S5151 - Voting And Shuffling For Fewer Atomic Operations

Elmar Westphal, Forschungszentrum Jülich GmbH

Contents

- On atomic operations and speed problems
- A possible remedy
- About intra-warp communication
- Description of the algorithm
- Benchmarks
- Sample code (appendix)

On Atomic Operations And Speed Problems

- With every new GPU-generation, atomic operations became faster, but they are still comparatively slow and not natively available for all data types
- Atomic operations not natively available (i.e. double precision atomicAdd) can often be implemented using an atomicCAS loop
  - May lead to branch divergence for address collisions within the same warp, stalling all threads in the warp
    - This leads to **severe performance penalties** for algorithms that perform atomic operations on a small number of data items in a warp

A Possible Remedy

- Perform the operation on colliding addresses within the warp first
- Update target data using one atomic operation per address per warp:
  - Lowers atomic operation count in general
  - Avoids branch divergence in CAS loops
- Can be implemented using reduction sub-trees in the warps, in parallel
- Values can be exchanged using intra-warp communication

Intra-warp Communication

- Warp vote functions:
  - `__any(predicate)` returns non-zero if any of the predicates for the threads in the warp returns non-zero
  - `__all(predicate)` returns non-zero if all of the predicates for the threads in the warp returns non-zero
  - `__ballot(predicate)` returns a bit-mask with the respective bits of threads set where predicate returns non-zero

Intra-Warp Communication/ Bit Operations

- Data exchange:
  - `__shfl(value, thread)` returns value from the requested thread (but only if this thread also performed a `__shfl()`-operation)
    - available in different flavors for more specialised tasks (not needed here)
- Useful bit operations:
  - `__ffs(value)` returns the index of first (least significant) set bit
  - `__popc(value)` returns the number of set bits

The Algorithm

- Here "key" shall be defined as a value used to determine the target address of an atomic operation (or the address itself)
- Two stage algorithm:
  - Stage 1: find out which elements share the same key within each warp
  - Stage 2: pre-process these using subtrees within warps, in parallel
- First step can be expensive, but pays off if result can be reused
- Subtrees are traversed using bit-patterns obtained in stage 1

Stage 1 - Finding Peers

- Set all lanes unassigned
- While we have unassigned lanes
  - Find all lanes with the same key as in the least unassigned lane
  - Remove found lanes from unassigned lanes
  - If this lane is included, store found lanes as peers and exit loop
- Loop always iterates as many times as we have different keys in warp

Stage 1 - Example

	Peers
0	
1	
2	
3	
4	
5	
6	
7	

Keys: 1 2 3

Iteration 1:

- all threads are still active
- lowest active thread (0) has key 2
- __ballot(key==2) returns 10010001

Stage 1 - Example

	Peers
0	10010001
1	
2	
3	
4	10010001
5	
6	
7	10010001

Keys: 1 2 3

Iteration 1:
- all threads are still active
- lowest active thread (0) has key 2
- __ballot(key==2) returns 10010001
- keep this for all threads with key==2

Stage 1 - Example

	Peers
0	10010001
1	
2	
3	
4	10010001
5	
6	
7	10010001

Keys: 1 2 3

Iteration 1:

- lowest active thread (0) has key 2
- __ballot(key==2) returns 10010001
- keep this for all threads with key==2
- these threads are now done

Stage 1 - Example

	Peers
0	10010001
1	
2	
3	
4	10010001
5	
6	
7	10010001

Keys: 1 2 3

Iteration 2:

- some threads are still active
- lowest active thread (1) has key 3
- __ballot(key==3) returns 00100110

Stage 1 - Example

	Peers
0	10010001
1	00100110
2	00100110
3	
4	10010001
5	00100110
6	
7	10010001

Keys: 1 2 3

Iteration 2:

- some threads are still active
- lowest active thread (0) has key 3
- __ballot(key==3) returns 00100110
- keep peers and deactivate threads

