Karlsruhe Institute of Technology

GPGPU: Data-Parallel Algorithms

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.

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 Vulkan compute 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 *shared 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 or Vulkan) 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.

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 (*gl_LocalInvocationID.x* in Vulkan compute shaders): threads 0-31 belong to the first warp,

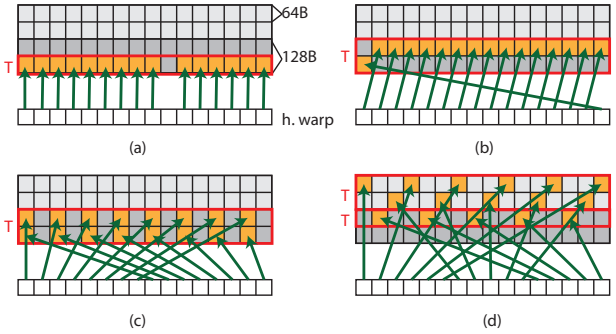

Figure 1: *Examples of memory coalescing when accessing aligned blocks of memory by a half warp on the latest CUDA architecture. (a): single 64-byte transaction. (b): unaligned access of an aligned 128-byte block still results in a single 128-byte transaction. (c): sequential access of a 128-byte-aligned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

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 (the recommendations also apply to Vulkan).

2.2 Shared 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 shared memory in Vulkan maps to the CUDA architecture.

The shared memory 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 shared 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 (Ampere) partitions the shared memory into 32 banks of 32-bit words. Much older NVIDIA GPUs were using 16 banks. A linear array placed in the shared 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).

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 shared 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 access 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.

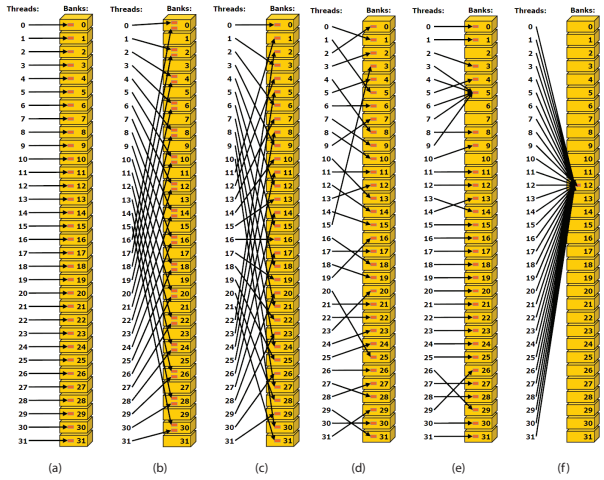

Figure 2: *Without bank conflicts, the on-chip shared memory can operate at its maximum throughput. Examples without bank conflicts: linear addressing aligned to the warps (a), linear addressing with stride of 3 32-bit words (c), random permutation of addresses, each using different banks (d). Interestingly (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.*

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

```
block[ LID.y * gl_WorkGroupSize.x + LID.x ] = M[ GID.y *
    SizeX + GID.x ];
```

When the invocations 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 `gl_WorkGroupSize` - 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 * gl_WorkGroupSize.x + 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 shared memory. The padding of data in the memory always depends on the number of banks. Unfortunately, this number can

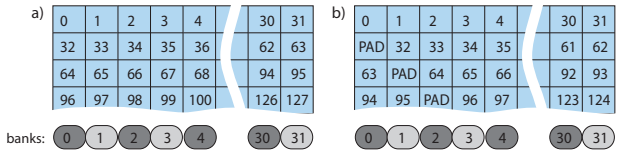

Figure 3: *By inserting a padding word after each 32 words, we can completely eliminate bank conflicts from the kernel rotating a matrix tile in the shared memory. For simplicity, we use the local indexing of a 32x32 tile to demonstrate the realigned positions of the elements.*

be different on various architectures, so you always need to check the specification or appropriate programming guide.

3 Task 1: Parallel Reduction

The previous assignment has guided you through all relevant steps from pipeline creation to dispatch. This assignment is mostly concerned with kernels and their submission to the GPU. As a result, you mostly only need to complete the compute kernels and record corresponding commands into the command buffers. All other steps are pre-filled. Shader code resides in `shaders/`, host code in `src/A2Task1Solution/` and `src/A2Task2Solution/`. From the file naming you can deduce where to add your implementation for a task. For example, the shader code for the sequential algorithm from task 1 should be implemented in `shaders/A2Task1Sequential.comp`, while the host code should be implemented in `src/A2Task1Solution/Sequential.cpp`.

