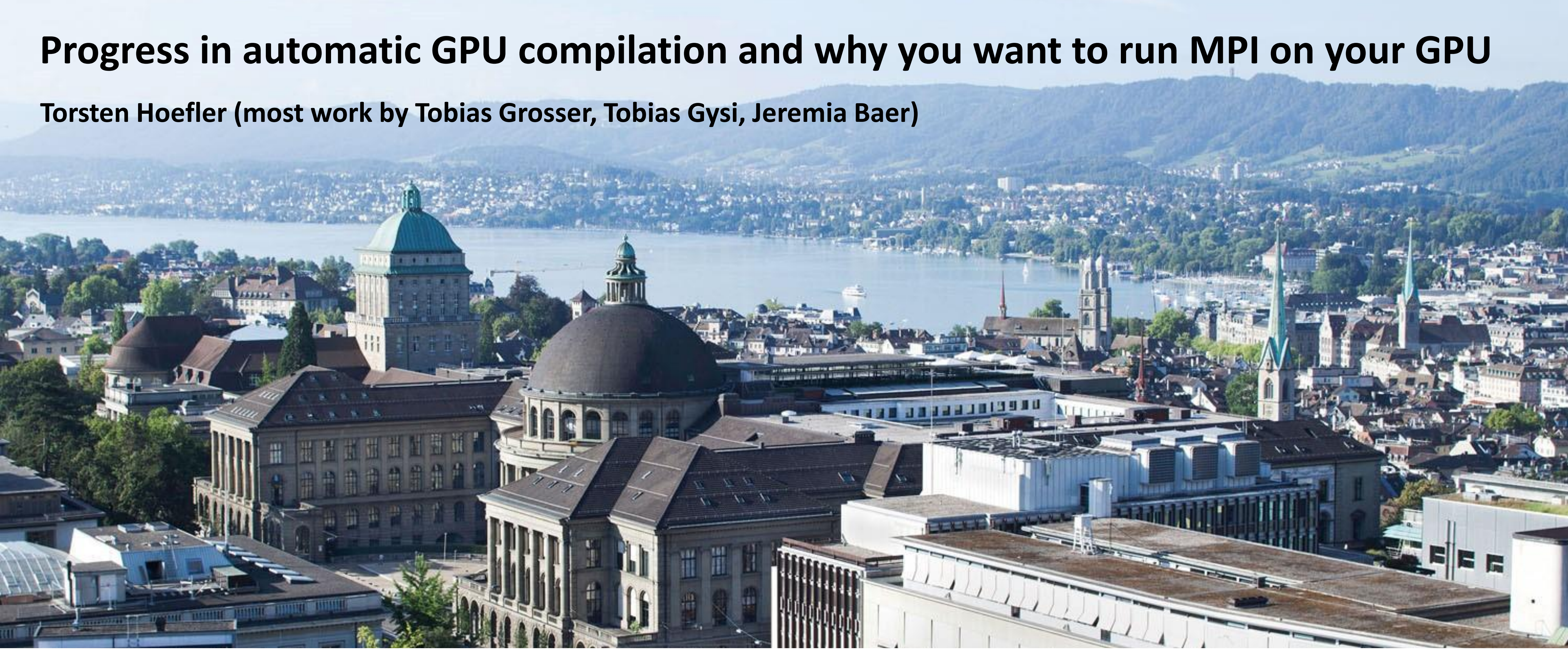
Progress in automatic GPU compilation and why you want to run MPI on your GPU

Torsten Hoefler (most work by Tobias Grosser, Tobias Gysi, Jeremia Baer)



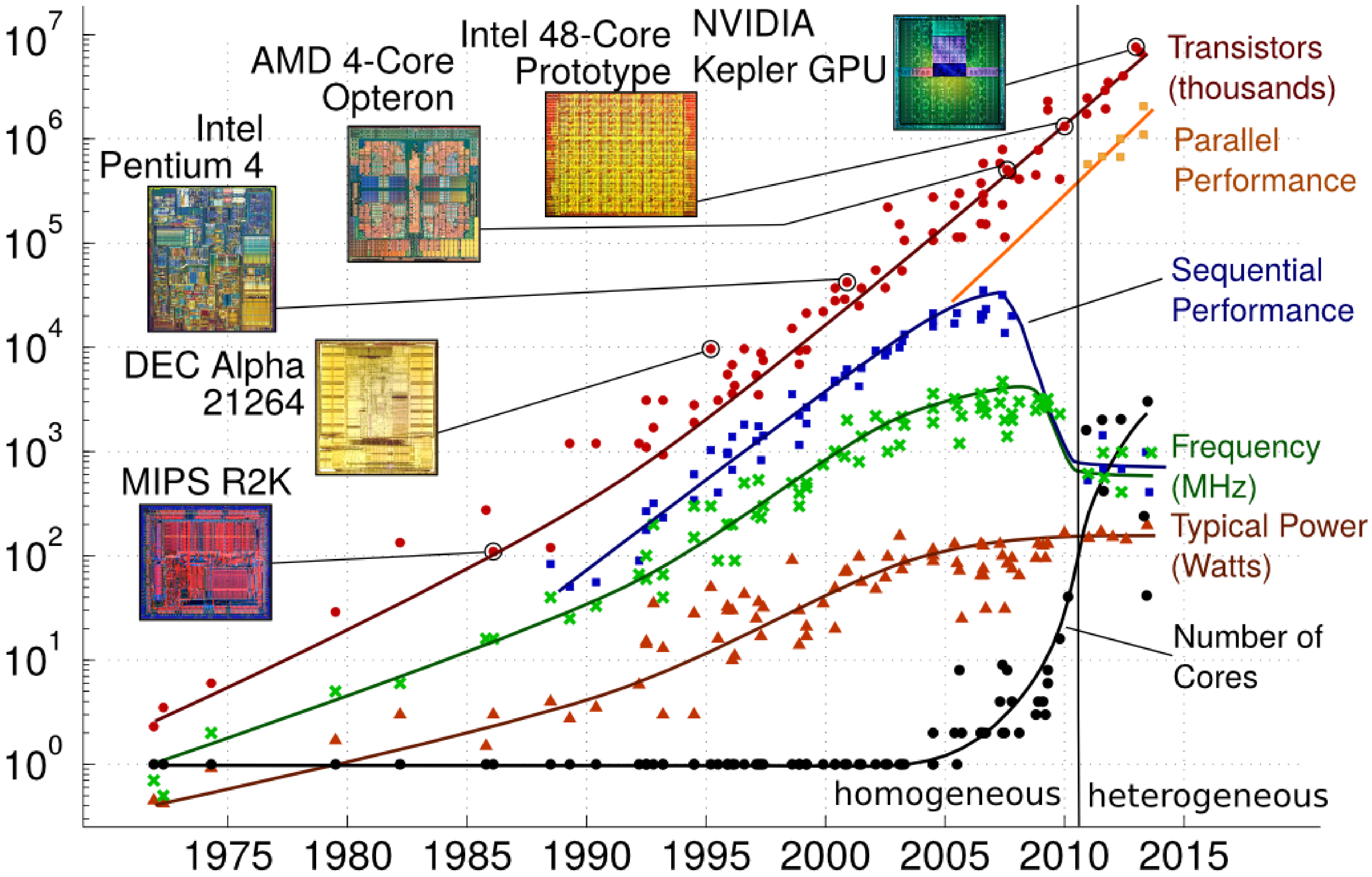
pasc
Platform for Advanced Scientific Computing

suk·cus
Swiss university conference

ETH-RAT

CSCS

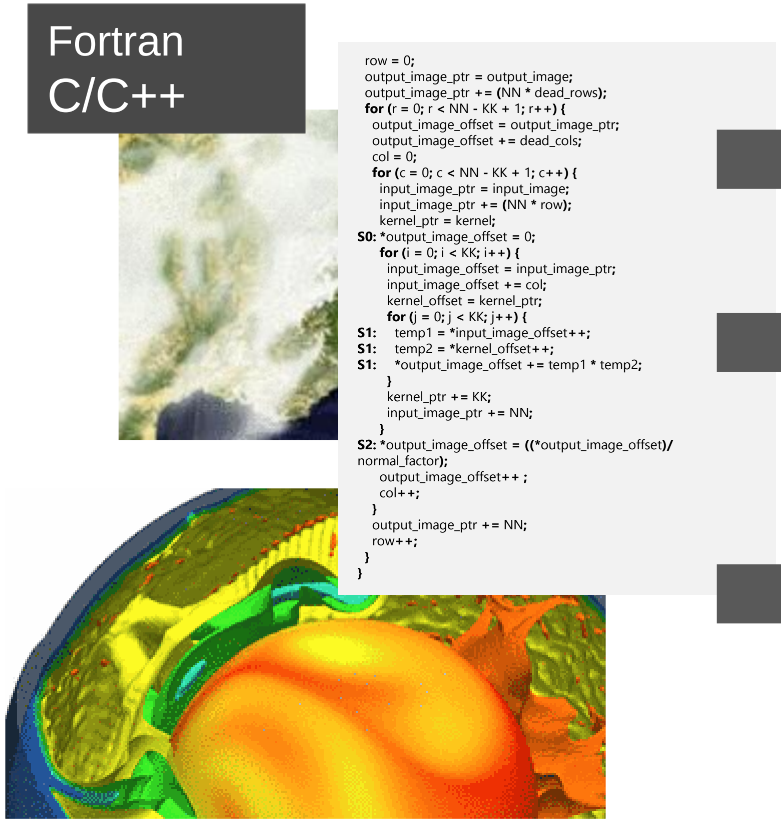
Evading various "ends" – the hardware view



