Adapting HEP-Frame for a Streaming Approach to a Linear Algebra OLAP Engine

Daniel Rodrigues

- Work Plan - Adapting HEP-Frame for a Streaming Approach to a Linear Algebra OLAP Engine
- RPD
- Presentation
- Code

Advisors: Alberto Proença, André Pereira

Goals

➔ To improve the performance of the current LAQ system, by adapting HEP-Frame to new requirements:

- to implement simultaneous input-data-streaming and processing, in current LAQ system (current version: input-data-batch)
- to port the development of the current LAQ-engine to HEP-Frame
- to add new features in HEP-Frame run-time-engine to address input-data-streaming and stream-scheduling
- to tune, evaluate and compare the performance of LAQ-engine with other database systems, namely MonetDB

Scheduling

- **November 2018**
  - Analyse the internal structure of HEP-Frame
  - Identify the key points requiring modifications (LAQ & HEP-Frame)
- **December 2018**
  - Implement data streaming in the current linear algebra database system
- **January 2019**
  - Port the development and execution of the LAQ engine into a modified version of HEP-Frame
- **March 2019**
  - Test and validate the correctness of the port using some TPC-H benchmarks
- **June 2019**
  - Profile, identify, and improve bottlenecks in HEP-Frame I/O and processing schedulers
- **July 2019**
  - Write the dissertation

Structure of the HEP-Frame

29-nov-18

Key points that require modifications:

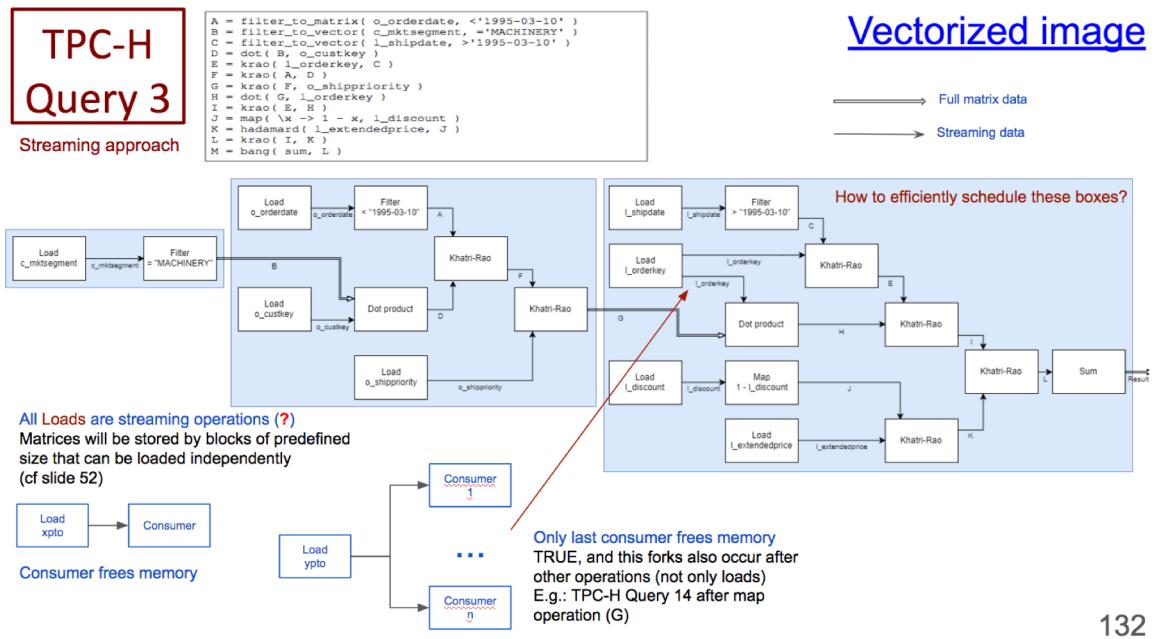
1. New DS: to support multiple input-data-streams for multiple consumers

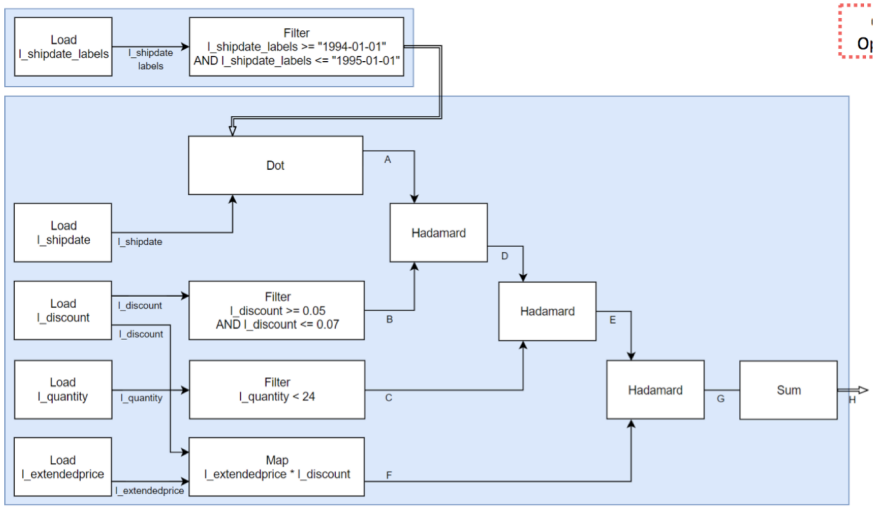
TPC-H Query 3

Streaming approach

```
A = filter_to_matrix( o_orderdate, <'1995-03-10' )
B = filter_to_vector( c_mktsegment, ='MACHINERY' )
C = filter_to_vector( l_shipdate, >'1995-03-10' )
D = dot( B, o_custkey )
E = krao( l_orderkey, C )
F = krao( A, D )
G = krao( F, o_shippriority )
H = dot( G, l_orderkey )
I = krao( E, H )
J = map( \x -> 1 - x, l_discount )
K = hadamard( l_extendedprice, J )
L = krao( I, K )
M = bang( sum, L )
```

Vectorized image





Structure of the HEP-Frame

Key points that require modifications:

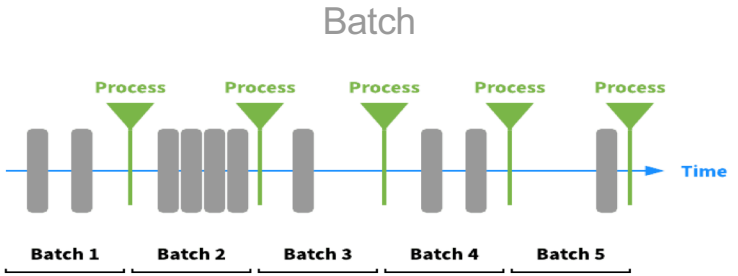
1. New DS to support input-data-streams
   a. Multiple-input-data-stream: schedule several parallel-load-operation(s)
   b. New-DP is now partial, not the full-pipeline
   c. Multiple-consumers: manage memory
   d. DataStore: only frees memory **when?**
2. Scheduling new-DS with partial-DP
3. ...

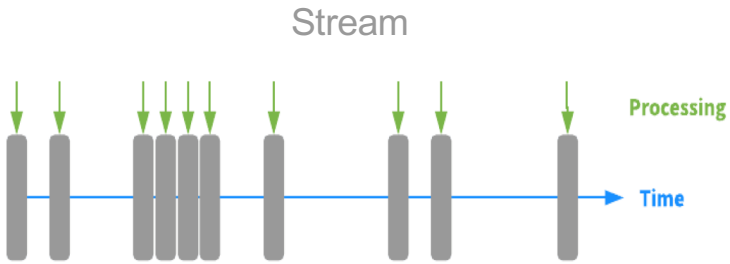
06-dez-18

- Diferenças entre batch, mini-batch e streaming
- Descrição detalhada do problema de streaming (como func uma arq de streaming); spsc, mpms, …
- Descrição detalhada das frameworks (programação, abordagem e performance); incluir STL C++, possivelmente outras; restrições às estruturas de dados?
- Exemplos práticos de programação
- explicação/indicar qualidades ou limitações

Batch, mini-batch and streaming

1. **Batch**: the received data is organized into sets and these sets are processed after a certain period of time.
2. **Mini-Batch**: The idea is similar to batch processing, however the time periods are shorter.
3. **Streaming**: The data is processed individually and the time between data reception and processing is expected to be minimal.



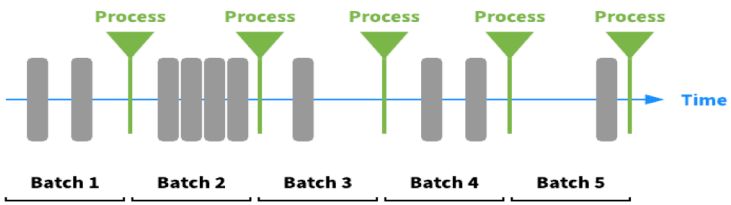


13-dez-18

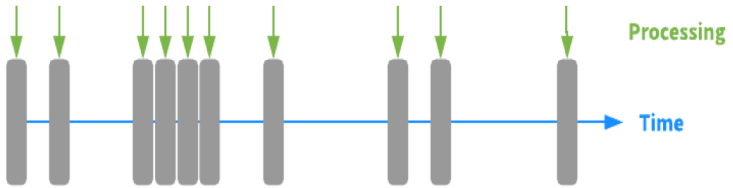
Batch, mini-batch and streaming

	Batch / Mini-batch	Stream
Responsiveness	Low	High (Shorter runtime)
Data flow control	Low	High (Higher overhead)
Processing mode	Discontinuous	Continuous (long-running tasks)

Batch



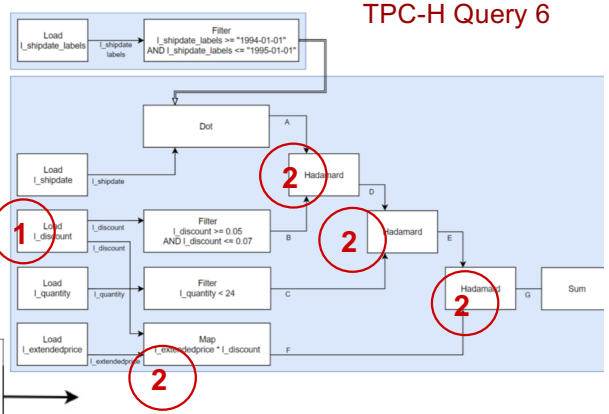
Stream



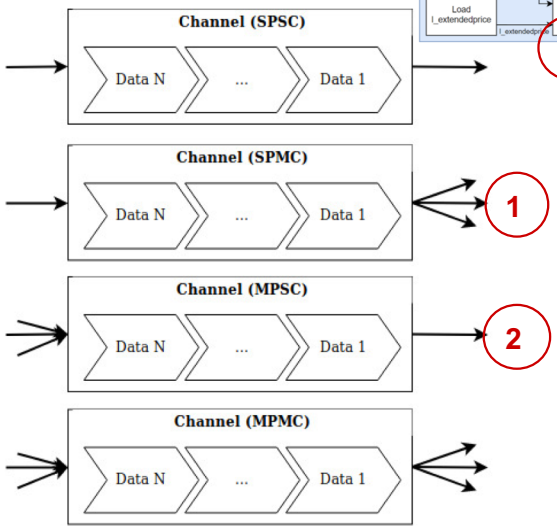
Stream processing - Architecture and Design

1. Stream processing architecture as a graph, where:
   a. Nodes represent producers and/or consumers
   b. Edges represent communication channels

TPC-H Query 6



1. Different settings for channels
   a. Single-Producer-Single-Consumer (SPSC)
   b. Single-Producer-Multiple-Consumer (SPMC)
   c. ...



From slide 4:

Key points that require modifications:

1. New DS: to support multiple input-data-streams for multiple consumers

Requirements

1. Architecture must support data streaming
2. At thread level
3. Flexibility in communication channels
4. Flexibility in memory control
5. Efficiency

Available tools:

- TBB
- FastFlow
- Boost (lockfree)
- Boost (Synchronized Queues)*
- RaftLib
- STL (Not found)
- OpenMP (?)

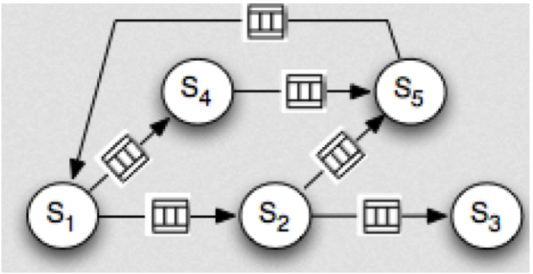
Comparison of libraries

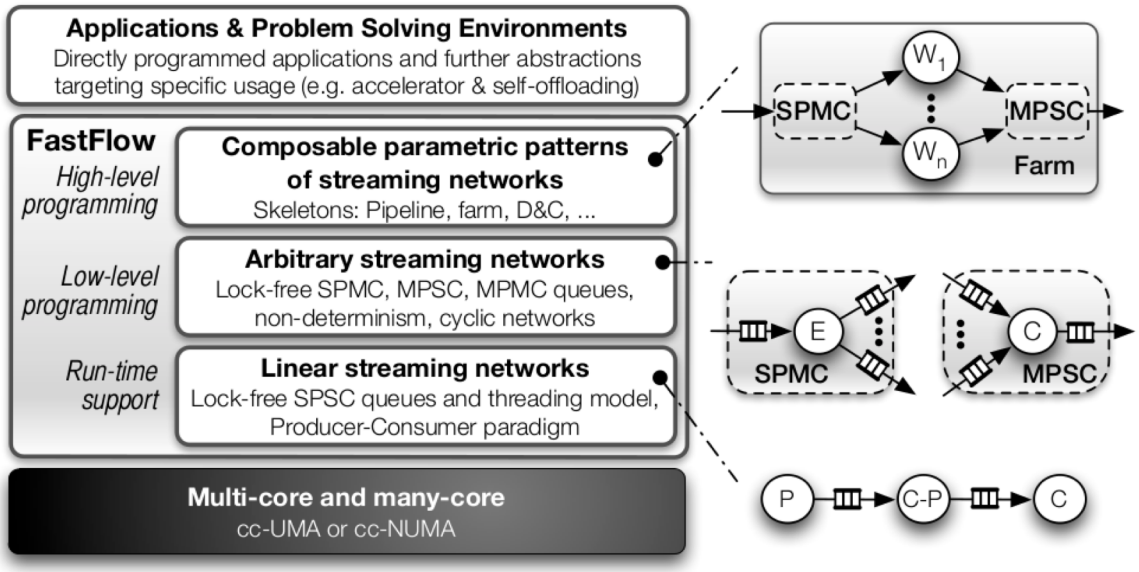
(unfinished)

	TBB	FastFlow	Raftlib
Communication flexibility	High	Medium (Limited)	Very High
Work control (run time)	-	-	-
Runtime	Long	Short	Medium
Architecture and design	Good (low focus on the data flow)	Good architecture (complex implementation)	Simple and good
Documentation	Superficial (TBB-Tutorial)	Good	Good
Last update	-	7 month ago (github)	1 month ago (github)

FastFlow

- Specifically targeting cache-coherent shared-memory
- Lock-free queue (FIFO)



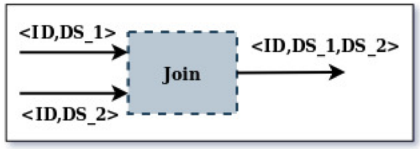


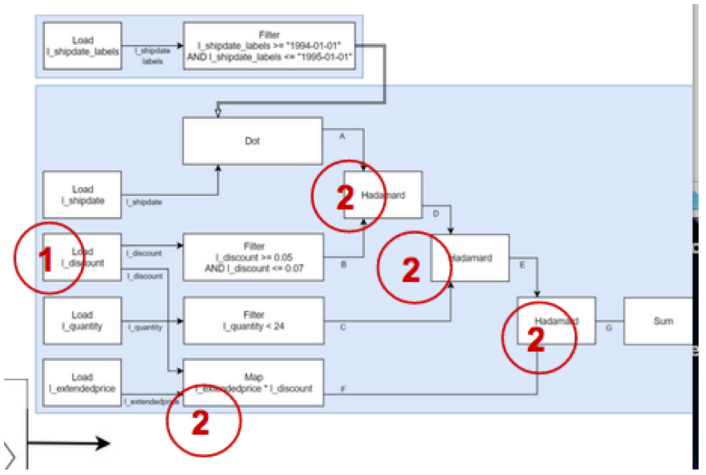
07-jan-19

Implementation of the data stream in the LA system: Requirements

1. Support multiple-output-data-stream (Slide 9(1))
   a. Since DSs are not changed by consumers, they can be shared by all, so references to DS are sent to all consumers associated with the producer

2. Support multiple-inputs-data-stream (Slides 4, 9(2))
   a. When the input elements are joined, the elements are ordered by their ID and when all the elements of the set are received a new output element with the same ID is created and send





(Slide 9)

3. Identification of the DS:

   The order is not guaranteed in the processing of the data stream

   Join of the DSs (Multiple-Producer) (Slide 9 (2))
   a. In the creation of the data stream is added an identifier to the structure

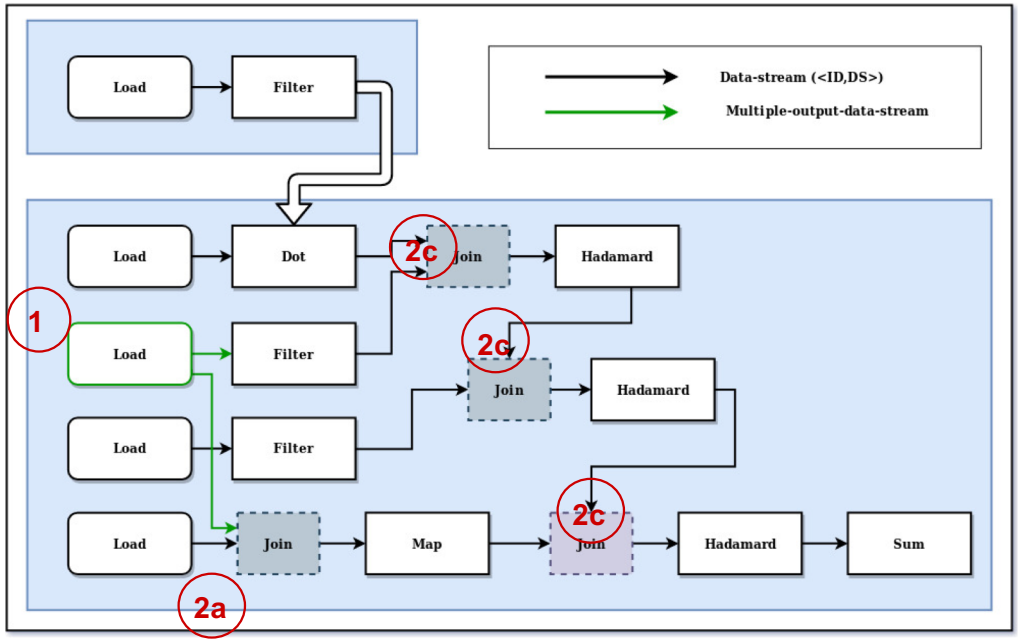
4. Guarantee correct memory management:

Especially in the case of Multiple-Consumers (Slides 5, 9)
Only last consumer frees memory

a. A counter is added to the structure of each data-stream

b. A removal function is made available

c. Operations on the counter are atomic



Data structure to represent the channel: Non-blocking data structures (Shared Memory)

- Wait-free
  - Every concurrent operation is guaranteed to be finished in a finite number of steps.
  - No efficient implementations were found, its implementation usually implies a large overhead
- Lock-free
  - Some concurrent operations are guaranteed to be finished in a finite number of steps
  - Too many concurrent accesses degrades performance
- Obstruction-free
  - A concurrent operation is guaranteed to be finished in a finite number of steps
  - If another concurrent operation interferes there is no guarantee to progress when multiple threads execute concurrently

Lock-free data structures will be a better choice to optimize the latency of a system (If consumers are fed properly).

X

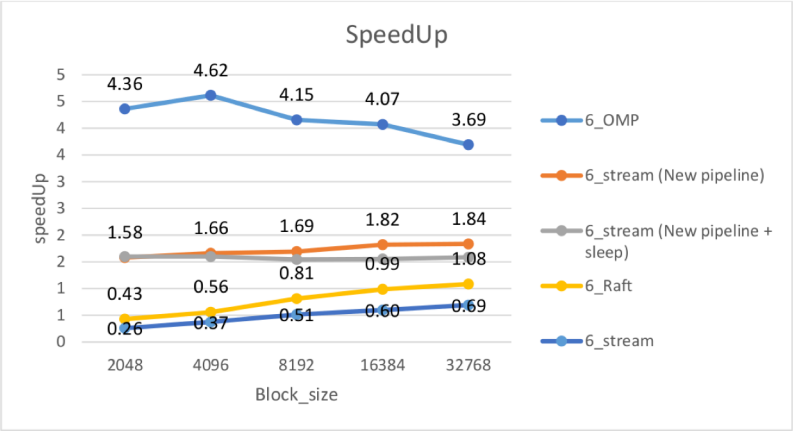
Decision: to implement normal lock

Lockfree

1. Advantages
    a. Enables the implementation of MIMO
    b. Provides better performance than blocking structures if consumers are well fed
2. Disadvantages
    a. Adding and removing elements may fail
        i. This provides active waits if the thread has no more tasks to perform (Task Scheduler - HEP-Frame)
    b. The size of the structure should be weighted
        i. Memory allocation is not lock-free, choosing to have a dynamic allocation can lead to standby situations
        ii. Fixed size can lead to waits when the data structure is full

Performance analysis of data stream implementation

1. Reasons for OMP to perform better than the stream approach:
   a. Communication management
      i. Communication is significantly overloaded, especially because of active waiting.
   b. Better use of the temporary locality
      i. In the case of OMP, the generated data is processed by the same thread, which allows a better use of the temporal location of the data.
   c. Poor management of resources
      i. As there is no management of threads and tasks, the use of available resources is very poor.



Perf report Q6_stream	without lock	with lock
method	Overhead	Overhead
lockfree::pop	23%	9%
free	5%	8%
malloc	4%	5%

2. Improvements that can be made in the current version data stream
    a. Scheduling data loads (DS in HEP-Frame)
        i. Current HEP-Frame schedules a single DS to define a shared structure
        ii. The stream approach requires several concurrent DS (& queues) shared by few consumer threads
    b. Free the memory once datablocks are consumed
        i. Single consumer: straight!
        ii. Multiple consumers: last consumer frees the memory
    c. Operation outputs that are null — Already done!
        i. The size of the vectors is taken into account, but the blocks continue to be transmitted
    d. Merge of join-boxes with following operations
        i. Reduce the nº or running threads
    e. Management queues (data structures): lock-free or …?
        i. Test to see which approach has better performance

Other aspects to consider in the LA system

1. File reading can be improved for the data stream version
   a. Is the reading not done sequentially, constantly implying the opening of the same file, has an impact on performance?
   b. Reading the DS of the same file in parallel? (Slide 4)
2. Weight of constant allocation and memory release has a relevant performance impact
   a. is it possible to reuse some blocks to avoid allocation and release of memory?

07-fev-19

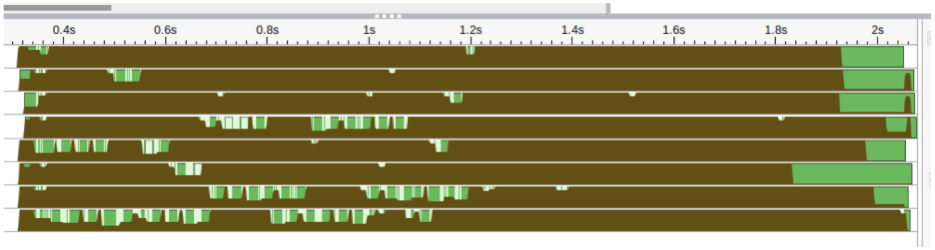
Intel Vtune

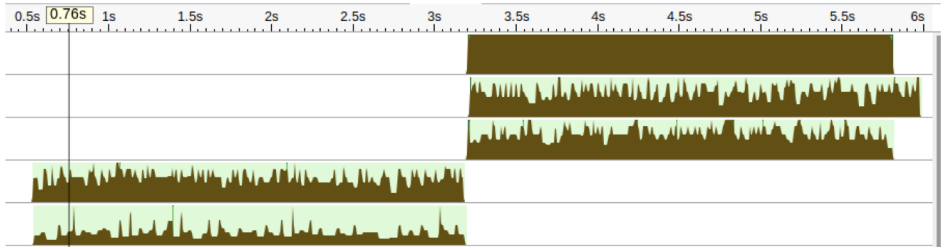
- Hotspots
  - Analysis of the execution of the functions/code
- Threading
  - Analysis of threads execution, synchronization and waiting time
- Microarchitecture
  - Analysis of CPU and Memory bottlenecks
- Memory Accesses
  - Analysis of accesses to memory at various levels, cache, main memory, NUMA distribution
- Other important work analyzes
  - Memory consumption, HPC Performance, I/O Disk

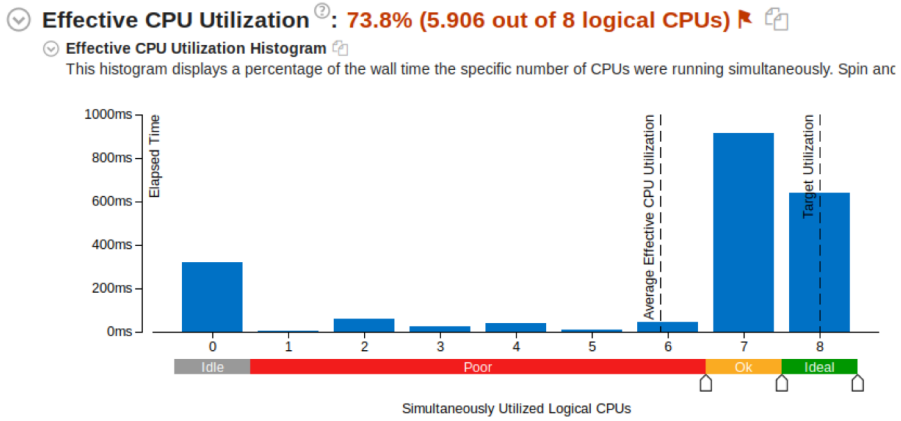
Vtune Performance Analysis (Laptop)

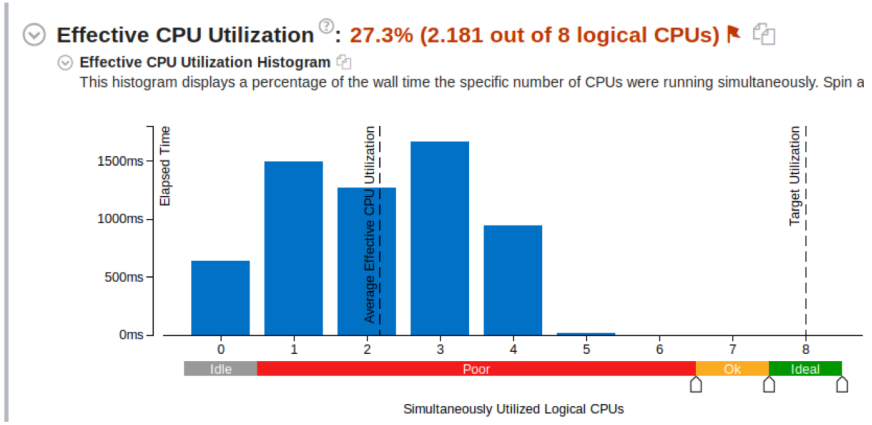
28 threads
12 low overhead manag/

- Thread performance
  - OMP(8 threads) vs Stream Approach(28 threads), More tests will be needed (cluster-SSD/HDD)







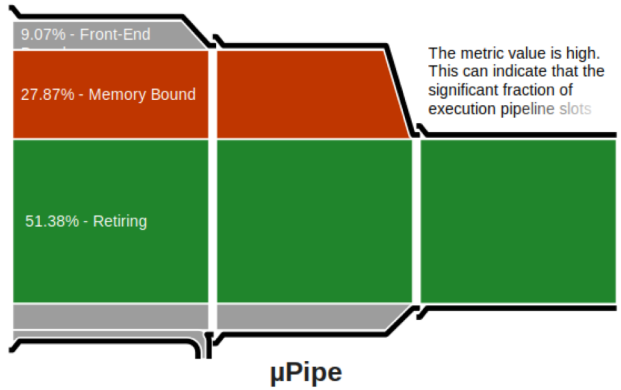


Microarchitecture Exploration

- Memory Bound
  - improvement in memory utilization
  - However, the performance of the stream version is still lower than the OpenMP version.
  - The high percentage of retired instructions is partly due to the use of atomic variables

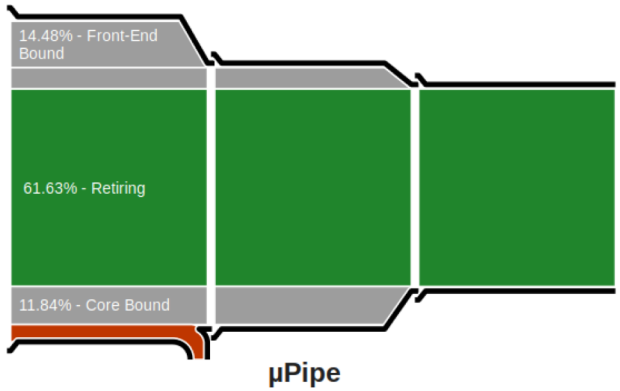
OpenMP version

Elapsed Time: 197.448s		
Clockticks:	33,732,400,000	
Instructions Retired:	34,325,200,000	
CPI Rate:	0.983	
Retiring:	51.4%	of Pipeline Slots
Front-End Bound:	9.1%	of Pipeline Slots
Bad Speculation:	3.5%	of Pipeline Slots
Back-End Bound:	36.1%	of Pipeline Slots
Memory Bound:	27.9%	of Pipeline Slots
Core Bound:	8.2%	of Pipeline Slots
Total Thread Count:	200	
Paused Time:	156.220s	



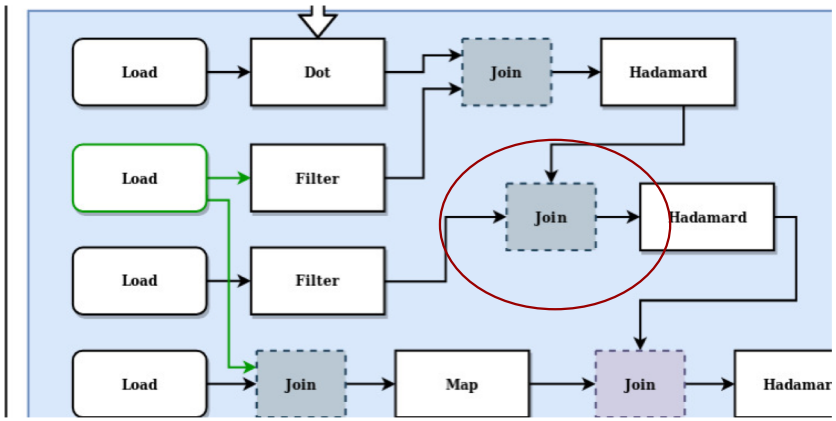
Stream version

Elapsed Time: 324.263s		
Clockticks:	31,406,960,000	
Instructions Retired:	63,006,320,000	
CPI Rate:	0.498	
Retiring:	61.6%	of Pipeline Slots
Front-End Bound:	14.5%	of Pipeline Slots
Bad Speculation:	5.4%	of Pipeline Slots
Branch Mispredict:	5.2%	of Pipeline Slots
Machine Clears:	0.2%	of Pipeline Slots
Back-End Bound:	18.5%	of Pipeline Slots
Memory Bound:	6.7%	of Pipeline Slots
Core Bound:	11.8%	of Pipeline Slots
Total Thread Count:	440	
Paused Time:	189.578s	



Join/lockfree Impact

- The join operation takes 22% to 40% of the runtime. Too long!!!
- Lockfree mechanisms cause active waits that occupy the CPU with useless work. (The reordering can reduce this problem)



8GB SSD

Function Stack	CPU Time: Total ▼ Effective Time by Utilization (Idle Poor Ok Ideal Over)
func@0x11b50	99.9%
Filter_DecimalVectorBlock::operation	25.1%
stream::Producer<std::tuple<engine::F	21.9%
stream::Join<engine::FilteredBitVect	21.9%
Map::operation	13.1%
stream::Producer<engine::DecimalVec	9.2%
Hadamard_FilteredBitVectorBlock::ope	7.8%
Dot_BitmapBlock::operation	6.1%
Hadamard_DecimalVectorBlock::opera	4.0%
stream::Producer<std::tuple<engine::F	3.9%
Sum::operation	3.2%
stream::Producer<engine::BitmapBloc	2.7%
stream::Producer<std::tuple<engine::D	2.4%

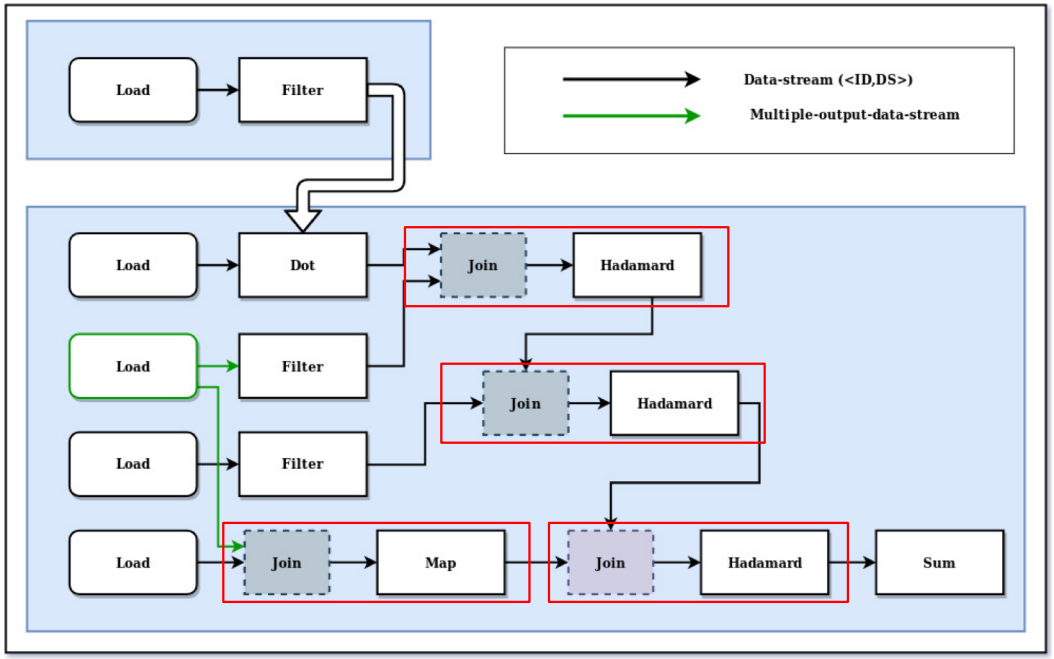
8GB HDD

Function Stack	CPU Time: Total Effective Time by Utilization ▼ (Idle Poor Ok Ideal Over)
Total	100.0%
clone	93.0%
start_thread	93.0%
func@0x11b50	93.0%
stream::Producer<std::tuple<	40.4%
stream::Join<engine::Filter	40.4%
boost::lockfree::queue<s	21.7%
stream::Consumer<engi	11.4%
boost::atomics::atomic<b	2.5%
boost::atomics::atomic<b	2.4%
boost::atomics::detail::ba	0.2%
Filter_DecimalVectorBlock::o	11.6%
Hadamard_FilteredBitVectorB	11.5%
Dot_BitmapBlock::operation	7.5%

Improvements to the current code

- Joining operations
  - Decreased communication and synchronization mechanisms, increased work done by the thread
  - Decrease in the number of total threads

- Replacement of lockfree mechanisms by normal locking mechanisms
  - (Slide 22)



Improvements to the current code (Cont...)

- Reuse of vectors/blocks
  - For each operation there is always an allocation operation and a memory release operation, in certain cases these operations can be avoided

Function / Call Stack	CPU Time ▼	Module
__memmove_avx_unaligned_erms	1.128s	libc.so.6
__GI_	1.090s	libc.so.6
↖ __gnu_cxx::new_allocator<double>::deall	0.822s	q6_stream
↖ __gnu_cxx::new_allocator<double>::deall	0.268s	q6_stream
operator new	0.826s	libstdc++.so.6
std::__fill_n_a<unsigned long*, unsigned long,	0.608s	q6_stream
engine::dot	0.550s	q6_stream
boost::lockfree::queue<stream::Data_Stream_	0.486s	q6_stream
std::__fill_n_a<double*, unsigned long, double	0.424s	q6_stream
boost::lockfree::queue<stream::Data_Stream_	0.392s	q6_stream

- Reading the file blocks
  - The reading of the files does not present inefficiencies, contrary to what was considered Slide 19(1):
    1. File reading can be improved for the data stream version
       a. Is the reading not done sequentially, constantly implying the opening of the same file, has an impact on performance?

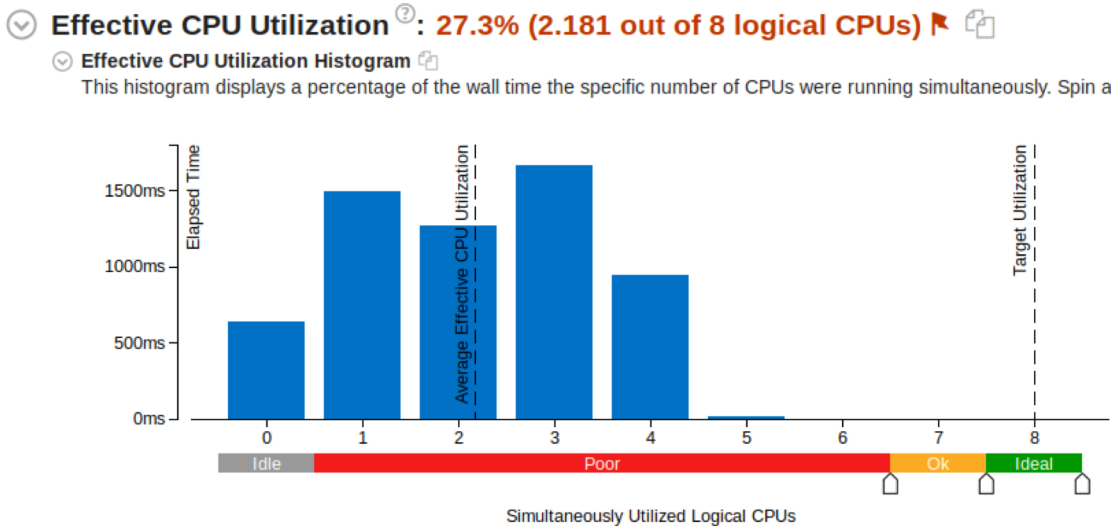
Implementation in HEP-Frame

- In theory, it is already possible to implement a preliminary version
  - Operations are introduced as propositions
  - The communication between the propositions is made by the stream system
- Treatment of loads
  - Improve the current DS to perform a pipeline of loads **Not yet!**
  - Aggregate loads with the data processing propositions **No!**
  - Introduction of dependencies and priorities **???**
- New level of dependence between propositions
  - Splitting the pipeline into two, creating two tables? **No need...**
- Communication between propositions
  - Implementation of MPSC mechanisms in the HEP-Frame
  - Memory control ensured by stream implementation

Changes made to the LAQ Stream

14-fev-19

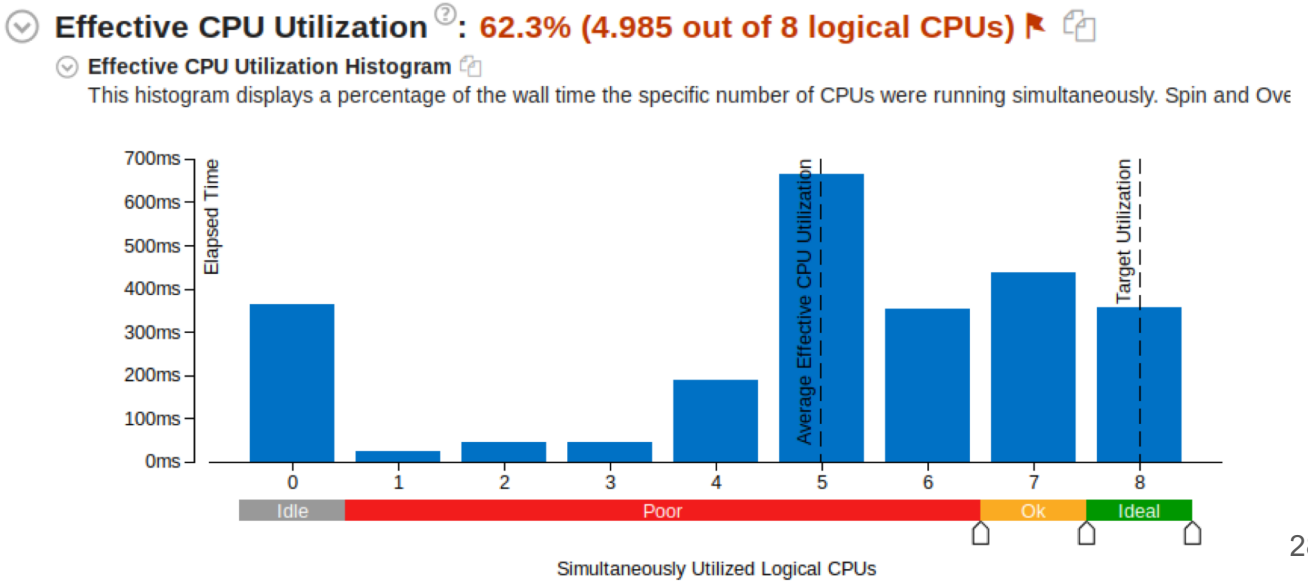
- Merge of join-operations
- Memory control adjustment (due to multithreading)
- Multithreading of operations
  - Parallel DS (Producers)
  - Parallel DP (Consumer/Join-Consumer & Producers)
  - In specific cases, user intervention is required (Sum operation)
- Still missing
  - Bug fix
  - Implementation with basic locks
  - Tests in the cluster



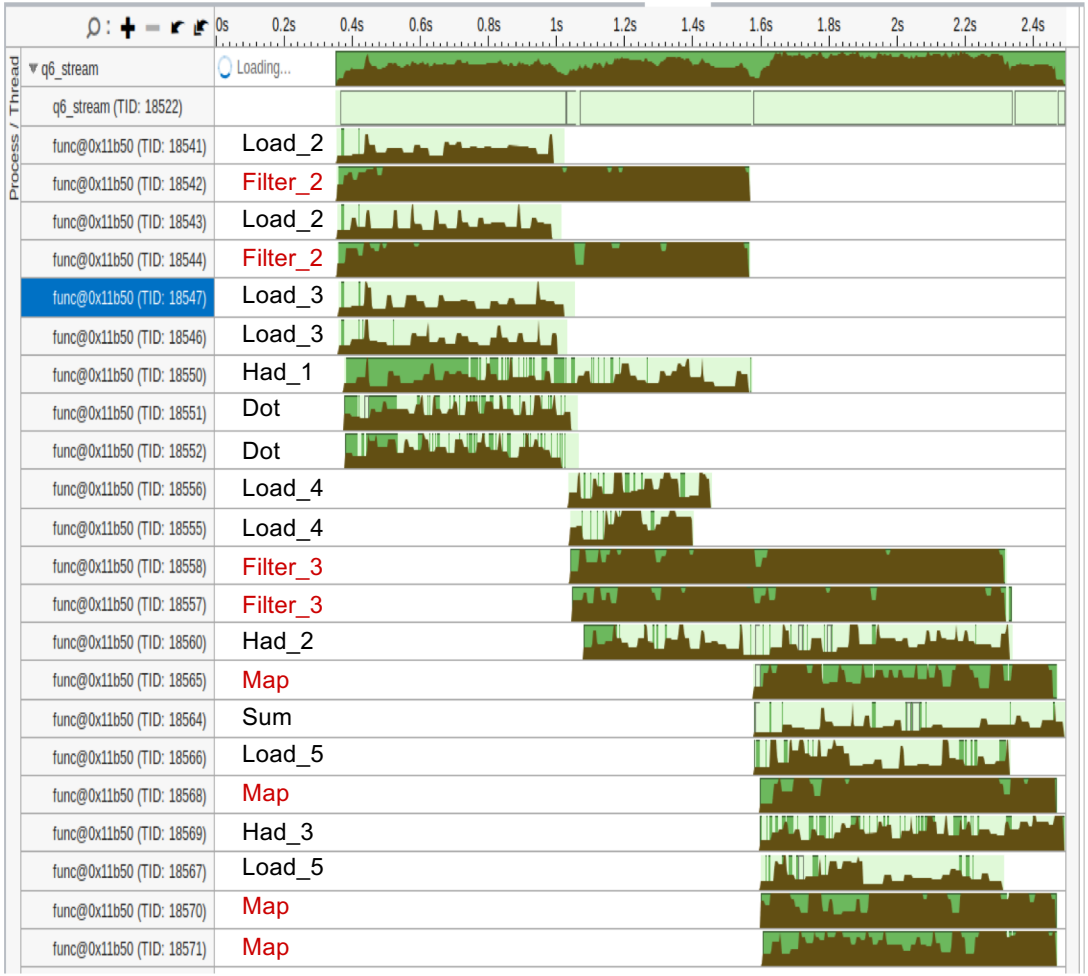
Tests performed on the Laptop (TPC-H 8 GiB data set):

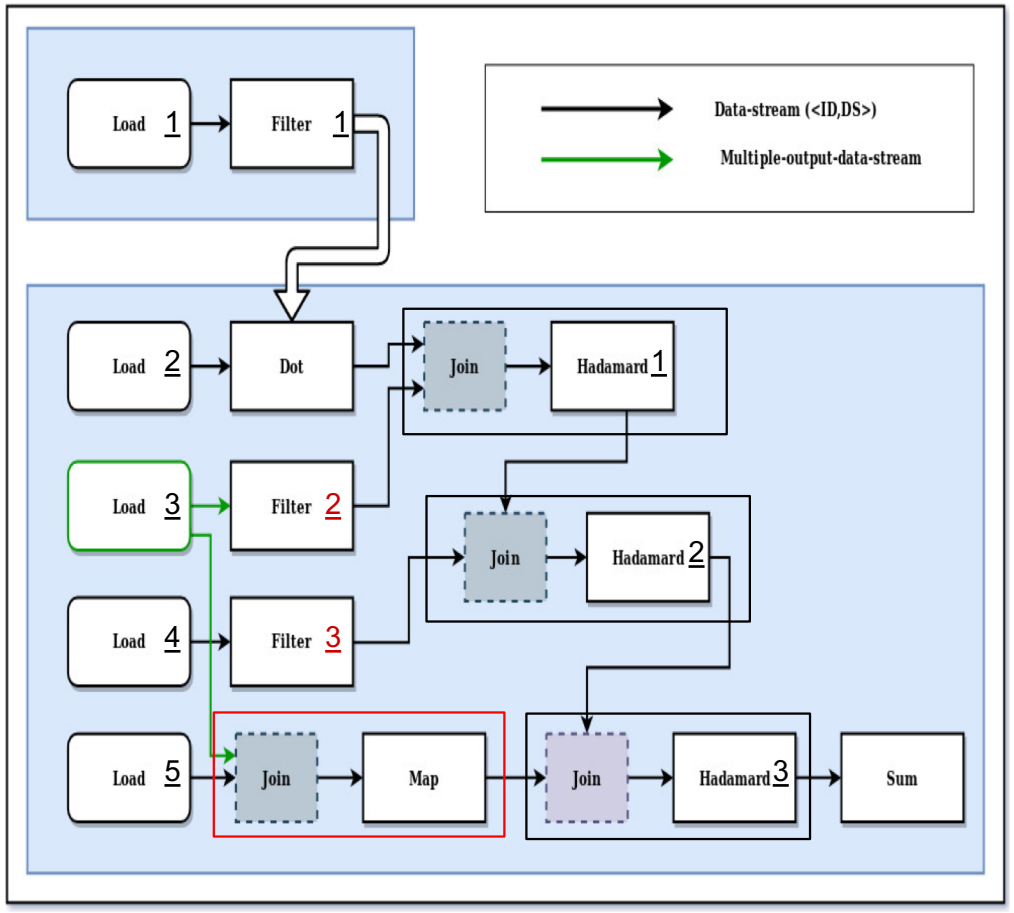
- Old version (Sequential Stream)
  - Without the mix of joins

- New version
  - Execution with predefined multithread, up to 4 threads per proposition



