IWOCL 2024

The 12th International Workshop on OpenCL and SYCL

Optimisation and Evaluation of Breadth First Search with oneAPI/SYCL on Intel FPGAs: from Describing Algorithms to Describing Architectures

Kaan Olgu (University of Bristol)

Tobias Kenter (Paderborn University), Jose Nunez-Yanez (Linkoping University), Simon Mcintosh-Smith (University of Bristol)

We all have a mutual connection with Pep Guardiola!

For my case :

No, I am not a professional footballer

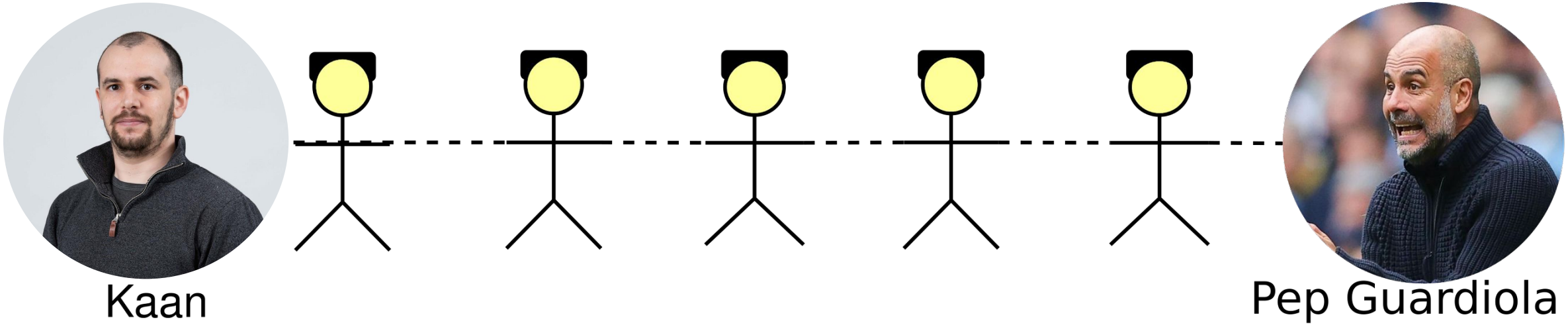
Yes, we are bald, but that’s not the only connection!

Kaan Olgu

Pep Guardiola

Six Degrees of Separation

All people are 6 or less connections away from each other!



Problem

Can we scale this up to real-life networks ?

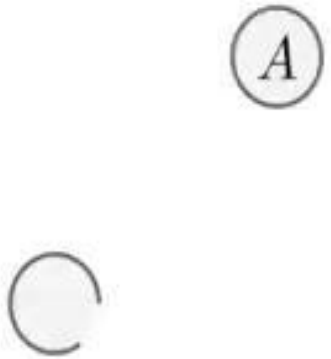
Agenda

1. **Introduction to FPGAs**
2. **Introduction to BFS**
3. **memoryBFS**
4. **streamingBFS**
5. **Performance Evaluation**
6. **Summary & Conclusion**

Introduction to FPGAs

- FPGA – Field Programmable Gate Arrays
- Two major players in the FPGA domain – AMD (prev. Xilinx) and Altera (part of Intel)
- Used to code with HDL (Hardware Description Languages) – (e.g. VHDL,Verilog)
- Offers:
  - massive parallelism – for each application, operations are customized and fixed to one location on the FPGA, then data flows through them
  - flexibility of reprogramming – it is possible to modify the architecture of the design according to the needs (e.g. adding more compute units, upgrading the design)
- Our aim is to explore with Intel oneAPI/SYCL to :
  - Achieve better performance – will be discussed in detail
  - Increase reproducibility and productivity

Introduction to Breadth First Search Algorithm

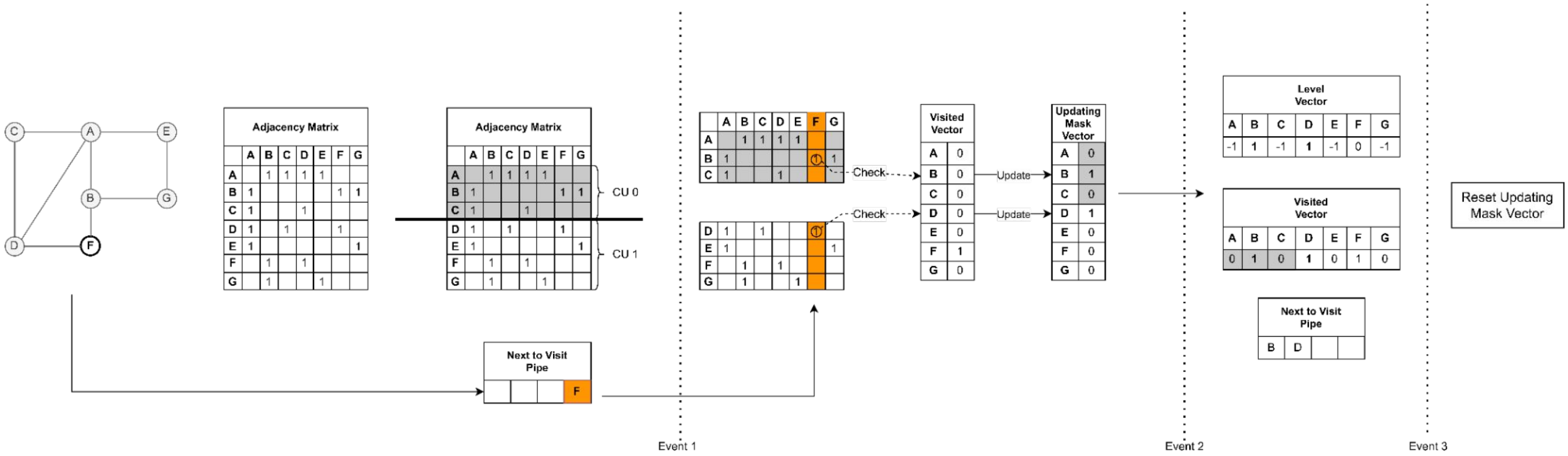


Introduction to Breadth First Search Algorithm

- **Why we need BFS?**
  - Essential for solving various real-world problems
  - Examples : Route planning in GPS navigation systems, social media, P2P networks.
  - **Performance Bottlenecks:**
    - Depends on irregular memory accesses rather than computation intensity!
    - Huge dataset sizes (trillions of edges)
    - Next to visit node is decided during the execution
    - Overall for BFS – Computing is cheap, but moving data is expensive!

Overview of execution with memoryBFS and streamingBFS

- Conceptual explanation from the video still holds!



