1 Parallel Prefix Scan

Parallel Prefix is an important primitive for parallel data operations.

We have an operator $\oplus$ which is associative and an ordered set of elements $[a_1, a_2, a_3, \ldots, a_n]$ and we are to compute the ordered set $[a_1, (a_1 \oplus a_2), \ldots, (a_1 \oplus a_2 \oplus \ldots \oplus a_n)]$

For example: When the operator $\oplus$ is $+$ (addition) and the input is $[1, 2, 0, 7, 8, 9]$ we get $[1, 3, 3, 10, 19, 27]$.

Parallel prefix is also commonly known as ”scan”. There are two types of scan called inclusive and exclusive scan. Inclusive means to compute the sums for all elements like we did above and exlusive scan means to shift an inclusive scan right by one element and inserting the identity element at the 0^{th} place.

In this document we will develop the scan operation for the GPU

For this tutorial you will need the following packages: Uncomment the lines and execute the cell to get the packages. It will take some time to get download and install these packages.

```
# using Pkg
# packages = ["BenchmarkTools", "CUDA", "Plots"]
# Pkg.add.(packages)
```

2 SECTIONS

1. Naive CPU implementation
2. CPU Parallelization
3. Hillis-Steele
4. Belloch
5. Applying SHFL intrinsics
6. Comparison to C++ CUDA and Thrust

2.1 1. Naive CPU implementation

The simplest way would be:

```
function scan_naive!(f, output, input) where T <: Number
    output[1] = input[1]
    for i = 2:length(input)
        @inbounds output[i] = f(output[i - 1], input[i])
    end
    output
end
```

```
scan_naive! (generic function with 1 method)
```

```
n = 5
A = collect(1:n) # [1, 2, 3, 4, 5]
B = similar(A)
println(A, "  ->  ", scan_naive!(+, B, A))
```

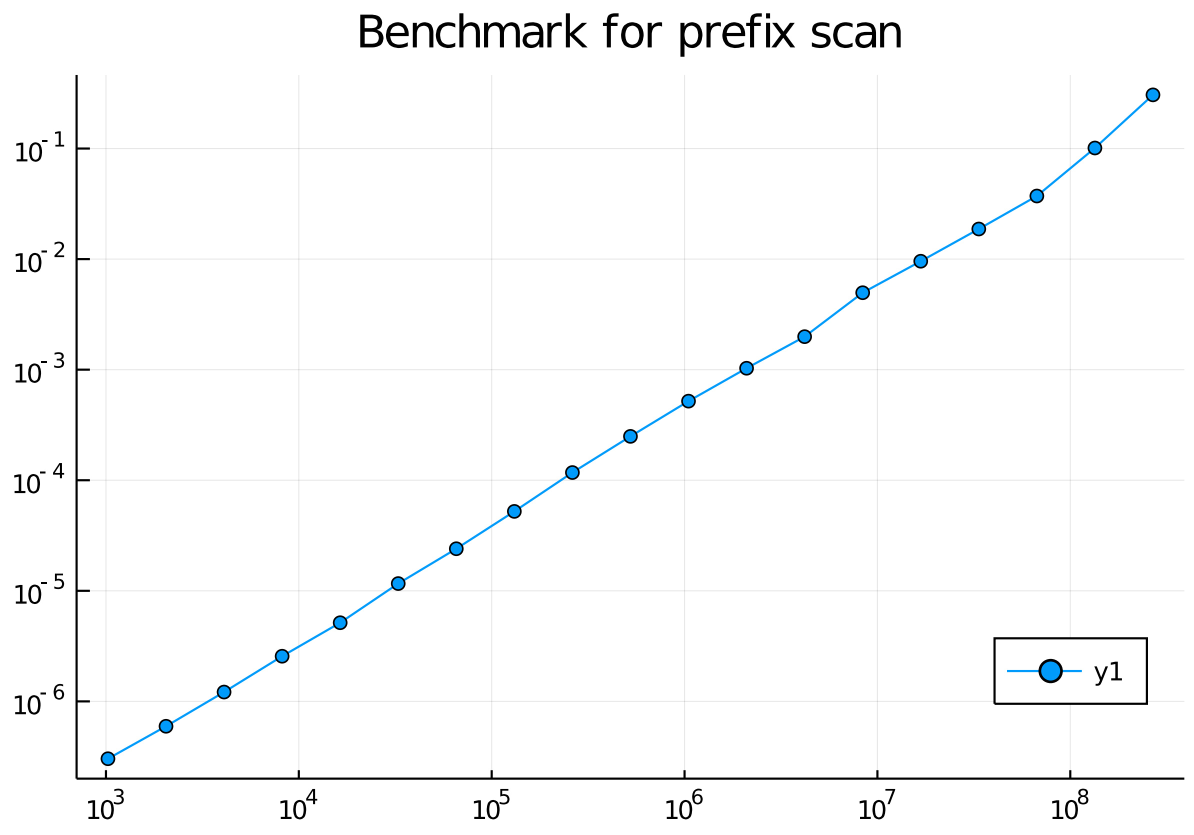
```
[1, 2, 3, 4, 5]  ->  [1, 3, 6, 10, 15]
```

```
A = fill(1., n)
B = similar(A)
println(A, "  ->  ", scan_naive!(+, B, A))
```

```
[1.0, 1.0, 1.0, 1.0, 1.0]  ->  [1.0, 2.0, 3.0, 4.0, 5.0]
```

