OPM hardware acceleration

TOM HOGERVORST

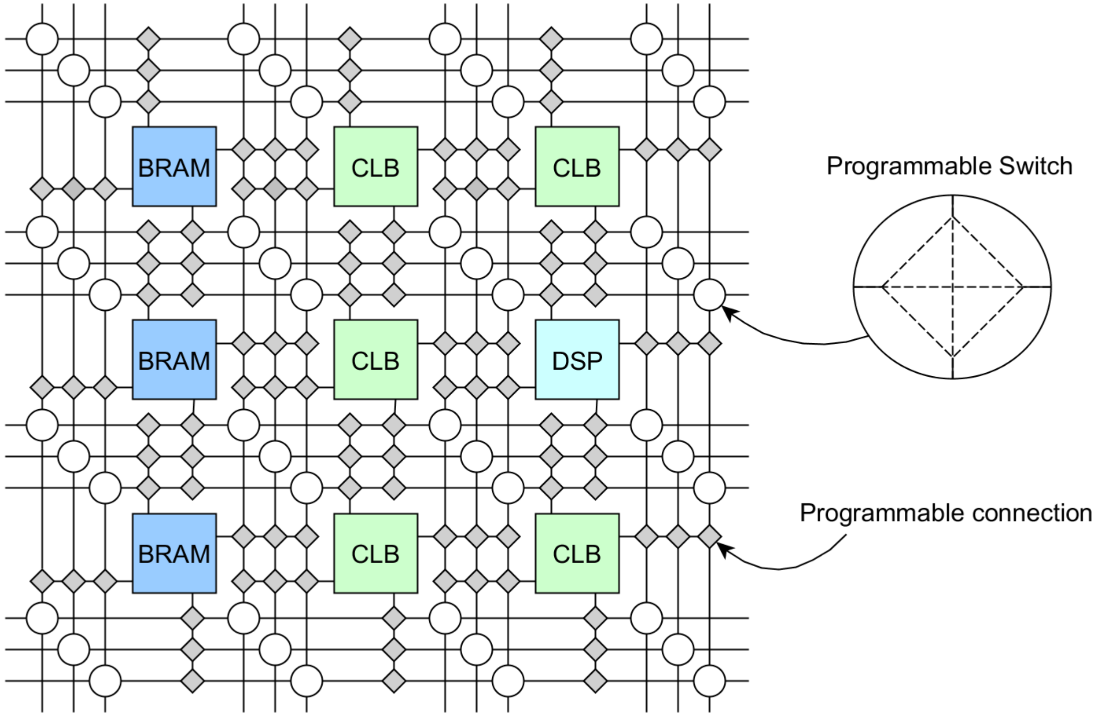
TONG DONG QIU

Introduction

- What we do: Acceleration of the flow simulator
  - Specifically: the linear solver
  - BiCGSTAB solver with ILU0 preconditioner
- Research into different platforms
  - FPGA
  - GPU

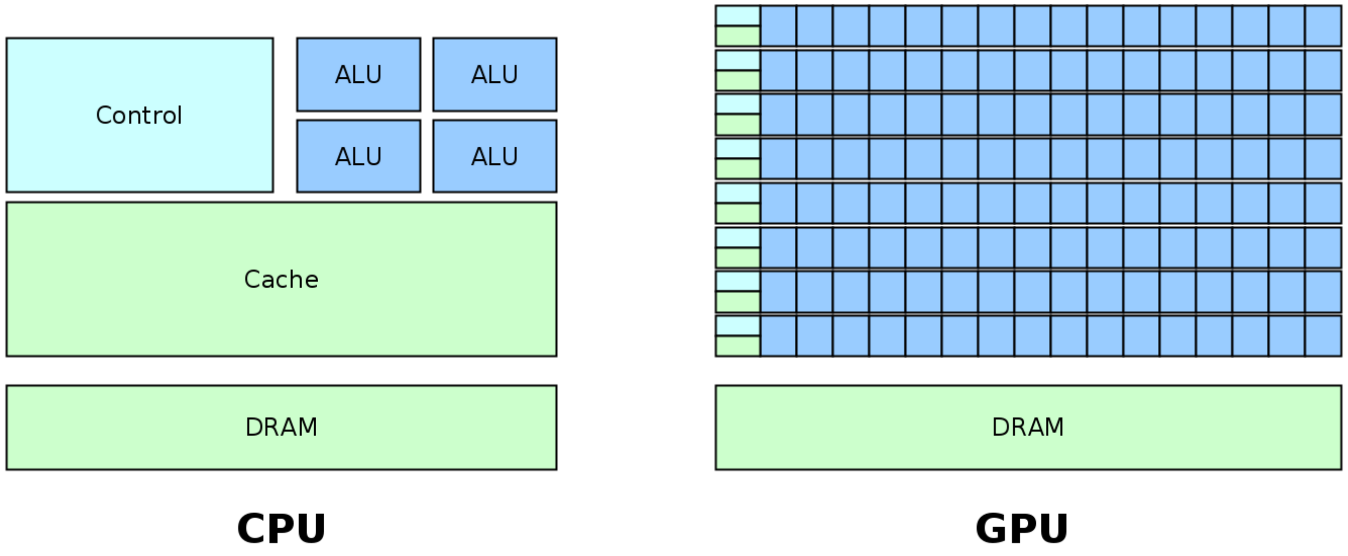
FPGA Introduction

- What are FPGAs?
  - Array of Configurable Logic Blocks
  - Also contains memory and DSP blocks
  - Fully programmable
- Why use FPGAs?
  - Hardware tailored to application
  - Exploit pipeline parallelism
  - Lower power consumption



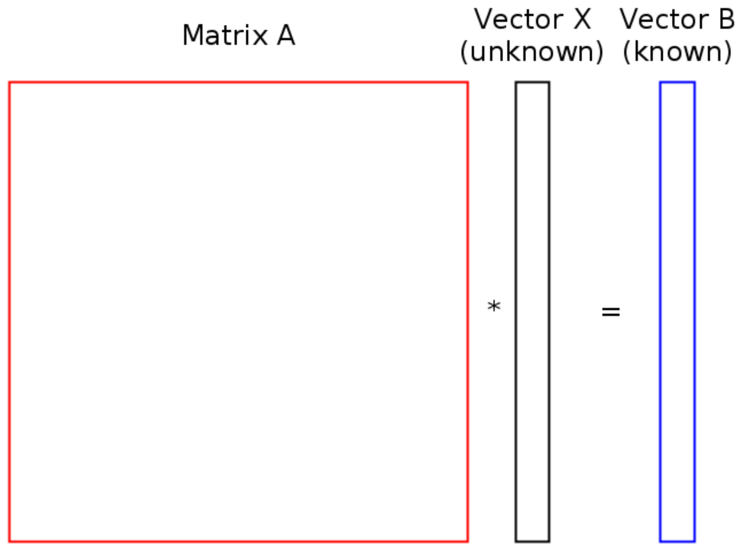
GPU Introduction

- What are GPUs?
  - Many cores with shared control
  - Small caches
  - Exploit parallelism
- Why use GPUs?
  - Massive SIMT parallelism
  - Large bandwidth



