GPU Computing: Data-Parallel Algorithms

Dipl.-Ing. Jan Novák* Dipl.-Inf. Gábor Liktor† Prof. Dr.-Ing. Carsten Dachsbacher‡

Abstract

In this assignment we will focus on two fundamental data-parallel algorithms that are often used as building blocks of more advanced and complex applications. We will address the problems of parallel reduction and parallel prefix sum (scan). These techniques can be implemented in a very straightforward way, however, by optimizing them further we can achieve up to an order of magnitude higher performance.

We will also introduce theoretical measures, e.g. parallel work, that can classify whether the parallel algorithm is optimal or not. As performance is the main motivation throughout the assignment we will also introduce the basics of GPU profiling.

The deadline: 14:00, 18.05.2011.

1 Performance Metrics of Parallel Algorithms

When evaluating the cost of sequential algorithms, we usually classify them using complexity metrics such as (asymptotically) *optimal* algorithm, or (asymptotically) *best known* solution (not optimal, yet the best we know). The analysis of complexity of parallel algorithms has one additional aspect to be considered: the number of processing units (PU). Having p PUs we expect the parallel algorithm to perform ideally p times faster than its sequential counterpart, achieving a *linear* speedup. As this is often impossible (some algorithms are proven to have worse complexity when evaluated in parallel), we need additional metrics to classify the quality of parallel algorithms.

1.1 Parallel Time

Parallel time $T(n, p)$, where n is the size of the input and p is the number of processors, is the time elapsed from the beginning of the algorithm till the moment when the last processor finishes the computation. Instead of using seconds, we often express parallel time by counting the number of parallel computation steps.

The speedup of a parallel over sequential implementation can be expressed as $T(n, 1)/T(n, p)$. As mentioned earlier, mostly we wish to achieve *linear* speedup (p times faster performance), but in some cases we can even experience *super-linear* speedup, e.g. when the parallel algorithm better matches the hardware characteristics, or the parallel computation takes greater advantage of caching.

1.2 Parallel Cost

Parallel cost $C(n, p)$ can be expressed as the product of the parallel time and the number of processors: $C(n, p) = p \times T(n, p)$. It gives us the total number of operations as if all the processors were working during the entire computation, which is not always the case as some processors can idle.

We consider a parallel algorithm to be *cost-optimal* if the parallel cost asymptotically equals to the sequential time. In other words, the total sequential time is uniformly divided in between all processors, all taking approximately the same number of steps.

1.3 Parallel Work

Parallel work $W(n, p)$ measures the actual number of the executed parallel operations. It can be also expressed as the sum of the number of active processors over all parallel steps. In the context of GPUs we can define parallel work more precisely: let p_i be the number of active processors in step i, then $W(n, p) = p_1 + p_2 + \ldots + p_k$, where k is the total number of parallel steps.

A *work-optimal* algorithm performs asymptotically as many operations as its sequential counterpart. Notice that parallel work does not consider the possible idleness of individual processors. If an algorithm is cost-optimal it will always also be work-optimal, but not vice-versa. You will find examples of work-optimal (but not cost-optimal) algorithms in the reduction and prefix sum assignments.

2 Performance Optimization

The previous assignment described the basic concepts of GPU programming that were then demonstrated using two very simple OpenCL tasks. In this assignment, we introduce further architectural constraints that become important when tuning the performance. Previously, we have briefly mentioned the concept of *coalescing*: aligned access to the global memory that enables partial hiding of memory latency. We also talked about the very fast *local memory*, allowing user-controlled caching of items. We will further detail these features of modern GPUs and add a few more hints for increasing the throughput, e.g. unrolling of loops.

There are parallel optimization strategies that can only justify themselves after a closer examination of the GPU memory architecture. While the ultimate goal of parallel programming standards (e.g. OpenCL) is to hide the hardware and provide the programmer with a high level abstraction, there is a level of performance that can only be achieved if the program fully exploits capabilities of the architecture - and this holds especially for designing memory access patterns.

2.1 Memory Coalescing and Warps

While changing the memory access pattern of a given parallel implementation will not change its algorithmic complexity, it can result in significant speedup. The explanation is that the global memory on the GPU is really slow: on current hardware a single memory read or write operation can take as many as 400 clock cycles (a typical logical or arithmetic operation consumes 2-8 cycles). Waiting for the result of a memory operation appears as *latency* in the execution. If the number of threads executed on the same multiprocessor is high enough, the hardware can effectively hide the latency by scheduling other threads for execution while the current thread waits for data from the memory. Therefore, minimizing global memory accesses and maximizing the number of threads per multiprocessor is crucial.

A key to minimize global memory accesses is memory coalescing. The memory interface of each streaming multiprocessor can load or store multiple data elements in parallel. On the CUDA architecture it means that instead of executing a single load / store operation per thread sequentially, the device memory is accessed via 32-, 64-, or 128-byte memory transactions. By organizing your memory accesses to address items in 32-, 64-, or 128-byte segments of memory, the number of load/store transactions can reduce significantly.