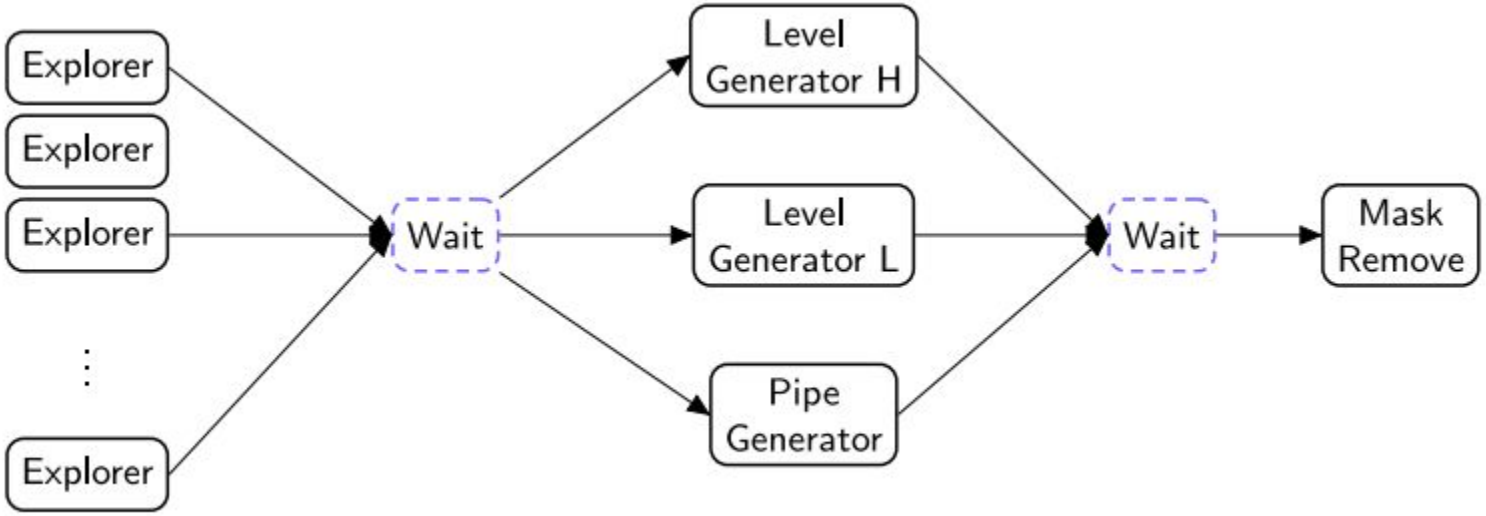
Our Approaches - memoryBFS

Highlights:

- Uses off-chip DDR memory to share data between kernels (USM programming model)
- Leverages automatic cache memory to mitigate random memory accesses
- Coarse Grained Parallelism
- Coalesced Writeback Support
- Designed following the Intel oneAPI guidance

memoryBFS

- 4 kernels:
  - parallel explorer: perform dot product operation (works on same shared buffers)
  - parallel levelgen: update the level vector (works on same shared buffers)
  - pipegen: generate the list of newly visited nodes
  - mask remove: prepare the binary vectors for the next stage
- Load balancing is important!



Overview of Organisation

- **Main Memory :**
  - Stores graph data read from binary files in CSR format
- **CPU :**
  - Initializing the timer
  - Loading graphs from main memory
  - Assign graph partitions to each compute unit
  - Bridge between main memory and FPGA
- **FPGA :**
  - Calculations and data requests
  - Where all the magic happens

- **Python Helper Script**
  - Converts .txt graph datasets to .bin format.
    - .bin files reduces the storage space ~10x
  - Partitions the graph into smaller subgraphs based on user-specified compute units.
  - Partitioning options:
    - Horizontal split based on the number of non-zeros (edge split).
    - Split by the number of rows (node split) within the adjacency matrix.

CSR format

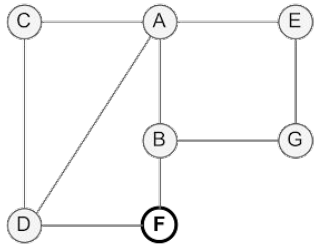
- The Compressed Sparse Row/Column
- Stores the graph data in 2 vectors
- Index pointers indicates the nth position at the indices, difference between (n+1) and (n) th elements is the total number of neighbours
- Indices show the neighboring elements

Edge Split vs Node Split, which one is better ?

- It varies!
- Table on Right, Showcases disparities in partitions, with node or edge counts differing by over 2x.
- Choosing row vs. edge split depends on whether node processing or edge processing drives execution time.
- For our data and design, edge split proves to be better

	P	Edge Split Nodes (Rows)	Edge Split Edges (# of nnz)	Row Split Nodes (Rows)	Row Split Edges (# of NNZ)
R19-32	0	86,344	4,193,964	130,112	5,012,055
	1	112,872	4,193,714	130,112	4,200,974
	2	98,328	4,194,006	130,112	5,505,490
	3	222,652	4,193,886	129,860	2,057,051
		520,196	*16,745,479*	*520,196*	*16,745,479*
R21-32	0	246,016	11,177,708	342,080	17,204,973
	1	281,000	11,179,568	342,080	14,102,567
	2	262,648	11,178,334	342,080	5,947,637
	3	295,168	11,179,021	342,080	19,287,164
	4	258,928	11,179,313	342,080	7,560,258
	5	708,340	11,178,018	341,700	2,969,363
		2,052,100	*66,937,357*	*2,052,100*	*66,937,357*

- Edge Split Example :