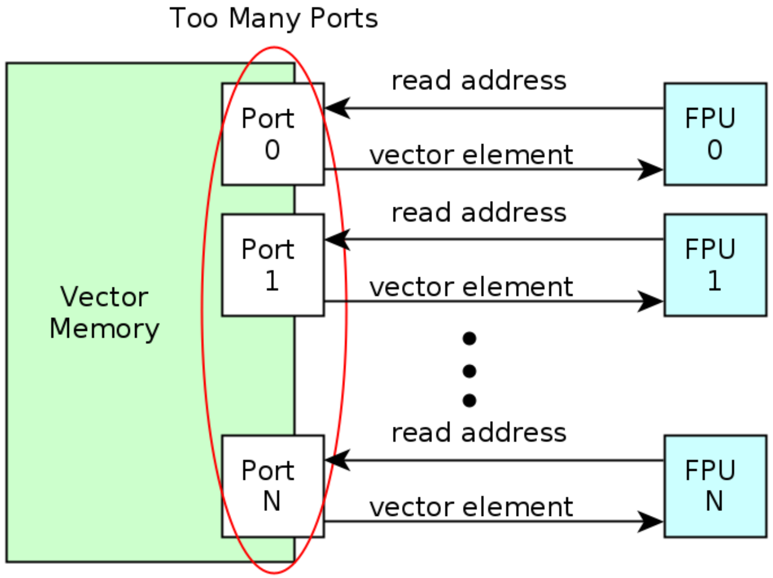
The Solver Function

- Solver application consists of:
  - Sparse Matrix operations (SPMV, apply_ILU0)
  - Vector operations (dot, axpy)
- Main challenges:
  - Random accesses in matrix operations
  - Apply_ILU0 is sequential



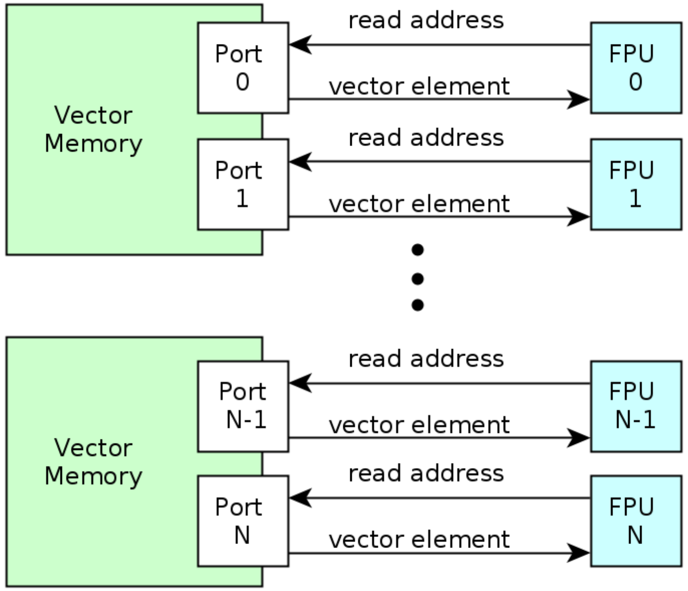
The FPGA accelerator

- Problem: many parallel random accesses to same array



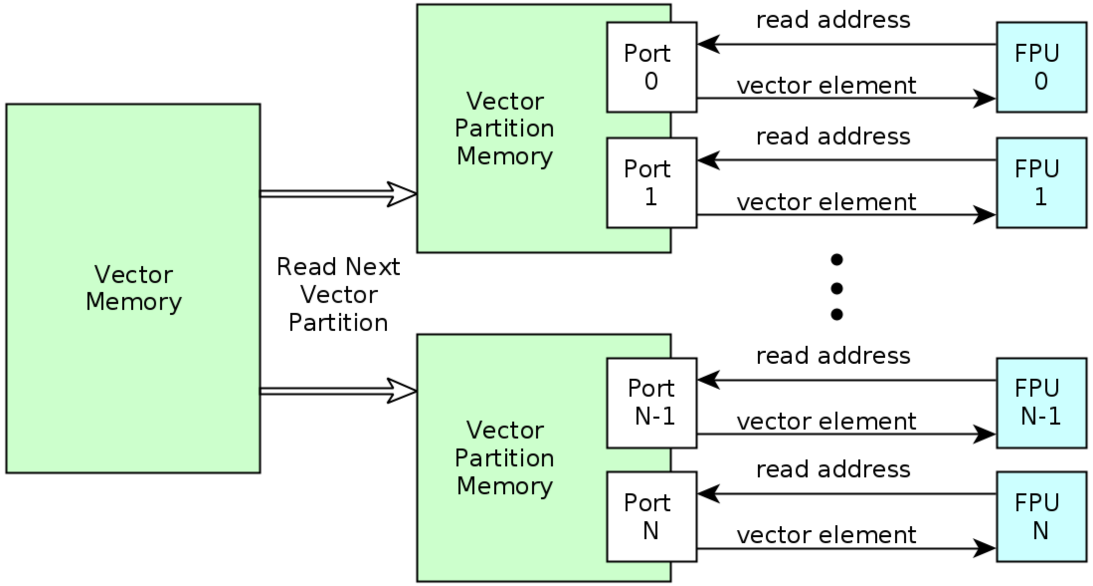
The FPGA accelerator

- Problem: many parallel random accesses to same array
  - Partial Solution: Duplicate the vector
    - Not scalable: larger vectors don't fit on board multiple times