*e-mail: jan.novak@kit.edu

†e-mail: gabor.liktor@kit.edu

‡e-mail: dachsbacher@kit.edu

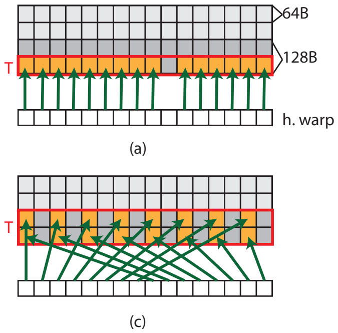


Figure 1: *Examples of memory coalescing when accessing aligned blocks of memory by a half warp on the latest CUDA architecture. (a): single 64-byte transaction. (b): unaligned access of an aligned 128-byte block still results in a single 128-byte transaction. (c): sequential access of a 128-byte-aligned address with stride 2 results in a single 128-byte transaction. (d): sequential access with stride 3 results in one 128-byte transaction and one 64-byte transaction.*

CUDA scheduler executes threads in groups of 32 parallel threads, called *warps*. You can imagine a warp as the smallest unit of parallel execution on the device: each thread in a specific warp executes the same instruction. If there was a divergence within a warp, e.g. half of the threads fulfilled the conditions of an *if* clause, while the other half continued to the *else* clause, all the threads within the warp execute both of the branches in the code serially, disabling those threads in the warp that are not within the corresponding branch. Note, that threads in different warps can take arbitrary execution paths without loss in the instruction throughput. Optimizing your code to get coherent execution based on warps is really simple by using the local indexing of threads (*get_local_id(0)* in OpenCL kernels): threads 0-31 belong to the first warp, threads 32-63 to the second warp, etc.

Knowing about warps also helps the programmer to achieve coalesced memory accesses. Ideally, the memory interface of a streaming multiprocessor can perform global read/write operations for 16 threads (a half warp) in a single transaction, if all addresses in the half warp fall into the same aligned segment of memory. This is a 16x speedup compared to the worst case, where all accesses are unaligned. Older architectures required these accesses to also be sequentially aligned according to the thread indices in the warp. The later CUDA capability 2.0 architecture can also coalesce "shuffled" access patterns inside aligned memory segments, see Figure 1 for examples. For a more precise description, refer to the NVIDIA OpenCL Best Practices Guide.

2.2 Local Memory

We have already seen a simple example in the previous assignment (matrix rotation) where memory coalescing for both load and store operations was inherently not possible without local data exchange between threads. We have used a fast on-chip memory to provide an intermediate storage, so the threads could perform coalesced write to the global memory. Here we describe how the OpenCL local memory maps to the CUDA architecture.

The local memory (or *shared memory* in CUDA terminology) is very fast, in terms of speed on par with registers. However, access patterns play an important role again, if we want to reach the maximum throughput. We should not think of local memory as a opaque block of registers that could be randomly accessed, but rather as successive words of the same size, aligned to *banks*. The current CUDA architecture (Fermi) partitions the local memory into 32 banks of 32-bit words. Older NVIDIA GPUs were using 16 banks. A linear array placed in the local memory is organized into banks in the following manner: the first word is placed in bank 0, second word in bank 1, and so on up to the 32nd word that falls into the last bank with number 31. The array is then wrapped and the 33rd word will be placed in bank 0 again (below the first word).

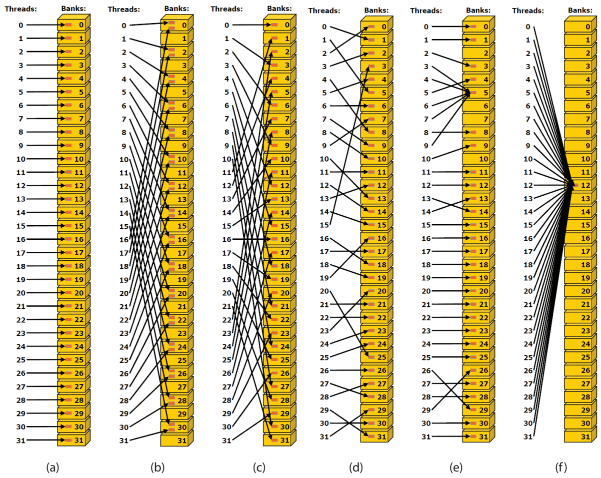


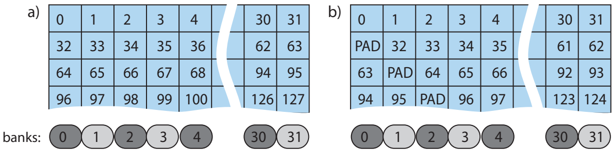
Figure 2: *Without bank conflicts, the on-chip local memory can operate at its maximum throughput. Examples without bank conflicts: linear addressing aligned to the warps (a), linear addressing with stride of 3 32-bit words (c), random permutation of addresses, each using different banks (d). Interestingly (e) and (f) are also conflict-free, as multiple threads read the same address through the same bank, where a single memory broadcast is performed. Using linear addressing with stride of 2 words, however causes 2-way bank conflicts (b). The image is courtesy of NVIDIA.*

An important feature is that each bank can process only a single memory request at the time (except for the broadcast mechanism). If every thread within the same warp accesses a word in different banks, the local memory can operate at its maximum throughput. However, if two or more threads in the same warp operate on different 32-bit words belonging to the same bank, their instructions will be serialized, as the bank can perform one operation in the same time. We call this situation *bank conflict*. In the worst case, all 32 threads of the warp accesses the same bank, which results in a 32-way conflict creating a potential performance bottleneck. An exception to the previously described rule is when the threads read from the same position in the bank. Then the hardware can perform a single load operation and broadcasts the result among all participating threads, without performance issues. Figure 2 illustrates examples where bank conflicts occur and access patterns avoiding such conflicts.