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

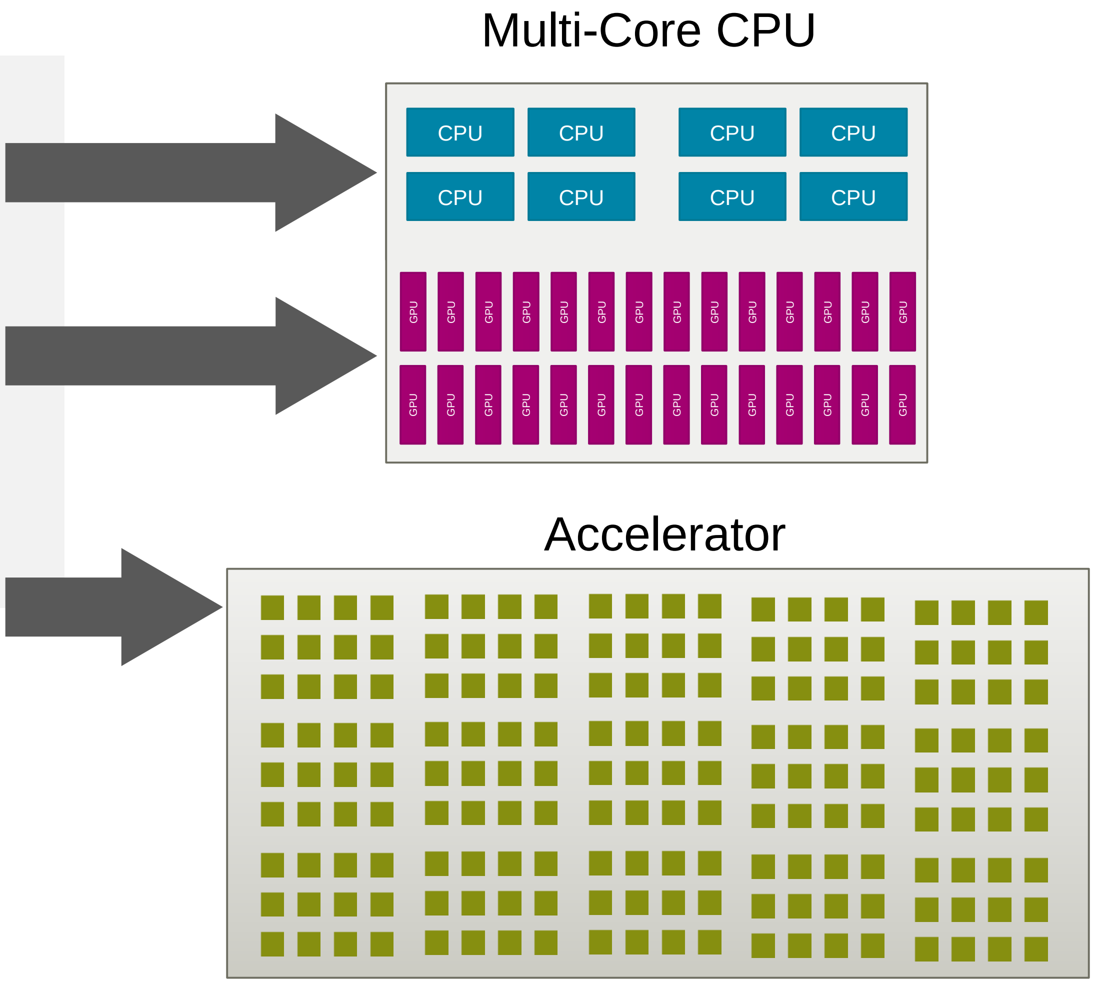
Sequential Software

Parallel Hardware

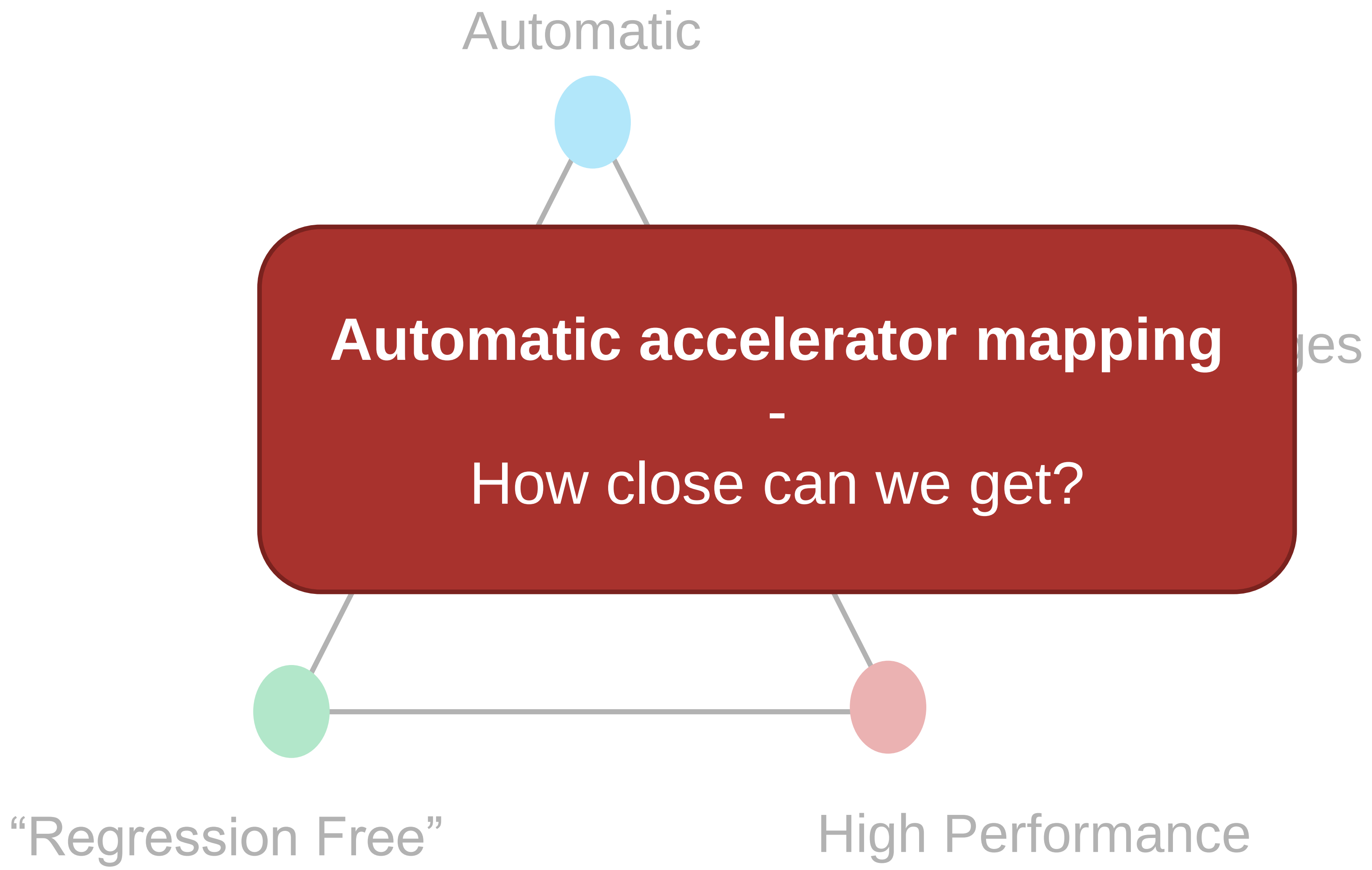


Fortran
C/C++

```
  row = 0;
  output_image_ptr = output_image;
  output_image_ptr += (NN * dead_rows);
  for (r = 0; r < NN - KK + 1; r++) {
    output_image_offset = output_image_ptr;
    output_image_offset += dead_cols;
    col = 0;
    for (c = 0; c < NN - KK + 1; c++) {
      input_image_ptr = input_image;
      input_image_ptr += (NN * row);
      kernel_ptr = kernel;
S0: *output_image_offset = 0;
      for (i = 0; i < KK; i++) {
        input_image_offset = input_image_ptr;
        input_image_offset += col;
        kernel_offset = kernel_ptr;
        for (j = 0; j < KK; j++) {
S1:       temp1 = *input_image_offset++;
S1:       temp2 = *kernel_offset++;
S1:       *output_image_offset += temp1 * temp2;
        }
        kernel_ptr += KK;
        input_image_ptr += NN;
      }
S2: *output_image_offset = ((*output_image_offset)/
normal_factor);
      output_image_offset++ ;
      col++;
    }
    output_image_ptr += NN;
    row++;
  }
}
```



Design Goals



Automatic accelerator mapping

-

How close can we get?

