TECHNISCHE UNIVERSITÄT DRESDEN
Center for Information Services and High Performance Computing (ZIH)

Analyzing Offloading Inefficiencies in Scalable Heterogeneous Applications

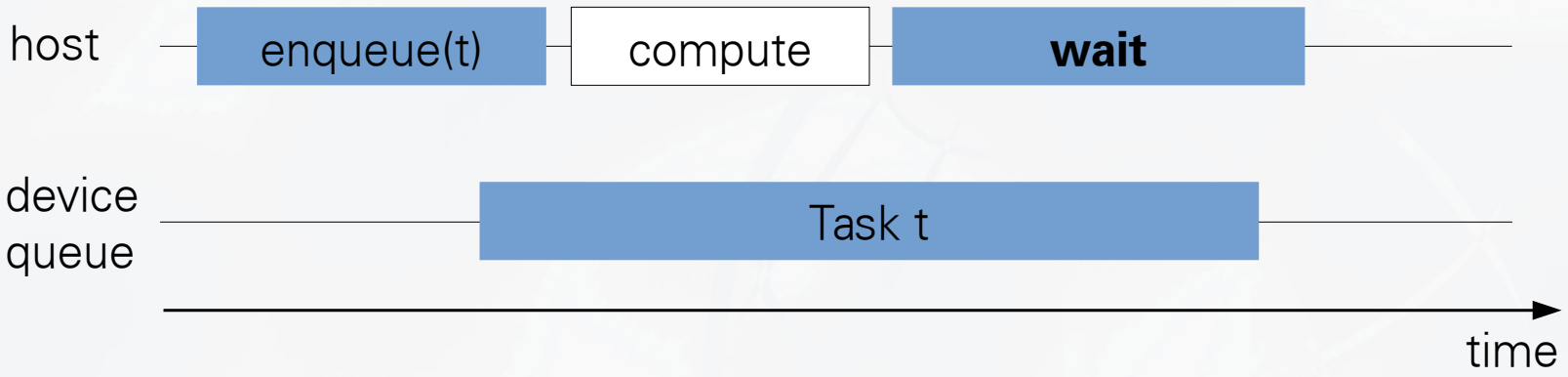
2nd International Workshop on Performance Portable Programming Models for Accelerators (P^3MA)

June 22nd, 2017

Robert Dietrich, Ronny Tschüter, Guido Juckeland, and Andreas Knüpfer

ZIH
Center for Information Services & High Performance Computing

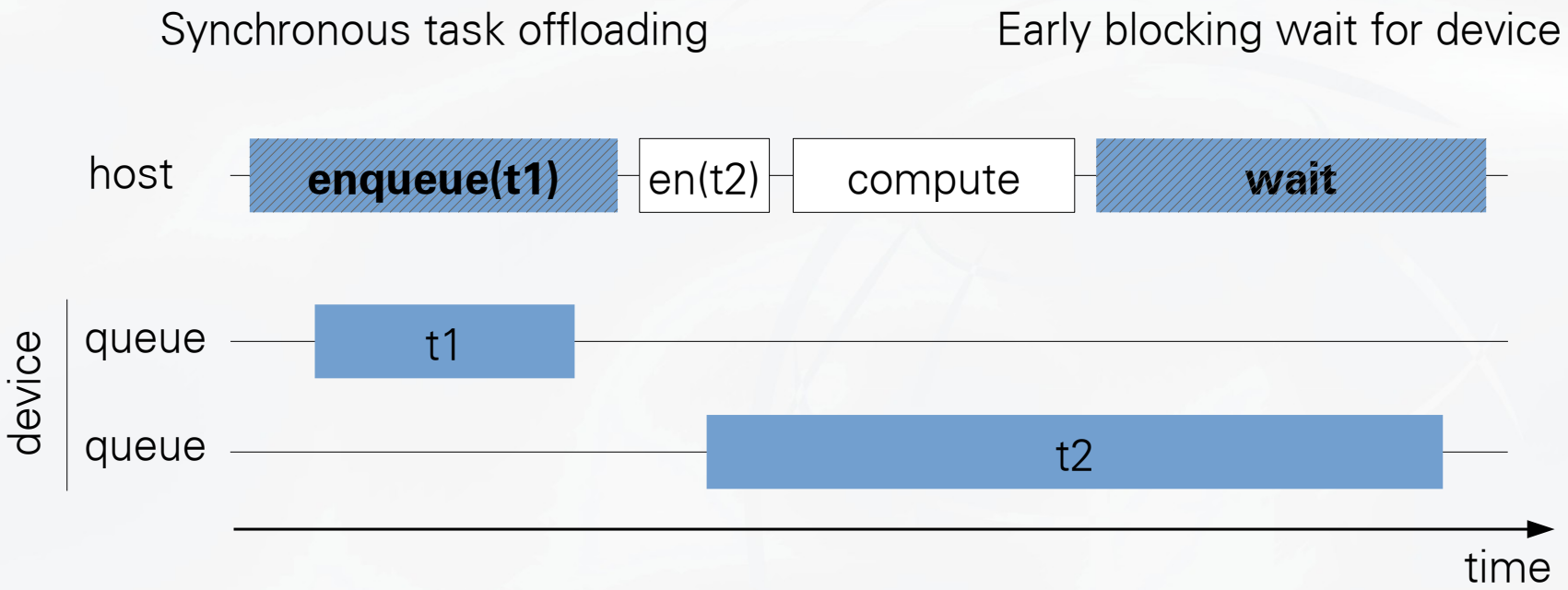
Offloading



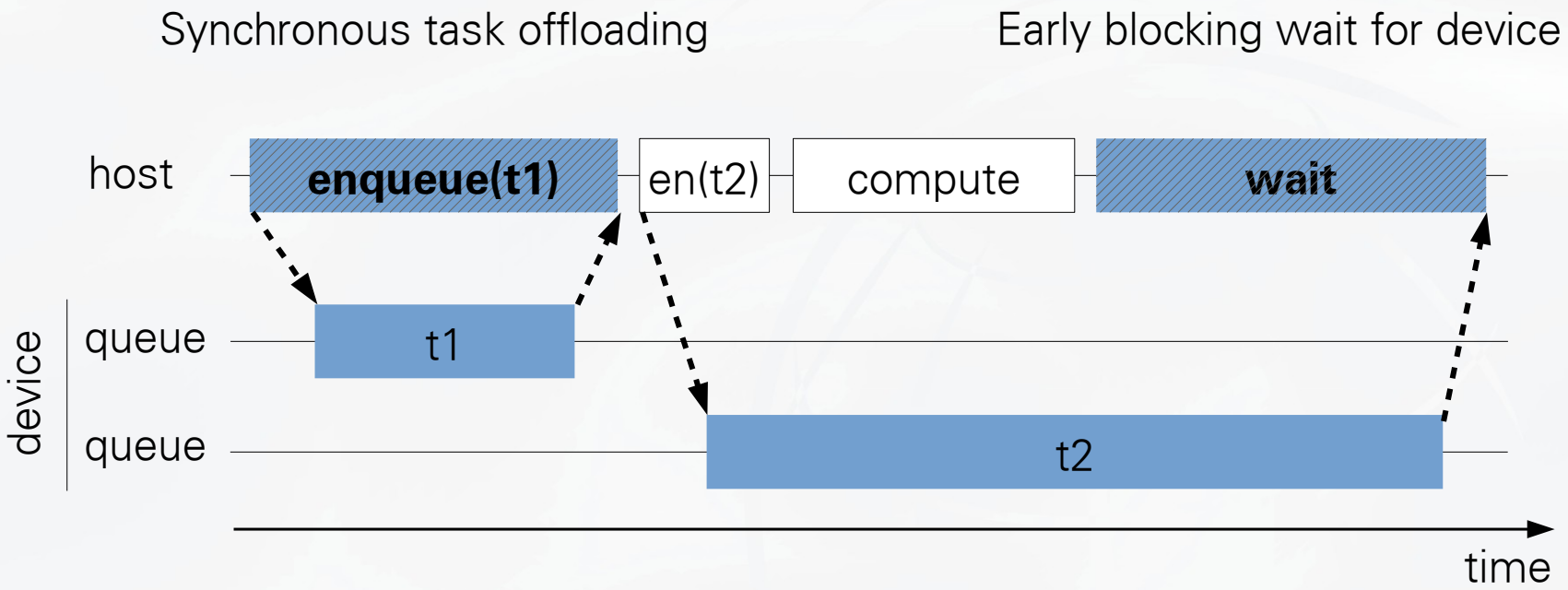
Offloading Inefficiencies – Pattern Overview