Now please refer back to the previous assignment for a brief optimization. When loading elements to a tile in the local memory, the following code snippet was used:

```
block[ LID.y * get_local_size(0) + LID.x ] = M[ GID.y *
     SizeX + GID.x ];
```

When the work-items start writing data back to global memory, a warp of 32 threads accesses the same column of the tile at the same time. Note, that in the worst case - depending on



a)

0	1	2	3	4	30	31
32	33	34	35	36	62	63
64	65	66	67	68	94	95
96	97	98	99	100	126	127

b)

0	1	2	3	4	30	31
PAD	32	33	34	35	61	62
63	PAD	64	65	66	92	93
94	95	PAD	96	97	123	124

Figure 3: *By inserting a padding word after each 32 words, we can completely eliminate bank conflicts from the kernel rotating a matrix tile in the local memory. For simplicity, we use the local indexing of a 32x32 tile to demonstrate the realigned positions of the elements.*

get_local_size - this can create a 32-way bank conflict if all elements in the column lie in the same bank. The solution is to introduce padding to the local data array: after each 32nd element we insert an empty element. This enables rotating each row of the tile without bank conflicts (also see Figure 3):

```
int index = LID.y * get_local_size(0) + LID.x;
int offset = index / NUM_BANKS;
block[index + offset] = M[ GID.y * SizeX + GID.x ];
```

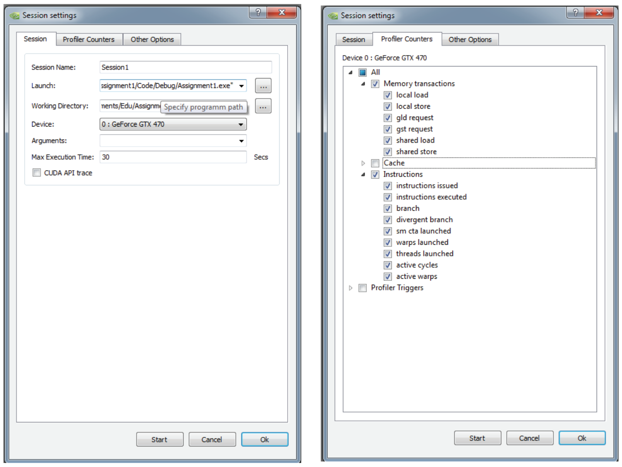
This is a practice you should generally follow when placing data in the local memory. The padding of data in the memory always depends on the number of banks. Unfortunately, this number can be different on various architectures, so you always need to check the specification or appropriate programming guide.

3 Profiling Basics

Once your parallel OpenCL implementation outputs valid results, your solution is algorithmically correct. However, it is still possible that you are not utilizing properly the hardware resources, and the code could perform much better using a few simple modifications, e.g. rearranging the memory accesses according to Sect. 2.1 and Sect. 2.2. By measuring the execution time of the kernel, it is possible to ”blindly” modify the code and see if the performance improved. Instead of such empirical experiments, you should use a profiler. GPU vendors offer dedicated profilers that can gather detailed information from the GPU driver by querying hardware performance counters. The profiling results provide the programmer with fine-grained information about the performance issues of kernels.

In order to profile CUDA applications (OpenCL and CUDA compiles to the same intermediate language on CUDA hardware), we can use the NVIDIA Compute Visual Profiler, which is automatically installed with the CUDA Toolkit. Here we briefly overview the main features of this useful application. The profiler is an application that gets set up between your program and the OpenCL driver, so it is able to catch any command enqueued to the command queue and gather performance data during the execution of commands. For this reason, you will need to launch your application inside the profiler, instead of directly running the executable itself.

Having created a new profiler project, you can run your application in the scope of a new session. When creating the session, you can select the path of your executable, set the working directory (this is important, as you probably load your kernel code from there), and select the performance counters you would like to capture. Figure 4 shows a possible setup.



Session settings

Session | Profiler Counters | Other Options

Session Name: Session1
Launch: ssignment1/Code/Debug/Assignment1.exe"
Working Directory: nents/Edu/Assignm Specify programm path
Device: 0 : GeForce GTX 470
Arguments:
Max Execution Time: 30 Secs
☐ CUDA API trace

Session settings

Session | Profiler Counters | Other Options

Device 0 : GeForce GTX 470

- ■ All
  - ☑ Memory transactions
    - ☑ local load
    - ☑ local store
    - ☑ gld request
    - ☑ gst request
    - ☑ shared load
    - ☑ shared store
  - ☐ Cache
  - ☑ Instructions
    - ☑ instructions issued
    - ☑ instructions executed
    - ☑ branch
    - ☑ divergent branch
    - ☑ sm cta launched
    - ☑ warps launched
    - ☑ threads launched
    - ☑ active cycles
    - ☑ active warps
- ☐ Profiler Triggers

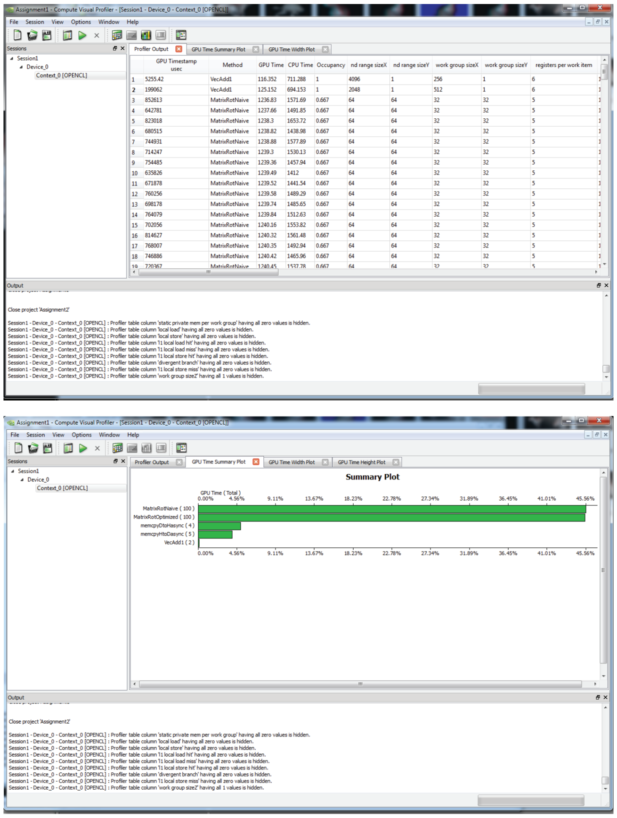
Figure 4: *Creating a new profiler session capturing performance counters of memory transactions and instructions.*

