

GPU Architectures SS2019

Dominik Schörkhuber & Thomas M. Galla

Overview

- Execution and measurement framework
- Dimensions of Optimization
- Implementation variants
- Measurement results
- Summary and takeaways

Execution & Measurement Framework

- Automated execution of all algorithm implementations
- Command line configurable
 - sample size, window size, iterations
- Verifying correctness
 - Comparison with Lemire's implementation
- Measurement of (average) runtime of each algorithm
- Easy to add new algorithm implementations

Sample Session

```
$ ./streaming_min_max_comparison -s 10000000 -w 500 -i 11
```

Performing a comparison using the following parameters:

window_size = 500

sample_size = 10000000

number_of_iterations = 11

lemire = 2315.039919 milliseconds

cuda plain - cuda malloc = 234.845854 milliseconds

cuda plain - page locked memory = 3055.436376 milliseconds

cuda plain - page locked shared memory = 103.675406 milliseconds

thrust_naive = 221.098370 milliseconds

thrust = 235.481252 milliseconds

cuda plain - cuda tiled = 127.926657 milliseconds

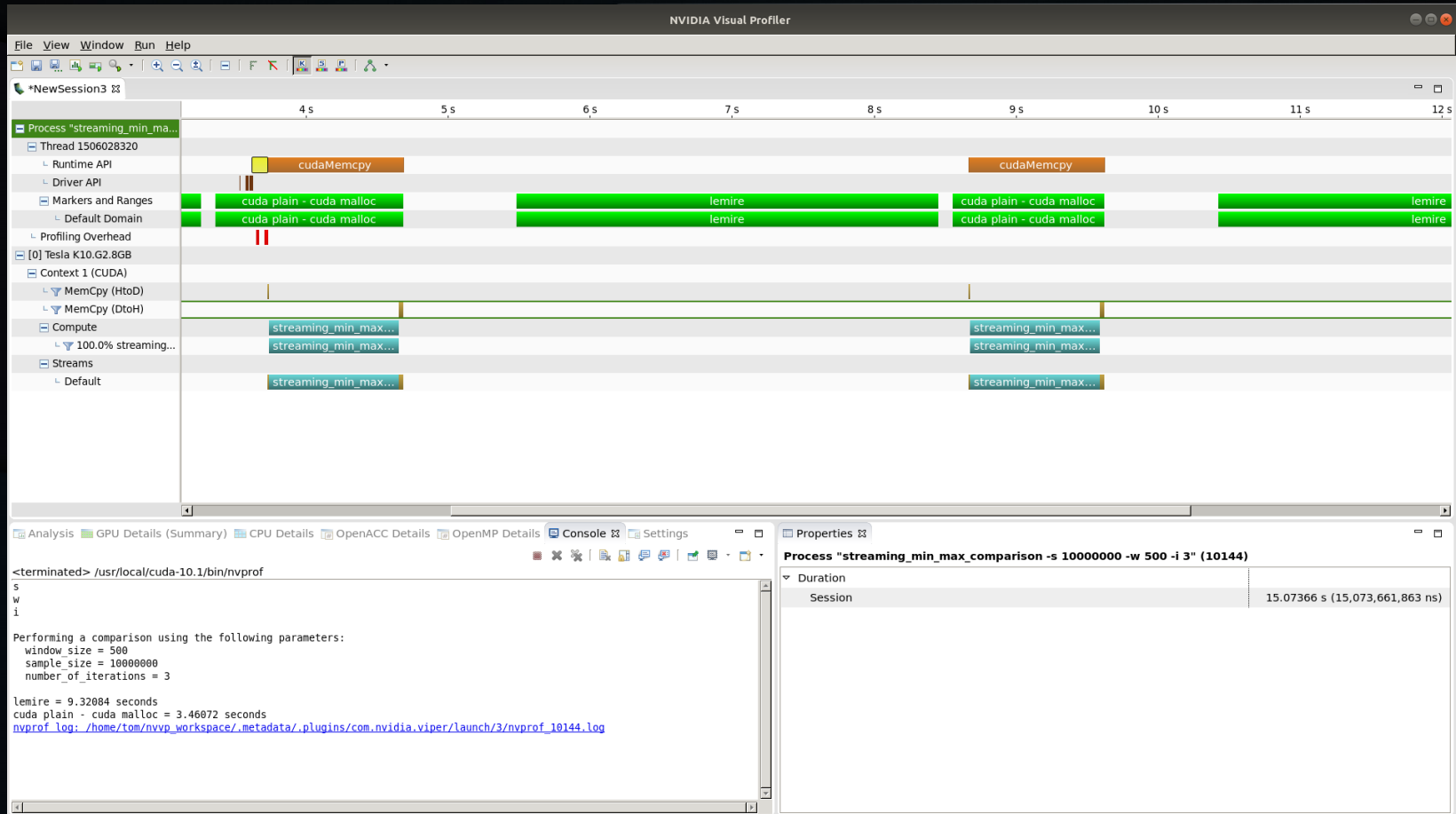
Dimensions of Optimization

- CUDA framework
 - Plain CUDA
 - Thrust
- Memory allocation & data transfer
 - Explicit memory allocation and transfer
 - Page-locked host memory
 - Page-locked host memory + shared memory as cache
- Parallelization strategy
 - One thread per sliding window/output value („linear scan“)
 - Binary reduction and tiling („log linear scan“)

Explicit Memory Alloc. & Transfer (1)

- `cudaMalloc()/cudaFree()/cudaMemcpy()`
- Expectation
 - Allocation overhead
 - Memory transfer overhead
 - Memory copy overhead (1x each direction!)

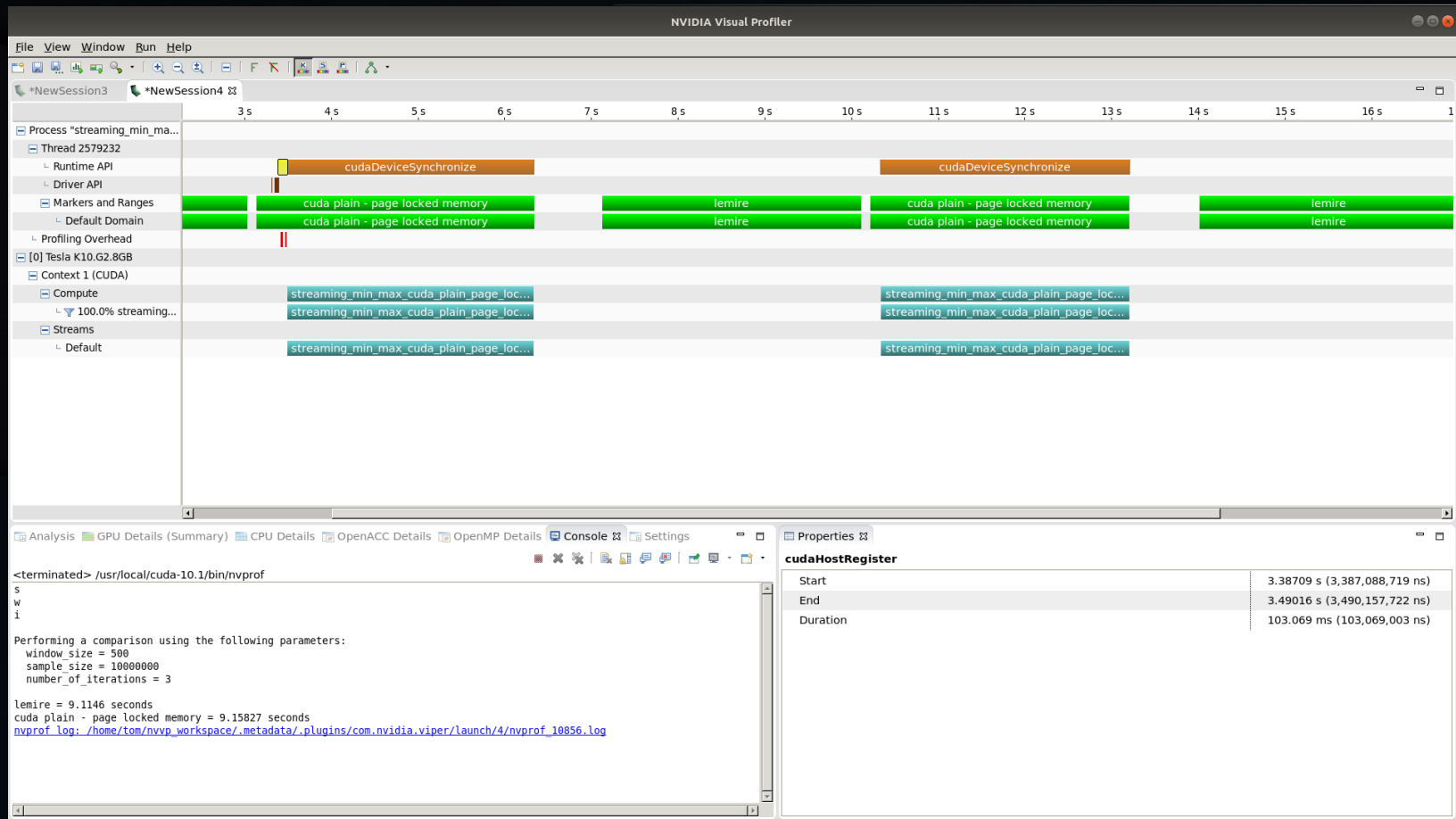
Explicit Memory Alloc. & Transfer (2)



