PORTABILITY FOR THE PATATRACK WITH ALPAKA

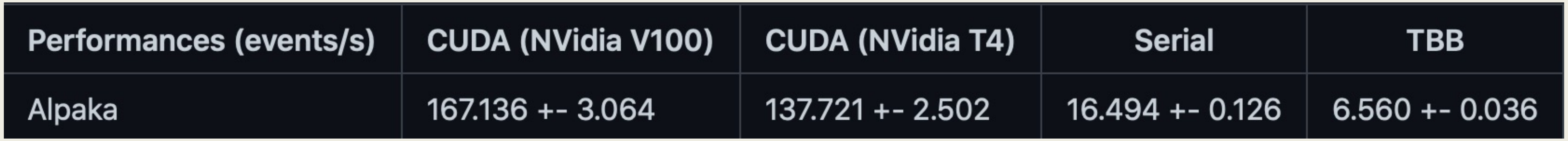
Antonio Petre

Mentors
Wahid Redjeb
Felice Pantaleo

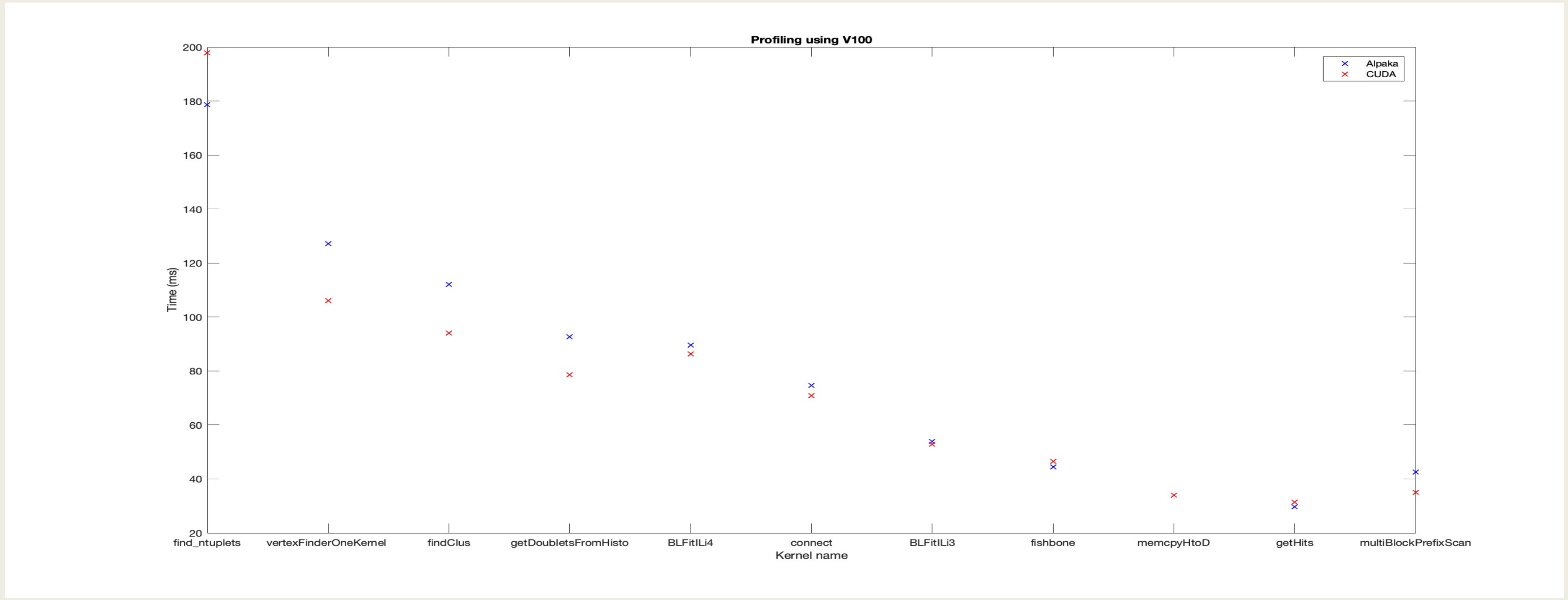
Goal & Tasks

- The goal was to port and optimize the Patatrack using the Alpaka library
- Tasks:
  - *Profiling Alpaka*
  - *Tests for atomic and barriers*
  - *Change assert in Alpaka application*
  - *Add ScopedContext*
- Full report: https://github.com/antoniopetre/pixeltrack-standalone/blob/gsoc/gsoc-documentation.md