Pattern	Accounting
Early blocking wait for device	Time between synchronization start and offloading task end
Host-synchronous offloading	Execution time of the offloaded task
Early test for completion	Execution time of unsuccessful tests
Idle offloading device	Time when no device task is active (sum)
Late data transfer	Transfer time
Multiple consecutive data transfers	Product of the overhead of one transfer and the number of excessive transfers

Blocking Host-Device Synchronization



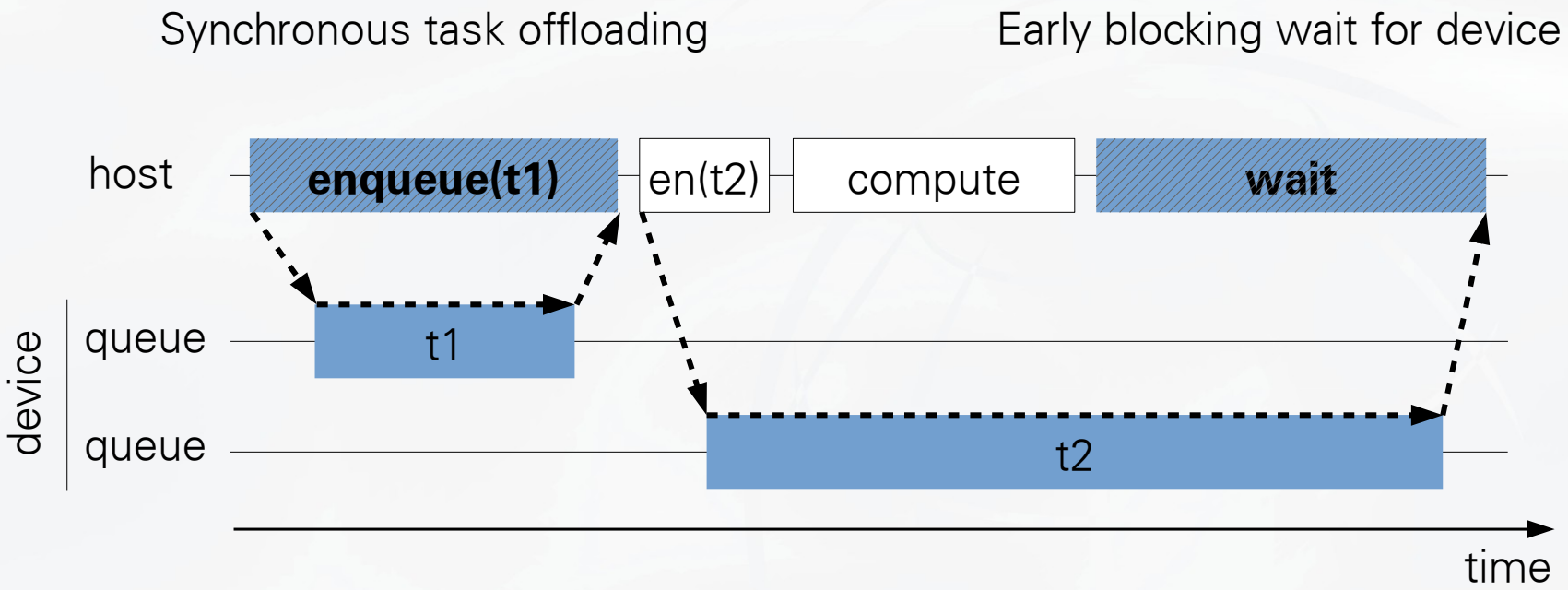
Blocking Host-Device Synchronization



Event dependencies emerge from offloading semantics:

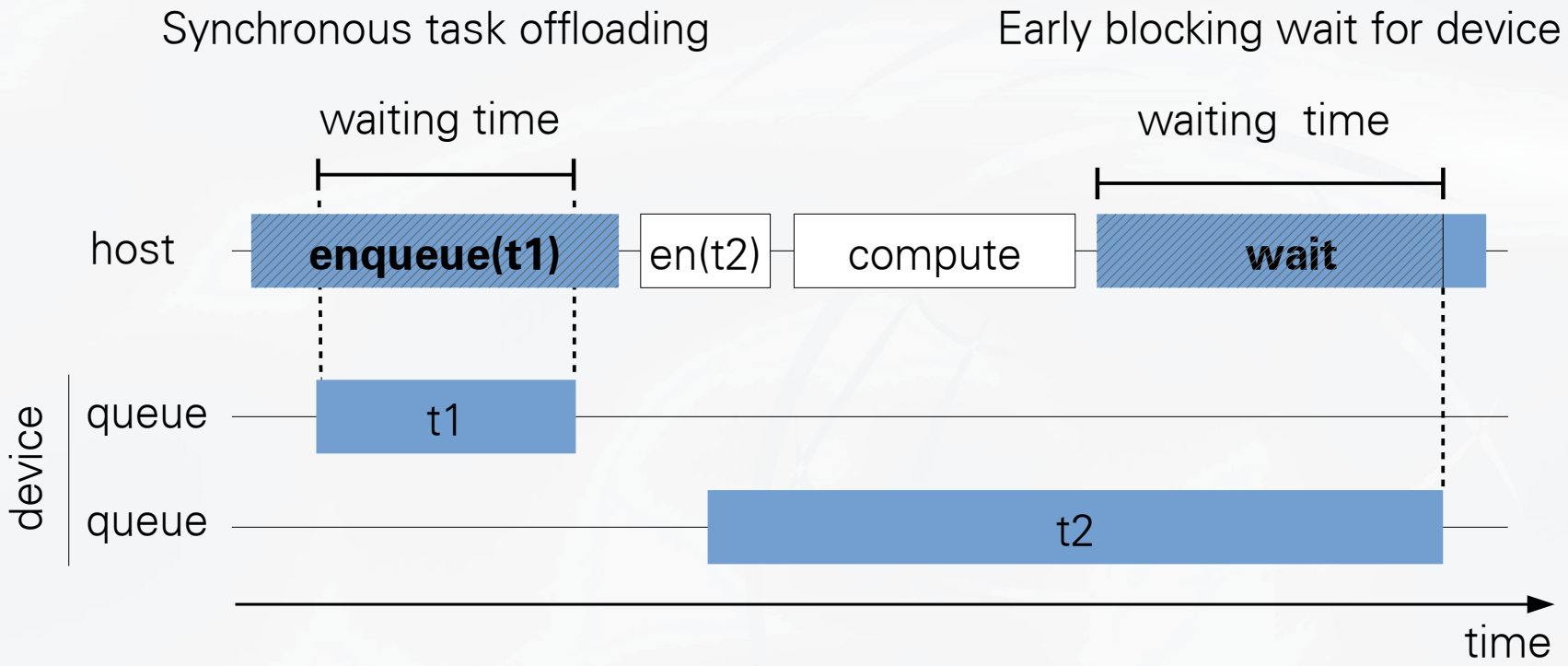
- enqueue start happens before before resp. device task start
- wait end happens after resp. task end