Stage 1 - Example

	Peers
0	10010001
1	00100110
2	00100110
3	
4	10010001
5	00100110
6	
7	10010001

Keys: 1 2 3

Iteration 3:

- some threads are still active
- lowest active thread (3) has key 1
- __ballot(key==1) returns 01001000

Stage 1 - Example

	Peers
0	10010001
1	00100110
2	00100110
3	01001000
4	10010001
5	00100110
6	01001000
7	10010001

Keys: 1 2 3

Iteration 3:

- some threads are still active
- lowest active thread (0) has key 3
- __ballot(key==1) returns 01001000
- keep peers and deactivate threads
- no active threads left, we are done

ok, but how do I…

- …find lanes sharing a certain key:
  - `peers=__ballot(my_key==other_key)`
- …find the other key:
  - `other_key=__shfl(my_key,first_unassigned_thread)`
- …find the first unassigned thread:
  - `first_unassigned_thread=__ffs(unassigned_threads)-1`
- …update the bit mask of unassigned threads
  - `unassigned_threads^=peers`

Similarities To Other Algorithms

- Some of these operations can be found in other/similar contexts, e.g.:
  - Warp aggregated atomic filtering as described in

    http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/

Stage 2 - Pre-process Using Sub-trees

Using the bit-pattern generated in stage 1:

- Find lane's relative position among its peers
- Drop all peer entries with same or lower lane ID
- Repeat, until this lane's value was used:
  - Add next peer's value* with higher lane ID, if it exists
  - Delete all lanes that were just added from all peer bit-patterns

* "wrong" order if used in larger scopes, but no problem if staying in warp and easier to implement here

Stage 2 - Example

	Peer bitmask	Idx by peer	Idx by peer (binary)	Initial value
0	xx54x3xx2xx1xxx0	0	000	9
1	x4xxxxx3xx2xx10x	0	000	8
2	x4xxxxx3xx2xx10x	1	001	2
3	4xxx3x2xx1xx0xxx	0	000	6
4	xx54x3xx2xx1xxx0	1	001	2
5	x4xxxxx3xx2xx10x	2	010	7
6	4xxx3x2xx1xx0xxx	1	001	1
7	xx54x3xx2xx1xxx0	2	010	4
8	x4xxxxx3xx2xx10x	3	011	7
9	4xxx3x2xx1xx0xxx	2	010	6
10	xx54x3xx2xx1xxx0	3	011	1
11	4xxx3x2xx1xx0xxx	3	011	8
12	xx54x3xx2xx1xxx0	4	100	7
13	xx54x3xx2xx1xxx0	5	101	8
14	x4xxxxx3xx2xx10x	4	100	4
15	4xxx3x2xx1xx0xxx	4	100	7

Stage 2 - Example

Clear out the peers we don't need to add

Add the next peer to our left (if any)

	Peer bitmask	Idx by peer	Idx by peer (binary)	Initial value	Value after iteration 1
0	xx54x3xx2xx1xxxx	0	000	9	11
1	x4xxxxx3xx2xx1xx	0	000	8	10
2	x4xxxxx3xx2xxxxx	1	001	2	-
3	4xxx3x2xx1xxxxxx	0	000	6	7
4	xx54x3xx2xxxxxxx	1	001	2	-
5	x4xxxxx3xxxxxxxx	2	010	7	14
6	4xxx3x2xxxxxxxxx	1	001	1	-
7	xx54x3xxxxxxxxxx	2	010	4	5
8	x4xxxxxxxxxxxxxx	3	011	7	-
9	4xxx3xxxxxxxxxxx	2	010	6	14
10	xx54xxxxxxxxxxxx	3	011	1	-
11	4xxxxxxxxxxxxxxx	3	011	8	-
12	xx5xxxxxxxxxxxxx	4	100	7	15
13	xxxxxxxxxxxxxxxx	5	101	8	-
14	xxxxxxxxxxxxxxxx	4	100	4	4
15	xxxxxxxxxxxxxxxx	4	100	7	7