Once you are done with the setup, you can start the actual profiling. It will take longer than a standard execution, as the profiler usually needs to run your code in multiple passes. After the execution is complete you are presented with a table containing a row for all your kernel calls and (optionally) memory transactions. Each performance counter has a column assigned, so the performance of each kernel can be easily evaluated (Figure 5). Some of the important columns in this table:

- **GPU and CPU time** As their name suggests these are the execution times of the specific method on the CPU and the GPU, respectively.
- **Occupancy** A very important feature, which informs you how many threads are executed on a single streaming multiprocessor (SM). Occupancy is the ratio of active warps per multiprocessor to the maximum number of active warps. If this number is less than one, the scheduler of your GPU has unused capacity. The reason for this can be that your kernel uses too many registers, too much of local memory, or you simply do not have enough threads in your NDRange.
- **l1 shared bank conflicts** The number of bank conflicts in the OpenCL local memory during kernel execution. You should always examine this value when using local memory. Shared bank conflicts can be completely eliminated in most practical cases.

Another useful view is the summary plot, which summarizes the performance statistics of your kernels, and displays them as a bar chart showing the GPU time distribution among specific kernels.

For ATI GPUs, you can similarly employ the ATI Stream Profiler Tool that is distributed together with the ATI Stream SDK.



	GPU Timestamp usec	Method	GPU Time	CPU Time	Occupancy	nd range sizeX	nd range sizeY	work group sizeX	work group sizeY	registers per work item	
1	5255.42	VecAdd1	116.352	711.288	1	4096	1	256	1	6	1
2	199062	VecAdd1	125.152	694.153	1	2048	1	512	1	6	1
3	852613	MatrixRotNaive	1236.83	1571.69	0.667	64	64	32	32	5	1
4	642781	MatrixRotNaive	1237.66	1491.85	0.667	64	64	32	32	5	1
5	823018	MatrixRotNaive	1238.3	1653.72	0.667	64	64	32	32	5	1
6	680515	MatrixRotNaive	1238.82	1438.98	0.667	64	64	32	32	5	1
7	744931	MatrixRotNaive	1238.88	1577.89	0.667	64	64	32	32	5	1
8	714247	MatrixRotNaive	1239.3	1530.13	0.667	64	64	32	32	5	1
9	754485	MatrixRotNaive	1239.36	1457.94	0.667	64	64	32	32	5	1
10	635826	MatrixRotNaive	1239.49	1412	0.667	64	64	32	32	5	1
11	671878	MatrixRotNaive	1239.52	1441.54	0.667	64	64	32	32	5	1
12	760256	MatrixRotNaive	1239.58	1489.29	0.667	64	64	32	32	5	1
13	698178	MatrixRotNaive	1239.74	1485.65	0.667	64	64	32	32	5	1
14	764079	MatrixRotNaive	1239.84	1512.63	0.667	64	64	32	32	5	1
15	702056	MatrixRotNaive	1240.16	1553.82	0.667	64	64	32	32	5	1
16	814627	MatrixRotNaive	1240.32	1561.48	0.667	64	64	32	32	5	1
17	768007	MatrixRotNaive	1240.35	1492.94	0.667	64	64	32	32	5	1
18	746886	MatrixRotNaive	1240.42	1465.96	0.667	64	64	32	32	5	1
19	720367	MatrixRotNaive	1240.45	1537.78	0.667	64	64	32	32	5	1

```
Close project 'Assignment2'

Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'static private mem per work group' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'local load' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'local store' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'l1 local load hit' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'l1 local load miss' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'l1 local store hit' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'divergent branch' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'l1 local store miss' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'work group sizeZ' having all 1 values is hidden.
```

Close project 'Assignment2'

Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'static private mem per work group' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'local load' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'local store' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'l1 local load hit' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'l1 local load miss' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'l1 local store hit' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'divergent branch' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'l1 local store miss' having all zero values is hidden.
Session1 - Device_0 - Context_0 [OPENCL] : Profiler table column 'work group sizeZ' having all 1 values is hidden.

Figure 5: *The Compute Visual Profiler Tool displays detailed report on gathered performance counters in the Profiler Output. The Summary Plot can be helpful to identify the most expensive kernels that could be the bottleneck of your application.*

4 Task 1: Parallel Reduction

4.1 Algorithm Description

Parallel reduction is a data-parallel algorithm that is used to solve the problem of reducing all elements in the input array into a single value. An integral part of the problem is an *associative binary operation* that defines how two input items are reduced into one. If the operation is an addition, multiplication, or maximum value, the parallel reduction of all elements results in a sum, product, or maximum of the entire input, respectively. Such computations are frequently used in many applications, hence, the performance should be tunned up to maximum.

Without loss of generality, we will use *addition* as the binary operation throughout this assignment. The sequential implementation is straightforward: we need to visit all elements and reduce them into a single value - the sum of the input array:

```
result = input[0];
for (unsigned int i = 1; i < n; i++)
   result += input[i];
```

The code performs $n - 1$ operations computing the reduction sequentially with number of steps that scales linearly with the size of the input. We provide you with a skeleton of the code that already contains the functionality of validating results and testing the performance of individual GPU implementations.