Blocking Host-Device Synchronization

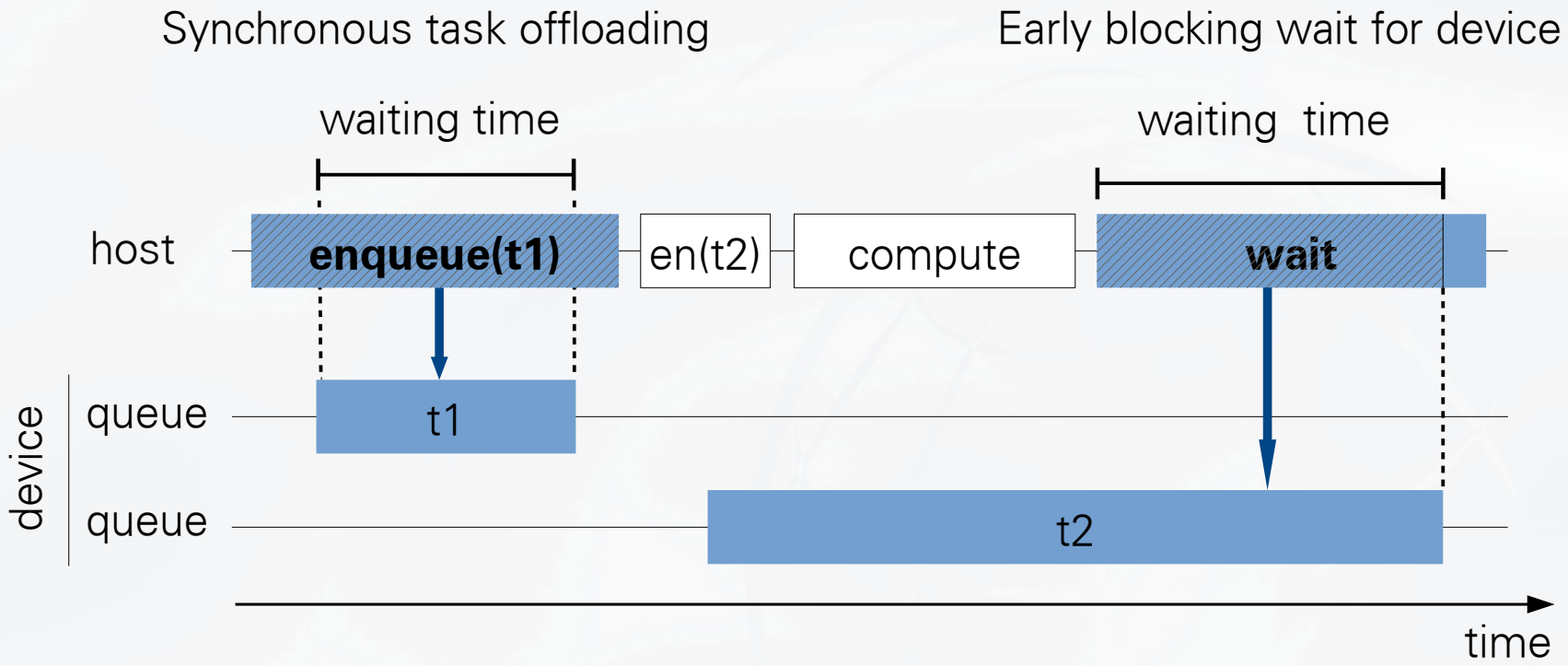


The critical path is the longest path without wait states.