Adjacency Matrix

	A	B	C	D	E	F	G
A		1	1	1	1		
B	1					1	1
C	1			1			
D	1		1			1	
E	1						1
F		1		1			
G		1			1		

Adjacency Matrix

	A	B	C	D	E	F	G	
A		1	1	1	1			CU 0
B	1					1	1	
C	1			1				
D	1		1			1		CU 1
E	1						1	
F		1		1				
G		1			1			

Automatic Caching in memoryBFS

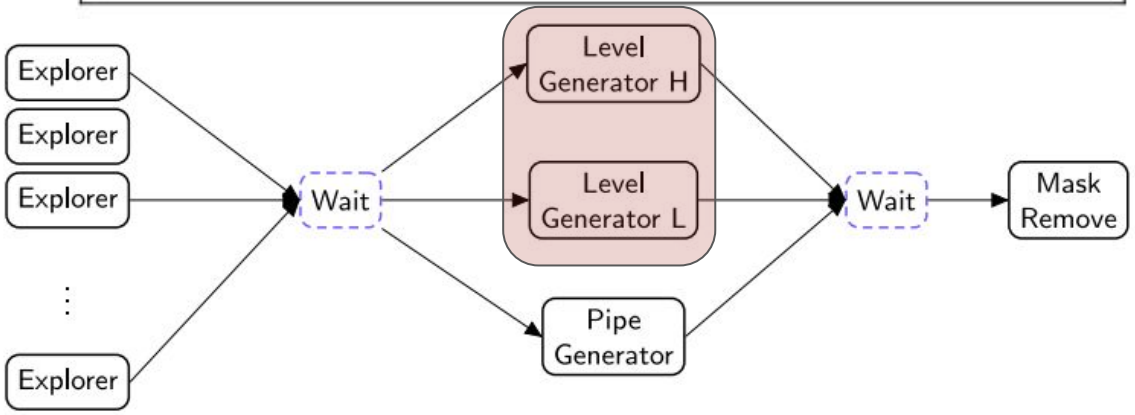
- Automatic Cache (1MB) - Experimented other sizes too this is sweet spot!
  - using CacheLSU = ext::intel::lsu<ext::intel::burst_coalesce<true>, ext::intel::cache<1024*1024>,ext::intel::statically_coalesce<false>>;

```
1         // define global work size for parallel_for function
2         range<1> gws (no_of_nodes_inside_pipe);
3         auto e = q.parallel_for<ExploreNeighbours<krnl_id>>(gws, [=] (id<1> iter)
4         [[intel::kernel_args_restrict]]  {
5           device_ptr<unsigned int> DevicePtr_start(usm_nodes_start+offset);
6           device_ptr<unsigned int> DevicePtr_end(usm_nodes_start + 1+offset);
7           device_ptr<unsigned int> DevicePtr_edges(usm_edges+offset_inds);
8           device_ptr<MyUint1> DevicePtr_visited(usm_visited);
9             // Read from the pipe
10            unsigned int idx = usm_pipe[iter];
11            // Process the current node in tiles
12            unsigned int nodes_start = DevicePtr_start[idx];
13            unsigned int nodes_end = DevicePtr_end[idx];
14            // Process the edges of the current nodes
15            for (int j = nodes_start; j < nodes_end; j++) {
16              int id = CacheLSU::load(DevicePtr_edges + j);
17              MyUint1 visited_condition = CacheLSU::load(DevicePtr_visited + id);
18              if (!visited_condition) usm_updating_mask[id]=1;
19            }
```

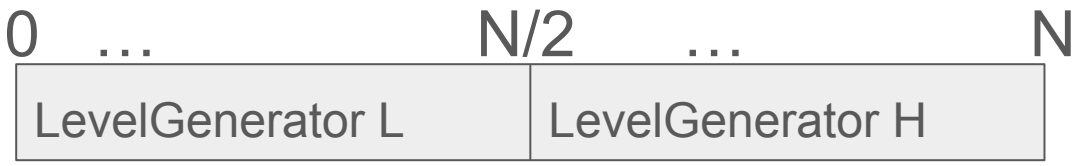
Coarse Grained Parallelism in memoryBFS

- LevelGen - 2 Kernels writing to same USM but different portions

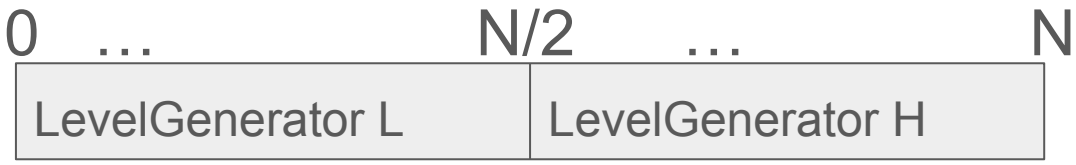
```
1   auto e =q.single_task<class LevelGenerator<
        krnl_id>>( [=]() [[intel::
        kernel_args_restrict]] {
2
3     #pragma unroll 8
4     [[intel::initiation_interval(1)]]
5     for(int tid =no_of_nodes_start; tid <
          no_of_nodes_end; tid++){
6       unsigned int condition = usm_updating_mask[
            tid];
7       if(condition){
8         usm_dist[tid] = global_level;
9         usm_visited[tid]=1;
10      }
11    }
12
13  });
```



usm_dist :



usm_visited :



Coalesced Writeback in memoryBFS

- Writeback stage processes data from nodes, filters and stores specific elements in a temporary buffer temp, and then transfers these elements to an output buffer usm_pipe in chunks to optimize memory access and processing efficiency.
- II becomes 1 and the performance is higher compared to writing back to memory in each iteration

Key Concepts:

- Buffering: Temporarily stores data to manage and optimize data flow, especially in hardware where I/O operations may be costly or need to be minimized.
- Data Streaming: Processed data is moved out in chunks, enabling efficient use of resources and ensuring continuous data processing and output.

Our Approaches - streamingBFS

memoryBFS Key Limitations:

- Scalability issues (limited with 4CU)
- Global memory access bandwidth issues, this was the main target to improve
- Performance results will be discussed later

streamingBFS - Complete Revamp to memoryBFS:

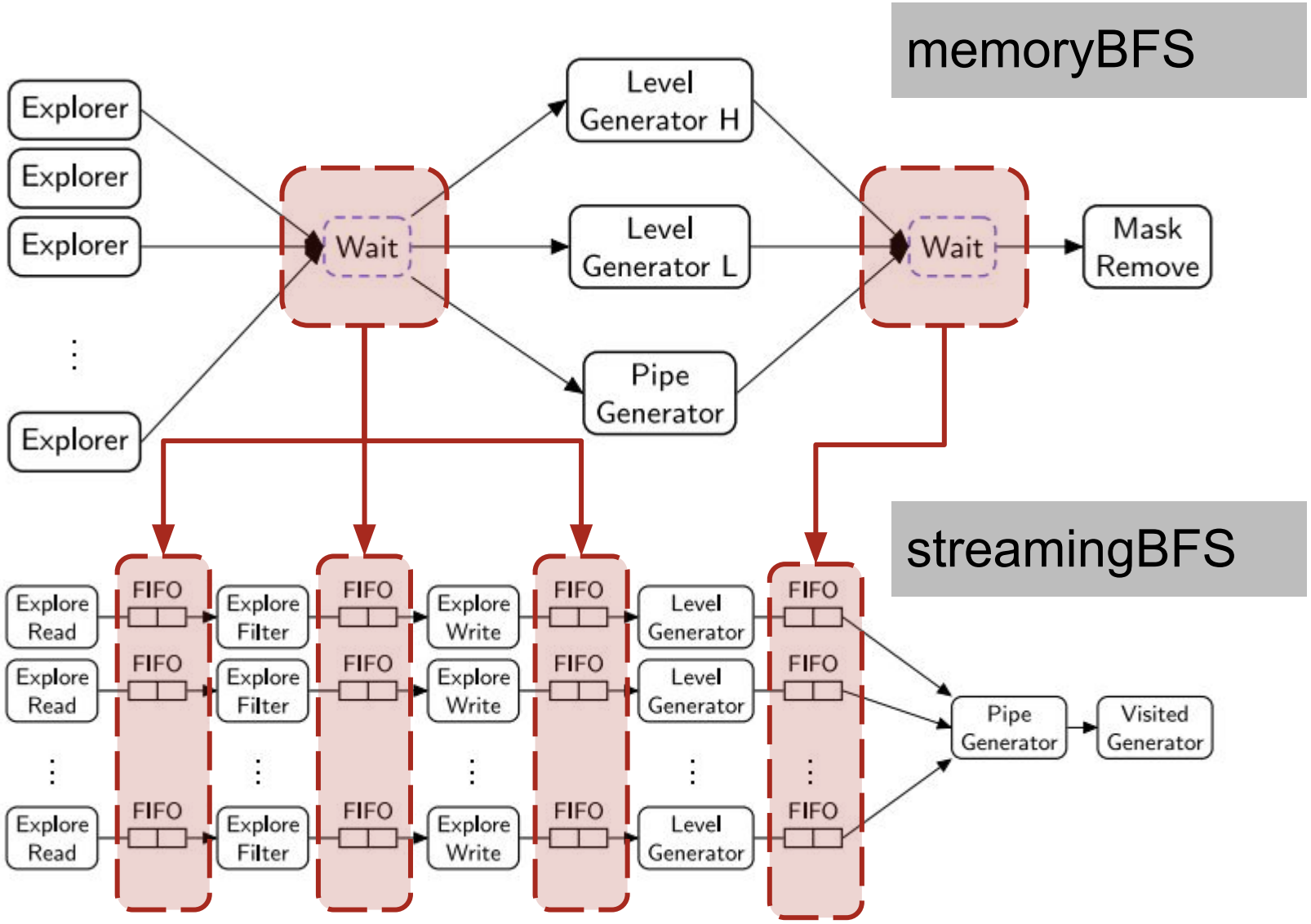
- Dataflow Execution Model - Uses pipes to stream data between kernels
- New Kernel Designs with Additions - Ensures synchronisation with minimal latency
- Compression of Data to Bytes and Bit Manipulations
- Replaces Some Off-chip Memory Accesses with On-chip Data Movement
- Modular design – modifying # CUs, FIFO and cache sizes from CMake variables