The primary purpose of using GPU's is for speed hence it is essential to almost always benchmark code. We will take a baseline measurement from our CPU code by creating a simple benchmark and using the `BenchmarkTools` library and visualize it's performance using a plot.

```
using Plots, BenchmarkTools

# Array lengths; From 2^10 = 1024 to 2^28
# 2^28 elements of Int32 will consume around 2GB of RAM. Reduce or increase as needed
sizes = 1 .<< collect(10:28)

function timer(f, op, len)
    bench = @benchmark $f($op, B, A) setup=(A = ones(Int32, $len); B = similar(A))
    median(bench).time / 1e9
end

t = [timer(scan_naive!, +, i) for i in sizes]
# Notice that the scale is log10
plot(sizes, t, title="Benchmark for prefix scan", scale=:log10, marker = :circle,
legend=:bottomright)
```



2.2 2. CPU parallelization

The main question is how can we improve upon our previous result. The issue with applying parallel computation here is that the i^{th} element **depends** on the $(i-1)^{th}$ element. This is called a loop carried dependency.

$$i^{th}$$

element **depends** on the $(i-1)^{th}$ element

However, remember that above we declared $\oplus$ to be associative. This effectively means that $a_1 \oplus a_2 \oplus a_3 = (a_1 \oplus a_2) \oplus a_3 = a_1 \oplus (a_2 \oplus a_3)$

This is the hint as to how we will solve this problem. You can try to come up with an algorithm and derive the time complexity now.

In parallel programming it is often useful think of the extreme cases, i.e. what if we had two processors and what if we had infinite processors.

Let t be the amount of time taken by a single processor to compute the prefix scan on a particular array. With two processors we can split the array in half and have both processors work on their separate halves which will take time $\frac{t}{2}$. Then both processors add the total sum of the first half to the second half in time $\frac{t}{4}$ with the total time being dropped to $\frac{t}{2} + \frac{t}{4} = \frac{3t}{4}$ which is a 1.33x speedup with double the processors. This is a good approach when thtotale number of processors are small like in our desktop CPU. With 6 processors we can expect a speedup around $\frac{t}{6} + \frac{1}{6} \cdot \frac{5t}{6} = \frac{11t}{36}$ which is 3.27x speedup.

The thing to remember is that in a practice we might not get these speedups. Creating threads has a cost, memory accesses may become a bottleneck and other practical problems may arise. The only way to verify is to cautiously benchmark code.

GPU vendors are known to claim unrealistic speedups. With a title like ”X algorithm is 100x faster on the GPU compared to the CPU.” The trick they use is it compare sequential unoptimized code to a fairly optimized GPU code.

2.2.1 Multithreading

To do multithreading in Julia you need to have a multi-core CPU and start Julia with the environment variable ”JULIA*NUM*THREADS” set to the number of threads you want. To start Julia with this you can either do `JULIA_NUM_THREADS=6 julia` in the terminal while starting Julia or set the environment variable `JULIA_NUM_THREADS` to the number of threads in your machine.

You can verify the number of threads using `Threads.nthreads()`

2.3 Multithreaded is not outperforming at this point. Not sure what's wrong, this will be investigated and fixed for the final tutorial.

```julia
import Base.Threads: nthreads, @threads, threadid, @spawn
nthreads()
```

```
6
```

We will use the `Threads.@threads` macro to parallelize our for-loop across various threads. Our strategy will be that for each thread we will store the result to a location in the first stage. Then we will do a prefix scan for these intermediate sums and finally broadcast back the sums to the array.

```julia
function scan_threaded!(op, output::Vector{T}, input::Vector{T}) where T <: Number
    # Each thread gets a segment
    segment_length = length(input) ÷ nthreads()

    # Store the intermedite sums in an array to broadcast back
    sums = Array{T}(undef, nthreads())

    # launch nthreads() and compute prefix sum for each segment
    @threads for i = 1:nthreads()
        low = 1 + (threadid() - 1)*segment_length
        high = threadid()*segment_length

        # last thread must compute remaining elements
        threadid() == nthreads() && (high = length(input))

        @inbounds begin
            output[low] = input[low]
            for j in (low + 1):high
```