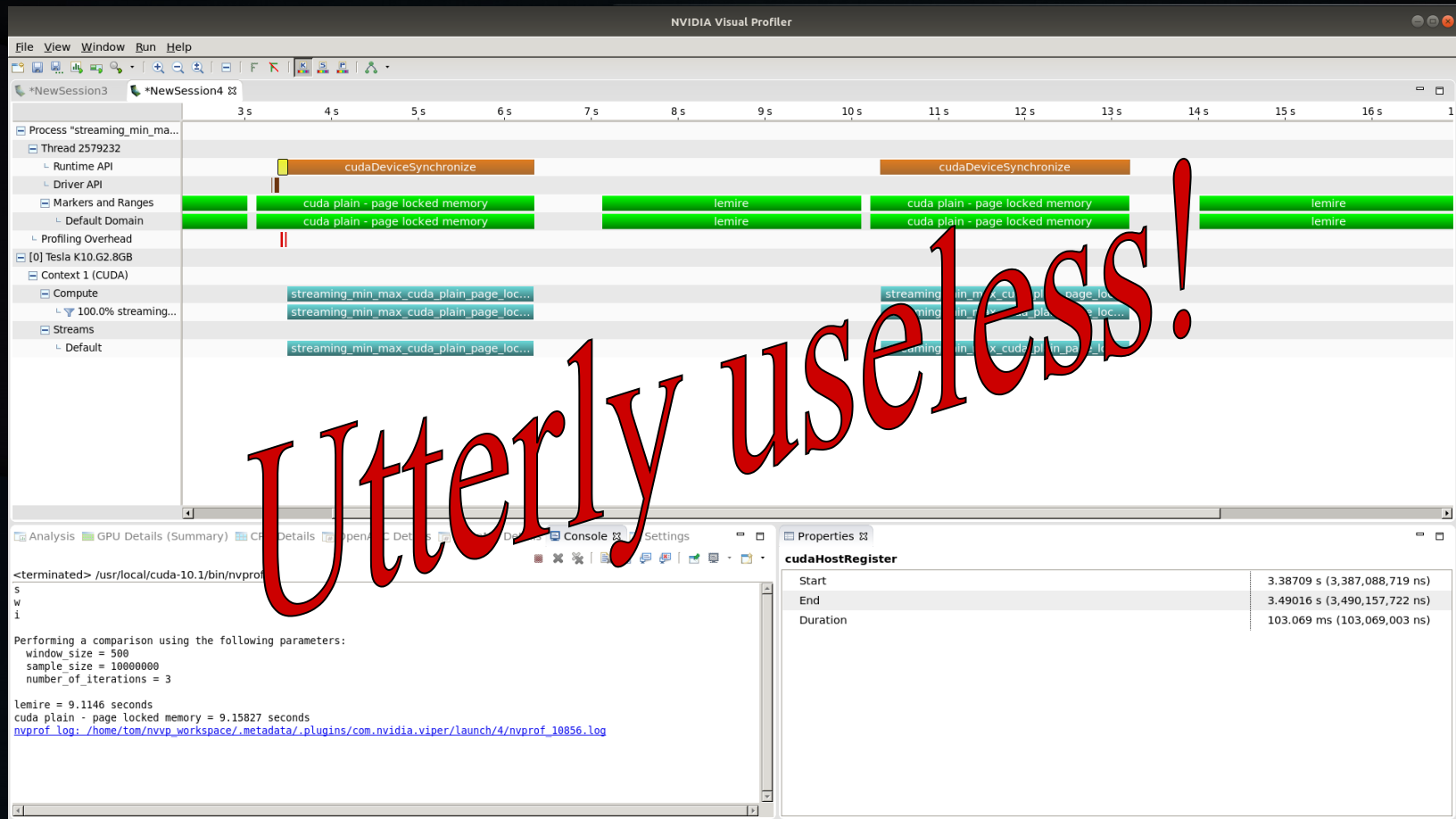
Page-Locked Host Memory (1)