Theory

Tool: Polyhedral Modeling

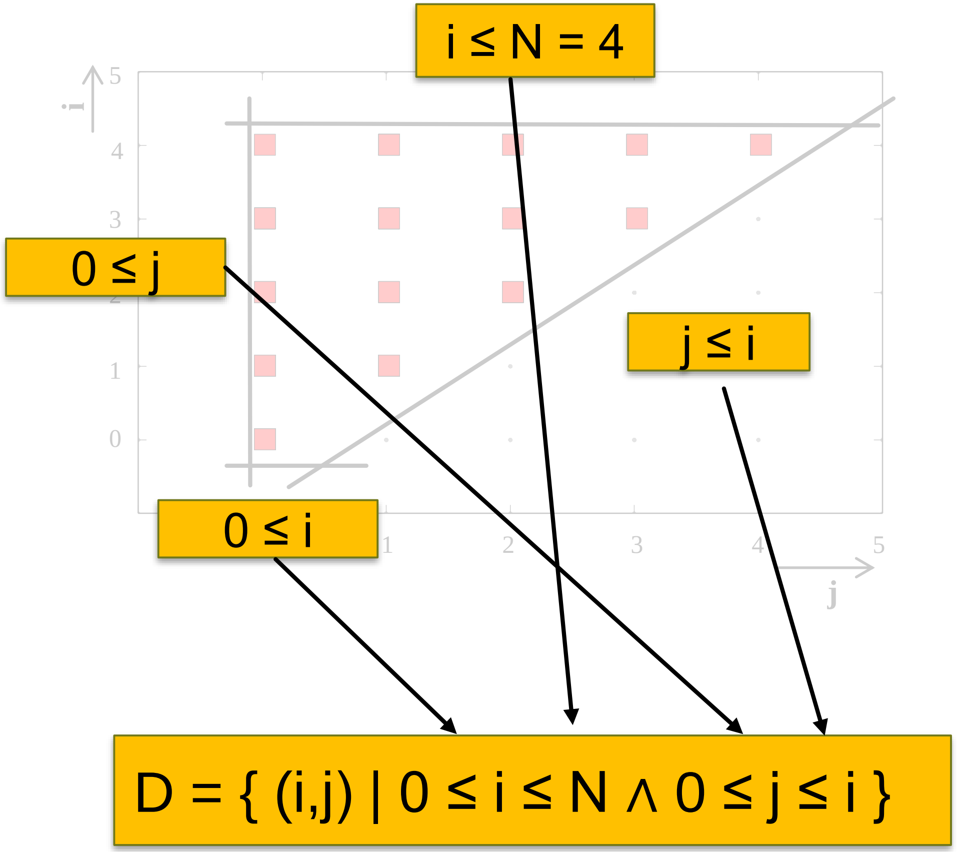
Program Code

```
for (i = 0; i <= N; i++)
  for (j = 0; j <= i; j++)
    S(i,j);
```

N = 4

(i, j) = (4,4)

Iteration Space



i ≤ N = 4

0 ≤ j

j ≤ i

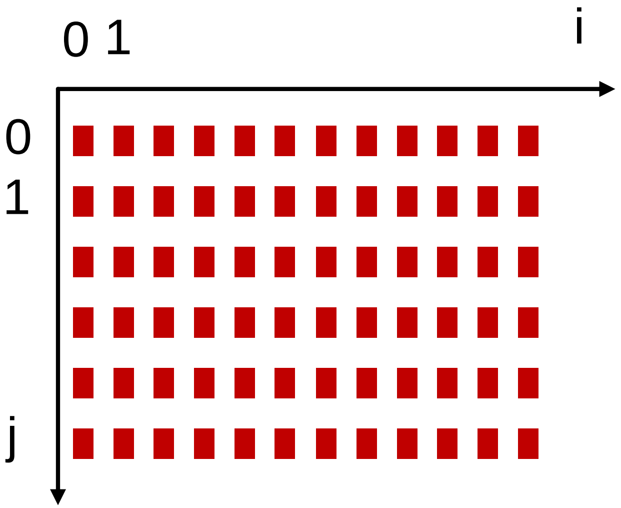
0 ≤ i

$D = \{ (i,j) \mid 0 \le i \le N \wedge 0 \le j \le i \}$

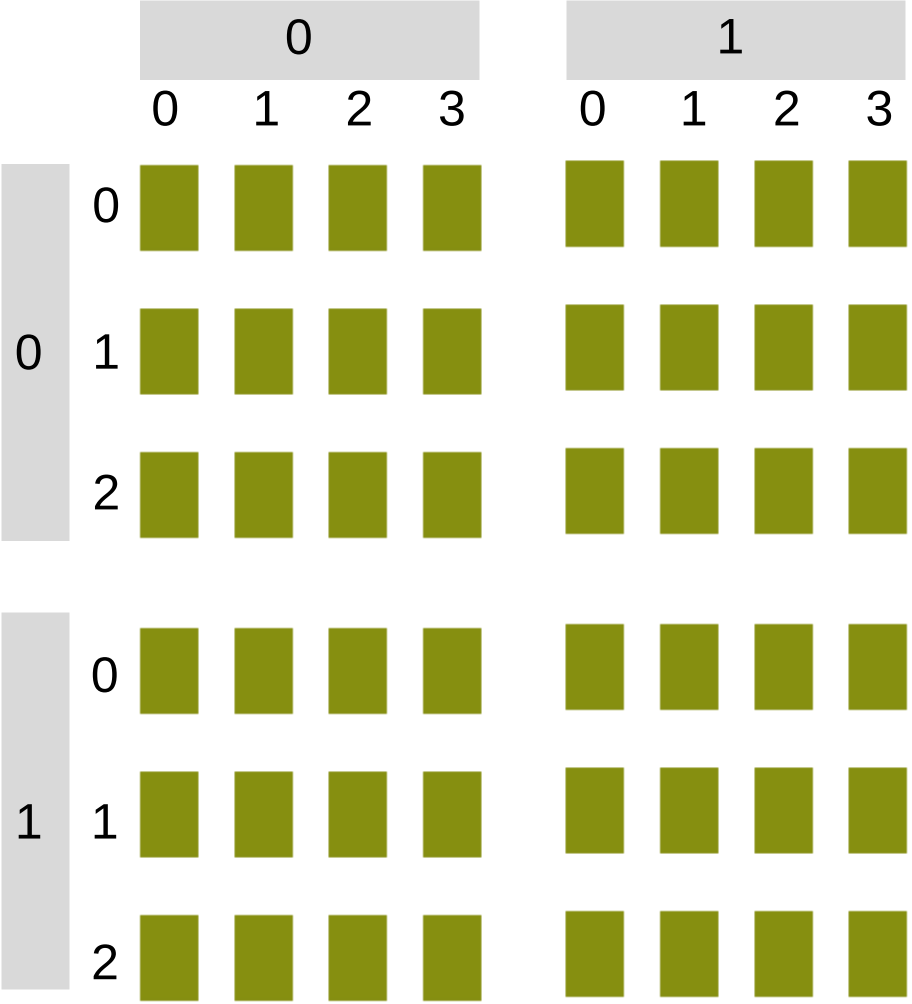
Tobias Grosser et al.: Polly -- Performing Polyhedral Optimizations on a Low-Level Intermediate Representation, PPL‘12