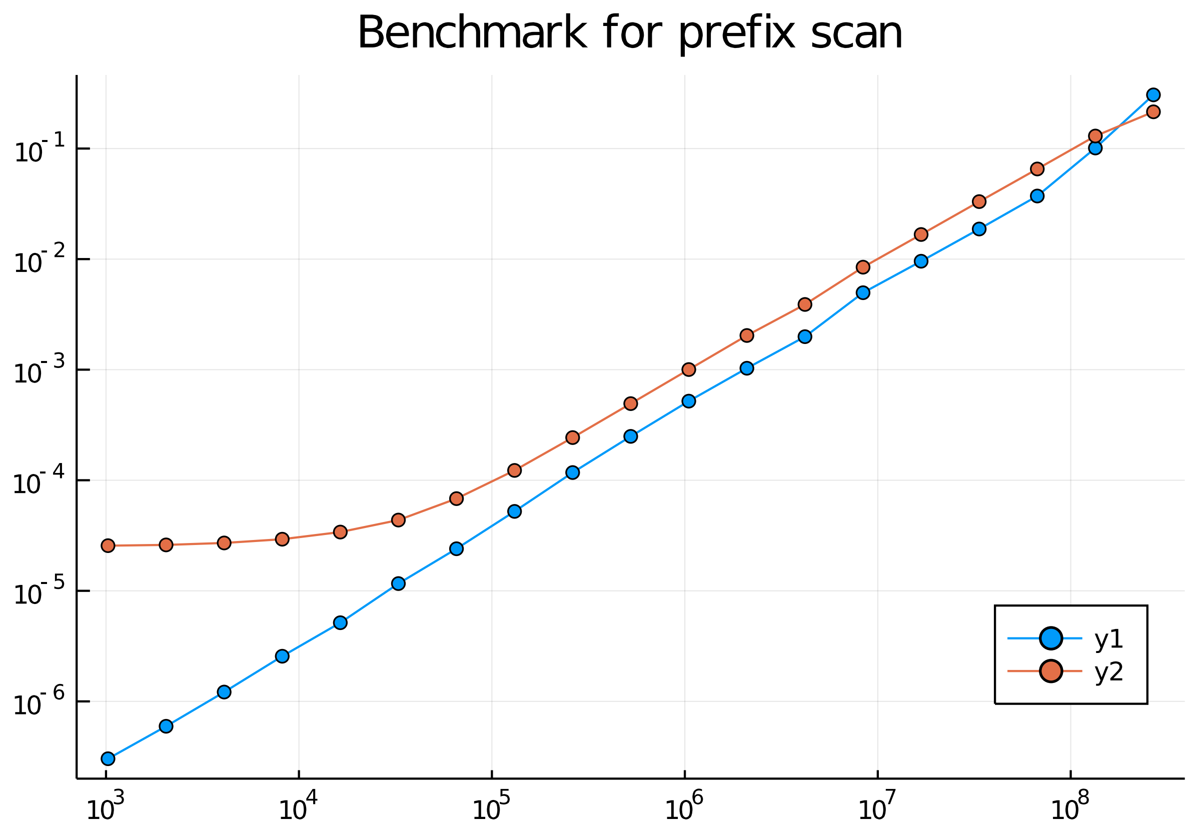
```
                output[j] = op(input[j], output[j - 1])
            end
            sums[threadid()] = output[high]
        end
    end

    # prefix sum on intermediate sums
    scan_naive!(op, sums, sums)

    # Broadcast intermediate sums
    @threads for i in (segment_length + 1):length(input)
        segment = (i - 1) ÷ segment_length
        i >= nthreads()*segment_length && (segment = nthreads() - 1)
        @inbounds output[i] = op(sums[segment], output[i])
    end
    output
end
```

```
scan_threaded! (generic function with 1 method)
```

```
using Test
A = ones(Float32, 1_000_000)
B = similar(A)
scan_naive!(+, B, A)

C = similar(A)
scan_threaded!(+, C, A)

# Check if B is approximately C. We check approximately because IEEE Floats are
# not associative so it is possible that each elements are not exactly equal to
# each other.

@test B ≈ C
```

```
Test Passed
```

```
t2 = [timer(scan_threaded!, +, i) for i in sizes]
plot!(sizes, t2, title="Benchmark for prefix scan", scale=:log10, marker = :circle,
legend = :bottomright)
```



With SIMD we have a vector of size W bits with each word taking w bits. Different CPU vendors have different implementations for their vector units. Intel has SSE and AVX, ARM has Neon, IBM Power has AIX,.etc. Vector units implement a number of features like different loads and stores, cross-lane operations for a certain W which will differ across versions. (?Section needs improvement, better description, is this even needed?)

Since Julia's compiler couldn't internally vectorize our function, we will have to implement this ourself. There are two options here, either write using vedor's SIMD intrinsics or use a higher-level abstraction. Intrinsics means code in higher level language which is very close to the assembly instruction it related to.

If we choose to write the assembly ourself we will be stuck to a single vendor and will eliminate portability and increase code complexity. However, it's not as bad as it sounds. A C++ version of this will be shared ahead. The other option is to use a higher-level abstraction. In Julia we have the SIMD.jl package which allows us to abstract in a generic vendor neutral fashion however there are a number of C++ templated vector SIMD packages which implement such functionality.

It's not within the scope of this document to discuss the SIMD and the multi-threaded implementation however we have code available for personal use.