Blocking Host-Device Synchronization

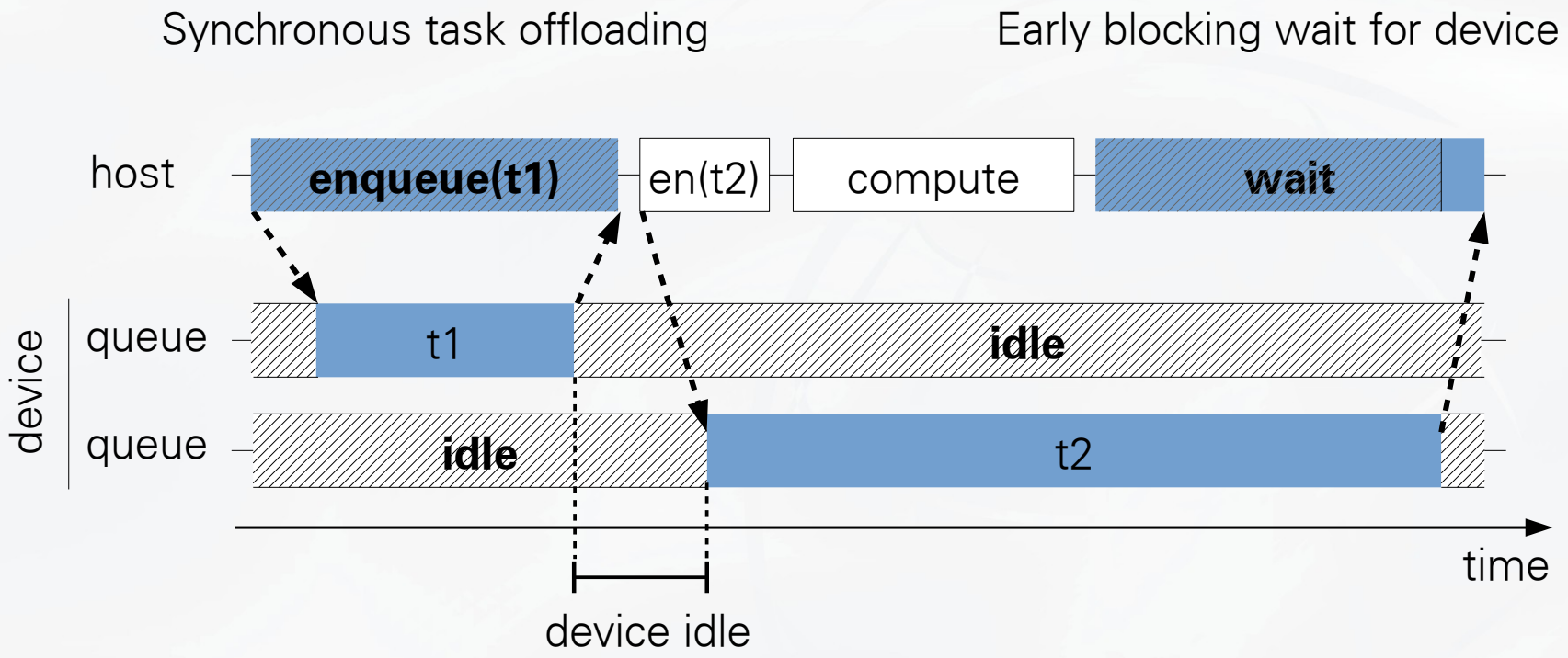


Blocking Host-Device Synchronization

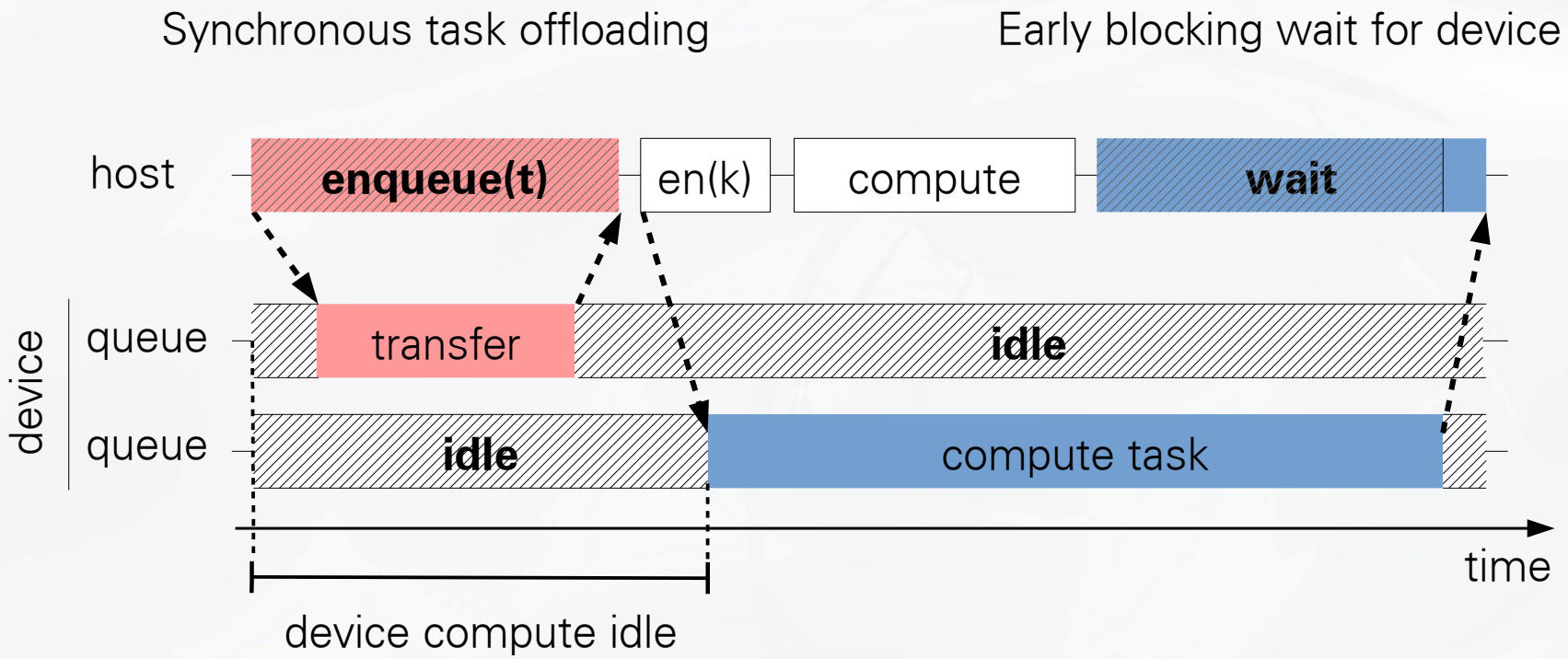


Device tasks can be blamed for causing wait states.

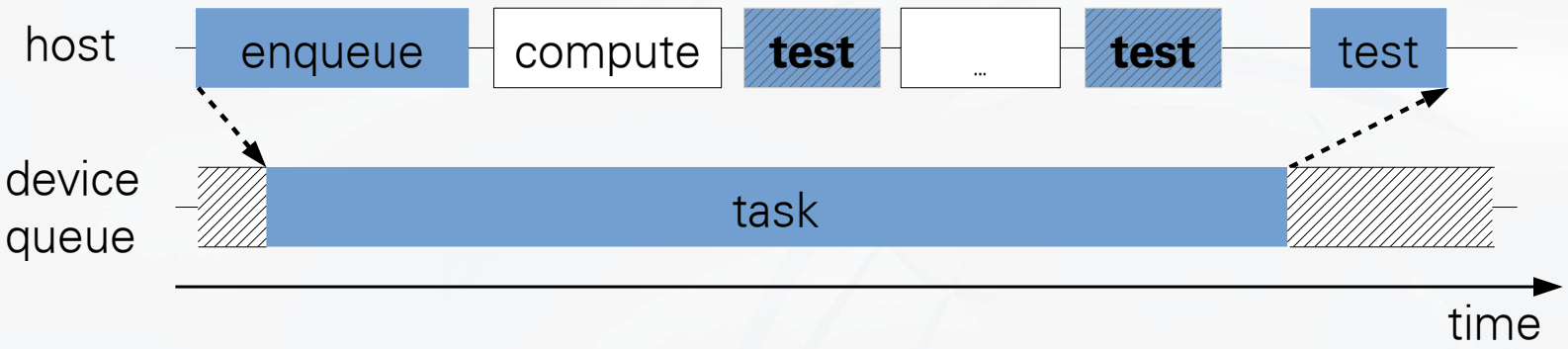
Blocking Host-Device Synchronization & Device Idle



Blocking Host-Device Synchronization & Device Idle



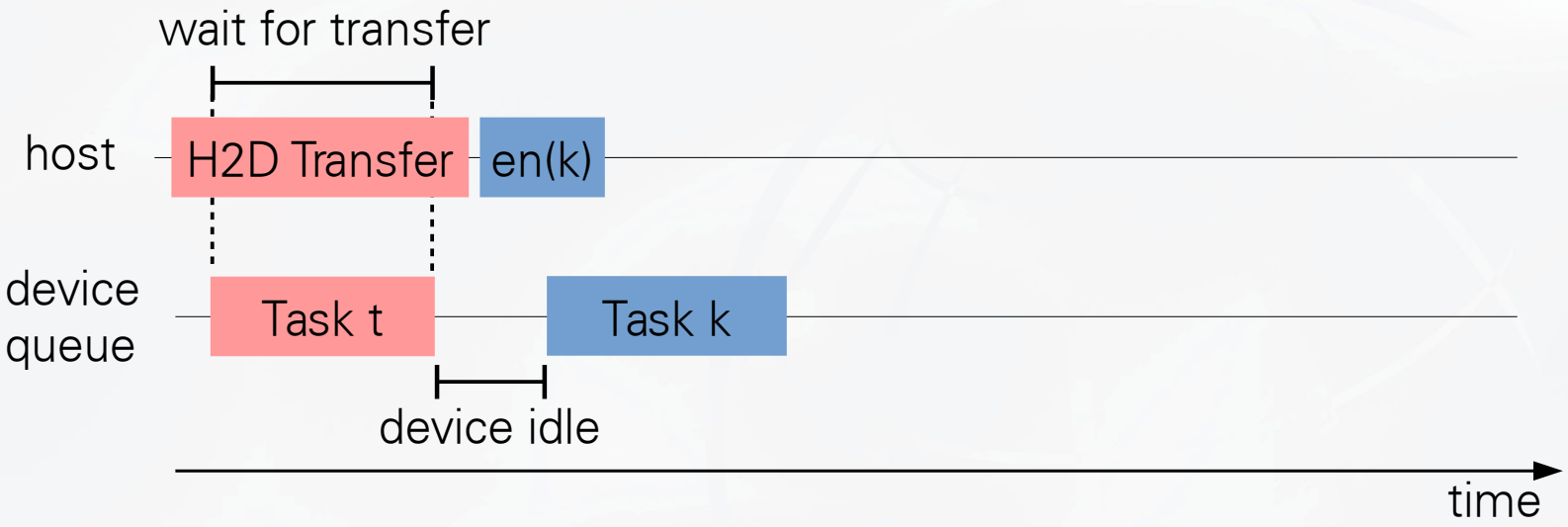
Offloading State – Query Routines, Non-blocking Sync