4.2 Skills You Learn

In order to complete this task you will need to implement four versions of the parallel reduction.

- We start with a non-coalesced implementation using an *interleaved* addressing to access elements that will be initially placed in the global memory.
- Coalescing will be achieved via a different - *sequential* - addressing scheme.
- In order to benefit from the local memory, we will use *kernel decomposition* and perform most of the reduction locally.
- We will also focus on further optimization, e.g. *loop unrolling*.

4.3 Parallel Implementation

In order to split the computation over several processing units, we will use a tree-based reduction shown in Figure 6. Note, that if the number of processing units equals the size of the input (which is the case of Figure 6), we will be able to perform the reduction in logarithmic time. In order to simplify the implementation we will only consider arrays with sizes that equal to powers of 2. Handling arrays with arbitrary size can be achieved via appropriate padding.

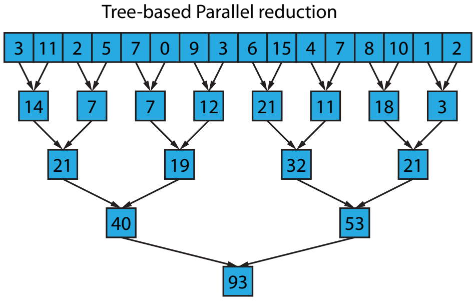


Figure 6: *Tree-based approach for parallel reduction.*

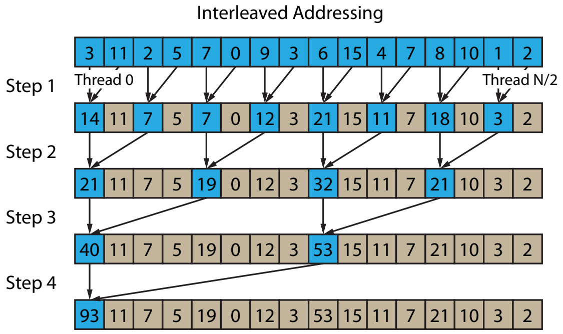


Figure 7: *Parallel reduction with interleaved addressing.*

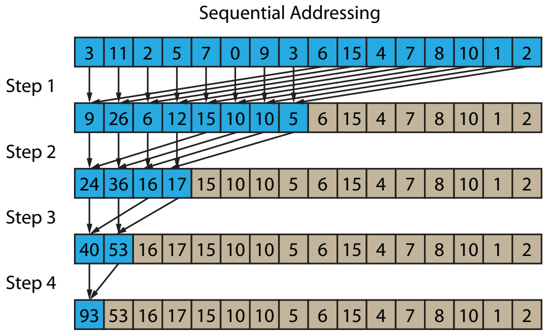


Figure 9: *Parallel reduction with sequential addressing.*

4.3.1 Interleaved Addressing

In order to perform the tree-based reduction on a GPU, we can use an interleaved addressing and always reduce two neighboring elements writing the result back into one of them, as shown in Figure 7. As long as we do not need to keep the original GPU array, we can perform the reduction *in-place* directly on the input array placed in the global memory.

Notice that the result from one step is always used as the input for the next step. This means that we need to synchronize work-items over the whole NDRange, otherwise work-items from one group could perform next step before the current step is finished by other work groups (see Figure 8 for an example). There is no OpenCL function for synchronizing the entire NDRange. However, we can exploit the fact that kernels are executed in a non-overlapping consecutive order: next kernel can start only after the current one is finished. Therefore, we can split the reduction into a number of kernel calls. Each step from Figure 7 will be accomplished by executing a kernel, which will perform all operations required for the step.

You should add the code of the reduction with interleaved addressing to the `Reduction_InterleavedAddressing` kernel in `Reduction.cl`. Furthermore, you will have to implement the `CReduction::Reduction_InterleavedAddressing` function to call the kernel with appropriate parameters as many times as necessary. On the high-level, this function should contain a loop that always computes parameters for launching the kernel, sets up correct arguments, and enqueues the kernel in the command queue. There are several approaches for mapping threads (work-items) to input elements, however, we recommend using only as many threads as necessary in each step. Then you only need to somehow compute the right offset and stride to accesses the two elements that the thread reduces.

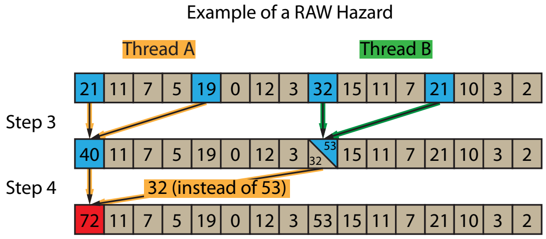


Figure 8: *We need to enforce synchronization after each step, otherwise RAW hazards may occur: thread A starts step 4 by reading an incorrect value (32) because thread B did not finish step 3 yet.*

Since debugging options of GPUs are highly limited, we advise you to proceed in smaller steps downloading the results back to the CPU (`clEnqueueReadBuffer`) and verifying them after each intermediate implementation step.

4.3.2 Sequential Addressing

The biggest drawback of the interleaved addressing is that the accesses to the global memory are not fully coalesced. The coalescing can be achieved quite easily just by using different - sequential - addressing. In each step, the threads should read two consecutive chunks of memory (first and second half of the elements), reduce them, and write the results back again in a coalesced manner. Figure 9 demonstrates sequential addressing.

Use kernel `Reduction_SequentialAddressing` and the corresponding function in `CReduction` to implement parallel reduction with coalesced accesses. Since this task is very similar to the previous one, you can reuse most of the implementation written for the interleaved addressing.