memoryBFS Recap

- Uses Unified Shared Memory (USM) to share data between kernels
- Leverages automatic cache memory to mitigate random memory accesses
- Coarse Grained Parallelism
- Coalesced Writeback Support
- Designed following the Intel oneAPI guidance

Dataflow Execution Model in streamingBFS

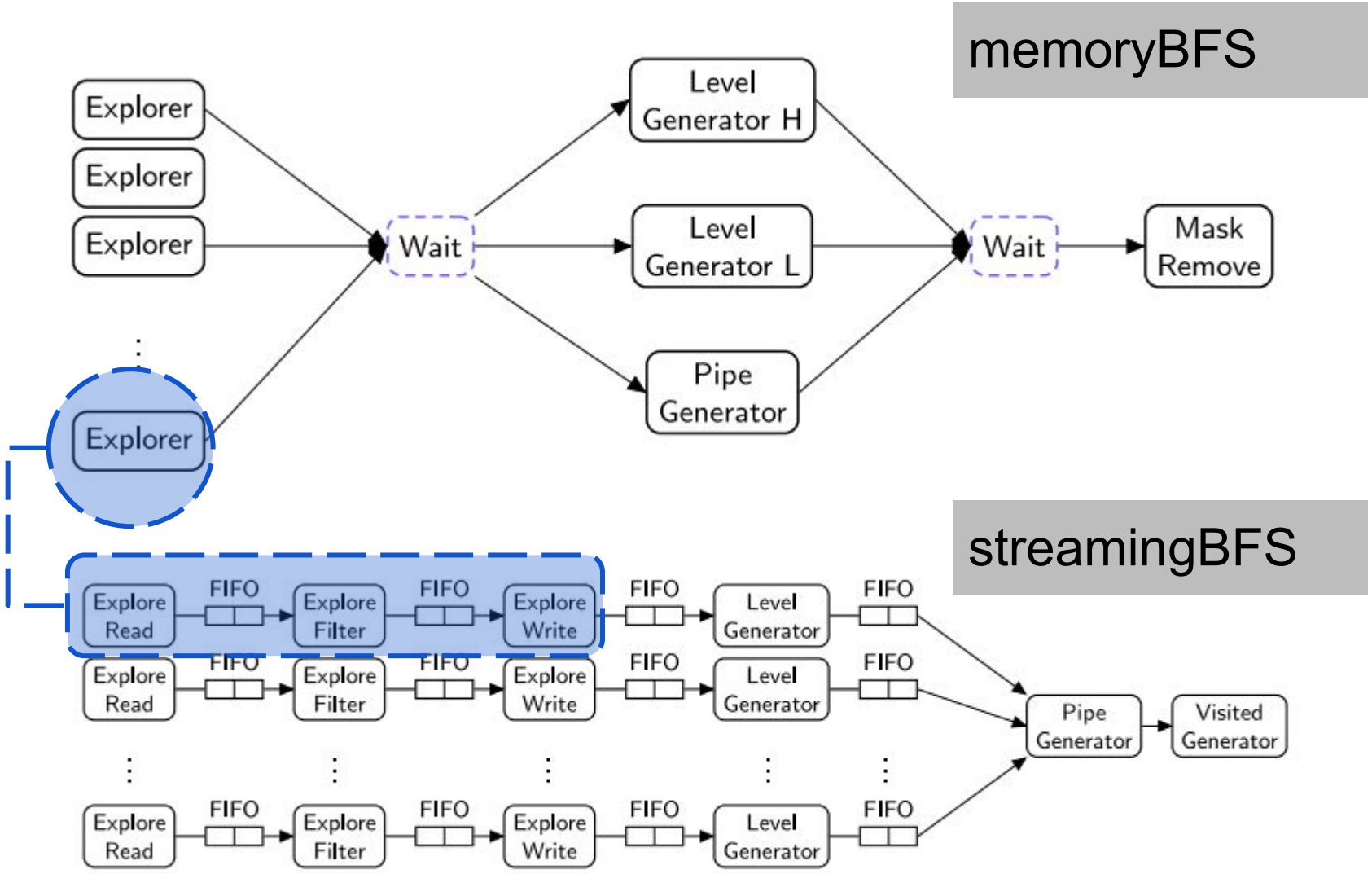
- Dataflow execution model employs pipes for efficient data exchange between kernels
- The synchronisation of kernels is achieved via pipes so q.wait() commands are no longer required
- Improves performance drastically