Stage 2 - Example

Clear out the peers we don't need to add (anymore)

Add the next peer to our left (if any)

	Peer bitmask	Idx by peer	Idx by peer (binary)	Initial value	Value after iteration 1	Value after iteration 2
0	xxx4xxxx2xxxxxxx	0	000	9	11	16
1	x4xxxxxxxx2xxxxx	0	000	8	10	24
2	x4xxxxxxxx2xxxxx	1	001	2	-	-
3	4xxxxx2xxxxxxxxx	0	000	6	7	21
4	xxx4xxxx2xxxxxxx	1	001	2	-	-
5	x4xxxxxxxxxxxxxx	2	010	7	14	-
6	4xxxxx2xxxxxxxxx	1	001	1	-	-
7	xxx4xxxxxxxxxxxx	2	010	4	5	-
8	x4xxxxxxxxxxxxxx	3	011	7	-	-
9	4xxxxxxxxxxxxxxx	2	010	6	14	-
10	xxx4xxxxxxxxxxxx	3	011	1	-	-
11	4xxxxxxxxxxxxxxx	3	011	8	-	-
12	xxxxxxxxxxxxxxxx	4	100	7	15	15
13	xxxxxxxxxxxxxxxx	5	101	8	-	-
14	xxxxxxxxxxxxxxxx	4	100	4	4	4
15	xxxxxxxxxxxxxxxx	4	100	7	7	7

Stage 2 - Example

Clear out the peers we don't need to add (anymore)

Add the next peer to our left (if any)

	Peer bitmask	Idx by peer	Idx by peer (binary)	Initial value	Value after iteration 1	Value after iteration 2	Value after iteration 3
0	xxx4xxxxxxxxxxxx	0	000	9	11	16	31
1	x4xxxxxxxxxxxxxx	0	000	8	10	24	28
2	x4xxxxxxxxxxxxxx	1	001	2	-	-	-
3	4xxxxxxxxxxxxxxx	0	000	6	7	21	28
4	xxx4xxxxxxxxxxxx	1	001	2	-	-	-
5	x4xxxxxxxxxxxxxx	2	010	7	14	-	-
6	4xxxxxxxxxxxxxxx	1	001	1	-	-	-
7	xxx4xxxxxxxxxxxx	2	010	4	5	-	-
8	x4xxxxxxxxxxxxxx	3	011	7	-	-	-
9	4xxxxxxxxxxxxxxx	2	010	6	14	-	-
10	xxx4xxxxxxxxxxxx	3	011	1	-	-	-
11	4xxxxxxxxxxxxxxx	3	011	8	-	-	-
12	xxxxxxxxxxxxxxxx	4	100	7	15	15	-
13	xxxxxxxxxxxxxxxx	5	101	8	-	-	-
14	xxxxxxxxxxxxxxxx	4	100	4	4	4	-
15	xxxxxxxxxxxxxxxx	4	100	7	7	7	-

Stage 2 - Example

Clear out the peers we don't need to add (anymore)

Nothing more to add for our result threads. We are done!

	Peer bitmask	Idx by peer	Idx by peer (binary)	Initial value	Value after iteration 1	Value after iteration 2	Value after iteration 3
0	xxxxxxxxxxxxxxxx	0	000	9	11	16	31
1	xxxxxxxxxxxxxxxx	0	000	8	10	24	28
2	xxxxxxxxxxxxxxxx	1	001	2	-	-	-
3	xxxxxxxxxxxxxxxx	0	000	6	7	21	28
4	xxxxxxxxxxxxxxxx	1	001	2	-	-	-
5	xxxxxxxxxxxxxxxx	2	010	7	14	-	-
6	xxxxxxxxxxxxxxxx	1	001	1	-	-	-
7	xxxxxxxxxxxxxxxx	2	010	4	5	-	-
8	xxxxxxxxxxxxxxxx	3	011	7	-	-	-
9	xxxxxxxxxxxxxxxx	2	010	6	14	-	-
10	xxxxxxxxxxxxxxxx	3	011	1	-	-	-
11	xxxxxxxxxxxxxxxx	3	011	8	-	-	-
12	xxxxxxxxxxxxxxxx	4	100	7	15	15	-
13	xxxxxxxxxxxxxxxx	5	101	8	-	-	-
14	xxxxxxxxxxxxxxxx	4	100	4	4	4	-
15	xxxxxxxxxxxxxxxx	4	100	7	7	7	-