Mapping Computation to Device

Iteration Space



Device Blocks & Threads

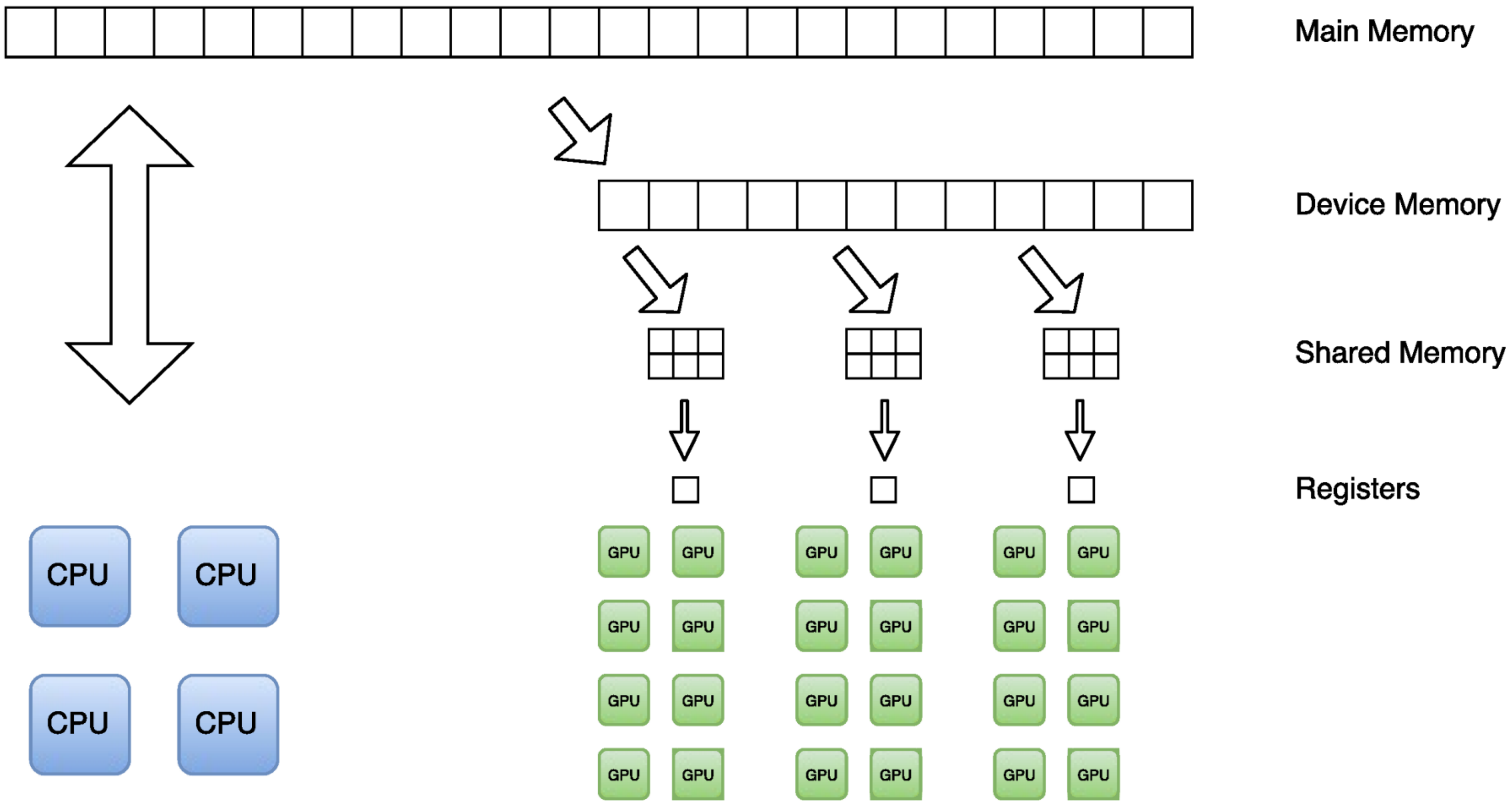


$$BID = \{(i,j) \to \left(\left\lfloor \frac{i}{4} \right\rfloor \% \, 2, \left\lfloor \frac{j}{3} \right\rfloor \% \, 2\right)\}$$

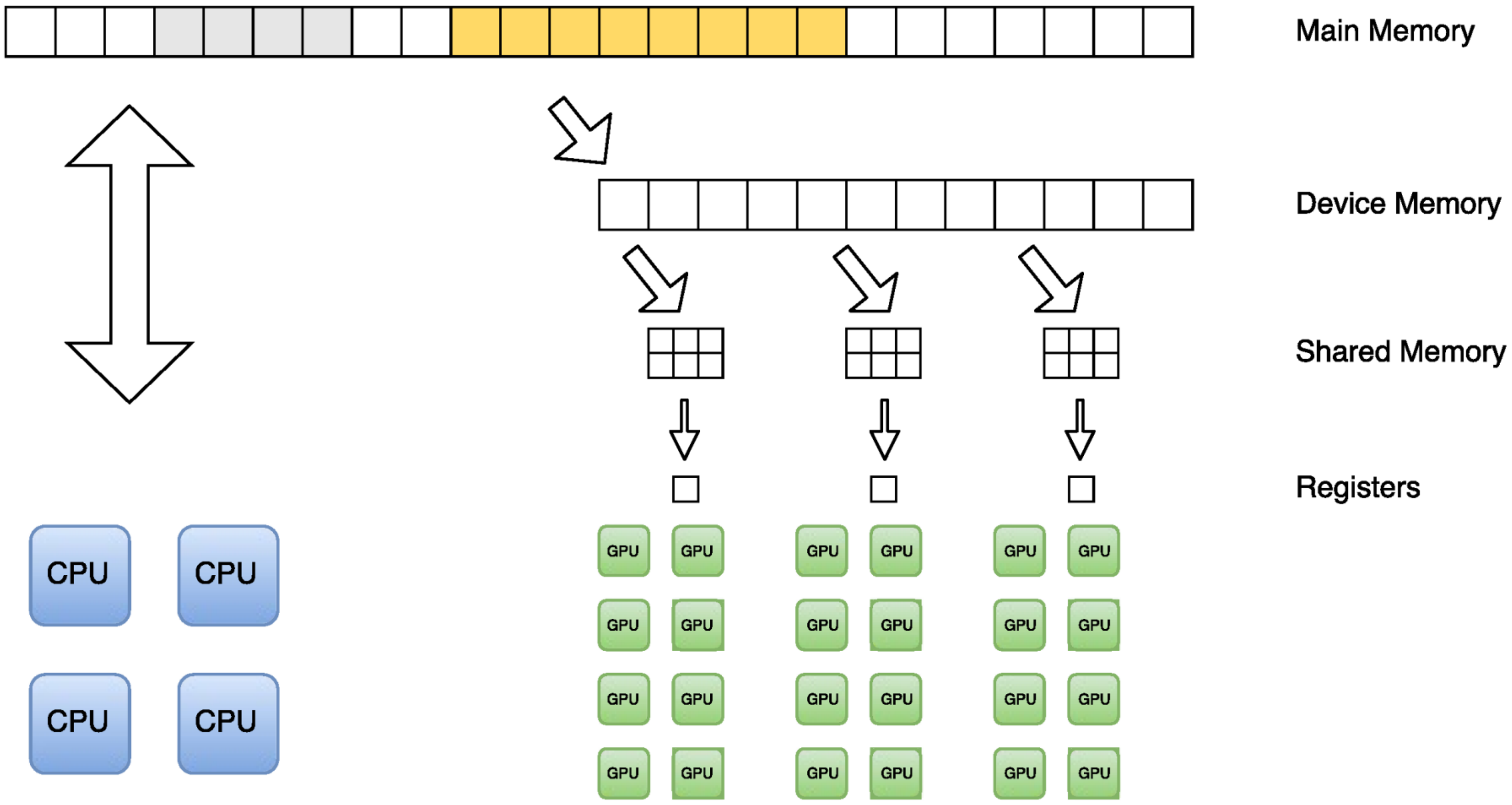
$$TID = \{(i,j) \to (i \,\%\, 4, j \,\%\, 3)\}$$

