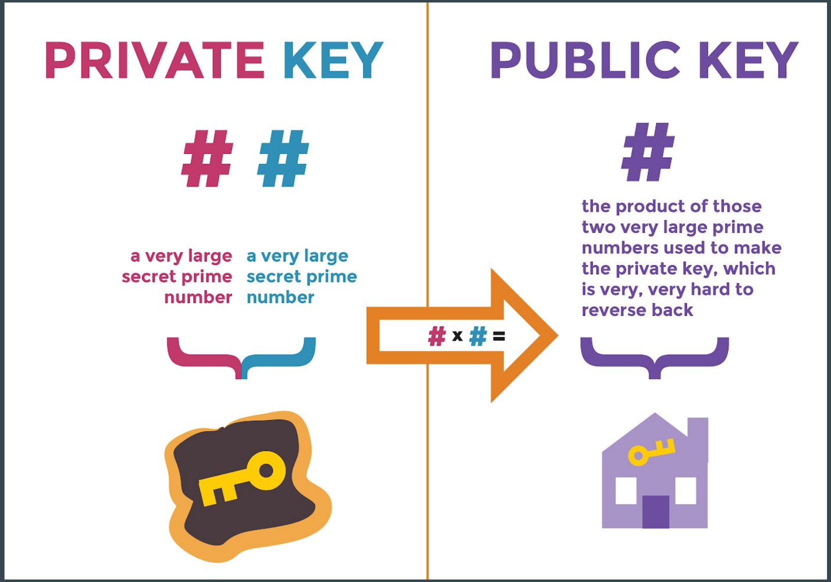
Widening the Sieve

Richard Muniu, Mickey Haregot, Kevin Zheng

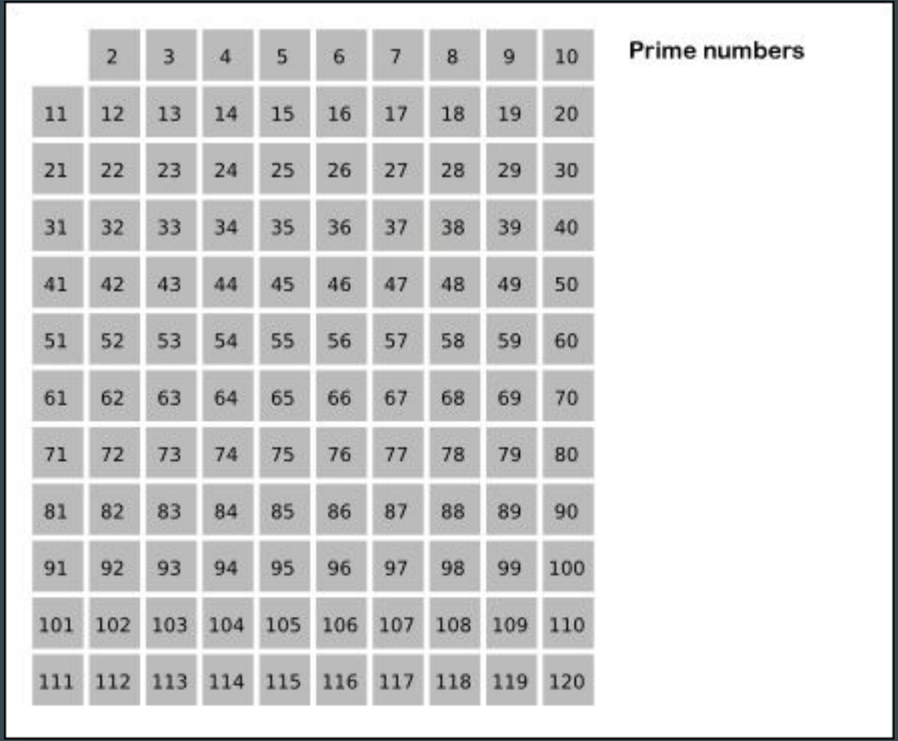
Finding Prime Numbers

- Prime numbers hard to find efficiently
- This makes them useful in security protocols
  - E.g. public/private key encryption
  - Rivest-Shamir-Aldeman (RSA) algorithm
- 2000 year-old algorithms still surprisingly effective today