Execution of threads per operation



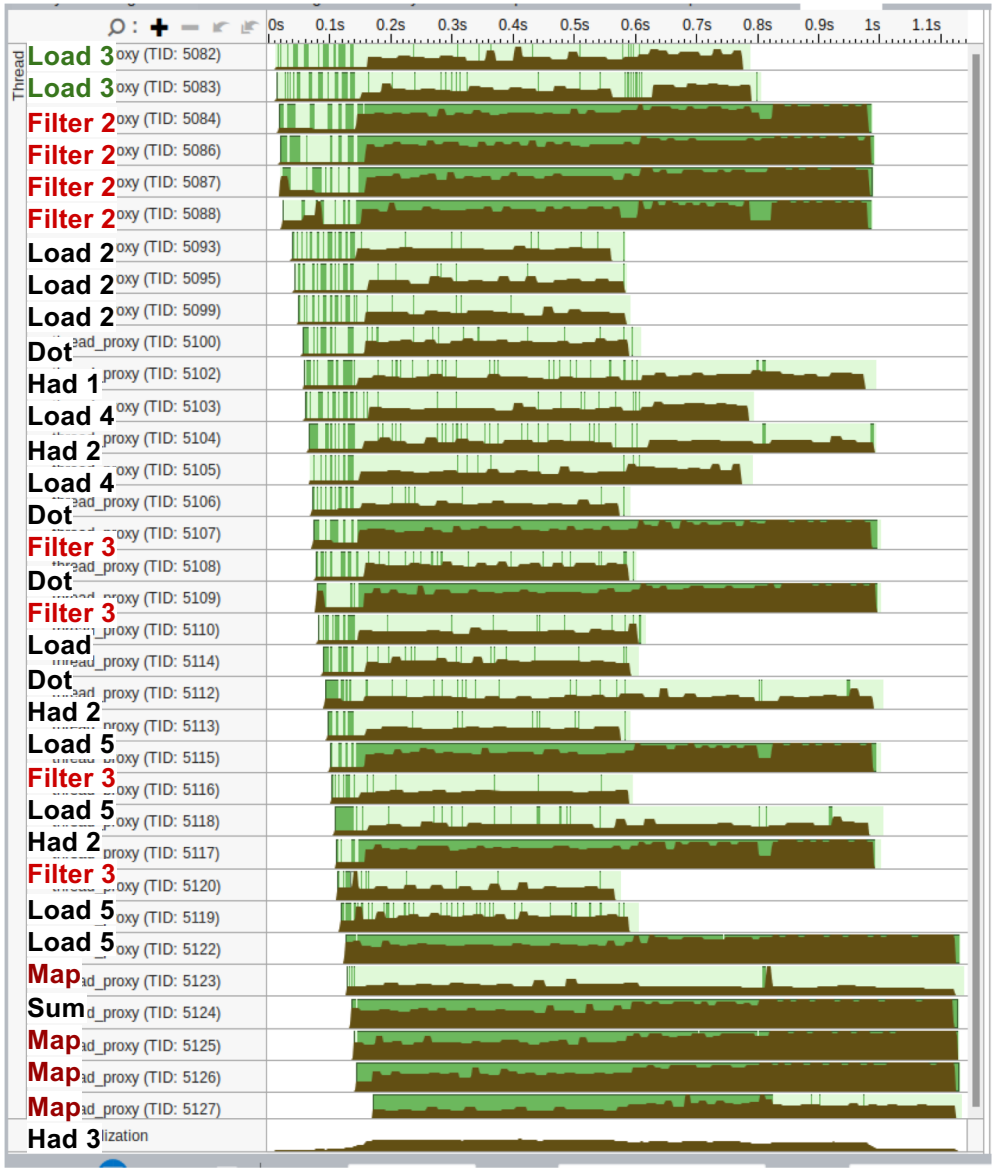


2 Heavy processing!

21-fev-19

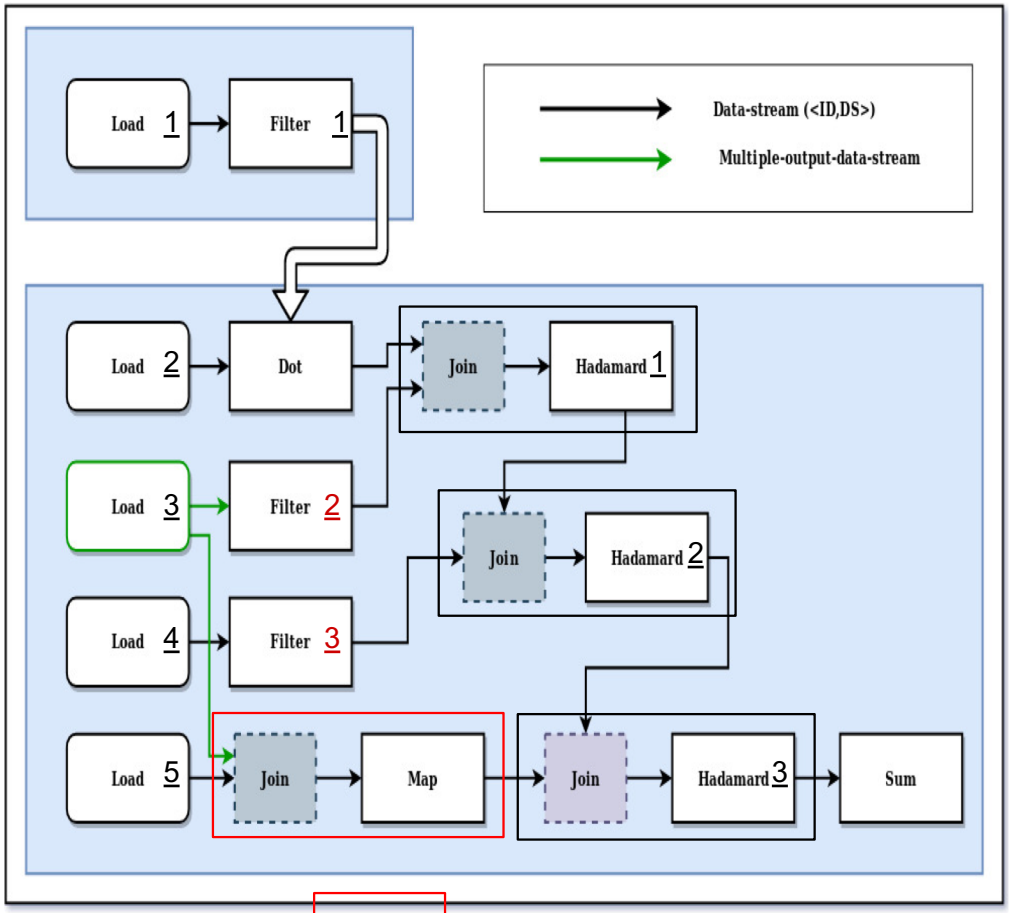
Implementation bugs in the stream approach

- LockFree-queue; diagnostic:
  - A LockFree queue does not support multiple consumers of the same product
  - Parallel execution of *pop function* may return the same value multiple times (*multiple consumers*)
    - causing pb's in memory management (multiple frees)
    - introducing errors in the result
- Solution to the LockFree-queue bug:
  - Find a lockfree solution that supports access by multiple consumers probably not possible...
  - Use a thread to distribute the work ...probably inefficient...
  - Replace LockFree-queue by a buffer; memory managed by producer ...ok; proposal?...
- Join Operations; diagnostic:
  - The structures used to aggregate stream elements are not thread safe
    - The same stream element is processed multiple times by different threads
- Join Operations bug; solution:
  - Control access to structures with locks



Tests performed on node 662 (2x 12-cores):

- TPC-H 4GiB, Block_Size=32768
- Intel compiler (flag -O2)



Impact of memory allocation and release

Function / Call Stack	CPU Time ▼ Effective Time
operator new	27.6%
_int_free	17.4%
__intel_ssse3_rep_memmove	12.4%
std::__fill_n_a<unsigned long*, unsign	10.8%
std::__fill_n_a<double*, unsigned long	8.4%
engine::dot	3.4%
std::vector<double, std::allocator<dou	2.5%
engine::filter	2.5%
std::basic_streambuf<char, std::char_	1.4%
engine::krao	1.3%
engine::DecimalVectorBlock::load	1.3%
free	1.2%
engine::lift	1.2%
_intel_fast_memmove	0.9%
operator delete	0.5%
__gnu_cxx::new_allocator<double>::a	0.5%
std::basic_streambuf<char, std::char_	0.5%
_intel_fast_memmove	0.5%
engine::BitmapBlock::load	0.4%
filter_var_b	0.3%
Selected 2 rows:	45.0%

Streaming

Function / Call Stack	Effective Time ▼
operator new	33.0%
_int_free	21.8%
__intel_ssse3_rep_memmove	9.0%
std::__fill_n_a<unsigned long*, unsigned long, uns	3.3%
↖ std::fill_n<unsigned long*, unsigned long, uns	3.3%
↖ engine::FilteredBitVectorBlock::FilteredBitVe	2.9%
↖ engine::FilteredDecimalVectorBlock::Filtere	0.3%
↖ engine::BitmapBlock::BitmapBlock ← engin	0.2%
engine::filter	2.8%
engine::dot	2.5%
std::vector<double, std::allocator<double>>::vecto	2.3%
free	2.2%
std::__fill_n_a<double*, unsigned long, double>	1.4%
engine::lift	1.2%
engine::krao	1.2%
engine::DecimalVectorBlock::load	1.2%
std::basic_streambuf<char, std::char_traits<char>>	0.8%
std::__copy_move<(bool)0, (bool)1, std::random_a	0.8%
_intel_fast_memmove	0.8%
Selected 2 rows:	54.9%

OpenMP

26-fev-19

Important points (tests Vtune)

- Larger Blocks Improve Stream Performance (**why?**)
  - More tests are required
  - The JA-OMP version does not benefit from very large blocks (Slide 35; **128Ki?**)
- Comparison between lockfree version and basic locks
  - Lockfree may have some benefits because of changes made (block_size & join bugs)
  - Lockfree bugs have not yet been resolved (slide 30)
- Explore more in detail the vtune tool
  - Work done by individual cores
  - Identifying tasks through vtune macros
- Increase the complexity of memory management (Slide 32) ?
  - Control the memory size used by the application
  - Perform allocations and memory releases more efficiently

Tests performed on node 662 (24-cores):

- TPC-H 8GiB, Block_Size=131072
- Intel compiler (flag -O2)
- Hyperthreading

Elapsed Time: 1.222s

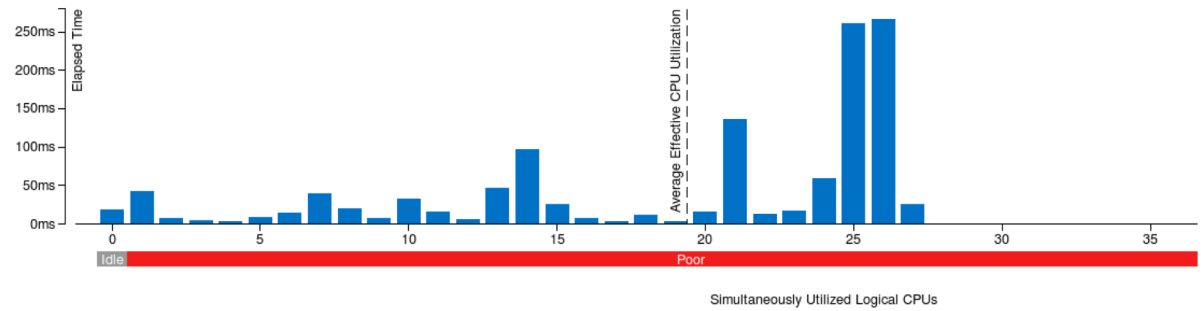
Paused Time: 0s

LockFree (31 worker-threads)

Effective CPU Utilization: 40.4% (19.385 out of 48 logical CPUs)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



Elapsed Time: 1.107s

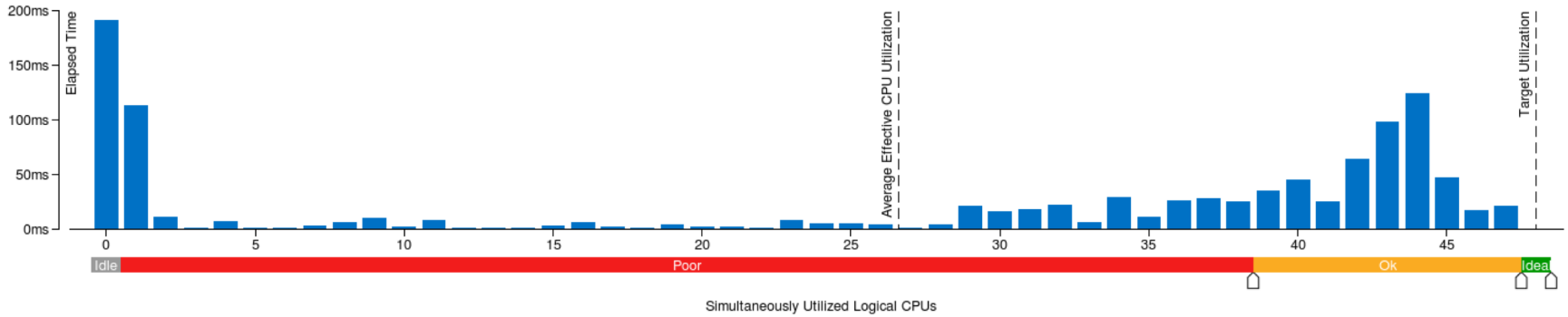
Paused Time: 0s

OMP (48 threads)

Effective CPU Utilization: 55.7% (26.724 out of 48 logical CPUs)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

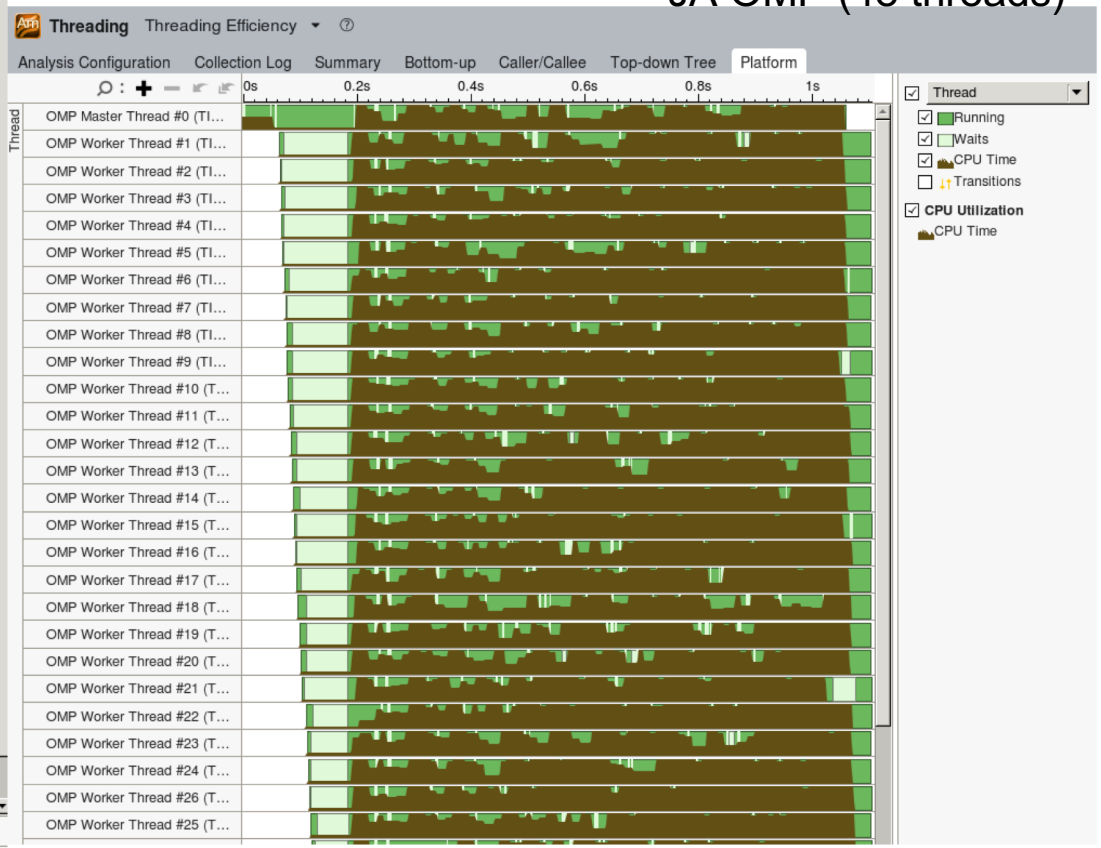


LockFree-queue (31 worker-threads)

Tests performed on node 662 (24-cores):

- TPC-H 8GiB, Block_Size=131072
- Intel compiler (flag -O2) vectorization?
- Hyperthreading try 1st w/out HT

JA OMP (48 threads)



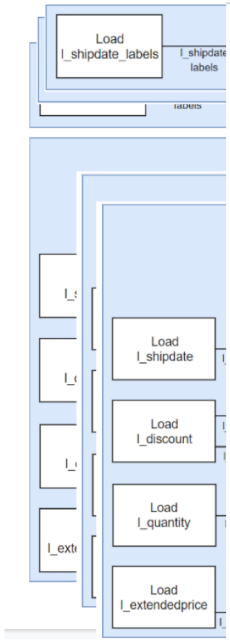
JA-OMP version

14-mar-19

- How operations are performed
  - Each block is one file and seems to correspond to one or more lines of each database table
  - Matrices (the database tables) are previously divided into blocks of predefined size, so each operation is applied to one or multiple blocks
  - Each thread runs the whole pipeline of operations in the `for` cycle block by block till last block
  - The processing of each set of blocks is sequential
  - It's necessary to read multiple blocks to complete the operations

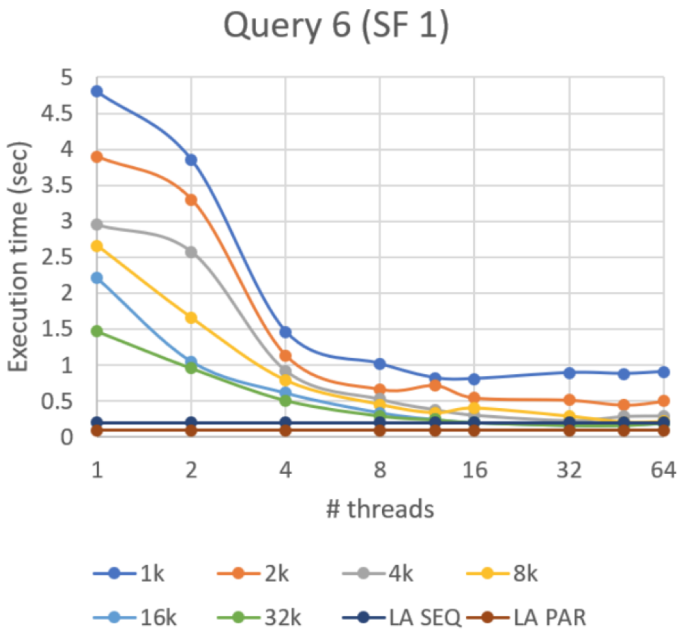
```
49   for (engine::Size i = 0; i < var_lineitem_shipdate->nLabelBlocks; ++i) {
50     var_lineitem_shipdate->loadLabelBlock(i);
51     filter(filter_var_a,{*(var_lineitem_shipdate->labels[i])},var_a_pred->b
52   }
53   #pragma omp parallel for
54   for(engine::Size i = 0; i < var_lineitem_shipdate->nBlocks; ++i) {
55         var_lineitem_shipdate->loadBlock(i);
56         dot(*var_a_pred, *(var_lineitem_shipdate->blocks[i]), var_a->blocks
57         var_lineitem_discount->loadBlock(i);
58         filter(filter_var_b,{*(var_lineitem_discount->blocks[i])},var_b->bl
59 /*Had*/ krao(*(var_a->blocks[i]),*(var_b->blocks[i]),var_c->blocks[i]);
60         var_lineitem_quantity->loadBlock(i);
61         filter(filter_var_d,{*(var_lineitem_quantity->blocks[i])},var_d->bl
62 /*Had*/ krao(*(var_c->blocks[i]), *(var_d->blocks[i]), var_e->blocks[i]);
63         var_lineitem_extendedprice->loadBlock(i);
64 /*MAP*/ lift(lift_var_f,{*(var_lineitem_extendedprice->blocks[i]),
65           *(var_lineitem_discount->blocks[i])
66             },var_f->blocks[i]);
67 /*Had*/ krao(*(var_e->blocks[i]), *(var_f->blocks[i]), var_g->blocks[i]);
68         sum(*var_g->blocks[i], var_h);
69   }
70
```

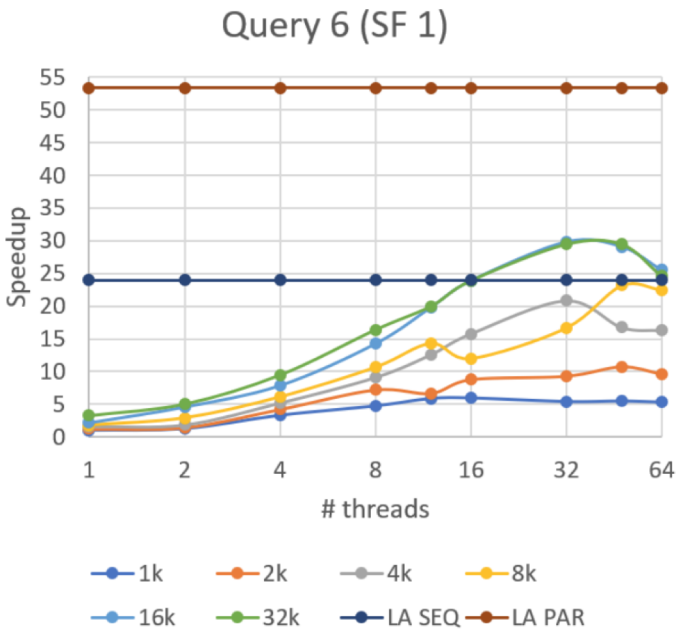
JA-OMP version *(taken from his slide set)*



Parallel Streaming

- Testing block sizes (#elements); relative to worst case (longest, 1k_elem)

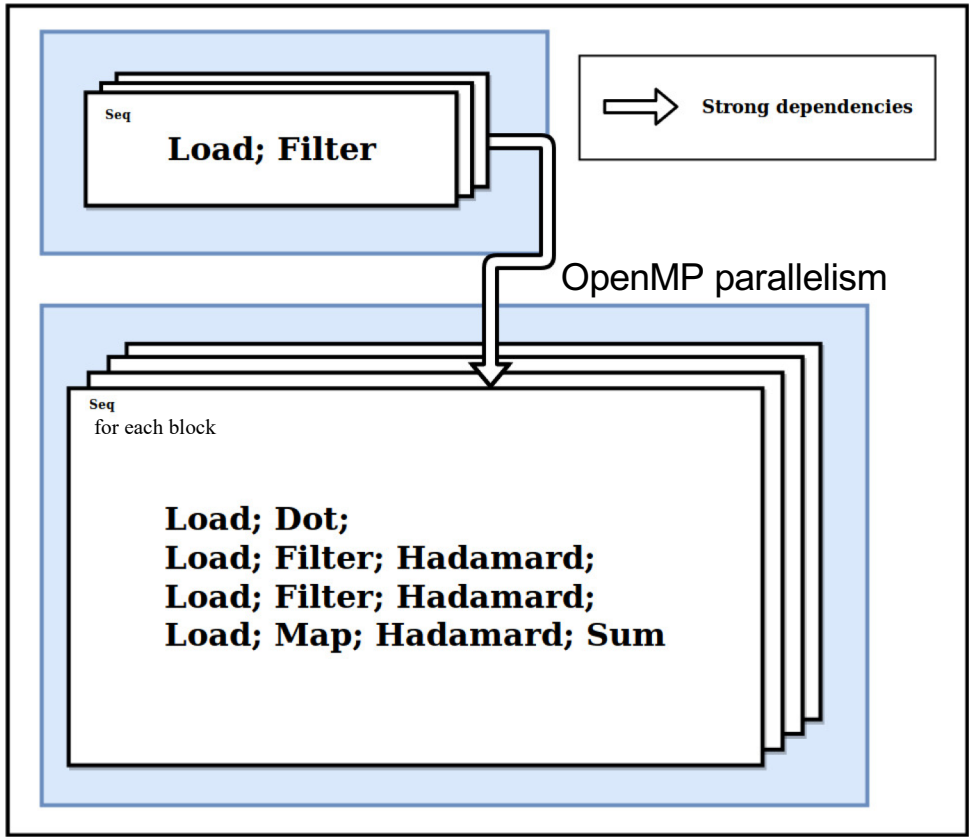




116

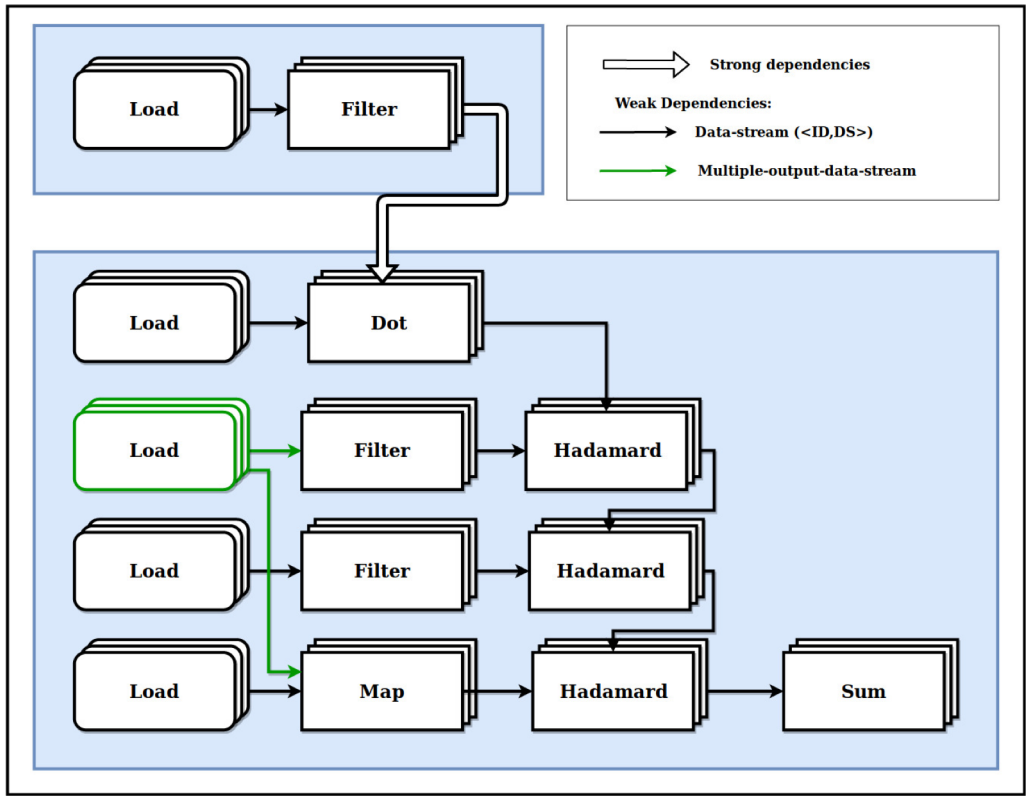
JA-OMP version

- Stream architecture
  - Since the data processing is done block by block this image represents a stream of data
  - Blocks are read as needed, sequentially at each iteration of the pipeline
  - The execution of a pipeline iteration is independent of the others
  - No communication structures are required: all required data per iteration are read in the execution of the iteration (except for strong dependencies)

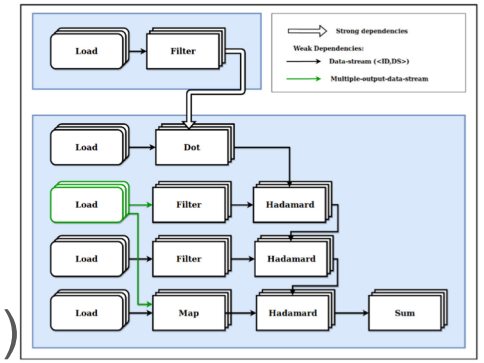


Stream version for HEP-Frame Q6

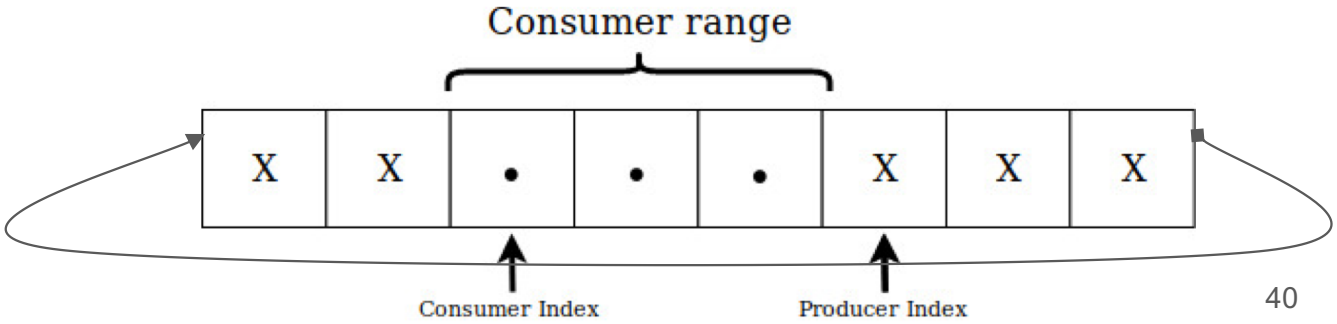
- How operations are performed
  - Each operation of LA has a degree of independence greater than the JA version
  - Data is being loaded & processed by block and results are used when needed
  - Each operation processes one block at a time as soon as it is available
  - Mechanisms and data structures to support the data flow between operations
  - All dependencies are respected



New version of communication channel



- Circular buffer (array of pointers) per consumer-op (with basic locks)
  - Supports multiple push & pop operations (producer & consumer)
  - Each element is removed from the array as soon as it is read by a consumer (consumer index update)
  - For now consumers are responsible for the release of memory -> this was improved!
  - For multi-consumer-op cases, each consumer-op owns its own array and the producer-op writes in both buffers and updates the producer index in both buffers
  - No bug found so far...



Analysis of vectorization reports

- The mechanisms & data structures to support data streaming between operations does not support vector computations
- LA operations may support vectorization since most calculations are on vectors
- More in slide 73

```
10  void filter(bool(*f)(std::vector<Decimal>),
11              const std::vector<DecimalVectorBlock>& in,
12              FilteredBitVectorBlock* out) {
13    std::vector<Decimal> v(in.size());
14    Size i, nnz = 0;
15    Size in_nnz = in[0].nnz, in_size = in.size();
16
17    for (i = 0; i < in_nnz; ++i) {
18      out->cols[i] = nnz;
19      for (Size j = 0; j < in_size; ++j) {
20        v[j] = in[j].values[i];
21      }
22      if ((*f)(v)) {
23        ++nnz;
24      }
25    }
26    out->cols[i] = nnz;
27    out->nnz = nnz;
28  }
```

SpeedUp & Block sizes with JA-OMP

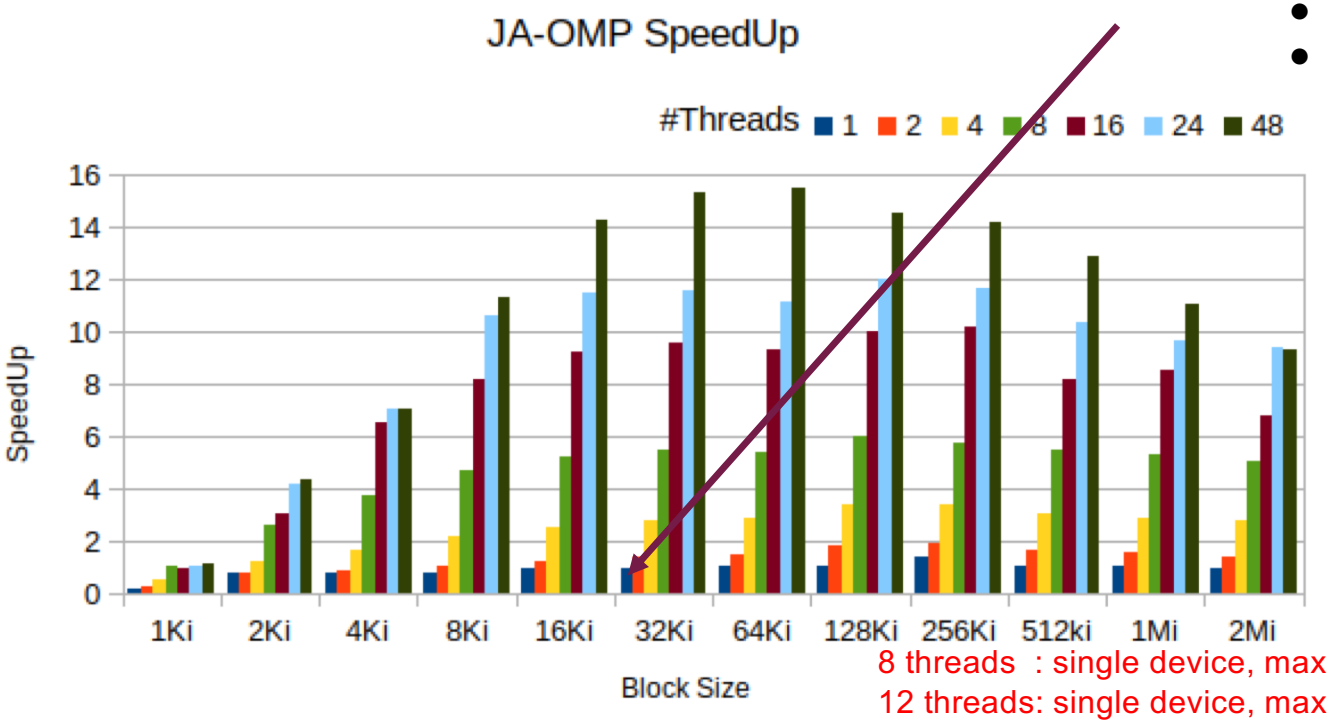
19-mar-19

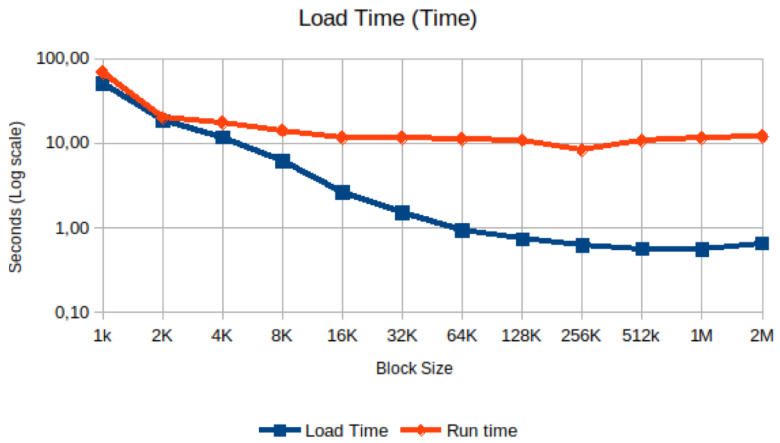
Block size: #elements ~~or #bytes?~~

Reference for SpeedUp: the sequential version Block size 32Ki

Tests performed on node 662 (12-cores per device):

- Two devices with HT
- TPC-H 8GiB
- Intel compiler (flag -O2)
- 3 best of 10 (5% max error interval)





8 threads : single device, max speedup 6

12 threads: single device, max speedup?

16 threads: one & half devices, max speedup 10

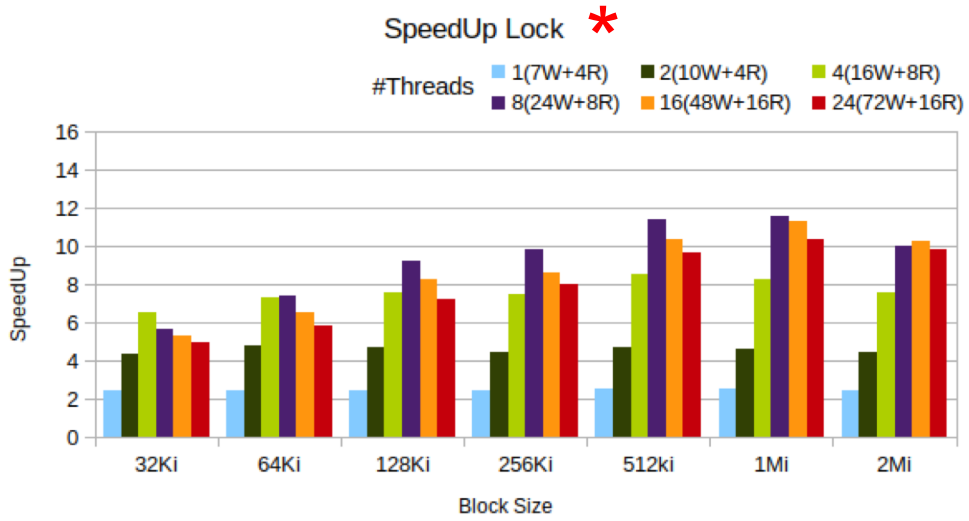
24 threads: all cores in two devices, no HT, max speedup 12

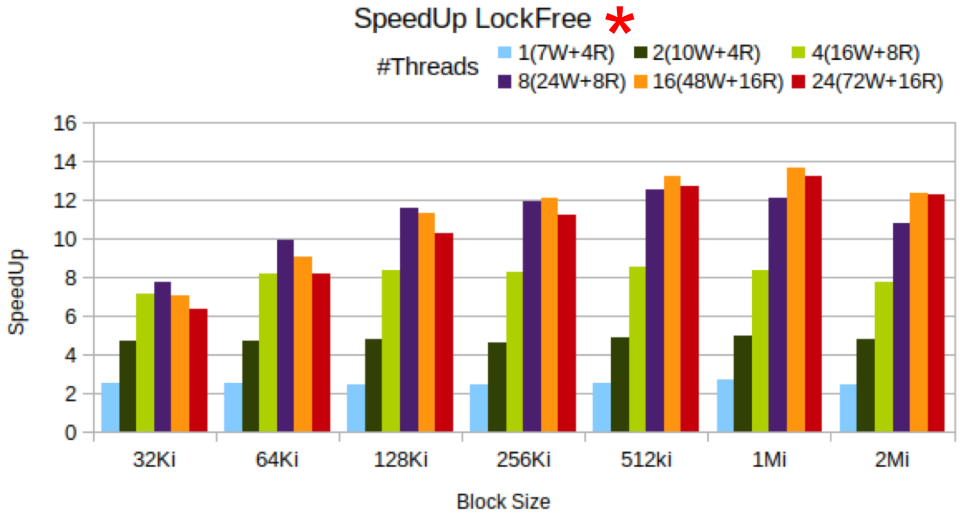
48 threads: all cores in two devices, with HT, max speedup <16

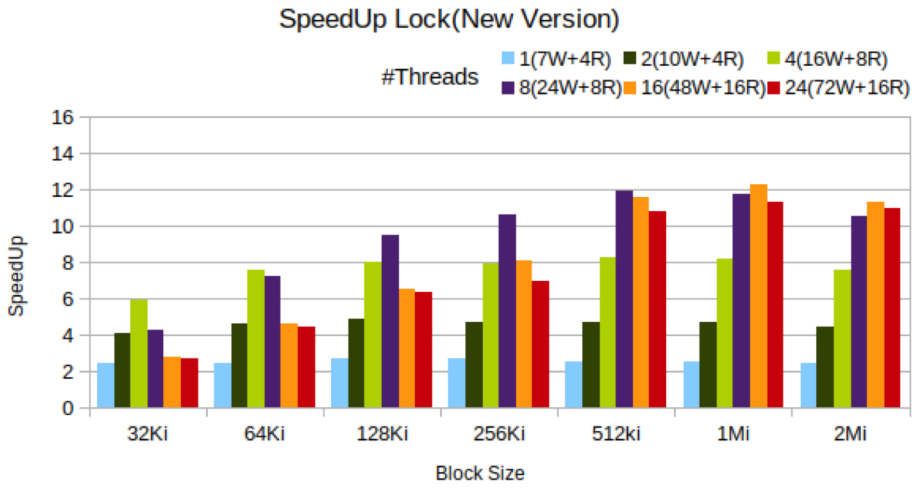
There is something wrong in this plot:

- speedup in vertical axis? No
- #threads? sequential

SpeedUp & Block sizes with Stream-version



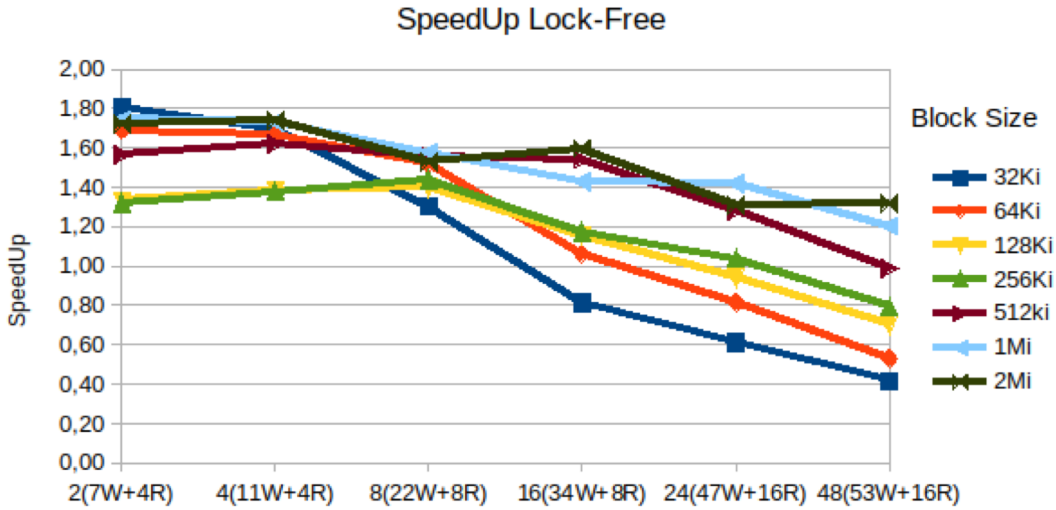


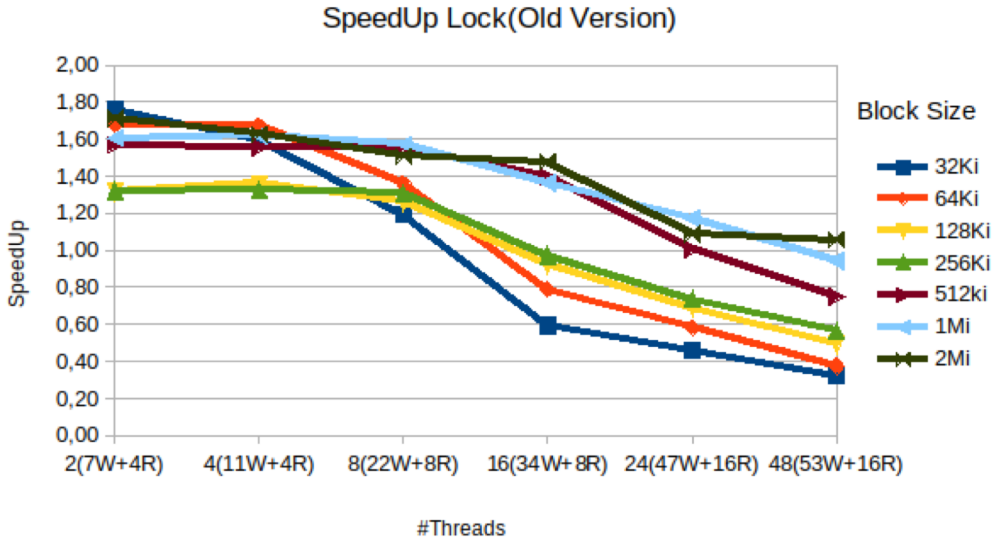


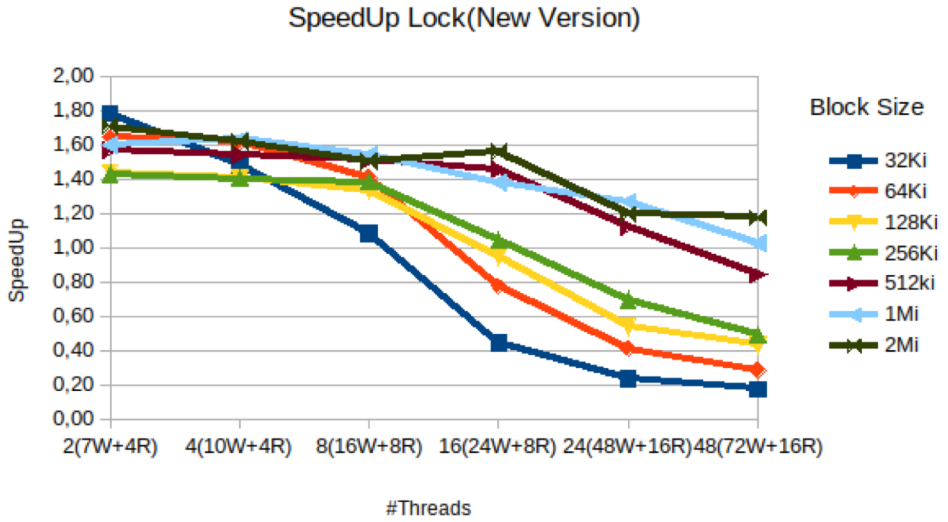
Reference for SpeedUp:
JA-OMP sequential version, Block size 32Ki

*These versions have bugs

Comparison between the implemented versions







Reference for SpeedUp: the corresponding JA-OMP version with corresponding #thread

2- (1*2Filter+1Map+3Had+1Dot+4Load)

4- (2*2Filter+2Map+3Had+2Dot+4Load)

8- (4*2Filter+4Map+2*3Had+4Dot+8Load)

16-(8*2Filter+8Map+2*3Had+4Dot+8Load)

24-(10*2Filter+10Map+3*3Had+8Dot+16Load)

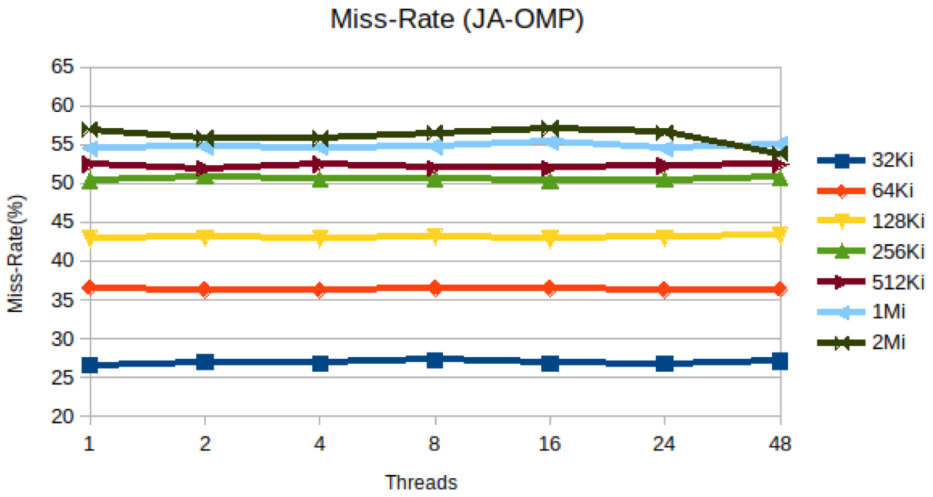
48-(12*2Filter+12Map+3*3Had+8Dot+16Load)

VTune analysis

- Instructions Retired
  - Number of instructions completely executed (Bad speculation does not count)
- Front-End Bound
  - Front-end is responsible for fetching operations that are executed later on by the Back-End
  - Front-End Bound is where no instructions are delivered while Back-End could have accepted them. For example, stalls due to instruction-cache misses
- Back-end Bound
  - The back-end is responsible for dispatching the instructions for their respective execution units

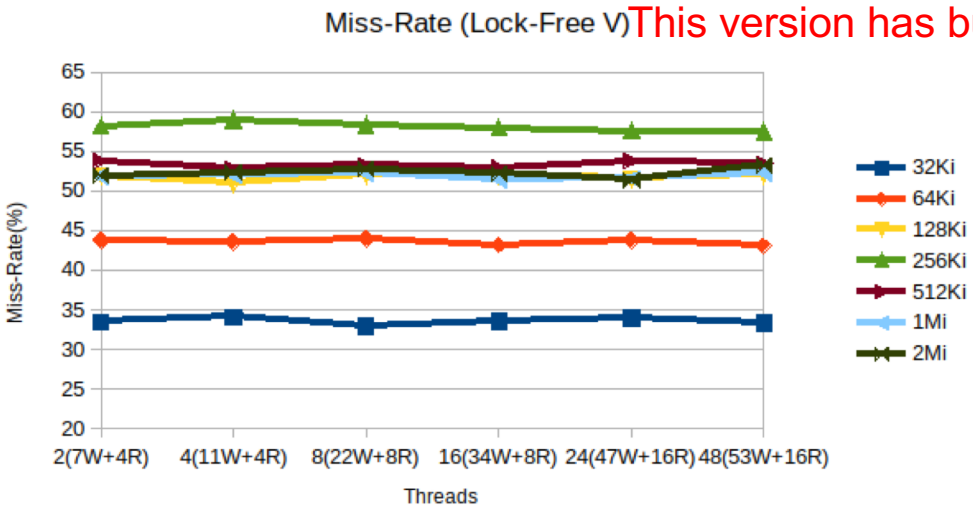
LLC cache misses

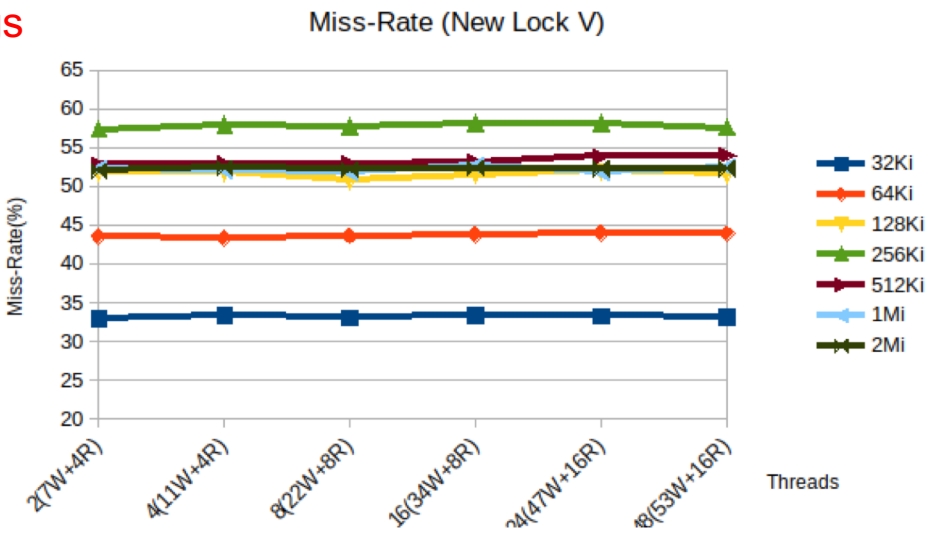
26-mar-19



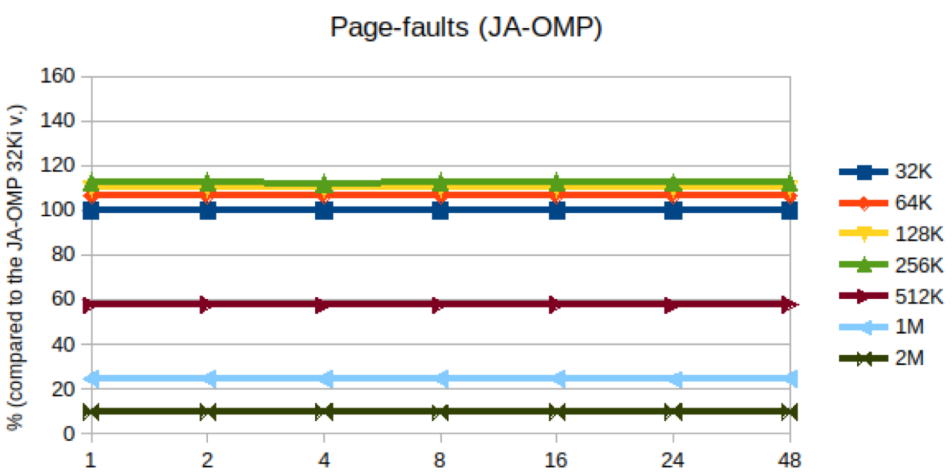
Tests performed on node 662 (12-cores per device):