T. Grosser, TH: Polly-ACC: Transparent compilation to heterogeneous hardware, ACM ICS'16

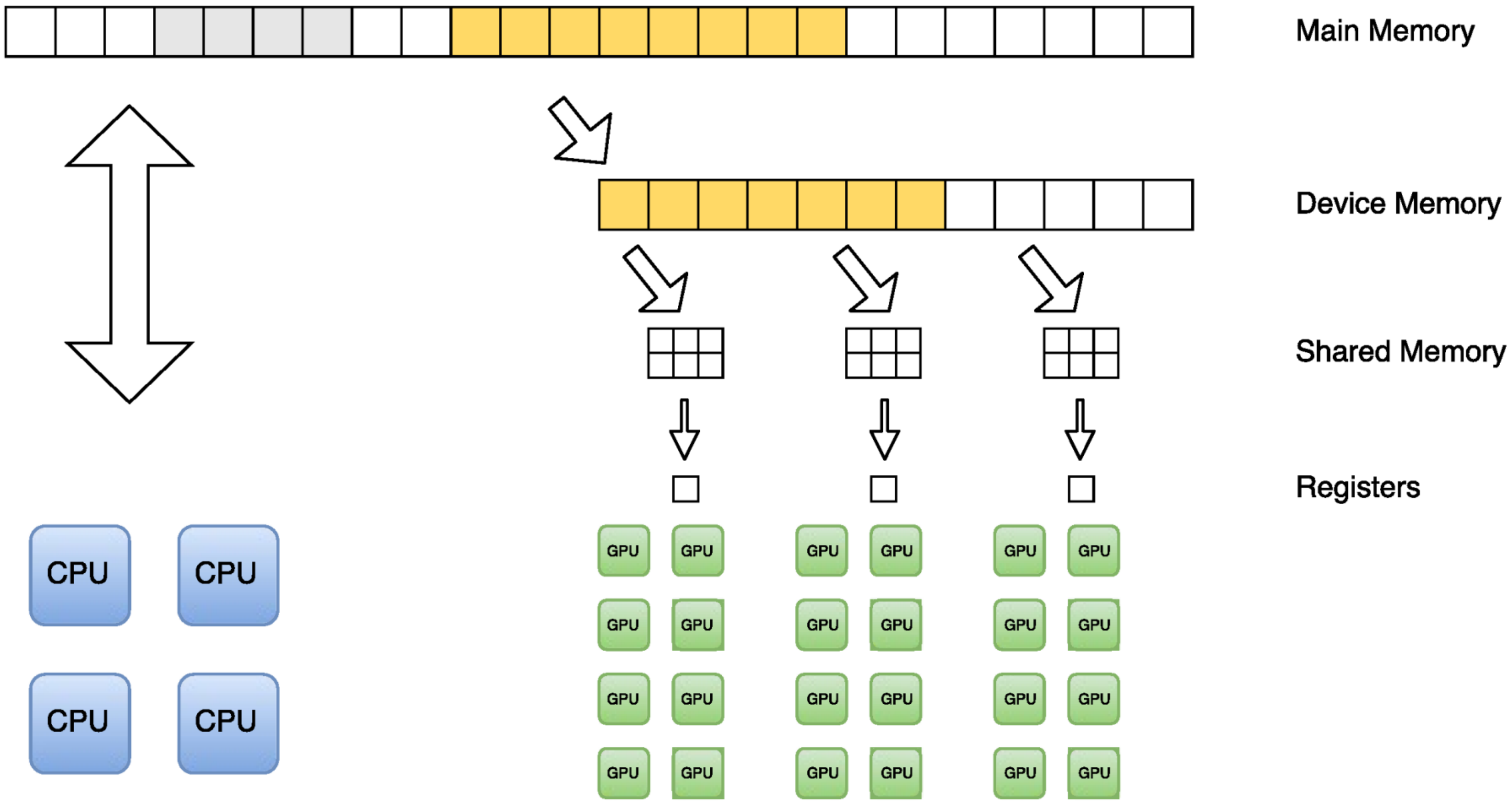
Memory Hierarchy of a Heterogeneous System