- `cudaHostRegister()/cudaHostDeregister()`
- Expectation
 - Allocation overhead eliminated
 - Memory transfer overhead still there
 - Memory copy overhead eliminated

Page-Locked Host Memory (2)



Page-Locked Host Memory (2)



Page-Locked Host Memory (3)

- Actually a massive deterioration
- Hypothesised causes
 - Lack of possibility to coalesce small memory accesses into single bulk transfer
 - Slow PCI bus
- Interestingly depends strongly on HW platform

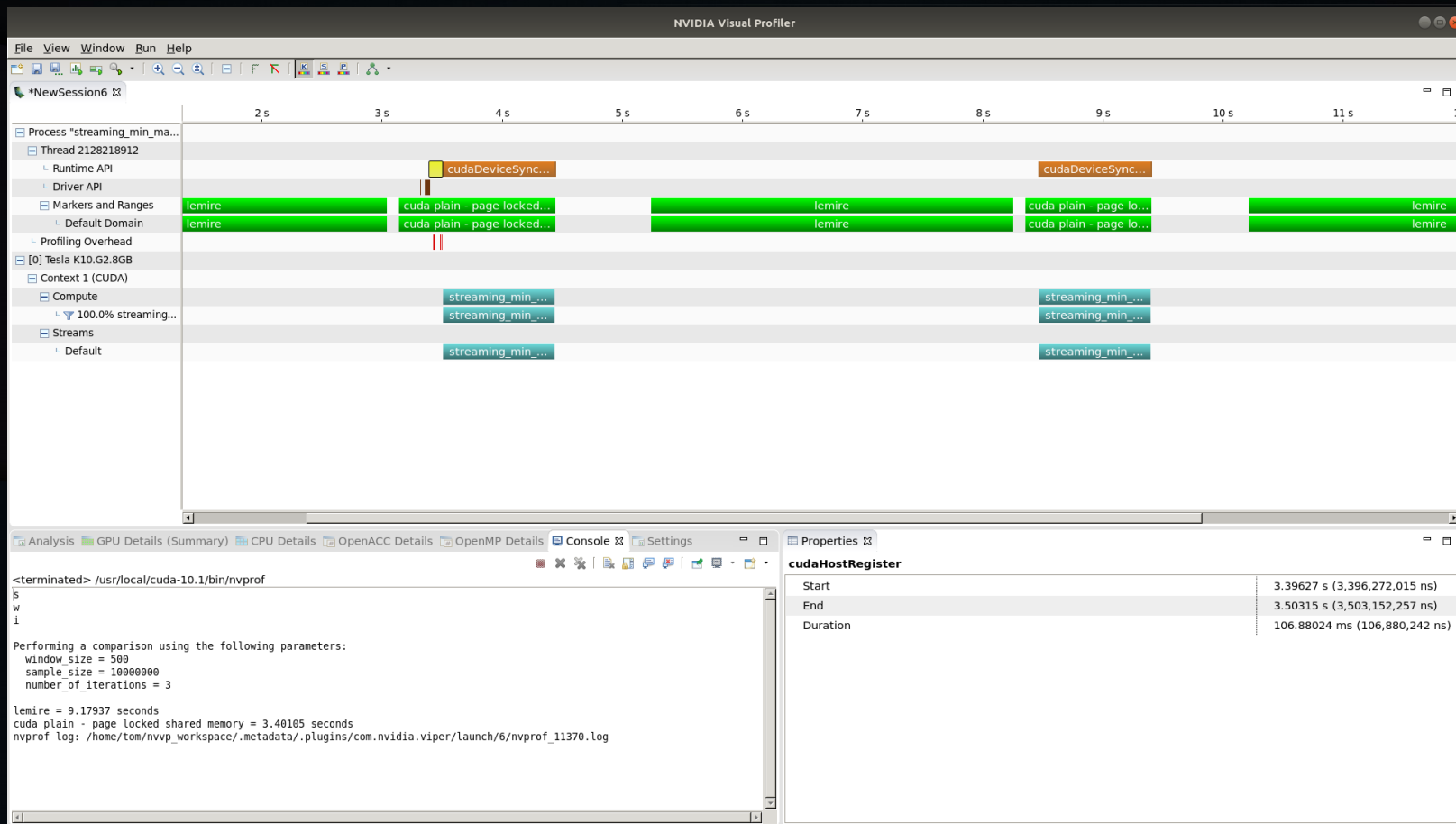
Page-Locked Host Mem. & Cache (1)

- Use page-locked host memory
- Use shared memory on GPU as program controlled cache
 - Shared memory as fast as L1 cache
 - Program controlled instead of LRU
- Expectation
