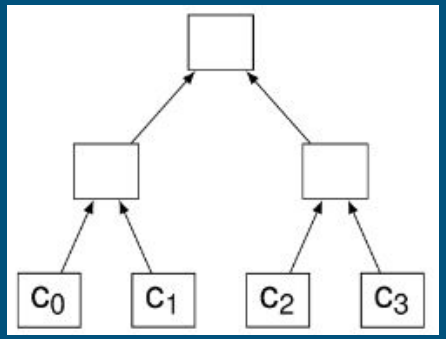
Parallelizing BLAKE3

Rehan Vipin & Samarth Mathur
Team - 6

Recap

BLAKE3 is the fastest cryptographic hash function (throughput - 5 GB/s). It has security equivalent to (or better than) existing standards and is being widely adopted.

It has a merkle tree structure, allowing unbounded parallelism. ->



At the end of phase-1 we had a serial-algorithm based version ready. It was a simple but a complete port of the BLAKE3 implementation.

We discussed multiple approaches available.

In Phase-2 we :

- Implemented ¾ of the them
- Design+Implemented a new algorithm
- Benchmarked, documented and examined, in-detail, every version we made.

10 MB/s -> 1.2 GB/s

4 Different techniques

- OpenMP
- SIMD (AVX2)
- GPU (Dark)
- GPU (Dynamic parallelism)

Before all of that, we needed a base -

A new technique in BLAKE3 hashing, parallelize wherever possible. We call this BLAZE3. Explained through diagrams from the notebook:

Rest through code and demo

Phase-2 Timeline

1. Design and implement new parallel algorithm
2. Implement openmp version and testing framework
3. Optimize OMP, profile, collect many tests
4. GPU - set up base (a & b done in parallel)
    a. Dark version
    b. Dynamic parallelism version
5. Profile, identify hotspots in GPU version, improve it
6. Implement SIMD v1, v2 ...

Optimizations, major and minor, were being done continuously.

OpenMP

Needed a fork-join model with nesting.

Used tasks in openmp with 2 threads and 3 levels of nesting.

Heavy on CPU - ~80%. Tried to set up enhancers - https://github.com/pmodels/bolt but documentation is horrible. Async too slow.

Designed to be closely related to serial version - just use the `-fopenmp` flag.

More in benchmarks...

GPU (Dynamic Parallelism)

Will show shortly.

GPU (Dark)

Same `hash_many` interface, but internally uses a new `dark hash` technique.

Similar to parallel-reduction, but not exactly!

Tried many techniques for improvement:

- Shared memory, reduce local memory ...
- Algorithmic improvements to minimize kernels & divergent code
- Thrust and pinned memory
- Zero-Copy Registered memory -> slowest!

More soon...

SIMD

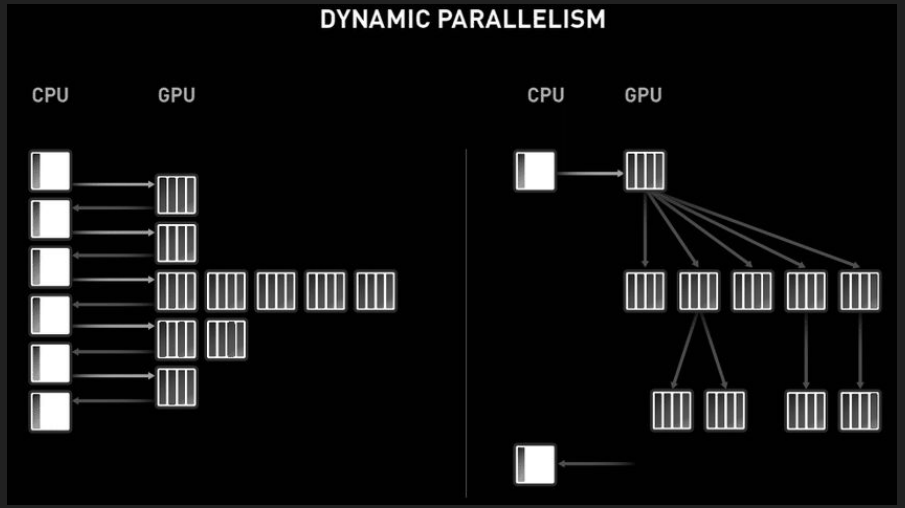
AVX2 provides access to 128 & 256 bit registers to speed up core functions. AVX-512 is not yet widely used.

