maximum

4 Algorithm 3: Reduction Maximum

4.1 Is the result correct?

The result is correct when N is a power of two and synchronization between threads is properly enforced. The reduction algorithm operates by iteratively comparing pairs of elements in parallel, halving the number of active threads at each step until only one maximum value remains. If synchronization barriers are omitted, some threads may read outdated values from shared memory, which can lead to incorrect results.

4.2 How does time increase as a function of input size?

Array Size	Microseconds
128	101
265	108
512	100
1024	108
2048	113
4069	113

Table 3: Microseconds als functie van array size (laatste dataset)

4.3 What happens if the input is larger than 2048?

No fundamental issues occur, as the reduction can be extended across multiple blocks. Each block computes a local maximum, and a second kernel call can combine these local results into the final global maximum.

4.4 Calling the kernel with fewer threads

If fewer threads are launched than data elements, each thread should process multiple elements. This can be achieved by loading several values per thread and performing a local reduction before contributing to the shared result. Such an approach keeps all threads efficiently utilized and avoids idle computation.

4.5 Tackling a 2D array

For a two-dimensional array, the reduction can be applied first along one dimension (e.g., rows) to produce partial maxima, and then along the other dimension (columns) to find the global maximum. This hierarchical reduction maintains parallel efficiency while extending naturally to multidimensional data. Another solution is flatten the array again.

4.6 Code

```
#include<iostream>
#include <cuda_runtime.h>
#include <cstdlib>   // Voor rand() en srand()
#include <ctime>     // Voor time()
#include <chrono>

__global__ void reduction_max(int *list, int* maxNum, int N){
    int tid = threadIdx.x + blockDim.x * blockIdx.x;

    for (int offset = 1; offset < N; offset *= 2){
        int idx = tid * 2 * offset;
        if (idx + offset < N){
            atomicMax(&list[idx], list[idx + offset]);
        }
        __syncthreads();
    }

    if (tid == 0)
        *maxNum = list[tid];
}

int main() {
    const int N = 128; // Should be 2^x
    int arr[N];
    //srand(static_cast<unsigned int>(time(0)));

    // Vul de array met random getallen (0 t.e.m. 1000 bijvoorbeeld)
    for (int i = 0; i < N; i++) {
    //  arr[i] = rand() % 200000; // Willekeurig getal tussen 0 en 200000
    arr[i] = i;
    }
    int maxNum = 0;
    int *p_arr, *p_maxNum;

    cudaMalloc((void**)&p_arr, sizeof(int)*N);
    cudaMalloc((void**)&p_maxNum, sizeof(int));
    cudaMemcpy(p_arr, arr, sizeof(int)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(p_maxNum, &maxNum, sizeof(int), cudaMemcpyHostToDevice);

    int numBlocks = 4;
    int blocksPerThread = N / (numBlocks*2);

    std::cout << "Numblocks:␣" << numBlocks << "␣and␣BlocksPerThread:␣" << blocksPerThread << std::endl;
    auto start = std::chrono::high_resolution_clock::now();
    reduction_max<<<numBlocks, blocksPerThread>>>(p_arr, p_maxNum, N);
    cudaDeviceSynchronize(); // ensure all gpu work is done
    cudaMemcpy(&maxNum, p_maxNum, sizeof(int), cudaMemcpyDeviceToHost);

    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    std::cout << "Time␣taken:␣" << duration.count() << "␣microseconds" << std::endl;
    std::cout << "The␣max␣num␣is:␣" << maxNum << std::endl;

    cudaFree(p_arr);
    cudaFree(p_maxNum);
    return 0;
}
```

Listing 3: CUDA Reduction Maximum

5 Conclusion

This report compared three different methods for finding the maximum value in an array: a sequential CPU algorithm, a GPU-based atomic maximum, and a parallel reduction approach. The CPU iterative method is simple, reliable, and shows predictable linear time growth with input size. It is well-suited for smaller datasets or situations where GPU acceleration is not required. The atomic maximum approach leverages GPU parallelism but exhibits a sharp increase in execution time for larger input sizes. This is due to contention on the atomic variable and insufficient thread coverage when the grid and block configuration is not scaled appropriately. While atomic operations ensure correctness, they can become a bottleneck under heavy parallel load. The reduction algorithm demonstrates stable and efficient performance even for larger inputs. By reducing the number of active elements at each iteration, it minimizes synchronization overhead and avoids excessive atomic operations. It can also be extended to multi-block reductions and multidimensional data with minimal adjustments.

In summary, while all three methods produce correct results, their efficiency differs greatly. For small arrays, the CPU method is sufficient. For medium arrays with simple parallelization, the atomic approach works but can become inefficient. For large datasets or performance-critical applications, the reduction algorithm is the preferred solution due to its scalability and efficiency.