ok, but again, how do I…

- …find a lane's relative position:
  - `relative_position=__popc(peers<<(32-lane))`
- …delete all bits up to this lane:
  - `peers&=(0xfffffe<<lane)`
- …find the next peer's index:
  - `next_peer=__ffs(peers)-1`

ok, but again, how do I…

- …retrieve the next peer value to add:
  - `t=__shfl(value,next_peer)` (important: add only if next_peer>=0!)
- …find out if this thread is done:
  - `done=relative_position&(1<<iteration)` (1)
- …remove the done threads from the peer bit-pattern:
  - `peers&=__ballot(!done)` (2)
- …find out when the loop is done
  - `while(__any(peers)) { … }`

(1)(2) these operations deactivate every second thread in each iteration.
(1) instead of counting and shifting, we may also “count by shifting”:

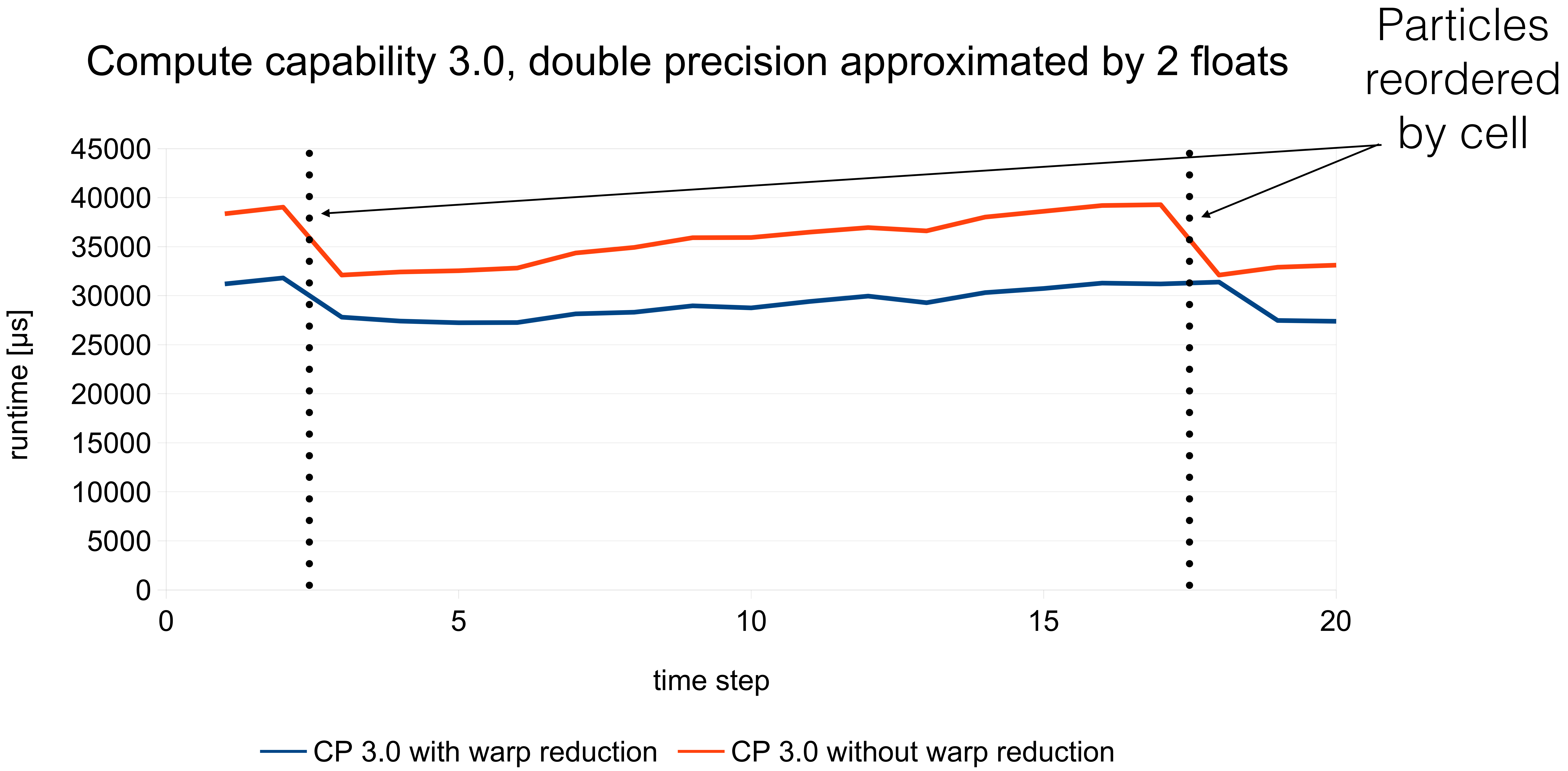
```
done=rel_pos&iteration;
iteration<<=1;
```