Activated only along with OpenMP. Together, they give us the fastest throughput.

Can also be used in single-thread mode but technically a “parallelism”, therefore separated.

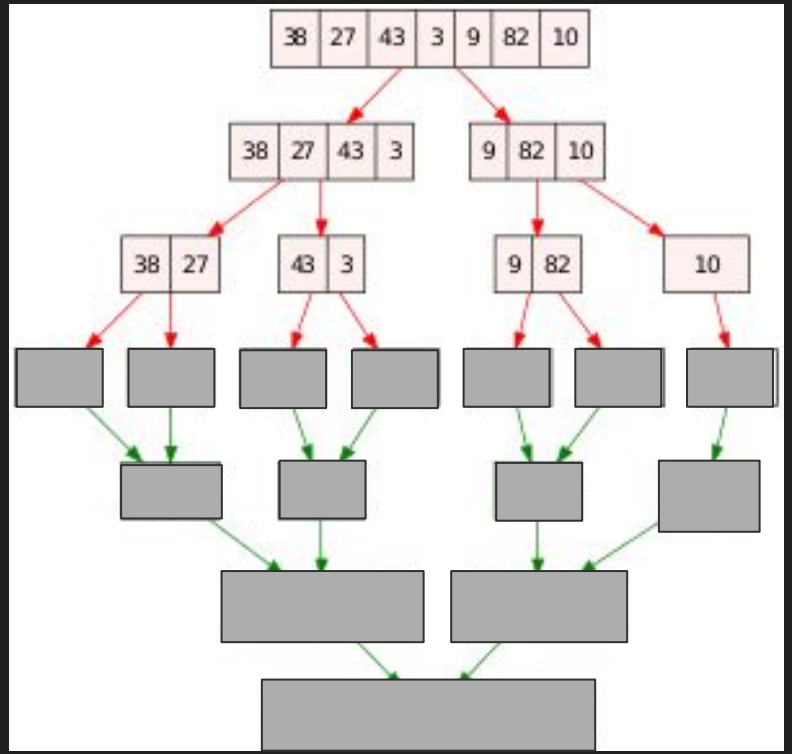
Not available in GPUs :(

Dynamic Parallelism

- It is an extension to the CUDA programming model enabling a CUDA kernel to create new thread grids by launching new kernels.
- 

- Used in algorithms like “The Mandelbrot Set”, essentially algorithms where recursive nature can’t be avoided.

How did we try it?



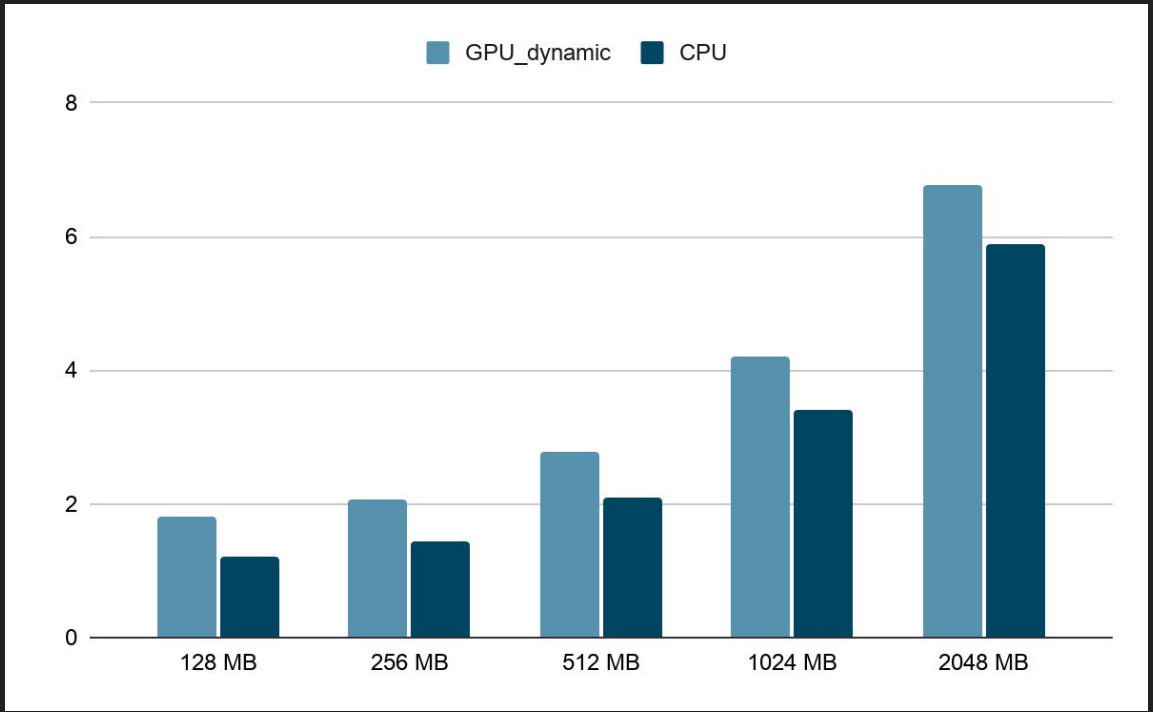
- A divide and conquer approach, somewhat like merge sort.
- Kernels launching recursively for every half.
- We split from the chunk in the middle
- Code walkthrough
- Unfortunately, it was slower and also incorrect.