4.3.3 Kernel Decomposition

Once the accesses to the global memory are well-aligned, we can continue optimizing the reduction further. There are two major problems with the current implementation. First, each step of the reduction requires a separate kernel launch to ensure that the results are written before they are read again. Since there is a constant overhead of executing a single kernel, reductions of large arrays will suffer from high execution overhead as many steps have to be taken. The second issue resides in frequent accesses to the slow global memory: for each reduction operation we perform two global reads and one global write.

Both described problems can be significantly suppressed by caching data in the fast local memory and performing many small reductions locally. This technique is also called kernel decomposition: instead of performing wide global steps, we bundle few consecutive steps together, split them horizontally, and compute several local reductions. The global result is then computed on top of these local reductions as shown in Figure 10. For large arrays we may need several levels to perform the reduction, however, the code of the kernel will always be the same, so we only need to correctly set the arguments for the kernel.

In order to succeed in implementing the decomposition, start with a low number of elements, e.g. 512, and test your local reduction first. You should load the data into local memory, perform the reduction, and write the result back to the global device memory. You can improve the performance by computing the first reduction step before storing the data in the local memory (this is not shown in

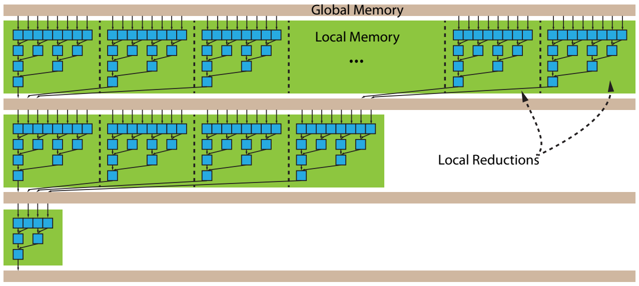


Figure 10: *Kernel decomposition bundles few reduction steps and splits them horizontally into several local reductions. As the size of an array that can be reduced within the local memory is limited, we need to perform several consecutive reduction steps that exchange data through the global memory.*

Figure 10): use 256 threads (work-items) for 512 elements , each reading one element from the first half and second element from the second half (using sequential addressing) of the element array. Then we add them together and store the result in the local memory. Local reduction of the 256 elements in the local memory is achieved by a single for loop. Each iteration should use only the appropriate number of threads (use an `if` statement and thread ID to enable the computation only for some threads). Do not forget to synchronize threads using a `barrier` after each operation that can result in read-after-write (RAW) hazards. Once you have the local reduction, use only one thread to write the result back to the global array to position that corresponds to the ID of the current work-group, so that the beginning of the array is consecutively filled with results of individual local reductions.

Do no forget to allocate enough of local memory for each kernel. If you follow the guidelines in the previous paragraph, you will need storage for one element per each thread. Since we want to run the reduction on large arrays, we still have to launch the kernel multiple times; however, much fewer times than without using the local memory.

4.3.4 Further Optimizations

The last optimization that we address within this task is *unrolling of loops*. If you correctly implemented the kernel decomposition, you have a loop with a barrier somewhere in the kernel. The barrier introduces an overhead since the threads need to synchronize, hence, we would like to avoid it if possible. We can take advantage of the fact the NVIDIA and ATI GPUs are SIMD architectures (Single Instruction Multiple Data) that execute the same instruction for several threads at the same time. In other words, the threads are grouped into so called *warps* (32 threads, NVIDIA) or *wavefronts* (16, 32, or 64 threads, depending on the actual ATI GPU) that are executed simultaneously. Therefore, we do not need to synchronize threads within a single warp/wavefront because they are executed synchronously by definition.

Supposing that you used sequential addressing, you can omit the barrier for the last few steps when only threads of one warp/wavefront are reducing the remaining elements. The most efficient solution in general is to pull these few iterations out of the loop and hard-code them using constant strides. This leads to redundant and not very clean code, therefore, unrolling of loops should be always used as the very last optimization.

4.4 Evaluation

If you run the program with all kernels correctly implemented, you should get an output similar to the following

```
Validating results of task 1
  TEST PASSED!
Validating results of task 2
  TEST PASSED!
Validating results of task 3
  TEST PASSED!
Validating results of task 4
  TEST PASSED!

Testing performance of task 1
  average time: 8.30811 ms, throughput: 2.01938 Gelem/s
Testing performance of task 2
  average time: 2.33284 ms, throughput: 7.19177 Gelem/s
Testing performance of task 3
  average time: 1.90879 ms, throughput: 8.78946 Gelem/s
Testing performance of task 4
  average time: 1.16892 ms, throughput: 14.3527 Gelem/s
```

Reported time and speedup was measured on a GTX 470. Task 3 adds two elements before storing them in the local memory and task 4 unrolls the loop and avoids barriers for the last work-group.

The total amount of points reserved for the parallel reduction is 10 and they will be given for correctly implementing:

- Interleaved addressing (2 points)
- Sequential addressing (3 points)
- Kernel Decomposition (3 points)
- Unrolling of loops + barrier avoidance (2 points)

If you succeed in optimizing the implementation further, beyond the scope of the proposed techniques, you can gain up to 2 extra points.