Benchmarks

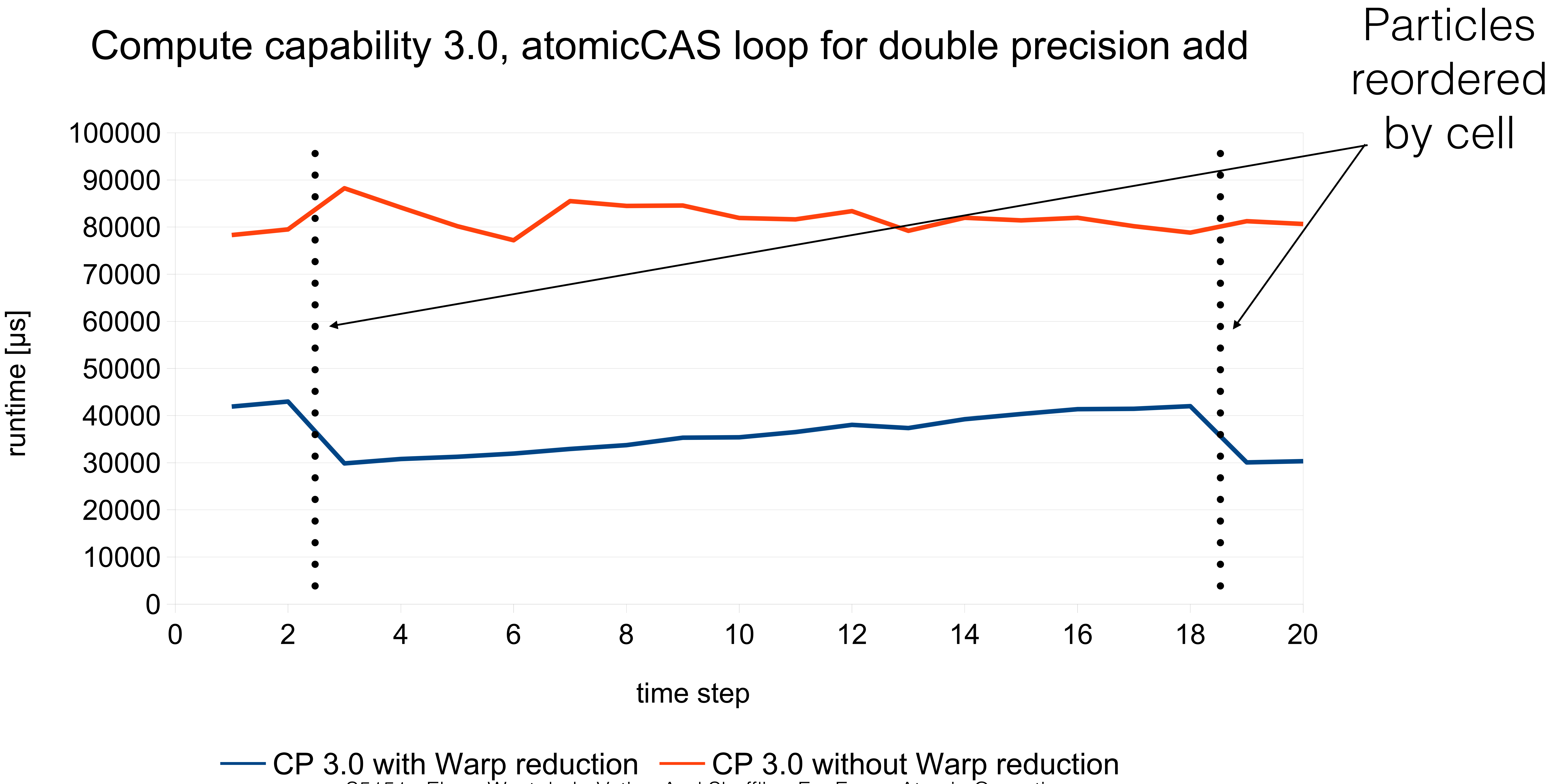
- Benchmarks are from real world MPC application (Multiparticle Collision Dynamics, a particle in cell code for hydrodynamic interactions *)
- Benchmark system used has 10M particles in 1M cells, resulting (unoptimized) in as many atomic adds per parameter per component per iteration
- Benchmarked kernel contains lots of DP computations, but runtime is dominated by 9 atomically added components per thread