Where did we fail?

- Intense testing and debugging for the past one week.

- So what all did we try to mitigate this?
  - d_actual_compress , a kernel in our implementation was passing individual test cases but was failing inside the program itself
  - Wrote individual test cases of each of these function
  - Wrote individual tests for the smaller kernels
  - In conclusion we found that d_actual compress was indeed receiving the right values but was “returning” wrong ones, even though its individual case was passing and the values computed inside were right.
  - A detailed .cu file of this test can be found in test_in directory
- But, the hashes were consistently wrong, so we did do a speed run.

Benchmarks

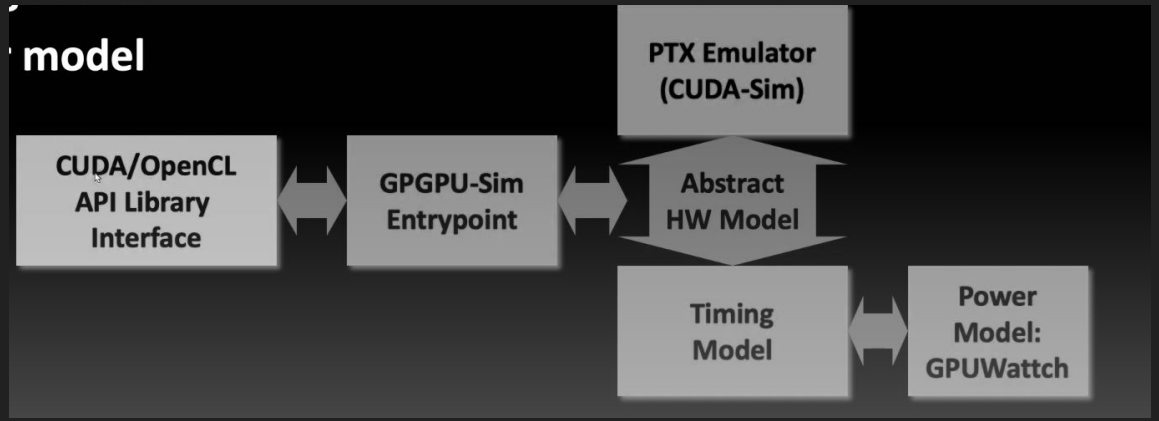


GPGPU-SIM

Optimization at the architectural level, not quite :(

What is GP-GPUSIM

- It is a cycle level simulator that can allow you to “make” architectural changes. You can tweek things at the architectural level by editing a configuration file.
- You can change things at the high level and low level.
-



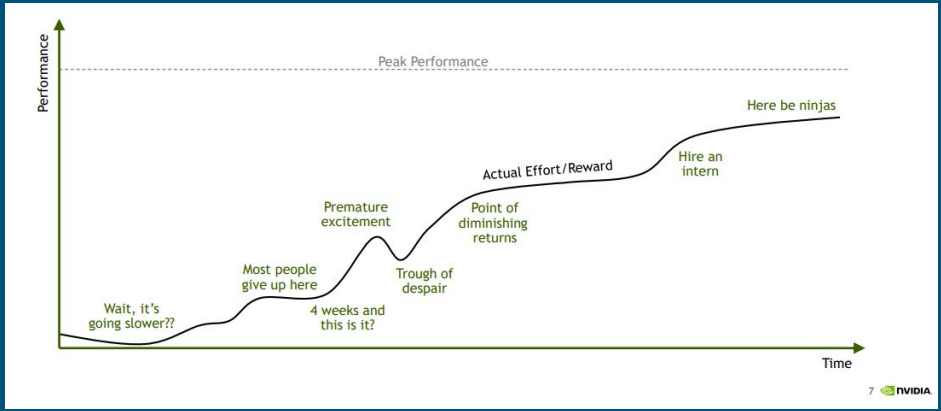
Why did this not work?