If you run into problems, it might be worthwhile to revise the steps from the previous assignment. It also helps to inspect your program through RenderDoc to see if Vulkan has correctly received all the parameters that you intended.

3.1 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.

3.2 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 shared memory, we will use *kernel decomposition* and perform most of the reduction locally.
- We will also focus on further optimization, e.g. *loop unrolling*.

3.3 Parallel Implementation

In order to split the computation over several processing units, we will use a tree-based reduction shown in Figure 4. Note, that if the number of processing units equals the size of the input (which is the case of Figure 4), 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.

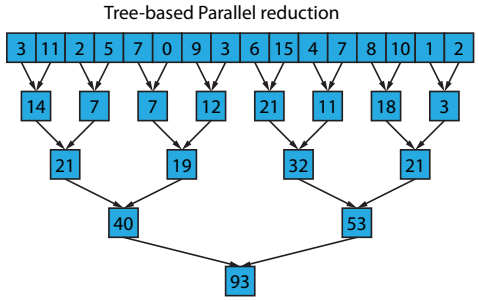

Figure 4: *Tree-based approach for parallel reduction.*

3.3.1 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 5. 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 invocations across all work groups, otherwise invocations from one group could perform the next step before the current step is finished by other work groups (see Figure 6 for an example). There is no GLSL function for synchronizing all invocations (this would entirely defeat the streaming architecture of a GPU, as the intermediate state of all invocations would need to be stored somewhere). However, if we split the computation into separate kernels, we can ensure proper ordering. Each step from Figure 5 will be accomplished by executing a kernel, which will perform all operations required for the step.

Compared to other APIs like OpenCL or CUDA, there is one large caveat in Vulkan, however: By default, all kernels that are submitted for dispatch into a queue (be it with a single or separate command buffers), may be executed completely out of order. This is absolutely not what we initially wanted, since we need each kernel to only start execution after the previous one has finished. In many cases, this allows the driver to maximize parallelism by executing multiple dispatches at once. In our case, however, we need to perform additional synchronization.

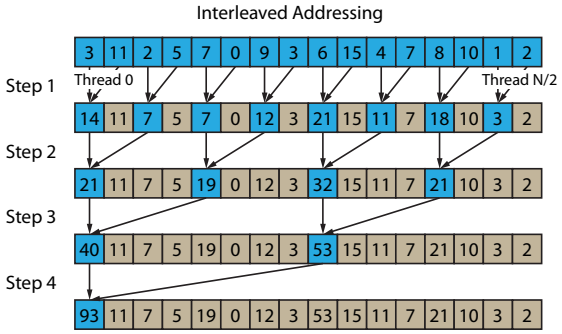

Figure 5: *Parallel reduction with interleaved addressing.*

In the last assignment, we have already learned about fences as a synchronization primitive between GPU and host. In theory we could use it here to solve the issue, by issuing a kernel and then immediatly waiting for its completion with a fence before we issue the next kernel. The problem with this approach is that for each kernel submission, we now have to perform a costly round-trip from host to GPU (submission of the kernel) and from GPU to host (waiting for the fence), which introduces a lot of latency. Ideally, we would like to pay this cost only once, by submitting all work upfront and then waiting for everything to finish. For this exact purpose, Vulkan exposes pipeline barriers as a more lightweight synchronization primitive. Contary to fences, it only allows to synchronize work on the GPU and only work that has been submitted to the same queue. But this allows for enforcing the synchronization directly on the GPU, without additional involvement of the CPU. Do not confuse pipeline barriers with barriers in compute shaders! While the latter provides synchronization between invocations within a work group of a dispatch, the former provides synchronization between entire dispatches. Pipeline barriers are recorded to command buffers just like compute shader dispatches:

```
// ...

cb.dispatch(dx, dy, dz);

// The next dispatch depends on the previous dispatch,
// therefore insert pipeline barrier into command buffer.
cb.pipelineBarrier(
  vk::PipelineStageFlagBits::eComputeShader,
  vk::PipelineStageFlagBits::eComputeShader,
  vk::DependencyFlags(),
  {vk::MemoryBarrier(vk::AccessFlagBits::eShaderWrite, vk
       ::AccessFlagBits::eShaderRead)},
  {},
  {}
);

cb.dispatch(dx, dy, dz);

// ...
```

Pipeline barriers are more fine-grained than fences, but overall they are still rather coarse-grained. They only allow to synchronize work that has been submitted before it with work that will be submitted after it. In particular, you can't select individual dispatches that should wait between one another. You can select pipeline stages that should participate in the synchronization (the first two parameters), but this is mostly useful for graphics work, since compute pipelines contain only a single stage (compute shader) as opposed to the many stages that graphics pipelines contain (e.g. vertex, geometry and fragment shaders, color output, ...).

