Science and Technology Facilities Council

Pattern Matching: Migrating CUDA to oneAPI on Intel FPGA

Abdeslem Djaoui
Dmitry Emeliyanov & John Baines

Agenda

1 What is oneAPI

2 First target: A functionally correct version running on the FPGA

3 Optimizations landscape

4 Further Parallelization steps and techniques

5 Results so far: How does it compare to GPU

oneAPI in a nutshell

- Framework for programming accelerator devices based on SYCL (C++), plus Data Parallel (DPC++) additions
- Higher abstraction than HLS and OpenCL (“easier to program!”)
- Multiple device selectors target different devices
  - *cpu_selector, gpu_selector, accelerator_selector*
  - *fpga_emulator_selector, fpga_selector*
- Portability varies depending on device
- SYCL *queues* for offloading work from host to device
  - Similar to CUDA *Streams*
  - Used for both kernels and memory copy operations
  - After submission, host continues to execute, while device, eventually (asynchronously) does the work, unless *wait()* is specified.

oneAPI: Data Management

- 2 Options for data creation/movement between host and device
- Unified Shared Memory USM
  - Pointer based approach (like C++)
  - Both explicit and implicit data movement depending on allocation type (*device, host or shared)*
- Buffers and accessors (SYCL)
  - Buffers can be created and initialized with data residing on the host
  - Runtime performs implicit data movement from host to device
  - Buffers (not initial data) accessed from device and host using accessor objects
  - Different concept from USM
  - Can be mixed with USM if needed

oneAPI Kernels

- 2 types of kernels
- ND-range kernels (similar to CUDA kernels)
  - Suitable for porting CUDA to oneAPI on GPU
  - Tools available for easy porting of CUDA code to oneAPI ND-range on GPUs
  - Not easily optimized on FPGA
- Single Task kernels (FPGA specific)
  - Looks sequential in nature (like legacy code)
  - oneAPI compiler automatically tries to pipeline iterations in loops
  - Many options for optimization available
  - No tools available for easy porting of CUDA to single task kernels

Agenda

1 What is oneAPI

2 First target: A functionally correct version running on the FPGA

First Kernel Version: inspired by CUDA

- Host side code
  - USM for memory management
    - Explicit copy of data from host to device
    - Explicit copy of results from device to host
  - A Kernel loop for each event, on the host {
- On the device:
  - Single task kernel
  - Read event data from global to local memory (on chip)
  - Loop for *pattern blocks {*
    - Nested loop for *patterns for each block {*
      - Nested loop for pattern matching *accumulator count* { }}}
- Typical development flow:
  - Emulation (seconds) for functional correctness
  - report generation (minutes) for details about issues hindering the creation of an effective pipeline
  - generate hardware bitstream for programming FPGA (hours) and run it
  - Profiling it (also requires a separate hardware generation bitstream)

First version and results

- Some necessary restructuring
  - Error: Atomics not supported in systems with more that one global memory (DDR4 and USM)
  - Move some load/stores to local memory
  - Only update global memory after pipelined loop exists
    - One load operation per clock cycle, no need for Atomics
- Result: ~1000x slower than GPU
  - Compiler tries and fails to build an efficient pipeline
  - Outside blocks loop on the device executed serially
  - Memory dependencies in nested loops
- Low Fmax: 242MHz (Maximum 480MHz)
- But functionally correct
- Verdict: Code is fairly portable, not the performance.