5 Task 2: Parallel Prefix Sum (Scan)

In order to complete this task you will need to implement two versions of the parallel prefix sum (PPS, sometimes also called *scan*):

- Naïve parallel prefix sum
- Work-efficient parallel prefix sum

5.1 Algorithm Description

Similarly to parallel reduction, parallel prefix sum also belongs to popular data-parallel algorithms. PPS is often used in problems such as stream compaction, sorting, Eulerian tours of a graph, computation of cumulative distribution functions, etc. Given an input array $X = [x_0, x_1, ...x_{n-1}]$ and an associative binary operation $\oplus$ an *inclusive* prefix sum computes an array of prefixes $[x_0, x_0 \oplus x_1, x_0 \oplus x_1 \oplus x_2, \ldots, x_0 \oplus \ldots \oplus x_{n-1}]$. If the binary operation is addition, the prefix of i^{th} element is simply a sum of all preceding elements plus the i^{th} element if we want to have an inclusive prefix sum. If we do not include the element itself we talk about an *exclusive* prefix sum. Table 1 shows examples of an inclusive and exclusive scan.

Input	3	11	2	5	7	0	9	3
Inclusive Scan	3	14	16	21	28	28	37	40
Exclusive Scan	0	3	14	16	21	28	28	37

Table 1: *Examples of an inclusive and exclusive prefix sum.*

5.1.1 Sequential Implementation

Sequential implementation is trivial: we iterate over all input elements and cumulatively compute the prefixes. It can be implemented as:

```
prefix = 0;
for(unsigned int i = 0; i < n; i++) {
  prefix += input[i];
  result[i] = prefix;
}
```

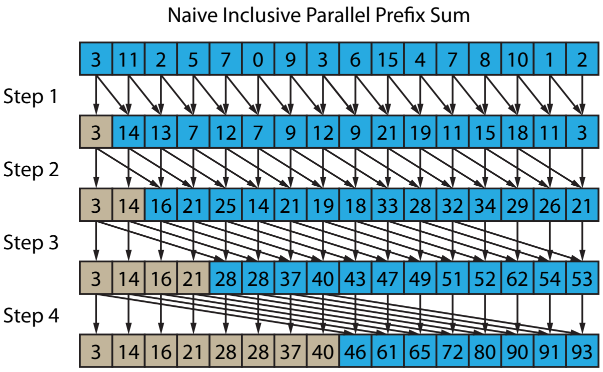
Similarly to the parallel reduction, the skeleton already contains all necessary routines for allocating and initializing necessary OpenCL variables. You only need to complete two kernels and functions that call them as described in the following section.

5.2 Parallel Implementation

In some sense, PPS is very similar to the parallel reduction. We will also perform the binary operation in a tree-like manner, though in a more sophisticated way. All implementations required for completing this task should be added to `Scan.cl` and `Scan.cpp` files. You can again consider only inputs with the width equal to a power of 2.

5.2.1 Naïve Prefix Sum

For the naïve implementation we will use the sequential addressing. Figure 11 demonstrates the basic concept of the parallel prefix sum. Notice that in each step, one item can be read by up to two threads, from which one will also write the result into this item. Even if we ensure synchronization via multiple kernel launches, the fact that threads read and write the same items in one step can potentially cause write-after-read (WAR) hazards: the thread that is responsible for computing the value of an item writes the result into the global array before another thread reads the initial value.



3 11 2 5 7 0 9 3 6 15 4 7 8 10 1 2
Step 1
3 14 13 7 12 7 9 12 9 21 19 11 15 18 11 3
Step 2
3 14 16 21 25 14 21 19 18 33 28 32 34 29 26 21
Step 3
3 14 16 21 28 28 37 40 43 47 49 51 52 62 54 53
Step 4
3 14 16 21 28 28 37 40 46 61 65 72 80 90 91 93

Figure 11: *An example of an inclusive naïve parallel prefix sum. Notice that the algorithm performs many more add operations than the sequential version.*

This can be easily avoided by using two arrays instead of one, also called double-buffering. At each step, one array will be used as the input for reading items and the other array as the output for writing the results. Notice that the input is never changed (can be declared as `const`) and the WAR hazards cannot occur. Since the output of one step is used as the input for the next one, we will need to periodically swap the arrays after each step. The technique somewhat reassembles table tennis: the data jumps periodically between the arrays as the ball jumps from one side to the other, therefore, many publications refer to it simply as the *ping-pong* technique.

The kernel for the naïve implementation is fairly simple: each thread has to either read and add two values writing the result to the appropriate position in the output array, or just propagate the already computed prefix from the input to the output.

5.2.2 Work-efficient Prefix Sum

If we carefully analyze the work-efficiency of the naïve algorithm, we find out that it performs $\Theta(n \log_2 n)$ addition operations, which is $\log_2 n$ more than the linear sequential scan. Thus, the naïve version is obviously not work-efficient, which can significantly slow down the computation, especially in the case of large arrays. Our goal in this section is to use an algorithm that removes the logarithmic term and performs only $\Theta(n)$ additions.

An algorithm that is optimal in terms of the number of addition operations was presented by Blelloch in 1989. The main idea is to perform the PPS in two hierarchical sweeps that manipulate data in a tree-like manner. In the first *up-sweep* (also called *reduce*) phase we traverse the virtual tree from leaves towards the root and compute prefix sums only for some elements (i.e. the inner nodes of the tree). In fact, we are computing parallel reduction with interleaved addressing (see Figure 12). The second phase, called *down-sweep*, is responsible for adding and propagating the intermediate results from inner nodes back to the leaves. In order to obtain correct result we need to overwrite the root (the result of the reduction phase) with zero. Then we simple descend through the tree and compute the values of the child nodes as:

- sum of the current node value and the former left child value in the case of the **right child**,
- the current node value in the case of the **left child**.

The down-sweep is shown in Figure 13.