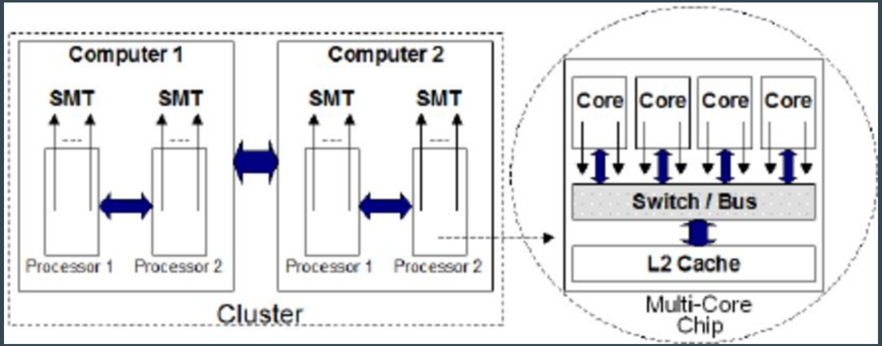
The Sieve of Eratosthenes

1. Have an array for each integer between 0 and some max N
2. Iterate through the array, whenever you find an “unmarked” element, mark all of its multiples.
3. All unmarked elements by end of the run are prime numbers
4. Features: unbalanced load, not embarrassingly parallel, low computation per step



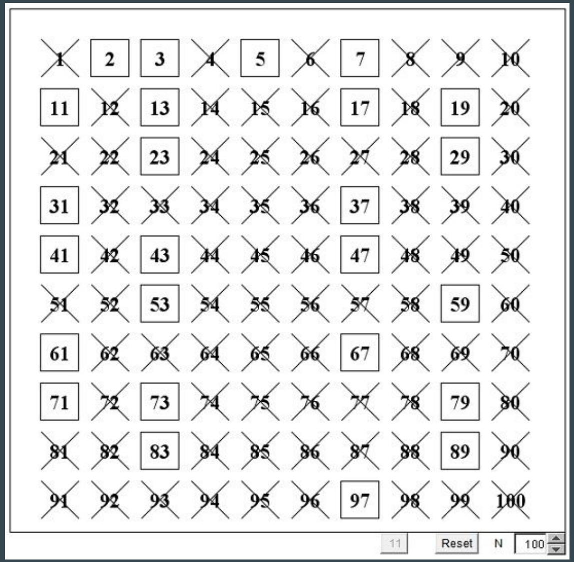
System Traits Desired

- Decided to use single machine
  - No communication
  - Used single node on cluster (Strelka)
- Reasons why
  - Sieve algorithm relatively simple
  - Using multiple nodes result in large communication costs
  - Results required to be aggregated at end of runs
    - May become large part of total runtime



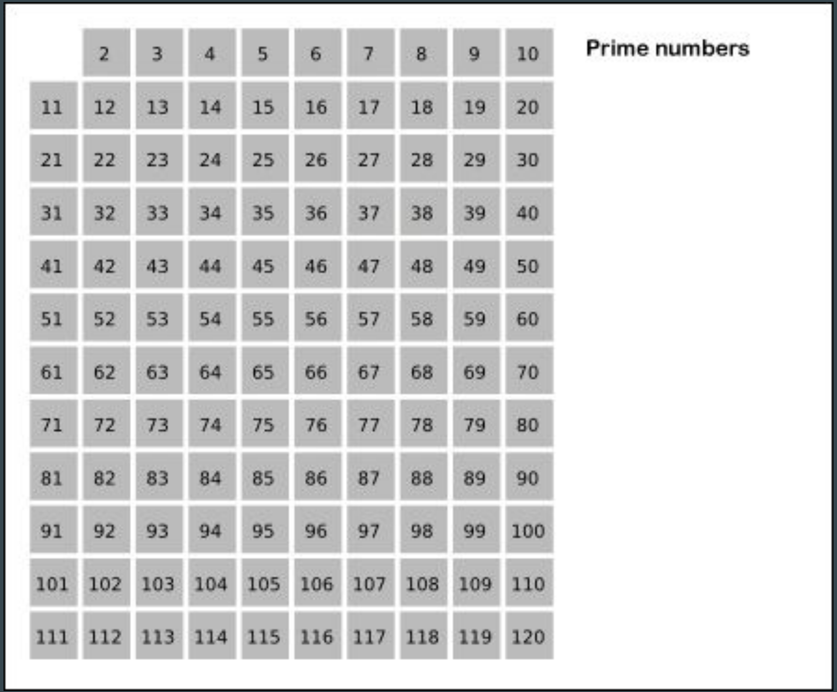
Methods of Implementation

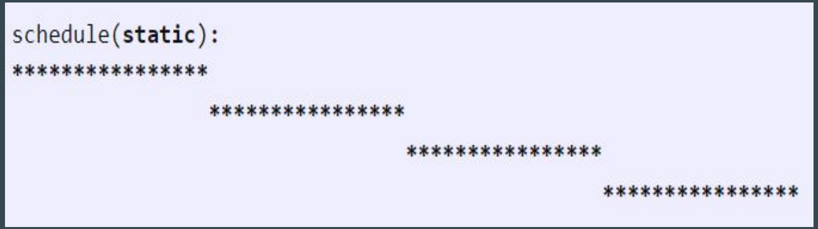
- Sequential
  - Baseline test
- OpenMP
  - Parallelized by base: each thread investigates a “unmarked” number separately
- CUDA
  - Massively parallel version of OpenMP implementation.
  - Takes advantage of SIMD architecture
- Balanced Pthreads
  - Each thread is responsible for a different section of the array and the base process broadcasts “bases” for the other threads to investigate