Vulkan additionally demands from the application developer to manually maintain GPU caches via flushing (write dirty cache entries to memory) and invalidation (clear stale cache entries). The GPU cannot do this on its own and in other APIs this is the responsibility of the driver. Cache management can be specified in pipeline barriers through the last three parameters. Since this is a rather advanced topic, it is enough for the purpose of this course that you copy the pipeline barrier code above in between dispatches that need to be synchronized.

There exist additional synchronization primitives like semaphores and events, and we have only scratched the surface of pipeline barriers themselves. You can get additional info online from the Vulkan specification [1] or blog posts [2]

You should add the code of the reduction with interleaved addressing in `A2Task1Interleaved`. Furthermore, you will have to prepare the pipeline and call the compute shader as many times as necessary. On the high-level, you need a loop when recording the command buffer that always computes parameters for the kernel, sets up correct arguments, and dispatches the kernel in the command buffer, with proper pipeline barriers in between. There are several approaches for mapping threads (invocations) 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.

Since debugging options of GPUs are highly limited, we advise you to use RenderDoc to check whether the compute shaders were dispatched with correct parameters (descriptor bindings, push constants, dispatch sizes) and to inspect buffer contents between dispatches. You can additionally download intermediate results back to the CPU (`fillHostWithStagingBuffer`) and verify them after each intermediate implementation step.

It's also important to make sure that you don't start threads that are damned to check some condition and find out that they must not do any actual work.

3.3.2 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 7 demonstrates sequential addressing.

[1] `https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/chap7.html`

[2] `https://themaister.net/blog/2019/08/14/yet-another-blog-explaining-vulkan-synchronization/`

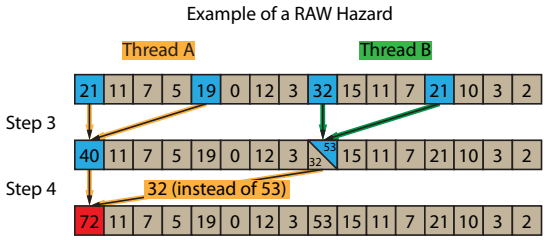

Figure 6: *We need to enforce synchronization after each step, otherwise RAW hazards may occur: thread A starts step 4 by reading an incorrect value (32) because thread B did not finish step 3 yet.*

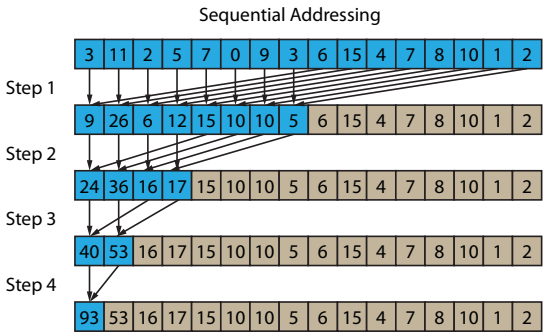

Figure 7: *Parallel reduction with sequential addressing.*

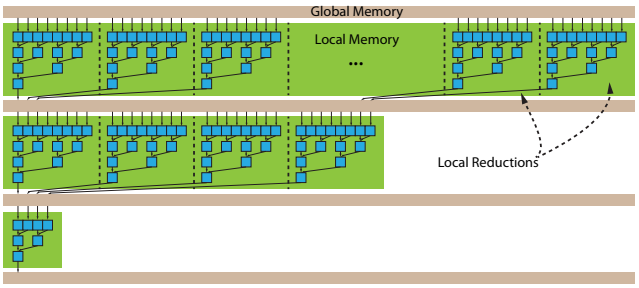

Figure 8: *Kernel decomposition bundles few reduction steps and splits them horizontally into several local reductions. As the size of an array that can be reduced within the shared memory is limited, we need to perform several consecutive reduction steps that exchange data through the global memory.*

3.3.3 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 shared 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 8. 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 shared 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 shared memory (this is not shown in Figure 8): use 256 threads (invocations) 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 shared memory. Local reduction of the 256 elements in the shared 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 workgroup, so that the beginning of the array is consecutively filled with results of individual local reductions.

Do no forget to allocate enough of shared 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 shared memory.