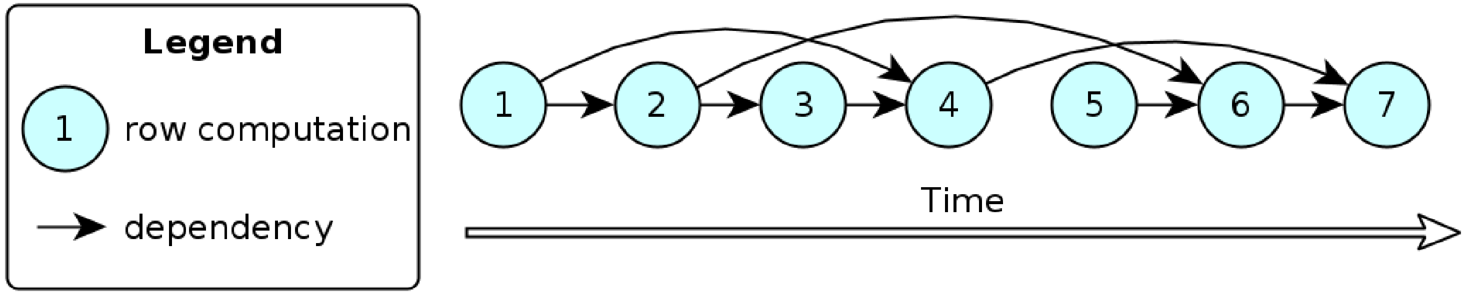
The FPGA accelerator

- Problem: many parallel random accesses to same array
  - Partial Solution: Duplicate the vector
    - Not scalable: larger vectors don't fit on board multiple times
  - Solution: Partition matrix and vector, duplicate vector partitions



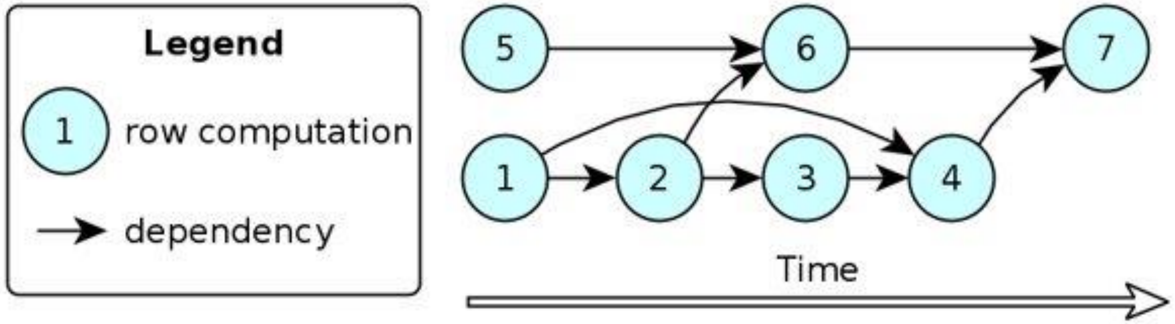
The FPGA accelerator

- Parallelize apply_ILU0
  - Sequential



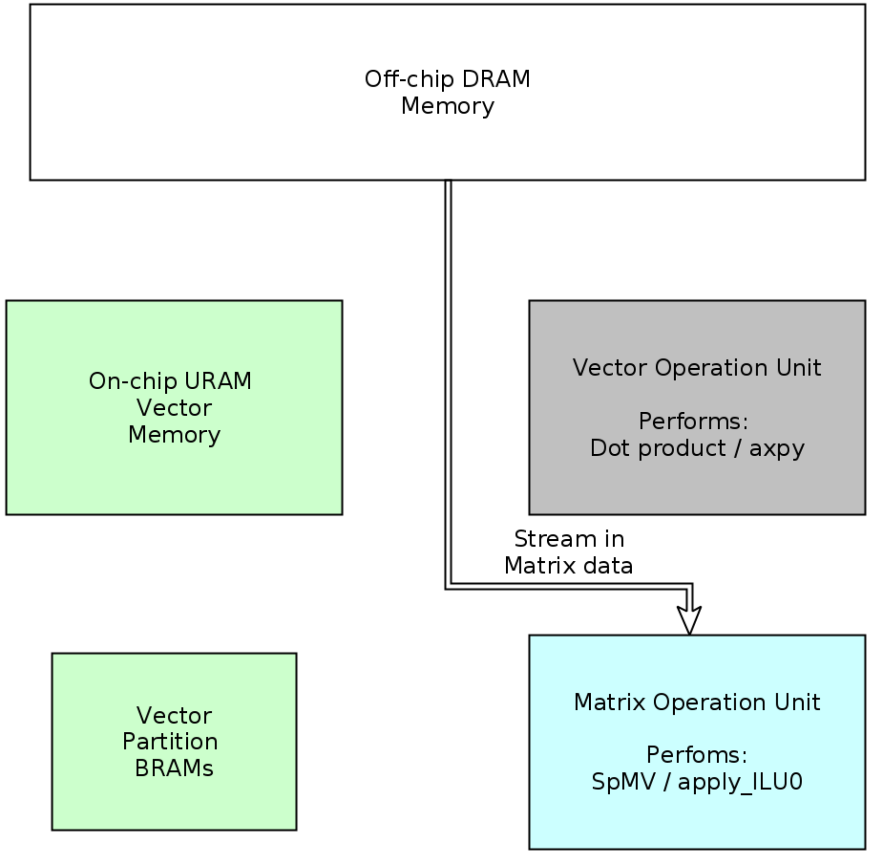
The FPGA accelerator

- Parallelize apply_ILU0
  - Sequential
  - Level-scheduling
- Choose levels as partitions