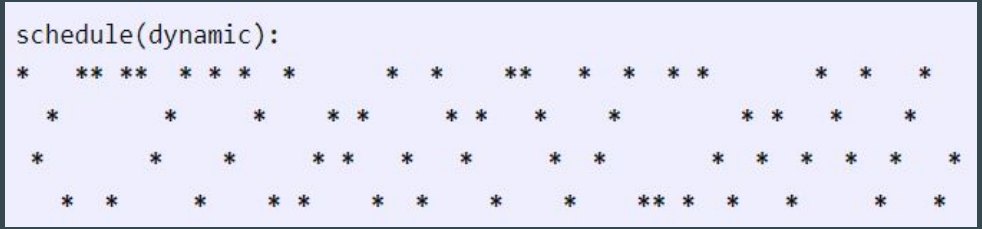


OpenMP

- Uses OpenMP's loop primitives
- Marks multiples in parallel
- Load balancing problem: heavy work assigned to one thread
- Resolved using dynamic scheduling of iteration chunks

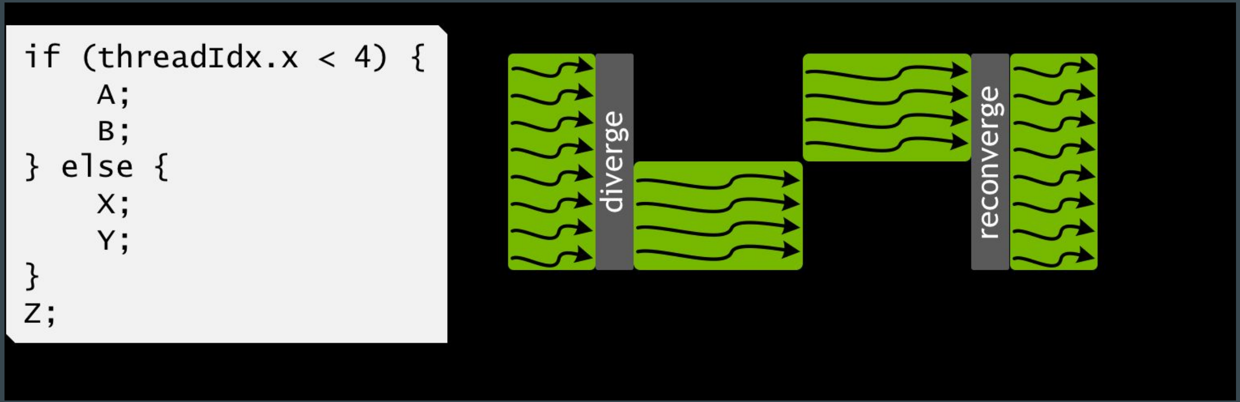






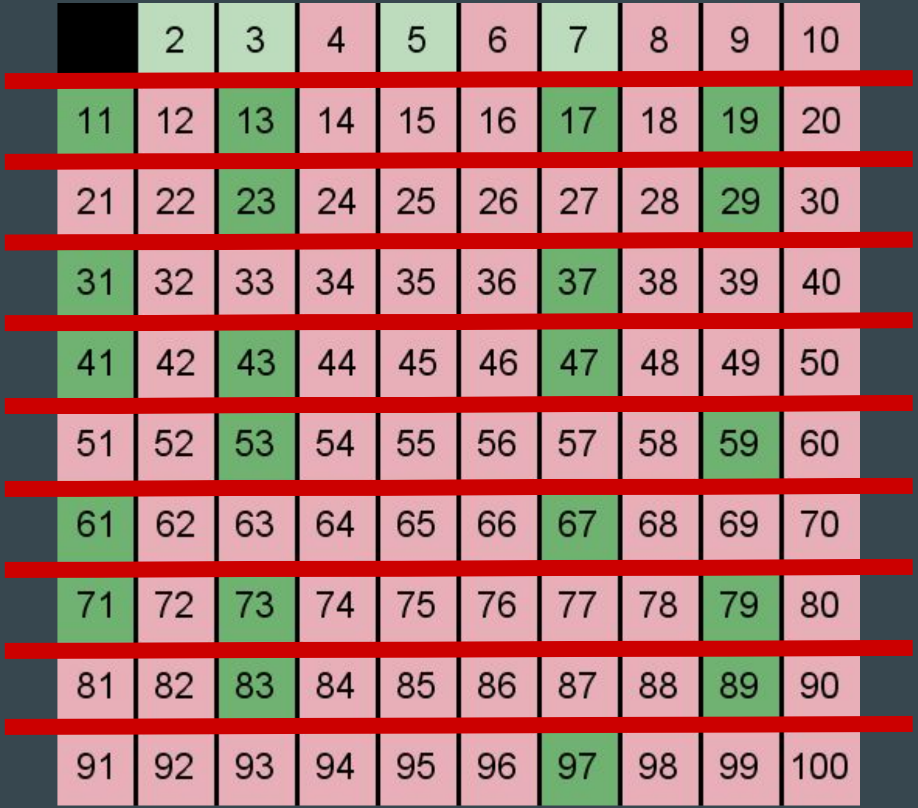
CUDA

- Similar to OpenMP implementation
- Massively parallel
- Every possible “base” has its multiples simultaneously marked off
- Takes advantage of SIMT architecture
- Multiple kernel calls to combat max-thread limitations



Balanced pthreads

- Splits up array into segments
- Master-worker model
- Meant to balance workload
- Master discovers “bases” and broadcasts them to workers
- Broadcast uses shared variable and barriers



Testing

- Compared sequential vs parallel runtimes on Strelka
- OpenMP/balanced pthreads: tested both seq_sieve and par_sieve
    - 32-core CPU
    - hi-mem partition
- CUDA: ran cudaSieve on
    - 4x NVIDIA 2080 Ti GPUs
    - gpu-02 partition.
- For both OpenMP and balanced pthread, dynamic allocation in RAM failed for arrays of size 10^{13} and above.
- For CUDA, dynamic allocation in DRAM failed for arrays of size 10^{12} and above.
- Used average values from three runs of seq_sieve and par_sieve for each method.