If you carefully inspect Figure 8, you may notice that write-after-read (WAR) hazards can occur. For example, consider the first iteration and the second local work group, which writes its result at the second position of the input array. This position stores the input for the first work group. If the second group happens to write the result before the first group loads, it will overwrite the input values for the first group. 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. You can simply create and additional binding point in your shader for the second buffer and extend your descriptor set accordingly. For the ping-pong itself, you need two descriptor sets that you alternate between after each loop iteration. In the first descriptor set, buffer 0 is bound to binding point 0, and buffer 1 to binding point 1. In the second descriptor set, buffer 0 is bound to binding point 1, and buffer 1 to binding point 0.

3.3.4 Further Optimizations

Another optimization that we address within this task is unrolling of loops. If you correctly implemented the kernel decomposition, you can unroll the last few steps to avoid instructions to control the loop `https://en.wikipedia.org/wiki/Loop_unrolling`. 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.

We can take advantage of the fact that NVIDIA and AMD 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 AMD GPU) that are executed simultaneously.

Some platforms like NVIDIA CUDA guarantee that a small number of threads within a warp (i.e. 32 threads) will follow the same instruction path and are synchronized by the hardware implementation. In such cases we can make use of the hardware functionality and omit the thread synchronization within a warp as well. How-

ever Vulkan does not guarantee that all threads within a wavefront are synchronized, even though they are executed in parallel at the same time, so with loop unrolling the use of barriers is still needed.

3.3.5 Local Reduction with Atomics

Atomic Functions

Atomic functions implement uninterruptible read-modify-write memory operations. They can be used to enable co-ordination among multiple threads and to serialize contentious updates from multiple threads. Only a limited set of types and data sizes, as well as operations are supported. Please refer to [3] for a complete list of all atomic operations available in Vulkan GLSL.

The following pseudo-code shows how `atomicAdd` works *semantically*:

```
int atomicAdd(inout int mem, int v)
{
  int old;
  exclusive_single_thread // pseudo code
  {
    // atomically perform load, add, store operations
    old = mem; // Load from memory
    mem = old + v; // Store after adding v
  }
  return old;
}
```

Exclusive access to the memory is granted to one thread and the memory content is updated by this thread. However when several threads issue an atomic operation for the same memory location, the order of the atomic operations is not deterministic. Each of the atomic operations returns the value which was previously stored at the memory location.

On older GPUs, atomic operations where considered to be very slow and therefore its use was discouraged. In contrast, on current GPUs atomics are highly optimized: NVIDIA claims that on current GPUs (Maxwell/Pascal), the costs for atomics that are accessing the global memory roughly correspond to the costs of a global memory read.

In this task you are required to do the local reduction inside work-groups using a single atomic variable and compare this with the other implementations. You get extra points if you also try different work-group sizes.

3.4 Evaluation

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

```
[Task1] evaluating Interleaved
// ...
Execution time: 12.3818 ms, Throughput: 10.8399 GE/s
TEST PASSED
[Task1] evaluating Sequential
// ...
Execution time: 3.38738 ms, Throughput: 39.6229 GE/s
TEST PASSED
[Task1] evaluating KernelDecomposition
// ...
Execution time: 1.52575 ms, Throughput: 87.9684 GE/s
TEST PASSED
[Task1] evaluating KernelDecomposition Unroll
// ...
Execution time: 1.44425 ms, Throughput: 92.9325 GE/s
TEST PASSED
[Task1] evaluating KernelDecomposition Atomic
// ...
Execution time: 1.34725 ms, Throughput: 99.6235 GE/s
TEST PASSED
```

[3] https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object

Reported time and speedup was measured on an Nvidia RTX 2080 Ti for 128 million elements. Task 3 adds two elements before storing them in the shared memory. If we further optimize the implementation by unrolling the loops (task 4), we gain better results. However, due to the optimizations made in recent hardware, using atomics provides at times easier and faster code.

The total amount of points reserved for the parallel reduction is 11 and they will be given for correctly implementing:

- Interleaved addressing (2 points)
- Sequential addressing (2 points)
- Kernel Decomposition (3 points)
- Unrolling of loops (2 points)
- Kernel Decomposition using Atomics and methods comparison (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.

4 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

4.1 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.*

4.1.1 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.

4.2 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.

4.2.1 Naïve Prefix Sum

For the naïve implementation we will use the sequential addressing. Figure 9 demonstrates the basic concept of the parallel prefix sum. 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. 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. In order to avoid WAR hazards we again need to use the *ping-pong* technique, i.e. use two arrays, one for storing the input and another for storing the results, swapping them after each execution of the kernel.