Agenda

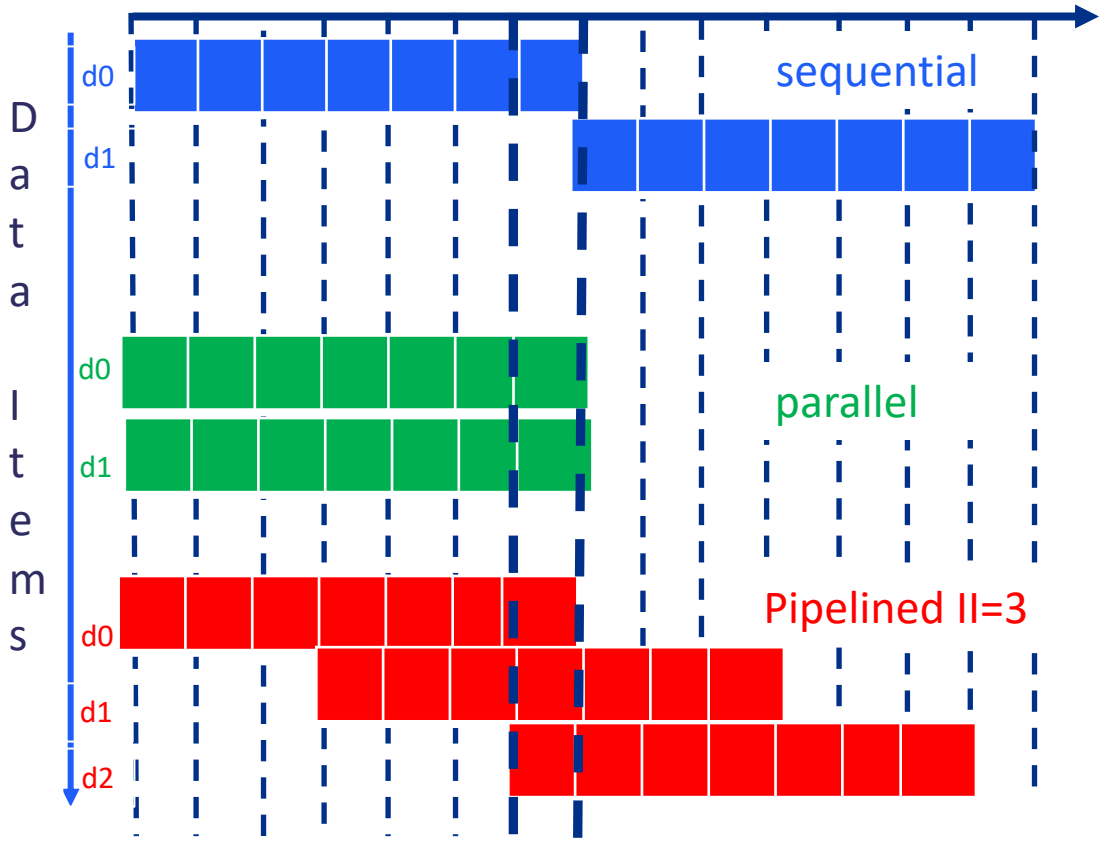
1 What is oneAPI

2 First target: Functionally correct version running on the FPGA

3 Optimizations landscape

Available FPGA execution types

- Sequential: one task at a time
- Parallel: Multiple tasks or operations executed simultaneously
  - GPU/CPU have pre-configured hardware (cores) for this
  - FPGA compiler automatically executes all independent operations on the FPGA spacial area in parallel (limited without modifications)
- Pipelined (FPGA specific):
  - Operations in a task split into multiple stages
  - Feed forward of data items
  - Multiple items running in parallel in different pipeline stages (~ Latency/Initiation Interval)
- Mix the types on the FPGA as needed
  - Implement parallelism similar to GPU/CPU (generally not too scalable)
  - Or go beyond what exists on GPU/CPU and devise own custom hardware that suits your application



First “efficient” pipeline

- Use buffers for both event and pattern data
  - Uploaded to device automatically before execution starts
- Use USM for results
  - Downloaded to host automatically by kernel
- A single loop over all patterns (merge blocks and patterns loops)
- Further modifications to lower the II
  - Unroll nested pattern matching count loop (removes some data dependencies)
  - Reduce global memory loads by coalescing loads
- II=1 achieved (best for throughput)
- Resulting Kernel code already quite different from initial CUDA code
- results:
  - Performance: 40 x slower than GPU
  - Will need 40 x parallelism or more for similar performance
  - Most used resource: RAM blocks: 8% (out of 68% available)
- Some benefit of major restructuring the code:
  - An obscure bug was discovered in the CUDA version