Splitting Large Kernels in streamingBFS

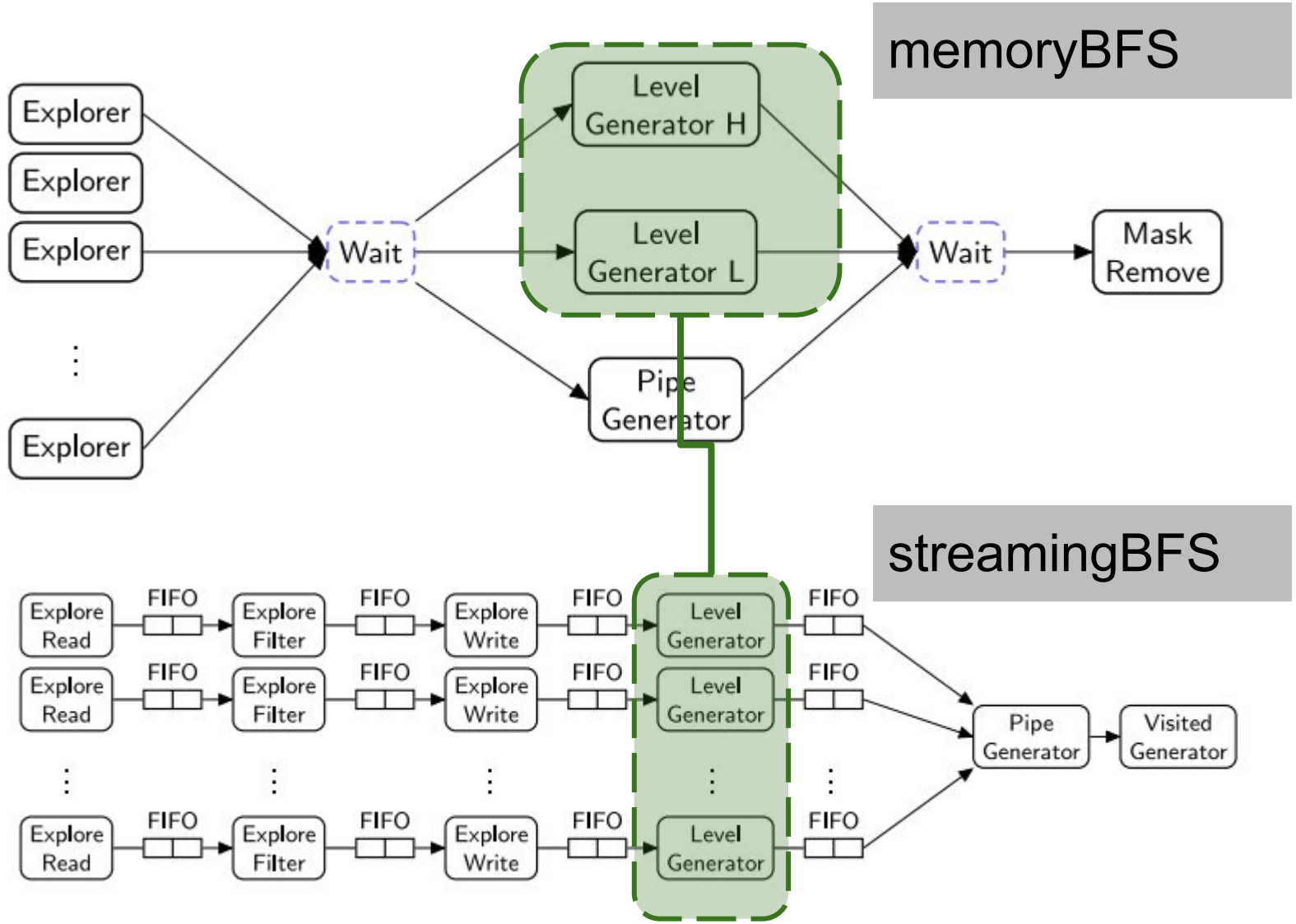
The explorer kernel is split into 3 FIFO kernels:

- Explore Read
- Explore Filter
- Explore Write



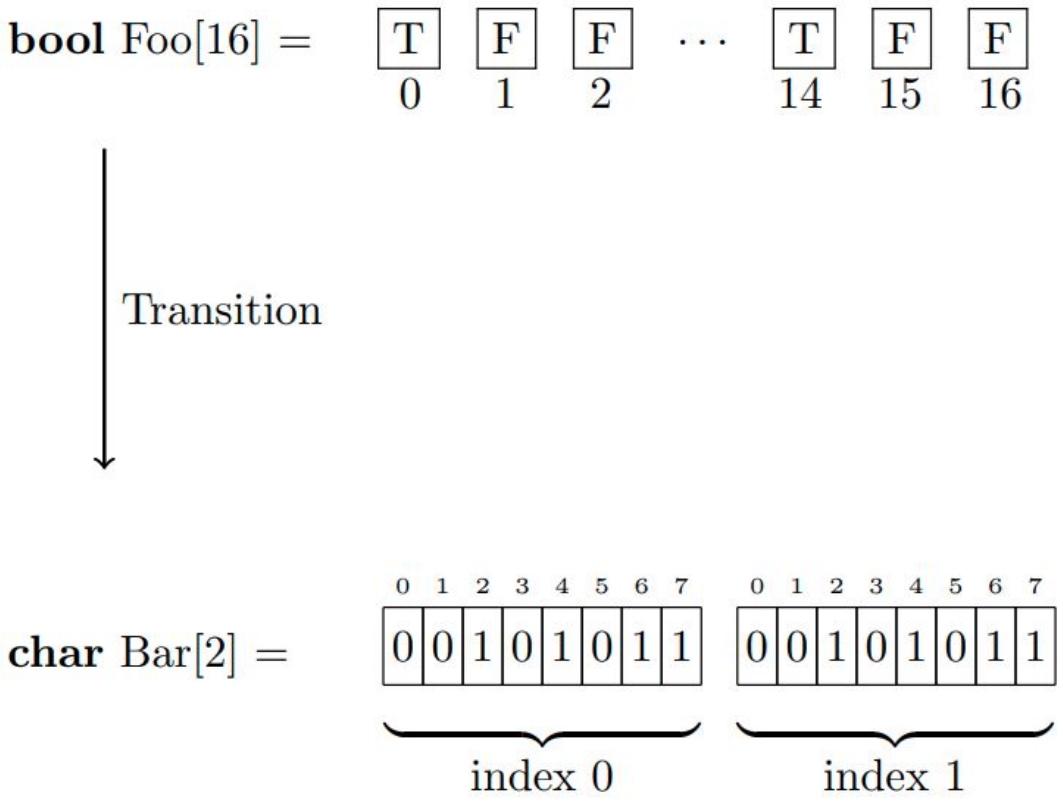
Increased Paralllelism in streamingBFS

Instead of 2 Level Generators,
we now have N Level generators
(N : number of compute units)