	Device	Stream/Queue	Task/Event
CUDA		cuStreamQuery	cuEventQuery
OpenCL			clGetEventInfo
OpenACC	acc_async_test_all	acc_async_test	acc_async_test

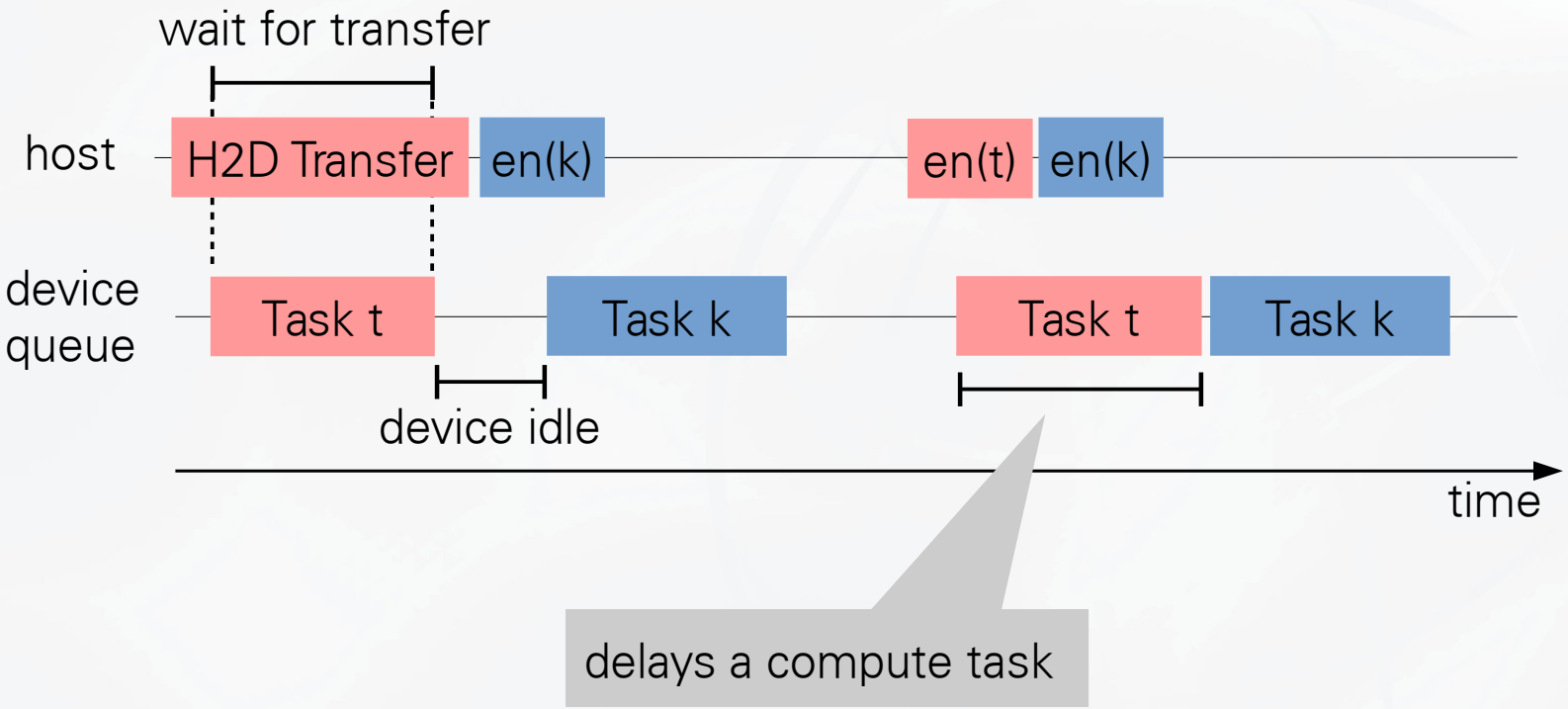
Late Transfer Pattern

A data movement task is considered inefficient, if it delays the execution of a device kernel (compute) and does not fully overlap with a kernel.