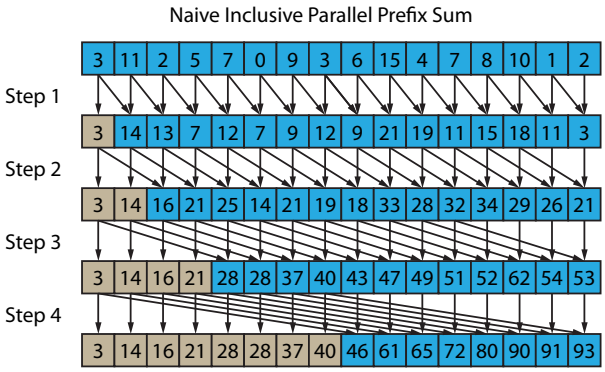

Figure 9: *An example of an inclusive naïve parallel prefix sum. Notice that the algorithm performs many more add operations than the sequential version.*

4.2.2 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 10). 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 simply 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 11.

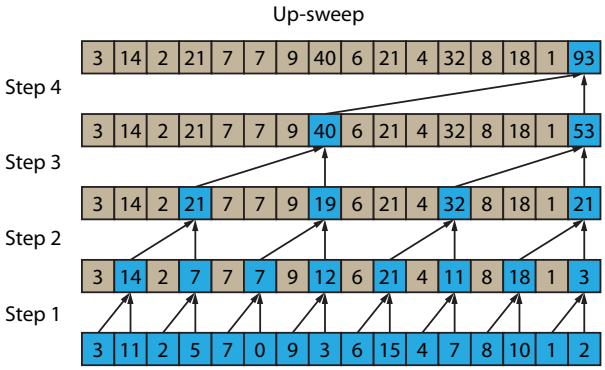

Figure 10: *Up-sweep of the work-efficient PPS is in fact a reduction with interleaved addressing.*

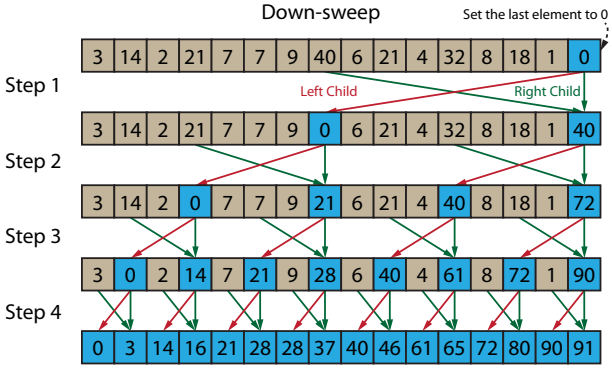

Figure 11: *Down-sweep propagates intermediate results from the inner nodes to the leaves.*

In order to achieve highest possible throughput, we will require you to decompose the computation and perform both sweeps in shared 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 shared 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 shared 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 shared memory that can contain $2N$ elements.

Having the data in the shared memory, we can perform the up-sweep within a single `for` loop. Then we explicitly write zero into the last element of the shared memory array and perform the down-sweep. Do not forget to use barriers whenever necessary. If you carefully inspect Figure 11 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 shared memory and write them back to the global device memory.

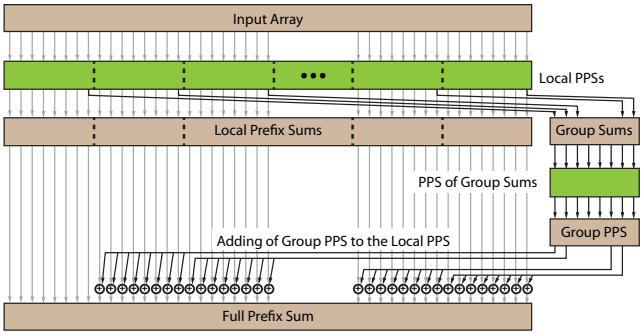

Figure 12: *PPS on large arrays performs several local PPS. The last elements of each work-group is written into a separate array over which we also perform a PPS that gives us the sum of all work-groups on the left from a particular work-group. As the last step we add these to the local prefix sums.*

4.2.3 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 shared 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. You can modify the existing kernel decomposition kernel.

4.2.4 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 (`A2Task2KernelDecompositionOffset.comp`) that only reads the right item and adds it to the entire work-group. The extension for large arrays is outlined in Figure 12.

4.3 Evaluation

The total amount of points reserved for this task is 9:

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

Evaluation notes: You get a *malus* if your code is not nicely written using meaningful variable names in proper English, or significantly lacks proper comments. Moreover, you won´t get points if your solution produces errors or way too many warnings and/or you can't explain theoretical concepts mentioned in the assignment.

If we got the impression that you handed in copy-pasted code, and you are unable to explain your solution, we will expel you from the course.