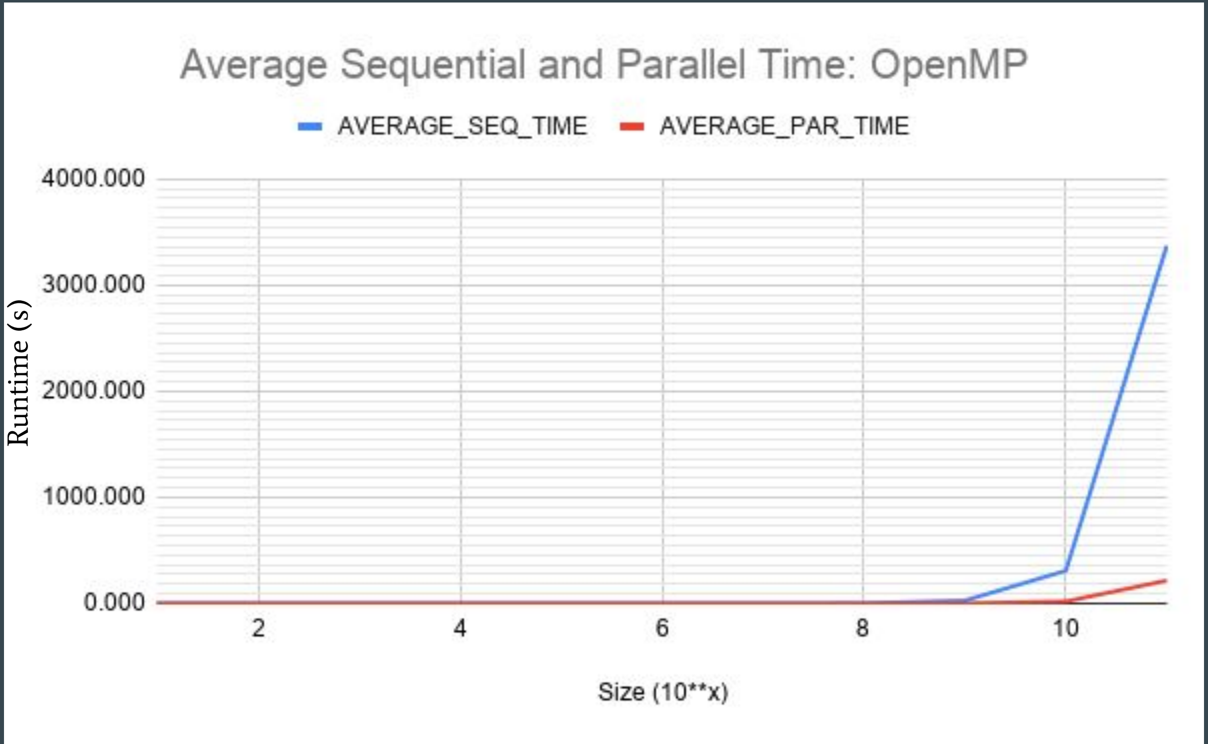
RESULTS AND DISCUSSION

OpenMP

- Nonlinear speedup of up to 16
- No speedup until arrays of size 10^7

Size (10^x)	Average Sequential Time(s)	Average Parallel Time(s)	Speedup
1	0.001	0.017	0.058
2	0.001	0.016	0.063
3	0.001	0.017	0.059
4	0.001	0.016	0.061
5	0.001	0.018	0.057
6	0.008	0.017	0.462