- Dark Hash uses a PTX instruction called “shf.l.wrap.b32” which is not yet implemented by gp-gpusim.

- The Dynamic parallelism approach did not work because dynamic parallelism throws error for newer versions of CUDA in gp-gpusim.

- The PTX instruction could’ve been implemented but we were short on time as it required a good amount of documentation and codebase exploration. Programming gp-gpusim for handling Dynamic Parallelism seemed like a task too complex.

Inferences, benchmarks & more...



Primitive implementations:

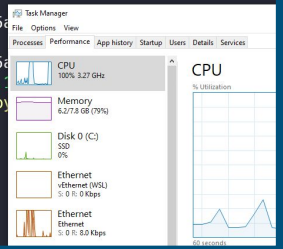
```
apollo_nox cuda on cpu-thread ❯ python bench.py 512000000 og
File size in bytes:  512000000
8e9a08f9552b774d4c08d5735715da83755a4feec0e6b3c88276cffdbb37e1ef
 Execution time: 20.88s
```

Serial

```
apollo_nox cuda on cpu-thread ❯ python bench.py 512000000 og
File size in bytes:  512000000
8e9a08f9552b774d4c08d5735715da83755a4feec0e6b3c88276cffdbb37e1ef
 Execution time: 9.13s
```

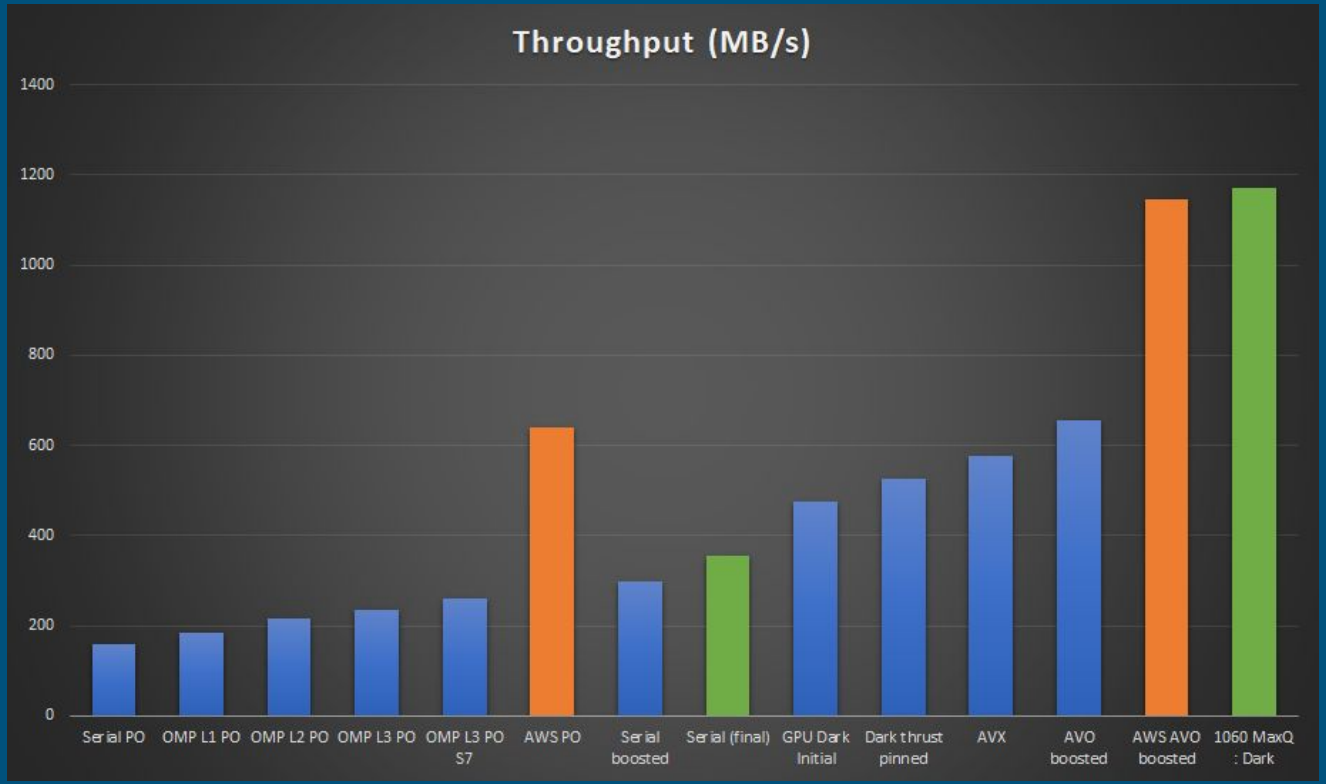
Two level nesting OMP

```
apollo_nox cuda on cpu-thread ❯ clang++ main.cpp -fopenmp
apollo_nox cuda on cpu-thread ❯ python test.py
Running test cases
35 test cases passed out of 35
apollo_nox cuda on cpu-thread ❯ python bench.py 512000000 og
File size in bytes:  512000000
8e9a08f9552b774d4c08d5735715da83755a
 Execution time: 7.70s
8e9a08f9552b774d4c08d5735715da83755a
 Original blake3 execution time: 0.
apollo_nox cuda on cpu-thread ❯ py
File size in bytes:  512000000
```