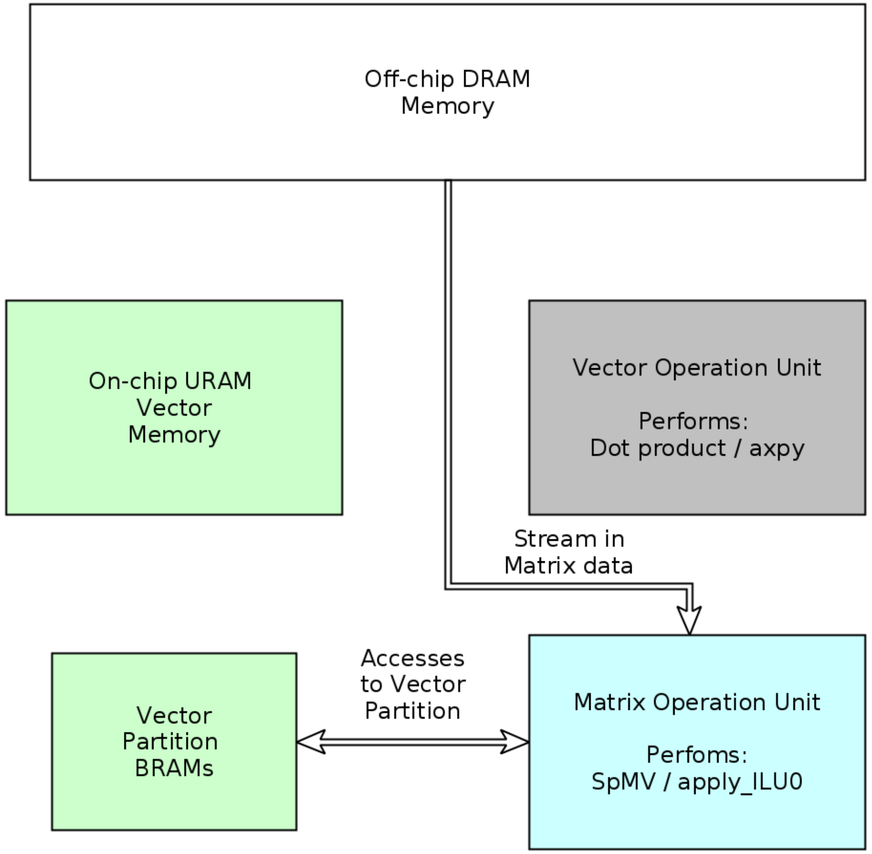
The FPGA accelerator

- Overview of matrix operation in kernel



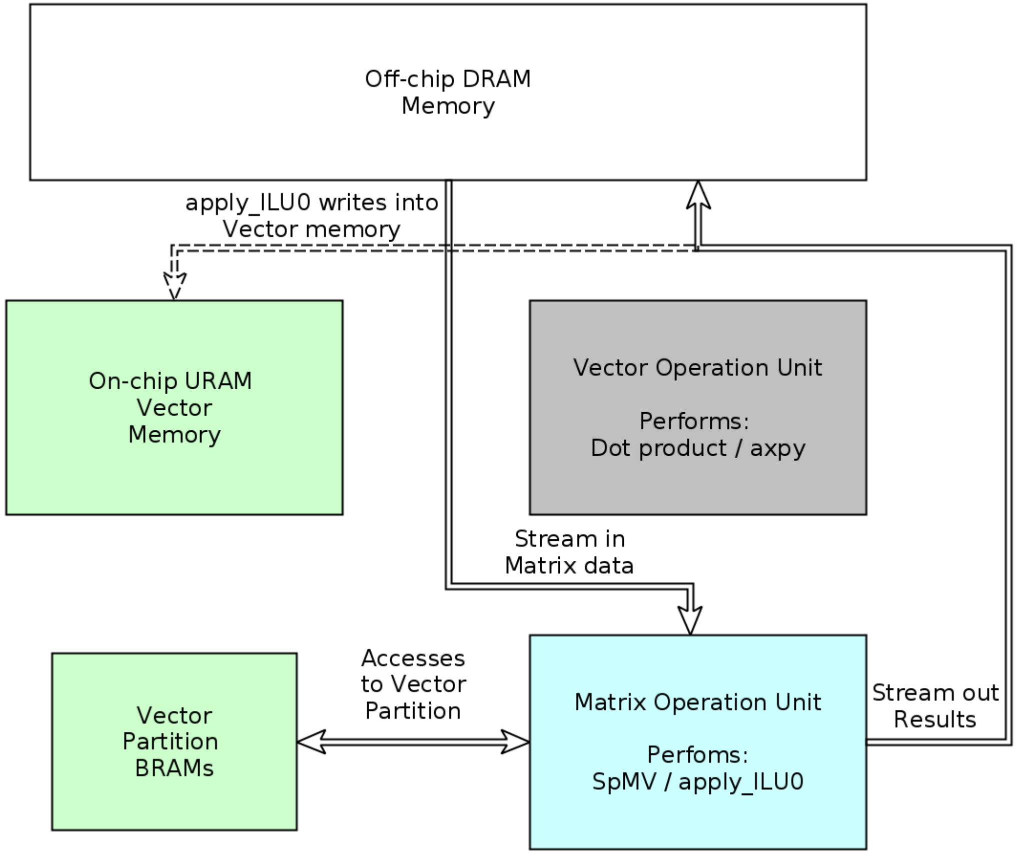
The FPGA accelerator

- Overview of matrix operation in kernel



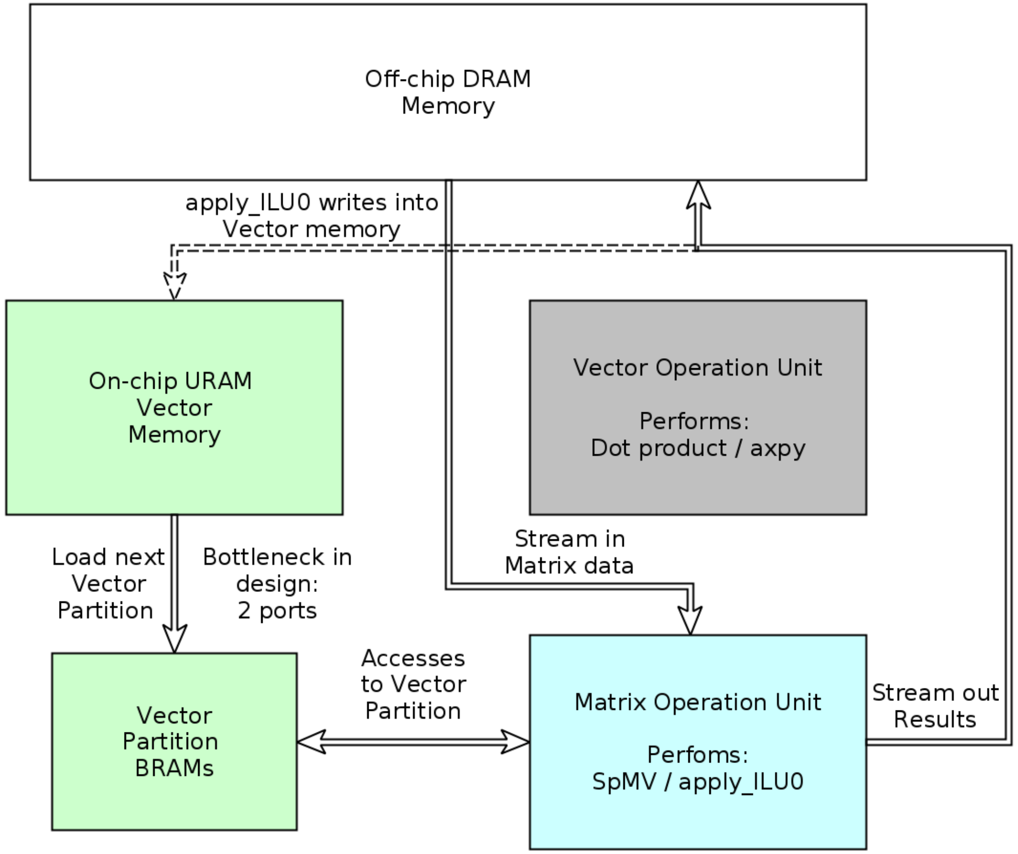
The FPGA accelerator

- Overview of matrix operation in kernel



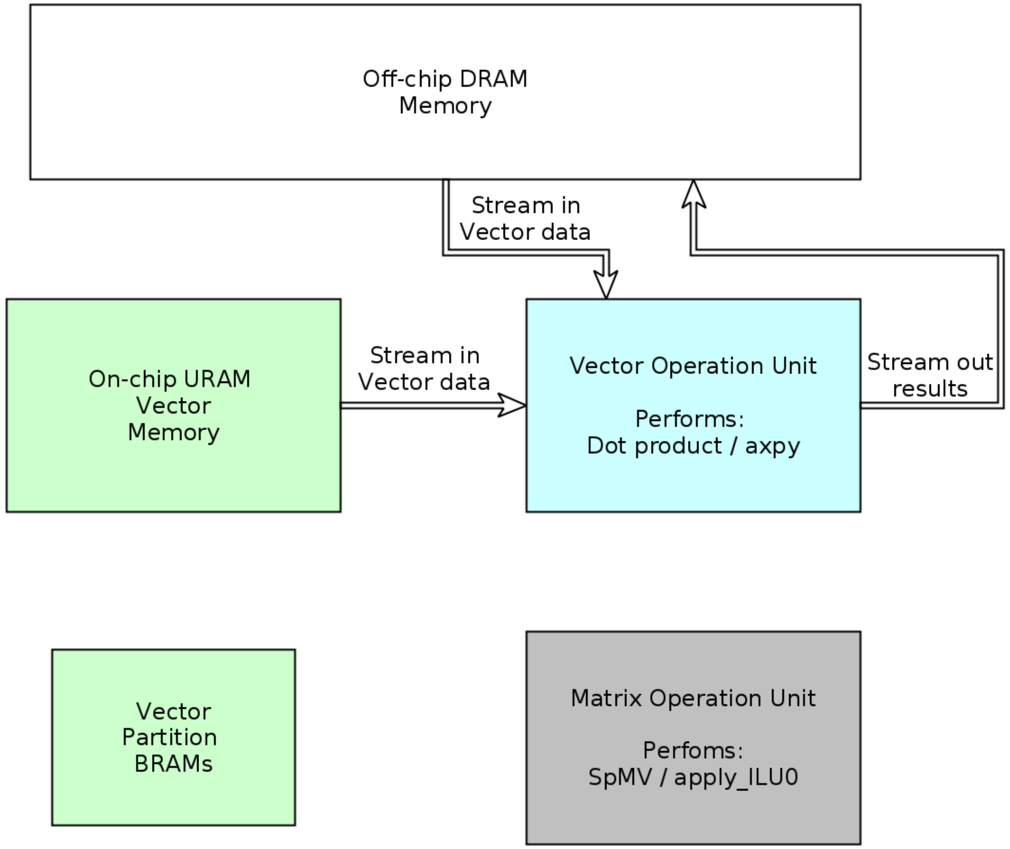
The FPGA accelerator

- Overview of matrix operation in kernel



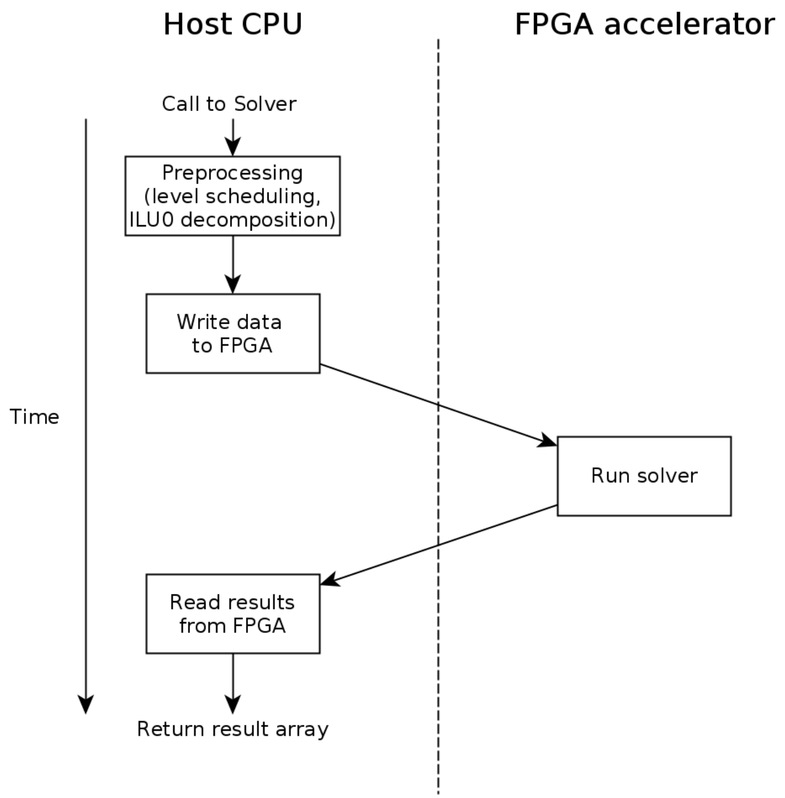
The FPGA accelerator

- Overview of vector operation in kernel



The FPGA accelerator

- Overview of using the accelerator



FPGA Results

- FPGA board specifications:

	Alveo U200	Alveo U280
LUTs	1,180,000	1,304,000
Registers	2,364,000	2,607,000
DSP Blocks	6,840	9,024
BRAM (on-chip) size // latency	9 MB // 2 cycles	9 MB // 2 cycles
URAM (on-chip) size // latency	34 MB // 4 cycles	34 MB // 4 cycles
DRAM (off-chip)	64 GB (4 channels)	32 GB (2 channels)
DRAM aggregated bandwidth // latency	77 GB/s // 130 cycles	38 GB/s // 130 cycles
HBM	n/a	8 GB (32 channels)
HBM aggregated bandwidth // latency	n/a	460 GB/s // 120 cycles

FPGA Results

- FPGA kernel specifications:

	Alveo U200	Alveo U280
LUTs	9.00%	7.83%
Registers	7.32%	6.57%
DSP Blocks	4.21%	3.18%
BRAM used	33.48%	26.88%
URAM used	4.17%	4.17%
Frequency	186 MHz	280 MHz

- CPU specifications:
  - Xeon Silver 4114 CPU @ 2.20GHz

FPGA Results

- Specifics of the current design:

	FPGA kernel
Number of FPUs	24 (8 in Matrix unit, 16 in vector units)
Internal bandwidth	4.5 GB/s (2 ports)
Ports to DRAM	2 read
Achieved bandwidth to DRAM	13 GB/s
Ports to HBM	3 read, 3 write
Achieved bandwidth to HBM	22 GB/s reading, 42 GB/s writing

FPGA Results

- FPGA solver integrated with flow
  - ISTLSolverEbos.hpp -> constructPreconditionerAndSolve()
  - BdaBridge to allow multiple backends
- Ran flow with NORNE into file
  - Results verified with ResInsight

	DUNE	FPGA kernel (U280)
Total time (s)	709.3	1409.1
Assembly time (s)	286.3	267.1
Linear solve time (s)	413.2	1088.6
Newton Iterations	1605	1616
Linear Iterations	24440	23721

FPGA Results

- Breakdown of FPGA kernel solver time
  - All times are accumulated over one run of flow on the NORNE testcase

	FPGA kernel time
Preprocessing (in software)	590.5 s
Memory setup	19.4 s
Transfer to/from FPGA DRAM+HBM	70.8 s
Kernel Solver	387.9 s

FPGA Conclusions

- Current work:
  - Pre-processing software optimization
  - Increasing HBM Bandwidth utilization
  - Increasing the number of FPUs
- Interesting possibility: no partitioning
  - Not scalable
  - Reduces pre-processing time by 80 %
  - Cumulative solver time estimations:

CPU solver	Current Design without partitioning	Design with double the amount of FPUs and HBM ports without partitioning
413.2 s	402.15 s	324.6 s

FPGA Conclusions

- Optimizing/Debugging is a slow process
- No specialized blocks for Double precision FP operations
- Memory latency to HBM high
- Frequency of FPGAs lower than CPU/GPU

GPU Implementation

- AMGX
- cusparse
- Zeroes on the diagonal
- WellContributions are included in matrix
- Pull Request 2209 on Github

GPU Implementation

- Nonzeroes of BCRSMatrix in contiguous memory
- Copy nonzeroes row-by-row to contiguous CPU memory
  - Copying row-by-row adds about 10s
- No Pinned memory

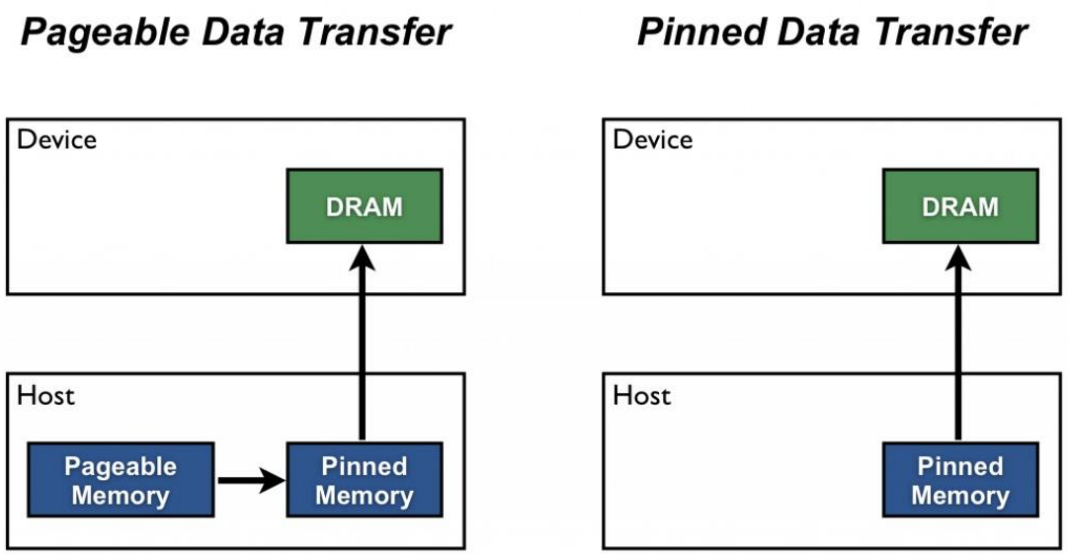


Image source: https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/

GPU Implementation

- Device used: RTX 2080Ti:
  - 4352 cores
  - 616 GB/sec memory bandwidth
  - 420 GFLOPS (double precision)