7	0.073	0.027	2.750
8	2.050	0.201	10.182
9	27.495	1.923	14.298
10	308.771	19.372	15.939
11	3371.608	216.177	15.596

OpenMP



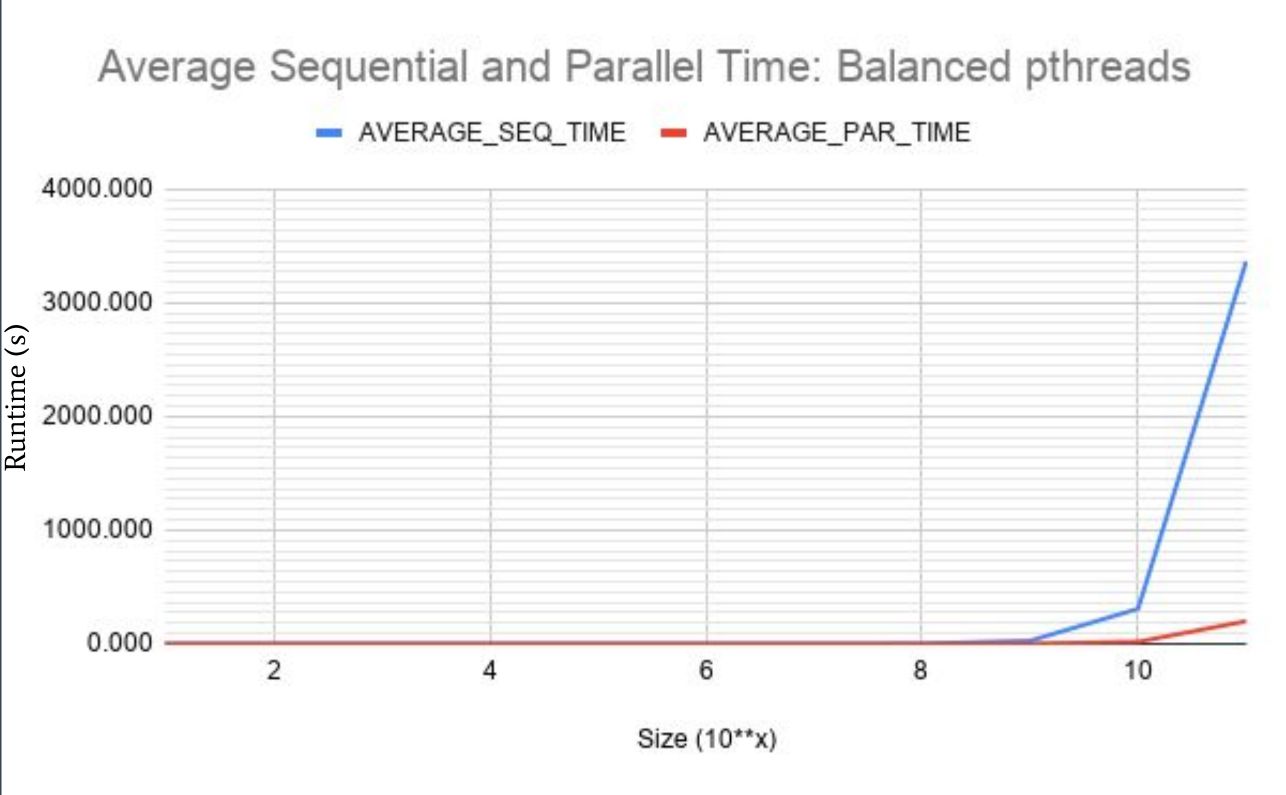
Average Sequential and Parallel Time: OpenMP
AVERAGE_SEQ_TIME
AVERAGE_PAR_TIME
4000.000
3000.000
2000.000
1000.000
0.000
Runtime (s)
2
4
6
8
10
Size (10**x)