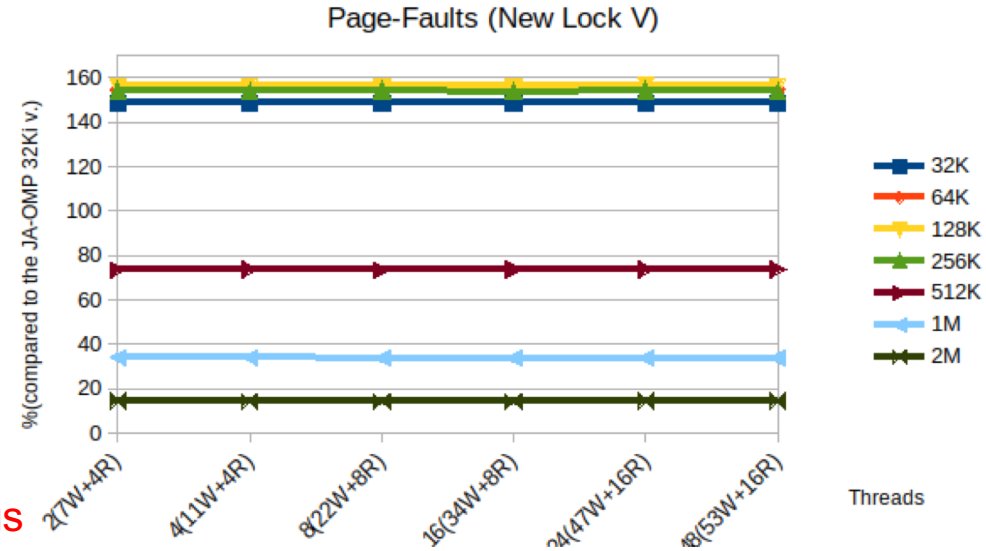
- Two devices with HT
- TPC-H 8GiB
- Intel compiler (flag -O2)
- 3 best of 10 (5% max error interval)
- Data collected with **Perf**



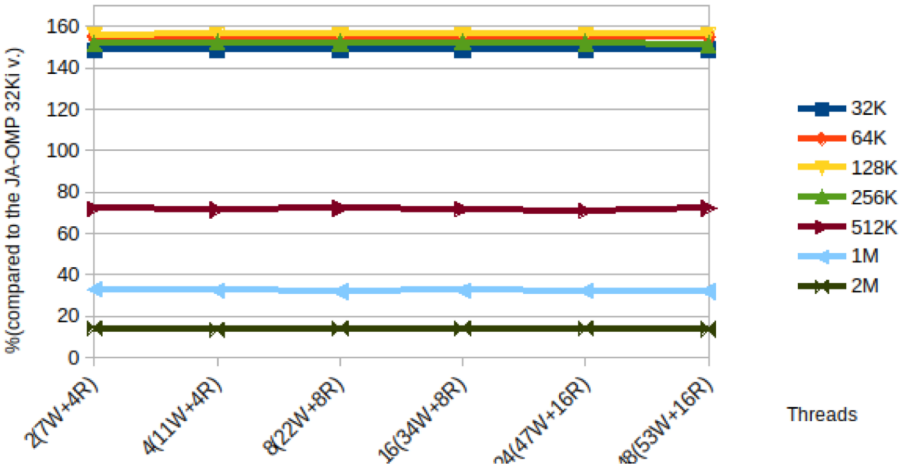


Page-faults





Page-Faults (Lock Free) This version has bugs



Reference: #thread

2- (1*2Filter+1Map+3Had+1Dot+4Load)

4- (2*2Filter+2Map+3Had+2Dot+4Load)

8- (4*2Filter+4Map+2*3Had+4Dot+8Load)

16-(8*2Filter+8Map+2*3Had+4Dot+8Load)

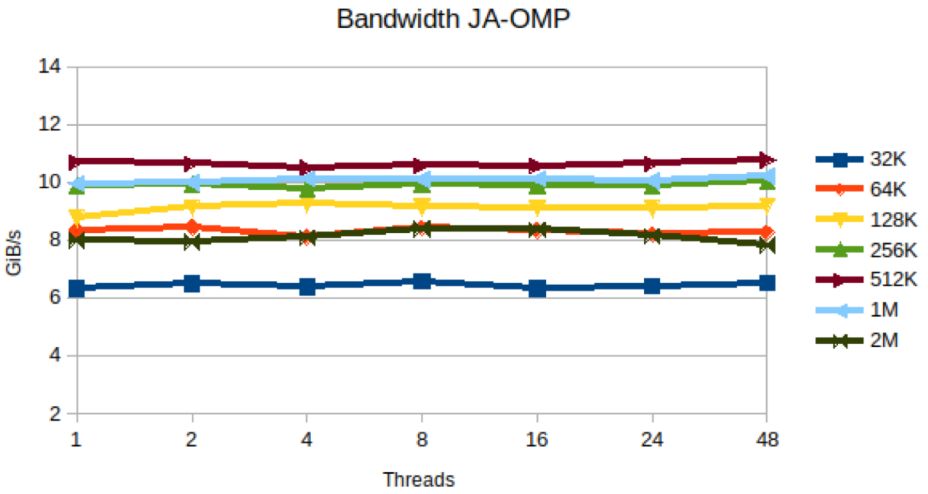
24-(10*2Filter+10Map+3*3Had+8Dot+16Load)

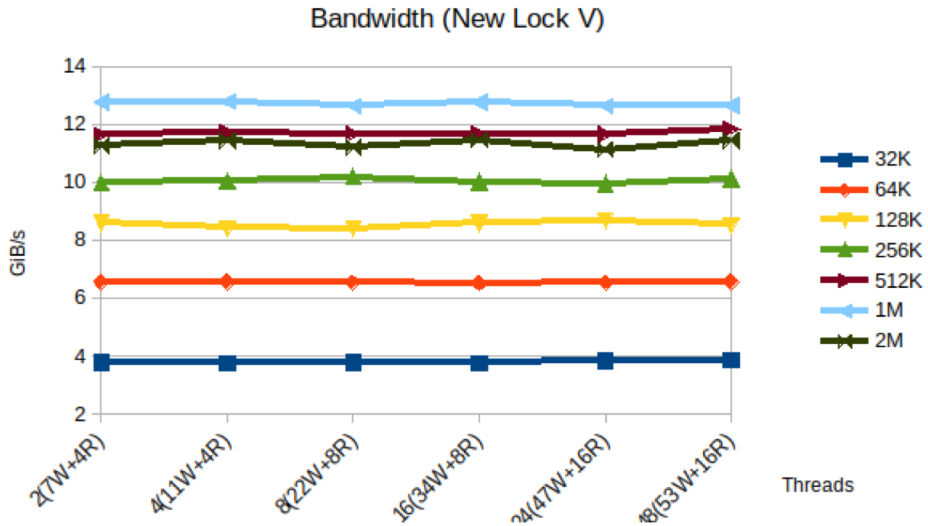
48-(12*2Filter+12Map+3*3Had+8Dot+16Load)

Bandwidth *(LLC misses)

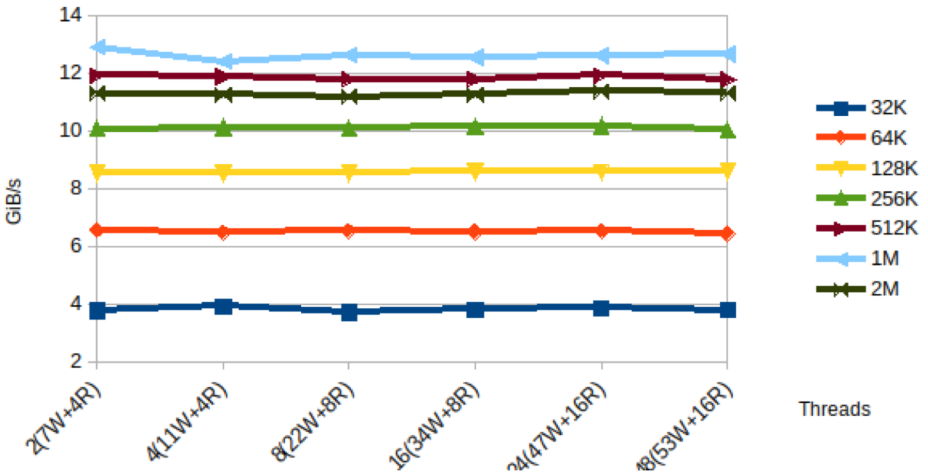
Tests performed on node 662 (12x2):

- TPC-H 8GiB
- Intel compiler (flag -O2)
- 3 best of 10 (5% max error interval)
- Data collected with **Perf**





Bandwith Lock-Free This version has bugs



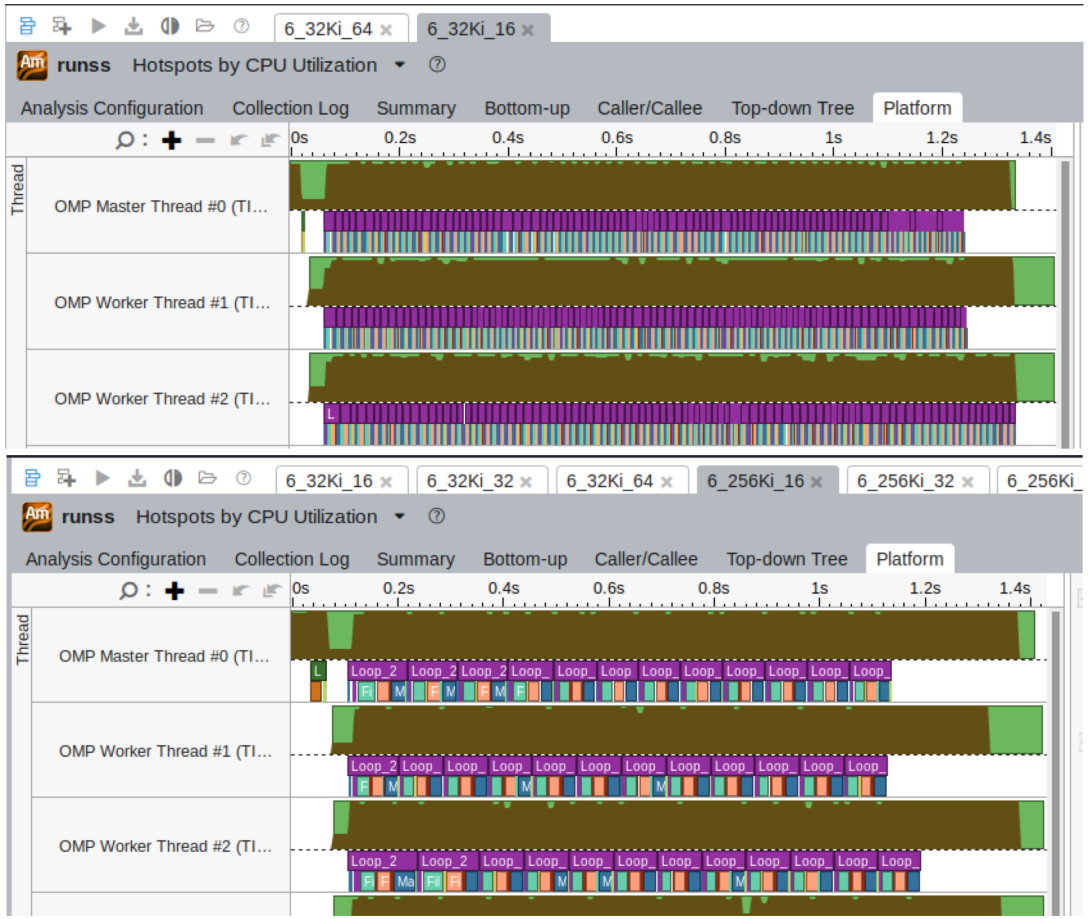
Reference: #thread

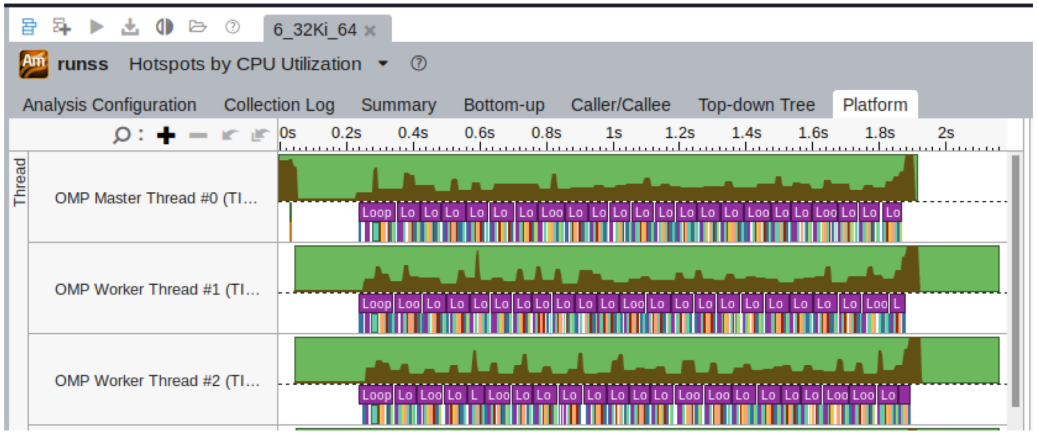
2- (1*2Filter+1Map+3Had+1Dot+4Load)
4- (2*2Filter+2Map+3Had+2Dot+4Load)
8- (4*2Filter+4Map+2*3Had+4Dot+8Load)
16-(8*2Filter+8Map+2*3Had+4Dot+8Load)
24-(10*2Filter+10Map+3*3Had+8Dot+16Load)
48-(12*2Filter+12Map+3*3Had+8Dot+16Load)

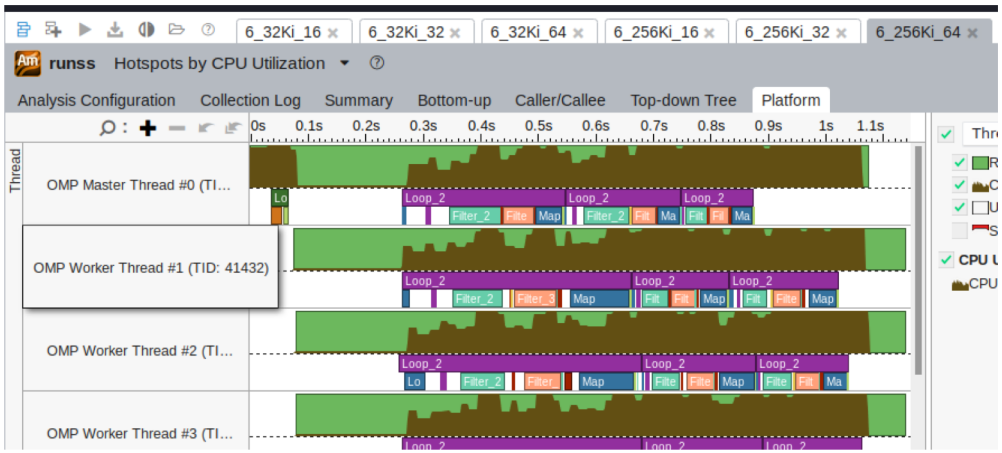
VTune JA-OMP

- Node 781 (2x 16-cores, HT)
- `icc -O2`
- TPC-H 8GiB

02-apr-19





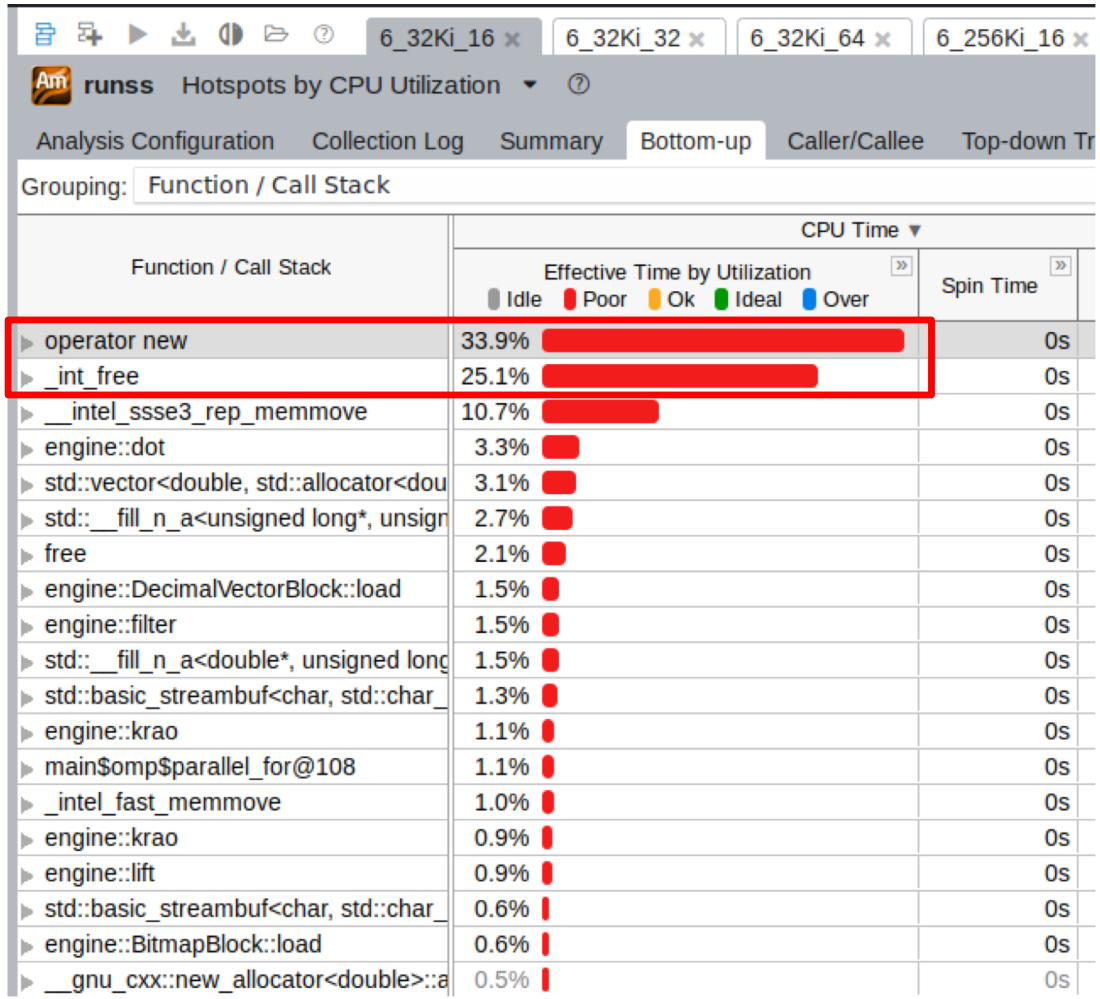


16 threads

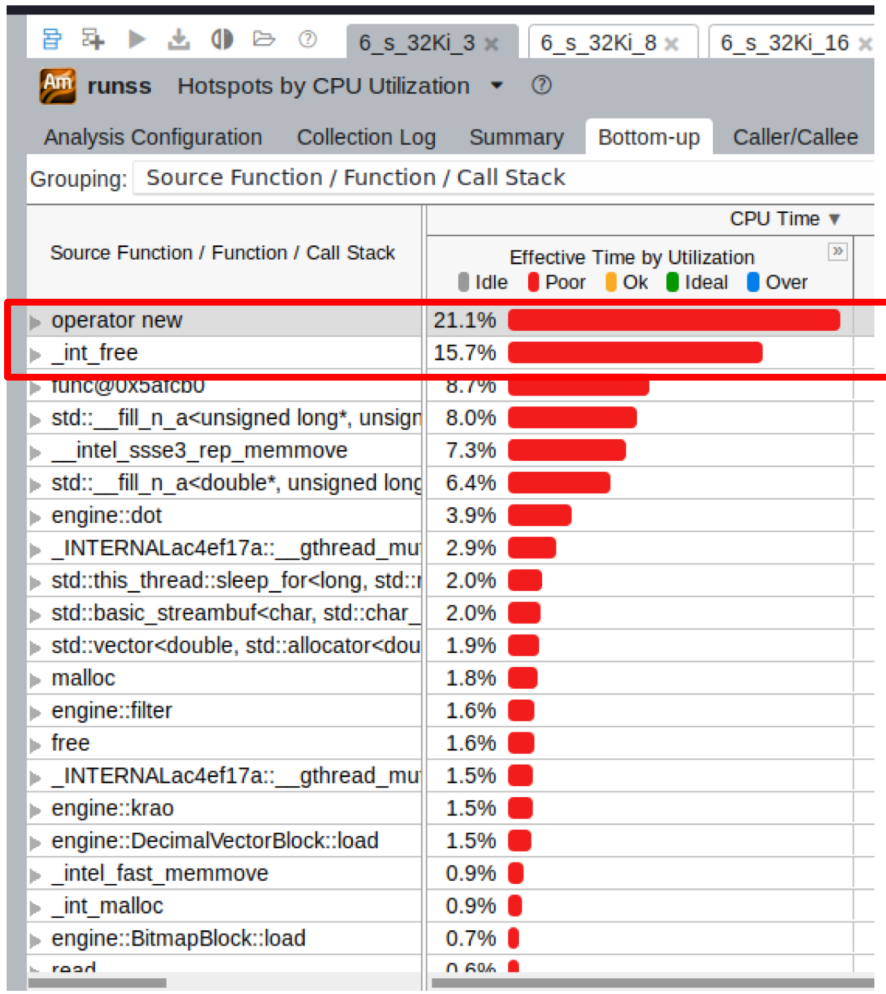
64 threads

Memory management has a significant impact

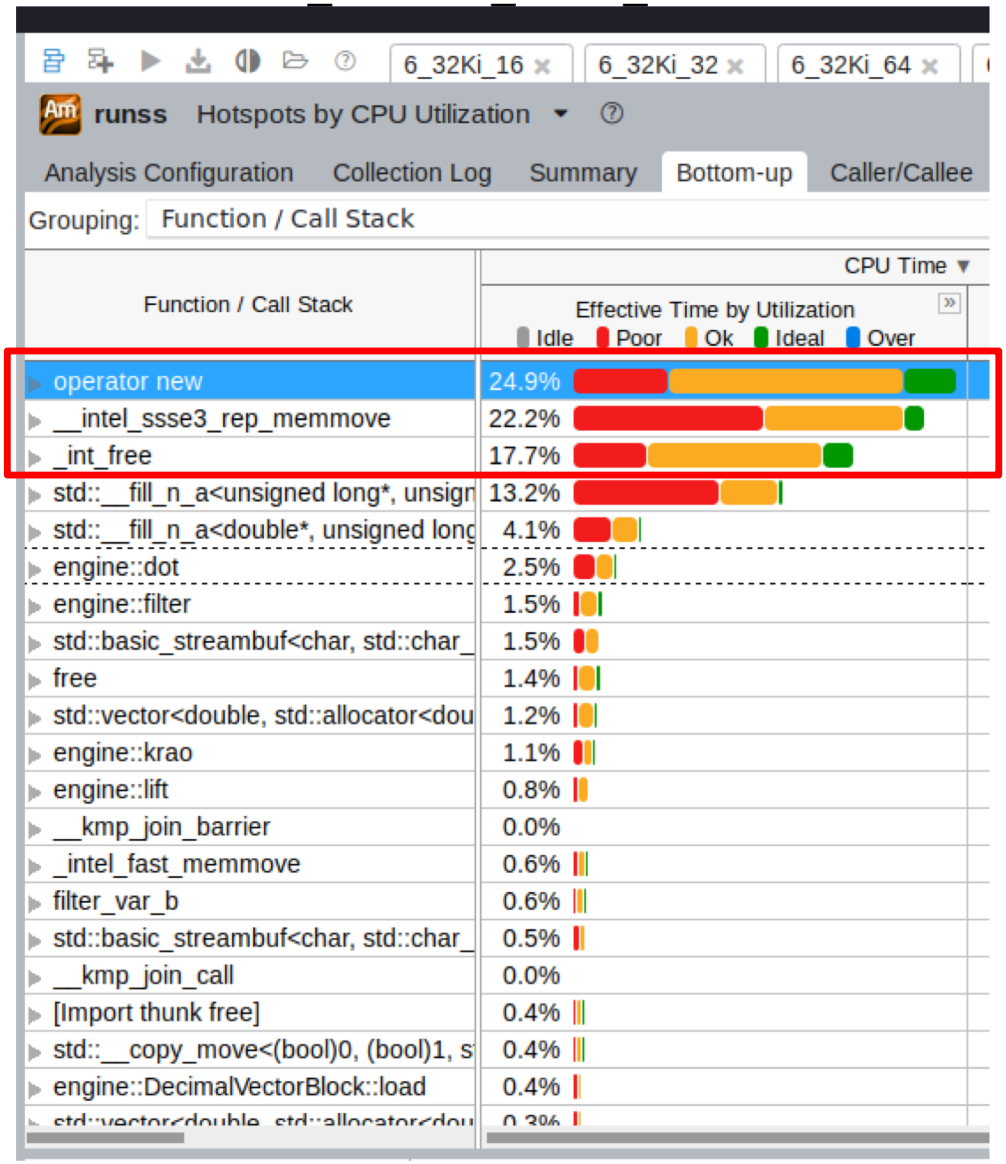
6_JA-OMP_32Ki_16



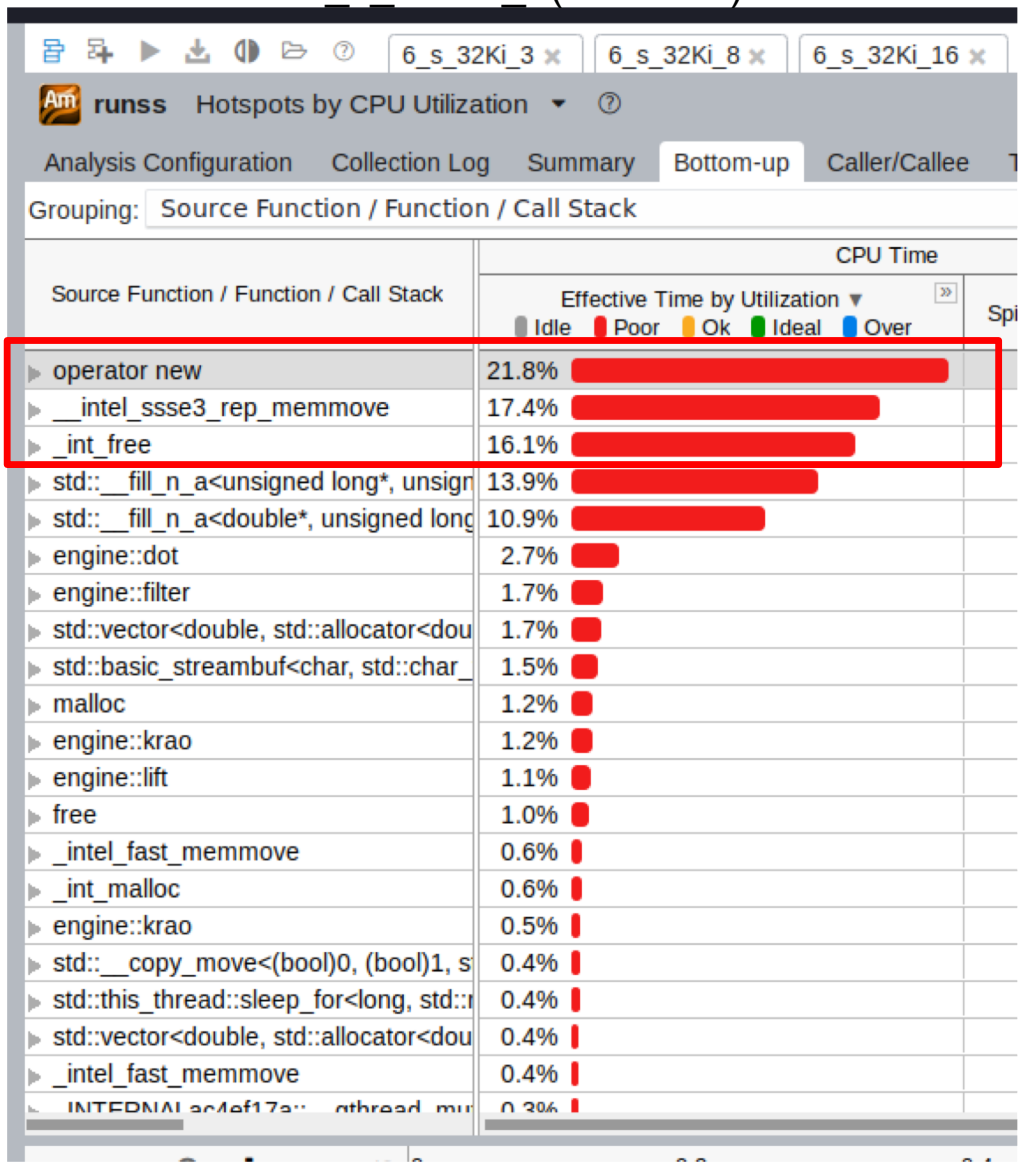
6_s_32Ki_3(16W+8R) 6_s_32Ki_(3M+2*3F+3*D+3*H+1*S+4*2L)



6_JA-OMP_256Ki_64

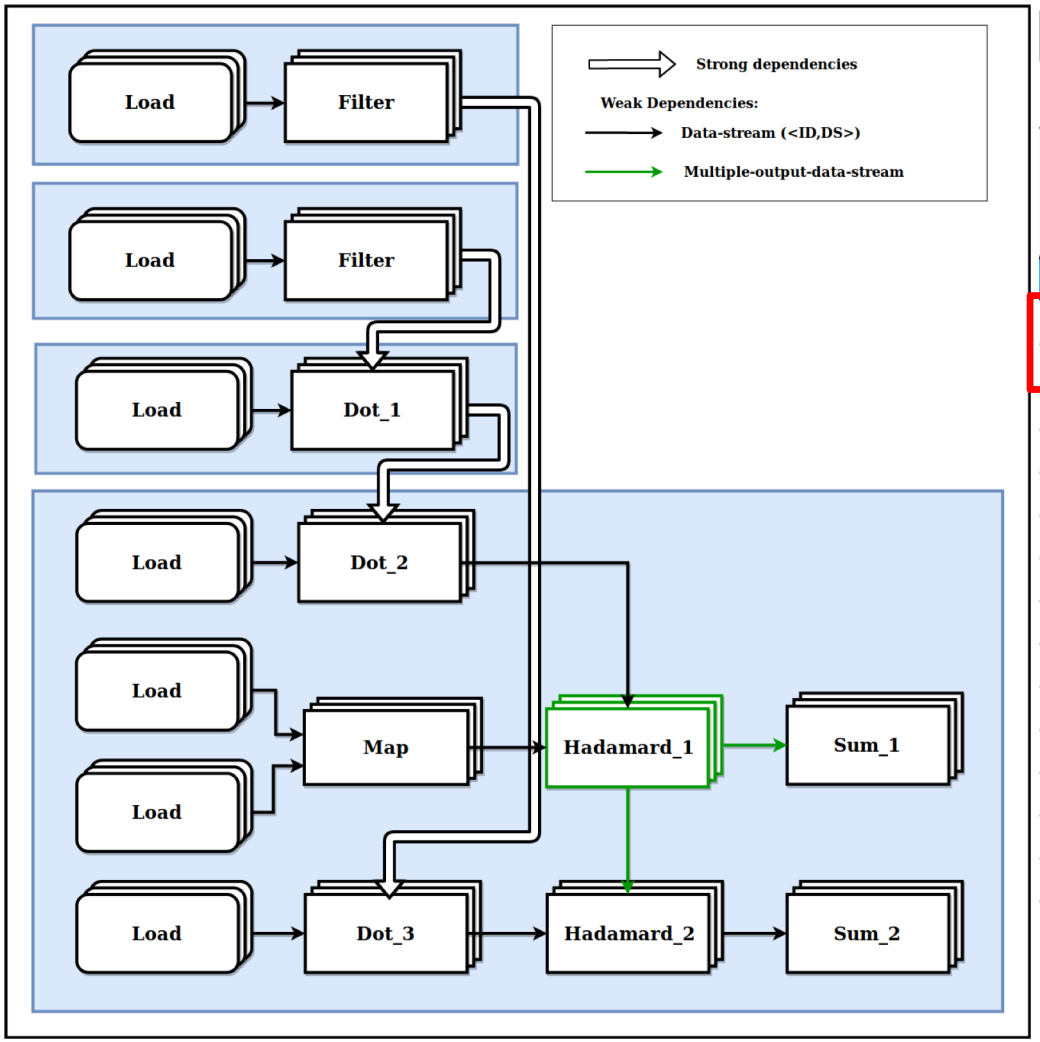


6_s_256Ki_3(16W+8R)



Stream version for HEP-Frame Q14

09-apr-19



Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee

Grouping: Source Function / Function / Call Stack

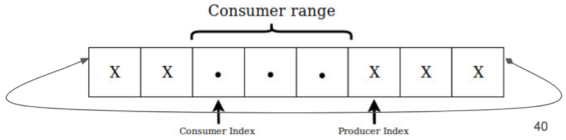
Source Function / Function / Call Stack	CPU Time ▼ Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)
engine::dot	20.5%
operator new	20.0%
_int_free	12.2%
__kmp_fork_barrier	0.0%
__intel_ssse3_rep_memmove	8.8%
std::__fill_n_a<unsigned long*, unsign	3.4%
std::__fill_n_a<double*, unsigned long	3.1%
std::basic_streambuf<char, std::char_	2.8%
_INTERNAL_25_____src_kmp_bar	0.0%
engine::BitmapBlock::load	1.7%
engine::DecimalVectorBlock::load	1.5%
engine::lift	1.5%
__kmp_join_call	0.0%
std::vector<double, std::allocator<dou	0.9%
__kmp_join_barrier	0.0%
free	0.6%

Mod's to previous circular buffers

23-apr-19

New version of communication channel

- Circular buffer (array of pointers) per consumer-op (with basic locks)
  - Supports multiple push & pop operations (producer & consumer)
  - Each element is removed from the array as soon as it is read by a consumer (consumer index update)
  - For now consumers are responsible for the release of memory -> this was improved!
  - For multi-consumer-op cases, each consumer-op owns its own array and the producer-op writes in both buffers and updates the producer index in both buffers
  - No bug found so far...

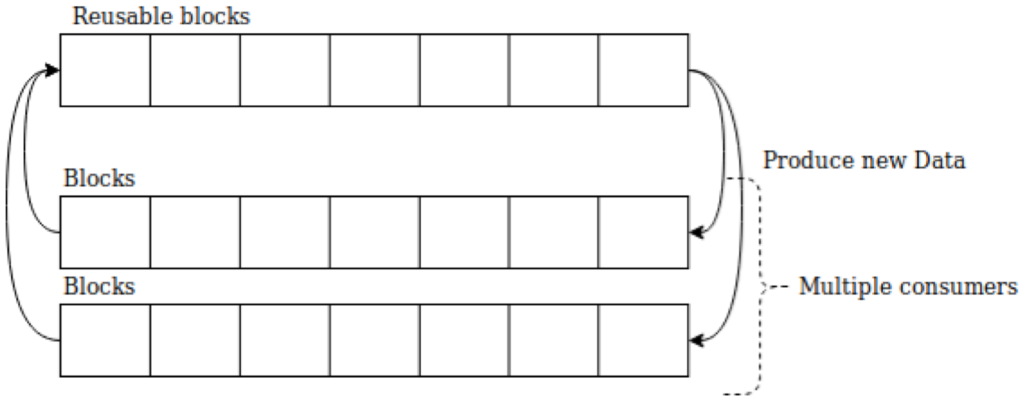


- Avoid active waits (implementation)
  - Put sleeping consumers who are waiting for producers (condiction_variables)
- Replace 4Ki (?) circular buffer by 1Ki queue
- Single mem alloc for the 1Ki queue with the block pointers
- Single mem alloc of each block with reuse of blocks already processed
  - Decrease the number of calls to memory management (new & free)
  - Sug1: use a single reusable mem allocation/free with room for several blocks
  - Sug2: keep using the circular buffer but with much fewer pointers
  - Sug3: is a buffer per consumer instead of per consumer-op a better approach?

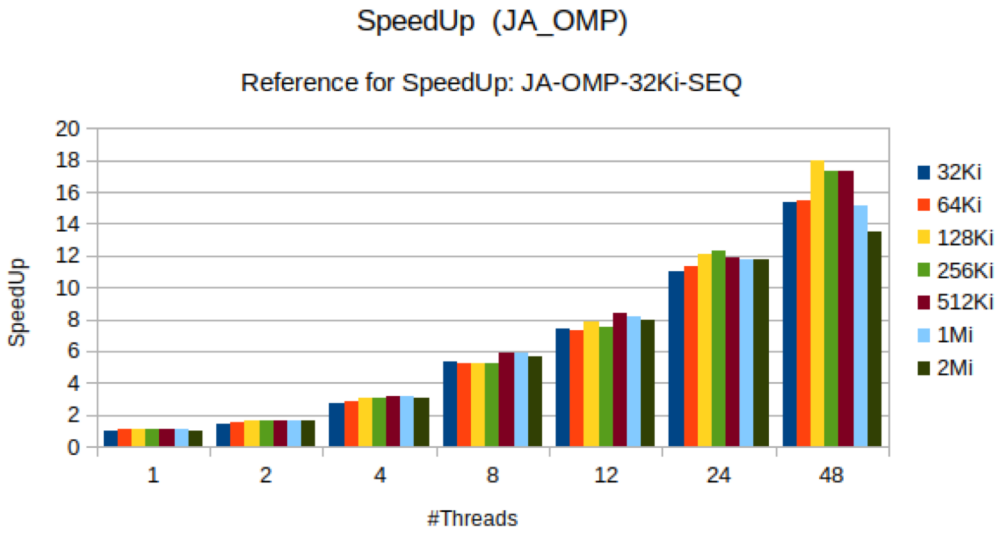
Reuse of blocks already processed

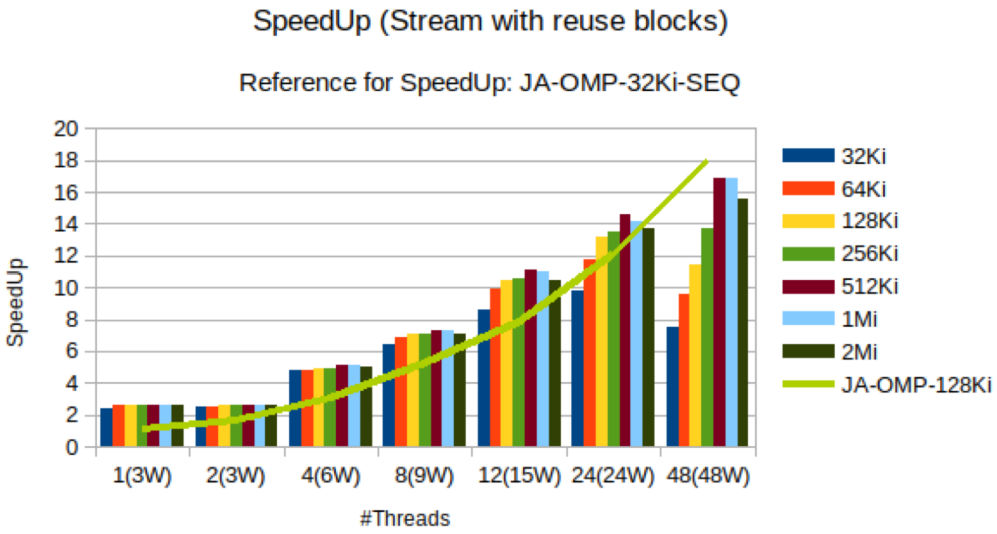
- New problems to consider
  - In cases of multiple consumers, processing the same block may occur more than once
    - Identify the consumers who have already processed the block
  - Operations that require a join may leave unprocessed blocks, which can cause deadlocks
    - Separation of free blocks and blocks waiting for processing

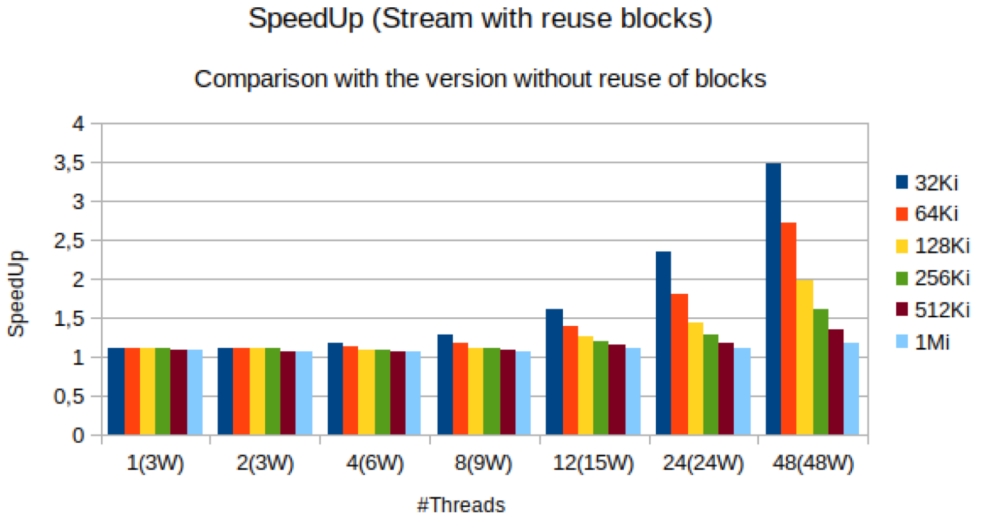
- New Buffer
  - When the blocks are consumed these are cleaned and re-entered into the queue of free blocks.