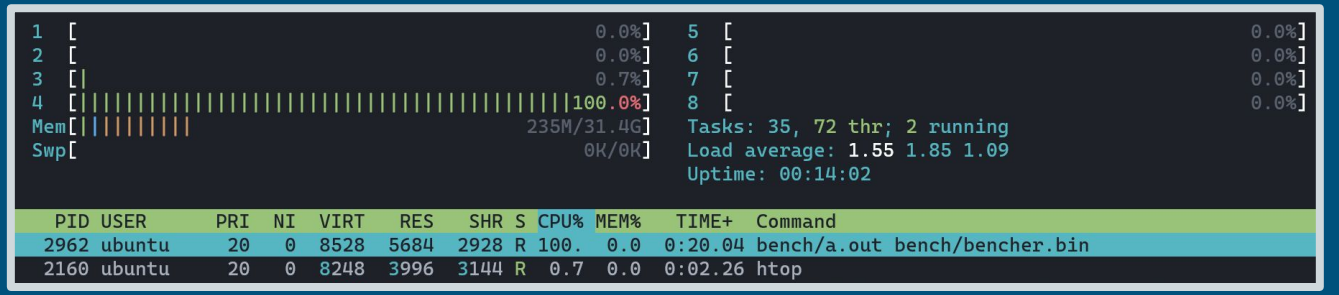
3 level nesting OMP

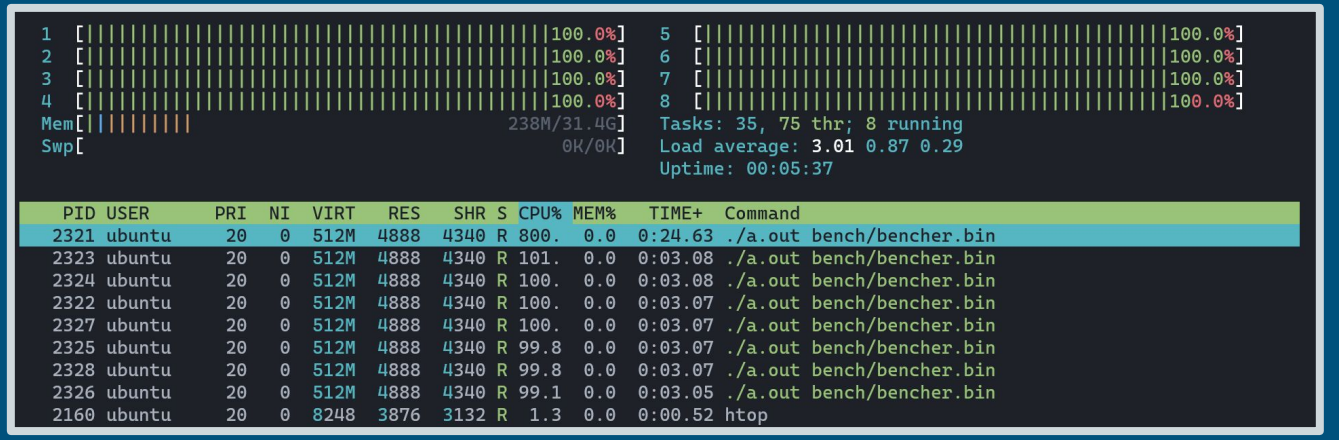
Long story short



Execution on AWS t2.2xlarge

```
ubuntu@ip-172-31-87-198:~/zipp/openmp$
File size in bytes:  4000000000
17de18ad8a9bfc16b082258537d843681d754b2
 Execution time: 3.53s
17de18ad8a9bfc16b082258537d843681d754b2
 Sequential execution time: 16.02s
17de18ad8a9bfc16b082258537d843681d754b2
 Original blake3 execution time: 0.30s
ubuntu@ip-172-31-87-198:~/zipp/openmp$
File size in bytes:  4000000000
17de18ad8a9bfc16b082258537d843681d754b2
 Execution time: 3.49s
17de18ad8a9bfc16b082258537d843681d754b2
 Sequential execution time: 15.92s
17de18ad8a9bfc16b082258537d843681d754b2
 Original blake3 execution time: 0.30s
ubuntu@ip-172-31-87-198:~/zipp/openmp$
File size in bytes:  4000000000
17de18ad8a9bfc16b082258537d843681d754b2
 Execution time: 3.45s
17de18ad8a9bfc16b082258537d843681d754b2
 Sequential execution time: 15.61s
17de18ad8a9bfc16b082258537d843681d754b2