*see GTC 2012, S0036, but since Kepler, using atomic operations can be faster than the method described back then

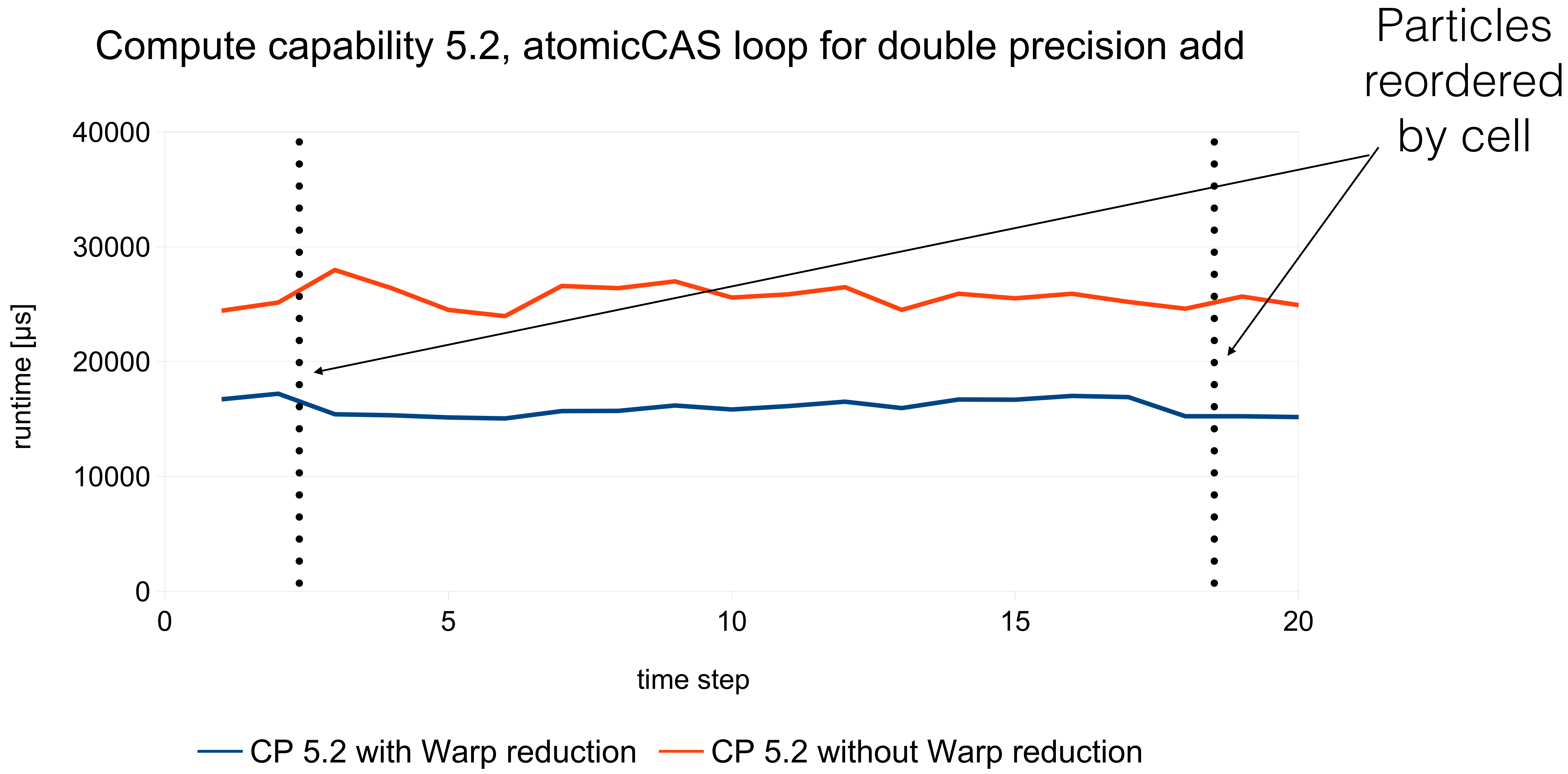
Runtimes for MPC "rotate" kernel



Runtimes for MPC "rotate" kernel

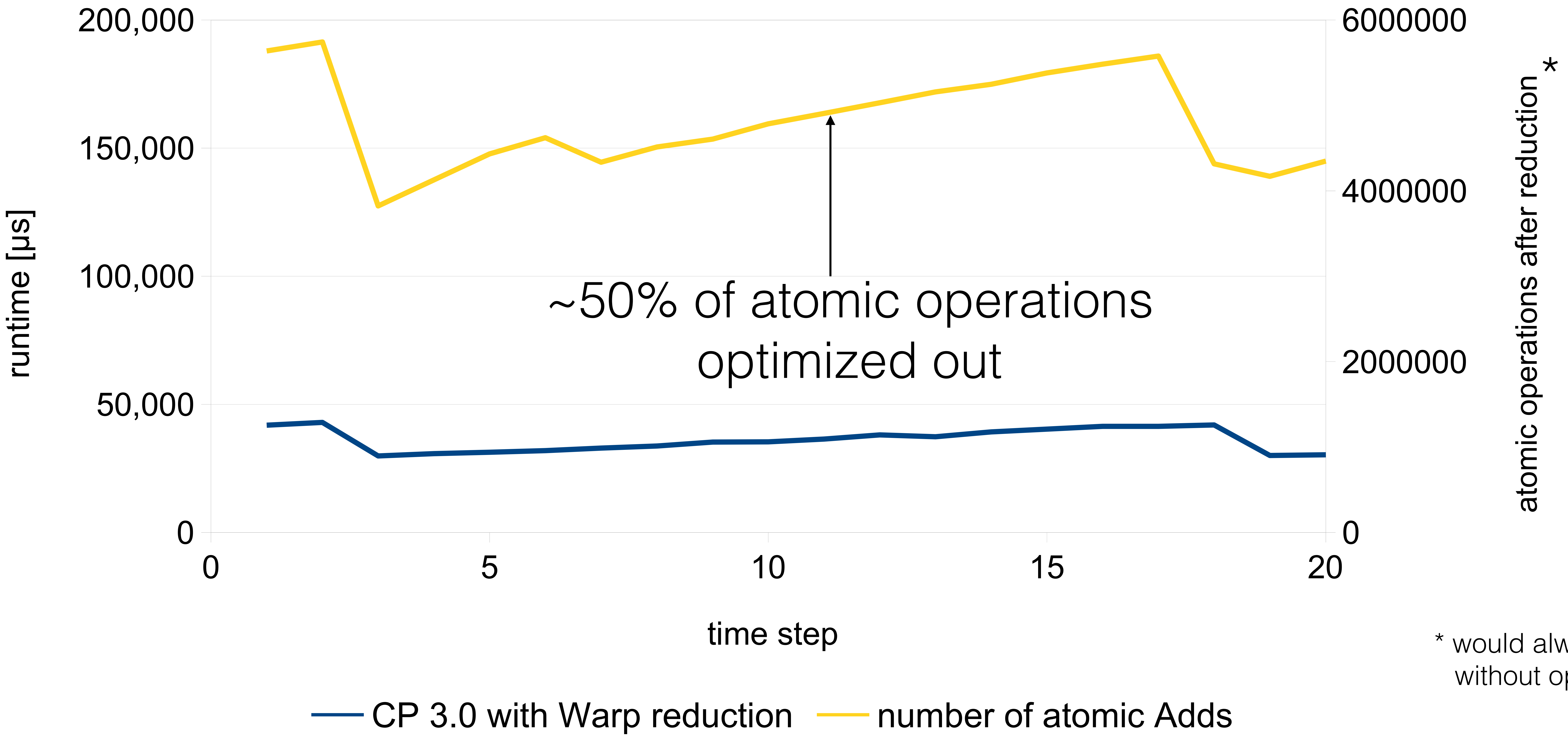


Runtimes for MPC "rotate" kernel



Kernel runtime vs. number of atomic Operations

Compute capabilty 3.0, atomicCAS loop for double precision add



* would always be 10M without optimisation

Conclusion / Outlook

- Useful for problems with a small number of different keys per warp
- Gain depends on architecture, precision and native availability of atomic operation (smaller if available)
- Idea might be extended from warps to blocks, but synchronization might become too expensive

Thank you for your time

Questions?

Appendix: code

```
template<typename G>
__device__ __inline__ uint get_peers(G my_key) {
  uint peers;