Agenda

1 What is oneAPI

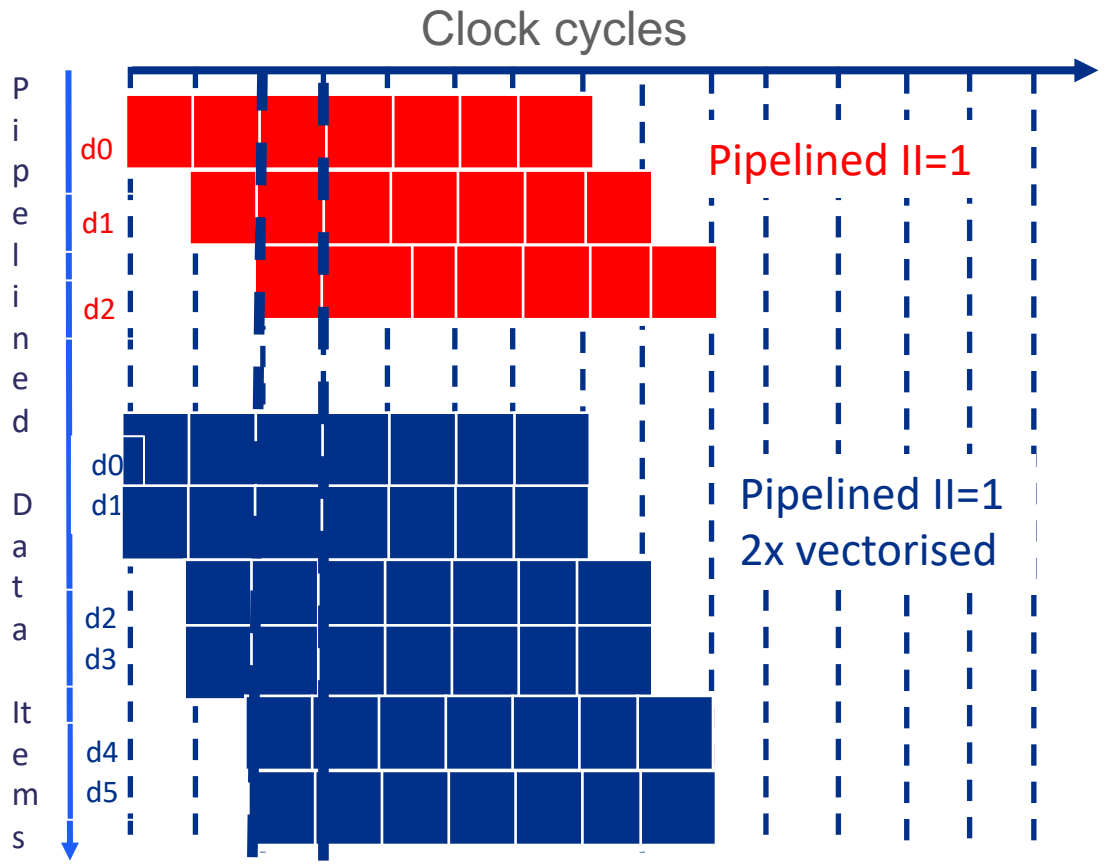
2 First target: A functionally correct version running on the FPGA