 Original blake3 execution time: 0.30s
```





Perf analysis for CPU version

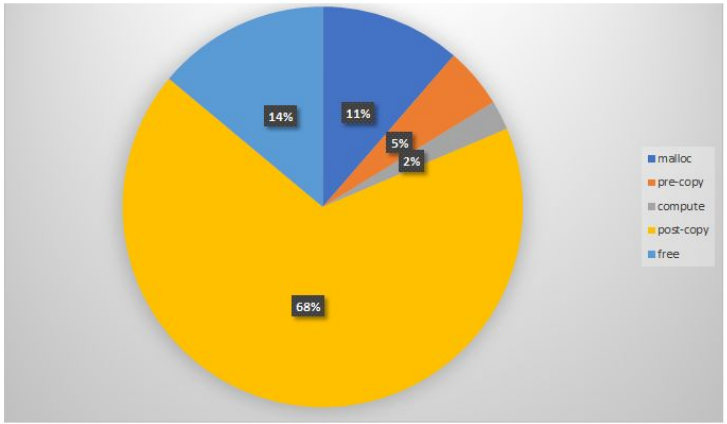
Overhead	Command	Shared Object	Symbol
71.83%	a.out	a.out	[.] compress
10.99%	a.out	a.out	[.] Chunk::compress_chunk
5.48%	a.out	[kernel.kallsyms]	[k] copy_user_enhanced_fast_string
1.47%	a.out	libc-2.31.so	[.] read
1.10%	a.out	a.out	[.] hash_many
1.04%	a.out	libstdc++.so.6.0.28	[.] std::istream::read
0.99%	a.out	libc-2.31.so	[.] 0x000000000018e8fa

Let’s move onto the GPU!

CUDA Insights

Dark hash (I)Nsight - v1

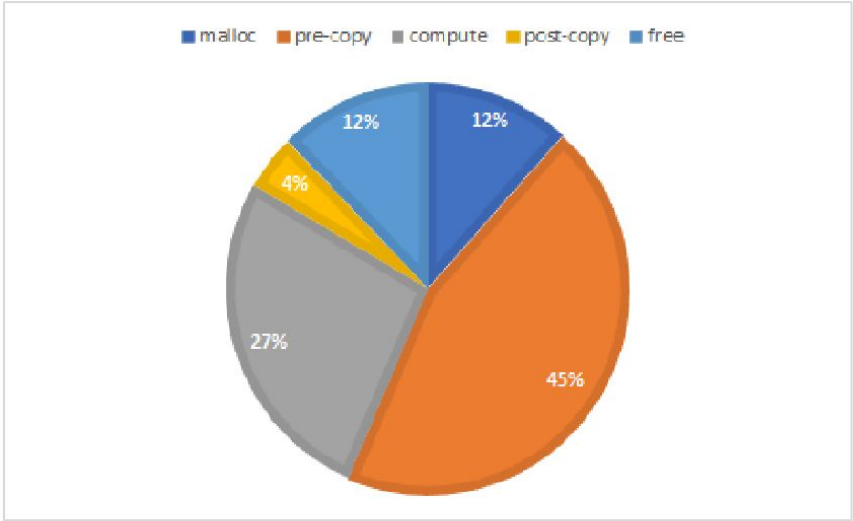
43	Function name	Timestamps	Duration	PID
44	cudaMalloc		163.240 µs	
45	cudaMemcpy		70.093 µs	
46	h_compute		14.261 µs	
47	h_compute		3.667 µs	
48	h_compute		2.430 µs	
49	h_compute		2.085 µs	
50	h_compute		2.002 µs	
51	h_compute		2.073 µs	
52	h_compute		2.005 µs	
53	h_compute		2.019 µs	
54	h_compute		1.963 µs	
55	h_compute		1.961 µs	
56	cudaMemcpy		967.043 µs	
57	cudaFree		200.419 µs	
58	cudaMalloc		185.205 µs	