  bool is_peer;
  uint unclaimed=0xffffffff;                  // in the beginning, no threads are claimed
  do {
    G other_key=__shfl(key,__ffs(unclaimed)-1);// get key from least unclaimed lane
    is_peer=(my_key==other_key);              // do we have a match?
    peers=__ballot(is_peer);                  // find all matches
    unclaimed^=peers;                         // matches are no longer unclaimed
  } while (!is_peer);                         // repeat as long as we haven’t found our match
  return peers;
}
```

Appendix: code

```
template <typename F>
__device__ __inline__ F add_peers(F *dest, F x, uint peers) {
  int lane=TX&31;
  int first=__ffs(peers)-1;                  // find the leader
  int rel_pos=__popc(peers<<(32-lane));      // find our own place
  peers&=(0xfffffffe<<lane);                 // drop everything to our right
  while(__any(peers)) {                      // stay alive as long as anyone is working
    int next=__ffs(peers);                   // find out what to add
    F t=__shfl(x,next-1);                    // get what to add (undefined if nothing)
    if (next)                                // important: only add if there really is anything
      x+=t;
    int done=rel_pos&1;                      // local data was used in iteration when its LSB is set
    peers&=__ballot(!done);                  // clear out all peers that were just used
    rel_pos>>=1;                             // count iterations by shifting position
  }
  if (lane==first)                           // only leader threads for each key perform atomics
    atomicAdd(dest,x);
  F res=__shfl(x,first);                     // distribute result (if needed)
  return res;                                // may also return x or return value of atomic, as needed
}
```