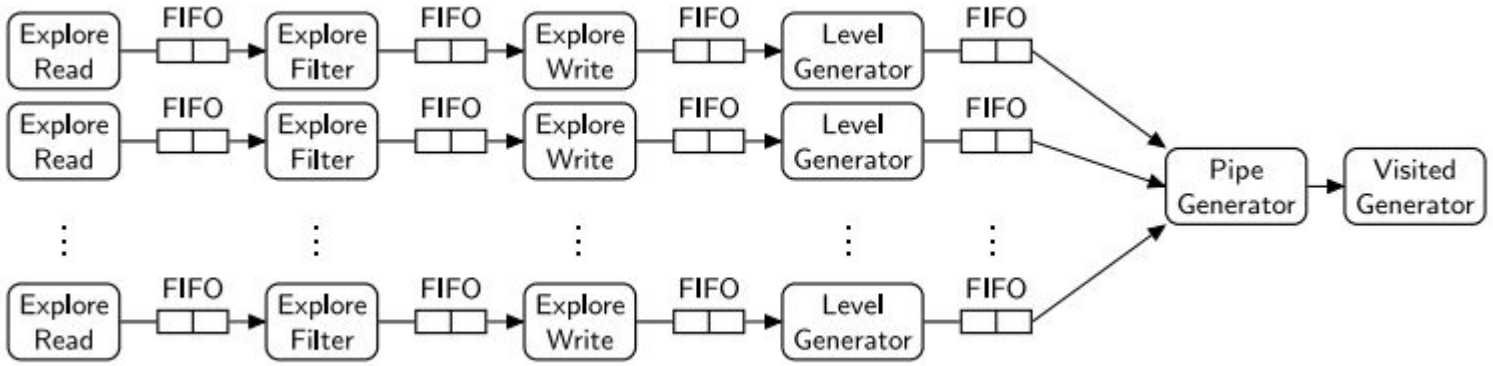
Bit Manipulations in streamingBFS

- Traditional BFS implementations (+ *memoryBFS*) use a boolean array to keep track of visited nodes.
- *streamingBFS* utilises byte storage with bit manipulations
- This method allowed us to reduce the bottlenecks associated with data transfers
- Increased locality when combined with On Chip Memory
- Example: Node 10 Visit Update
  - Bit Index: (10 / 8 = 1)
  - Bit Position: (10 % 8 = 2)



Pipe Working Mechanism in streamingBFS

1. Iterate over number of nodes with tile size *NUM_BITS_VISITED*
2. In each chunk, initialize *StreamingData* class to hold data and valid flag.
3. Populate *StreamingData* class with valid data and padding with non valid data and marking padding data as not valid.
4. After processing the chunk, write the data to an output pipe for further processing.
5. Finally, write back terminating bits to signal to the next kernel that the current processing is complete.



Improvements with streamingBFS

- Sort-Filter Kernel: Modified version of shift register, where we insert new data into the next empty space

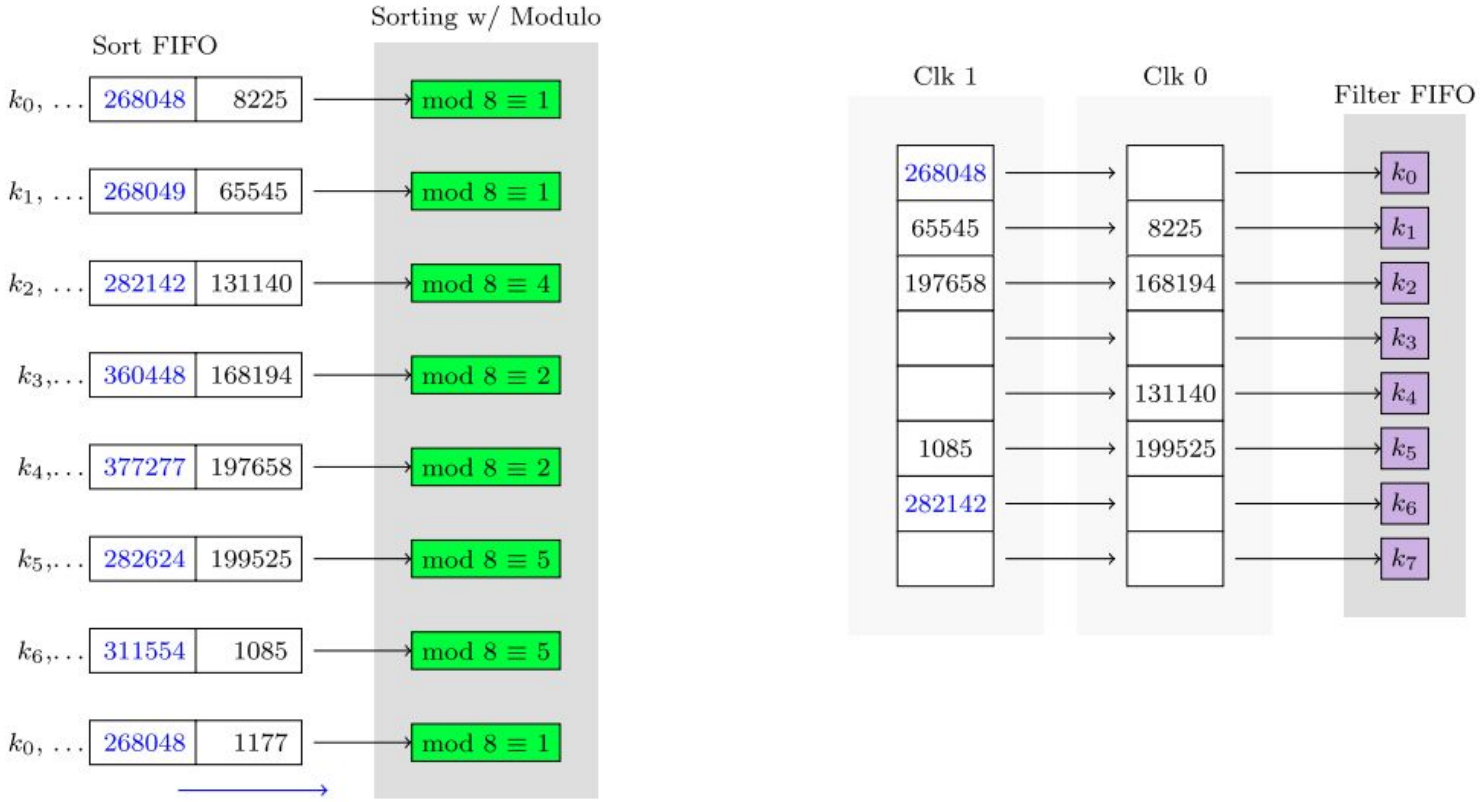


Figure 4: Sort-Filter Mechanism Overview of `streamingBFS`.

Improvements with streamingBFS

- On Chip Memory with Cache
  - Helps us to reduce the write back to memory II to be 1
  - Increases locality
  - Same implementation from oneAPI samples repository

- Execution Steps:
  - Read the pipe
  - Check if the data is valid (not done signal)
  - Read the value from OnChip Memory (It stores 4 last read data in cache so relatively quick access)
  - Toggle the required bits to 1
  - Write back to memory

Summary of streamingBFS

streamingBFS:

- Highlights:
  - Dataflow Execution Model - Uses pipes to stream data between kernels
  - New Kernel Designs with Additions - Ensures synchronisation with minimal latency
  - Compression of Data to Bytes and Bit Manipulations
  - Replaces Some Off-chip Memory Accesses with On-chip Data Movement
  - Modular design – modifying # CUs, FIFO and cache sizes from CMake variables

- Key Limitations:
  - Sort-Filter Kernel II=2
  - During the routing and timing of the design, larger designs have a frequency hit

- The synthesis & performance comparisons shows better insights about memoryBFS and streamingBFS

Synthesis Results for memoryBFS

- Compiler: Intel oneAPI 23.2.0, Target: Intel Stratix 10 GX 2800 FPGA on a Bittware 520N card.
  Synthesis backend: Quartus 20.4.0,

- Only a small fraction of available resources are used
- Generated load caches are not included in these numbers
- There is a slight decrease in clock frequency with more CUs, but the main limitation is that competition on memory bandwidth limits further performance scaling.