3 Optimizations landscape

4 Further Parallelization steps and techniques

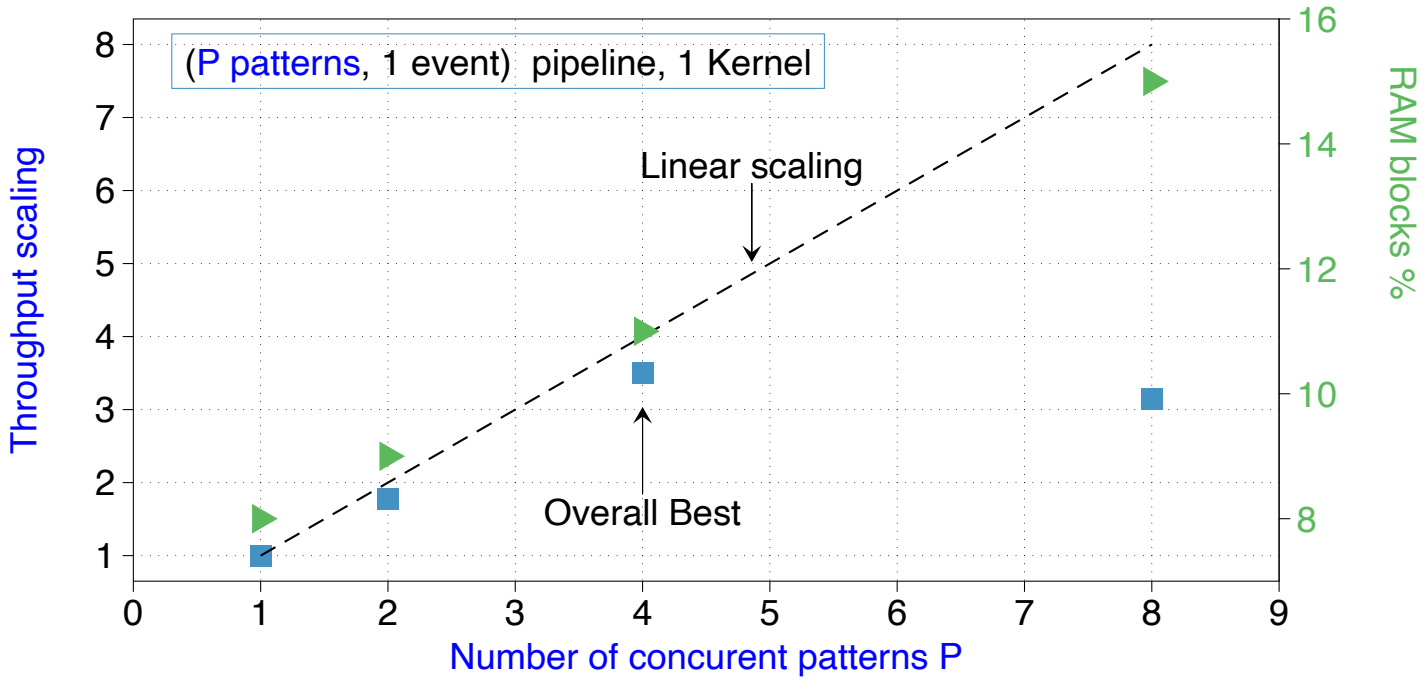
4.1 Parallelism: 1D Vectorization execution

- Type of parallel execution where operations performed on multiple data-items simultaneously for each event e
  - For 2 items d0 and d1:
    - 2 x Parallel operations on 2 data-items at different pipeline stages in each clock cycle
    - Implemented in hardware as a deeper pipeline with 2 parts: (d0, e) – (d1-e)
- Similar to SIMD on CPU or warps on GPU (best analogy)
- Difference is:
  - Hardware for CPU/GPU already exists
  - For FPGA, need to help compiler build required hardware by structuring code accordingly
- Cost: Resources needed for implementing the deeper pipeline