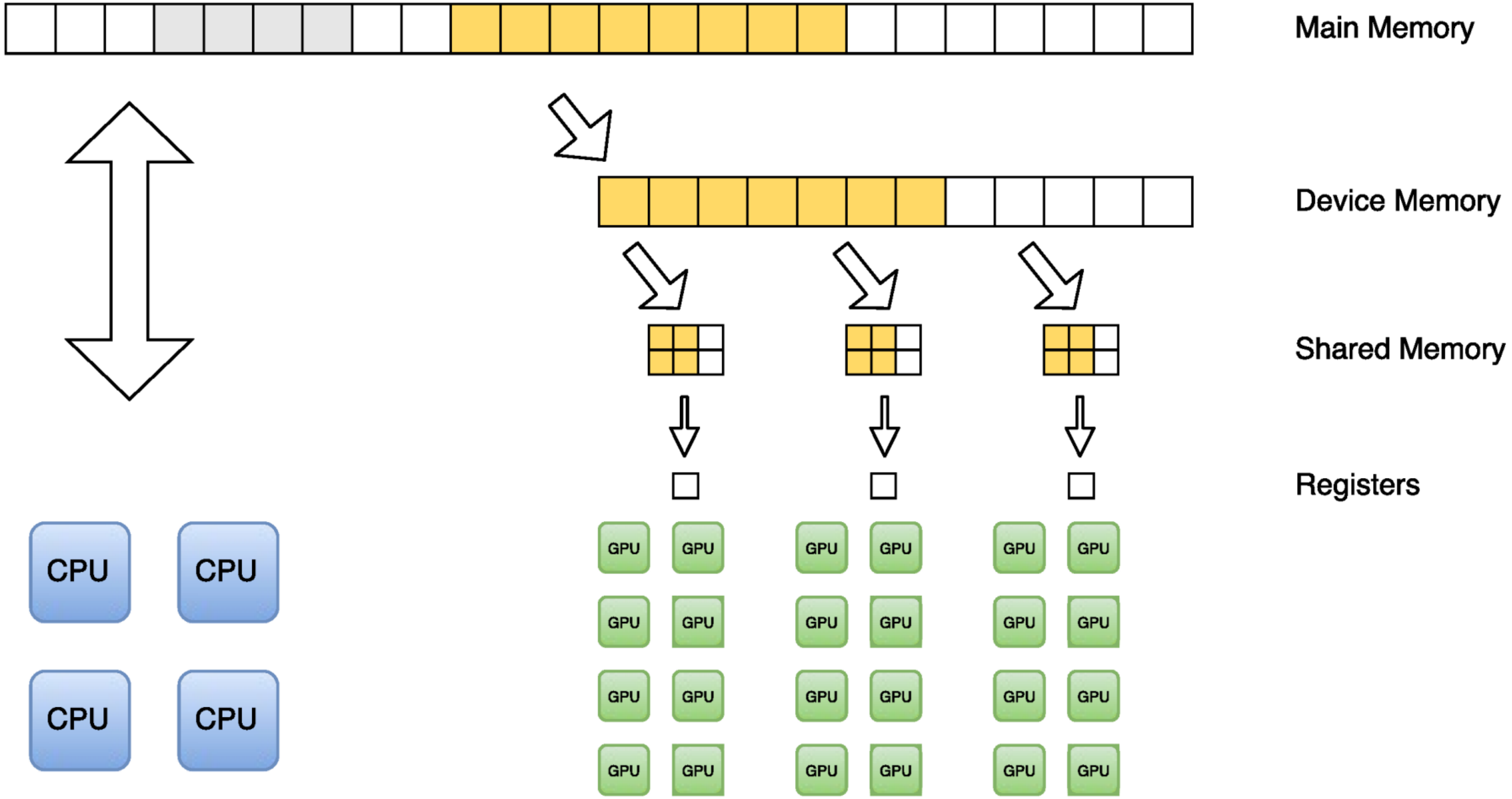
Host-device date transfers



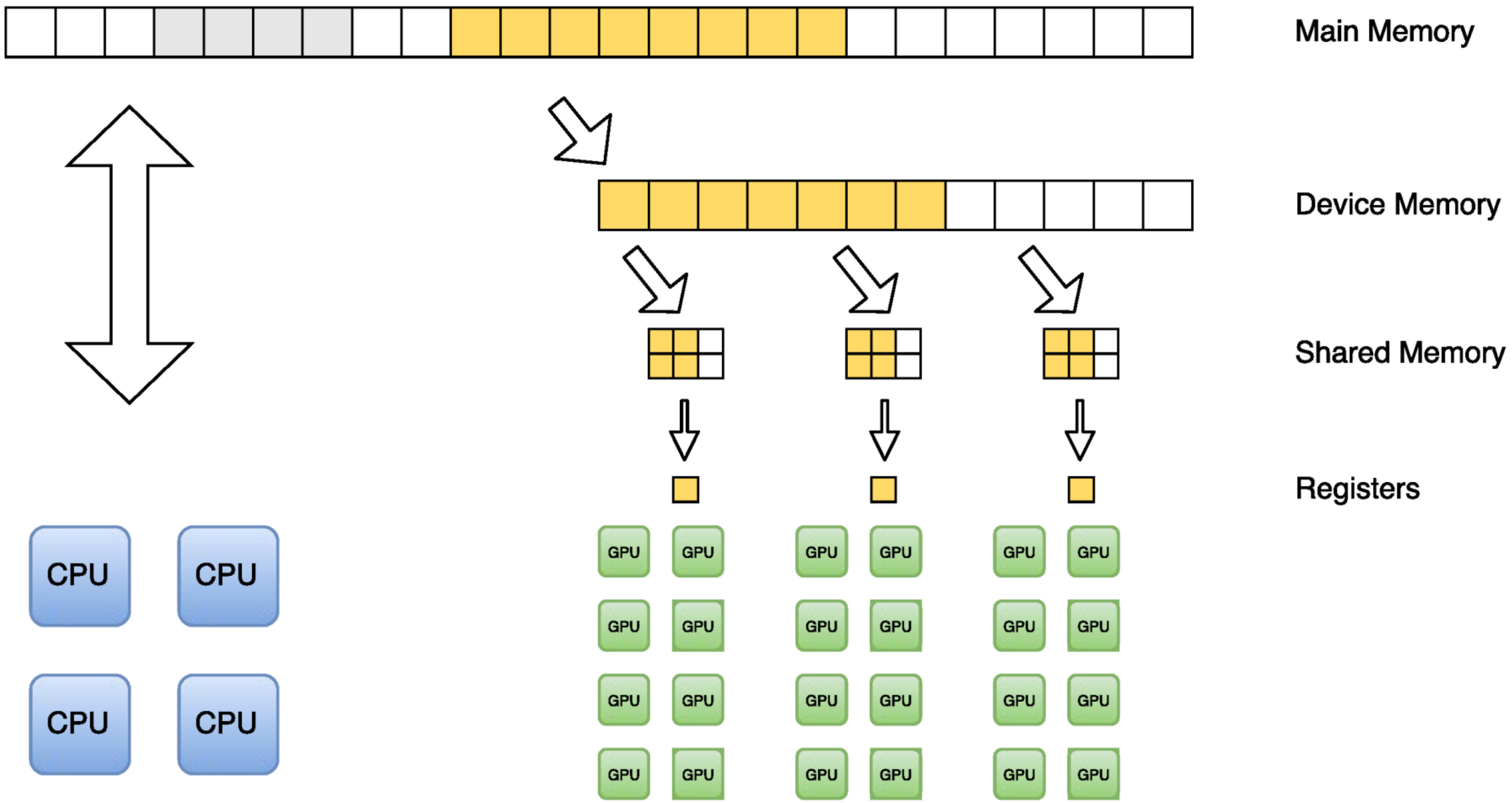
Host-device date transfers



Mapping onto fast memory



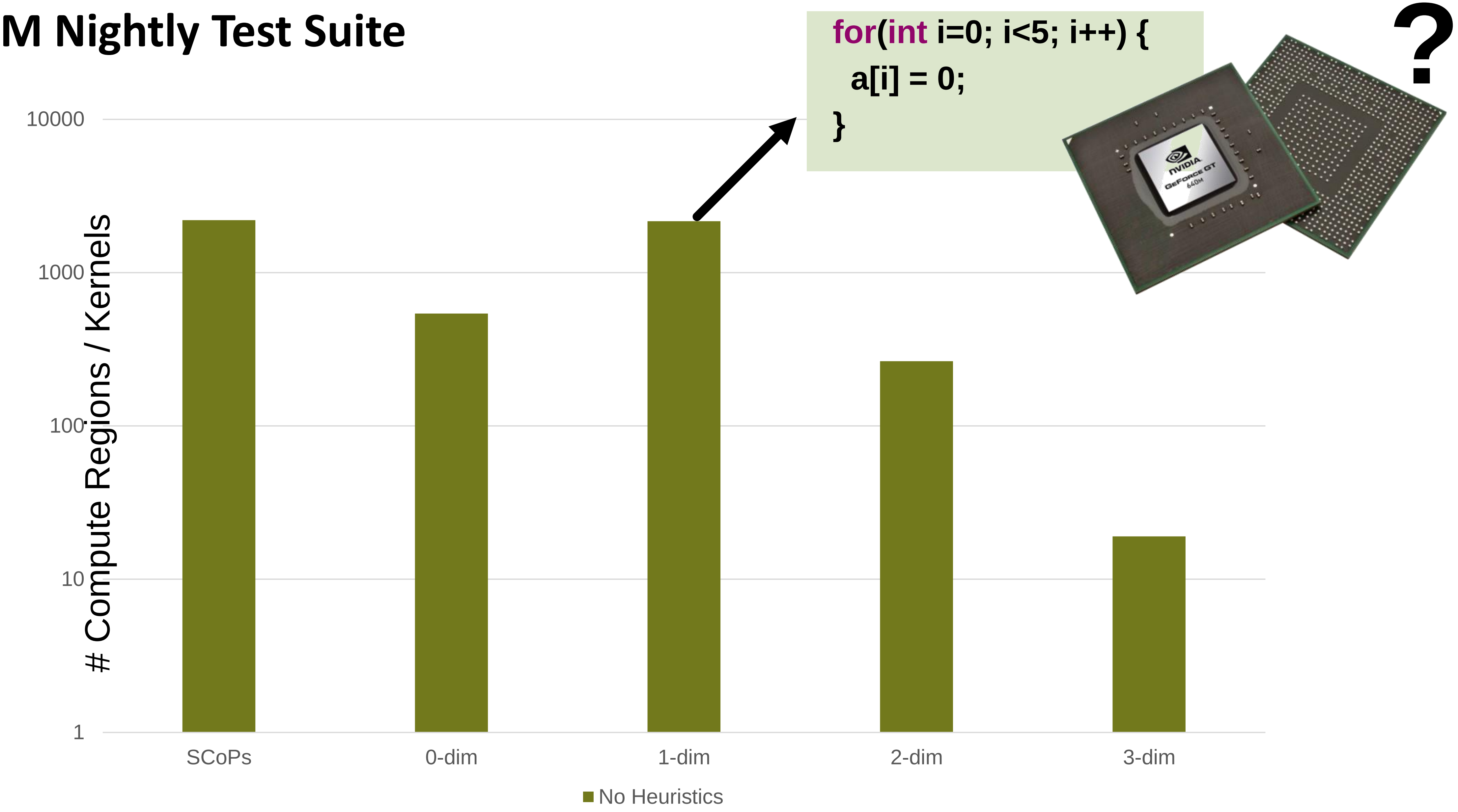
Mapping onto fast memory



Verdoolaege, Sven et al.: Polyhedral parallel code generation for CUDA, ACM Transactions on Architecture and Code Optimization, 2013

Practice

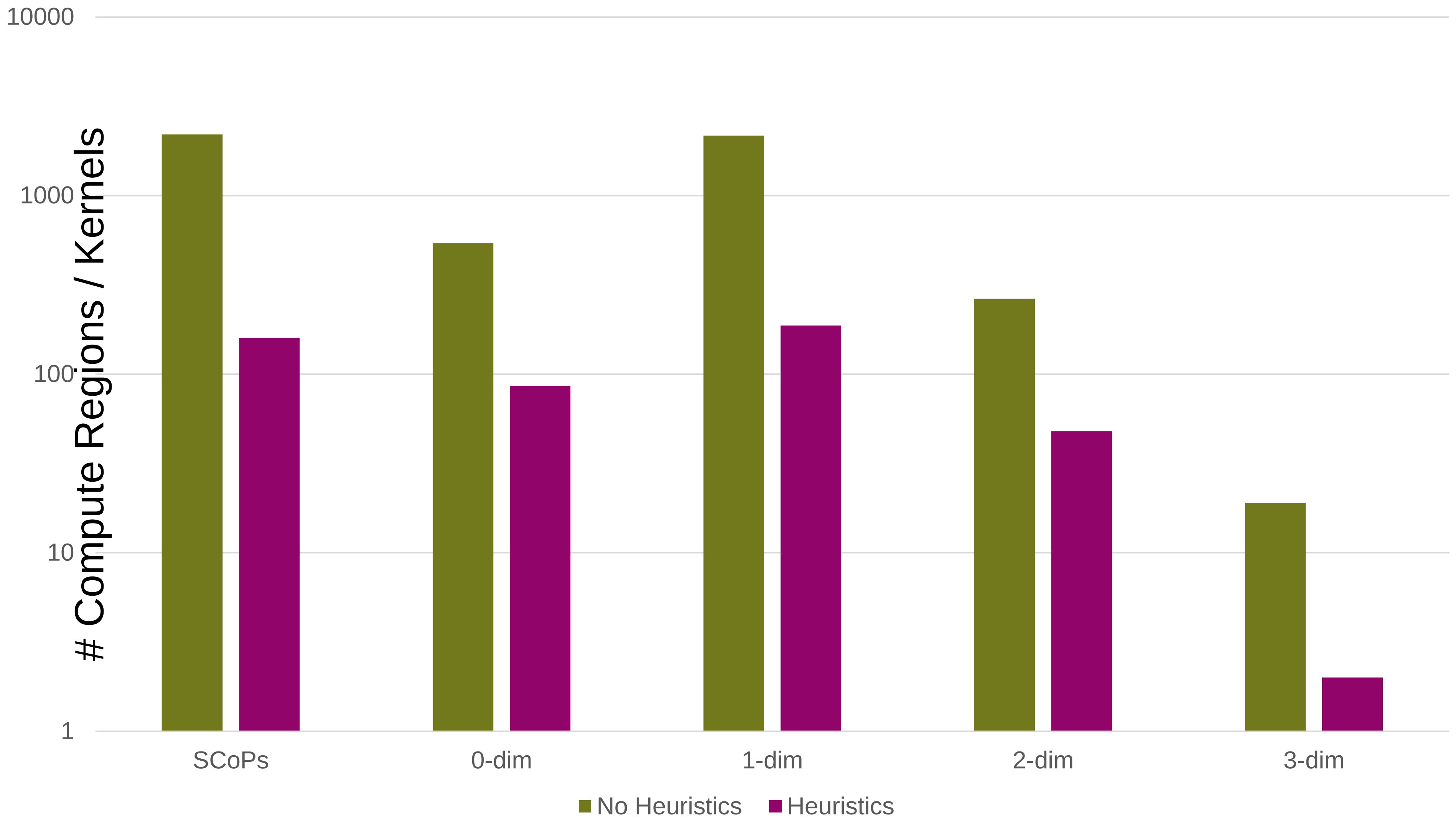
LLVM Nightly Test Suite



```
for(int i=0; i<5; i++) {
  a[i] = 0;
}
```