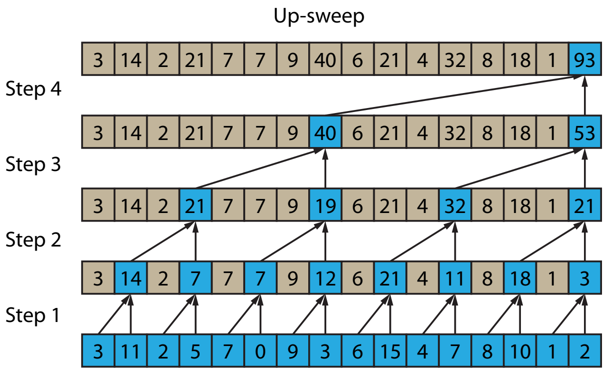


Figure 12: *Up-sweep of the work-efficient PPS is in fact a reduction with interleaved addressing.*

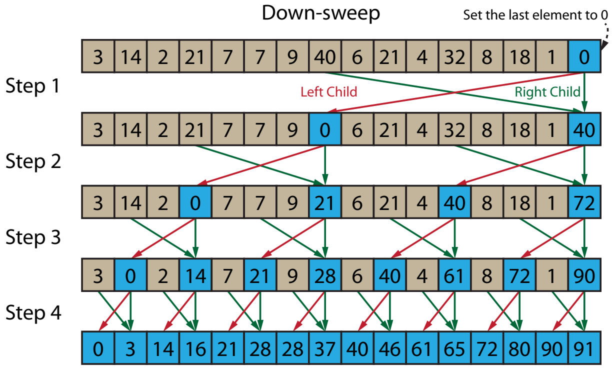


Figure 13: *Down-sweep propagates intermediate results from the inner nodes to the leaves.*

In order to achieve highest possible throughput, we will require you to decompose the computation and perform both sweeps in local memory (the concept is similar to kernel decomposition in the parallel reduction task). The performance advantages should be clear, we just need to sort out some minor optimization issues. We advise you start with a small input array that fits into the local memory (e.g. 512 elements). We will describe the extension for large arrays shortly, but for a correct implementation, it is crucial to have the local PPS working without any errors.

The kernel should begin by loading data from global to local memory. For a work-group of size N, we recommend to load and process 2N elements to avoid poor thread utilization: if we used only N elements, half of the threads would be inactive during the first and last step of the up and down sweep, which is not good for the performance. Obviously, you need to allocate local memory that can contain $2N$ elements.

Having the data in the local memory, we can perform the up-sweep within a single `for` loop. Then we explicitly write zero into the last element of the local memory array and perform the down-sweep. Do not forget to use barriers whenever necessary. If you carefully inspect Figure 13 you will notice that we compute an exclusive PPS. To compute the inclusive version, you just need to load the values from the global memory, add them to results in the local memory and write them back to the global device memory.

5.2.3 Avoiding Bank Conflicts

In the theoretical part of this assignment we talked about bank conflicts that occur if different threads from the same work-group access (different) data in the same bank. This will definitely penalize the performance so we should try to achieve a conflict-free addressing of the individual elements. Notice that as the stride between two elements is getting bigger, more and more threads will access the same bank as we proceed further. An easy solution is to reserve a bit more local memory and add some padding. You can wrap the address (offset) to the shared memory with a macro to conveniently experiment with different paddings. A simple conflict-free access can be achieved by adding 1 to the address for every multiple of the number of banks. This will ensure that elements originally falling into the same bank will be shifted to different banks. Use the profiler to make sure you do not get any bank conflicts.

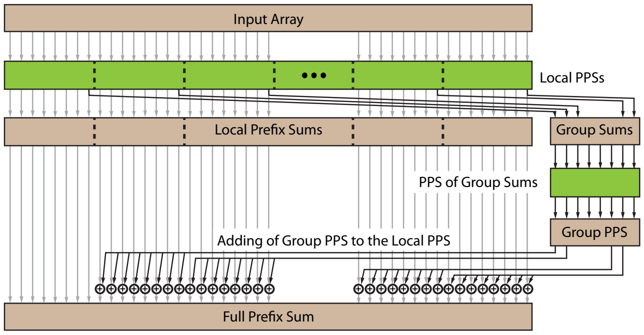


Figure 14: *PPS on large arrays performs several local PPS. The last elements of each work-group is written into a separate array over which we also perform a PPS that gives us the sum of all work-groups on the left from a particular work-group. As the last step we add these to the local prefix sums.*

5.2.4 Extension for Large Arrays

In order to extend the work-efficient PPS to support large arrays, we need to divide the input into a number of work-groups that perform multiple local PPSs. Then we will take the last prefix of each work-group and store it in an auxiliary array that is reserved only for sums of individual work-groups. Subsequently, we can run a PPS on these sums, which will add the sums of the preceding blocks (blocks to the left). Notice that we can use the same PPS kernel, we just need to use the auxiliary array with sums as the input. In order to add the prefix of block sums to the array with results of the local PPSs, you will have to implement a very simple kernel (`Scan_WorkEfficientAdd`) that only reads the right item and adds it to the entire work-group. The extension for large arrays is outlined in Figure 14.

5.3 Evaluation

The total amount of points reserved for this task is 10 and they are split as:

- Naïve PPS (2 points)
- Local work-efficient PPS (3 point)
- Conflict-free local memory access (2 point)
- Extension for large arrays (3 point)

Optional: as we encourage further optimizations and custom tuning of the code, we announce a competition for the fastest PPS running on 16 million (or more) elements. You can voluntarily sign in during the evaluation. The performance of all participating implementations will be measured right after the evaluation. The two fastest implementations will gain 2 extra points! The precondition for participation is at least one extra optimization (e.g. loop unrolling or avoidance of barriers).