Pattern data 1D vectorization steps and results

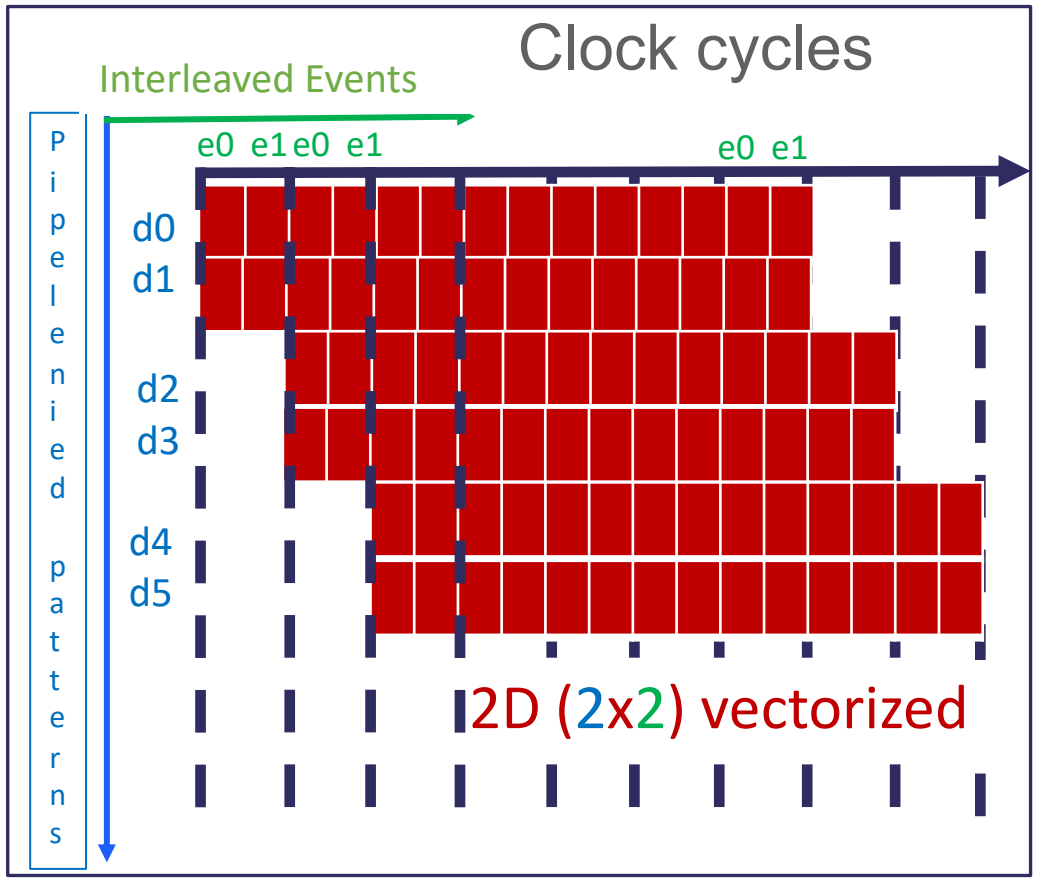
- Coalesce loads from global memory for a vector of P of (2, 4, 8) patterns
- Trip count for pipelined pattern loop is reduces by P
  - Improving throughput by P compared to 1 pattern
- On-Chip memory usage increases ~ linearly with P
- Scalability only limited by
  - Potential pipeline stalls, waiting for Global memory loads to finish
- Vector of 4 Patterns selected for next steps



Science and Technology Facilities Council

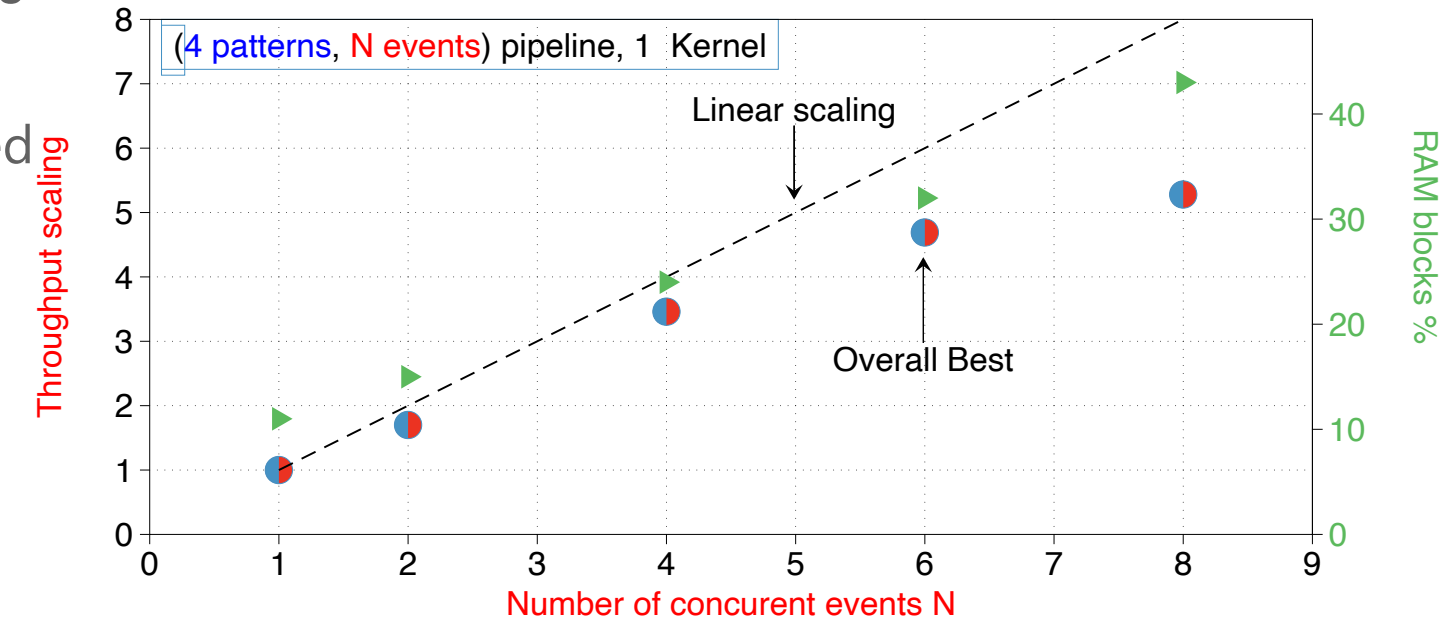
4.2 2D vectorization (patterns and events)