CUDA Insights

Dark Hash (I)Nsight-v2

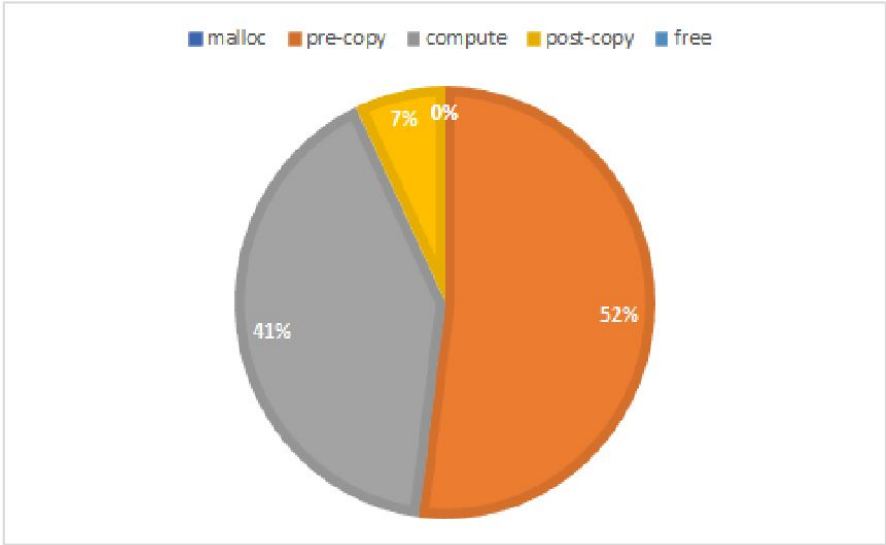
33	cudaFree	4.53855s	237.203 µs
34	cudaMalloc	4.54008s	190.663 µs
35	cudaMemcpy	4.54027s	99.956 µs
36	cudaDeviceSynchronize	4.54037s	631.416 µs
37	h_compute	4.541s	19.394 µs
38	h_compute	4.54102s	3.756 µs
39	h_compute	4.54103s	2.457 µs
40	h_compute	4.54103s	2.242 µs
41	h_compute	4.54103s	2.186 µs
42	h_compute	4.54104s	2.167 µs
43	h_compute	4.54104s	2.101 µs
44	h_compute	4.54104s	4.719 µs
45	h_compute	4.54105s	2.345 µs
46	h_compute	4.54105s	2.007 µs
47	cudaDeviceSynchronize	4.54105s	386.255 µs
48	cudaMemcpy	4.54144s	69.536 µs
49	cudaFree	4.54151s	203.042 µs
50	cudaMalloc	4.54348s	176.297 µs



CUDA Insights

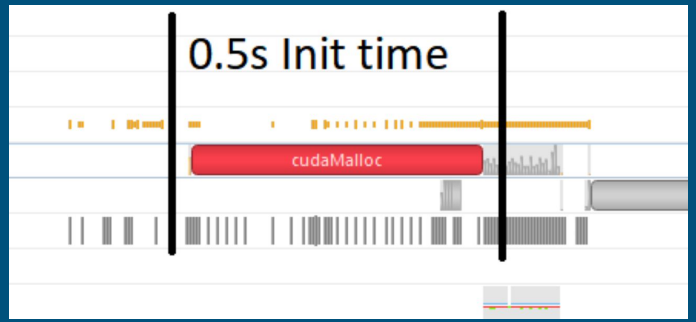
Dark hash (I)Nsight v3

86	cudaMemcpy	0.945279s	66.553 µs
87	cudaMemcpy	0.946224s	59.554 µs
88	cudaDeviceSynchronize	0.946284s	464.553 µs
89	h_compute	0.946749s	11.981 µs
90	h_compute	0.946762s	5.307 µs
91	h_compute	0.946767s	2.751 µs
92	h_compute	0.94677s	2.240 µs
93	h_compute	0.946773s	2.117 µs
94	h_compute	0.946775s	2.220 µs
95	h_compute	0.946778s	2.183 µs
96	h_compute	0.94678s	2.089 µs
97	h_compute	0.946782s	2.205 µs
98	h_compute	0.946785s	2.067 µs
99	cudaDeviceSynchronize	0.946787s	379.058 µs
100	cudaMemcpy	0.947166s	69.360 µs
101	cudaMemcpy	0.9481s	56.911 µs