 - Massive memory access overhead of pure page-locked memory drastically reduced

Page-Locked Host Mem. & Cache (2)

- Split computing kernel into two parts
 - Parallelized memory transfer into shared memory
 - Combined effort of all threads of a thread block
 - Actual computation on shared memory

Page-Locked Host Mem. & Cache (3)



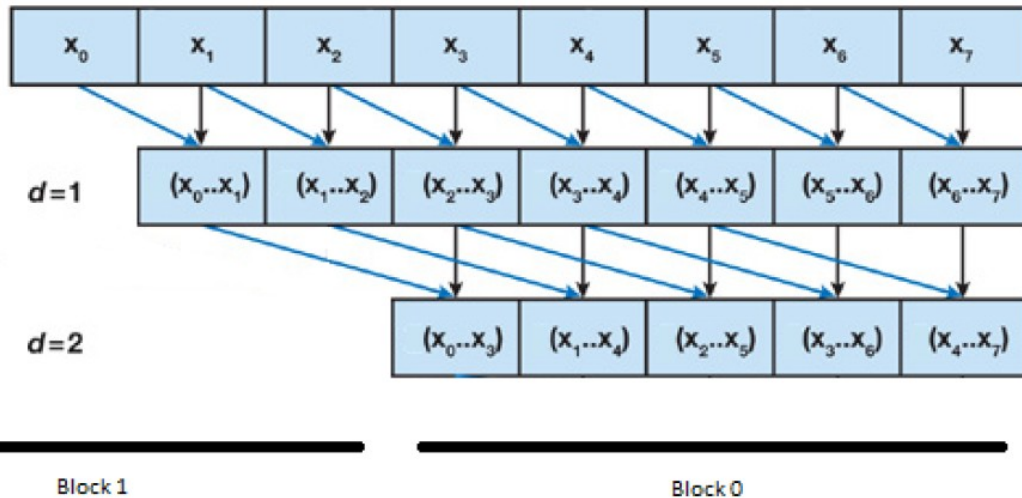
Linear Scan

- One thread per output
- Incremental computation

```
Data:  $w$  = window size,  $s$  = data size  
for each thread  $k$  in parallel do  
  minimum = input[k]  
  maximum = input[k]  
  for each position  $i$  in a window of length  $w$  do  
    minimum = min(minimum, input[k + i])  
    maximum = max(maximum, input[k + i])  
  end  
  minimaout[k] = minimum  
  maximaout[k] = maximum  
end
```