TODO's: Just provide sourcecode and benchmarks. No need to discuss here

1. SIMD Implementation (2x for Int32 on AVX-2):
2. C++ Intel AVX-2 Intrinsics:
3. Multi-threaded implementation(3.27x on 6 cores): Not getting these results. Investigation pending.
4. SIMD + Multi-threading(6.54x ?):

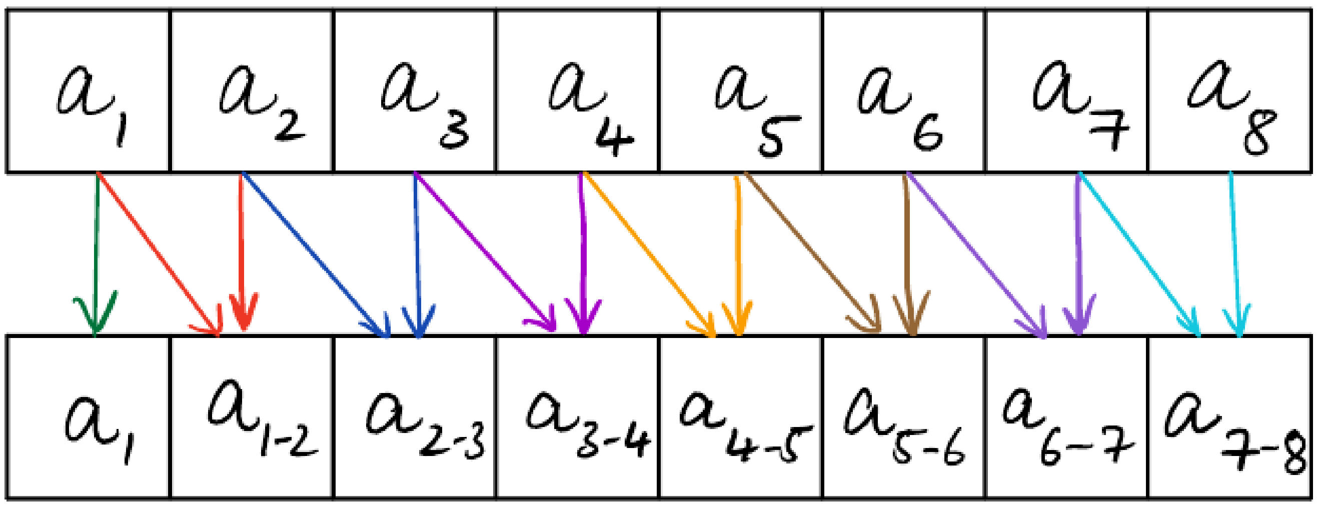


Figure 1: hillis-1

2.3.1 3. Hillis-Steele algorithm

In this section and the next we will discuss two important algorithms called Hillis-Steele and Belloch-scan. The original inspiration for both of these algorithms comes from the quest to improve circuits for parallel adders. The challenge in improving parallel adders was to have a balance between speed and circuit area. Our problem is related as we too are trying to balance between speed and processors. Hillis-Steele is related to the Kogge-Stone adder and Belloch is related to the Brent-Kung adder.

Hillis Steele's original paper here.

In the previous section we considered the parallelization with multiple CPU threads. But, in this section we move to the GPU where we have a very large number of light-weight threads which can easily be in the hundred thousand range.

To understand the algorithm let's first fix the array length to 8 since that will be easy to visualize. Let's say the number of processors $p = 8$ which is the length of the array.

Now what is the first step our processors should do ? One logical move could be that each processor p_i does $b_i = a_i \oplus a_{i-1}$ where p_1 copies the first element. This results in an array with some partial prefix sums. For out convenience let a_{i-j} denote $a_i \oplus a_{i+1} \oplus ... \oplus a_j$. Therefore our output array should have a_{1-i} at position i.