Balanced pthreads

- Nonlinear speedup of close to 17
- No consistent speedup until arrays of size 10^7

Size (10^x)	Average Sequential Runtime	Average Parallel Runtime	Speedup
1	0.005	0.002	2.500
2	0.001	0.002	0.429
3	0.001	0.003	0.375
4	0.001	0.005	0.200
5	0.001	0.009	0.154
6	0.007	0.016	0.458
7	0.071	0.046	1.547
8	2.204	0.228	9.653
9	27.518	2.440	11.278
10	308.638	18.786	16.429
11	3364.497	199.944	16.827

Balanced pthreads

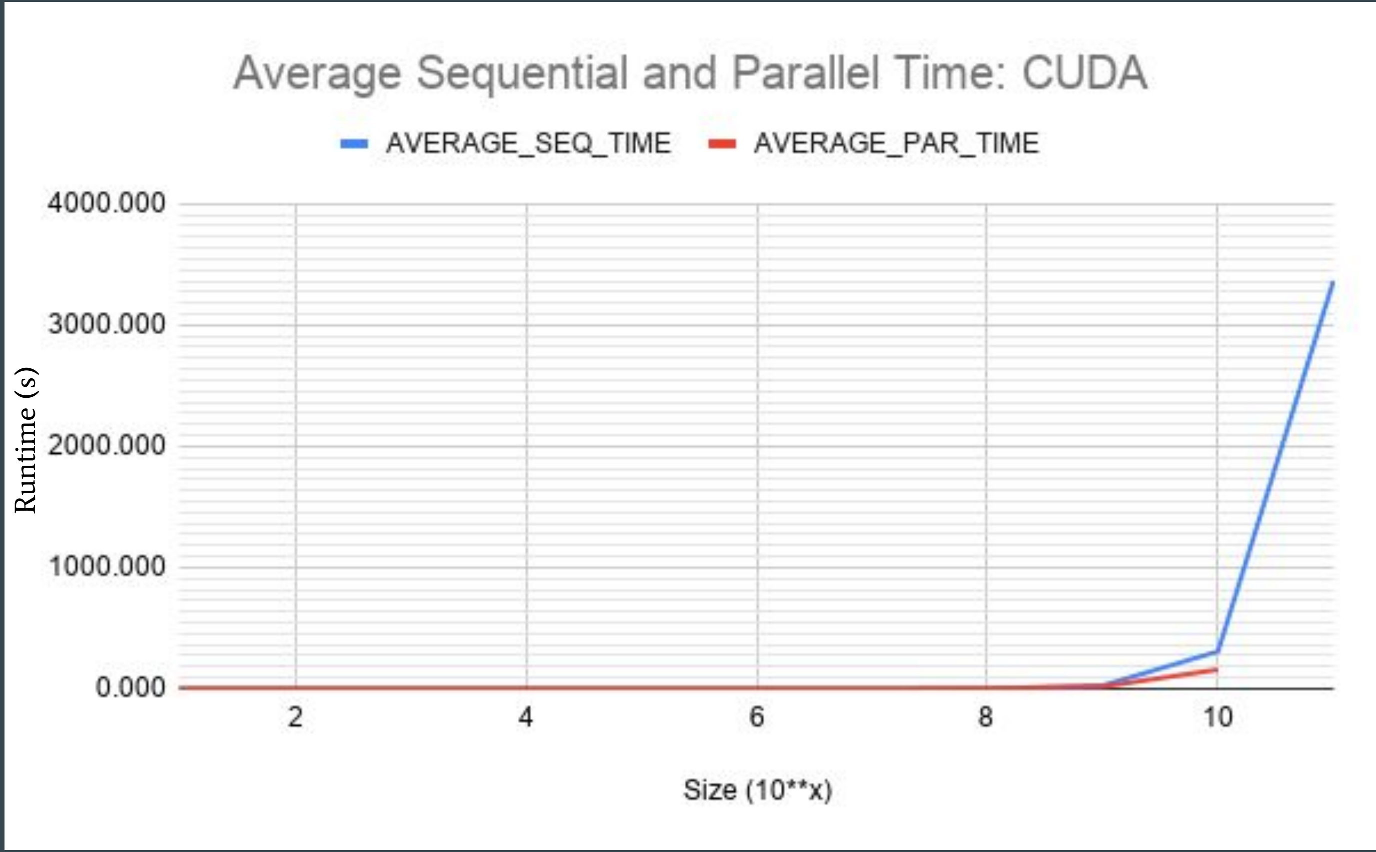


CUDA

- Nonlinear speedup of up to 2
- No consistent speedup until arrays of size 10^9, but relatively consistent runtime until that point
- Lower performance than both other implementations