Log Linear Scan

- Binary reduction
- Combine results



Data: w = window size, s = data size

for each thread k in parallel do

for $d = 0; d < \lfloor \log_2(w) \rfloor; d++$ do

if $k + 2^d < s$ then

$\text{minima}[k] = \min(\text{minima}[k], \text{minima}[k + 2^d])$

$\text{maxima}[k] = \min(\text{maxima}[k], \text{maxima}[k + 2^d])$

end

end

if $k < s - w + 1$ then

$\text{minimum} = \text{minima}[k]$

$\text{maximum} = \text{maxima}[k]$

for $i = 0; i < w - \lfloor \log_2(w) \rfloor; i++$ do

$\text{minimum} = \min(\text{minimum}, \text{input}[k + i + 1])$

$\text{maximum} = \max(\text{maximum}, \text{input}[k + i + 1])$

end

$\text{minima}_{\text{out}}[k] = \text{minimum}$

$\text{maxima}_{\text{out}}[k] = \text{maximum}$

end

end

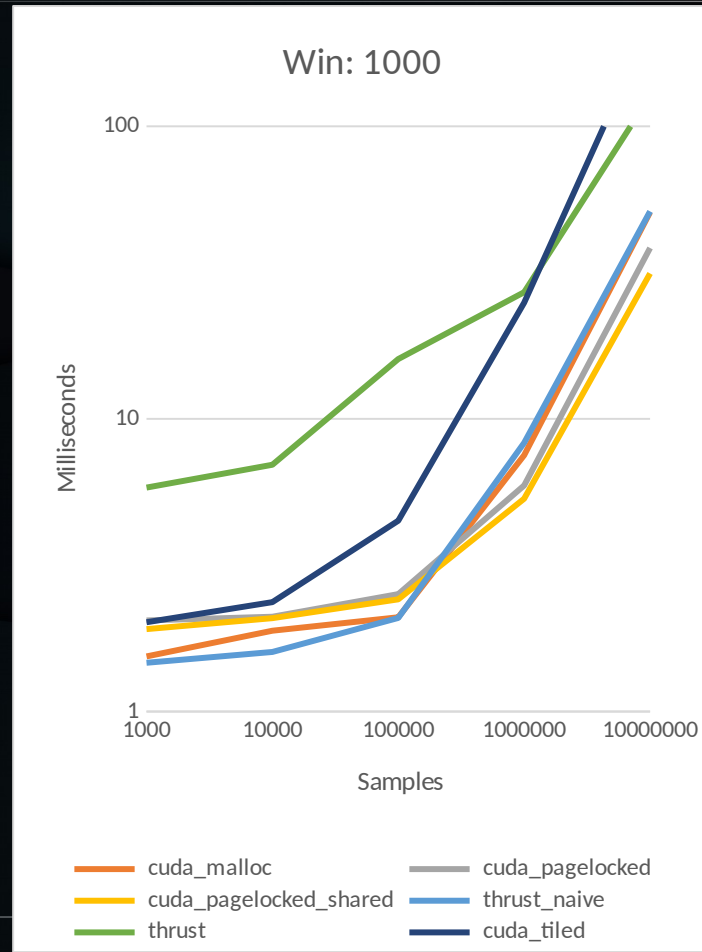
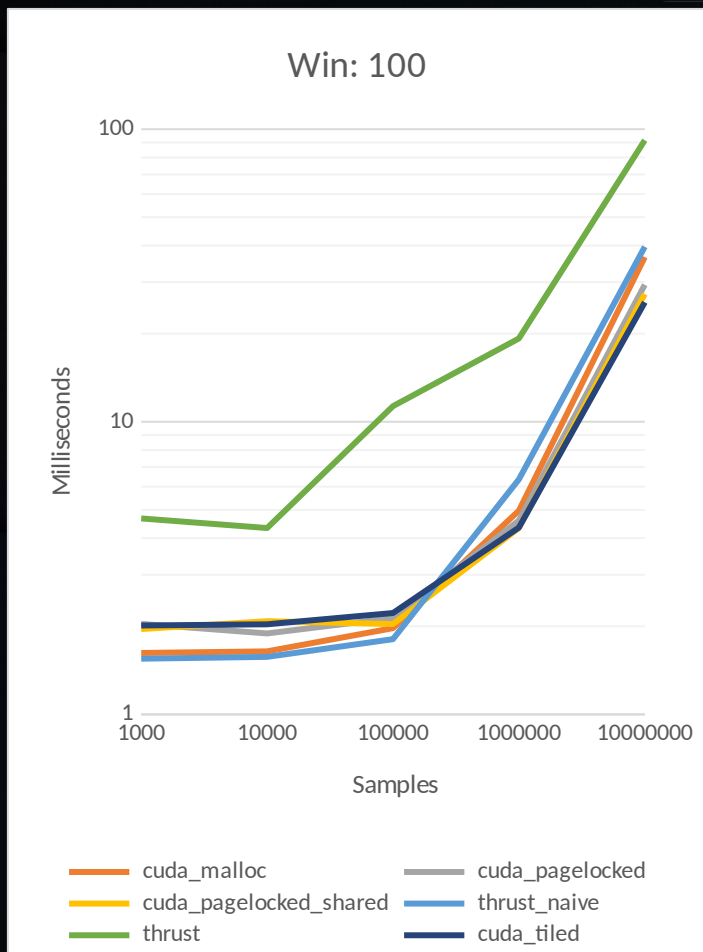
Implementation of Algorithms