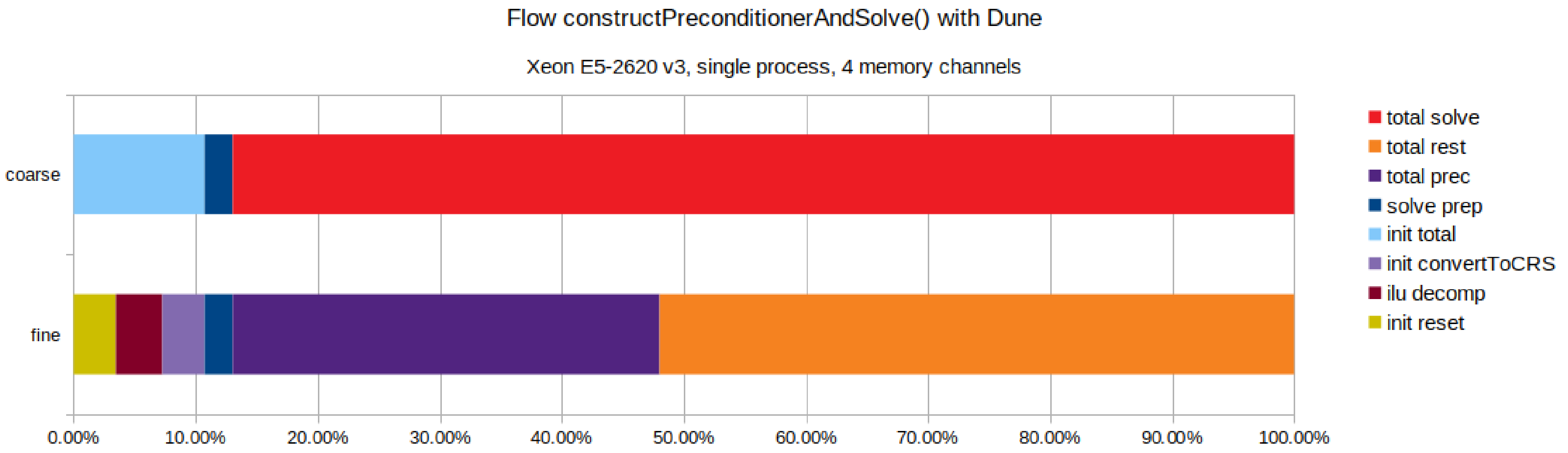
GPU Results

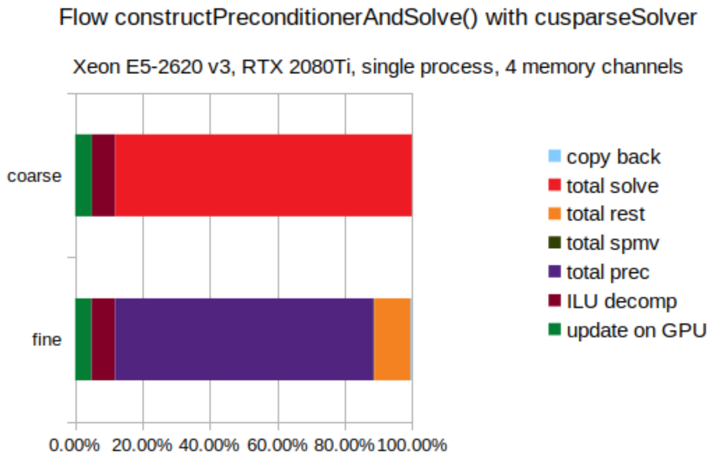
- Xeon E5-2620 v3, RTX2080Ti, 4 memory channels
- Verified output with ResInsight

	Dune	cusparse, RTX2080Ti	Speedup	Speedup (%)
Total time (s)	811	505	1.61	38
Assembly time (s)	303	308		
Linear solve time (s)	447	137	3.26	69
Newton iterations	1597	1629		
Linear iterations	24120	23593		
constructPreconditionerAndSolve()	430	119	3.61	72
BiCGStab	383	99	3.87	74
Transfer time (s)	0	5.9		

BigData Accelerate

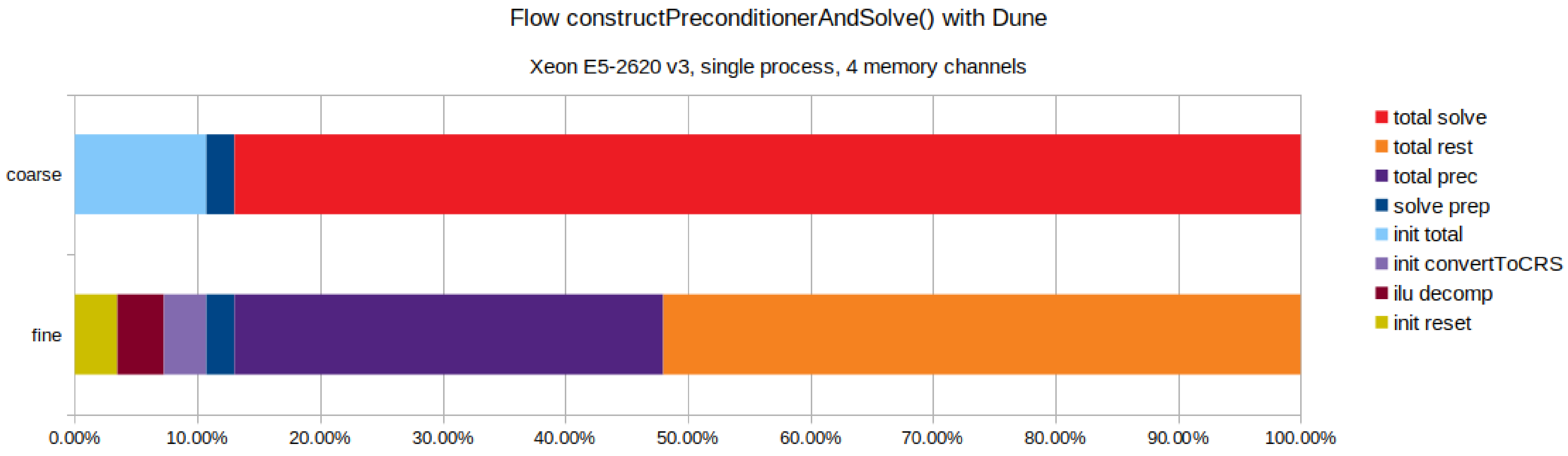
GPU Results





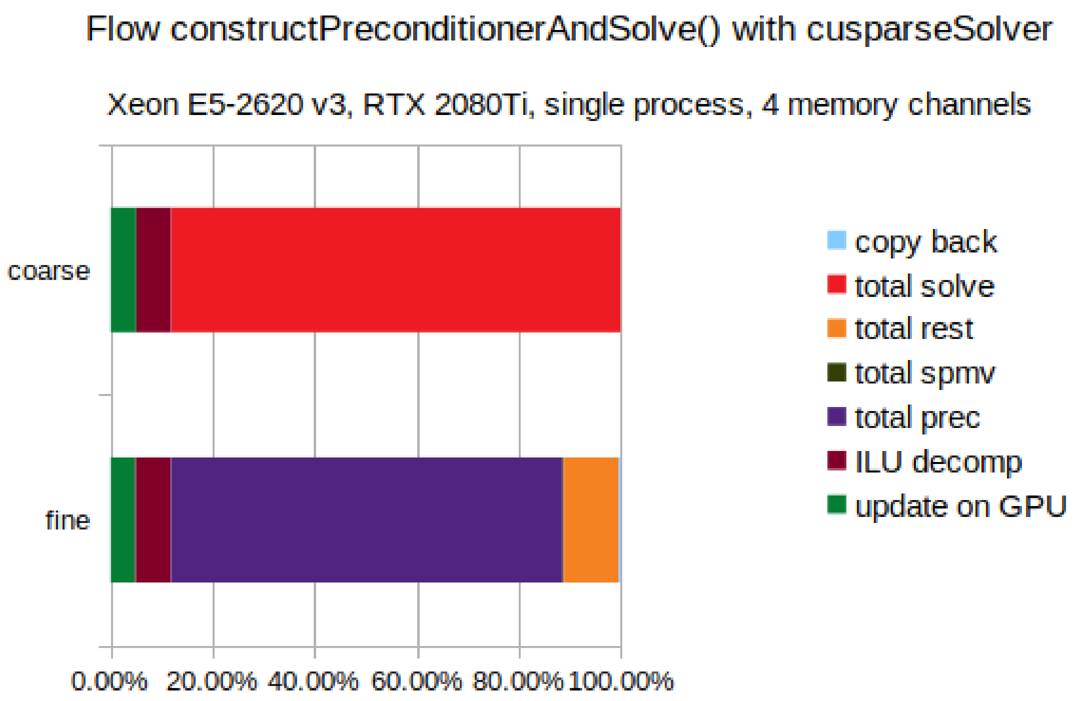
BigData Accelerate

GPU Results



Flow constructPreconditionerAndSolve() with Dune
Xeon E5-2620 v3, single process, 4 memory channels
coarse
fine
0.00%
10.00%
20.00%
30.00%
40.00%
50.00%
60.00%
70.00%
80.00%
90.00%
100.00%
total solve
total rest
total prec
solve prep
init total
init convertToCRS
ilu decomp
init reset