SpeedUp

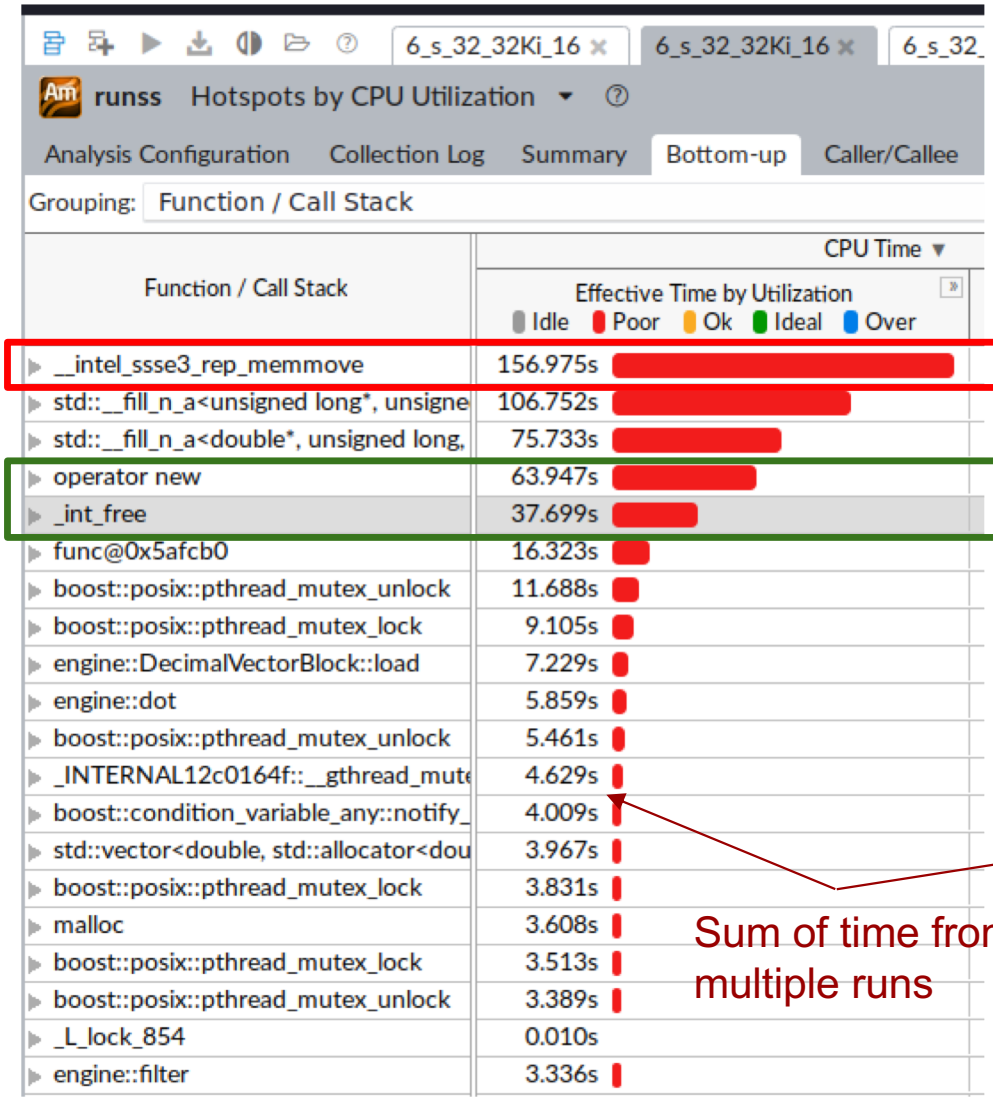






- Node 662 (12-cores per device):
- icc -O2
- TPC-H 32GiB
- 3 best of 15 (5% max error interval)

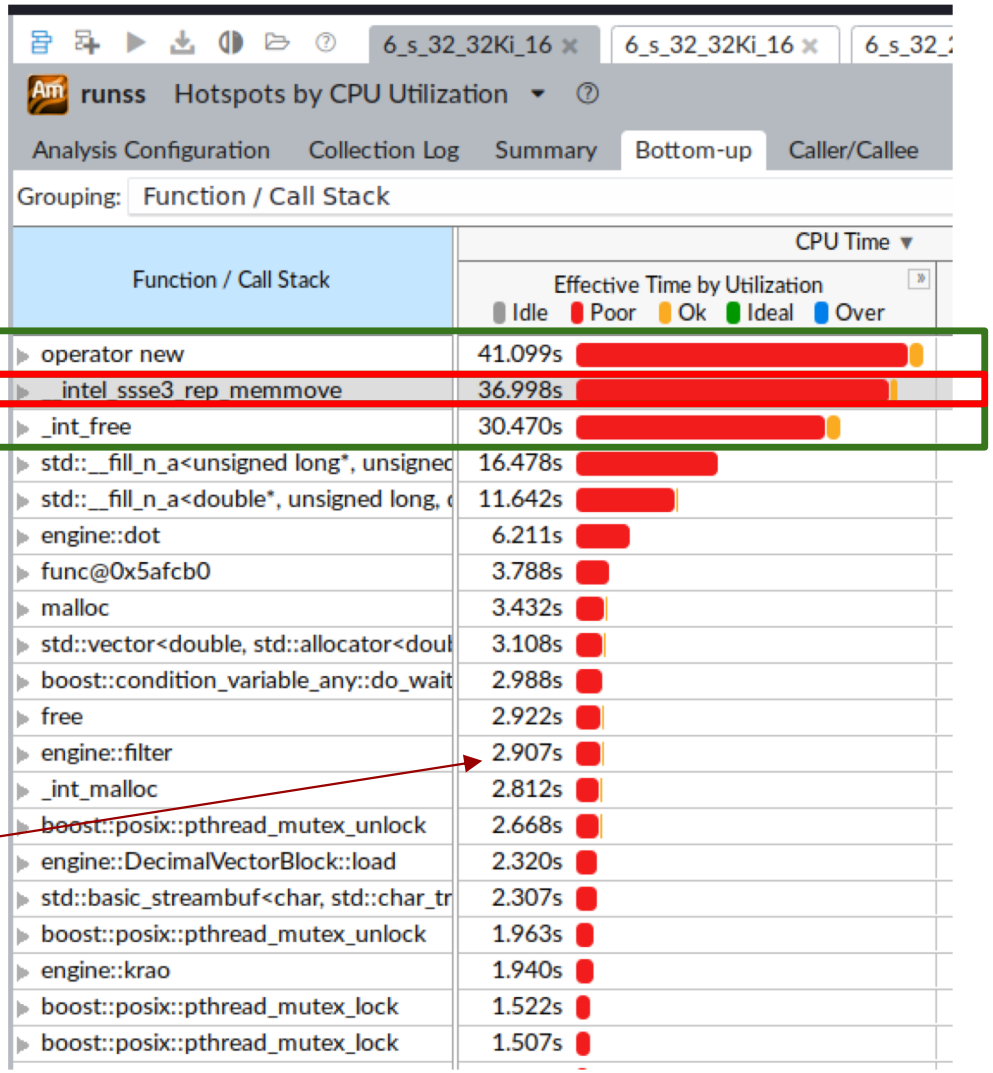
6_Stream_32Ki_16(65W+20R)
No reuse of blocks



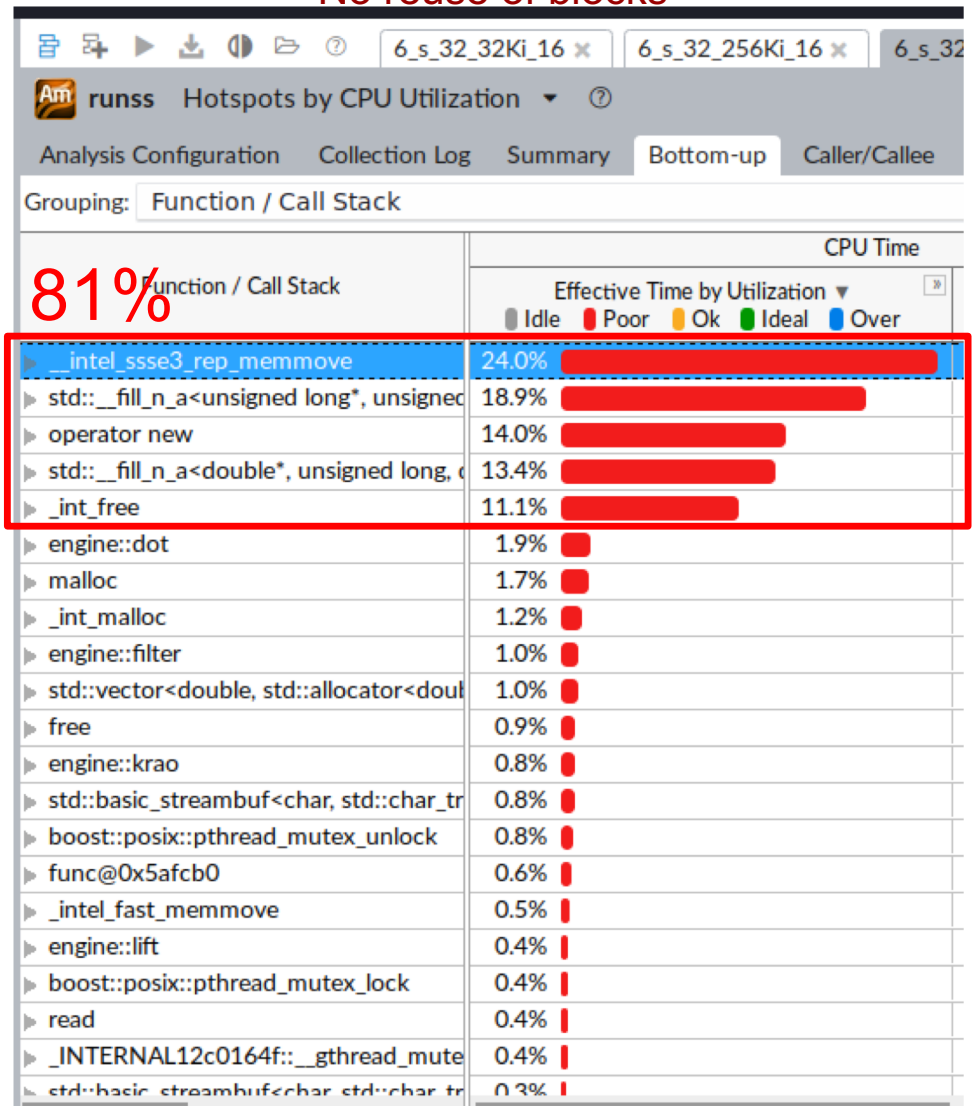
Sum of time from multiple runs

6_Stream_32Ki_16(65W+20R)
New Version

- Node 781 (2x 16-co
- `icc -O2`
- TPC-H 32GiB

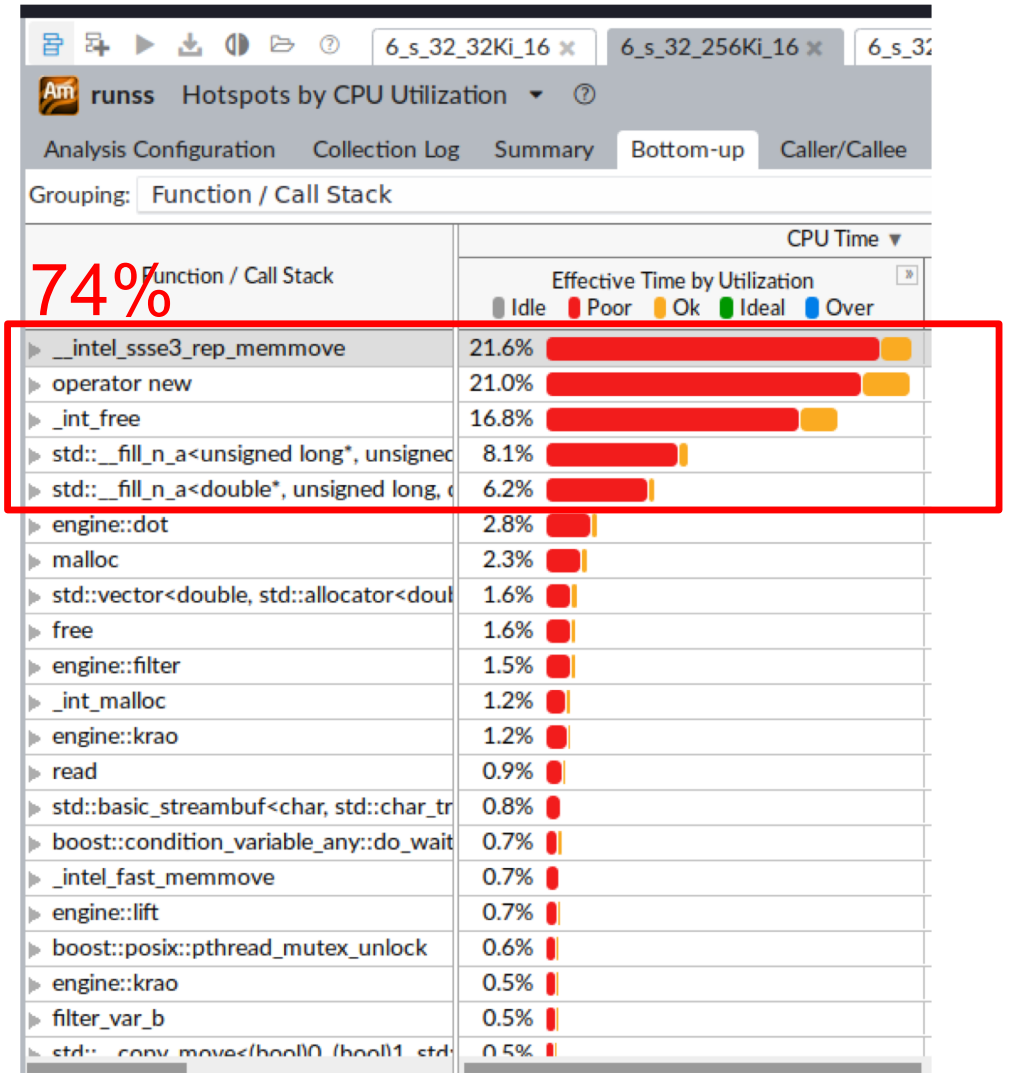


6_Stream_256Ki_16(65W+20R)
No reuse of blocks

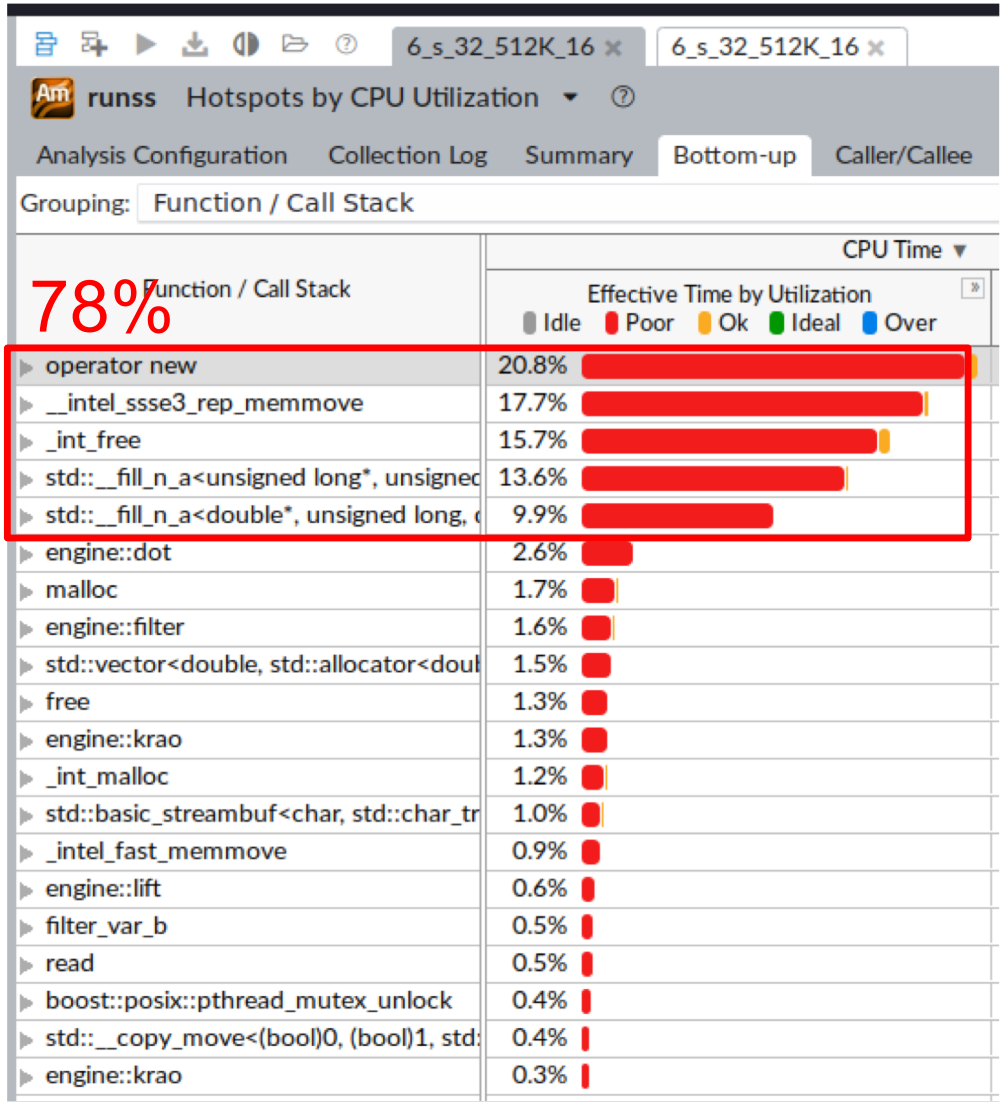


6_Stream_256Ki_16(65W+20R)
New Version

- Node 781 (2x 16-co
- `icc -O2`
- TPC-H 32GiB

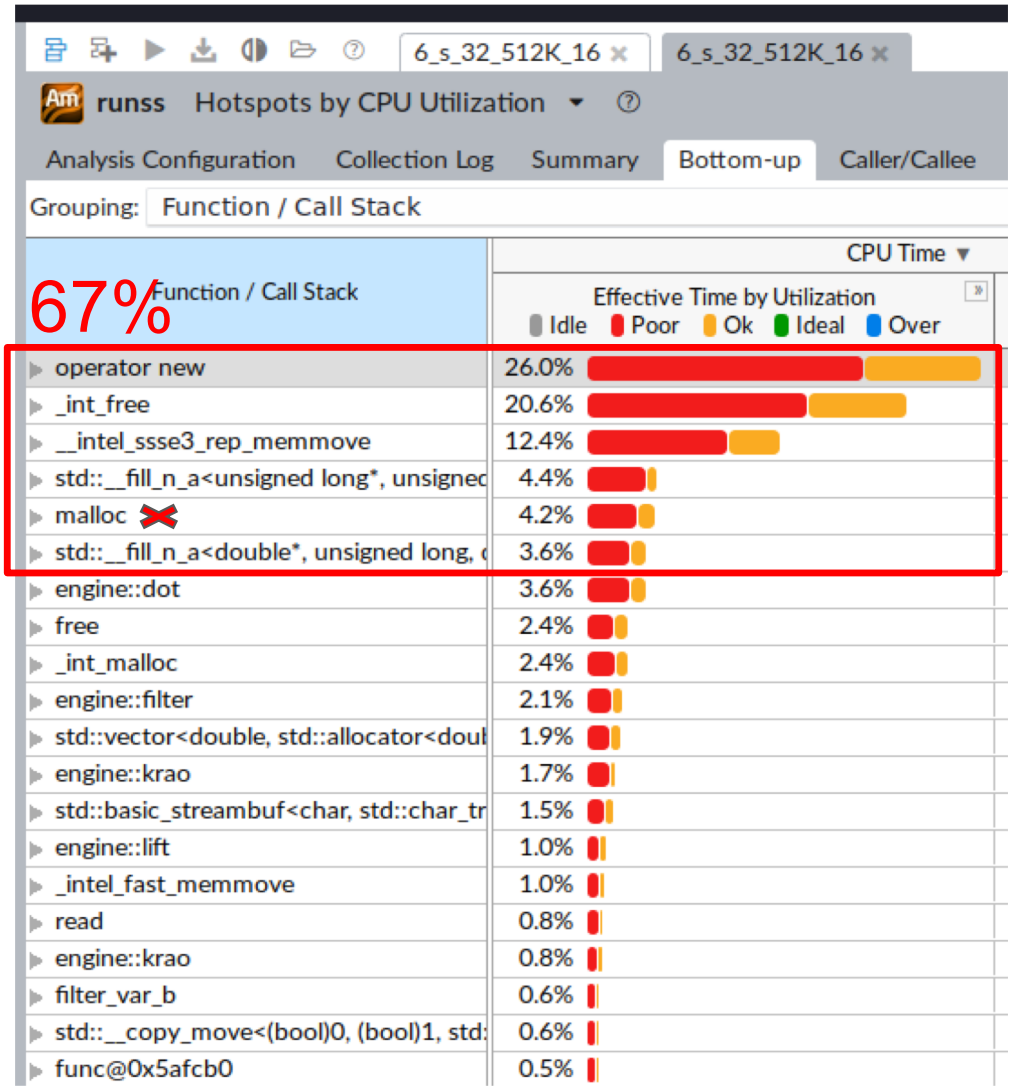


6_Stream_512Ki_16(65W+20R)
No reuse of blocks



6_Stream_512Ki_16(65W+20R)
New Version

- Node 781 (2x 16-co
- `icc -O2`
- TPC-H 32GiB



07-mai-19

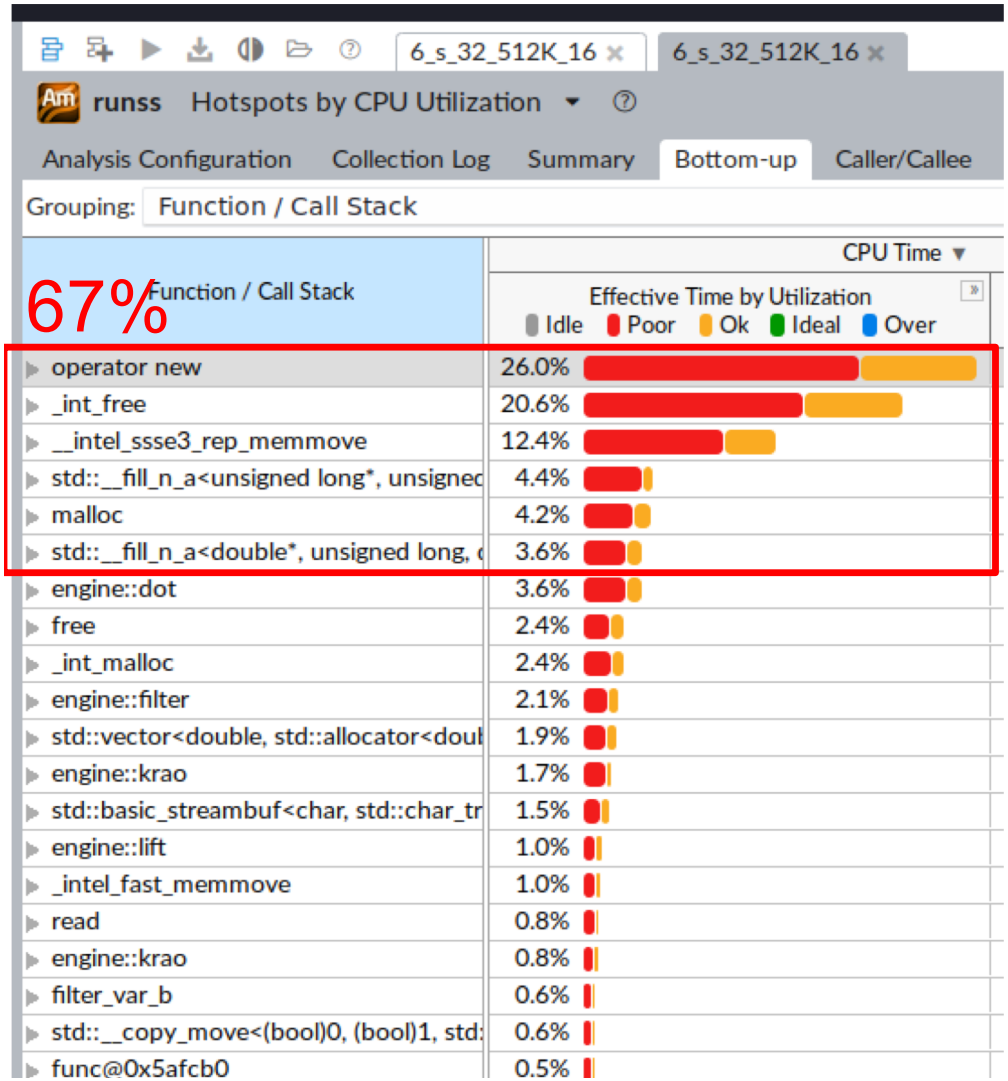
Reason for the large percentage of new operations

- Duplication of the std::vector structure when it is passed as parameter without proper care

```
11  void lift(Decimal(*f)(std::vector<Decimal>),
12            const std::vector<DecimalVectorBlock>& in,
13            DecimalVectorBlock* out) {
14    std::vector<Decimal> v(in.size());
15    Size in_nnz = in[0].nnz;
16    Size in_size = in.size();
17    for each element of the block
18    for (Size i = 0; i < in_nnz; ++i) {
19      for (Size j = 0; j < in_size; ++j) {
20        v[j] = in[j].values[i];
21      }
22      out->values[i] = (*f)(v);      (*f)(&v)
23    }
24    out->nnz = in_nnz;
25  }
```

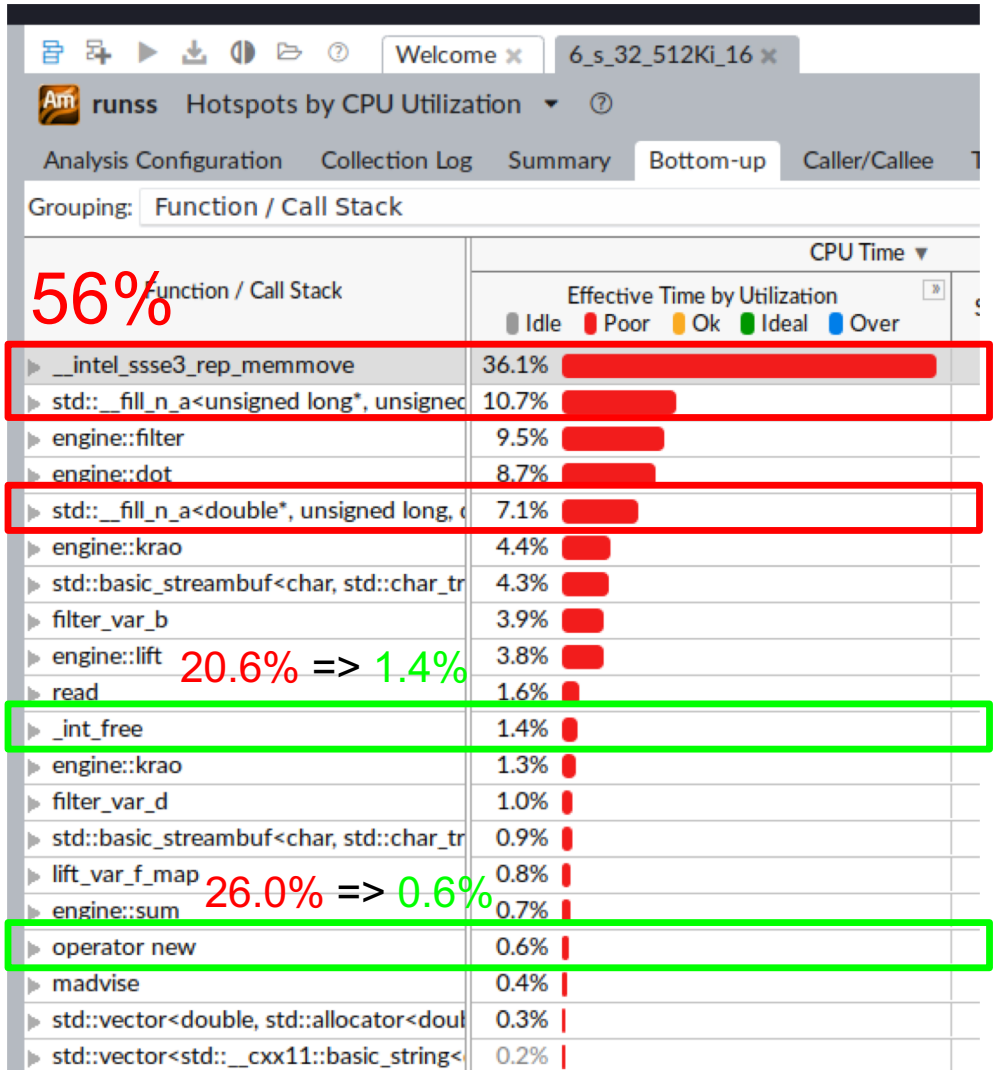
```
10  void filter(bool(*f)(std::vector<Decimal>),
11              const std::vector<DecimalVectorBlock>& in,
12              FilteredBitVectorBlock* out) {
13    std::vector<Decimal> v(in.size());
14    Size i, nnz = 0;
15    Size in_nnz = in[0].nnz, in_size = in.size();
16    for each element of the block
17    for (i = 0; i < in_nnz; ++i) {
18      out->cols[i] = nnz;
19      for (Size j = 0; j < in_size; ++j) {
20        v[j] = in[j].values[i];
21      }
22      if ((*f)(v)) {      (*f)(&v)
23        ++nnz;
24      }
25    }
26    out->cols[i] = nnz;
27    out->nnz = nnz;
28  }
```

6_Stream_512Ki_16(65W+20R)
Reuse of Blocks



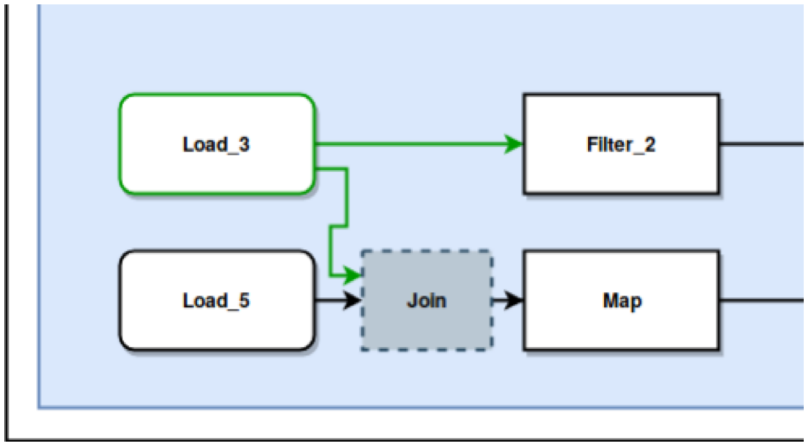
6_Stream_512Ki_16(65W+20R)
Reuse of Blocks (Vector optimized)

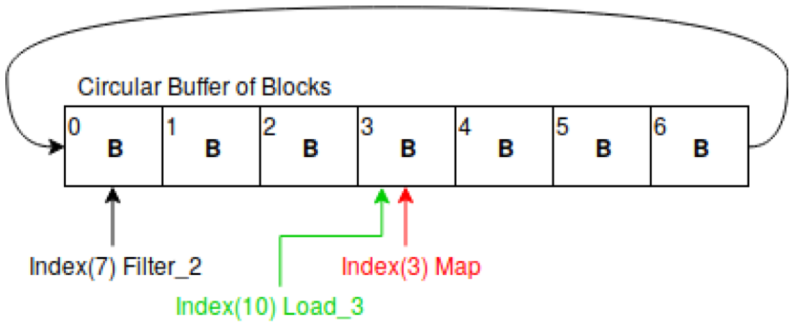
- Node 781 (2x 16-
- `icc -O2`
- TPC-H 32GiB



Circular buffers: can lead to loss of performance

- Unwanted waits
  - At a consumer with multi-producer dependencies (e.g., Map): when the consumer has to wait for data from one producer
  - At a producer with multi-consumer dependencies (e.g., Load_3): when the producer can't rewrite the blocks that are held by a consumer, it has to wait for data to be read by that consumer (e.g., Filter)
- Confirm with measurements





Operational intensity Dot

```
void dot(const FilteredBitVector& A,
         const BitmapBlock& B,
         FilteredBitVectorBlock *C) {
  Size i, nnz = 0;
  Size b_nnz = B.nnz;
  for (i = 0; i < b_nnz; ++i) {
    C->cols[i] = nnz;

    auto a_size = A.blocks[0]->cols.size() - 1;

    auto b_row = B.rows[i];
    auto a_col = b_row % (a_size);
    auto block = b_row / (a_size);

    if (A.blocks[block]->cols[a_col+1] >
        A.blocks[block]->cols[a_col]) {
      ++nnz;
    }
  }
  C->cols[i] = nnz;
  C->nnz = nnz;
}
```

- 1 operation/operand(one element of the Block)
  - ~~2 int add operations~~
  - 1 division
  - ~~1 int comparation~~

Operational intensity Filter

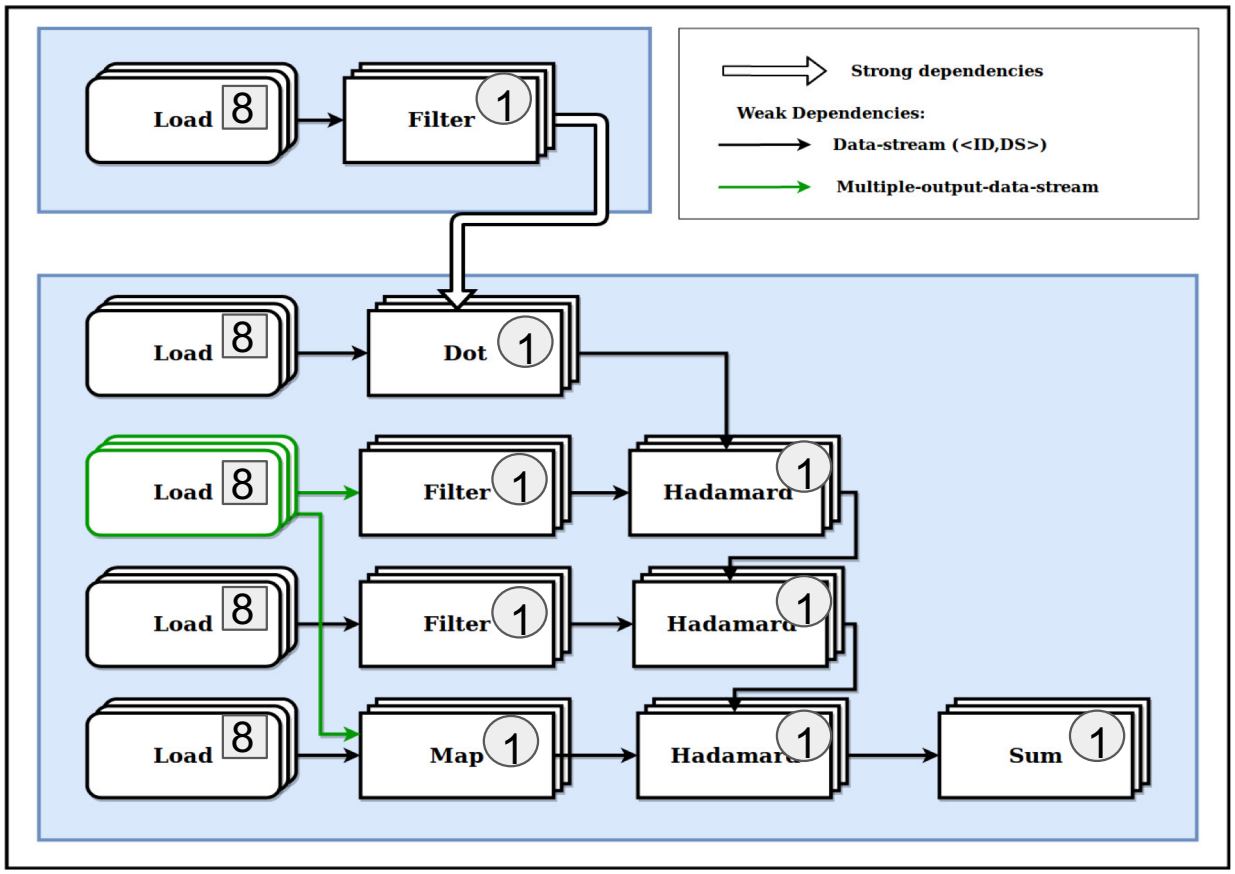
```
22
23  inline bool filter_var_a(std::vector<engine::Literal> *args)
24  {
25      return (*args)[0]>="1994-01-01"&&(*args)[0]<"1995-01-01";
26  }
```

```
void filter(bool(*f)(std::vector<Literal> *),
            const std::vector<LabelBlock>& in,
            FilteredBitVectorBlock* out) {
  std::vector<Literal> v(in.size());
  Size i, nnz = 0;
  Size in_nnz = in[0].nnz, in_size = in.size();

  for (i = 0; i < in_nnz; ++i) {
    out->cols[i] = nnz;
    for (Size j = 0; j < in_size; ++j) {
      v[j] = in[j].labels[i];
    }
    if ((*f)(&v)) {
      ++nnz;
    }
  }
  out->cols[i] = nnz;
  out->nnz = nnz;
}
```

- 1 operation/operand(one element of the Block)
  - ~~3 int add operations~~
  - 1 filter function

Operational Intensity



1 Operational intensity/operand

8 Bytes/operation

(2*9)/(8*5)=0.225 Op/Byte

Tasks

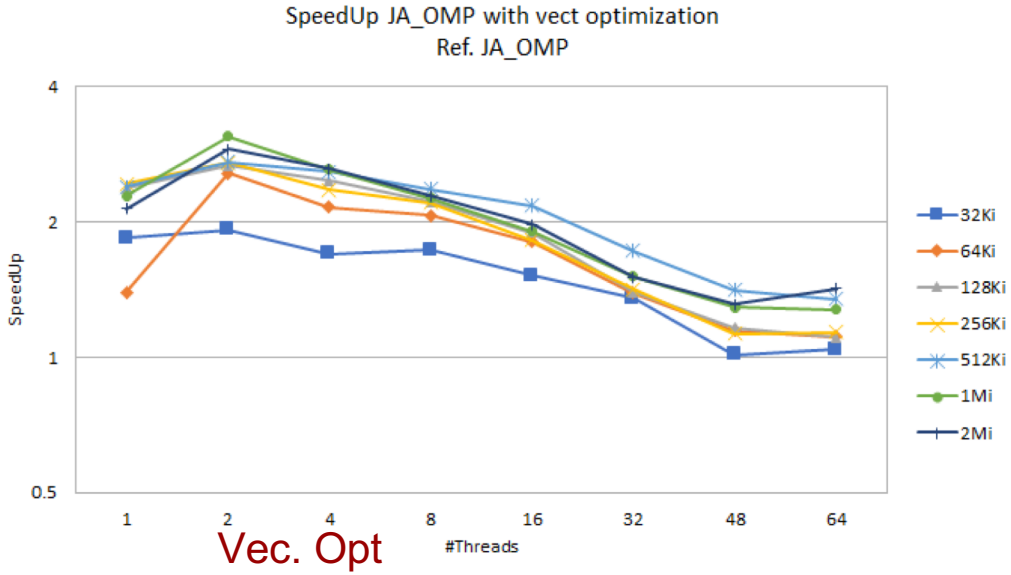
- SpeedUp (ref. JA-OMP)
- Thread scheme
- Callgrind
- Waiting time (Circular Buffer)
- Analysis of vectorization reports
- Confirmation of operational intensity (PAPI counters & Intel Advisor)
- -O3

21-mai-19

SpeedUp comparison OMP_JA

- Comparison of the optimizations applied to the **OMP_JA** version of the query 6
- The next slide will compare the versions with **block reuse** and **vector optimization** in relation to the version without optimization
- We can verify that the **reuse of blocks** has some improvements to the smaller block size (32Ki, 64Ki, 128Ki and 256Ki) which is justified by the greater amount of allocations and releases of memory

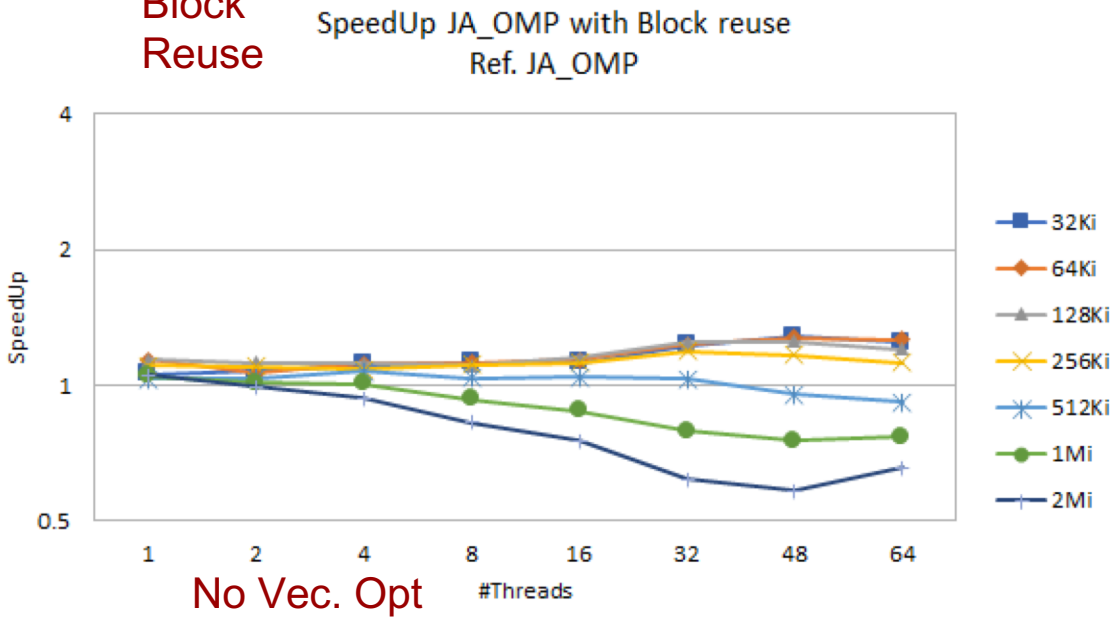
- Node 781 (2x 16-cores, HT)
- `icc -O2`
- TPC-H 32GiB
- Ref: no block-reuse, no vector opt



No Block Reuse

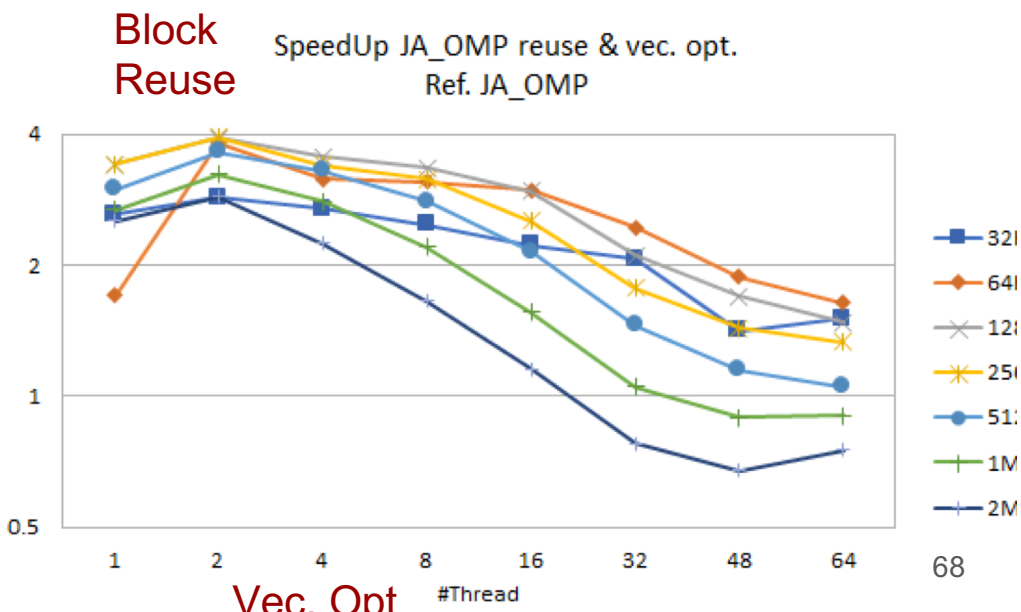
Vec. Opt

Block Reuse



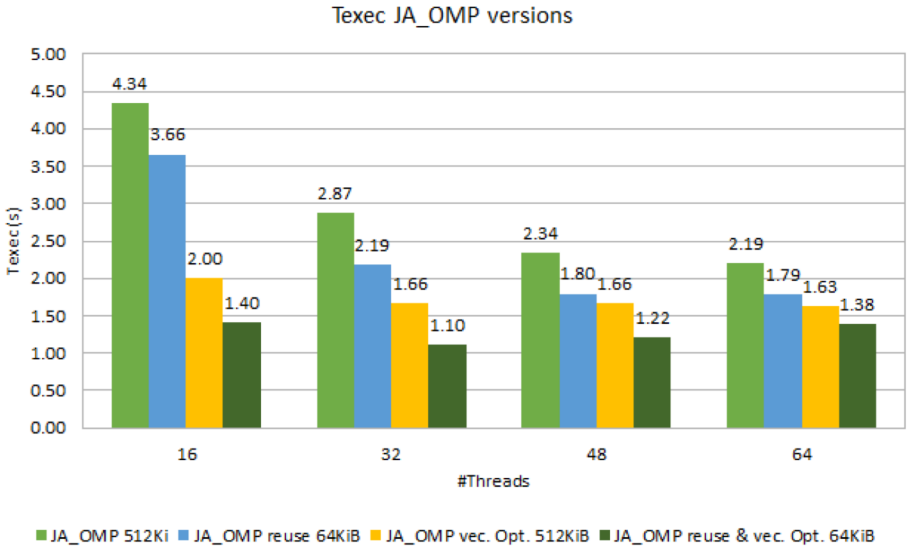
No Vec. Opt

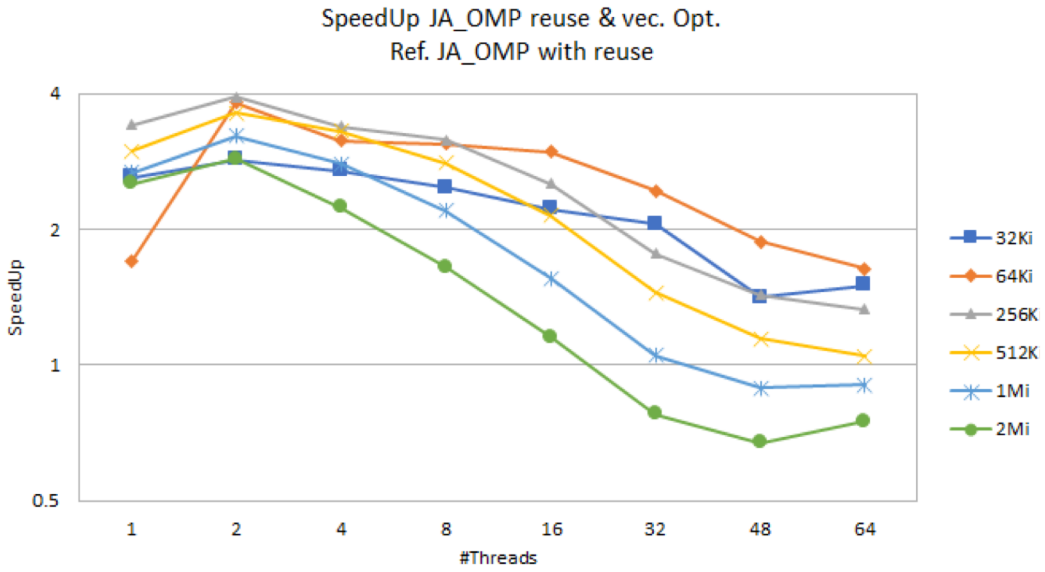
Block Reuse

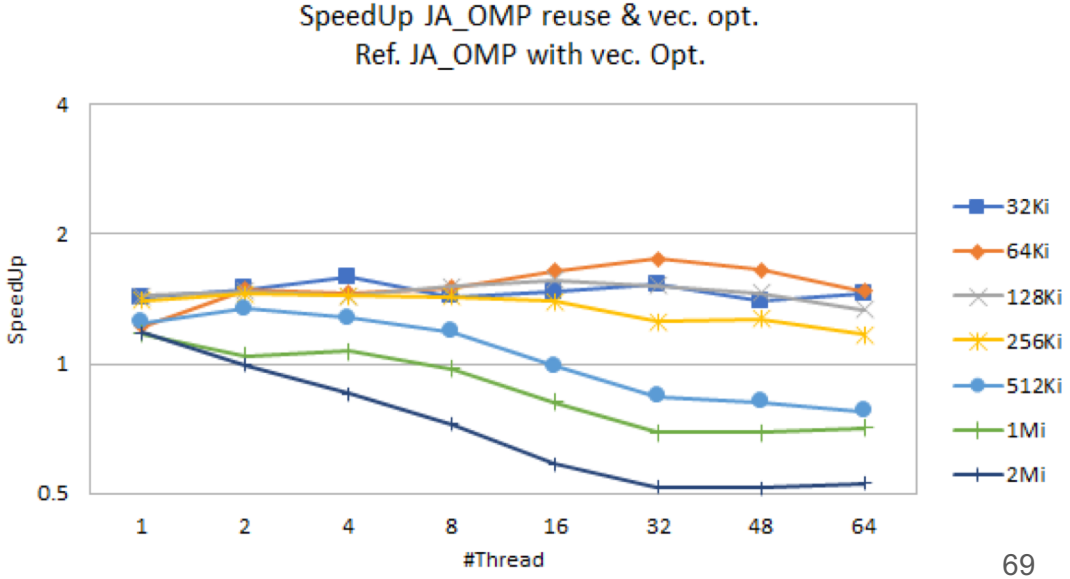


Vec. Opt

- Node 781 (2x 16-cores, HT)
- `icc -O2`
- TPC-H 32GiB







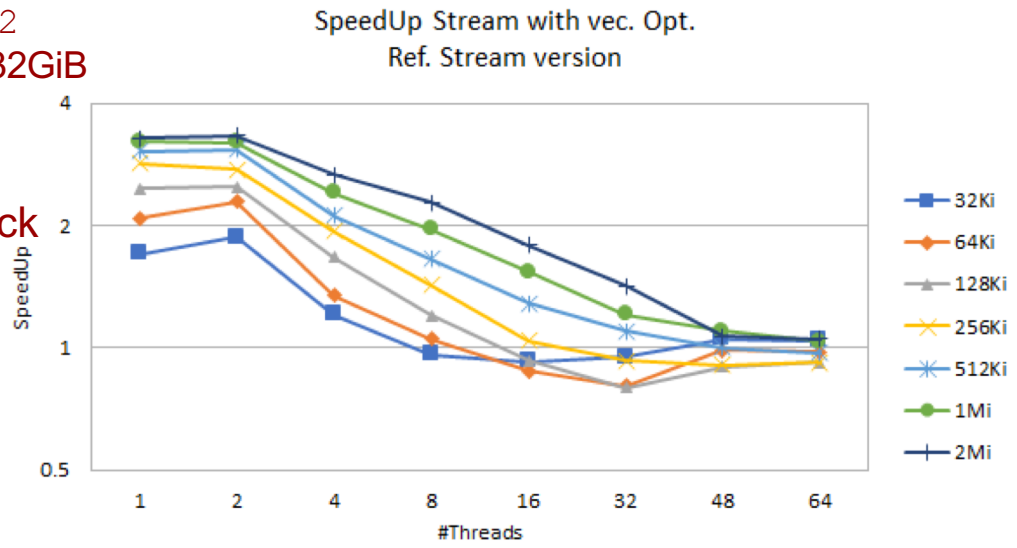
SpeedUp comparison Stream

- Comparison of the optimizations applied to the **Stream** version of the query 6
- The next slide will compare the versions with **block reuse** and **vector optimization** in relation to the stream version without optimization
- In this case the **reuse of blocks** presents results for all block sizes, according to the number of memory allocation and release operations

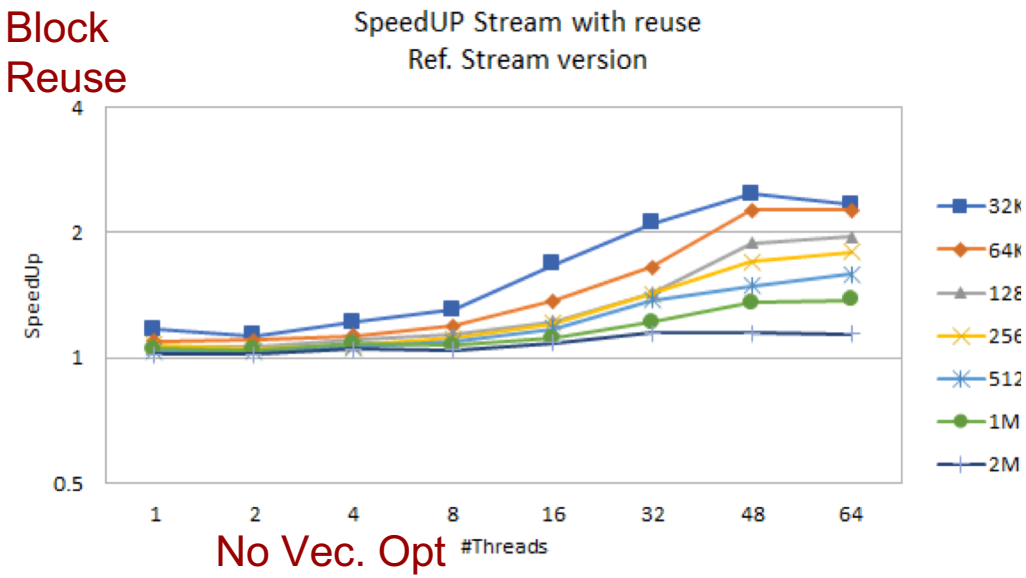
	Total	(*3)	(*4)	(*3)	(*1)
1	11	1	1	1	1
2	15	1	2	1	1
4	22	2	2	2	2
8	26	3	2	2	3
16	37	5	2	3	5
32	53	8	3	3	8
48	88	16	3	4	16
64	120	24	3	4	24