- CUDA
 - `cuda_malloc`: malloc – linear scan
 - `cuda_pagelocked`: page-locked – linear scan
 - `cuda_pagelocked_shared`: page-locked & shared memory – linear scan
 - `cuda_tiled`: page-locked & shared memory – log linear scan & tiling
- Thrust
 - `thrust_naive`: linear scan
 - `thrust`: log linear scan & tiling

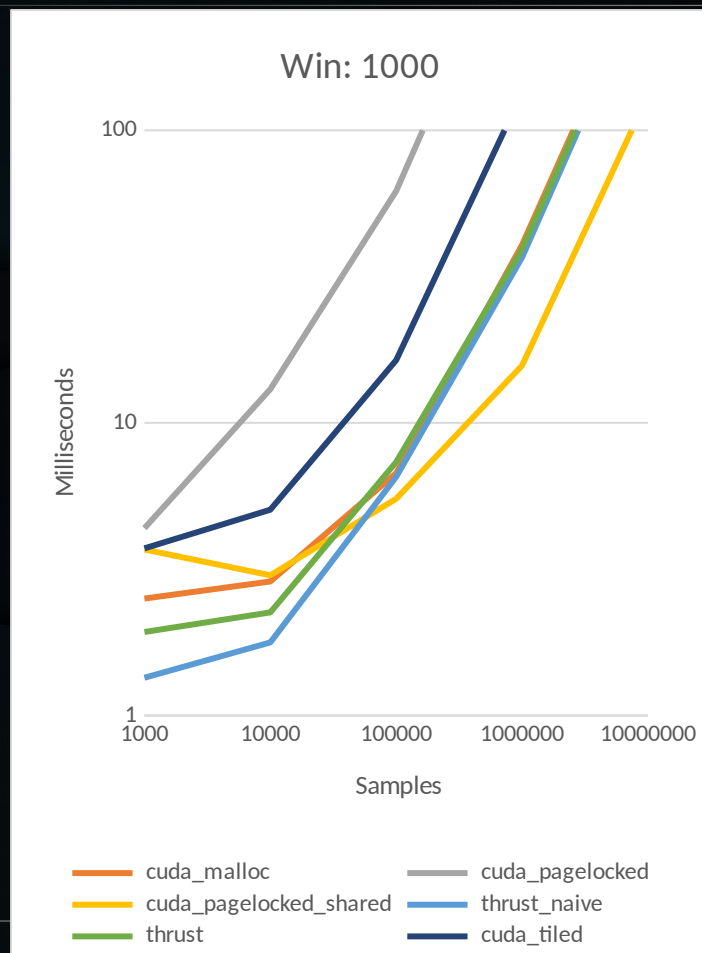
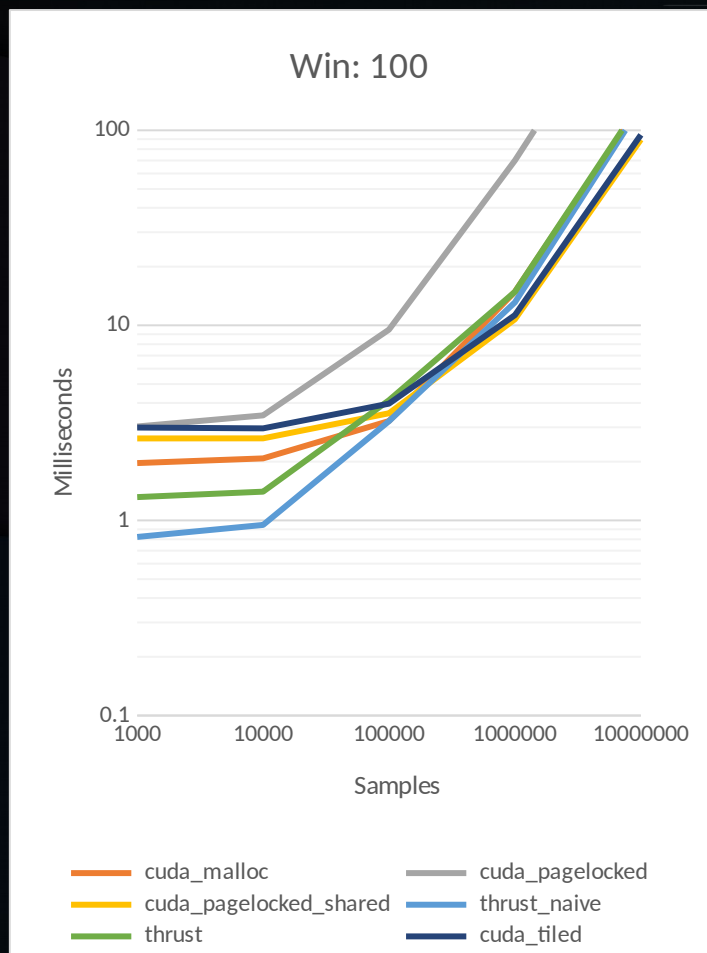
Measurements

- Conducted for all algorithm flavors
- For various sample and window sizes
- On two different HW platforms
 - NVIDIA RTX2070
 - NVIDIA Tesla K10.G2

Results – NVIDIA RTX 2070



Results – NVIDIA Tesla K10.G2



Summary & Takeaways (1)

- Performance of code running on a GPU is influenced by a multitude of factors, thus
 - Know your workload and your HW
 - Measure carefully and optimize
 - If measurement result are inconclusive, measure again ...
- Optimization for one HW/workload might be a deterioration on another
 - Consider going for a hybrid solution
 - multiple kernels
 - select appropriate one during run-time (based on HW and workload)

Summary & Takeaways (2)

- Don't be too clever! - a straight forward „naive“ solution might outperform a complex „smart“ solution

Summary & Takeaways (2)

- Don't be too clever! - a straight forward „naive“ solution might outperform a complex „smart“ solution
- „Premature optimization is the root of all evil“
[Donald Knuth]