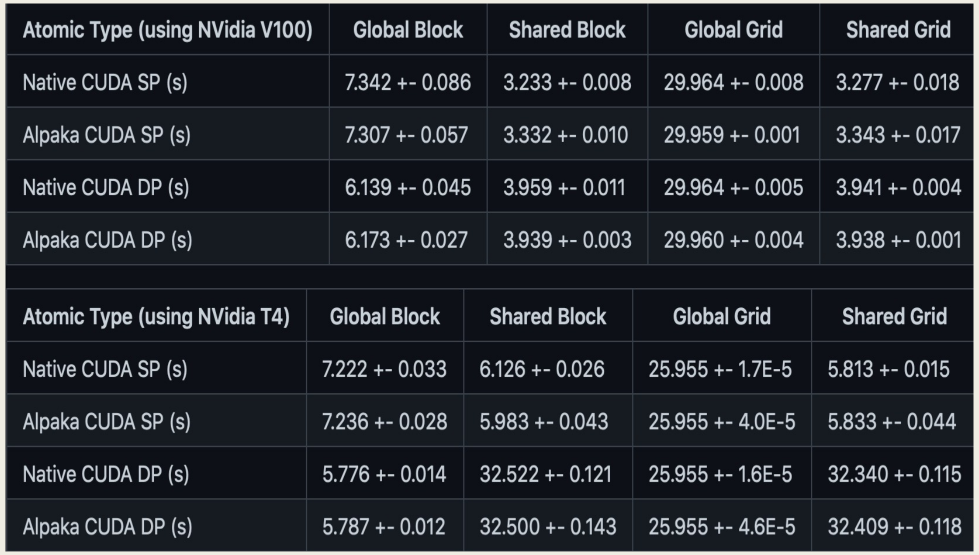
Profiling Alpaka



Performances (events/s)	CUDA (NVidia V100)	CUDA (NVidia T4)	Serial	TBB
Alpaka	167.136 +- 3.064	137.721 +- 2.502	16.494 +- 0.126	6.560 +- 0.036