CUDA Insights

```
==21456== NVPROF is profiling process 21456, command: a.exe bench/bencher.bin
4610597608768d9d795efdb7cafc5c68ab62616e4d606884cf693ff7ba56d35e
==21456== Profiling application: a.exe bench/bencher.bin
==21456== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   56.67%  746.98ms      1919  389.26us  1.9840us  408.26us  [CUDA memcpy HtoD]
                   43.25%  570.12ms     19136  29.793us  4.7360us  218.30us  h_compute(Chunk*, int, int)
                    0.08%  1.0491ms      1919     546ns     320ns     704ns  [CUDA memcpy DtoH]
      API calls:   80.75%  1.48715s      3838  387.48us  27.300us  773.40us  cudaMemcpy
                   13.82%  254.54ms         1  254.54ms  254.54ms  254.54ms  cudaMalloc
                    3.36%  61.887ms     19136  3.2340us  1.9000us  58.600us  cudaLaunchKernel
```



This one is using thrust + pinned memory, but missing a few optimizations we added later on. Ran on my GPU. My GPU(Low energy) only has a transfer rate of 3Gbps Samarth's GPU(Gaming laptop) can transfer at 12Gbps

What if we use a better GPU?

```
==22247== NVPROF is profiling process 22247, command: ./a.out bench/bencher.bin
9c8ff3575045aa272e21e4d3a512f03908abc6dbd0115dc508210f65adf2022b
==22247== Profiling application: ./a.out bench/bencher.bin
==22247== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   62.19%  626.73ms     39124  16.018us  6.1120us  637.48us  h_compute(Chunk*, int, int)
                   37.43%  377.21ms      3915  96.350us     832ns  227.08us  [CUDA memcpy HtoD]
                    0.38%  3.8643ms      3915     987ns     800ns  18.849us  [CUDA memcpy DtoH]
      API calls:   78.69%  1.00646s      7830  128.54us  9.9240us  779.35us  cudaMemcpy
                   12.01%  153.59ms         1  153.59ms  153.59ms  153.59ms  cudaMalloc
                    9.04%  115.61ms     39124  2.9550us  2.0810us  499.55us  cudaLaunchKernel
                    0.09%  1.1296ms         7  161.37us  4.9390us  607.61us  cudaFreeHost
                    0.09%  1.0922ms         7  156.03us  6.8980us  534.36us  cudaMallocHost
                    0.04%  450.64us         1  450.64us  450.64us  450.64us  cuDeviceTotalMem
                    0.03%  405.22us       101  4.0120us     395ns  180.74us  cuDeviceGetAttribute
                    0.01%  143.88us         1  143.88us  143.88us  143.88us  cudaFree
                    0.01%  79.739us         1  79.739us  79.739us  79.739us  cuDeviceGetName
                    0.00%  34.854us         1  34.854us  34.854us  34.854us  cuDeviceGetPCIBusId
                    0.00%  3.9780us         3  1.3260us     558ns  2.6800us  cuDeviceGetCount
                    0.00%  2.6630us         2  1.3310us     417ns  2.2460us  cuDeviceGet
                    0.00%  2.4340us         7     347ns     175ns     522ns  cudaGetLastError
                    0.00%     959ns         1     959ns     959ns     959ns  cuDeviceGetUuid
```

Overall results

Speed up over sequential version :

- OpenMP + AVX - 4.5
- GPU Dark - 3.5
- GPU Dynamic Parallelism -

End result: Easily 2.5x faster than the inbuilt SHA-256 hash utility on Windows.

Original BLAKE3 is 8x faster than us, but ...

Work split up

Rehan

Contribution -

- Parallel algorithm
- OpenMP
- AVX
- GPU (Dark)

Hours worked -

75

Samarth

Contribution -

- Serial code
- Parallel algorithm
- GPU (Dynamic parallelism)
- GPU Optimizations

Hours worked -

80

Thank you!