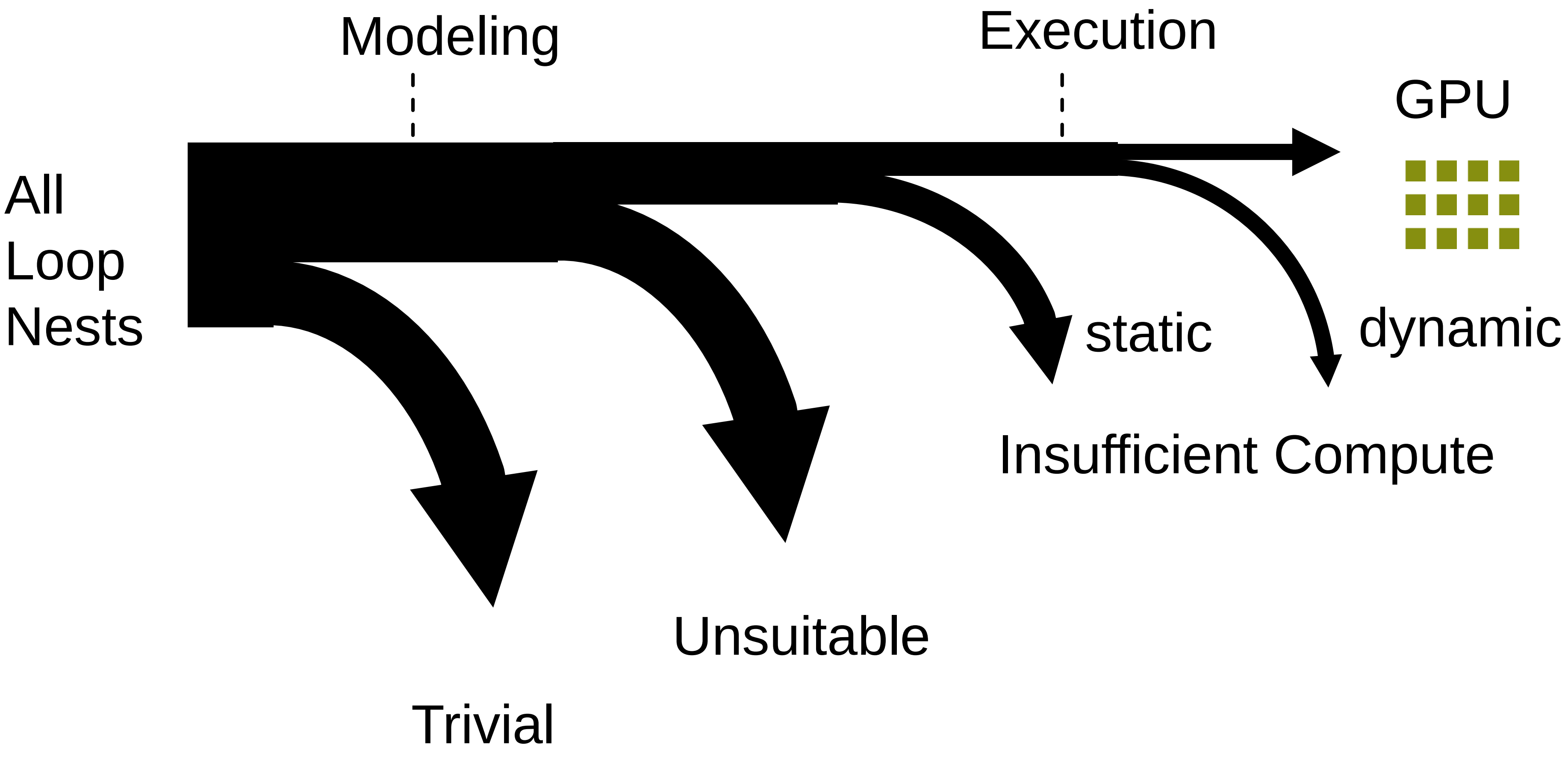
T. Grosser, TH: Polly-ACC: Transparent compilation to heterogeneous hardware, ACM ICS'16

LLVM Nightly Test Suite



T. Grosser, TH: Polly-ACC: Transparent compilation to heterogeneous hardware, ACM ICS'16

Profitability Heuristic



T. Grosser, TH: Polly-ACC: Transparent compilation to heterogeneous hardware, ACM ICS'16

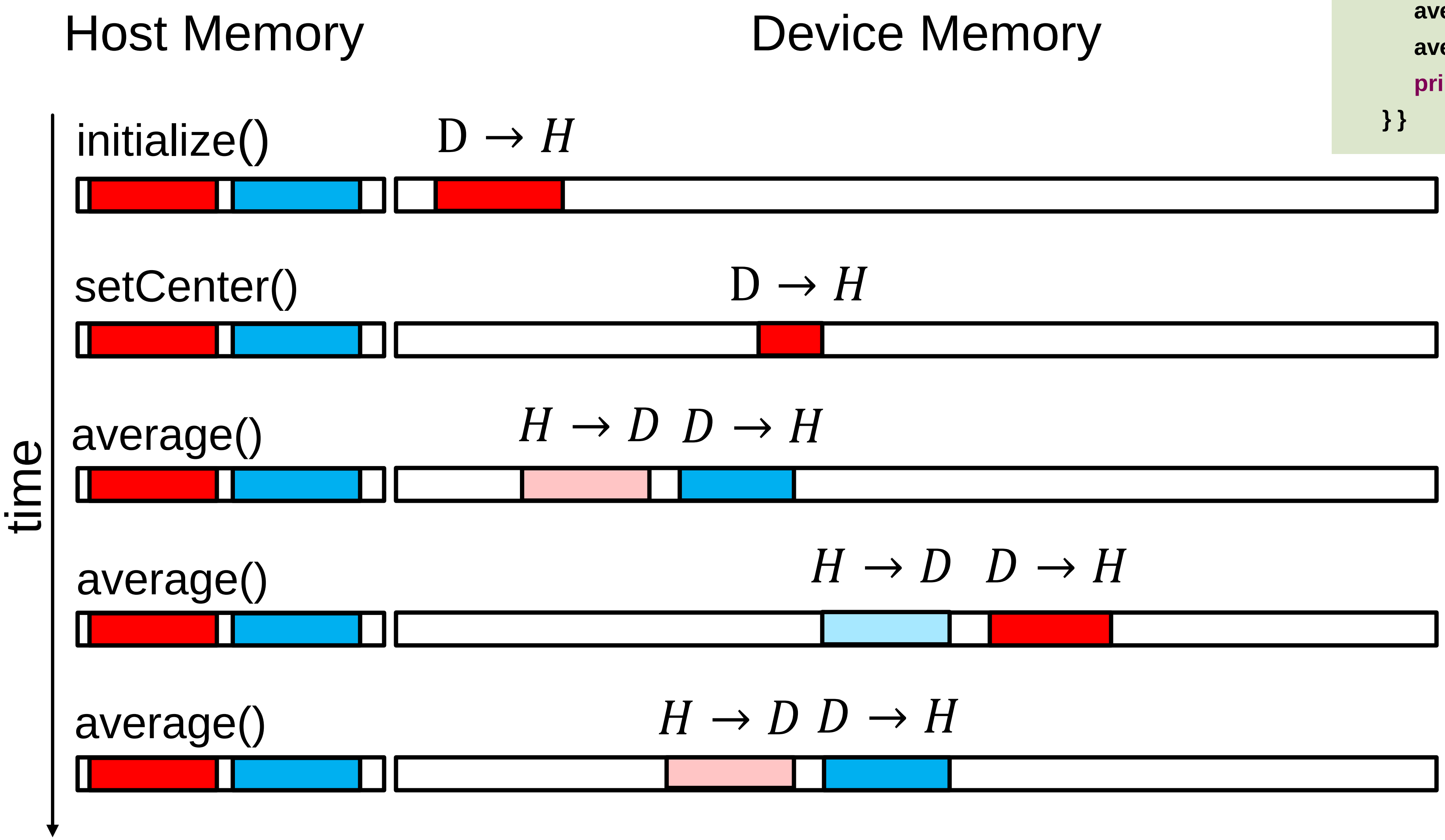
From kernels to program – data transfers

```
void heat(int n, float A[n], float hot, float cold) {

    float B[n] = {0};

    initialize(n, A, cold);
    setCenter(n, A, hot, n/4);

    for (int t = 0; t < T; t++) {
        average(n, A, B);
        average(n, B, A);
        printf("Iteration %d done", t);
    }
}
```