- Node 701 (2x 16-cores, HT)
- `icc -O2`
- TPC-H 32GiB

No Block Reuse

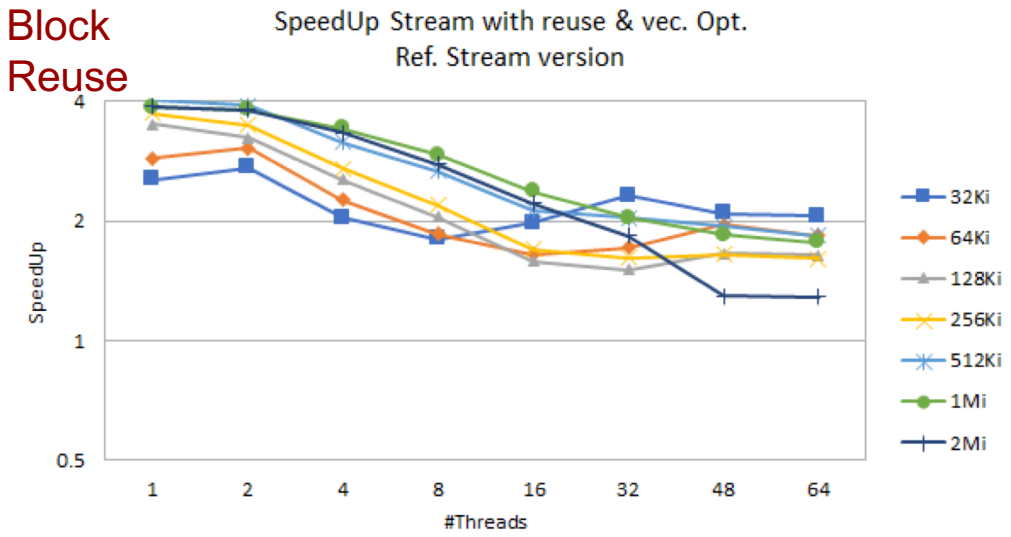


Block Reuse



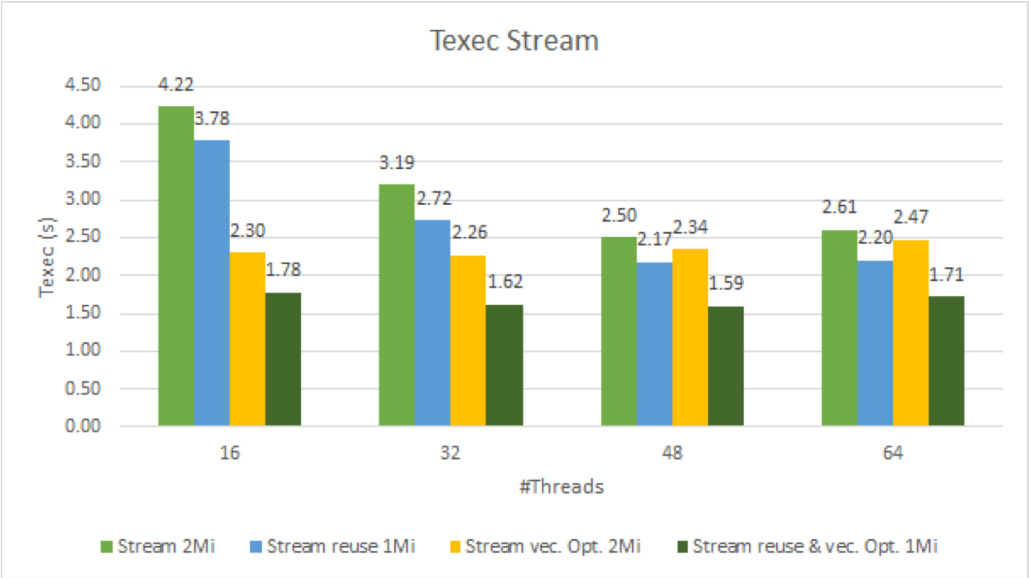
No Vec. Opt

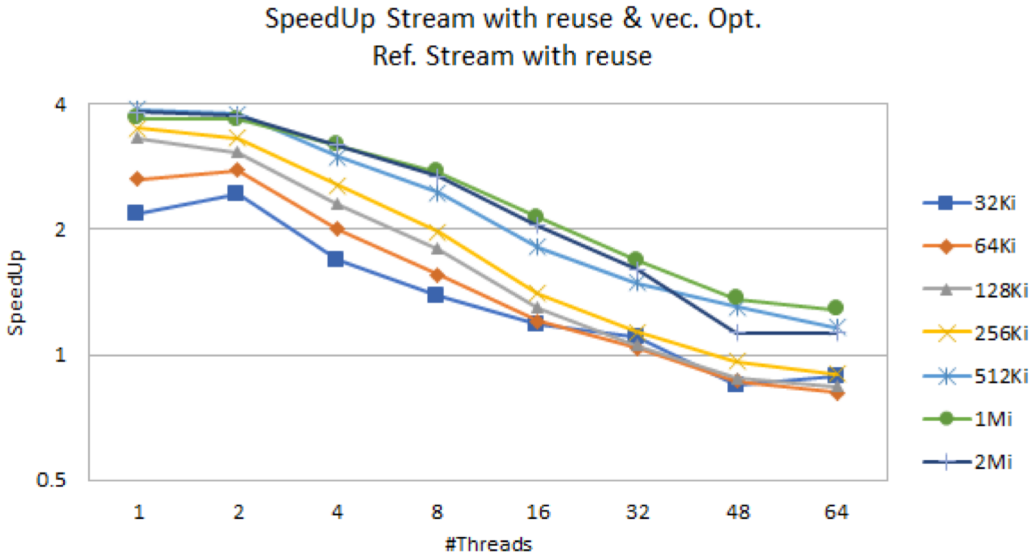
Block Reuse

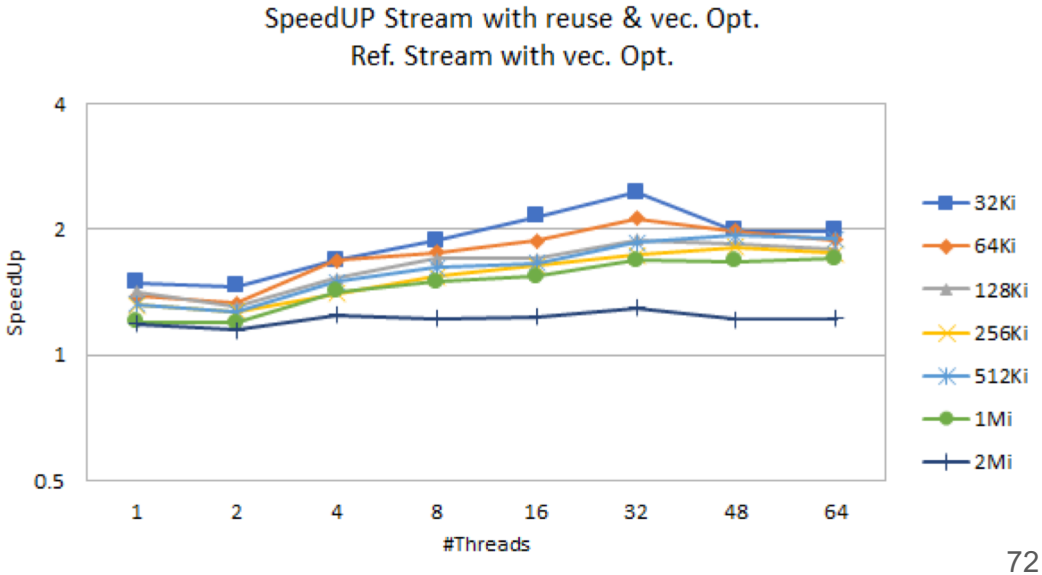


Vec. Opt

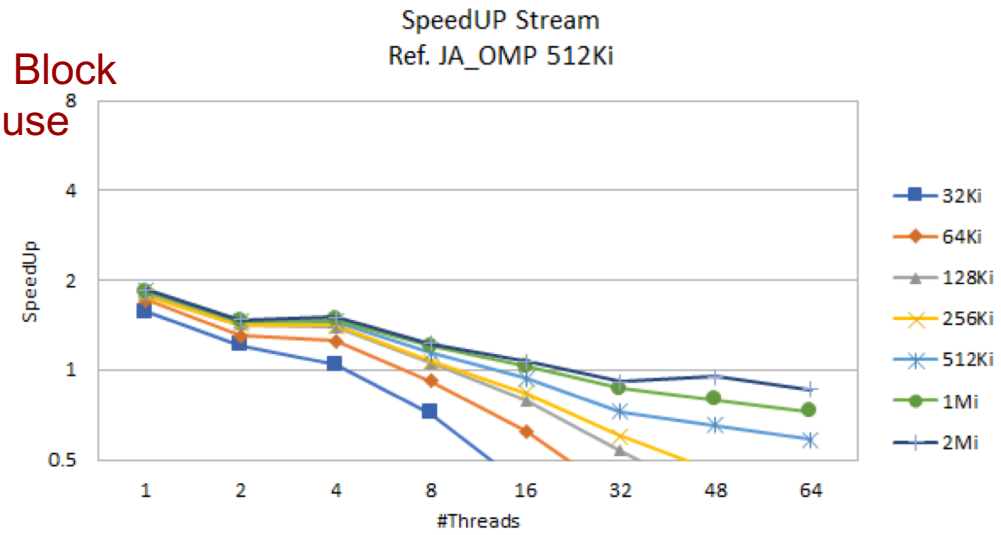
- Node 781 (2x 16-cores, HT)
- `icc -O2`
- TPC-H 32GiB

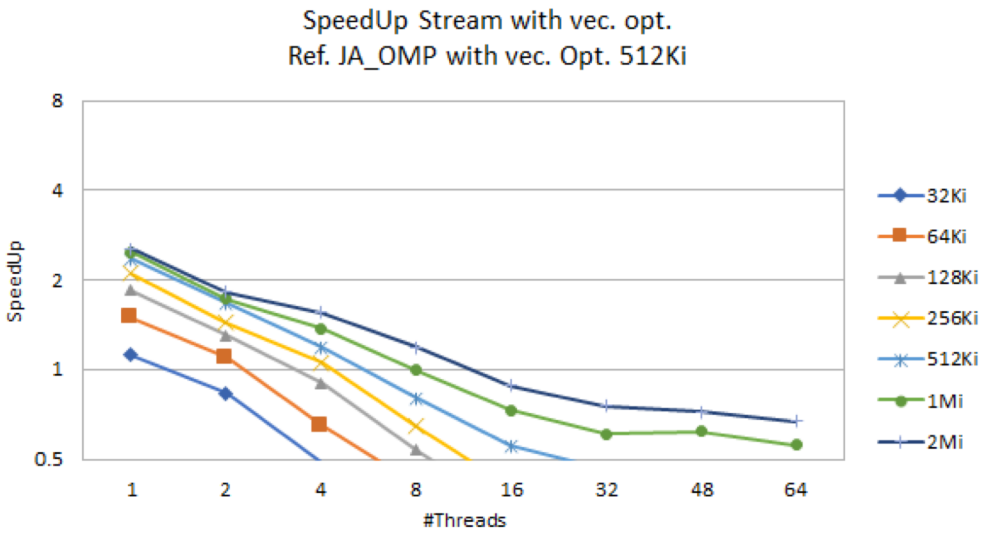




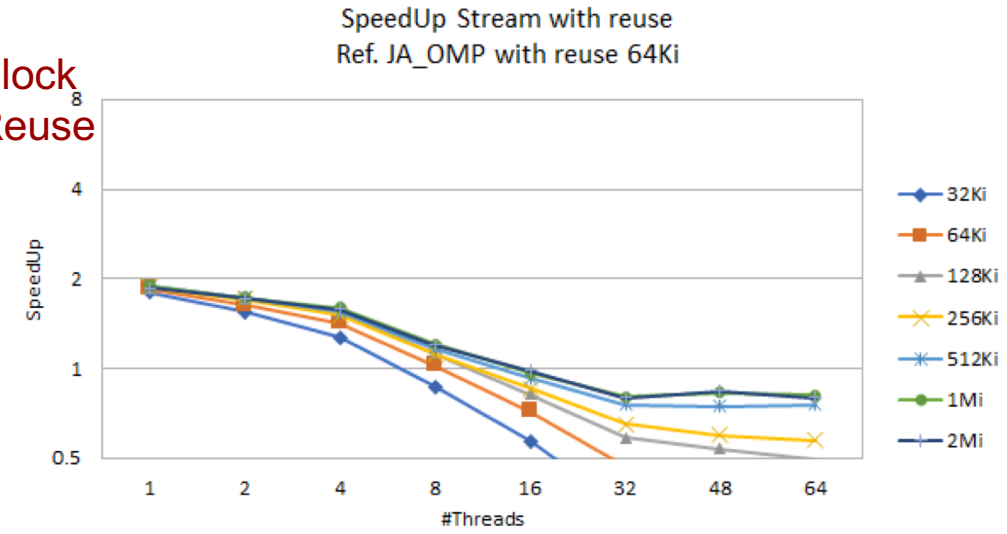


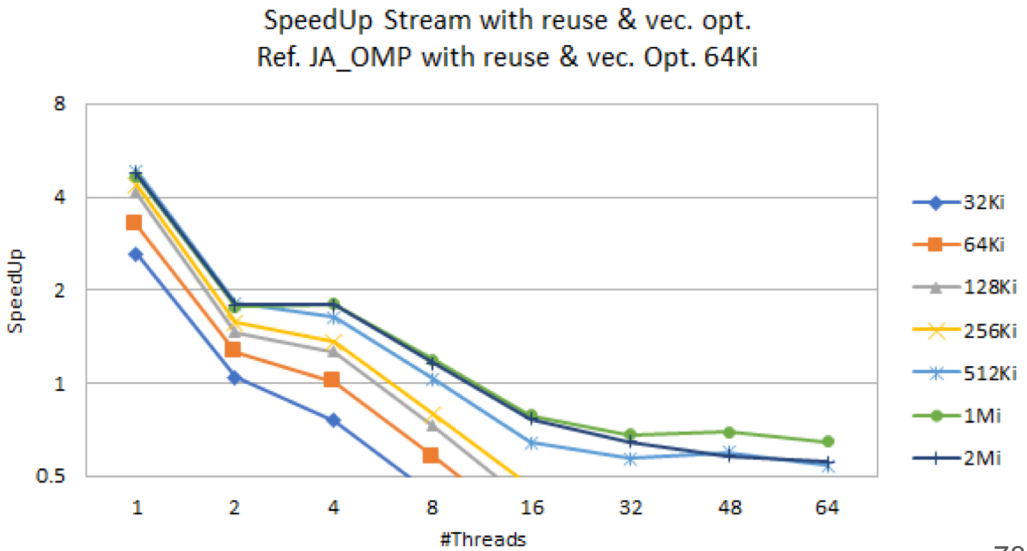
No Block Reuse





Block Reuse





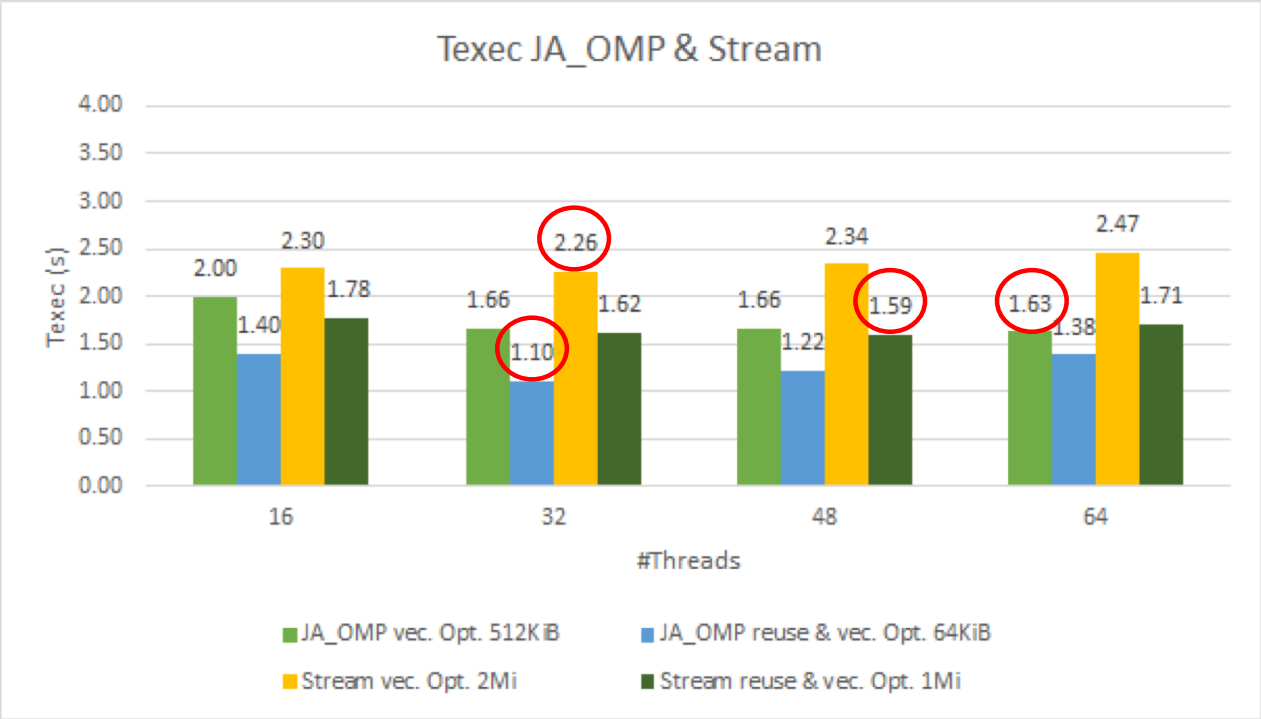
No Vec. Opt

Vec. Opt

Reasons for low performance of the stream version

- Overhead stream and pipeline
  - Poor distribution of work by threads (introduction of priorities)
  - Overload of lock mechanisms
- Cache
  - In the case of OMP, the generated data is processed by the same thread, which allows a better use of the temporal location of the data
  - In the case of the stream version each thread performs only one pipeline operation, which can increase the cache miss (Confirm with PAPI)
  - Note: The impact of the size of the blocks in the two implementations may be another indicator. Since the OMP version benefits from small blocks (64ki) and the stream version benefits from large blocks (1Mi)

- Node 781 (2x 16-cores, HT)
- `icc -O2`
- TPC-H 32GiB



OMP	Stream Total	Work (*3)	Read (*4)	Had (*3)	Dot (*1)
1	11	1	1	1	1
2	15	1	2	1	1
4	22	2	2	2	2
8	26	3	2	2	3
16	37	5	2	3	5
32	53	8	3	3	8
48	88	16	3	4	16
64	120	24	3	4	24

Vectorization on LA operations

- Dot; Filter; Lift(Map); krao(Hadamard); Fold(Sum)

Dot

- Data dependency and indirect & non-consecutive memory access prevent vectorization

```
void dot(const FilteredBitVector& A,
         const BitmapBlock& B,
         FilteredBitVectorBlock *C) {
  Size i, nnz = 0;
  Size b_nnz = B.nnz;
  auto a_size = A.blocks[0]->cols.size() - 1;

  for (i = 0; i < b_nnz; ++i) {
    C->cols[i] = nnz;

    auto b_row = B.rows[i];
    auto a_col = b_row % (a_size);
    auto block = b_row / (a_size);

    if (A.blocks[block]->cols[a_col+1] >
        A.blocks[block]->cols[a_col]) {
      ++nnz;
    }
  }
  C->cols[i] = nnz;
  C->nnz = nnz;
}
```

Filter

- Inner loop, function call (*f) and data dependency (nnz) impact the vectorization application

```
void filter(bool(*f)(std::vector<Decimal> *),
            const std::vector<DecimalVectorBlock>& in,
            FilteredBitVectorBlock* out) {
  std::vector<Decimal> v(in.size());
  Size i, nnz = 0;
  Size in_nnz = in[0].nnz, in_size = in.size();

  for (i = 0; i < in_nnz; ++i) {
    out->cols[i] = nnz;
    for (Size j = 0; j < in_size; ++j) {
      (v)[j] = in[j].values[i];
    }
    if ((*f)(&v)) {
      ++nnz;
    }
  }
  out->cols[i] = nnz;
  out->nnz = nnz;
}
```

- The inner loop can be removed, knowing the number of vector elements
- However, the data dependency can not be removed

```
void filter(bool(*f)(std::vector<Decimal> *),
            DecimalVectorBlock in,
            FilteredBitVectorBlock* out) {
  std::vector<Decimal> v(1);
  Size i, nnz = 0;
  Size in_nnz = in.nnz, in_size = 1;

  for (i = 0; i < in_nnz; ++i) {
    out->cols[i] = nnz;
    (v)[0] = in.values[i];
    if ((*f)(&v)) {
      ++nnz;
    }
  }
  out->cols[i] = nnz;
  out->nnz = nnz;
}
```

Lift(Map)

- Inner loop and function call (inline) prevent vectorization
- The inner loop can be removed, knowing the number of vector elements
- In this second version the loop is vectorizable

```
LOOP BEGIN at /usr/include/c++/7.4.0/bits/stl_algobase.h(752,7)
   remark #15389: vectorization support: reference *__first has
   c++/7.4.0/bits/stl_algobase.h(754,3) ]
   remark #15381: vectorization support: unaligned access used i
   remark #15305: vectorization support: vector length 2
   remark #15399: vectorization support: unroll factor set to 2
   remark #15309: vectorization support: normalized vectorizatio
   remark #15300: LOOP WAS VECTORIZED
   remark #15451: unmasked unaligned unit stride stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 4
   remark #15477: vector cost: 3.000
   remark #15478: estimated potential speedup: 1.170
   remark #15488: --- end vector cost summary ---
LOOP END
```

```
void lift(Decimal(*f)(std::vector<Decimal> *),
          const std::vector<DecimalVectorBlock>& in,
          DecimalVectorBlock* out) {
  std::vector<Decimal> v(in.size());
  Size in_nnz = in[0].nnz;
  Size in_size = in.size();

  for (Size i = 0; i < in_nnz; ++i) {
    for (Size j = 0; j < in_size; ++j) {
      v[j] = in[j].values[i];
    }
    out->values[i] = (*f)(&v);
  }
  out->nnz = in_nnz;
}
```

```
void lift_2(Decimal(*f)(std::vector<Decimal> *),
          const std::vector<DecimalVectorBlock>& in,
          DecimalVectorBlock* out) {
  std::vector<Decimal> v(in.size());
  Size in_nnz = in[0].nnz;
  Size in_size = in.size();

  for (Size i = 0; i < in_nnz; ++i) {

    v[0] = in[0].values[i];
    v[1] = in[1].values[i];

    out->values[i] = (*f)(&v);
  }
  out->nnz = in_nnz;
}
```

Krao(Hadamard)

- Indirect memory access & data dependency prevent vectorization

```
void krao(const FilteredBitVectorBlock& A,
          const DecimalVectorBlock& B,
          FilteredDecimalVectorBlock *C) {
  Size i, nnz = 0;
  Size a_col_size = A.cols.size();

  for (i = 0; i < a_col_size - 1; ++i) {
    if ( A.cols[i] < A.cols[i+1] ) {
      C->values[nnz] = B.values[i];
      ++nnz;
    }
  }
  C->cols = A.cols;
  C->nnz = nnz;
}
```

Fold(Sum)

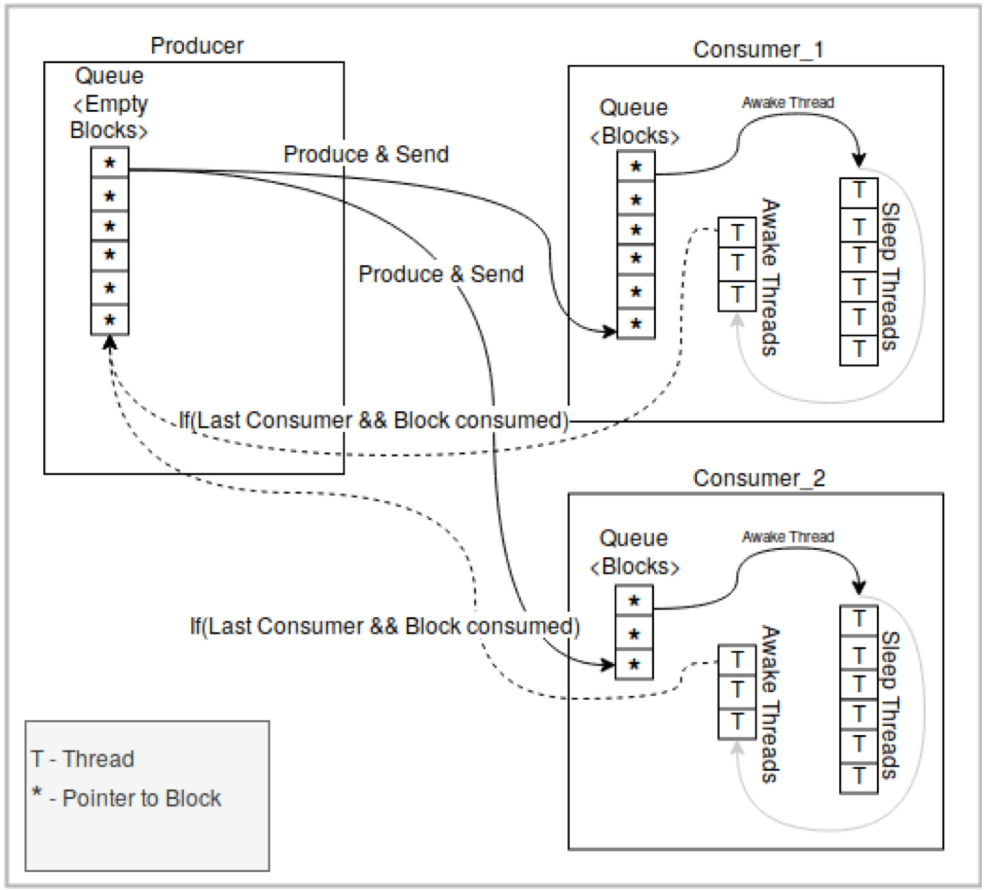
- The loop already supports vectorization

```
void sum(const FilteredDecimalVectorBlock& in,
         Decimal *acc) {
  for (auto& v : in.values) {
    (*acc) += v;
  }
}
```

```
LOOP BEGIN at engine/src/fold.cpp(11,18)
<Multiversioned v1>
   remark #15388: vectorization support: reference *U4d_V.U4d_V has aligned access   [ engine/src/
   fold.cpp(12,15) ]
   remark #15305: vectorization support: vector length 2
   remark #15399: vectorization support: unroll factor set to 4
   remark #15309: vectorization support: normalized vectorization overhead 1.450
   remark #15300: LOOP WAS VECTORIZED
   remark #15442: entire loop may be executed in remainder
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 5
   remark #15477: vector cost: 2.500
   remark #15478: estimated potential speedup: 1.900
   remark #15488: --- end vector cost summary ---
LOOP END
```

SPMC communication scheme

- Threads in the operations are local, similar to a thread pool for each operation
- Threads are awoken as soon as blocks are added to the queue
- Threads go back to sleep if there are no more blocks to consume in the queue



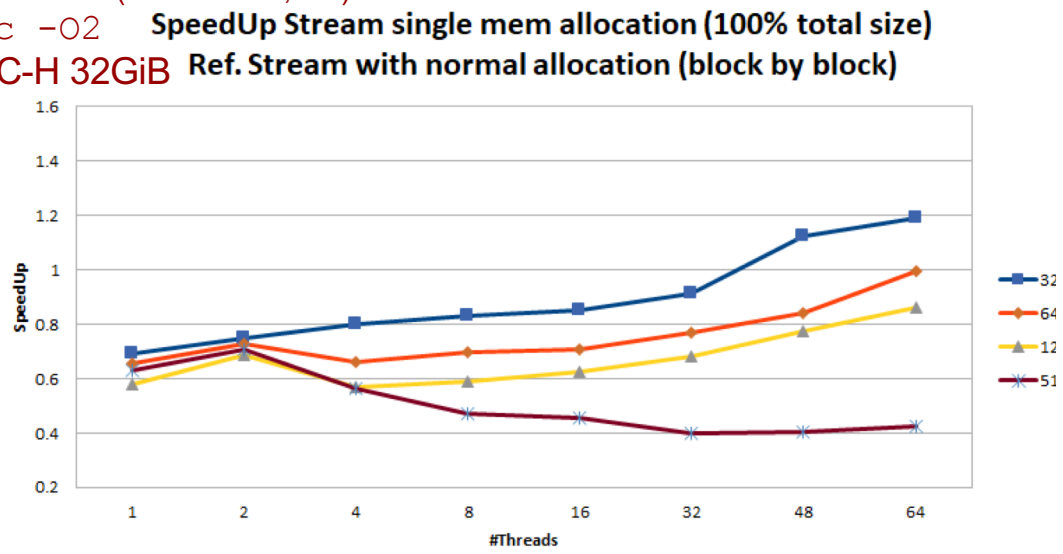
04-jun-19

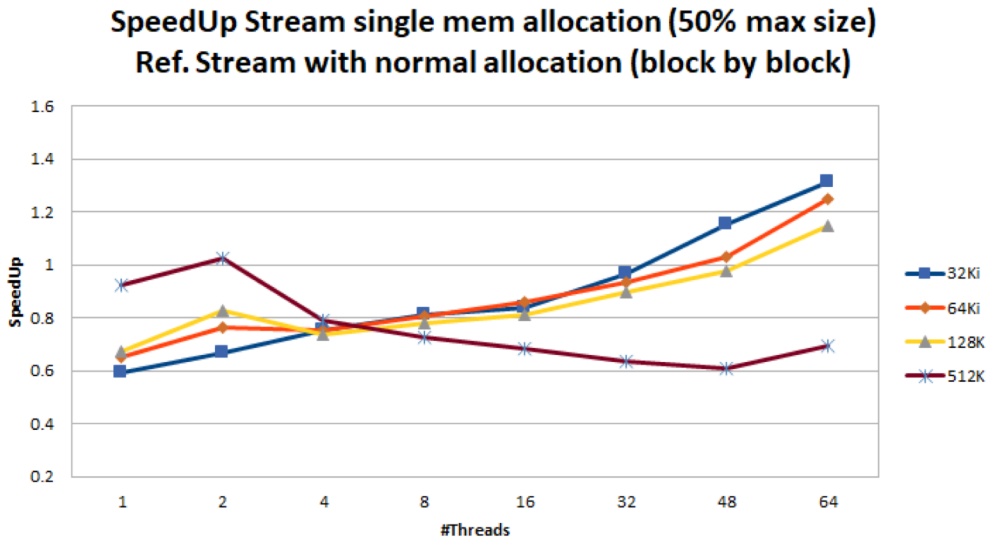
Stream with single mem alloc

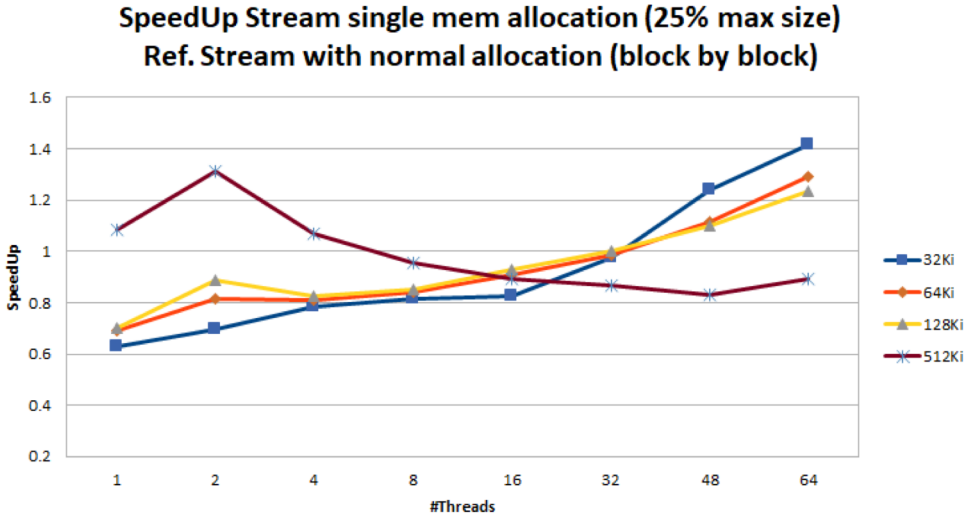
- Single memory allocation favors smaller block size versions, which require a greater number of allocations, however this is only true when the number of threads is high (32+), where the normal versions have degraded performance
- In the case of versions with larger block size (512Ki), the single memory allocation degrades its performance or does not bring significant gains
- Alternatively, allocate small set of blocks whenever it is necessary, rather than a single operation

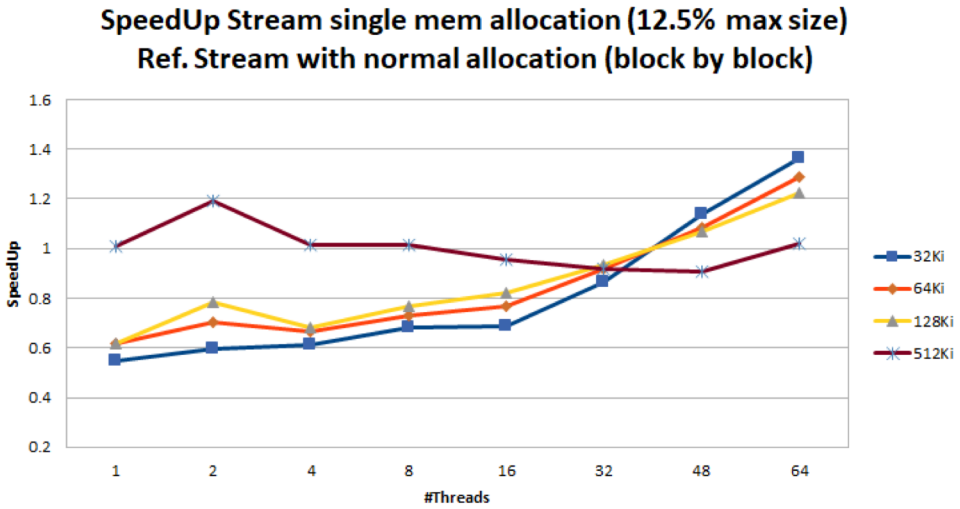
- Measurements are missing for sizes 256Ki, 1Mi and 2Mi

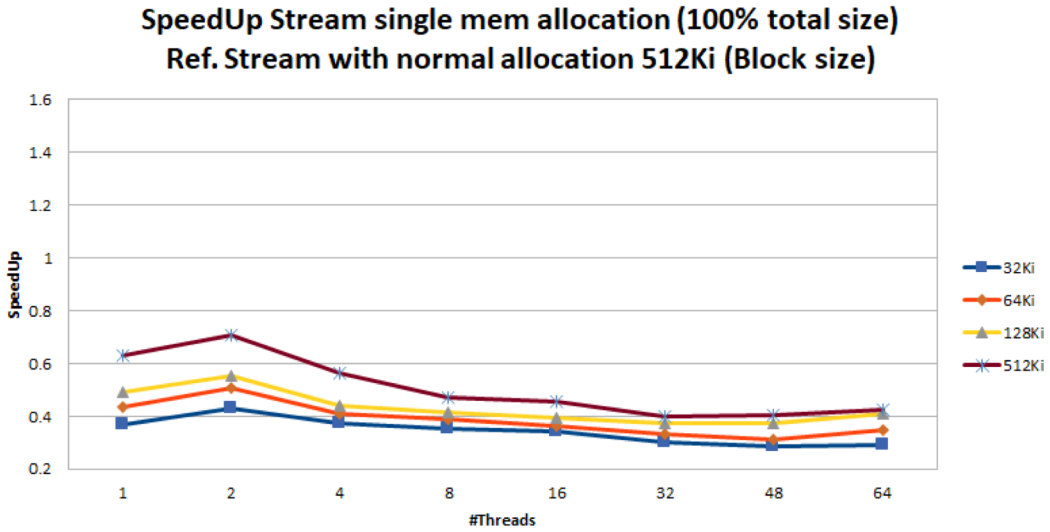
- Node 781 (2x 16-cores, HT)
- `icc -O2`
- TPC-H 32GiB

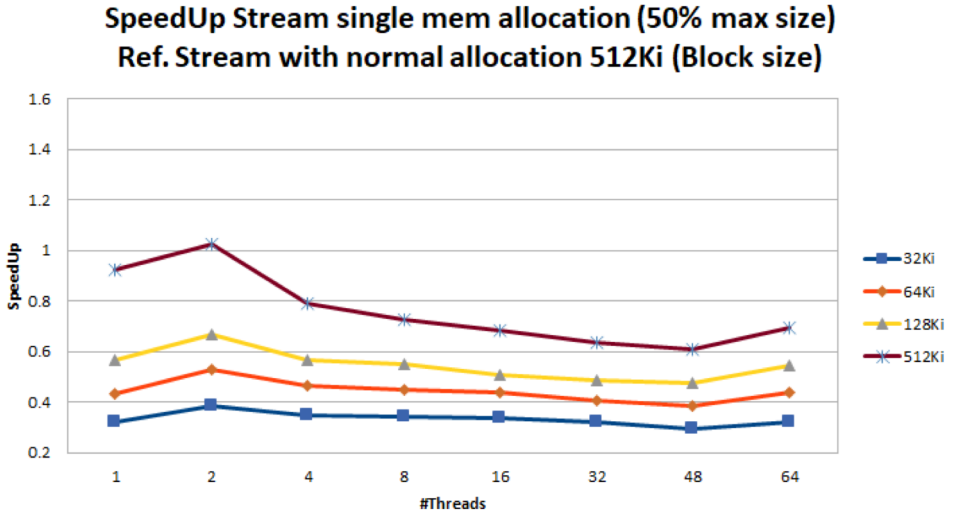


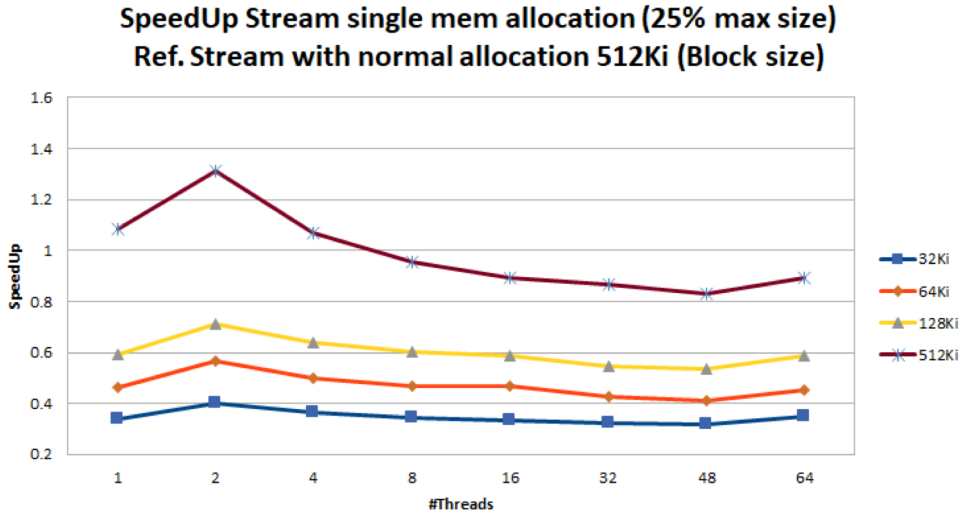


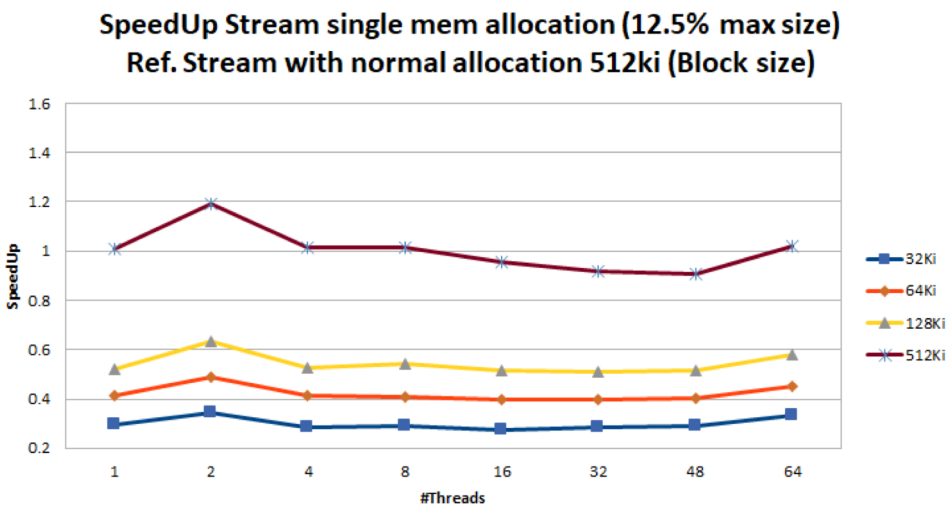





SpeedUp Stream single mem allocation (100% total size)
Ref. Stream with normal allocation 512Ki (Block size)
SpeedUp
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
1
2
4
8
16
32
48
64
#Threads
32Ki
64Ki
128Ki
512Ki


SpeedUp Stream single mem allocation (50% max size)
Ref. Stream with normal allocation 512Ki (Block size)
SpeedUp
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
1
2
4
8
16
32
48
64
#Threads
32Ki
64Ki
128Ki
512Ki

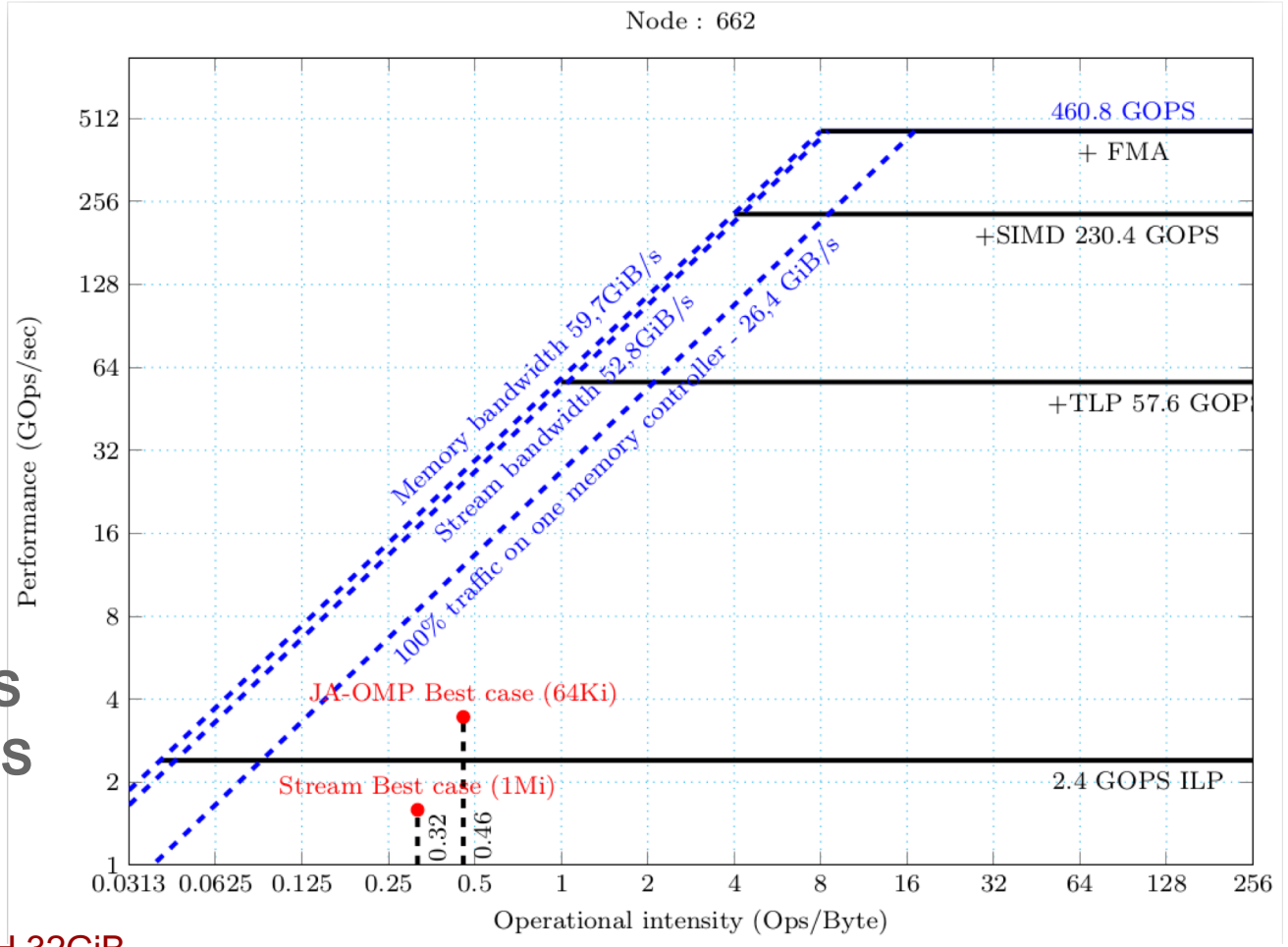

SpeedUp Stream single mem allocation (25% max size)
Ref. Stream with normal allocation 512Ki (Block size)
SpeedUp
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
1
2
4
8
16
32
48
64
#Threads
32Ki
64Ki
128Ki
512Ki


SpeedUp Stream single mem allocation (12.5% max size)
Ref. Stream with normal allocation 512ki (Block size)
SpeedUp
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
1
2
4
8
16
32
48
64
#Threads
32Ki
64Ki
128Ki
512Ki

10-jun-19

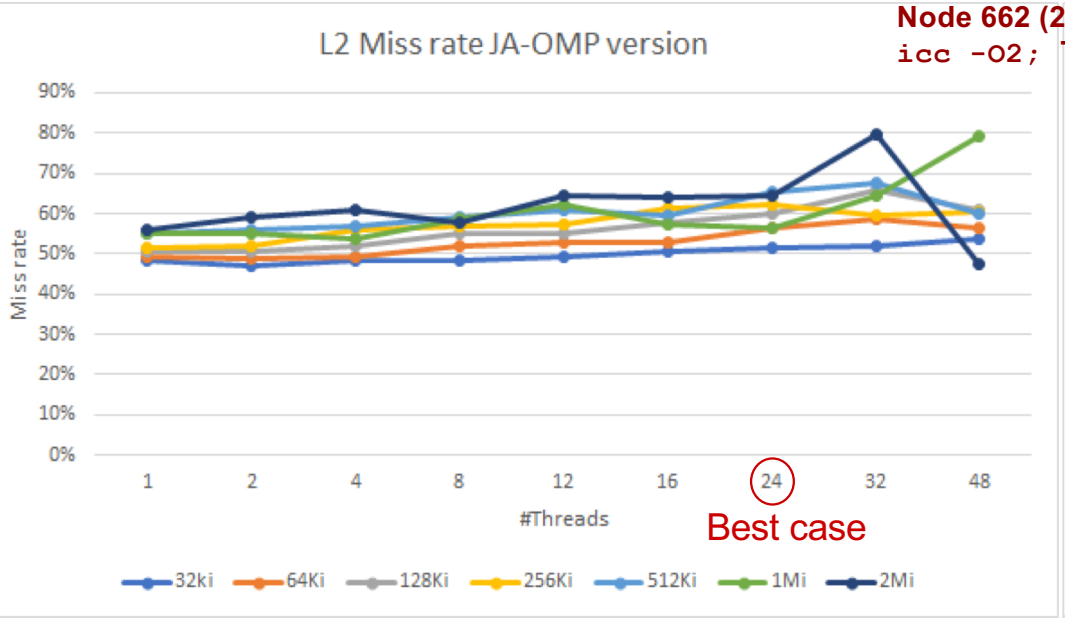
Roofline

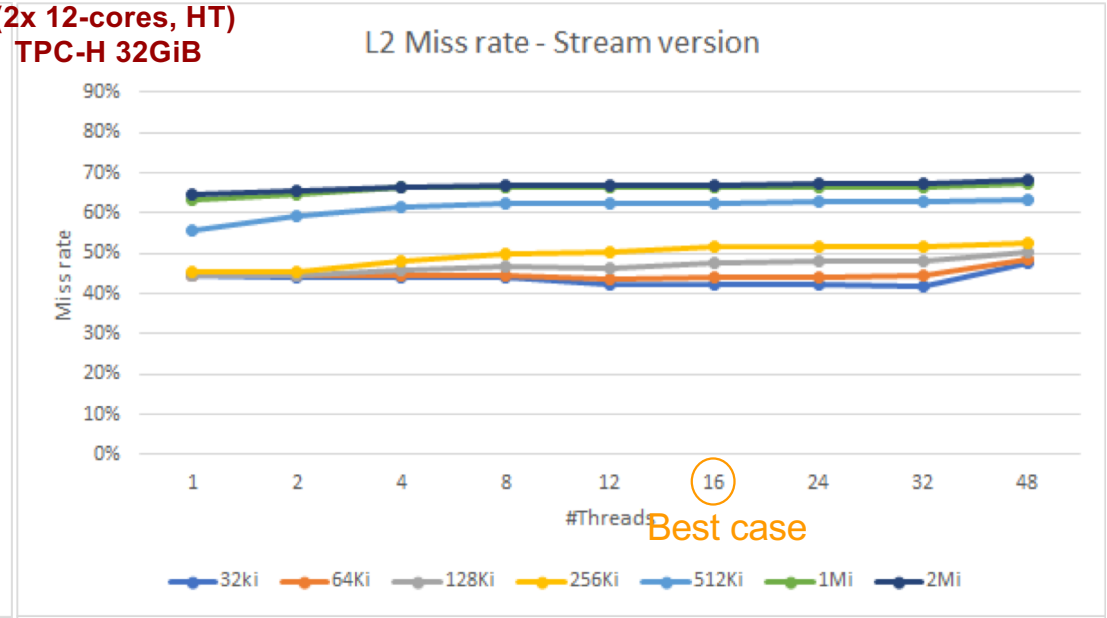
- Intel Xeon E5-2695 v2 (662)
- Frequency 2.4GHz
- Ivy Bridge
- AVX-256 & FMA (4 add + 4 mul)
- ILP -> 2.4GOPS
- +TLP -> 24*2.4=57.6GOPS
- +SIMD -> 4*57.6=230.4GOPS
- +FMA -> 2*230.4=460.8GOPS