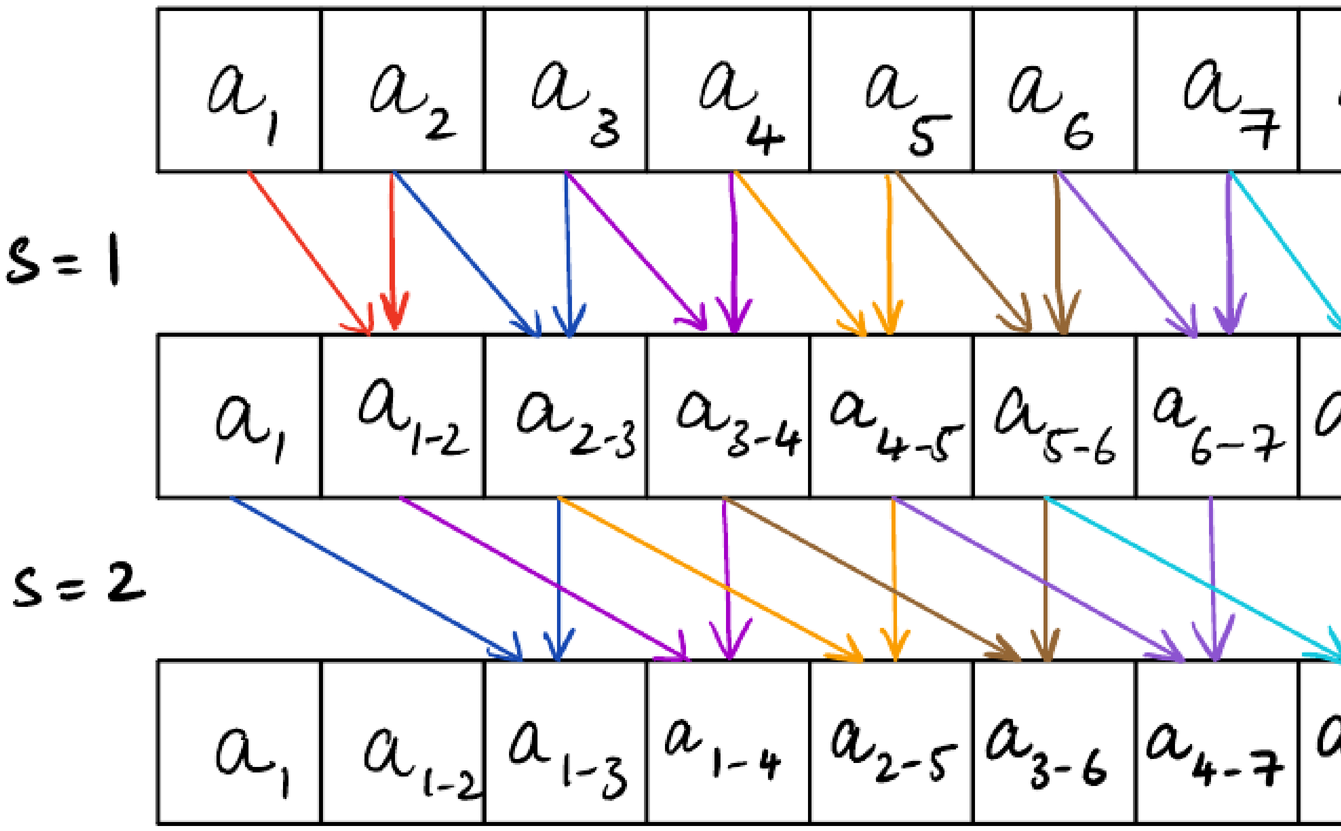
What could be the next logical step ?

Currently $b_i = a_i \oplus a_{i-1}$ where $i > 1$. The 1^{st} and 2^{nd} elements have achieved the required output.

Increasing the step size to 2 we will get.

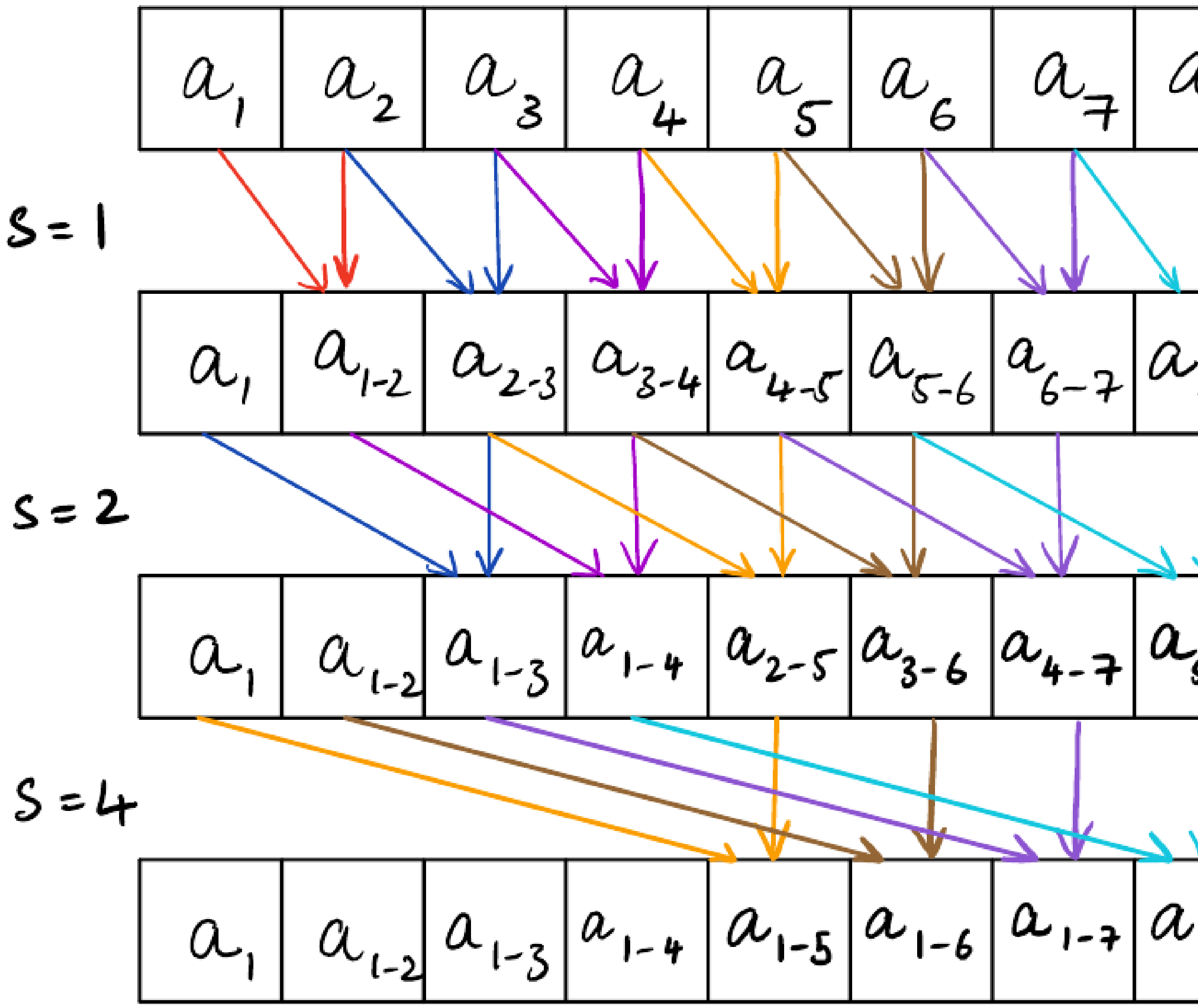
Notice that the fourth element is a_{1-4} and for the first 4 elements we have the desired output. Now what should the next step size be ?