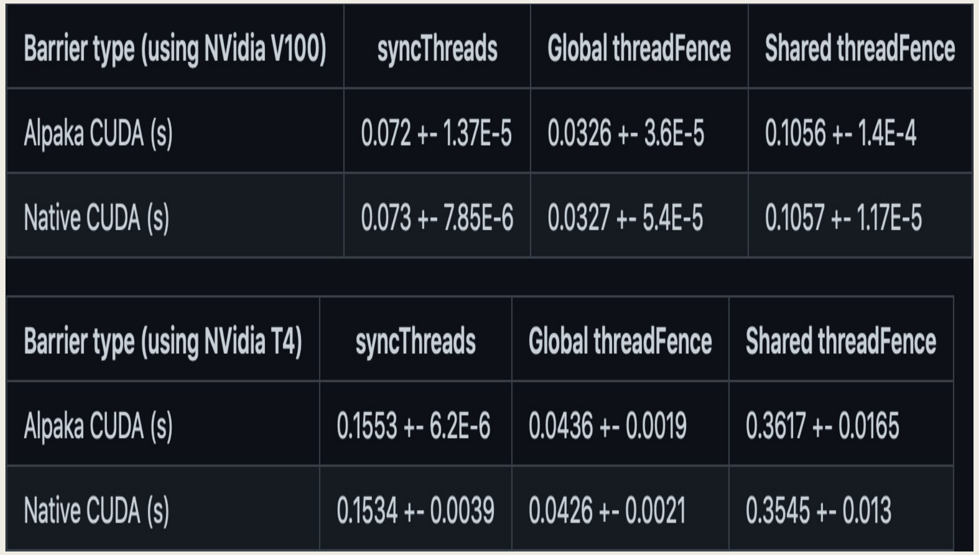


Tests for atomics and barriers



Atomic Type (using NVidia V100)	Global Block	Shared Block	Global Grid	Shared Grid
Native CUDA SP (s)	7.342 +- 0.086	3.233 +- 0.008	29.964 +- 0.008	3.277 +- 0.018
Alpaka CUDA SP (s)	7.307 +- 0.057	3.332 +- 0.010	29.959 +- 0.001	3.343 +- 0.017
Native CUDA DP (s)	6.139 +- 0.045	3.959 +- 0.011	29.964 +- 0.005	3.941 +- 0.004
Alpaka CUDA DP (s)	6.173 +- 0.027	3.939 +- 0.003	29.960 +- 0.004	3.938 +- 0.001

Atomic Type (using NVidia T4)	Global Block	Shared Block	Global Grid	Shared Grid
Native CUDA SP (s)	7.222 +- 0.033	6.126 +- 0.026	25.955 +- 1.7E-5	5.813 +- 0.015
Alpaka CUDA SP (s)	7.236 +- 0.028	5.983 +- 0.043	25.955 +- 4.0E-5	5.833 +- 0.044
Native CUDA DP (s)	5.776 +- 0.014	32.522 +- 0.121	25.955 +- 1.6E-5	32.340 +- 0.115
Alpaka CUDA DP (s)	5.787 +- 0.012	32.500 +- 0.143	25.955 +- 4.6E-5	32.409 +- 0.118



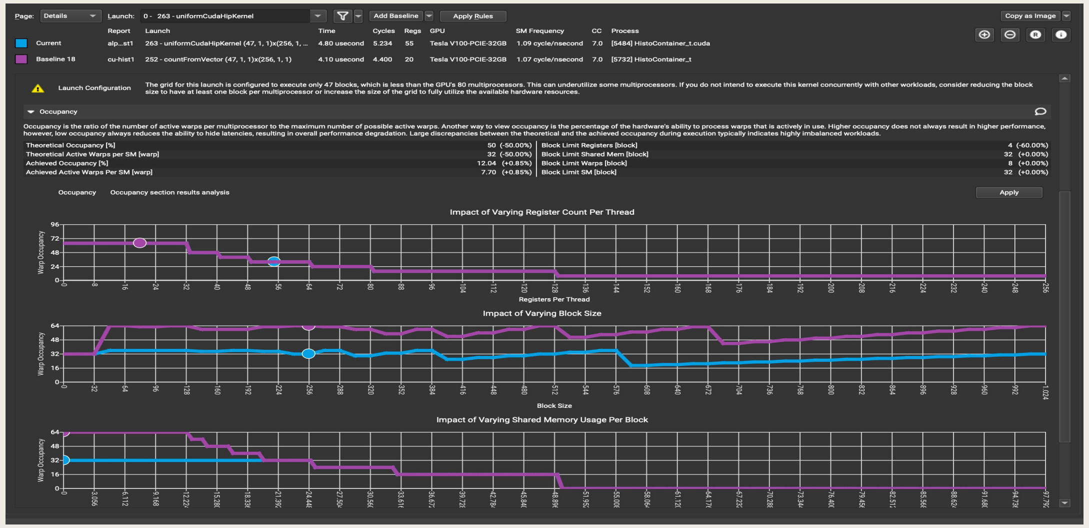
Barrier type (using NVidia V100)	syncThreads	Global threadFence	Shared threadFence
Alpaka CUDA (s)	0.072 +- 1.37E-5	0.0326 +- 3.6E-5	0.1056 +- 1.4E-4
Native CUDA (s)	0.073 +- 7.85E-6	0.0327 +- 5.4E-5	0.1057 +- 1.17E-5

Barrier type (using NVidia T4)	syncThreads	Global threadFence	Shared threadFence
Alpaka CUDA (s)	0.1553 +- 6.2E-6	0.0436 +- 0.0019	0.3617 +- 0.0165
Native CUDA (s)	0.1534 +- 0.0039	0.0426 +- 0.0021	0.3545 +- 0.013