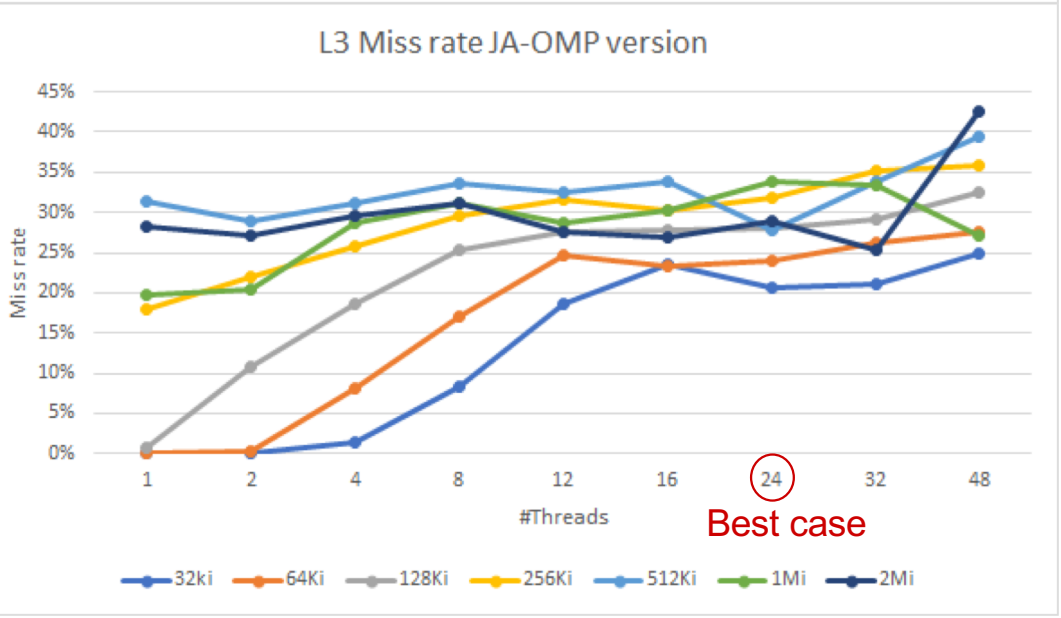


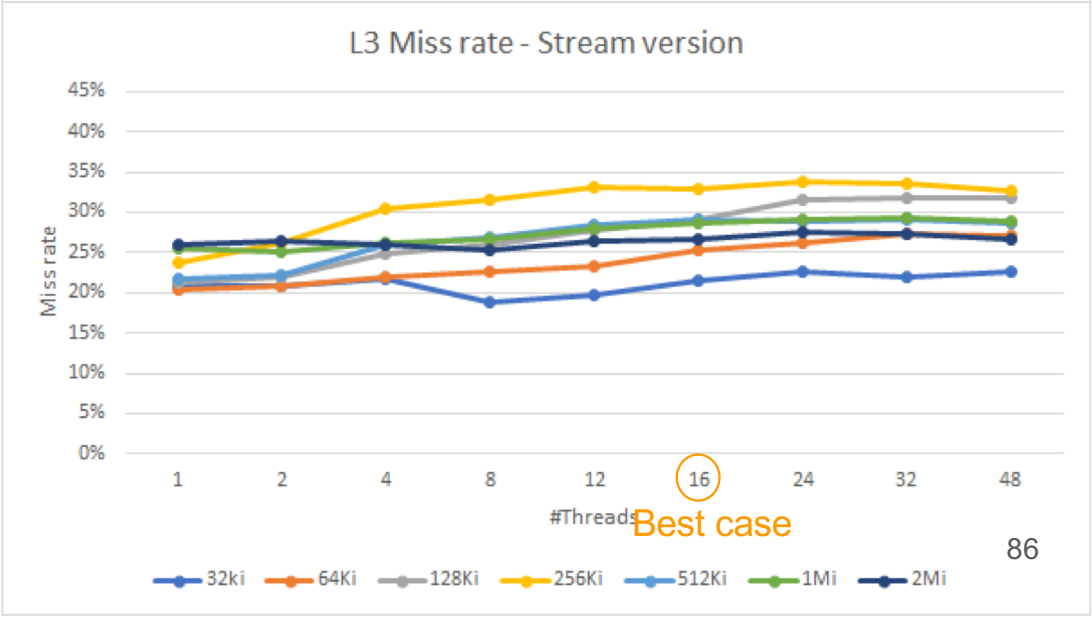
- Node 662 (2x 12-cores, HT) ; `icc -O2;` TPC-H 32GiB

Node 662 (2x 12-cores, HT)
`icc -O2;` TPC-H 32GiB

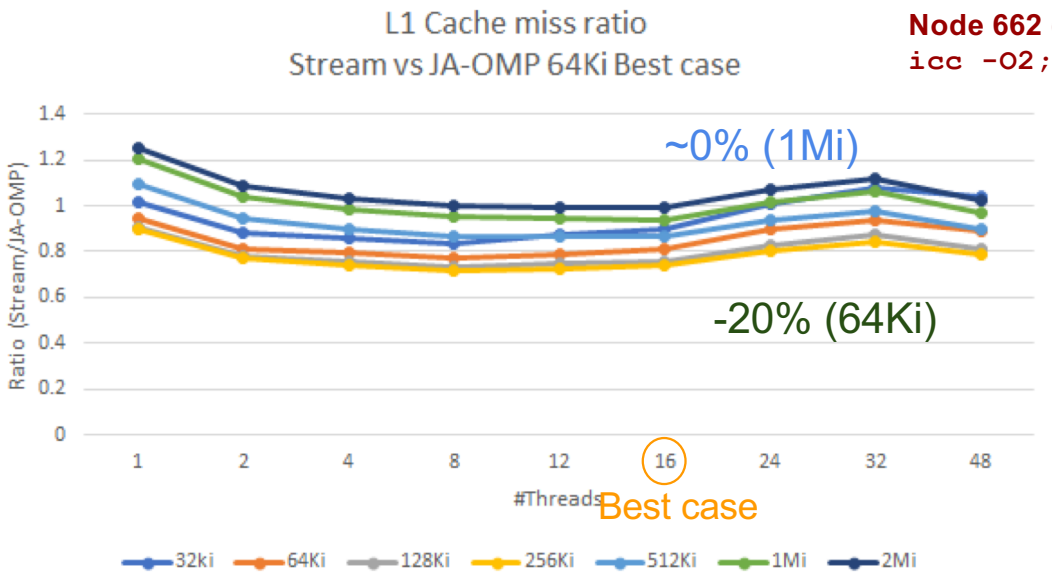


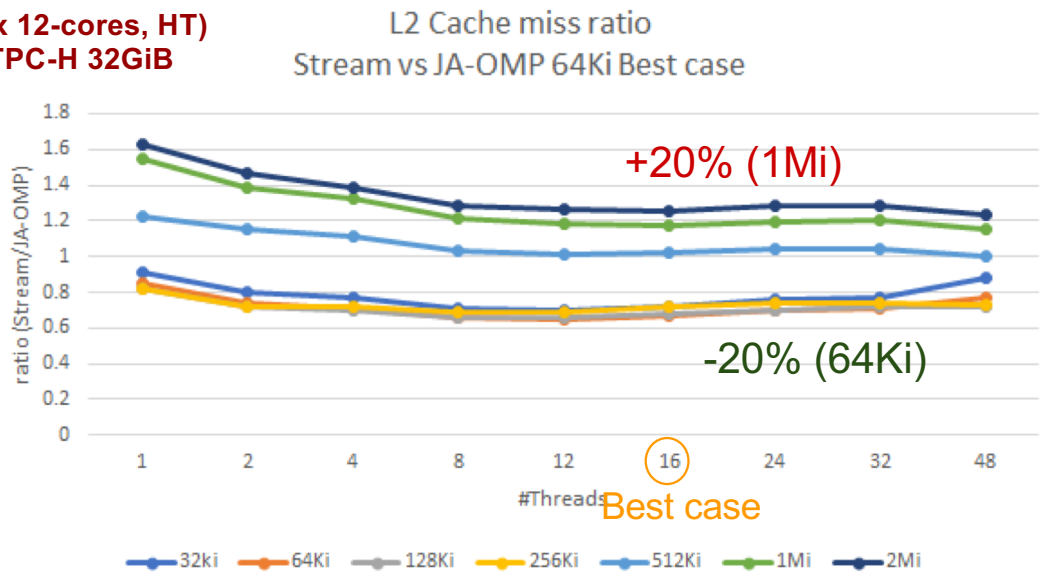


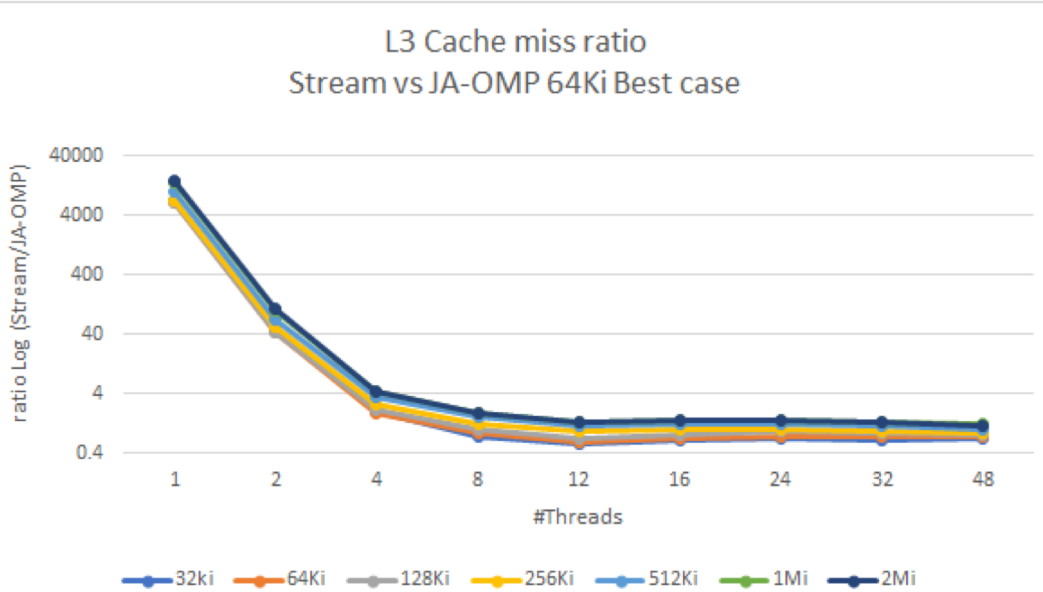


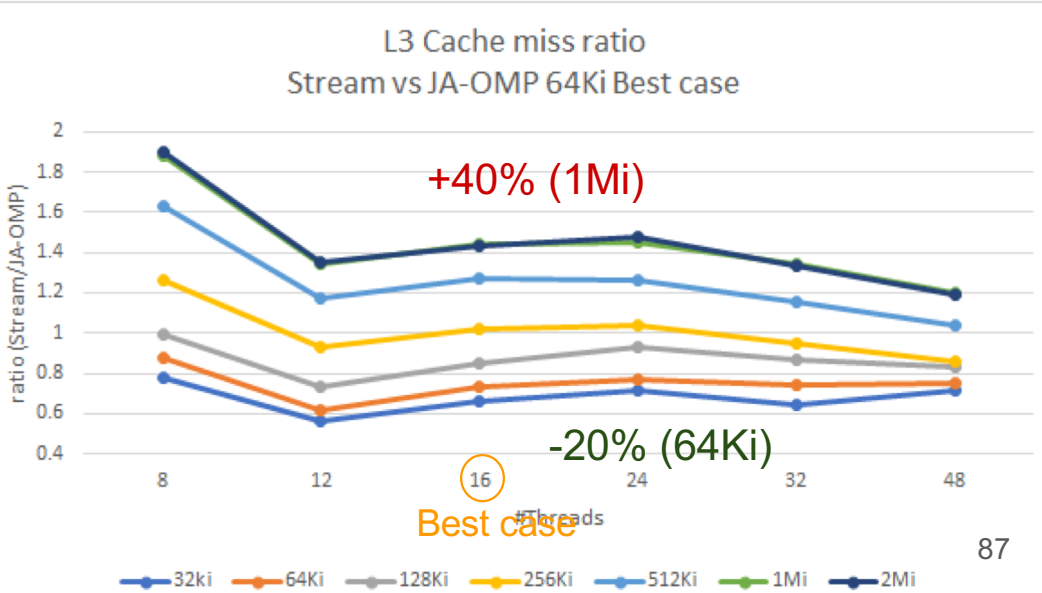


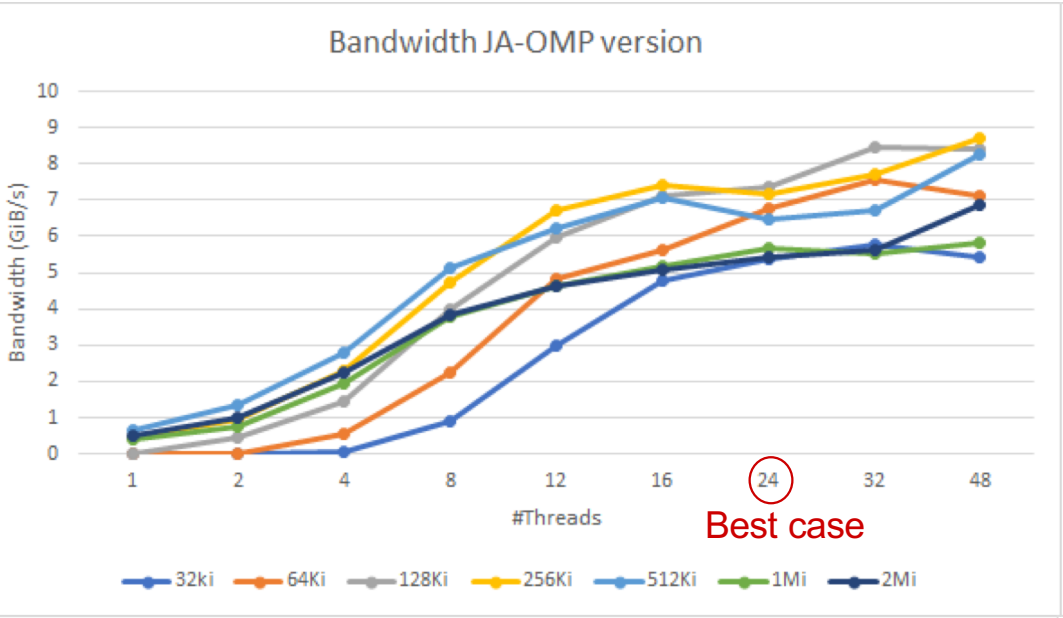
Node 662 (2x 12-cores, HT)
`icc -O2;` **TPC-H 32GiB**

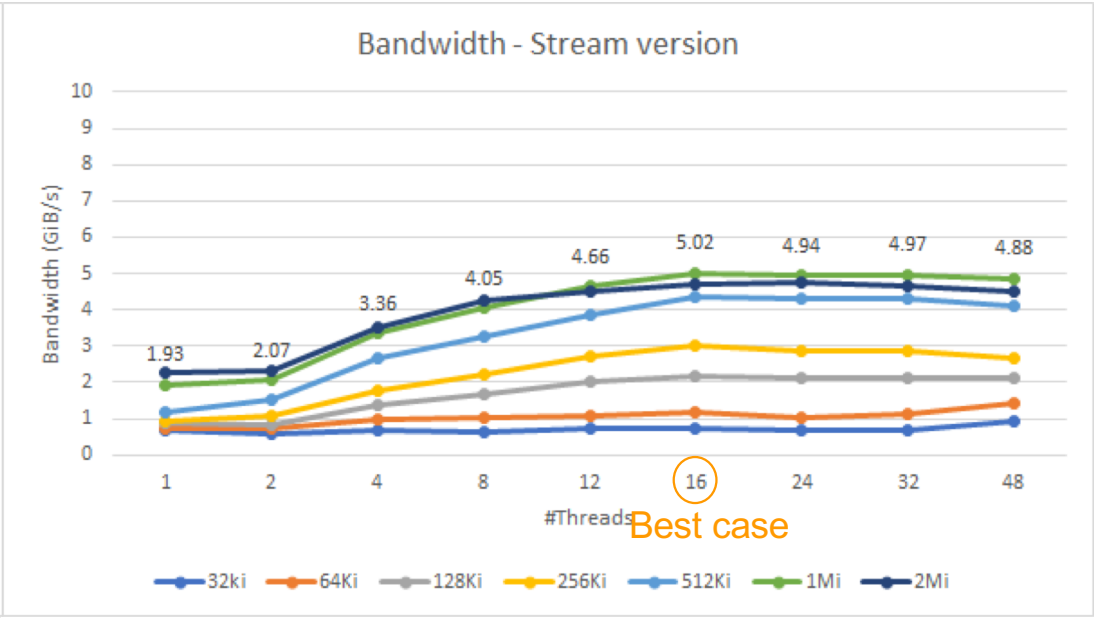


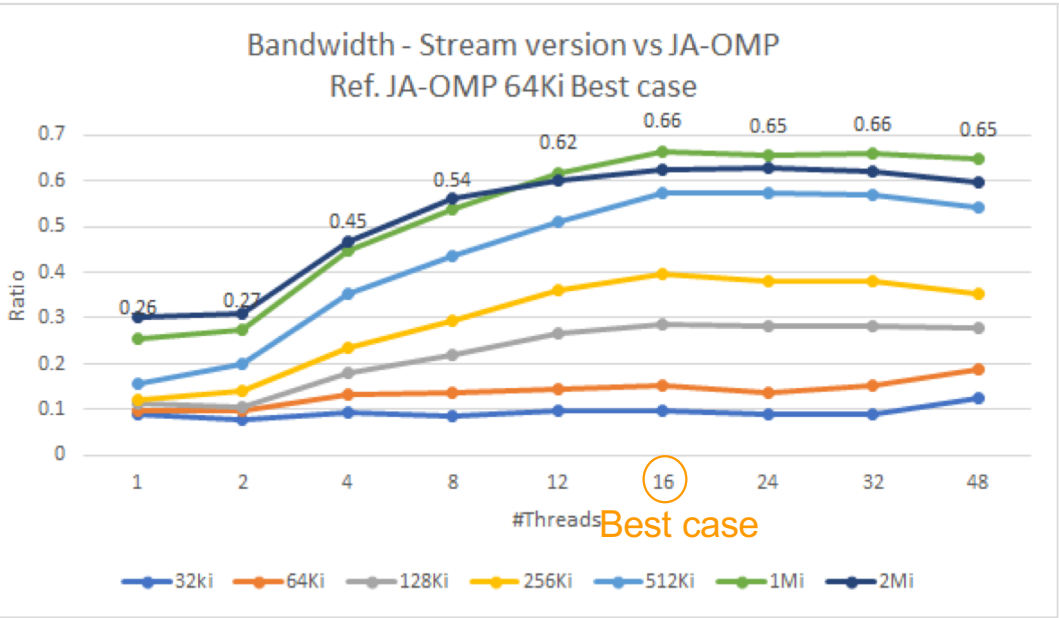






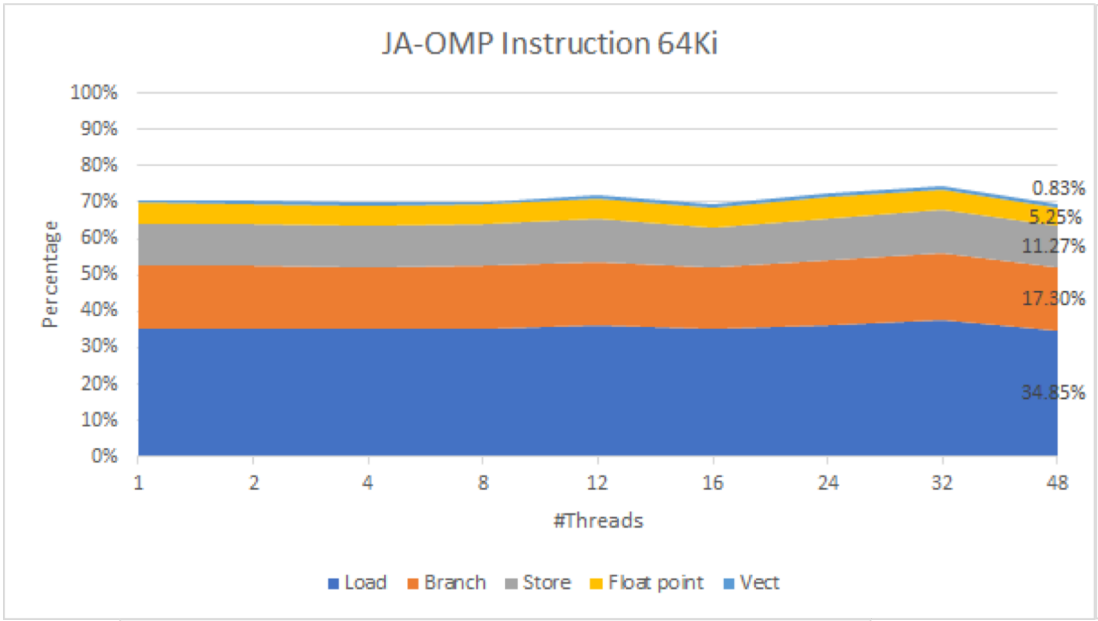


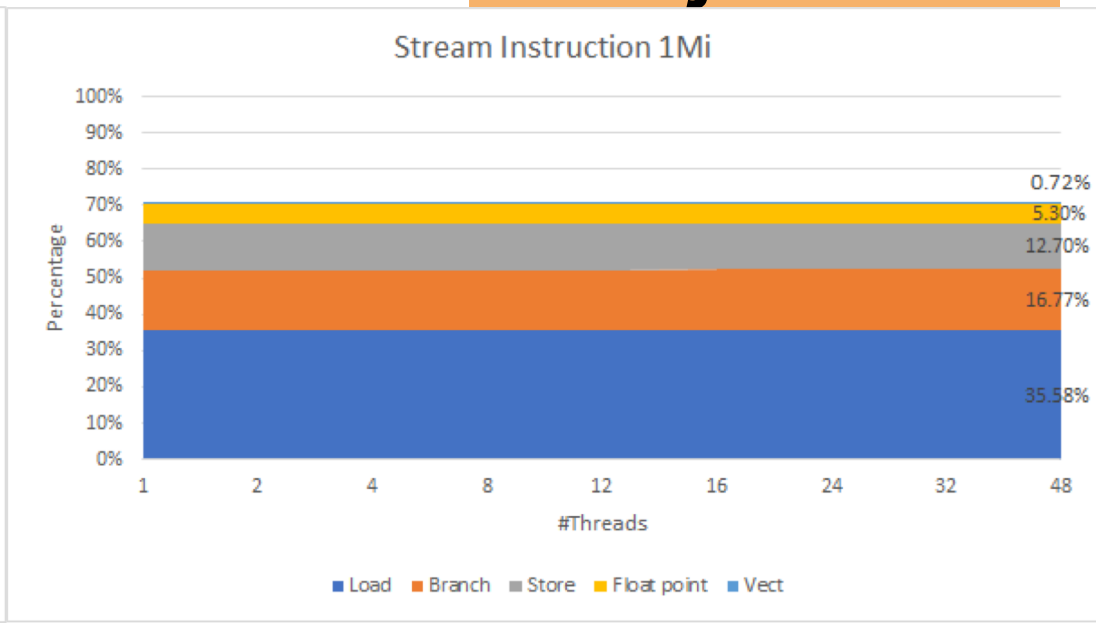


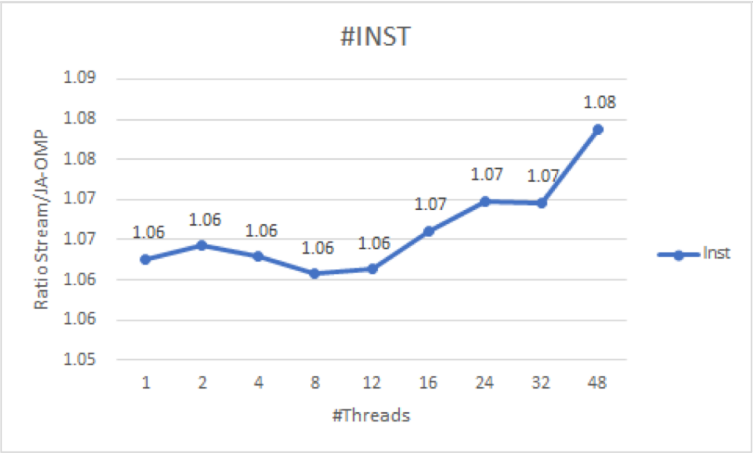


- Node 662 (2x 12-cores, HT)
- `icc -O2`
- TPC-H 32GiB

18-jun-19

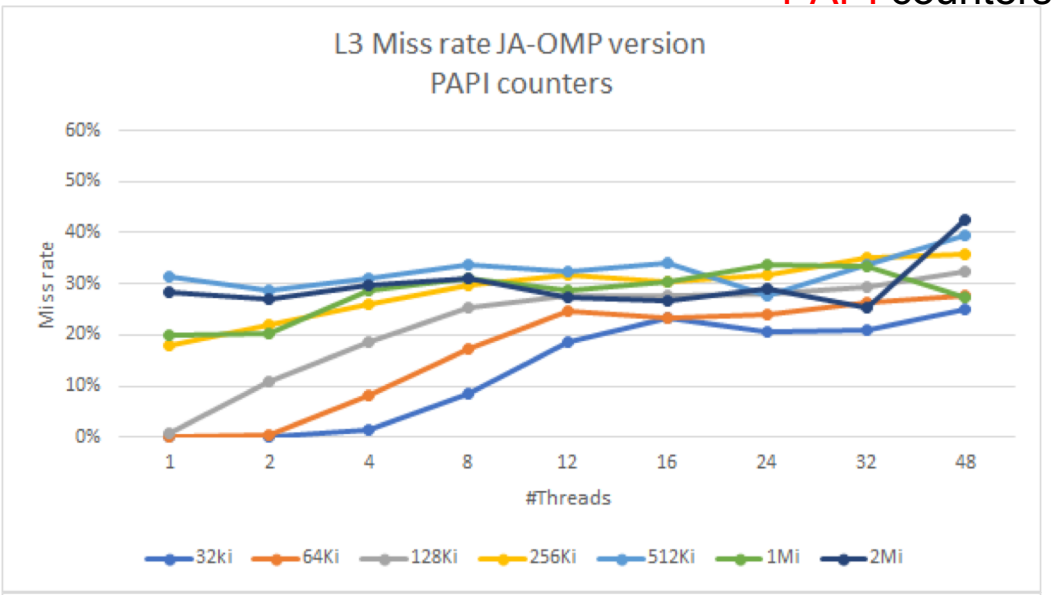

JA-OMP Instruction 64Ki
Percentage
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
1
2
4
8
12
16
24
32
48
#Threads
0.83%
5.25%
11.27%
17.30%
34.85%
Load
Branch
Store
Float point
Vect

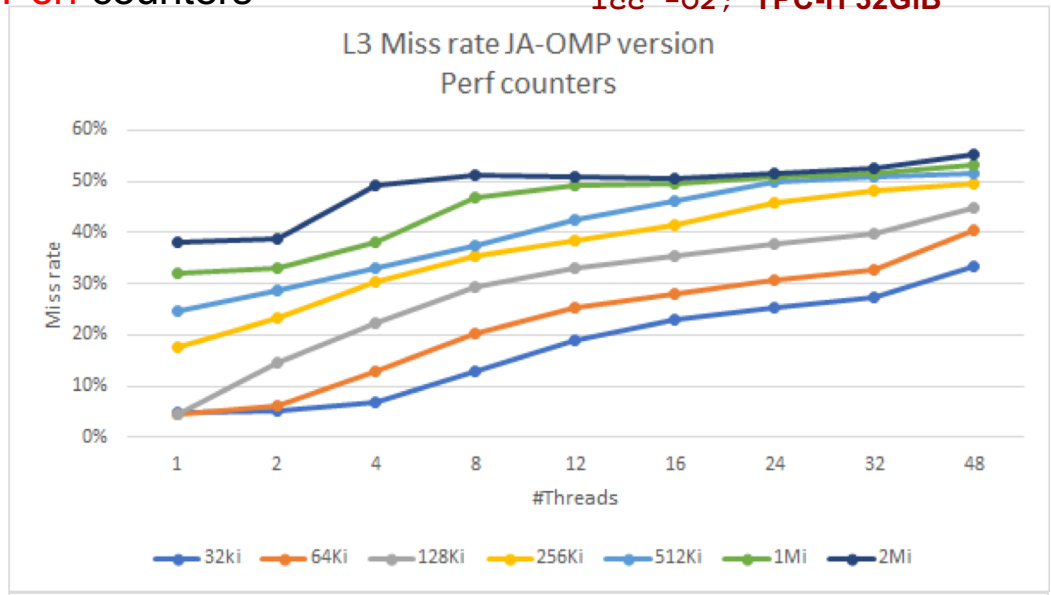

Stream Instruction 1Mi
Percentage
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
1
2
4
8
12
16
24
32
48
#Threads
0.72%
5.30%
12.70%
16.77%
35.58%
Load
Branch
Store
Float point
Vect

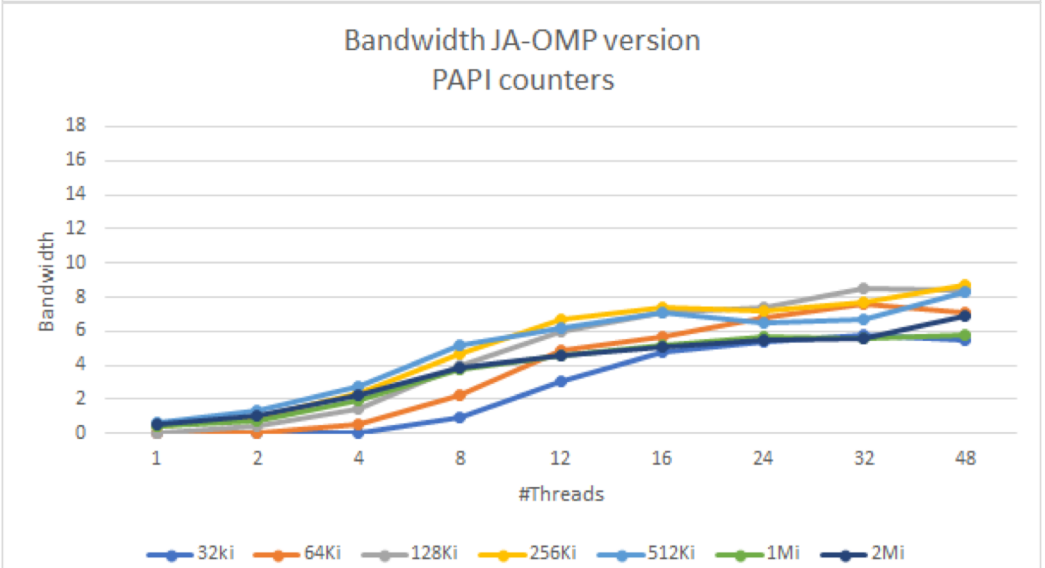

#INST
Ratio Stream/JA-OMP
1.09
1.08
1.08
1.07
1.07
1.06
1.06
1.05
1.06
1.06
1.06
1.06
1.06
1.07
1.07
1.07
1.08
1
2
4
8
12
16
24
32
48
#Threads
Inst

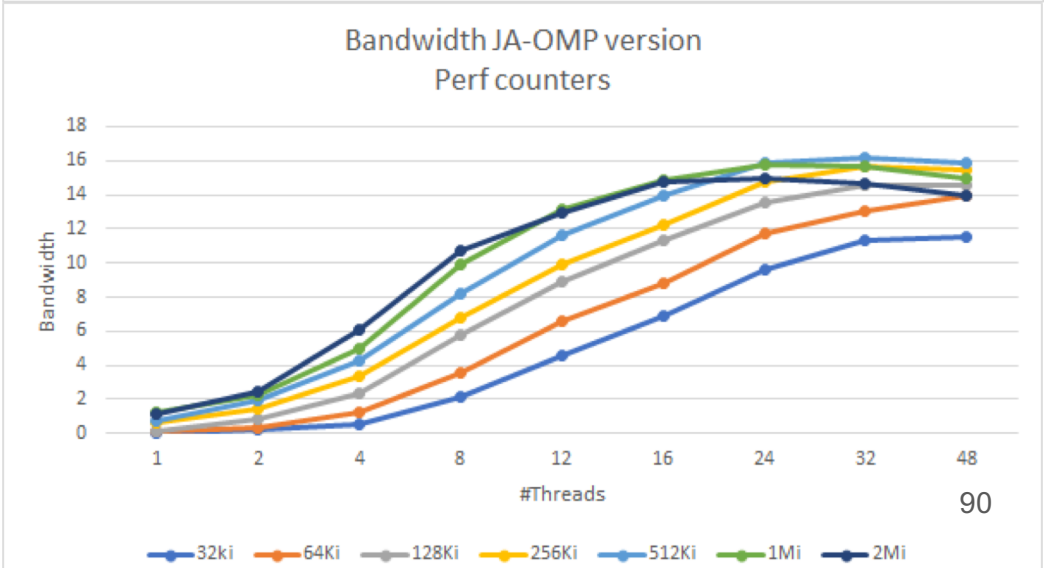
PAPI counters vs Perf counters

Node 662 (2x 12-cores, HT)
`icc -O2;` TPC-H 32GiB

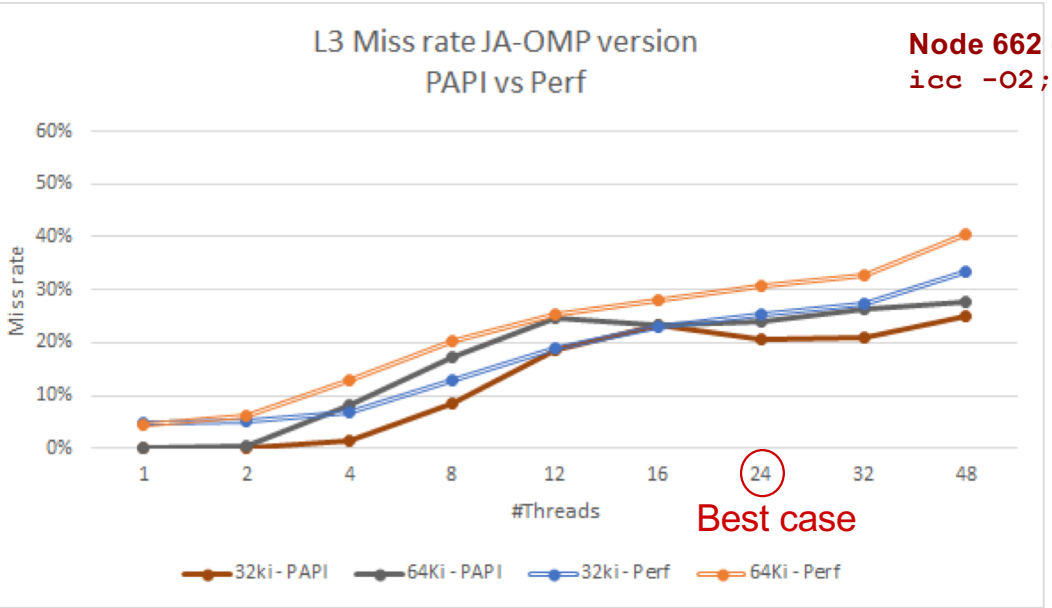


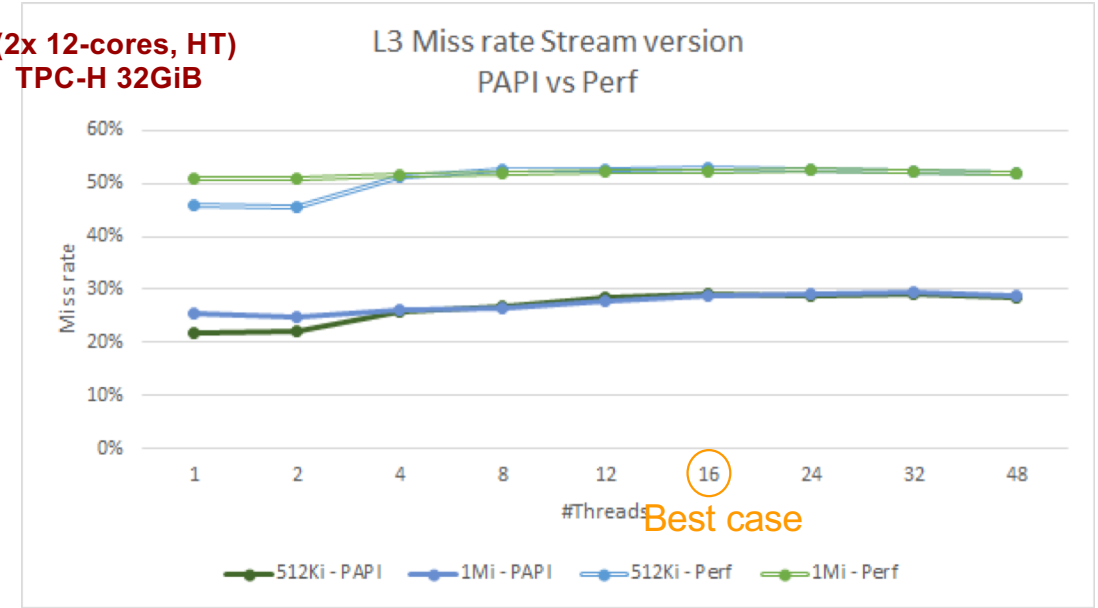


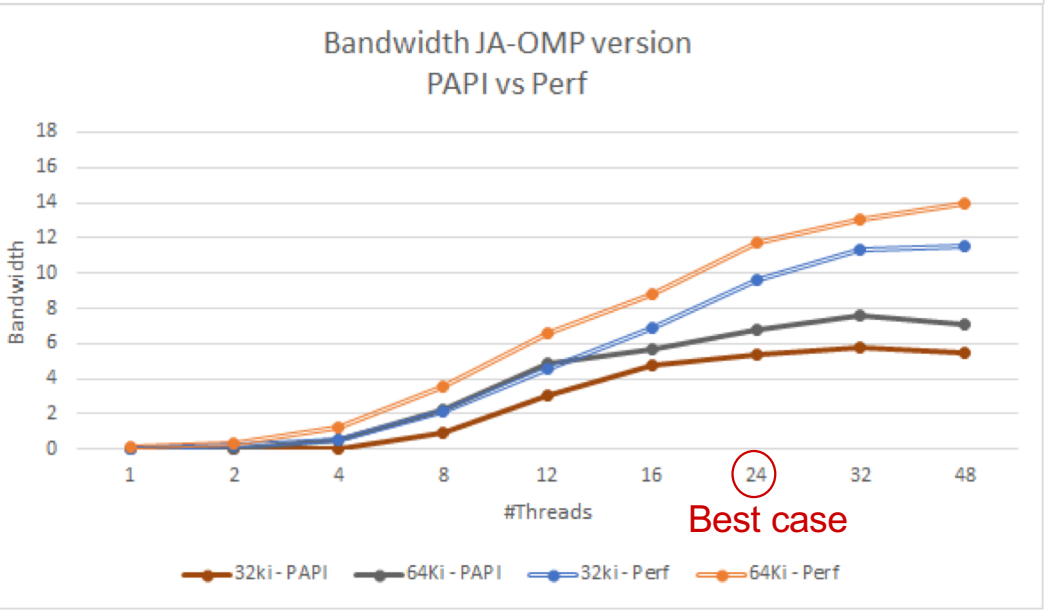


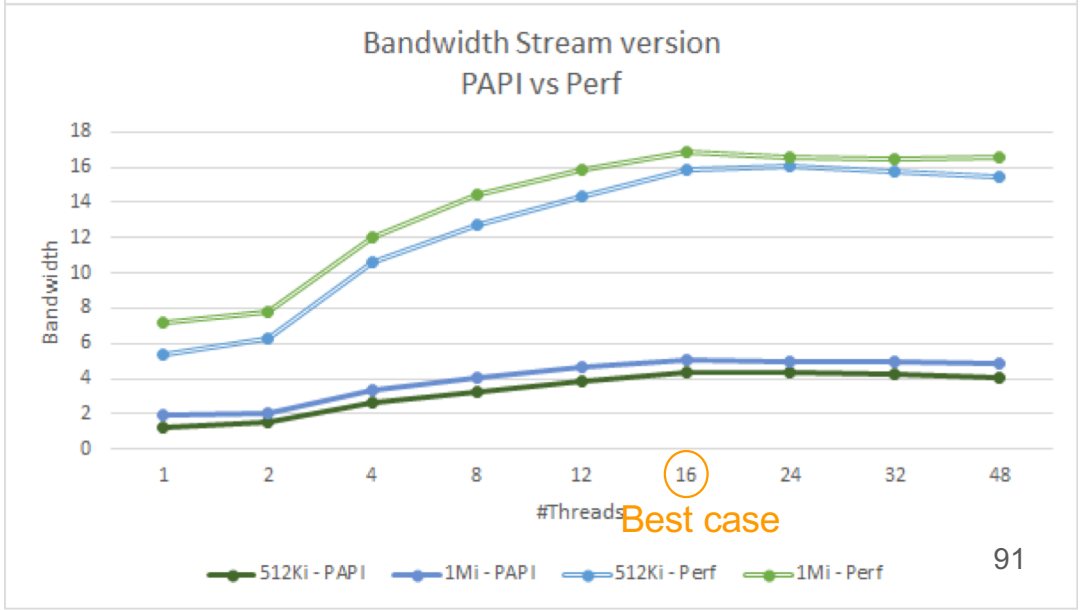


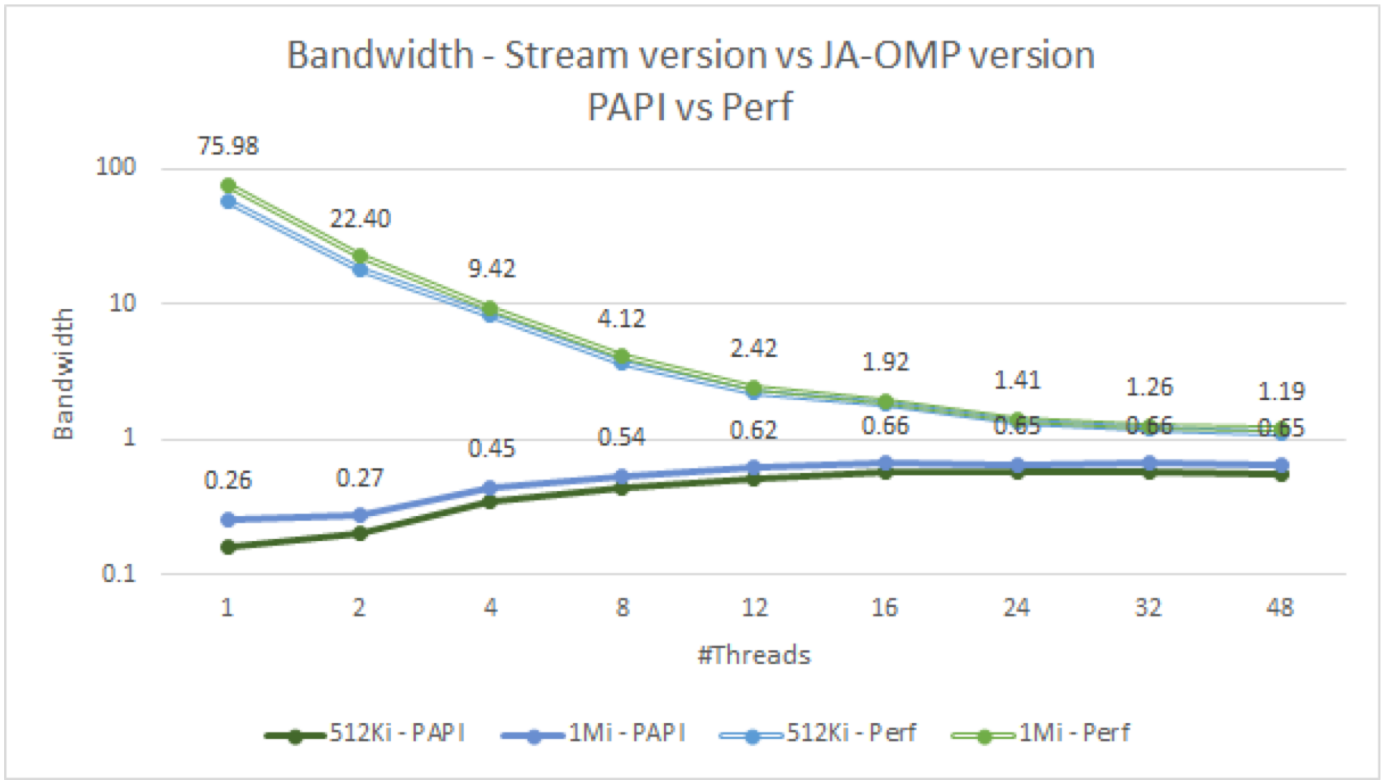
Node 662 (2x 12-cores, HT)
`icc -O2;` **TPC-H 32GiB**







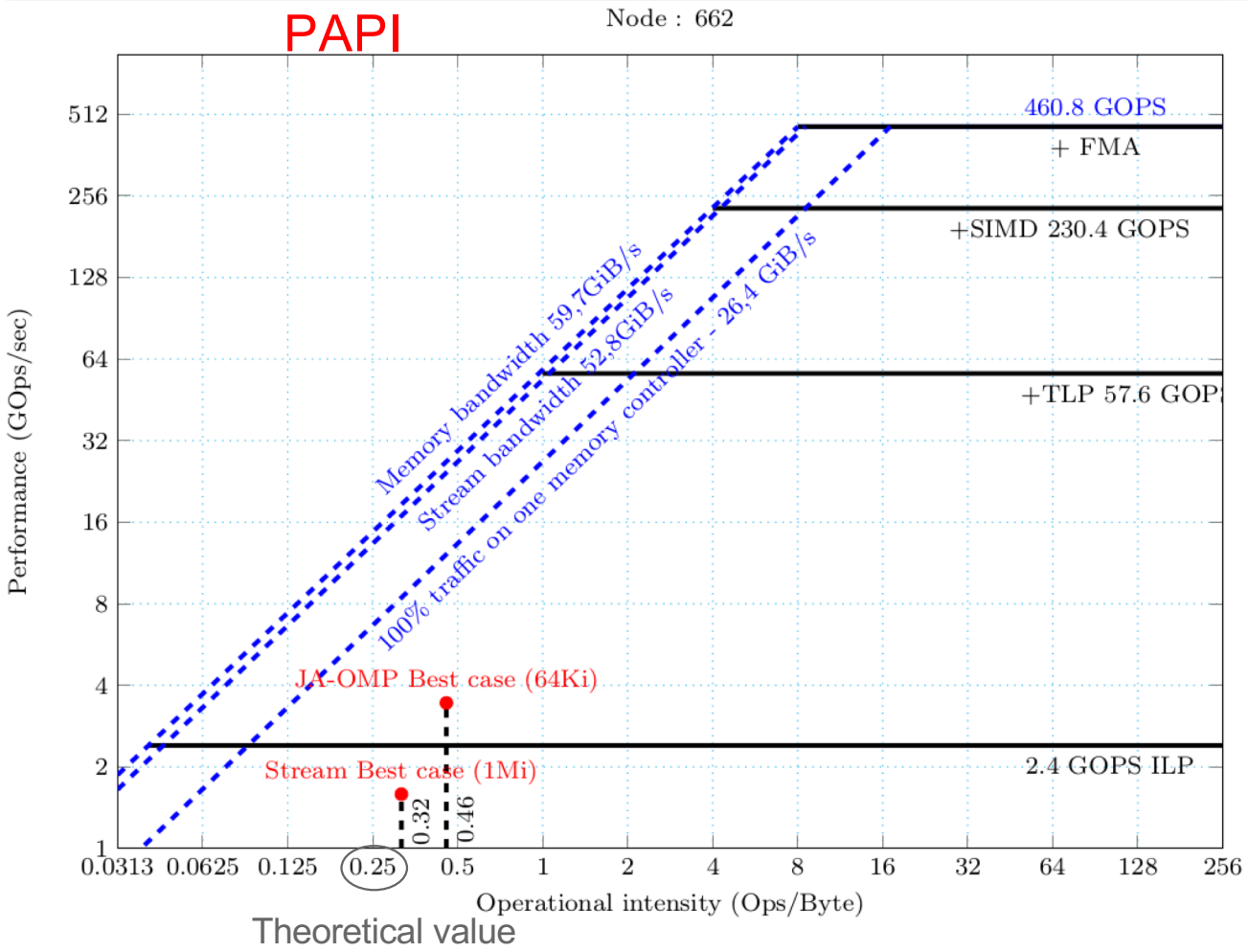




Node 662 (2x 12-cores, HT)
`icc -O2;` TPC-H 32GiB

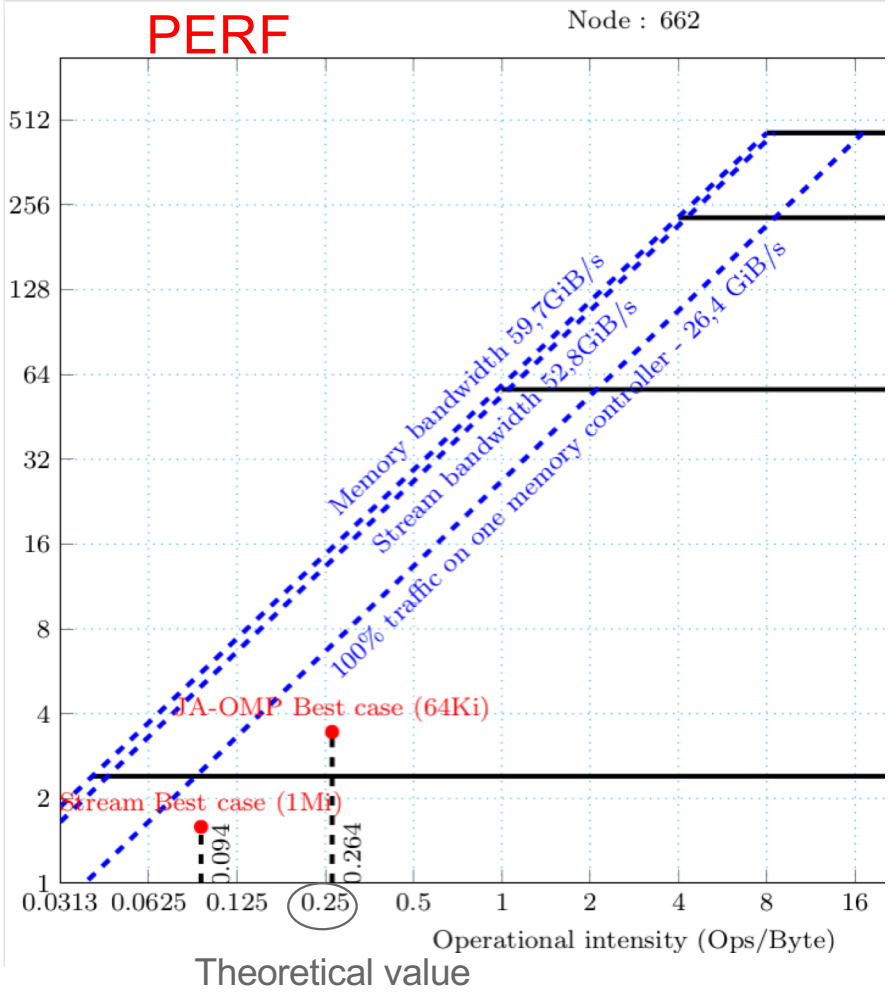
PAPI counters vs Perf counters

PAPI



Theoretical value

PERF



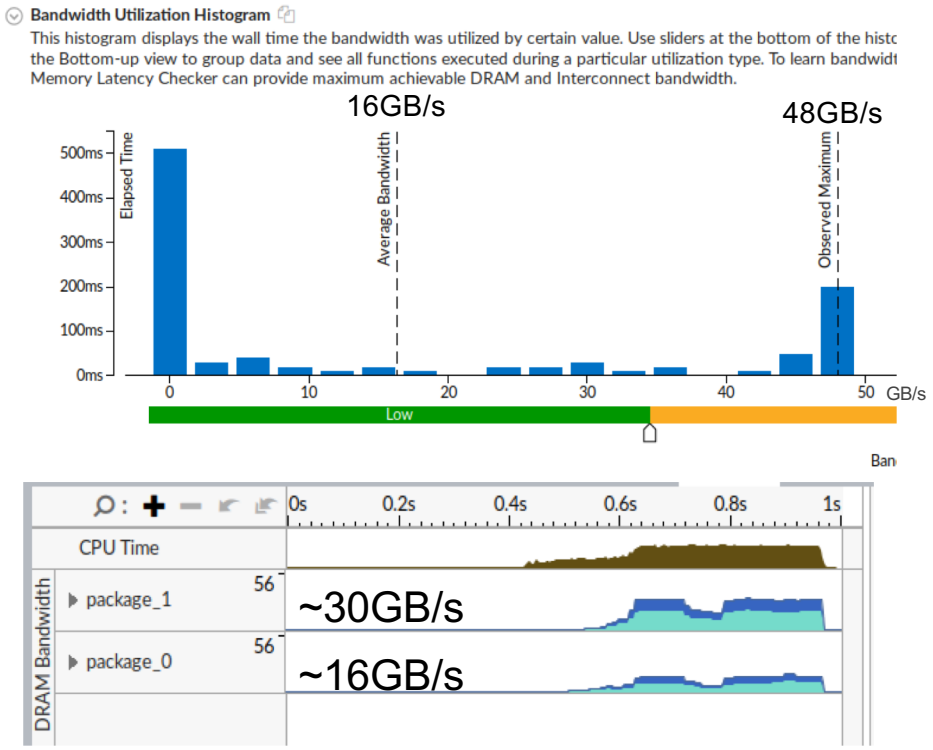
Theoretical value

Node 662 (2x 12-cores, HT)
`icc -O2;` **TPC-H 32GiB**

Bandwidth - Confirmation of values

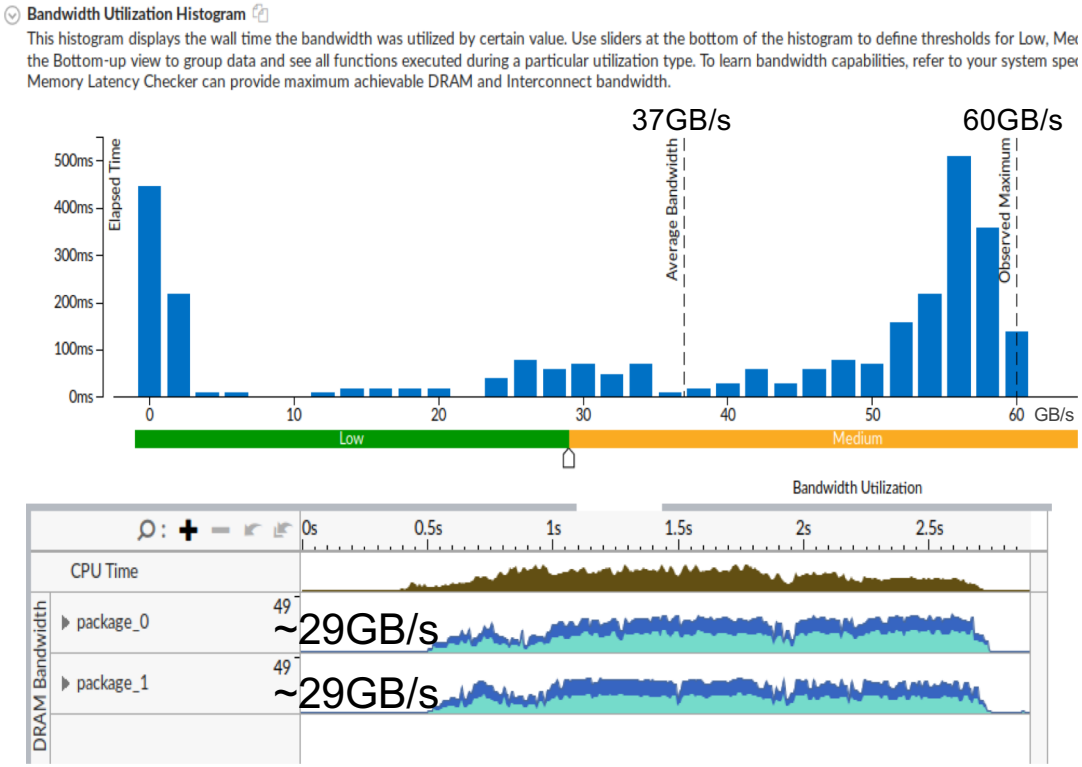
02-jul-19

JA-OMP version Best case (64Ki)