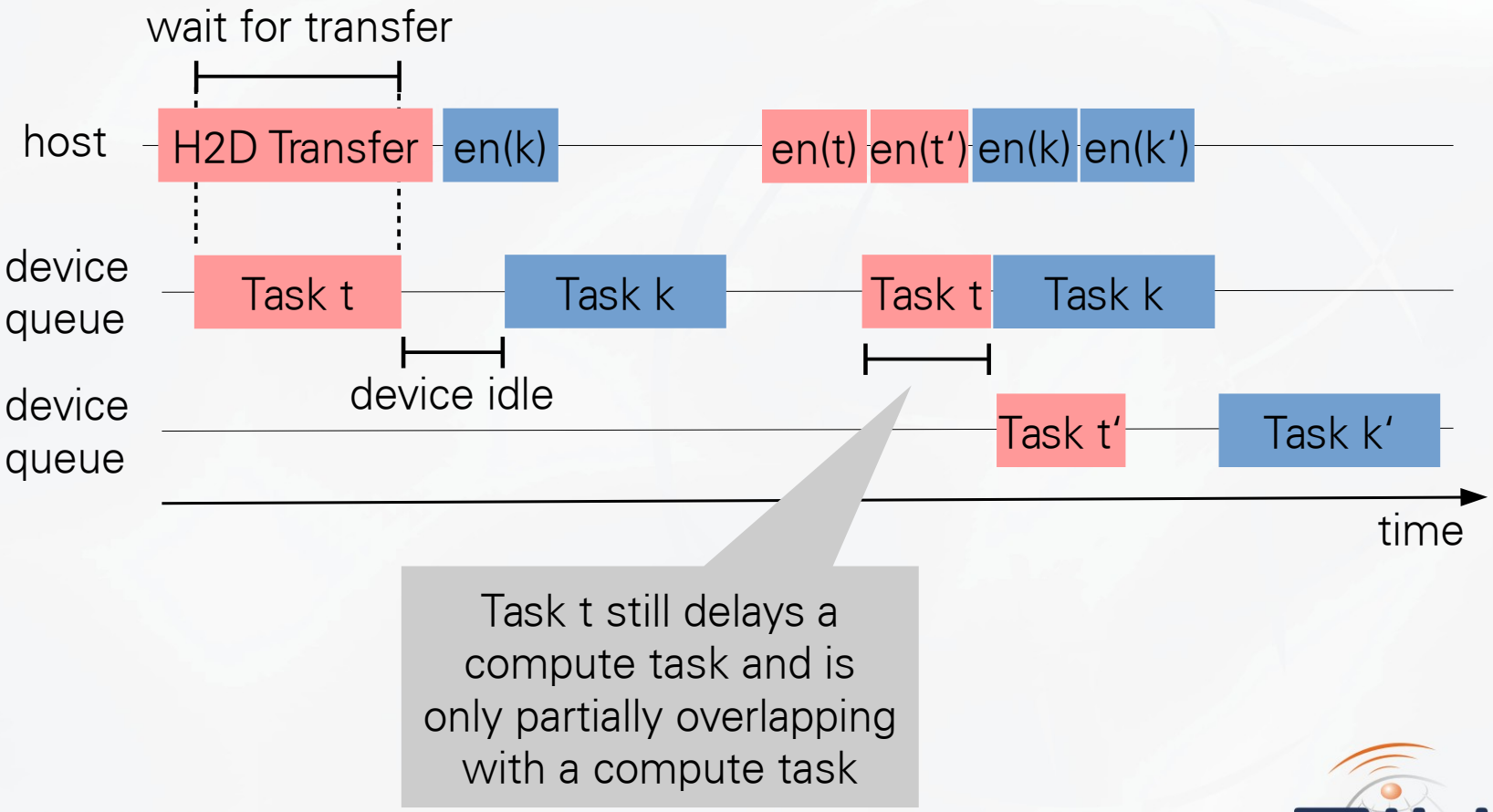
Late Transfer Pattern

A data movement task is considered inefficient, if it delays the execution of a device kernel (compute) and does not fully overlap with the kernel



Late Transfer Pattern

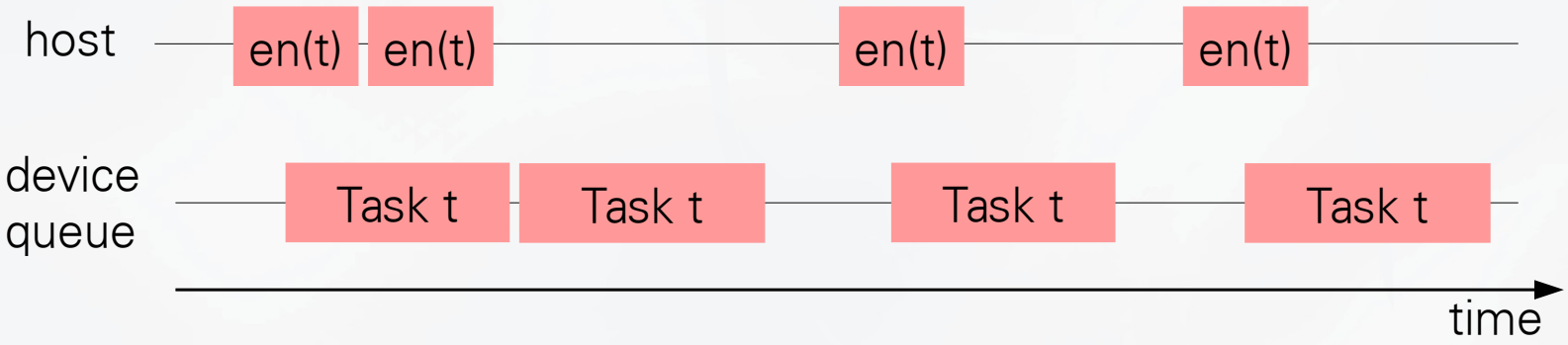
A data movement task is considered inefficient, if it delays the execution of a device kernel (compute) and does not fully overlap with the kernel



Consecutive Data Transfers

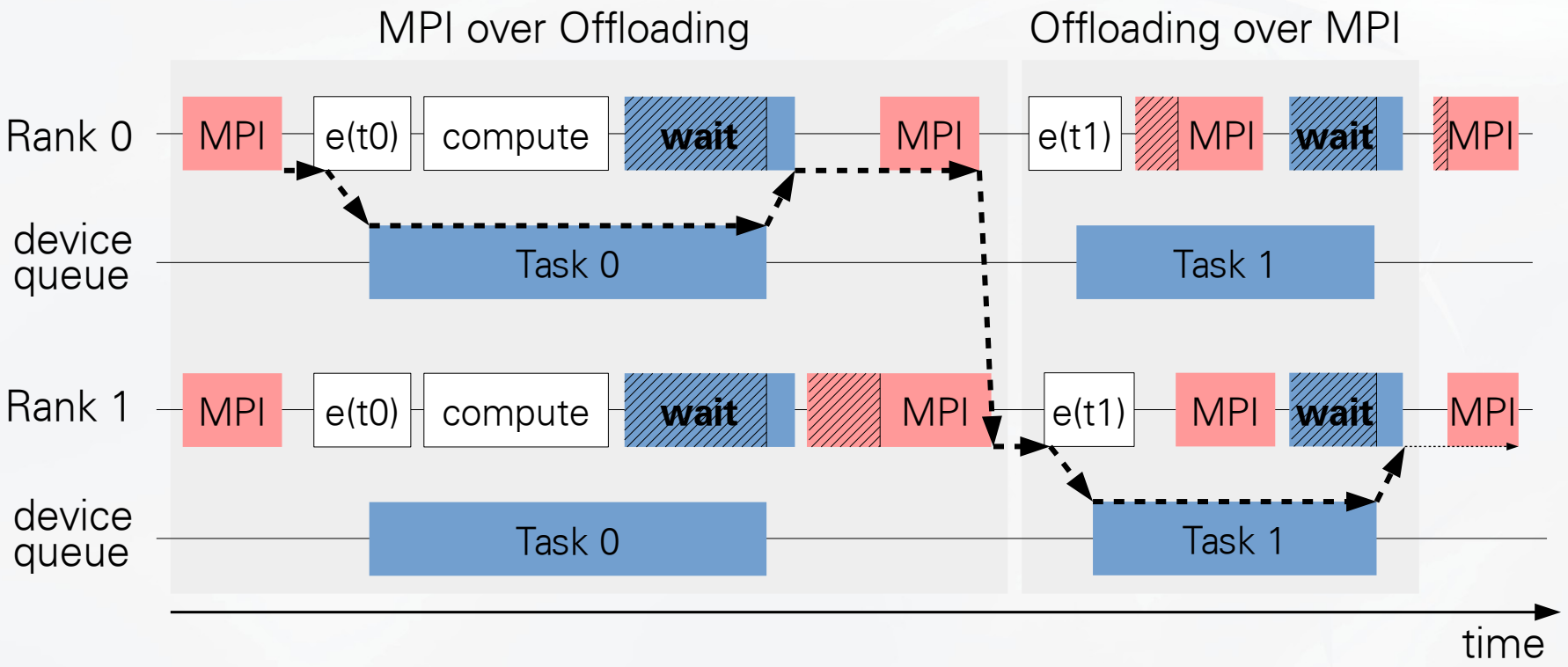
Two consecutive data movements are inefficient, if

- they are tasks on the same device
- they are not overlapping
- neither is overlapping with a compute task on the same device
- no compute task on the same device is executed between them
- no communication of any other paradigm is initiated or ends between the trigger of the offloading data movements