It should be 4 as a_{2-5} needs a_1 not a_{1-2} and similarly for others. This is our last step for an array of size 8.



a_1 a_2 a_3 a_4 a_5 a_6 a_7
$s=1$
a_1 a_{1-2} a_{2-3} a_{3-4} a_{4-5} a_{5-6} a_{6-7}
$s=2$
a_1 a_{1-2} a_{1-3} a_{1-4} a_{2-5} a_{3-6} a_{4-7}

Figure 2: hillis-2



a_1 a_2 a_3 a_4 a_5 a_6 a_7
s=1
a_1 a_{1-2} a_{2-3} a_{3-4} a_{4-5} a_{5-6} a_{6-7}
s=2
a_1 a_{1-2} a_{1-3} a_{1-4} a_{2-5} a_{3-6} a_{4-7}
s=4
a_1 a_{1-2} a_{1-3} a_{1-4} a_{1-5} a_{1-6} a_{1-7}

Figure 3: hillis-3

This is the main idea behind Hillis-Steele.

The pseudocode for this algorithm is:

```
for p in parallel do
    for j = 1:log(n)
        if p_i >= 2^j # p_i = processor with index i
            output[k] = input[k - 2^j] \ensuremath{\oplus} input[k]
        else
            output[k] = input[k]
```

But why do we need two different arrays ? Is it be possible to store back to input array ?

In short **NO**, the simple reason is that all processors of a GPU don't work simultaneously. Refer to the GPU architecture guide for detailed information on this topic. To adjust for this we will share an implementation of a double-buffered Hillis-Steele scan.

```julia
using CUDA

# This may take some time if being executed for the first time.



function scan_hillis!(f, B::CuDeviceArray{T, N, P}, A) where {P, N, T<:Number}
    n = length(A)
    temp = @cuDynamicSharedMem(T, 2*n)
    index = threadIdx().x

    # Load variable from global memory into shared memory
    temp[index] = A[index]

    # Ensure that all threads have finished loading
    sync_threads()

    # Double-buffered Hillis-Steele means that we are going to use
    # two different arrays and alternate between both of them as
    # input and output arrays respectively
    #
    # We will segment a single array into two parts for simplicity,
    # in effect input[i] = temp[pin*n + i] and
    #          output[i] = temp[pout*n + i]

    pout = 0
    pin = 1

    offset = 1
    while offset <= n

        # Flip input and output arrays
        pout = 1 - pout
        pin = 1 - pin

        if index > offset
            temp[pout*n + index] = f(temp[pin*n + index],temp[pin * n + index - offset])
        else
            temp[pout*n + index] = temp[pin * n + index]
        end
        offset *= 2
        sync_threads()
    end

    B[index] = temp[pout*n + index]
    return
end
```

```
scan_hillis! (generic function with 1 method)
```

```
A = CUDA.ones(Int32, 32)
B = similar(A)
sz = 2*sizeof(Int32)*length(A)
@cuda threads=length(A) shmem=sz scan_hillis!(+, B, A)
```

```
print(B)
```

```
Int32[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]
```

```
A = rand(512)
B = similar(A)
scan_naive!(+, B, A)

A_gpu = cu(A)
B_gpu = similar(A_gpu)
sz = 2*sizeof(Float64)*length(A_gpu)

@cuda blocks=1 threads=length(A_gpu) shmem=sz scan_hillis!(+, B_gpu, A_gpu)
B_gpucpy = collect(B_gpu)

@test B ≈ B_gpucpy
```

```
Test Passed
```

Now the above function cannot compute for arbitrarily sized Arrays. This is because we are only using 1 block to compute rather than multiple blocks. The idea of extending our algorithm to arbitrarily sized arrays is similar to the CPU multithreaded version. Each block will store it's partial sum in an intermediate sums array, then we perform scan on this array and broadcast back the sums.