Table 2: `MemoryBFS` Utilisation Comparison of Different Compute Units for Stratix 10 FPGA.

CU	FRQ	ALUTs	FFs	MLABs	RAMs
1	297.50	76.5k (4%)	97.7k (3%)	326 (1%)	487 (4%)
2	286.67	91.4k (5%)	123.3k (3%)	359 (1%)	721 (6%)
3	265.62	108.3k (6%)	149k (4%)	392 (1%)	955 (8%)
4	273.33	125.3k (7%)	176.7k (5%)	425 (1%)	1.1k (9%)

The values are for the kernel partition only and do not include the generated caches, which use additional RAMs not reported here.
CU: Number of Compute Units
FRQ: Final clock frequency after place and route
ALUTs: Adaptive Look-up-Tables
FFs: Flip-Flops, RAMs: Random-Access Memory Blocks
MLAB : Memory Logic Array Block, FRQ: Frequency [MHz]

Synthesis Results for streamingBFS

- Compiler: Intel oneAPI 23.2.0, Target: Intel Stratix 10 GX 2800 FPGA on a Bittware 520N card. Synthesis backend: Quartus 20.4.0,

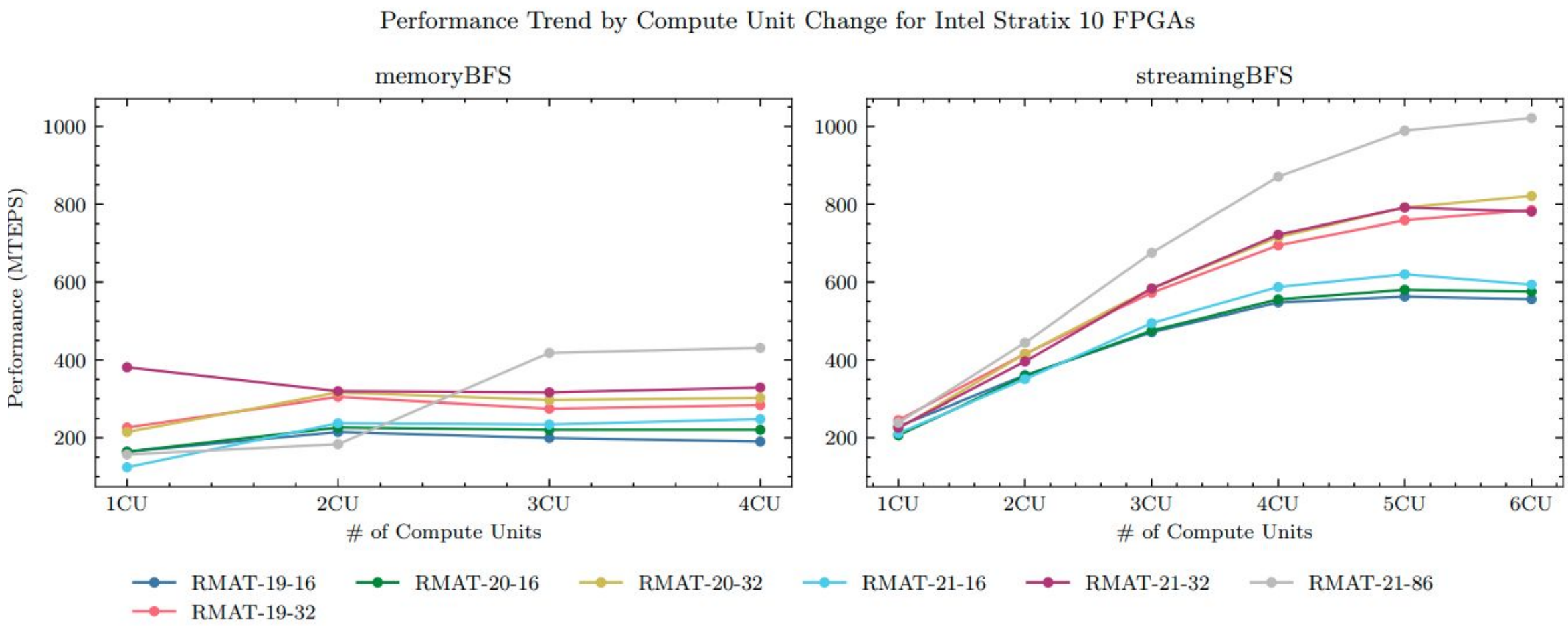
- As the caches are now instantiated as part of the kernels, they show up as part of the resource utilization.
- For smaller cache sizes, routing and timing limit the number of CUs to 6. The overall lower clock frequencies are also a symptom of this.

Table 3: StreamingBFS Utilisation Comparison of Different Compute Units for Stratix 10 FPGA.

	CU	FRQ	ALUTs	FFs	MLABs	RAMs
32738	1	223.33	108k (6%)	162k (4%)	639 (1%)	682 (6%)
	5	191.67	426k (23%)	671k (18%)	3.3k (4%)	2.7k (23%)
65536	1	226.67	100k (5%)	148k (4%)	553 (1%)	890 (8%)
	6	202.86	451k (24%)	709k (19%)	3k (3%)	4.3k (37%)
131072	1	215.00	103k (5%)	150k (4%)	559 (1%)	1.5k (13%)
	6	186.11	465k (25%)	724k (19%)	2.9k (3%)	8.2k (70%)
262144	1	218.75	109k (6%)	168k (5%)	672 (1%)	2k (17%)
	5	192.86	425k (23%)	701k (19%)	3.2k (3%)	9.4k (80%)
393216	1	198.21	109k (6%)	167k (4%)	658 (1%)	3.6k (31%)
	2	205.56	188k (10%)	302k (8%)	1.3k (1%)	7k (60%)

Kernel resources only, also including manually implemented caches with size in bytes per partition(vertical numbers left).

Performance Variation against # of CU



- streamingBFS removes the bandwidth limitations occurred in memoryBFS!