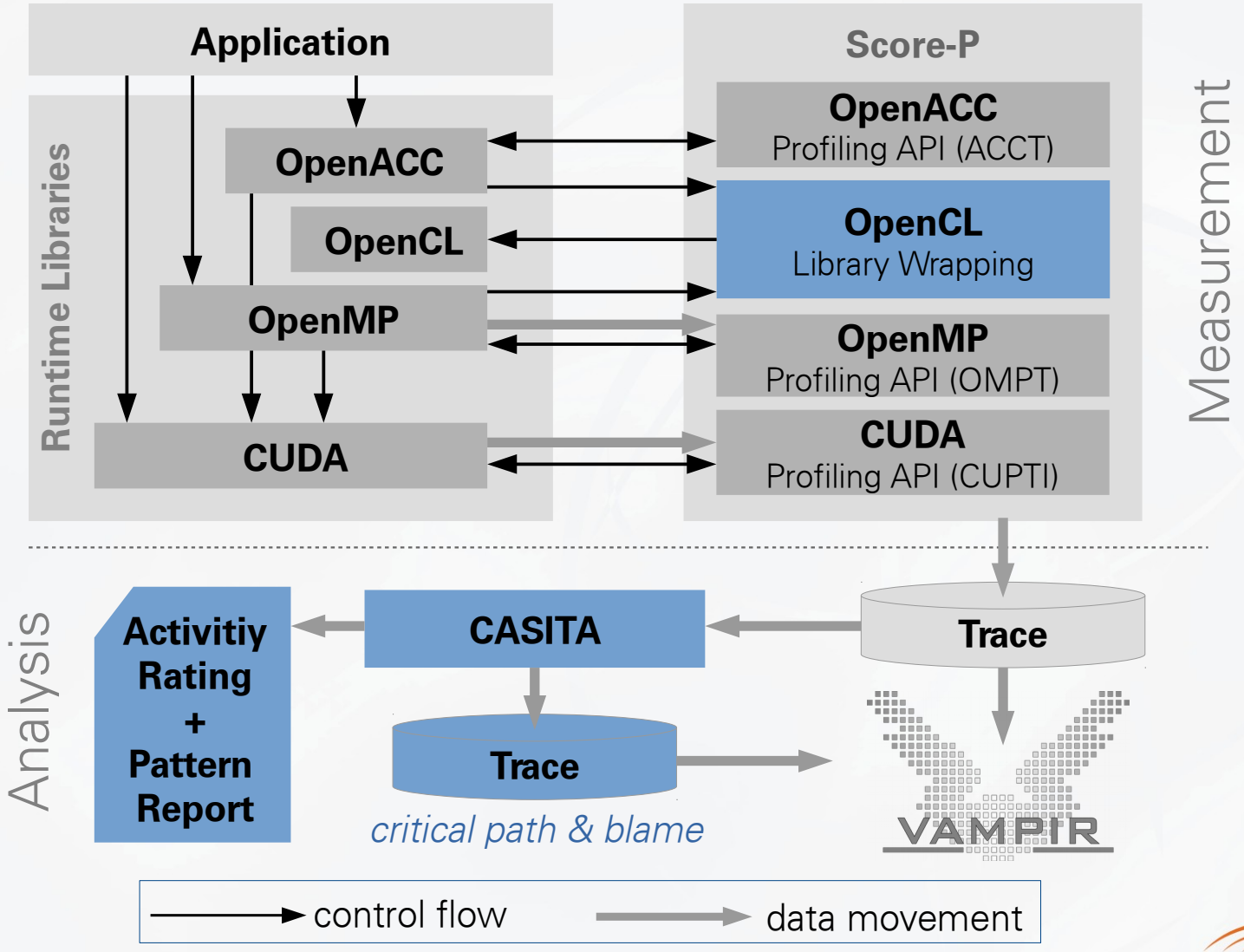


MPI and Offloading

MPI, OpenMP, and offloading are most often used in a hierarchical fashion.
→ device idles during MPI communication

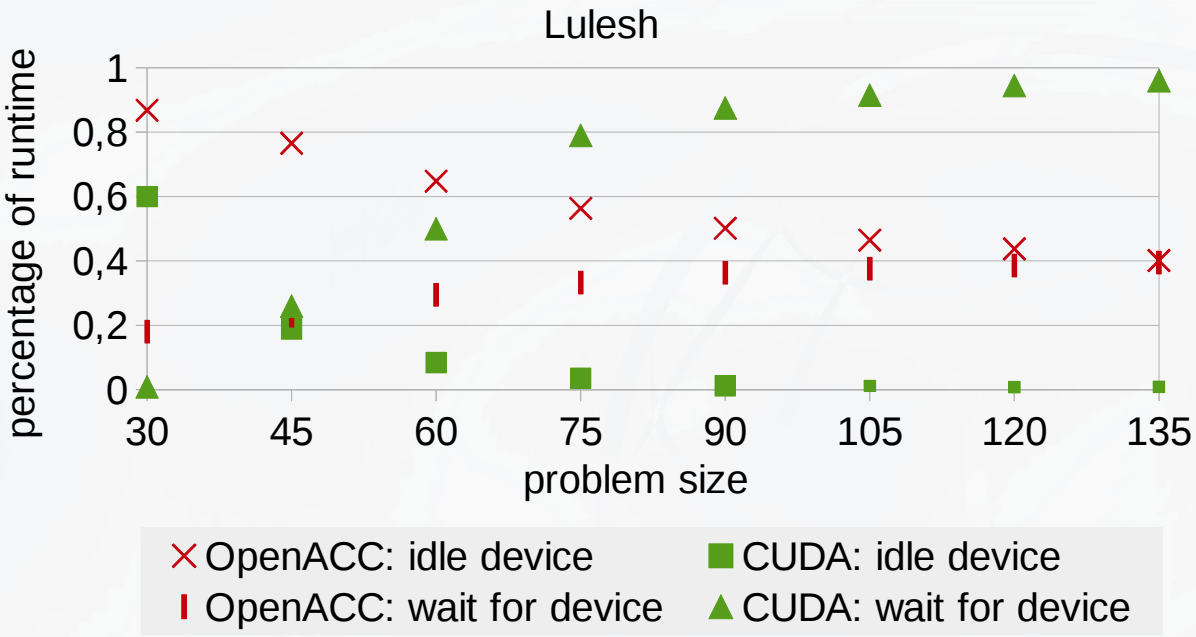


Performance Analysis Control and Data Flow



Application Studies

- Mini-Apps with CUDA and OpenACC: **Lulesh** and **Cloverleaf** (no load-balancing between host and device)



- **Gromacs** with OpenCL for offloading
  - uses MPI within OpenMP parallel regions
  - MPI communication concurrent to device compute kernels