Times are simillar

Change assert in Alpaka application

- Occupancy in the Alpaka version was lower


Page: Details
Launch: 0 - 263 - uniformCudaHipKernel
Add Baseline
Apply Rules
Copy as Image
Apply
Impact of Varying Register Count Per Thread
Warp Occupancy
Registers Per Thread
Impact of Varying Block Size
Block Size
Impact of Varying Shared Memory Usage Per Block

	Report	Launch	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
🟦 Current	alp...st1	263 - uniformCudaHipKernel (47, 1, 1)x(256, 1, ...	4.80 usecond	5.234	55	Tesla V100-PCIE-32GB	1.09 cycle/nsecond	7.0	[5484] HistoContainer_t.cuda
🟪 Baseline 18	cu-hist1	252 - countFromVector (47, 1, 1)x(256, 1, 1)	4.10 usecond	4.400	20	Tesla V100-PCIE-32GB	1.07 cycle/nsecond	7.0	[5732] HistoContainer_t

Launch Configuration: The grid for this launch is configured to execute only 47 blocks, which is less than the GPU's 80 multiprocessors. This can underutilize some multiprocessors. If you do not intend to execute this kernel concurrently with other workloads, consider reducing the block size to have at least one block per multiprocessor or increase the size of the grid to fully utilize the available hardware resources.

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	50 (-50.00%)	Block Limit Registers [block]	4 (-60.00%)
Theoretical Active Warps per SM [warp]	32 (-50.00%)	Block Limit Shared Mem [block]	32 (+0.00%)
Achieved Occupancy [%]	12.04 (+0.85%)	Block Limit Warps [block]	8 (+0.00%)
Achieved Active Warps Per SM [warp]	7.70 (+0.85%)	Block Limit SM [block]	32 (+0.00%)

Occupancy   Occupancy section results analysis

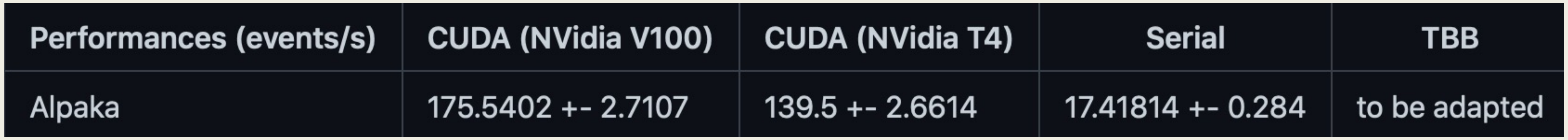
- In the CUDA version, "*assert*" is enabled by setting *GPU_DEBUG*
- I changed "*assert*" *with "ALPAKA_ASSERT_OFFLOAD" (assert is enabled only if ALPAKA_DEBUG_OFFLOAD_ASSUME_HOST* is set)

Add ScopedContext

- Difference in kernels time between Alpaka CUDA and Native CUDA ~ 0.07 seconds
- Difference for API calls time between Alpaka CUDA and Native CUDA ~ 3.60 seconds
- Example: 8004 streams created in Alpaka CUDA, only 2 streams created in Native CUDA
- Port the stream/event workflow from the Native CUDA to the Alpaka version
- Possible problem: some helper methods must be specialized for each accelerator

Problems & Final results

- Two significant problems:
  - TBB version doesn't work with the new stream/event logic (the validation tests fail + after asynchronous copy was added => Segmentation Fault)
  - The current version which works creates a new stream for every event (like the legacy version), but it uses the same workflow as the Native CUDA implementation. The Native CUDA version uses a reusable object, but this doesn't work in Alpaka (runtime errors). Important speedup can be obtained after these errors will be solved.
- Final results:



Performances (events/s)	CUDA (NVidia V100)	CUDA (NVidia T4)	Serial	TBB
Alpaka	175.5402 +- 2.7107	139.5 +- 2.6614	17.41814 +- 0.284	to be adapted

THANK YOU!