Average: 16 GB/s
PAPI: 8,4 GB/s
Perf: 16,8 GB/s

Stream version Best case (1Mi)



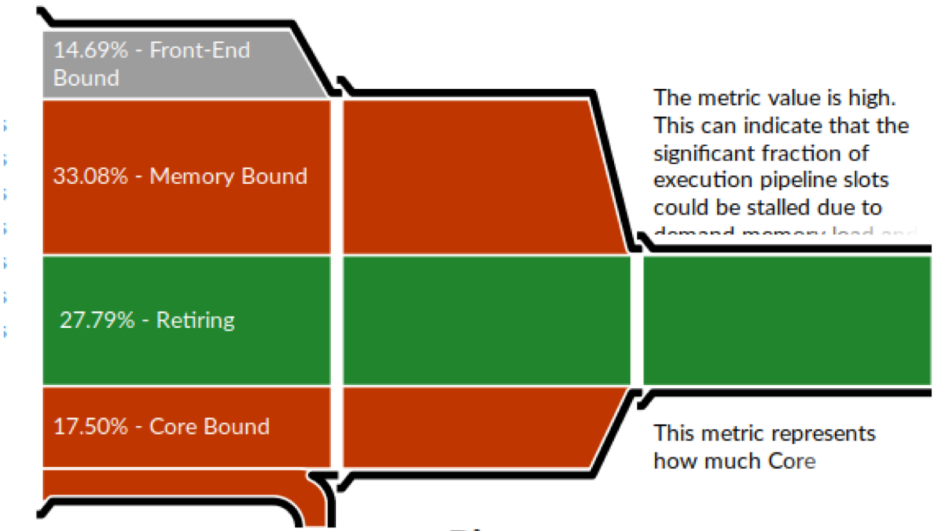
Average: 37 GB/s
PAPI: 5,2 GB/s
Perf: 17,8 GB/s

Node 662 (2x 12-cores, HT)
`icc -O2;` **TPC-H 32GiB**

Microarchitecture Exploration

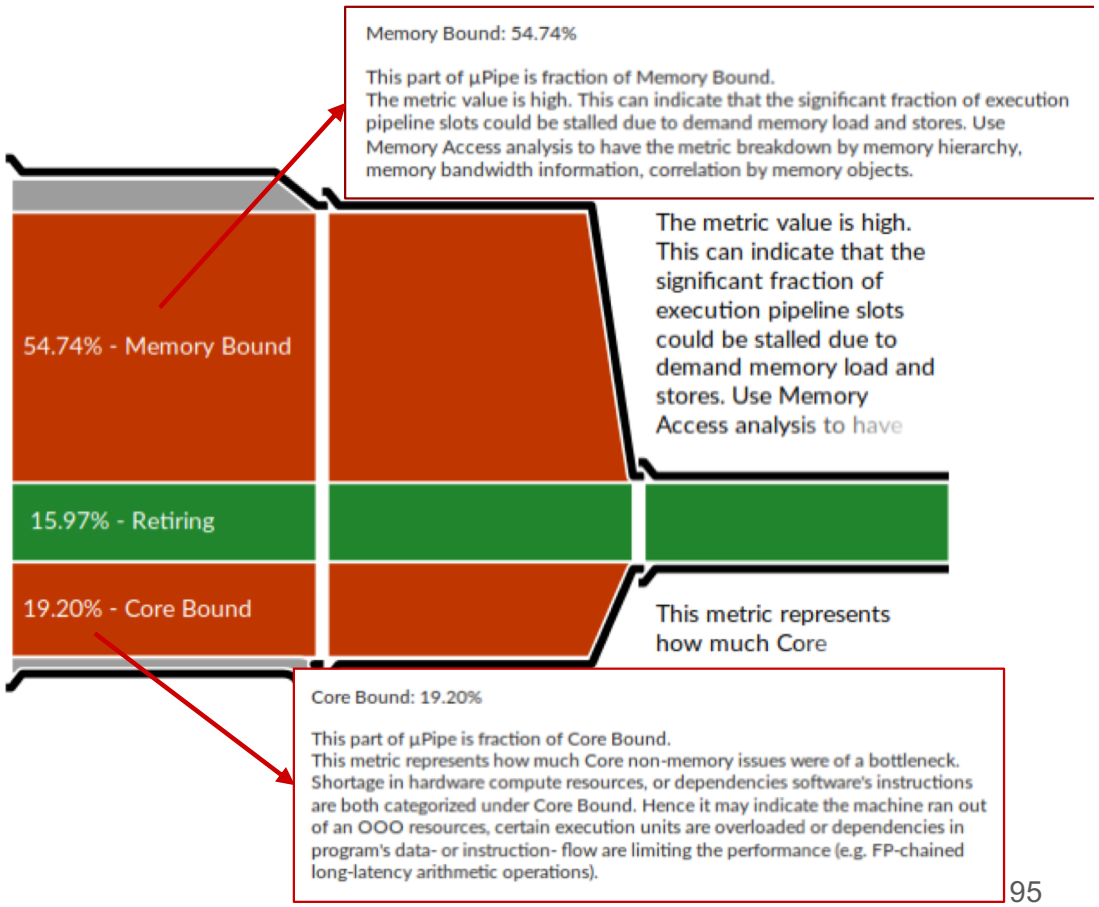
Explanation

- The percentage corresponds to the number of instructions that failed or were successfully executed by the pipeline

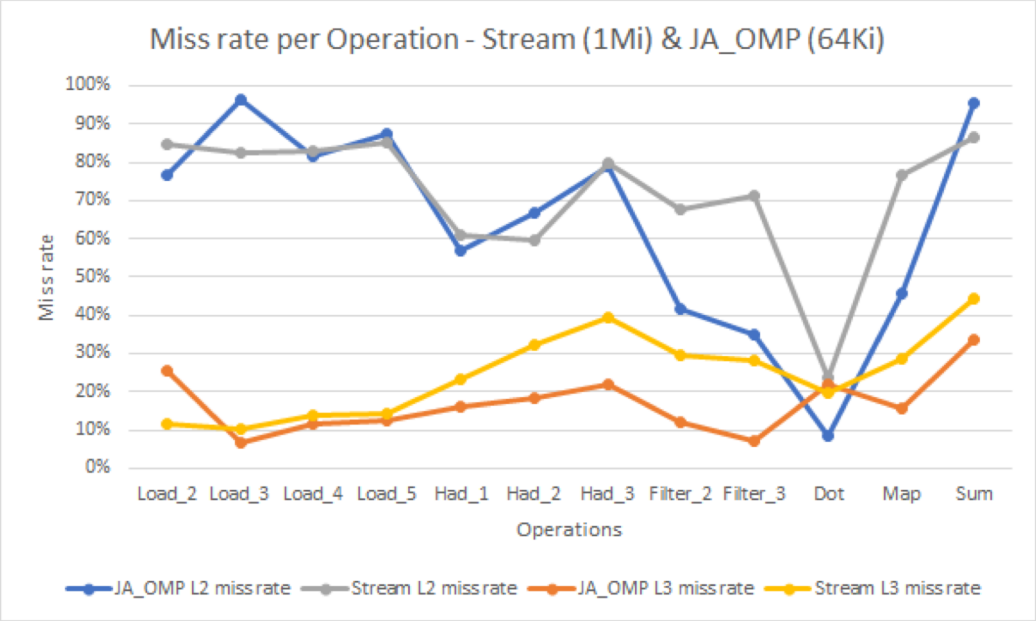


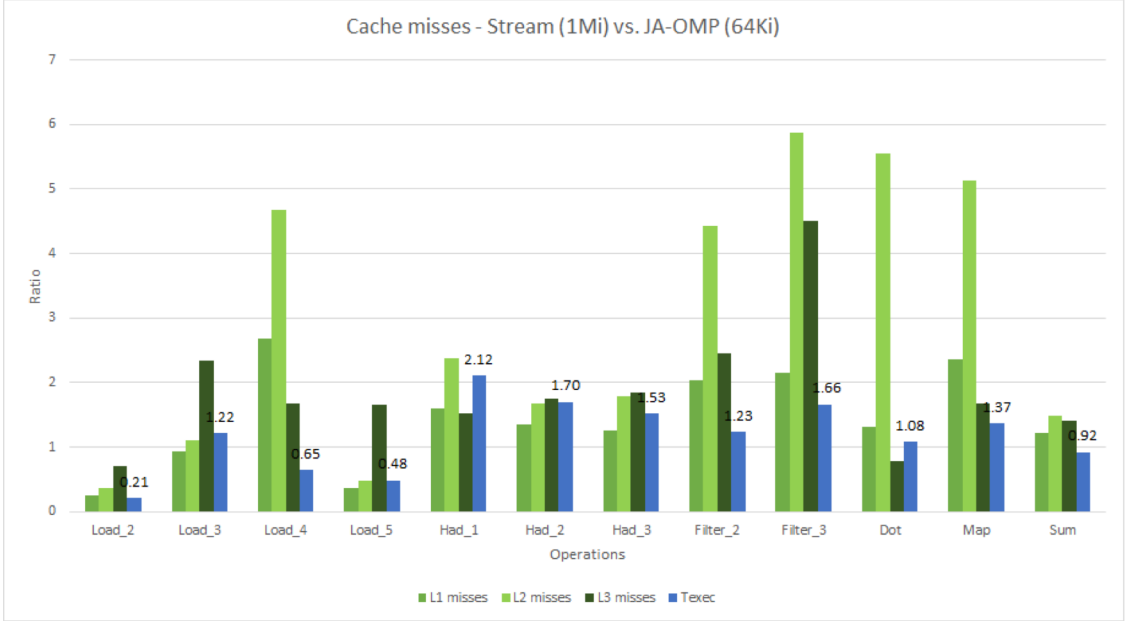
µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.



Comparison by Operations

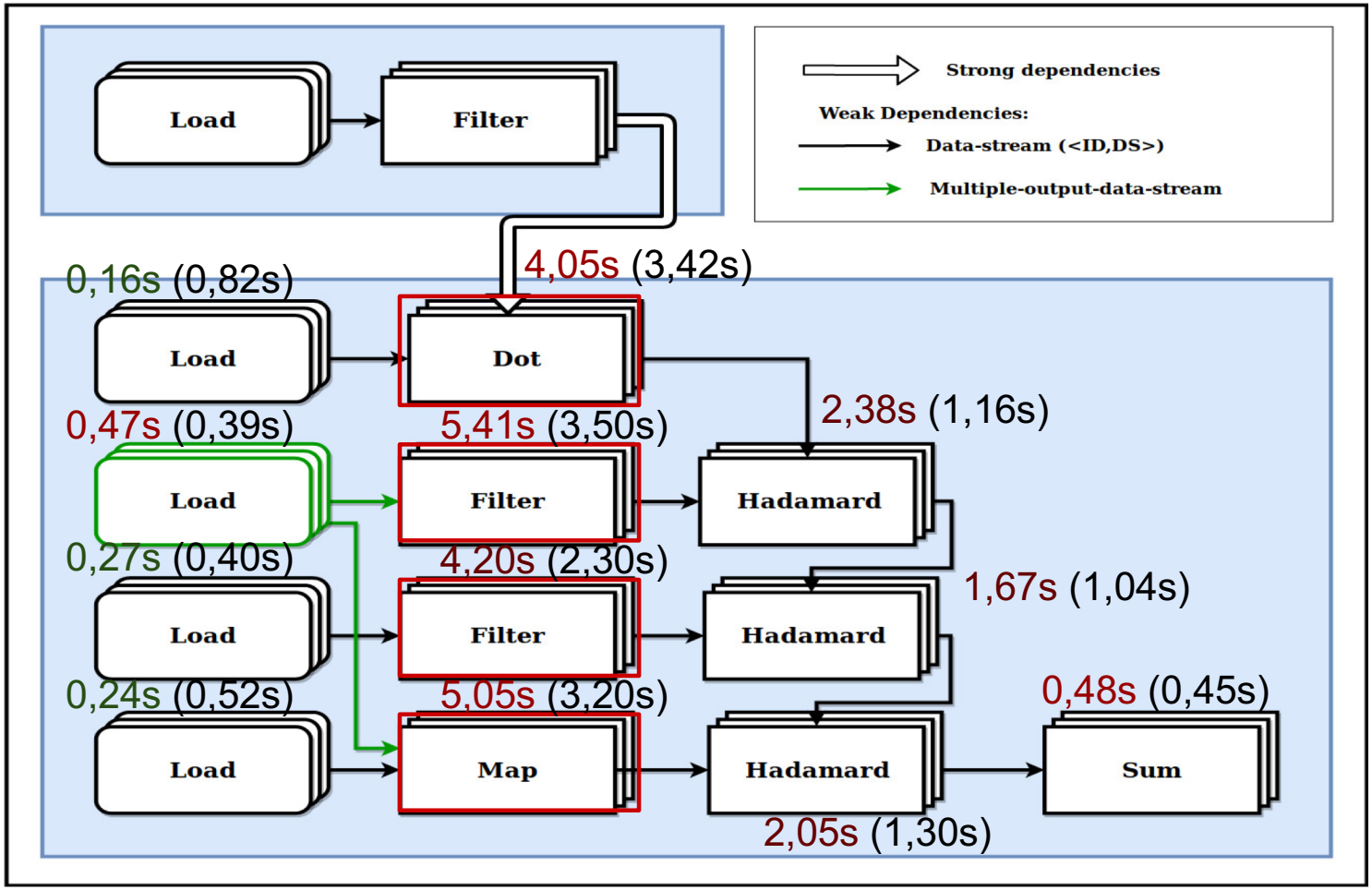




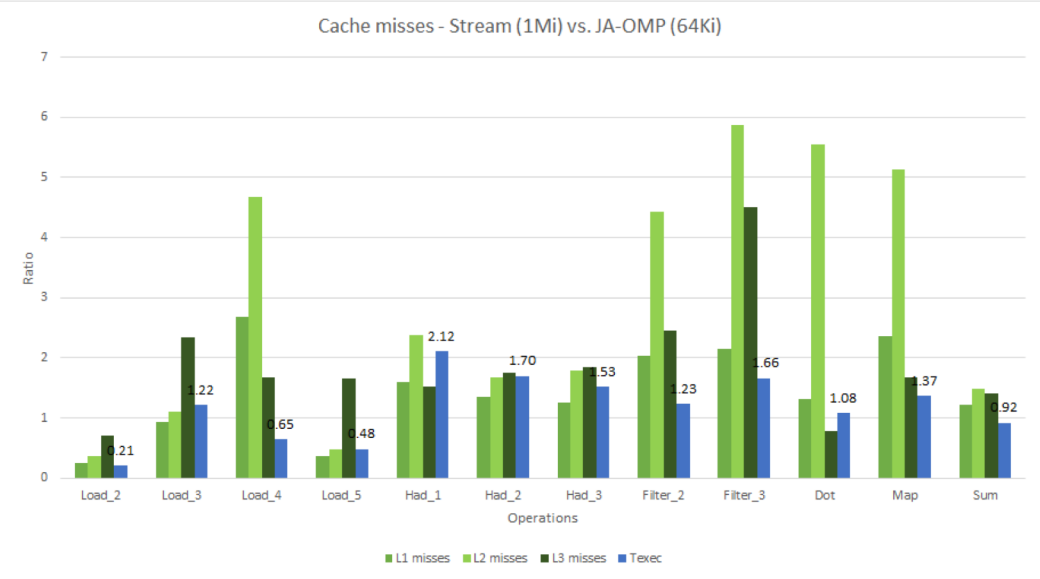
- In general it can be verified that in the stream version there are more misses in all the cache
- Which contributes to the loss of performance

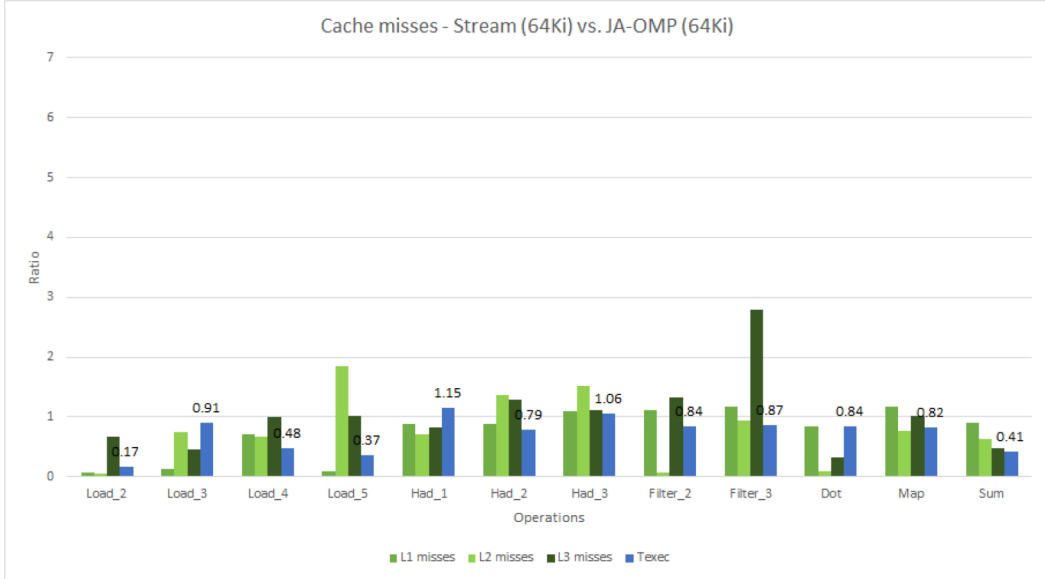
Node 662 (2x 12-cores, HT)
`icc -O2;` TPC-H 32GiB

Comparison by Operations - Stream (1Mi) vs JA_OMP (64Ki)



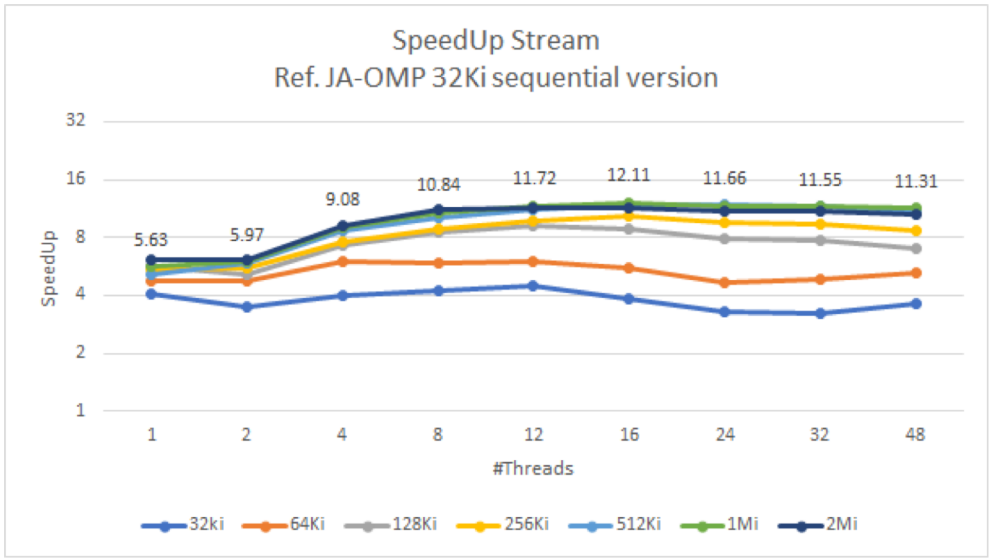
Node 662 (2x 12-cores, HT)
`icc -O2;` TPC-H 32GiB



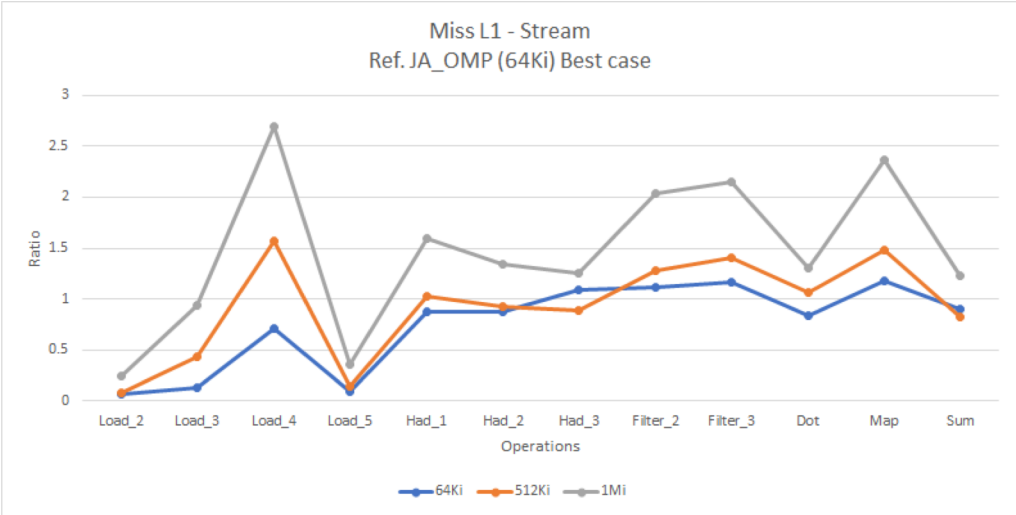


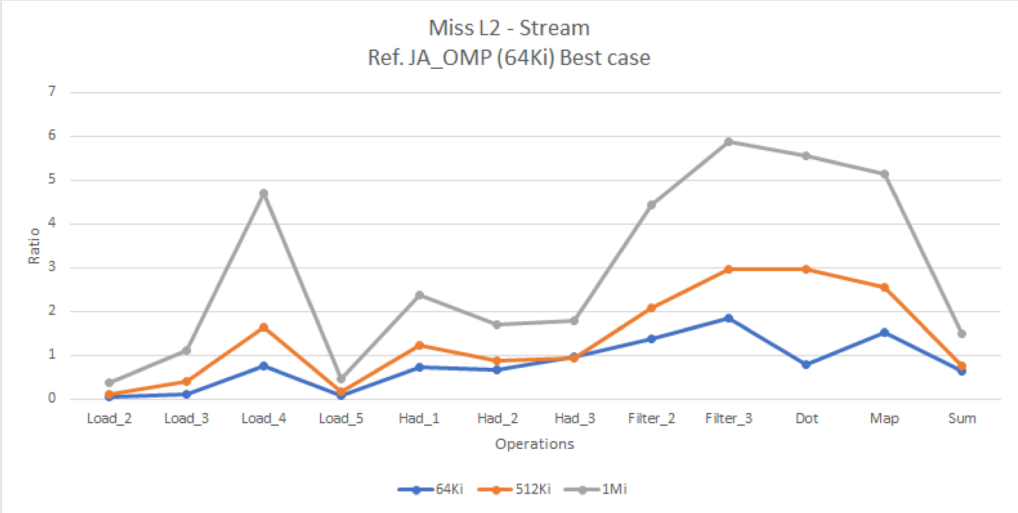
- It is observed that the version with blocks of 64Ki elements presents better results than the version with blocks of 1Mi, by the comparison of the cache miss and the execution time

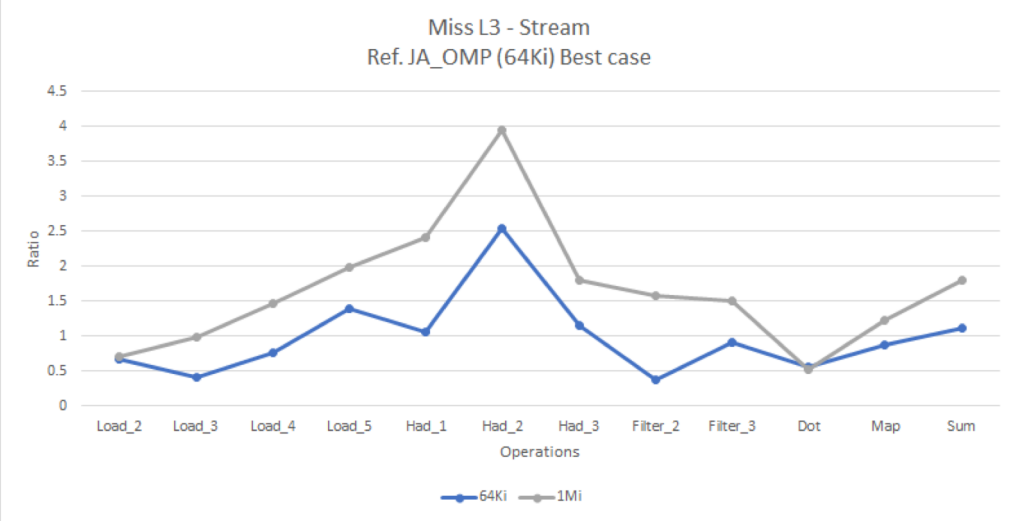
- However its total execution time is superior to version with blocks of 1Mi elements

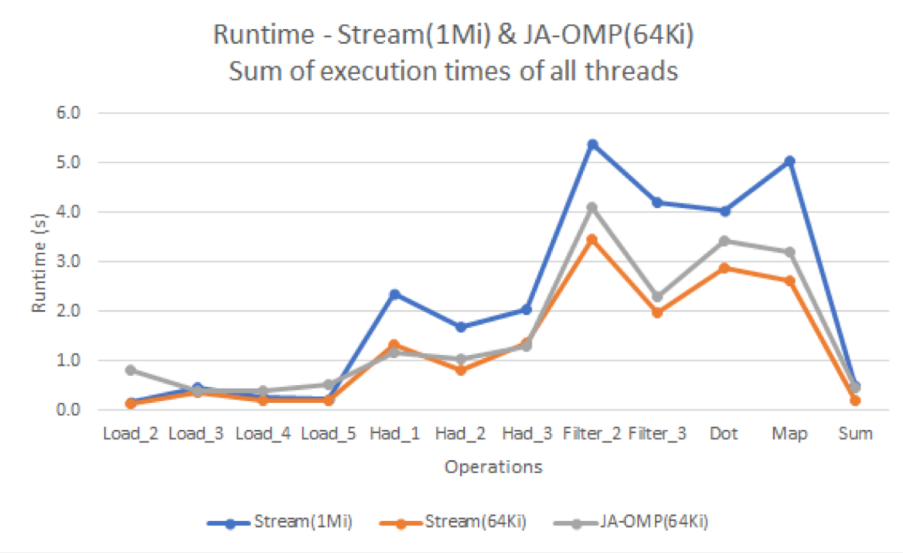


Cache misses - Stream



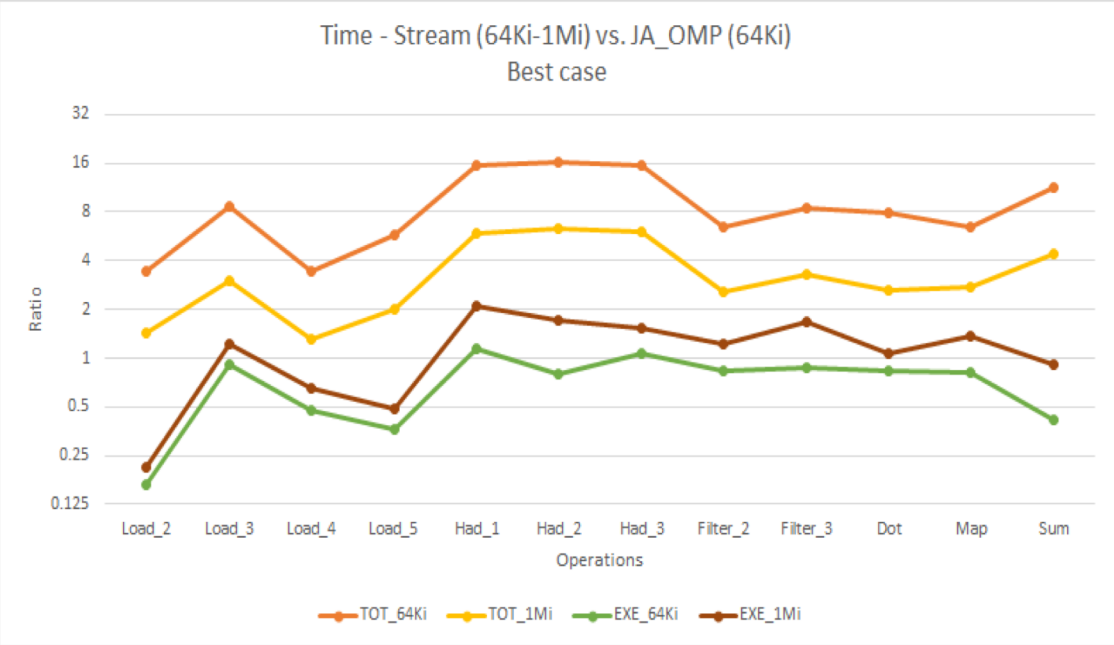






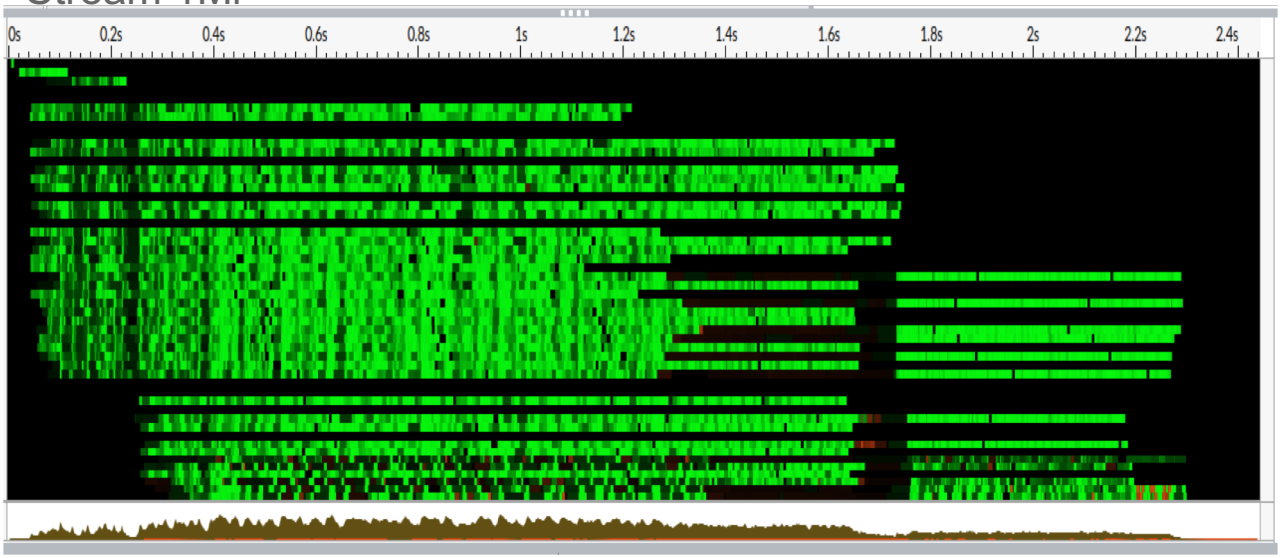
Runtime and waiting time

- The 64Ki block size version has significantly better runtimes
- However, it also has the worst overall time, meaning the waiting time is higher than the other version
  - This effect is worst with the reduction of the Block size (increase of the total number of Block to be processed).
- These facts imply that the overload of the synchronization mechanisms of the stream version limit the performance

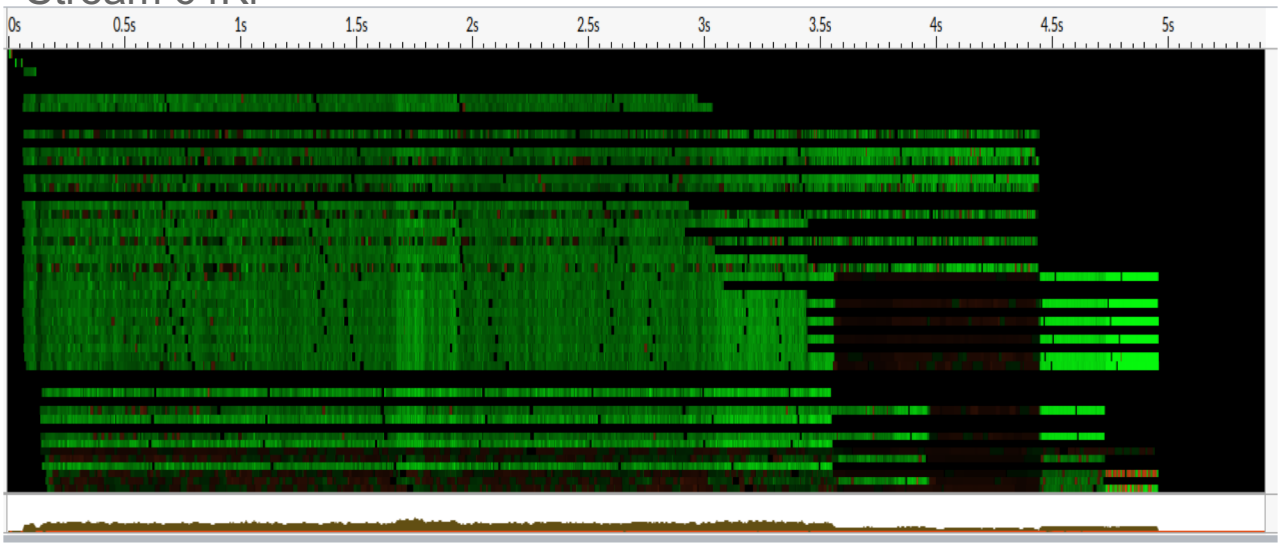


Node 662 (2x 12-cores, HT)
`icc -O2;` TPC-H 32GiB

Stream 1Mi



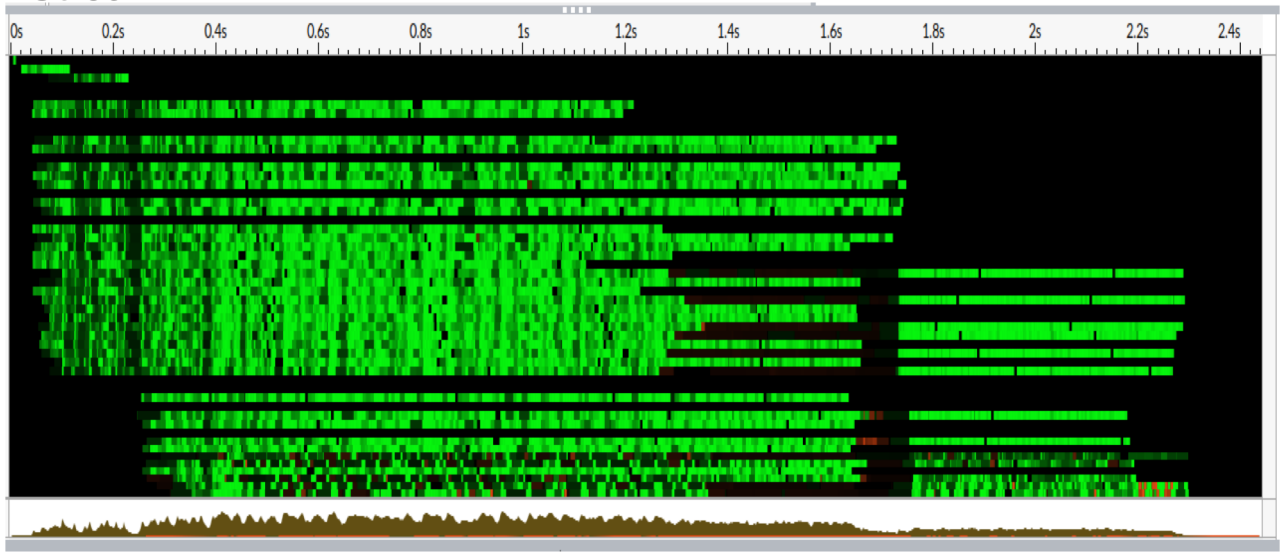
Stream 64Ki



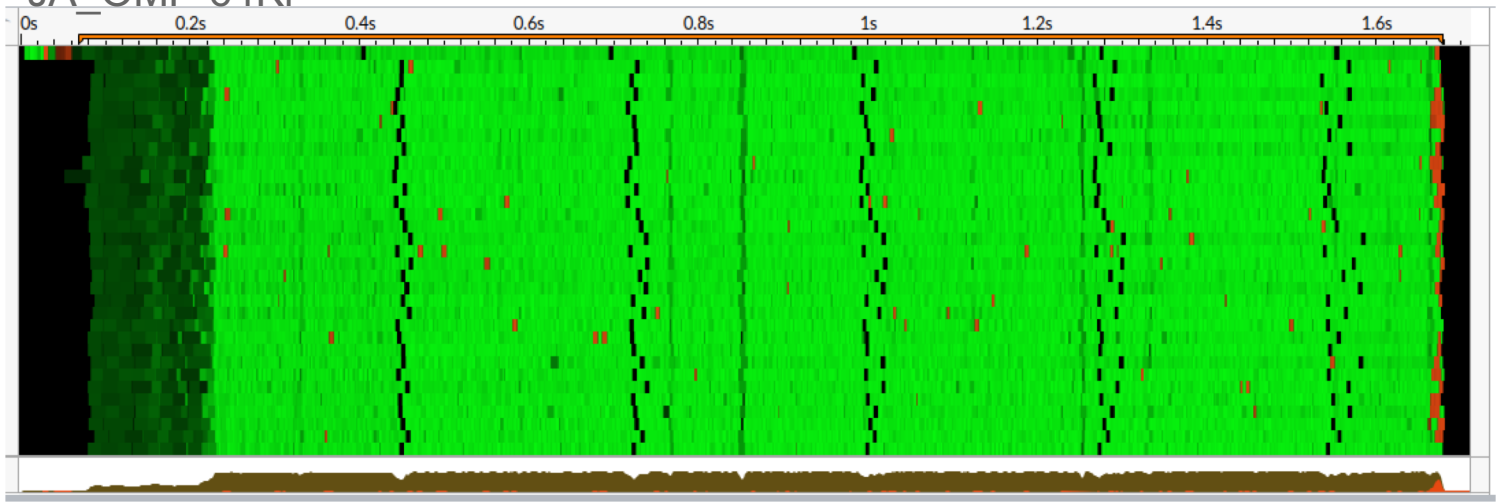
Cache 662	
L1	384KB (12*32KB)
L2	3MB (12*256KB)
L3	20MB

Block size	Size(Bytes)
32Ki	512KiB
64Ki	1MiB
128Ki	2MiB
256Ki	4MiB
512Ki	8MiB
1Mi	16MiB
2Mi	32MiB

Stream 1Mi



JA_OMP 64Ki



PAPI - Prefetch counters

```
| OFFCORE_REQUESTS
|          Offcore requests
|     :ALL_DATA_RD
|          Demand and prefetch read requests sent to uncore
|     :ALL_DATA_READ
|          Demand and prefetch read requests sent to uncore
|     :DEMAND_CODE_RD
|          Offcore code read requests, including cacheable and un-cacheables
|     :DEMAND_DATA_RD
|          Demand Data Read requests sent to uncore
|     :DEMAND_RFO
|          Offcore Demand RFOs, includes regular RFO, Locks, ItoM
```

?

(ALL_DATA != DEMAND + PREFETCH)

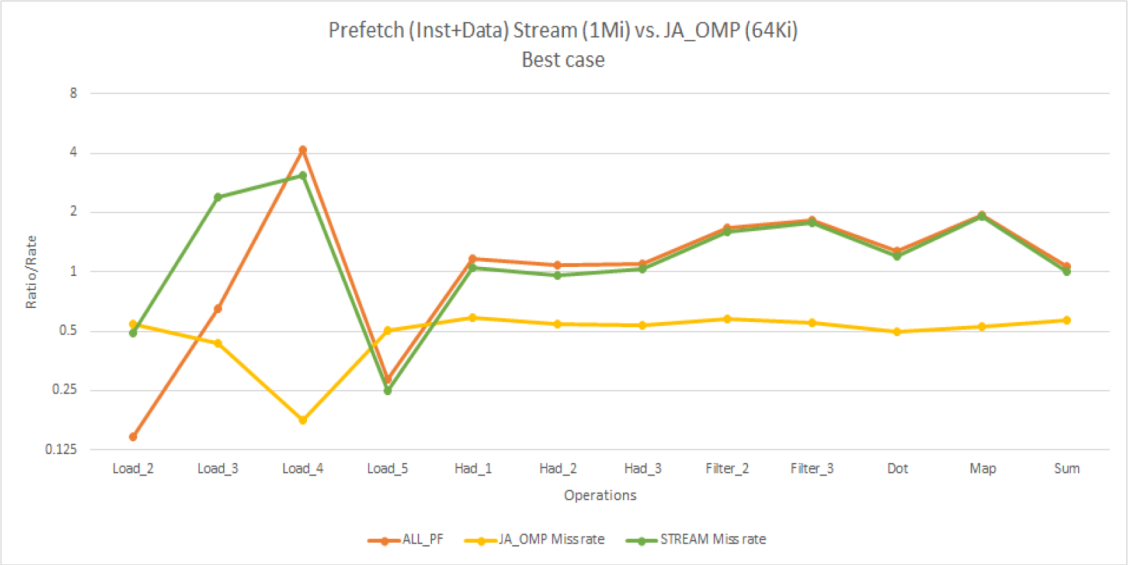
```
| L2_RQSTS
|          L2 requests
|     :ALL_CODE_RD
|          Any code request to L2 cache
|     :CODE_RD_HIT
|          L2 cache hits when fetching instructions
|     :CODE_RD_MISS
|          L2 cache misses when fetching instructions
|     :ALL_DEMAND_DATA_RD
|          Demand  data read requests to L2 cache
|     :DEMAND_DATA_RD_HIT
|          Demand data read requests that hit L2
|     :ALL_PF
|          Any L2 HW prefetch request to L2 cache
|     :PF_HIT
|          Requests from the L2 hardware prefetchers that hit L2 cache
|     :PF_MISS
|          Requests from the L2 hardware prefetchers that miss L2 cache
|     :ALL_RFO
|          Any RFO requests to L2 cache
|     :RFO_HIT
|          Store RFO requests that hit L2 cache
|     :RFO_MISS
|          RFO requests that miss L2 cache
```

```
| OFFCORE_RESPONSE_1
|          Offcore response event (must provide at least one request type and|
|           either any_response or any combination of supplier + snoop)
|     :DMND_DATA_RD
|          Request: number of demand and DCU prefetch data reads of full and |
|          partial cachelines as well as demand data page table entry cacheli|
|          ne reads. Does not count L2 data read prefetches or instruction fe|
|          tches
|     :DMND_RFO
|          Request: number of demand and DCU prefetch reads for ownership (RF|
|          O) requests generated by a write to data cacheline. Does not count|
|           L2 RFO prefetches
|     :DMND_IFETCH
|          Request: number of demand and DCU prefetch instruction cacheline r|
|          eads. Does not count L2 code read prefetches
|     :WB
|          Request: number of writebacks (modified to exclusive) transactions|
|     :PF_DATA_RD
|          Request: number of data cacheline reads generated by L2 prefetcher|
|          s
|     :PF_RFO
|          Request: number of RFO requests generated by L2 prefetchers
|     :PF_IFETCH
|          Request: number of code reads generated by L2 prefetchers
|     :PF_LLC_DATA_RD
|          Request: number of L3 prefetcher requests to L2 for loads
|     :PF_LLC_RFO
|          Request: number of RFO requests generated by L2 prefetcher
|     :PF_LLC_IFETCH
|          Request: number of L2 prefetcher requests to L3 for instruction fe|
|          tches
```

X

Incoherent data, all counters are mostly 0

- The prefetch data corresponds to requests made to the prefetching HW of the L2 cache, for instructions and data
- Because these counters count both types of prefetch (data and instructions) we can not draw conclusions about their influence in the execution of the program



- Analysis of the I / O operations of the disk (Dstat).

Optimisations

- Remove of initializations
- Copy Instructions
- Khatri-Rao/Krao changes
- Reuse of Data Structures (OMP & DSP version)
- Results

Remove of initialisations

The initialization previously used explicitly implies the writing of value 0 on each element of the vector.

```
116  FilteredDecimalMapBlock::FilteredDecimalMapBlock()
117    : Block(), values(BSIZE, 0), rows(BSIZE, 0), cols(BSIZE + 1, 0) {
118    cols[0] = 0;
119  }
```

This initialization of the values is not necessary, as each operation automatically rewrites all block-related parameters.

The resize function, in this case, allows the allocation of the required space without assigning values.