A reference implementation along with the benchmark is available here: (TODO)

Let's try to understand the asymptotic analysis to realize oppurtunities of improvement.

The time complexity is the amount of steps taken by our algorithm. For an array of size n we take $\lceil \log_2 n \rceil$ steps hence our time complexity is $O(\log_2 n)$.

Another important characteristic for parallel algorithms is its work-complexity which is the total amount of work done. Here it is the total number of operations ($\oplus$) done . For Hillis-Steele the total number of $\oplus$ operations $= \sum_{i=0}^{\log n} n - 2^i = O(n \log(n))$. In contrast our sequential algorithm computed the sum with work-efficiency $O(n)$

2.3.2 4. Belloch Scan

While the time complexity of $O(\log n)$ cannot be reduced, improving the work-efficiency should translate to better real world performance. Belloch scan reduces the work-efficiency of Hillis-Steele to $O(n)$.

Link to original paper here.

The idea is to do the work in two phases: upsweep and downsweep. Each phase can be visualized like a binary search tree. The height of a binary tree is $O(\log n)$ and it has $2n$ nodes where each node represents either a swap or $\oplus$.

In the *upsweep* stage each element is a leaf and and each node represents left child $\oplus$ right child. After the tree is complete, the root node is replaced by a neutral (w.r.t $\oplus$).

In the *downsweep* stage we begin at the root and traverse in the opposite direction. However in this stage we do the swap operation along with $\oplus$. $(L, R) \rightarrow (R, L + R)$