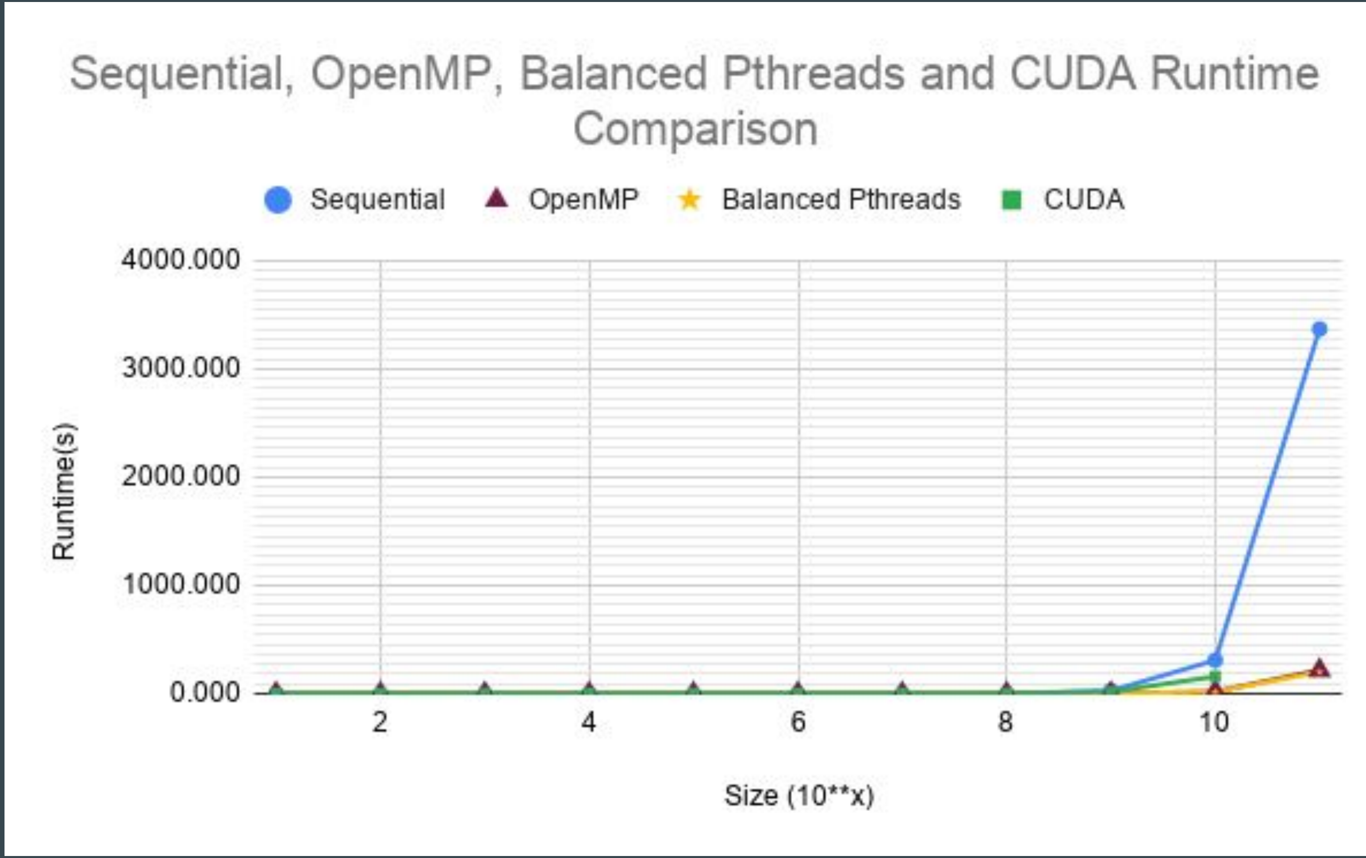
Size (10^x)	AVERAGE SEQ TIME	AVERAGE PAR TIME	Speedup
1	0.005	3.128	0.002
2	0.001	3.039	0.000
3	0.001	3.060	0.000
4	0.001	3.057	0.000
5	0.001	3.076	0.000
6	0.007	3.090	0.002
7	0.071	3.295	0.021
8	2.204	4.536	0.486
9	27.518	17.802	1.546
10	308.638	158.962	1.942
11	-	-	-