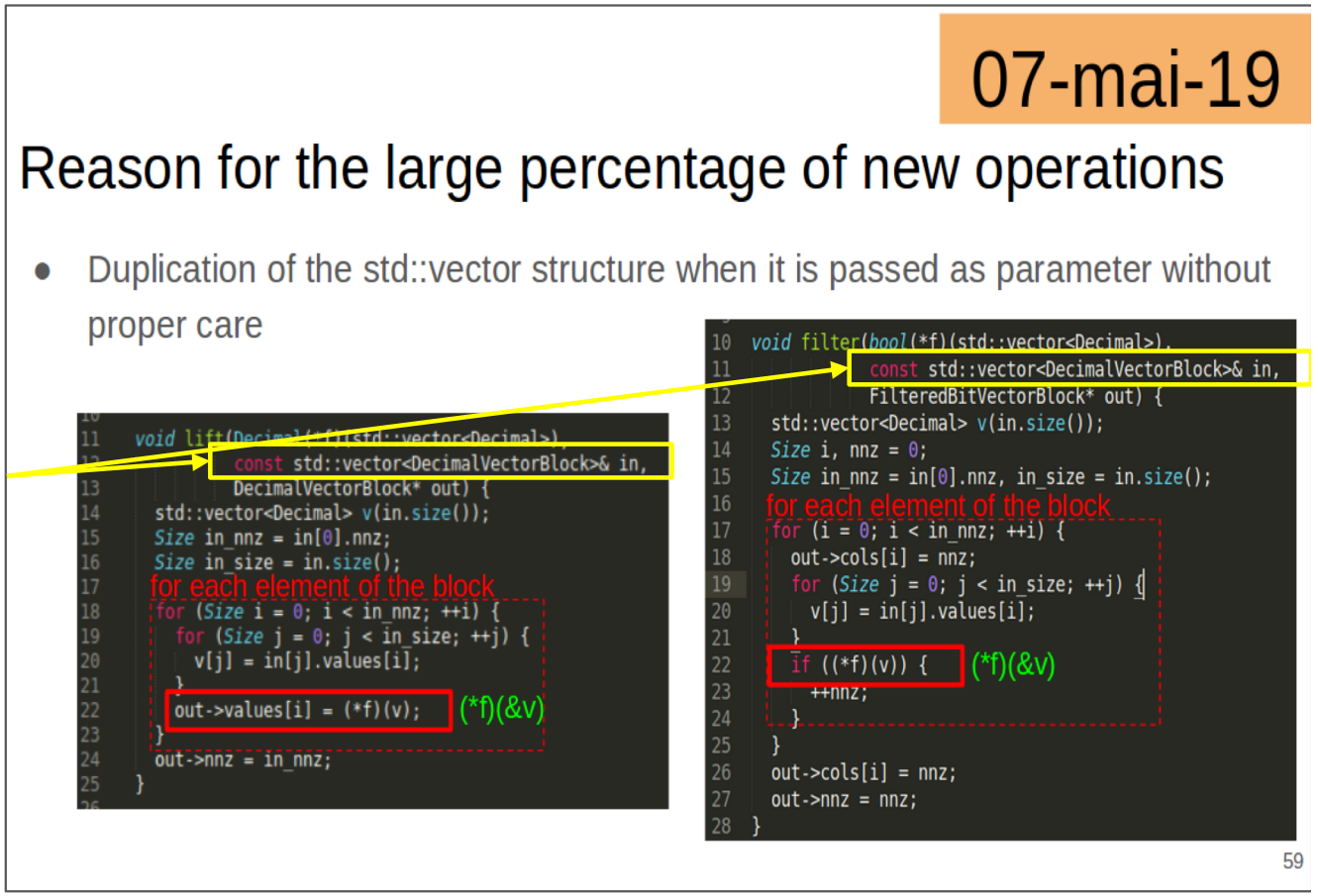
```
140  FilteredDecimalMapBlock::FilteredDecimalMapBlock()
141    : Block()
142  {
143    values.resize(BSIZE);
144    rows.resize(BSIZE);
145    cols.resize(BSIZE + 1);
146    cols[0] = 0;
147  }
```

Copy instructions

As mentioned in slide 59, certain argument passages may entail copy operations which is undesirable.

- Adds overhead
- It has no benefits in these cases

This behavior has also been observed when passing arguments to **Lift** and **Filter** operations.



Copy instructions (Cont…)

All vector passages are replaced by solutions that do not involve copy operations.

- The use of vectors was avoided whenever possible (1,2,3)
- In the remaining cases, solutions that did not involve copy operations were adopted (4) - Explicit use of pointers

```
filter(filter_var_a,
  {
    *(var_lineitem_shipdate->labels[i])
  },
  var_a_pred->blocks[i]);
```
1

```
filter(filter_var_b,
  {
    *(var_lineitem_discount->blocks[i])
  },
  var_b->blocks[i]);
```
2

```
filter(filter_var_d,
  {
    *(var_lineitem_quantity->blocks[i])
  },
  var_d->blocks[i]);
```
3

```
lift(lift_var_f,
  {
    *(var_lineitem_extendedprice->blocks[i]),
    *(var_lineitem_discount->blocks[i])
  },
  var_f->blocks[i]);
```
4

```
filter(filter_var_a,
    var_lineitem_shipdate->labels[i],
    var_a_pred->blocks[i]);
```
1

```
filter(filter_var_b,
    var_lineitem_discount->blocks[i],
    var_b->blocks[i]);
```
2

```
filter(filter_var_d,
        var_lineitem_quantity->blocks[i],
        var_d->blocks[i]);
```
3

```
std::vector<DecimalVectorBlock*> aux;
aux.push_back(var_lineitem_extendedprice->blocks[i]);
aux.push_back(var_lineitem_discount->blocks[i]);
lift(lift_var_f, &aux, var_f->blocks[i]);
```
4

Khatri-Rao/Krao changes

The Krao operation has also undergone some changes in its algorithm.

In its previous version, the number of non zero elements (>NNZ) was not taken into account.

The changes not only prevent access to unwanted elements, but also improve performance (reducing the number of operations).

```
A.nnz < B.nnz
// Iterate the column pointer array until the las element (<A.nnz)
for i = 0 to A.nCols AND A.cols[i] < A.nnz:
   // Store the current number of elements
```

```
krao (A, B, C):
    assert A.nCols = B.nCols
    nnz = 0
    // Pick the matrix with fewer elements
    if A.nnz < B.nnz:
        // Iterate the column pointer array
        for i = 0 to A.nCols:
            // Store the current number of elements
            C.cols[i] = nnz
            // Check if an element is present in both matrices
            if A.cols[i+1] > A.cols[i] and B.cols[i+1] > B.cols[i]:
                // Get the current indexes of the values and rows arrays
                Apos = A.cols[i]
                Bpos = B.cols[i]
```

From João Afonso(2018)

NNZ is the number of non zero elements in the block.
nCols is the total number of columns or block size.

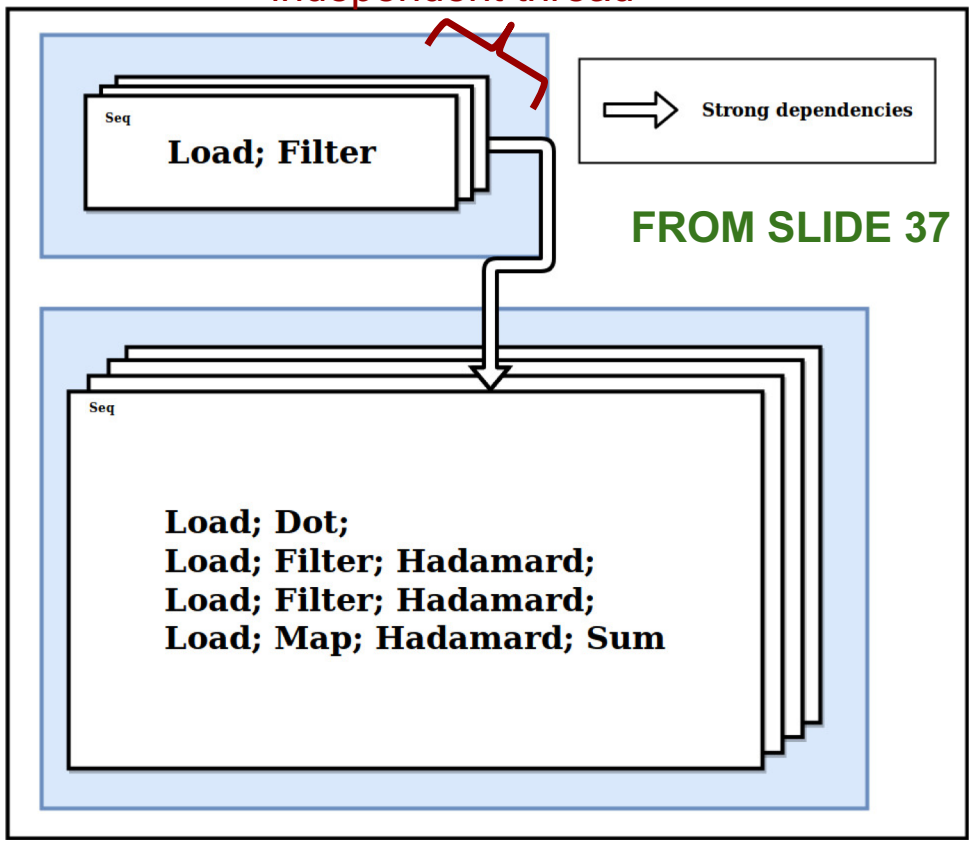
Reuse - OMP version

Each thread can allocate its own data set that is reused in each iteration of the pipeline.

It does not require any alteration/initialisation on the block data, all data is automatically rewritten by the respective operations.

Thus each thread only requires one allocation and one memory release, rather than repeating the process for each new iteration.

Multiple iterations of the pipeline, each iteration is performed by an independent thread



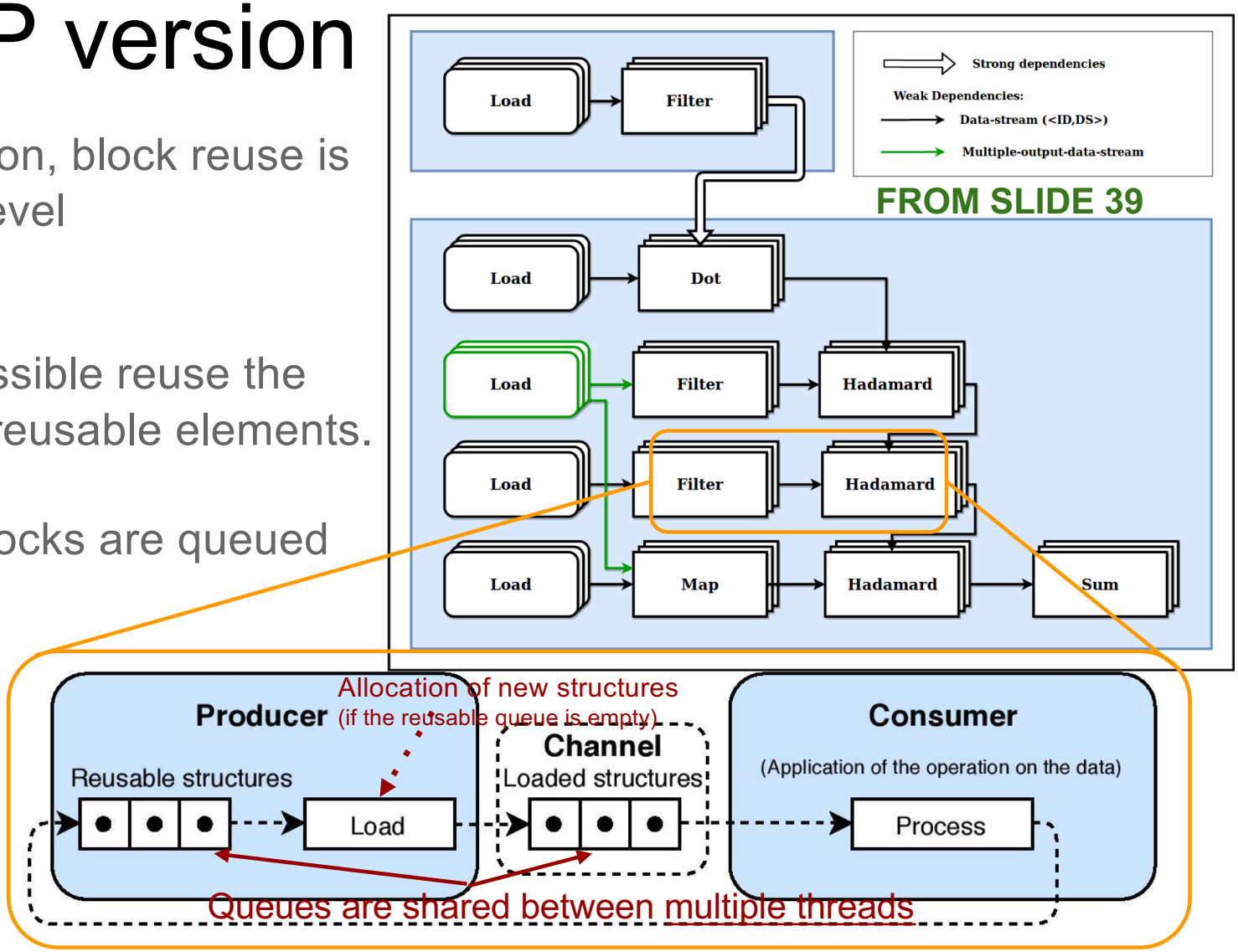
Reuse - DSP version

In the Stream/DSP version, block reuse is done at the operations level (Producer/Consumer).

Producers whenever possible reuse the blocks in their queue of reusable elements.

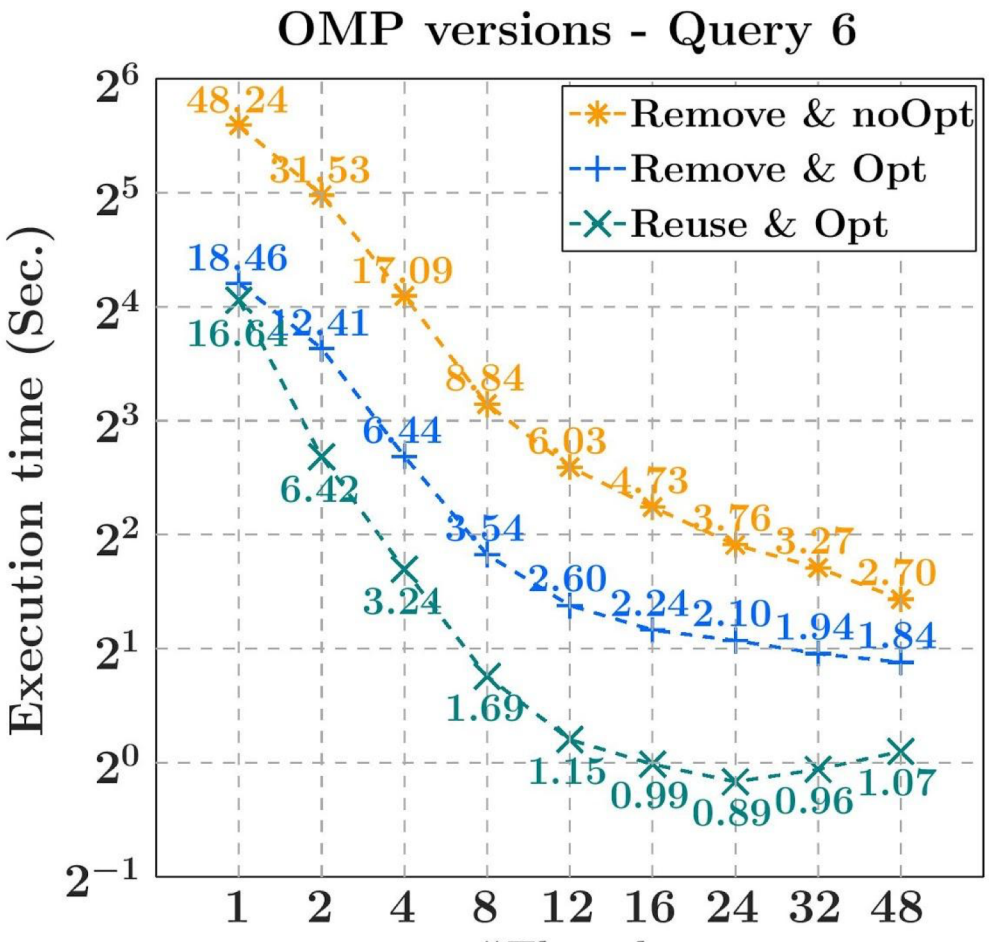
Previously consumed Blocks are queued for reusable elements.

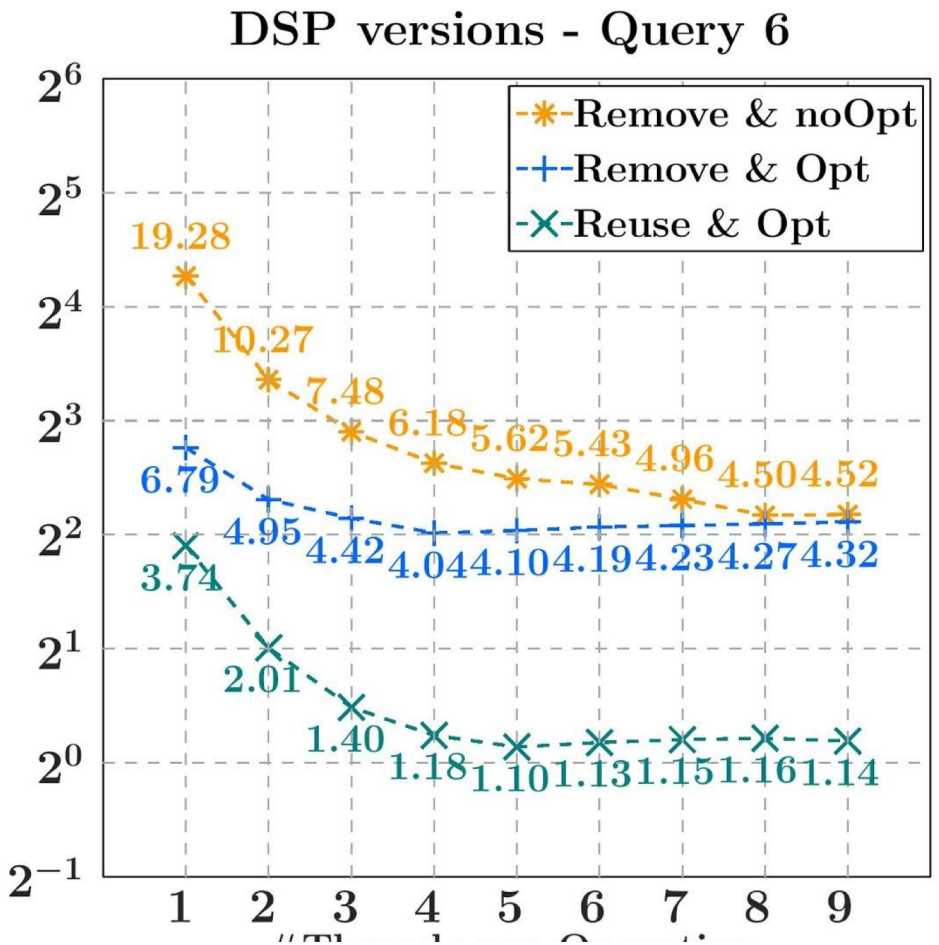
Queue access is controlled by mutual exclusion operations.



Results - Texec

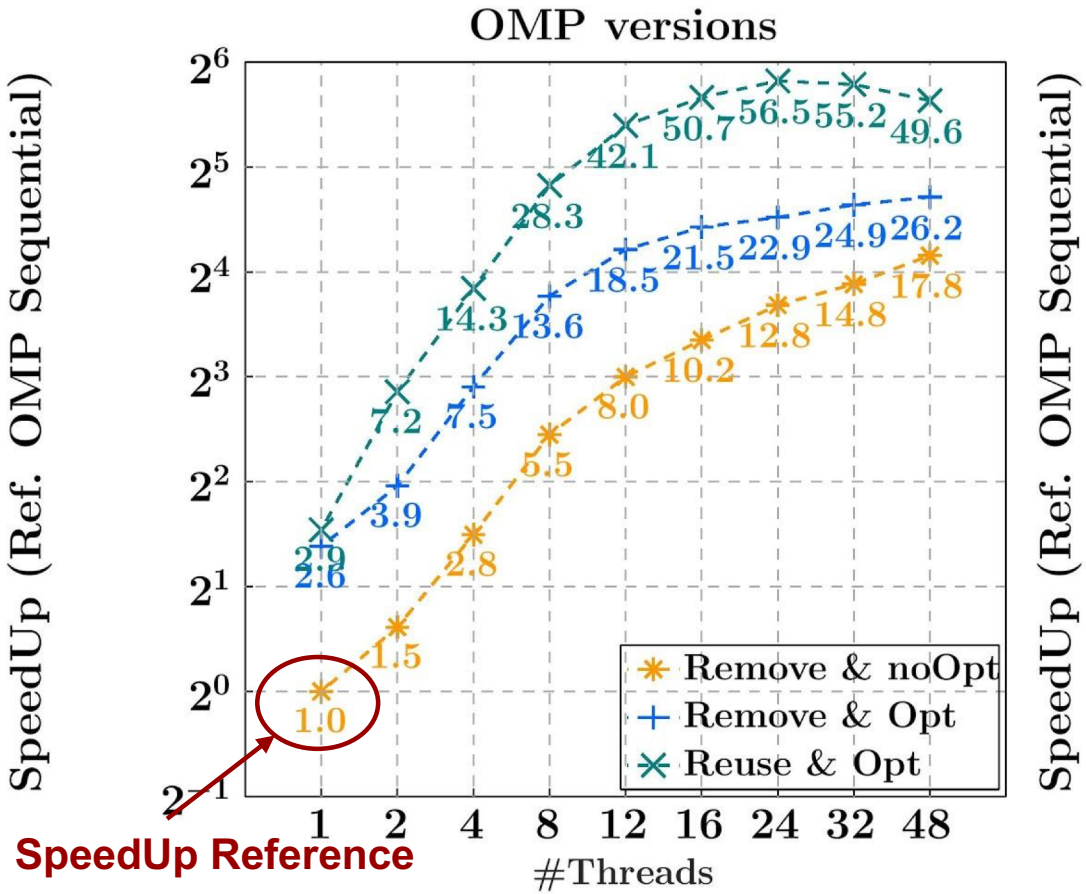
Query 6 - Node 662 - ICC -O3 - TCP-H 32GB

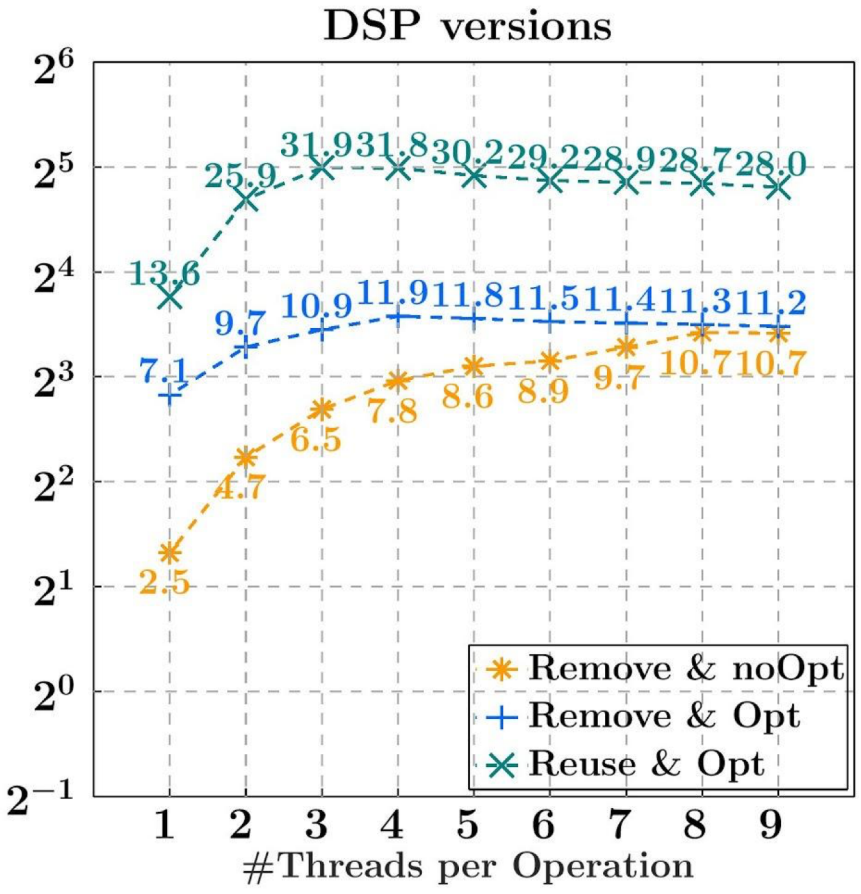




Results - SpeedUP

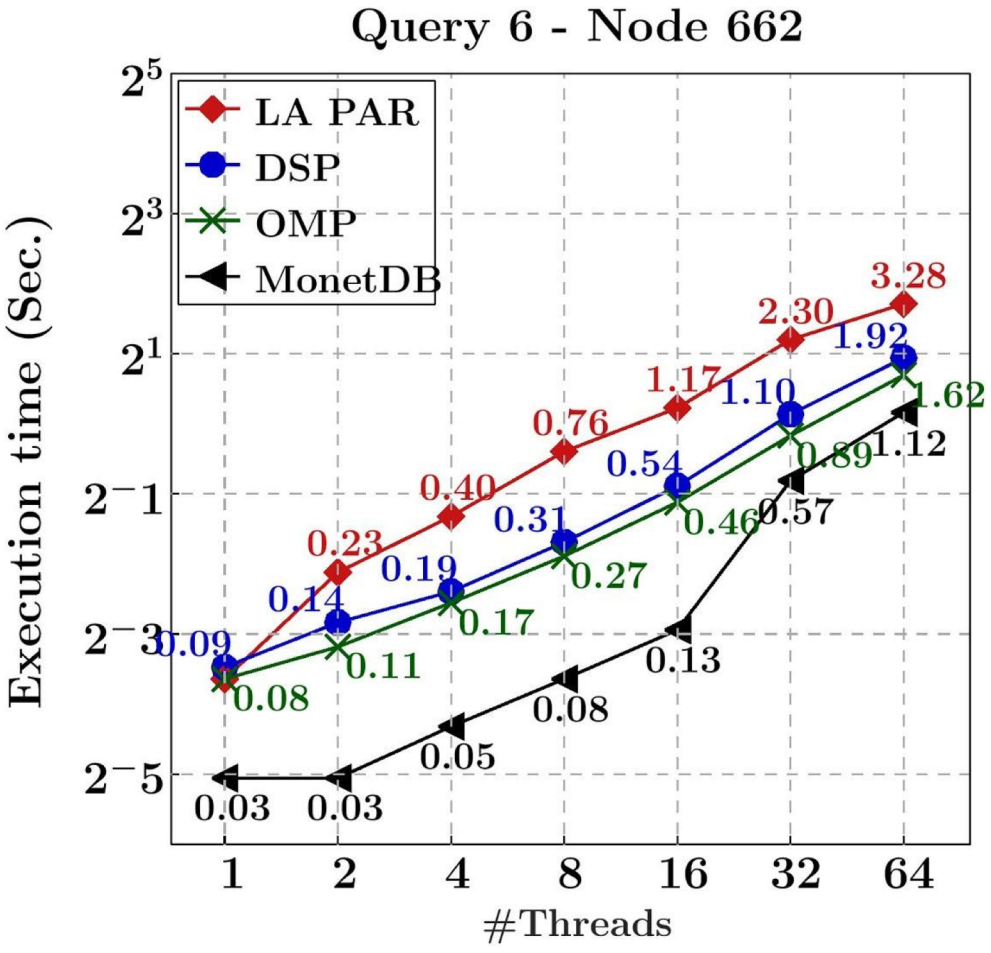
Query 6 - Node 662 - ICC -O3 - TCP-H 32GB

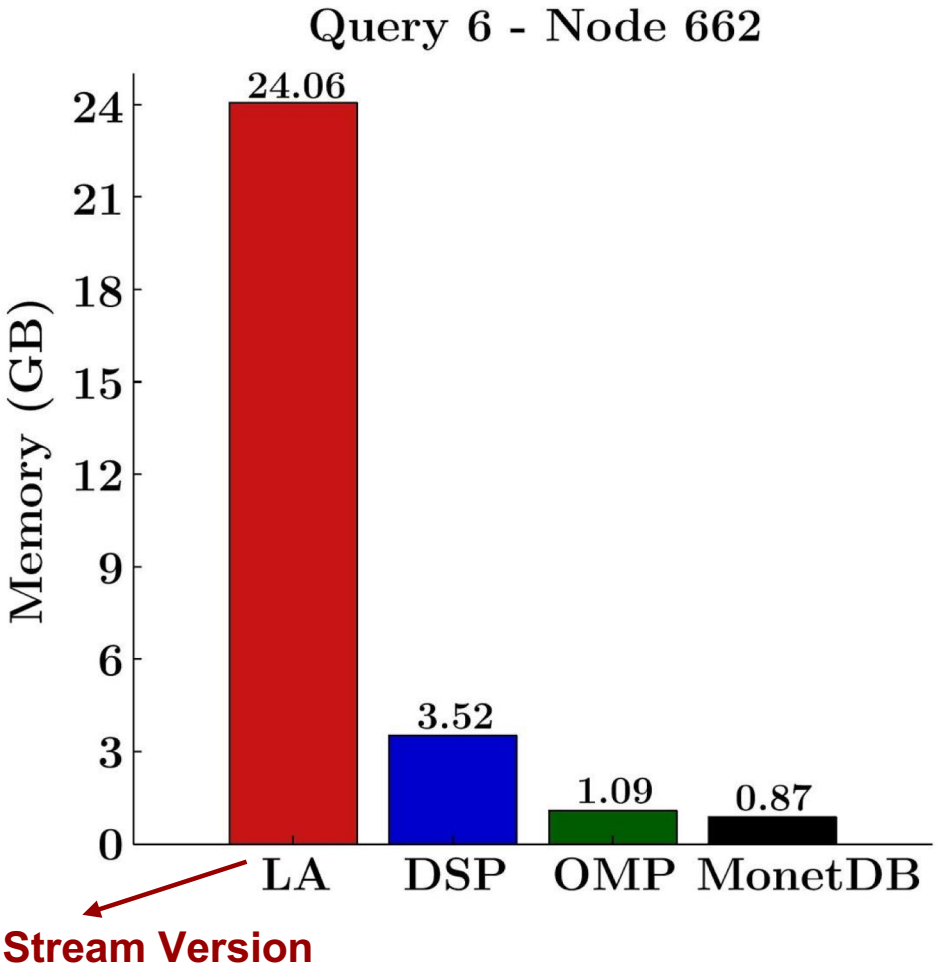




Results - SpeedUP

Query 6 - Node 662 - ICC -O3 - TCP-H 32GB



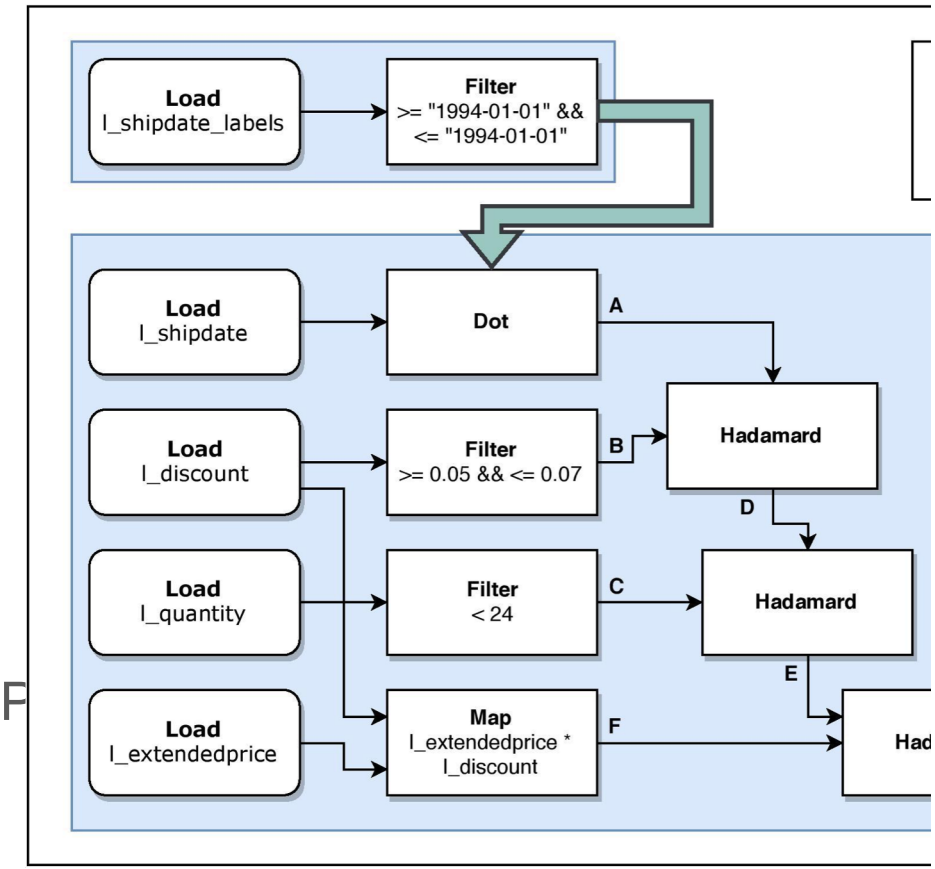


HEP-Frame implementation

- LA to HEP-Frame (ROOT libraries)
- HEP-Frame (event access DP & DS)
- New solution (Reuse & memory management)
- Results

LA to HEP-frame

- Strong dependencies delimit pipelines
- Each LA operation constitutes a task
- Number of block elements can be used as filter condition
- LA load operations are not supported in HEP Frame (ROOT library)



The LA operations kernel is ready to be integrated with HEP-Frame. However, it is necessary to develop a new DS/(Data load) library for the HEP-frame.

HEP-Frame - DP

Processing threads (DP) access events that meet the following condition:

Events that are within 100 units of the current event counter are assumed to be events properly loaded from memory.

Global variables

```
while (event_counter < _number_events) {
    if (this_thread_id < num_active_process_threads) {
        this_event_counter = event_counter;

        // checks if there are events to process in the file
        if (((this_event_counter+100) >= current_number_events) && !finished_reading) {
            thread_blocked = true;

            boost::this_thread::sleep_for( boost::chrono::microseconds(30) );//barriers
        } else {
```

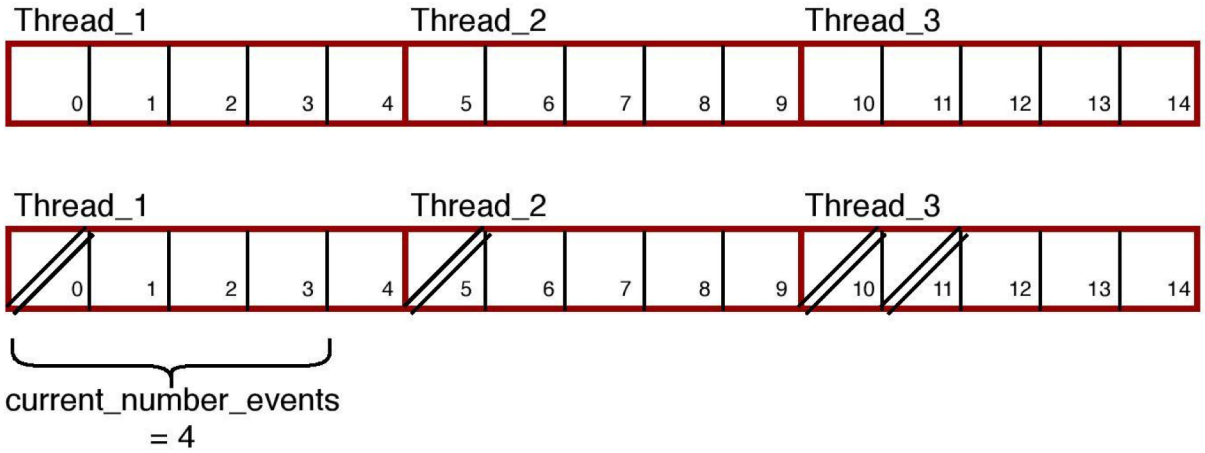
Value updated/changed by read threads

However, this measure is not secure enough to ensure that all events are properly loaded before being accessed by a processing thread.

Division of Events (DS)

If we assume that each thread is responsible for 5 events each and the following event array:

- The variable ensures that there are current_number_events events properly loaded
- However, does not guarantee that these events are in consecutive positions




```
if (thread_blocked) {
    update_event_count.lock();
    current_number_events += since_last;
    update_event_count.unlock();

    since_last = 0;

    thread_blocked = false;
    // boost::unique_lock<boost::mutex> _l
    // wait_for_event_load2.notify_all();
}
```

Solution implemented

As the distance between DP threads and DS threads is not a sufficiently safe measure, new means of verification have been added.

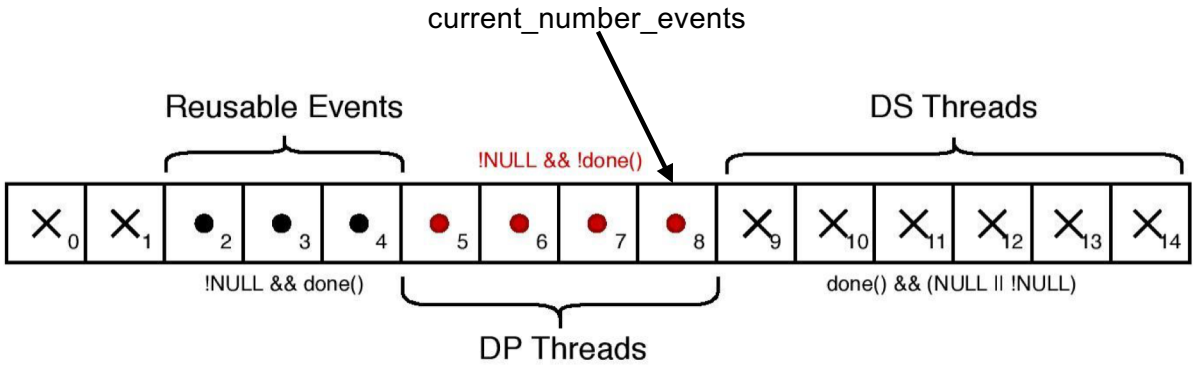
- By adding a state variable to each event, it is possible determine if the event is already loaded
- Due to the implementation of data structure reuse, the verification of the existence of the event is also required

```
while (this_event_counter < _number_events && event_counter < _number_events)
{
    if (this_thread_id < num_active_process_threads)
    {
        // checks if there are events to process in the file
        // !((safe() || finish) && (!NULL || !done())) => ((!safe() && !finish) || (NULL || done()))
        if ((((this_event_counter+1) >= current_number_events) && !finished_reading)
            || ((events[this_event_counter]==NULL) || events[this_event_counter]->done()))
        {

            thread_blocked = true;
            boost::this_thread::sleep_for( boost::chrono::microseconds(250) );//barriers
        } else
        {
```

Boundaries between DP&DS

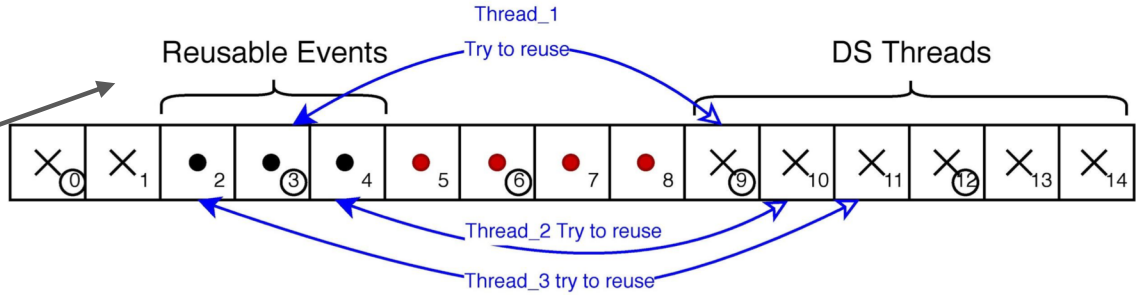
The accessible positions for each phase (DS and DP) are well delimited



- In this way the control over access is greater
- The overhead is reduced as there are only two more conditions (! NULL && !Done())
- It also allows the identification of the elements that are available for reuse

Event Reuse

With reuse mechanisms, DS threads reuse already processed events whenever possible.



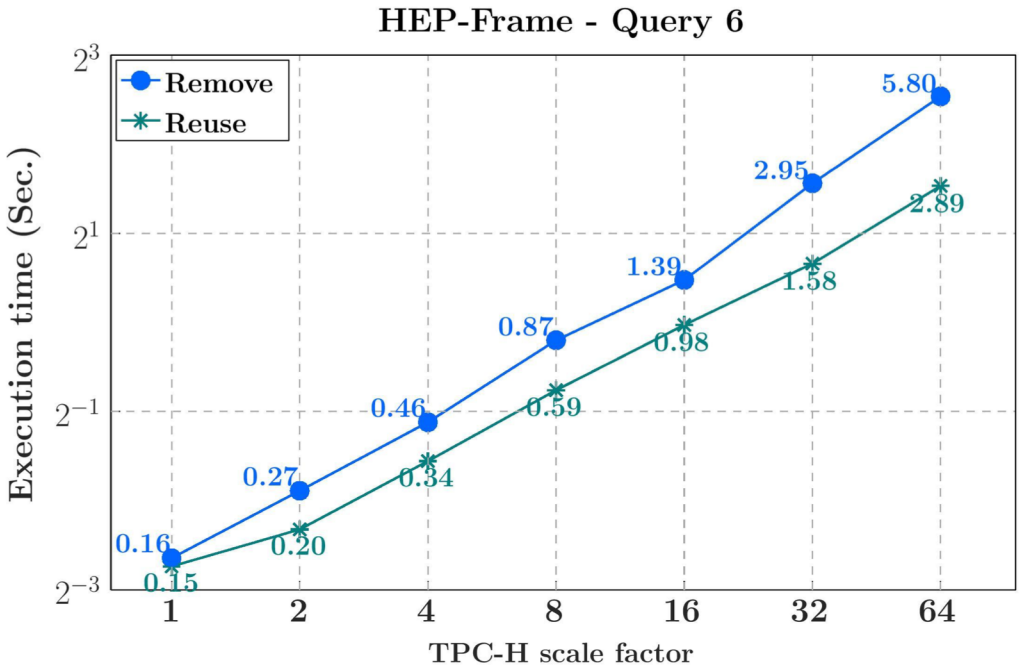
- The process requires verification of the event state and its change of position in the structure
- As events are split between DS threads there is no concurrency in their access, which contributes favorably to minimal overhead (Unlike the DSP version)
- Additionally, this structure allows the limitation of the number of allocated events (memory management)

Results

Query 6 - Node 662 - ICC -O3 - TCP-H 32GB
Block Size 64Ki - Max Mem. used 2GB

Preliminary tests show that thread concurrency over pipeline tasks has significant overhead, so the number of threads per iteration has been limited. The best results are obtained with 2 threads per iteration.

Event reuse brings significant performance gains.

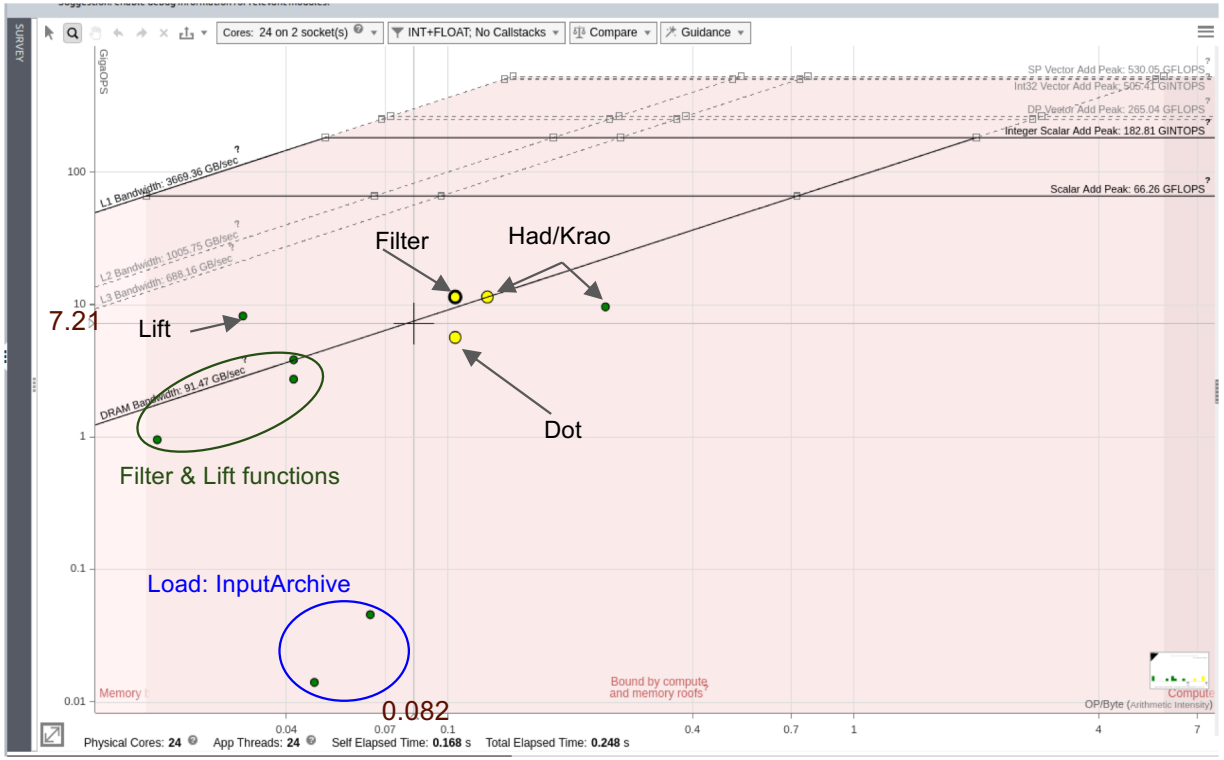


However, the performance presented is still behind the DSP and OMP versions, further analysis and testing is needed to identify possible limitations.

RoofLine

Query 6 - Node 662 - ICC -O3 - TCP-H 32GB
Block Size 64Ki

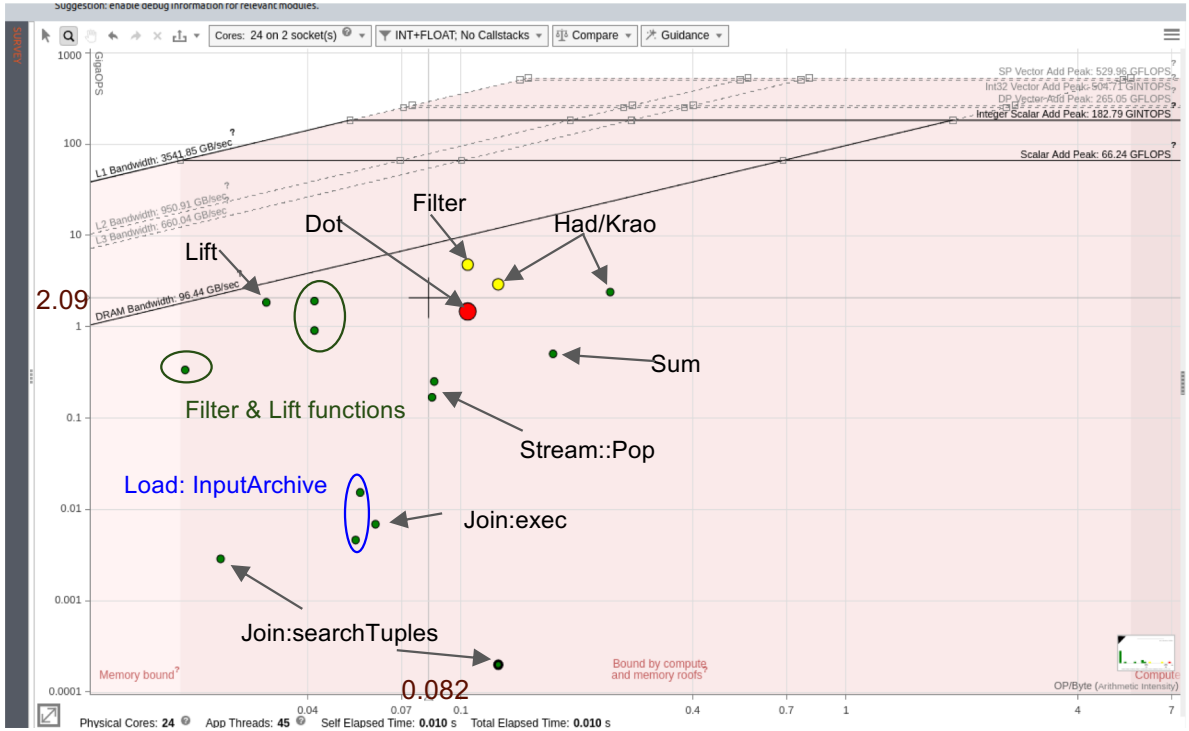
- All LA operations have a relatively low operating intensity. (< 0.4 Ops/byte)
- Load operations, in addition to low operating intensity, also perform poorly. This is expected considering that these operations load data directly from memory, while the remaining operations can take better advantage of the cache levels.



DSP version

- It can be verified that the management operations of the communication channels have a significant load in the execution of query 6.
- Join-related operations and access to the queues are underperforming, mainly due to the management of mutual exclusions.

Query 6 - Node 662 - ICC -O3 - TCP-H 32GB
Block Size 64Ki



DSP vs OMP version

Query 6 - Node 662 - ICC -O3 - TCP-H 32GB
Block Size 64Ki

- Note that the OMP version has 70% performance gains for almost all algebraic operations.
- In load operations the performance remains similar in both versions.