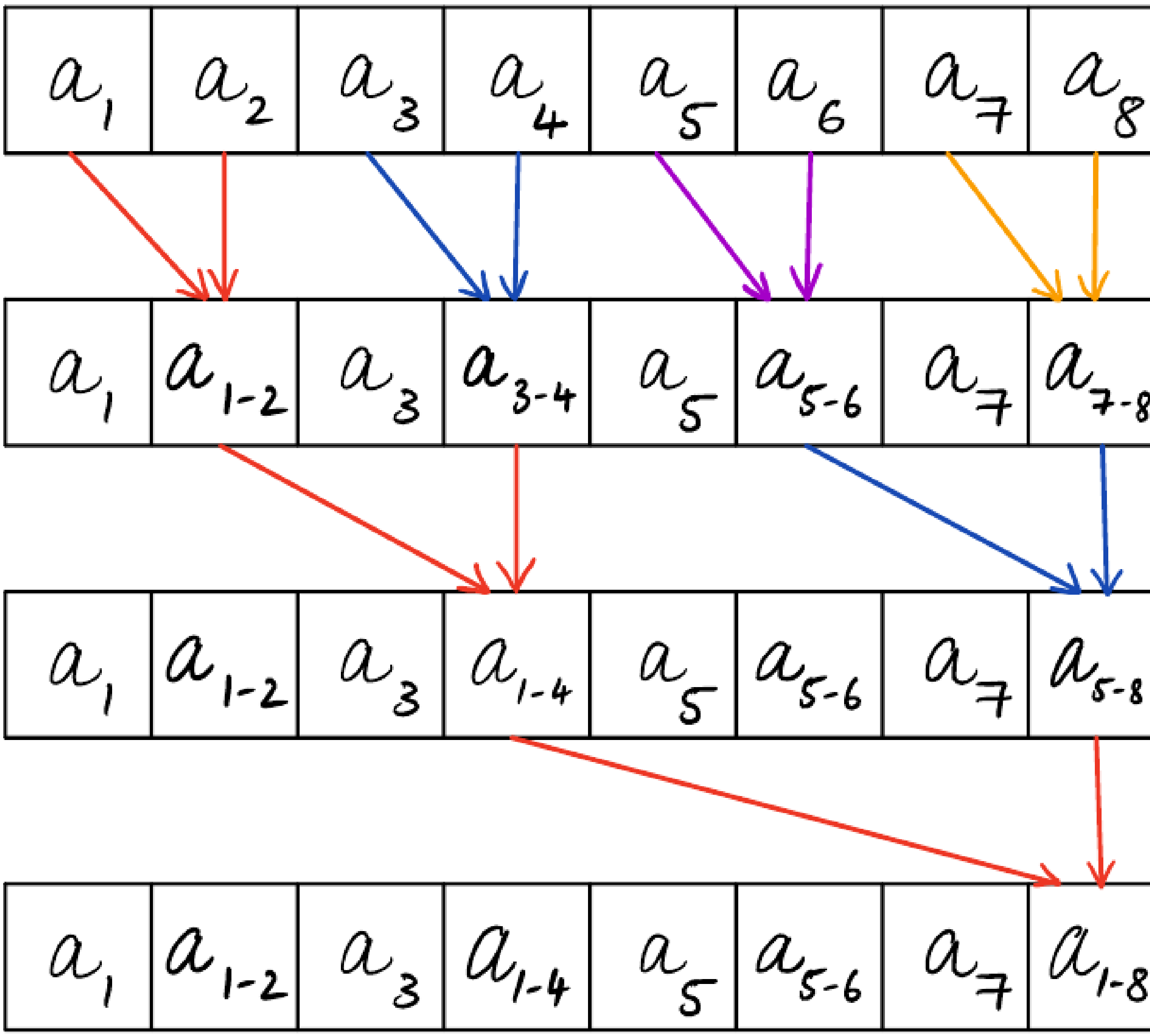
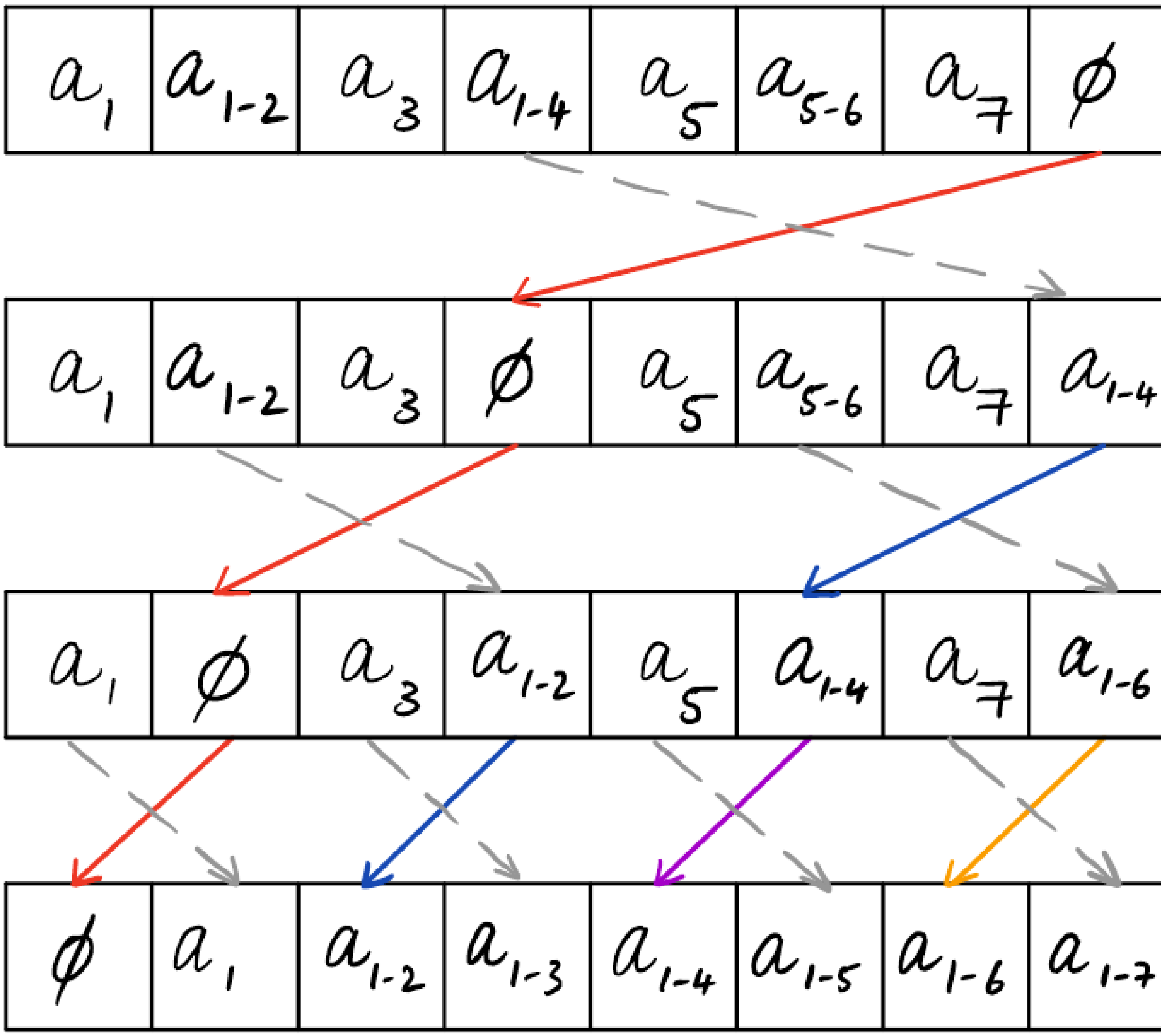


Figure 4: belloch-upsweep



a_1 a_{1-2} a_3 a_{1-4} a_5 a_{5-6} a_7 ϕ
a_1 a_{1-2} a_3 ϕ a_5 a_{5-6} a_7 a_{1-4}
a_1 ϕ a_3 a_{1-2} a_5 a_{1-4} a_7 a_{1-6}
ϕ a_1 a_{1-2} a_{1-3} a_{1-4} a_{1-5} a_{1-6} a_{1-7}

Figure 5: belloch-downsweep