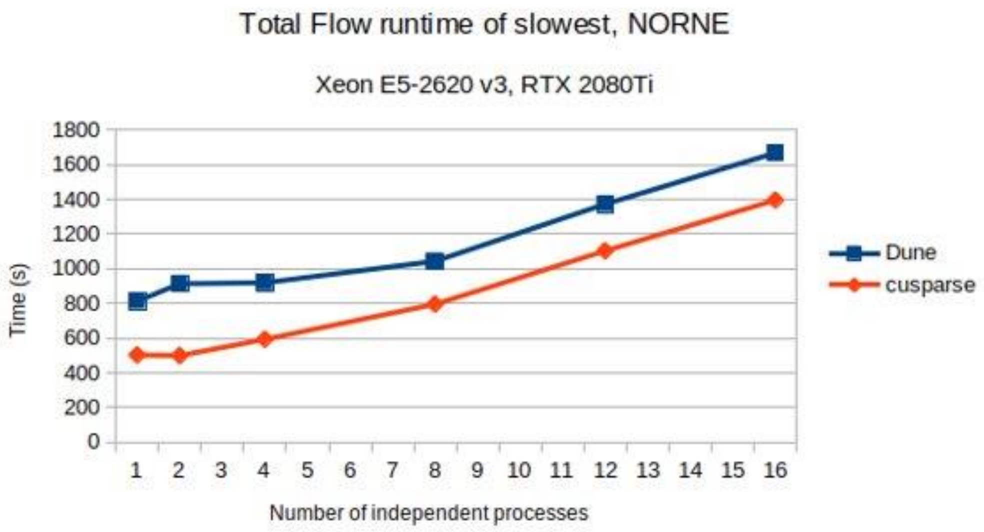
GPU Results

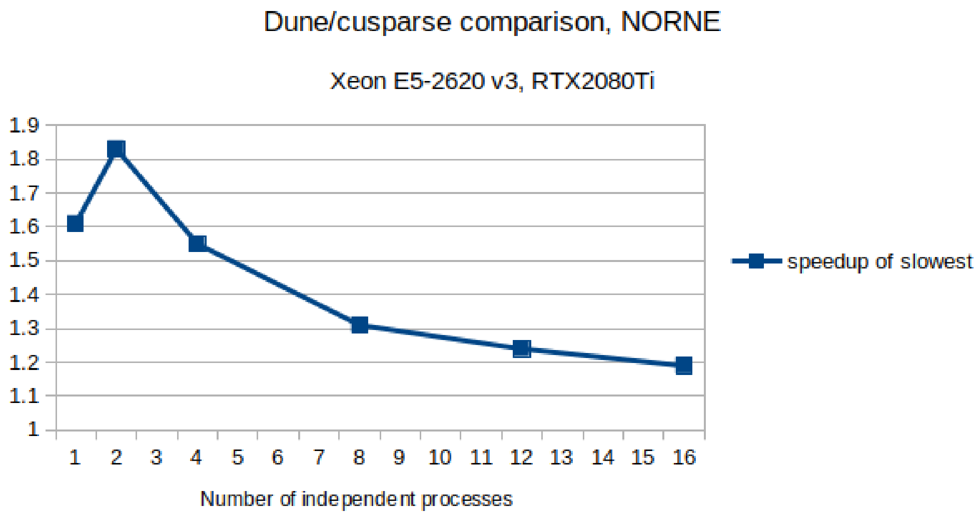


BigData Accelerate

GPU Results

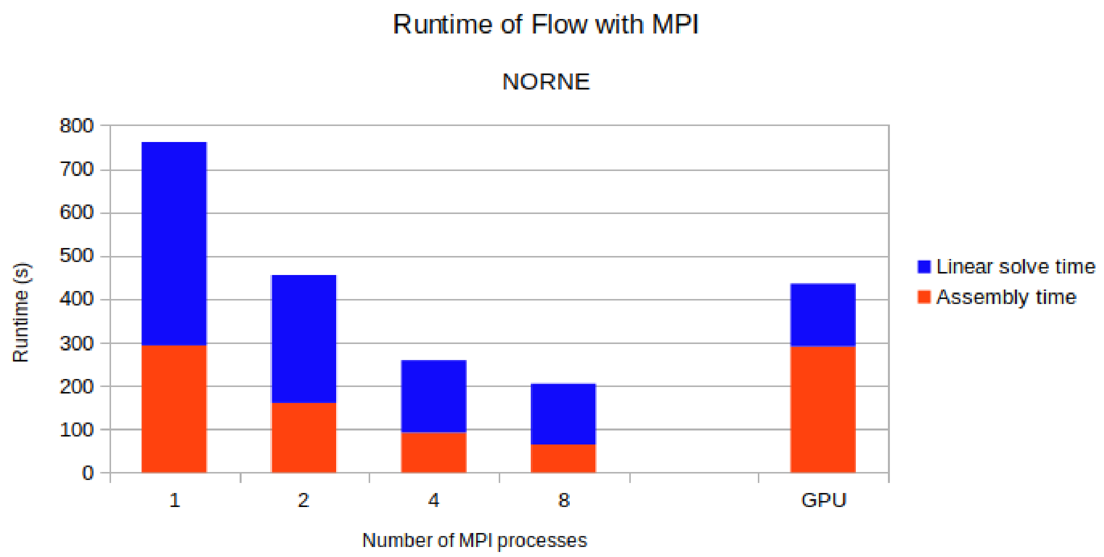
- Running multiple independent Flows





GPU Results

MPI Processes	1	2	4	8	GPU
Total time (s)	823	499	292	234	493
Assembly time (s)	293	160	92	64	290
Linear solve time (s)	468	295	167	141	145



BigData Accelerate

GPU Results

- ILU0 application is the bottleneck:
  - 88% of BiCGStab time
  - GPU memory bandwidth not utilized efficiently (13%)
  - GPU issue efficiency is only 16%

- cusparse is not designed for running multiple processes, could cause trashing

GPU Future Work

- Decouple wellcontributions for larger testcases
- OpenCL
- Manual ILU0 application
  - To better utilize the GPU for ensembles

Thank you

Special thanks to Equinor for making this research possible