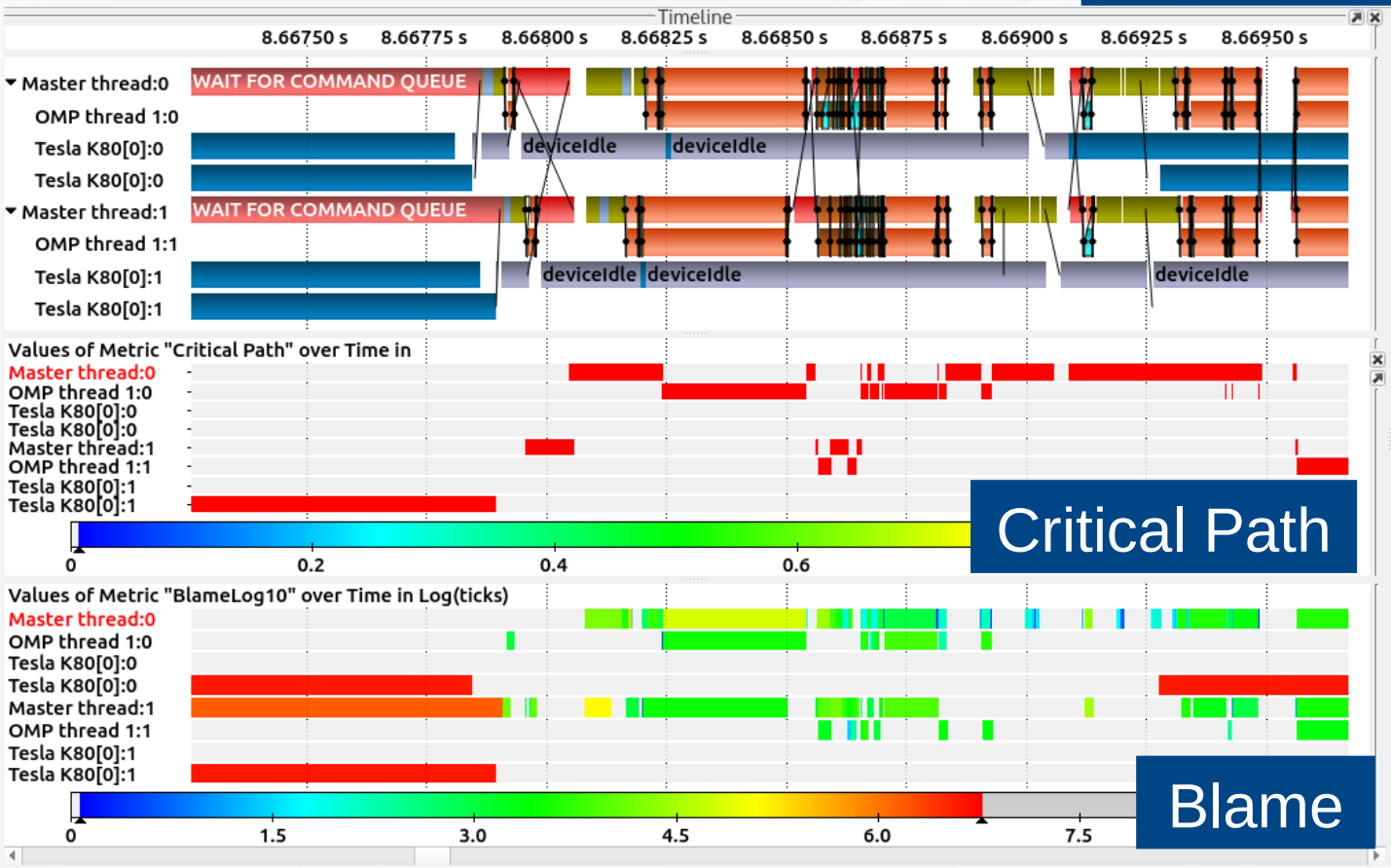
Gromacs Pattern Summary

	1Process, 2T, 1D	2P, 4Threads, 2D	8P, 16T, 8 Devices
Total program runtime	33.289s	28.86s	20.25s
Total waiting time	1.387s	7.047s	**79.954s**
MPI wait patterns		63 618 (1.681s)	**670 985 (55.837s)**
Late Sender		2 (0.000s)	8 813 (0.208s)
MPI_Sendrecv		50 760 (1.034s)	**480 367 (14.1208s)**
(Early) MPI_Waitall			57 878 (4.247s)
MPI collectives		12 852 (0.645s)	**123 465 (37.260s)**
OpenMP barrier wait	168 296 (0.729s)	400 620 (2.405s)	1 649 924 (12.652s)
Offloading			
Idle device	**13.590 (48.80%)**	**25.916s (56.40%)**	**105.042s (83.94%)**
Compute idle device	14.151 (50.81%)	26.674s (58.05%)	106.563s (85.15%)
Early blocking wait	**574 (0.492s)**	**8 509 (2.519s)**	**8 225 (3.288s)**
Consecutive transfers	6 108 (0.327s)	22 426 (0.581s)	57 317 (0.537s)
Avg. kernel startup	0.072ms	0.473ms	0.297ms

Gromacs Rating (2 MPI processes, 2 Threads, 2 GPUs)

Region	Time [s]	Time on CP [s]	CP Ratio [%]	Blame Ratio [%]	Blame on CP [s]
nbnxn_kernel_ElecEwTwinCut_*_opencl	**6,103**	0,51839	1,796032	**17,064718**	**0,446945**
!$omp parallel @pme.cpp:1092	9,481	2,05786	7,129749	3,462656	0,12569
!$omp parallel @clincs.cpp:2423	1,744	0,47304	1,638916	2,062237	0,084531
!$omp for @update.cpp:1659	6,189	1,55487	5,387079	2,66123	0,080662
!$omp for @pme-spread.cpp:900	9,610	2,26456	7,845894	1,698048	0,071251
!$omp for @domdec_constraints.cpp:530	0,177	0,07482	0,259248	1,625612	0,061066
!$omp for @nbnxn_search.cpp:4220	2,312	0,60371	2,091658	1,406513	0,056
!$omp for @nbnxn_grid.cpp:1359	0,507	0,15449	0,53527	1,420647	0,054797
!$omp for @pme.cpp:1207	9,328	1,26952	4,398443	2,10649	0,052324
!$omp for @nbnxn_atomdata.cpp:1171a	0,489	0,15013	0,52016	1,341189	0,051821
!$omp for @nbnxn_atomdata.cpp:1643	0,766	0,19552	0,677437	0,903087	0,048007
nbnxn_kernel_ElecEw_VdwL_*_opencl	**21,86**	0,42873	1,485416	**6,119703**	**0,047025**
!$omp for @domdec_topology.cpp:1985	0,269	0,09263	0,320948	1,116099	0,043454
!$omp for @constr.cpp:446	2,425	0,61785	2,140632	0,706439	0,034615
!$omp for @pme-spread.cpp:874	3,058	0,72319	2,505619	0,613851	0,027763
!$omp for @listed-forces.cpp:497	2,140	0,55925	1,937618	0,51759	0,023518

Gromacs – Vampir



Conclusion

- Offloading works similar in different offloading APIs (CUDA, OpenCL, OpenACC, OpenMP target)
- Pattern analysis exposes wait states (and real waiting times) on the host as well as device [compute] idle time
- Inefficiency patterns pinpoint on optimization opportunities
  - load balancing (waiting and idle times)
  - transfer bandwidth (consecutive data transfers)
  - ...
- Wait states are the basis for critical-path and root-cause analysis which allows to identify program regions that contribute to the total program runtime and/or caused other threads/tasks to wait
- Visual analysis with Vampir helps to better understand the execution inefficiency
- Required profiling information is available from the OpenACC 2.5 profiling interface, OMPT for OpenMP, CUPTI for CUDA, and OpenCL library wrapping