- Interleave loop event iterations from host into patterns loop pipeline on FPGA
  - Higher-level granularity data (events) processed in parallel with lower-level data (patterns)
  - For 2 events (e0,e1) and 2 patterns (do,d1):
    - Deeper pipeline constructed with 4 parts: (d0, e0)-(d0,e1)-(d1,e0)-(d1,e1)
    - All operation performed simultaneously in each clock cycle in different parts of pipeline
- This results in a hardware custom made 2D vectorization unit
  - No equivalent exists on CPU or GPU
- Most problems targeting GPUs can be thought of as operating on 2 levels of data granularity one on host and one on device
  - Could benefit from 2D vectorization on FPGAs to increase throughput



2D vectorization and results for N events

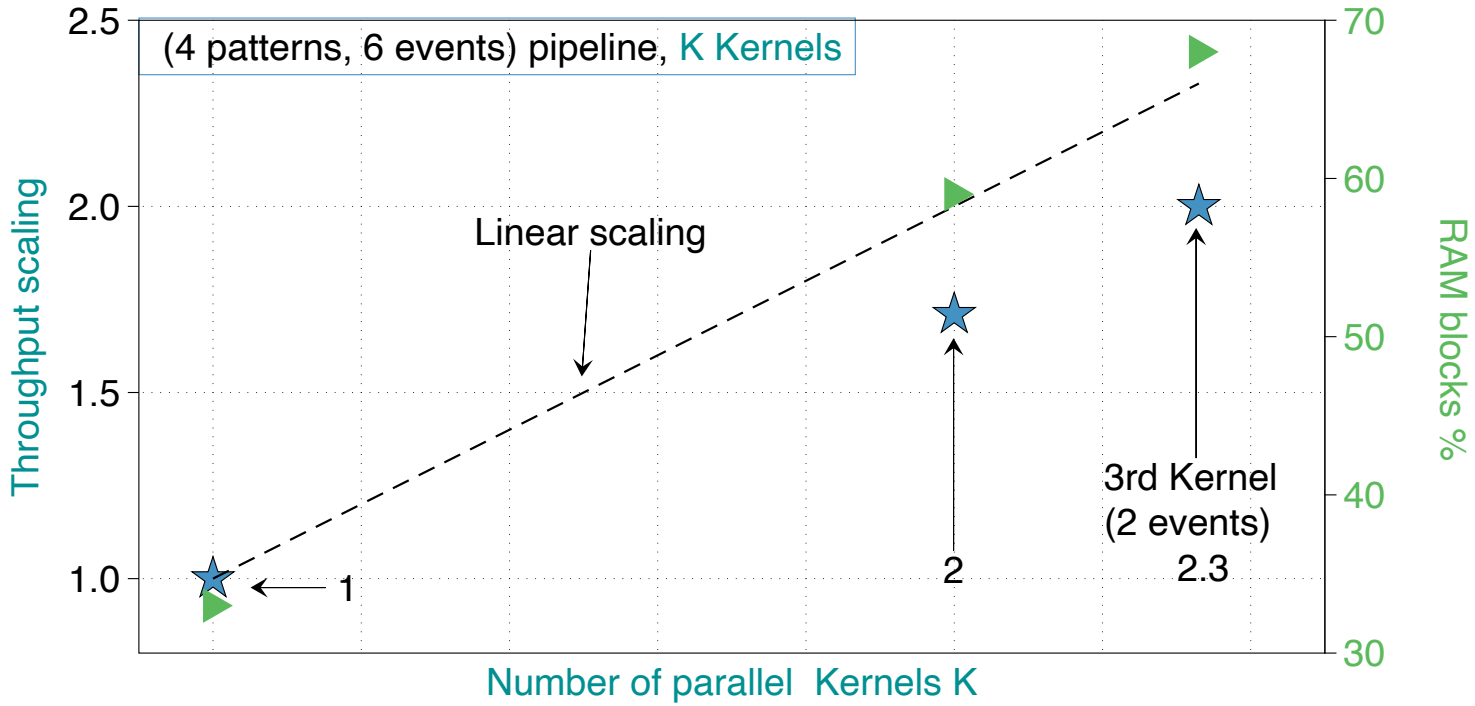
- Advantages of 2D vectorization:
  - Overhead of loading data from global memory shared by N events
  - Compacting pattern data results in only 1 load operation
  - This results in reduction of required accesses to Global memory
  - Overhead of Kernel launch also shared by N events
- N events x P patterns processed simultaneously in each cycle (4p x 6e=24)
- Good scaling compared to (4 patterns and 1 event)
  - Limited by:
    - increase in amount of data read before pipeline execution
    - Available resources (RAM blocks) on the FPGA



Science and Technology Facilities Council

4.3 Multiple Kernels and results

- Submit multiple task Kernels in parallel (already 2D vectorized)
  - "*Similar*" to Streaming Multiprocessors on GPU