Performance Evaluation

Table 5: Synthetic and Real-World Dataset Throughput (MTEPS) Comparison with State of Art Works.

	memoryBFS		streamingBFS		ThunGP	Chen	OBFS	HitG	GScale	Olgu	Dr.BFS
Graph	PV	Performance	PV	Performance	A HLS	B OpenCL	C OpenCL	D Verilog	C OPAE	E Verilog	F Verilog
DB		21.25 [1CU]		74.83 [2CU]	-	-	-	-	-	9.97	-
EU		4.25 [1CU]		47.51[1CU]	-	-	-	-	-	42.79	-
LJ		195.91 [4CU]		332.47[2CU]	309.21	77.86	475.7	136.14	450	-	-
OR		220.86 [4CU]		616.19 [4CU]	-	-	-	-	-	-	-
PK		206.57 [4CU]		520.04 [4CU]	420.82	104.73	557.8	-	-	-	-
PP		1.89 [1CU]		21.22 [1CU]	-	-	-	-	-	8.79	-
WG		34.91 [1CU]		87.97 [2CU]	58.4	25.09	-	-	-	-	-
WT		52.09 [2CU]		98.68 [4CU]	328.75	72.38	-	125.5	250	220	-
WV		9.67 [1CU]		94.54 [1CU]	-	-	-	-	-	73	-
YT		43.32 [1CU]		88.77[3CU]	-	-	79.4	-	-	9.43	-
R19-16		214.92 [2CU]		562.46 [5CU]	-	-	-	-	-	83.7	-
R19-32		305.01 [2CU]		785.19 [6CU]	933.8	245.8	787.7	-	-	150.19	-
R21-16		248.35 [4CU]		620.09 [5CU]	-	-	-	-	-	96.8	210
R21-32		381.09 [1CU]		791.38 [5CU]	1083.4	243.8	934.2	-	-	-	410
R21-86		431.08 [4CU]		1021.09 [6CU]	-	-	-	643.4	380	-	-

Performance values are reported in (MTEPS), PV: performance variation (MTEPS vs # compute units) Worst Best

FPGAs used Our work: Intel Stratix 10 GX 2800, B: Intel/Altera Stratix V GX, C: Intel Arria 10 GX 1150, F: Intel Arria 10 SX 660
A: AMD/Xilinx Alveo U250, D: AMD/Xilinx Virtex UltraScale+ XCVU5P, E: AMD/Xilinx ZedBoard

Performance Evaluation

Table 5: Synthetic and Real-World Dataset Throughput (MTEPS) Comparison with State of Art Works.

	memoryBFS		streamingBFS		ThunGP	Chen	OBFS	HitG	GScale	Olgu	Dr.BFS
Graph	PV	Performance	PV	Performance	A HLS	B OpenCL	C OpenCL	D Verilog	C OPAE	E Verilog	F Verilog
DB		21.25 [1CU]		74.83 [2CU]	-	-	-	-	-	9.97	-
EU		4.25 [1CU]		47.51[1CU]	-	-	-	-	-	42.79	-
LJ		195.91 [4CU]		332.47[2CU]	309.21	77.86	475.7	136.14	450	-	-
OR		220.86 [4CU]		616.19 [4CU]	-	-	-	-	-	-	-
PK		206.57 [4CU]		520.04 [4CU]	420.82	104.73	557.8	-	-	-	-
PP		1.89 [1CU]		21.22 [1CU]	-	-	-	-	-	8.79	-
WG		34.91 [1CU]		87.97 [2CU]	58.4	25.09	-	-	-	-	-
WT		52.09 [2CU]		98.68 [4CU]	328.75	72.38	-	125.5	250	220	-
WV		9.67 [1CU]		94.54 [1CU]	-	-	-	-	-	73	-
YT		43.32 [1CU]		88.77[3CU]	-	-	79.4	-	-	9.43	-
R19-16		214.92 [2CU]		562.46 [5CU]	-	-	-	-	-	83.7	-
R19-32		305.01 [2CU]		785.19 [6CU]	933.8	245.8	787.7	-	-	150.19	-
R21-16		248.35 [4CU]		620.09 [5CU]	-	-	-	-	-	96.8	210
R21-32		381.09 [1CU]		791.38 [5CU]	1083.4	243.8	934.2	-	-	-	410
R21-86		431.08 [4CU]		1021.09 [6CU]	-	-	-	643.4	380	-	-

Performance values are reported in (MTEPS), PV: performance variation (MTEPS vs # compute units) Worst Best

FPGAs used Our work: Intel Stratix 10 GX 2800, B: Intel/Altera Stratix V GX, C: Intel Arria 10 GX 1150, F: Intel Arria 10 SX 660
A: AMD/Xilinx Alveo U250, D: AMD/Xilinx Virtex UltraScale+ XCVU5P, E: AMD/Xilinx ZedBoard

Performance Evaluation

Table 5: Synthetic and Real-World Dataset Throughput (MTEPS) Comparison with State of Art Works.

	memoryBFS		streamingBFS		ThunGP	Chen	OBFS	HitG	GScale	Olgu	Dr.BFS
Graph	PV	Performance	PV	Performance	A HLS	B OpenCL	C OpenCL	D Verilog	C OPAE	E Verilog	F Verilog
DB		21.25 [1CU]		74.83 [2CU]	-	-	-	-	-	9.97	-
EU		4.25 [1CU]		47.51[1CU]	-	-	-	-	-	42.79	-
LJ		195.91 [4CU]		332.47[2CU]	309.21	77.86	475.7	136.14	450	-	-
OR		220.86 [4CU]		616.19 [4CU]	-	-	-	-	-	-	-
PK		206.57 [4CU]		520.04 [4CU]	420.82	104.73	557.8	-	-	-	-
PP		1.89 [1CU]		21.22 [1CU]	-	-	-	-	-	8.79	-
WG		34.91 [1CU]		87.97 [2CU]	58.4	25.09	-	-	-	-	-
WT		52.09 [2CU]		98.68 [4CU]	328.75	72.38	-	125.5	250	220	-
WV		9.67 [1CU]		94.54 [1CU]	-	-	-	-	-	73	-
YT		43.32 [1CU]		88.77[3CU]	-	-	79.4	-	-	9.43	-
R19-16		214.92 [2CU]		562.46 [5CU]	-	-	-	-	-	83.7	-
R19-32		305.01 [2CU]		785.19 [6CU]	933.8	245.8	787.7	-	-	150.19	-
R21-16		248.35 [4CU]		620.09 [5CU]	-	-	-	-	-	96.8	210
R21-32		381.09 [1CU]		791.38 [5CU]	1083.4	243.8	934.2	-	-	-	410
R21-86		431.08 [4CU]		1021.09 [6CU]	-	-	-	643.4	380	-	-

Performance values are reported in (MTEPS), PV: performance variation (MTEPS vs # compute units) Worst Best

FPGAs used Our work: Intel Stratix 10 GX 2800, B: Intel/Altera Stratix V GX, C: Intel Arria 10 GX 1150, F: Intel Arria 10 SX 660
A: AMD/Xilinx Alveo U250, D: AMD/Xilinx Virtex UltraScale+ XCVU5P, E: AMD/Xilinx ZedBoard

Attempted Optimisations Summary

Optimisation	Summary	Outcome
Split pointers for USM	Define several pointers to different parts of data	⬆️
Datatype	Use datatypes according to the needs	⬆️
Read-Only Cache	Boost read only performances	⬆️
Parallel For, NDRange Kernels	Improve the performance of parallel kernels	⬇️
Automatic Cache	Increase the locality of data	⬆️
Bit Manipulations	Compressing information we have	⬆️

Summary for Today

- 1021 MTEPS peak performance with 6 compute units for *streamingBFS*
- *streamingBFS* provides lots of insights what works best for FPGAs
- Compression of data helps to improve performance drastically
- Pipes for efficient communication between kernels
- Edge/Node split performance depends on dataset/implementation

Thank you!

- If you are interested in FPGAs you could apply for an account at Paderborn Noctua2 system to experiment