CUDA

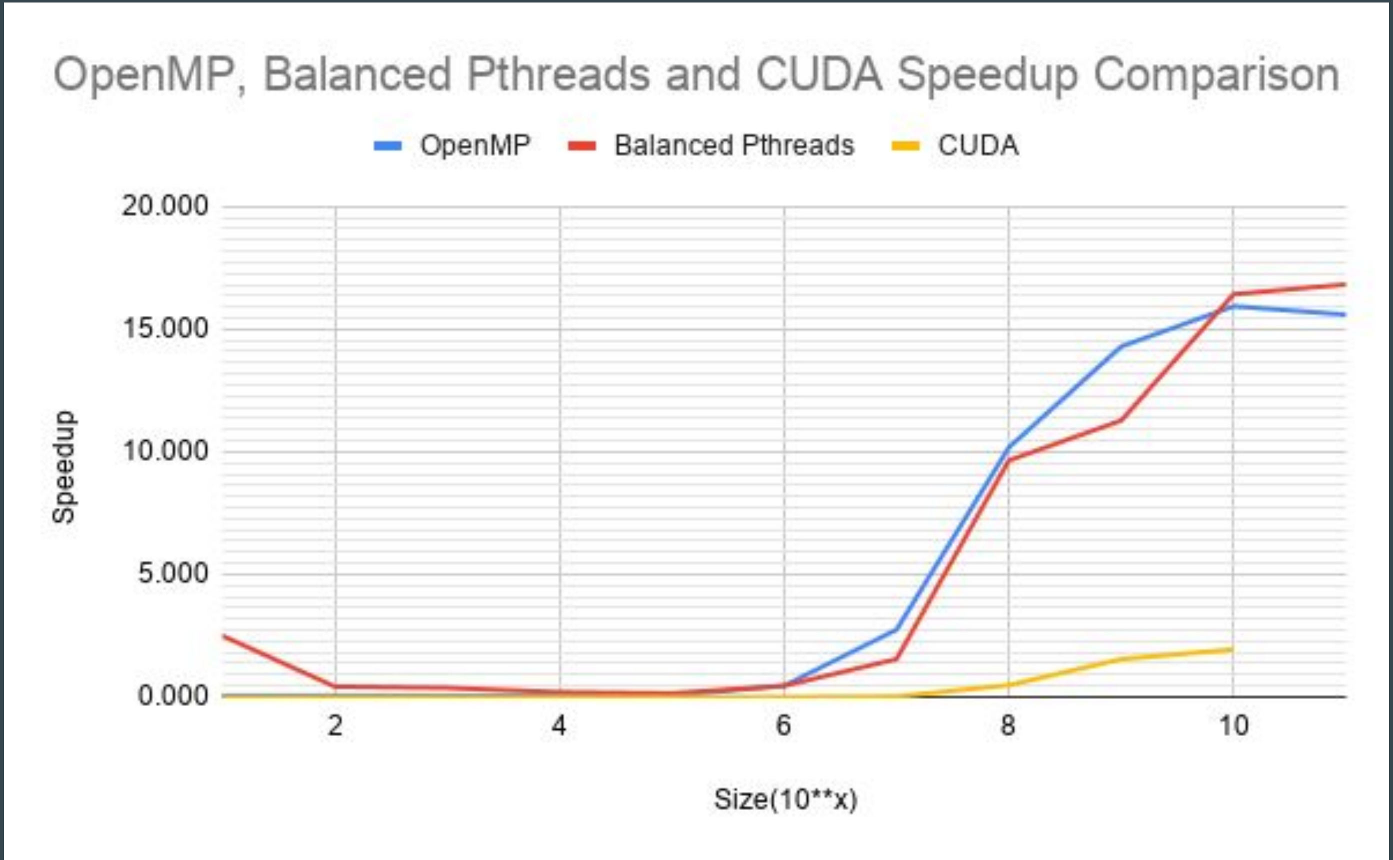


Average Sequential and Parallel Time: CUDA
AVERAGE_SEQ_TIME
AVERAGE_PAR_TIME
4000.000
3000.000
2000.000
1000.000
0.000
Runtime (s)
2
4
6
8
10
Size (10**x)

Runtime Trend and Comparison



Speedup Trend and Comparison



Conclusion and Future Directions

- A lot to gain from parallelization in form of speedup and efficiency
- Benefits are not obvious/present at low task sizes due to paralellization overheads
- CUDA performed slower than other methods
  - hardware limitations?
  - Potential software optimizations?
- Some future directions to consider:
  - Further optimizations to CUDA implementation
  - Explore distributed memory approach to eliminate max array size limitation

QUESTIONS

kzheng1@swarthmore.edu

mharego1@swarthmore.edu

rmuniu1@swarthmore.edu