- Good scaling, but limited by:
  - Total embedded memory usage reaching 100%
  - 3rd kernel has only 2 events
  - SYCL Queue overhead of launching kernels increases with the number of kernels
- In total 56-way parallelism achieved compared to initial pipeline (40 required):
  (2k x (4p x 6e)+1k x 4p x 2e)



Agenda

1 What is oneAPI

2 First target: A functionally correct version running on the FPGA

3 Optimizations landscape

4 Further Parallelization steps and techniques

5 Results so far: How does it compare to GPU

FPGA versus GPU

- Similar throughput
- Pattern Matching is a Memory Bandwidth intensive application
  - GPU has a huge advantage in this regard
- To summarise:
  - FPGA processes 14 events in 3 parallel kernels
  - GPU: 1 event, 1 kernel at a time
  - FPGA (with DDR4) results within 12% of FPGA (with HMB2)
  - 2D vectorization is critical in increasing throughput
  - Porting kernel CODE from GPU to FPGA requires extensive changes
  - Final Verdict: Performance is portable, not the code.

	FPGA	GPU
Device	Intel PAC D5005 Stratix 10 GX	NVIDIA TITAN V
Performance (for Pileup events)	**68 µs/event**	**59 µs/event Includes event data transfer**
Frequency	305 MHz	1.2GHz (1.5GHz boost)
Global memory	DDR4 2400	HBM2
Memory bandwidth (theoretical)	38.4 GB/s	653 GB/s
Very different architecture: Difficult to compare like with like	**56-way mixed parallelism: 3 kernels (2D vectorized) executed in parallel**	**80 streaming multiprocessors: 64 cores each, 5120 CUDA cores, Warps of 32 threads executed in "parallel"**