T. Grosser, TH: Polly-ACC: Transparent compilation to heterogeneous hardware, ACM ICS'16

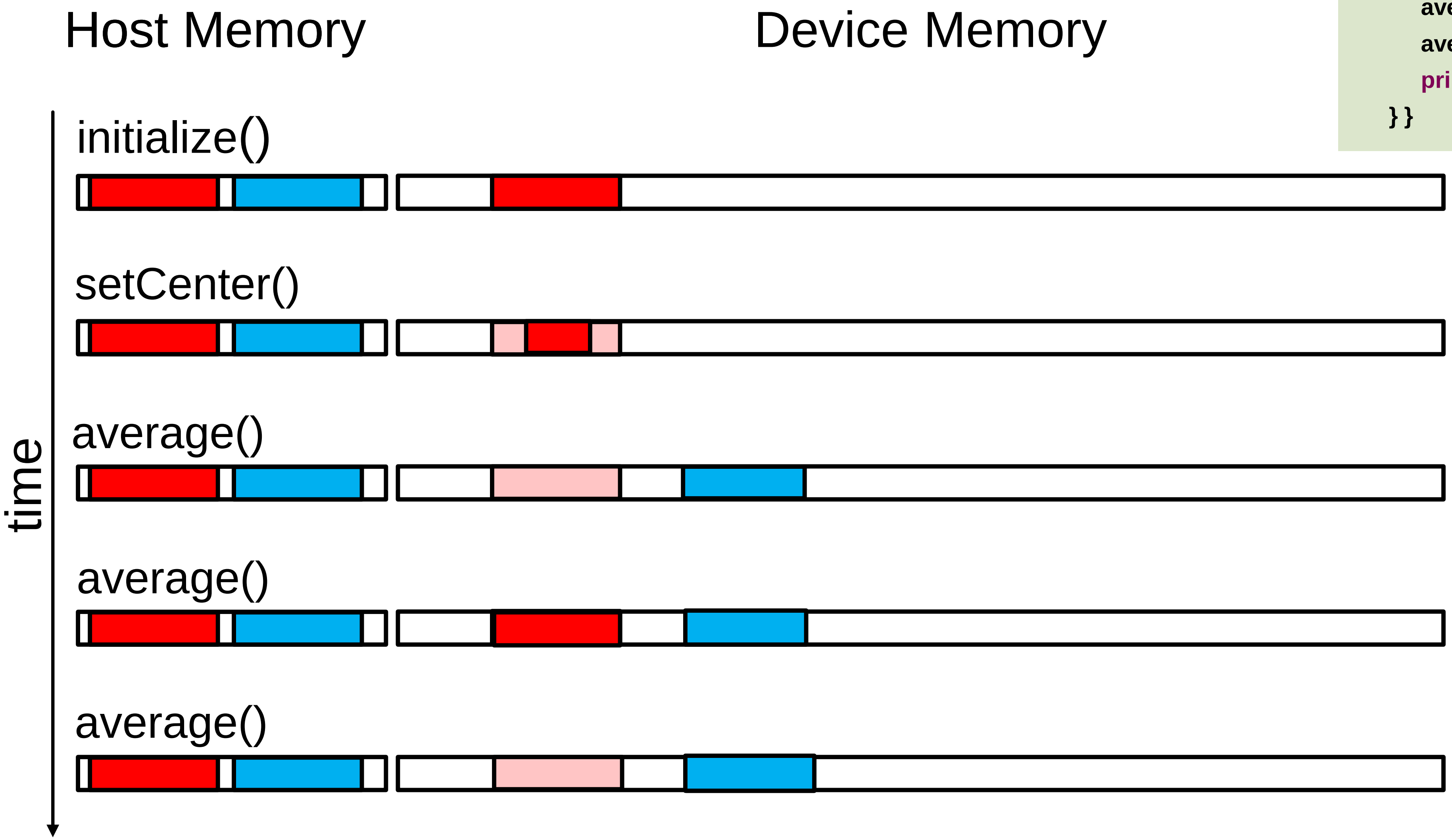
Data Transfer – Per Kernel

```
void heat(int n, float A[n], ...) {
  initialize(n, A, cold);
  setCenter(n, A, hot, n/4);
  for (int t = 0; t < T; t++) {
    average(n, A, B);
    average(n, B, A);
    printf("Iteration %d done", t);
} }
```



Data Transfer – Inter Kernel Caching

```
void heat(int n, float A[n], ...) {
   initialize(n, A, cold);
   setCenter(n, A, hot, n/4);
   for (int t = 0; t < T; t++) {
      average(n, A, B);
      average(n, B, A);
      printf("Iteration %d done", t);
  } }
```



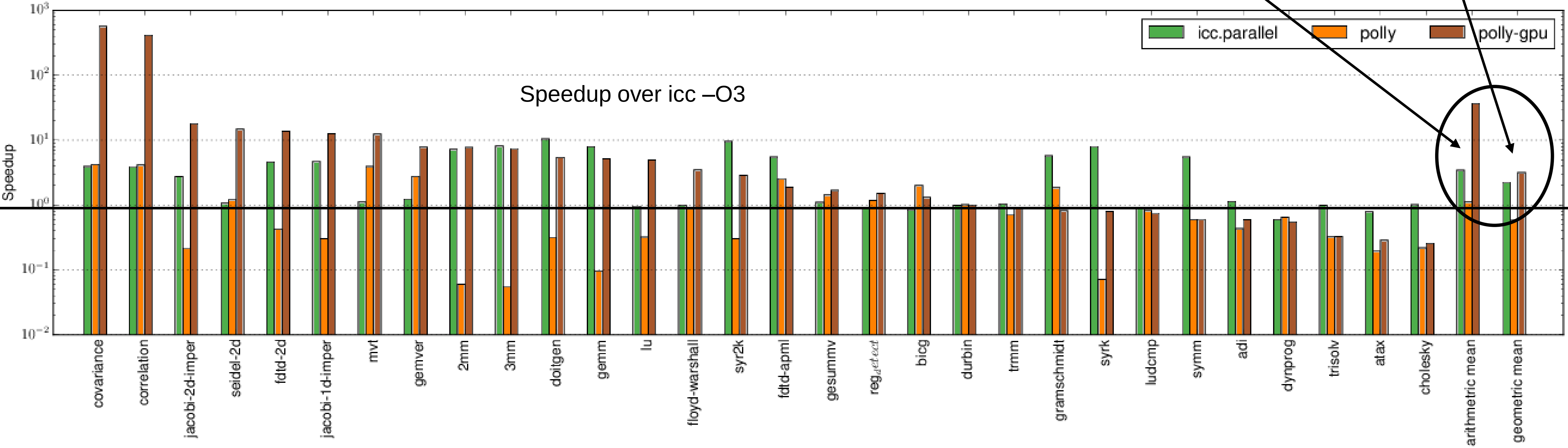
Evaluation

Workstation:	10 core SandyBridge	NVIDIA Titan Black (Kepler)
Mobile:	4 core Haswell	NVIDIA GT730M (Kepler)

Some results: Polybench 3.2

geomean: ~6x

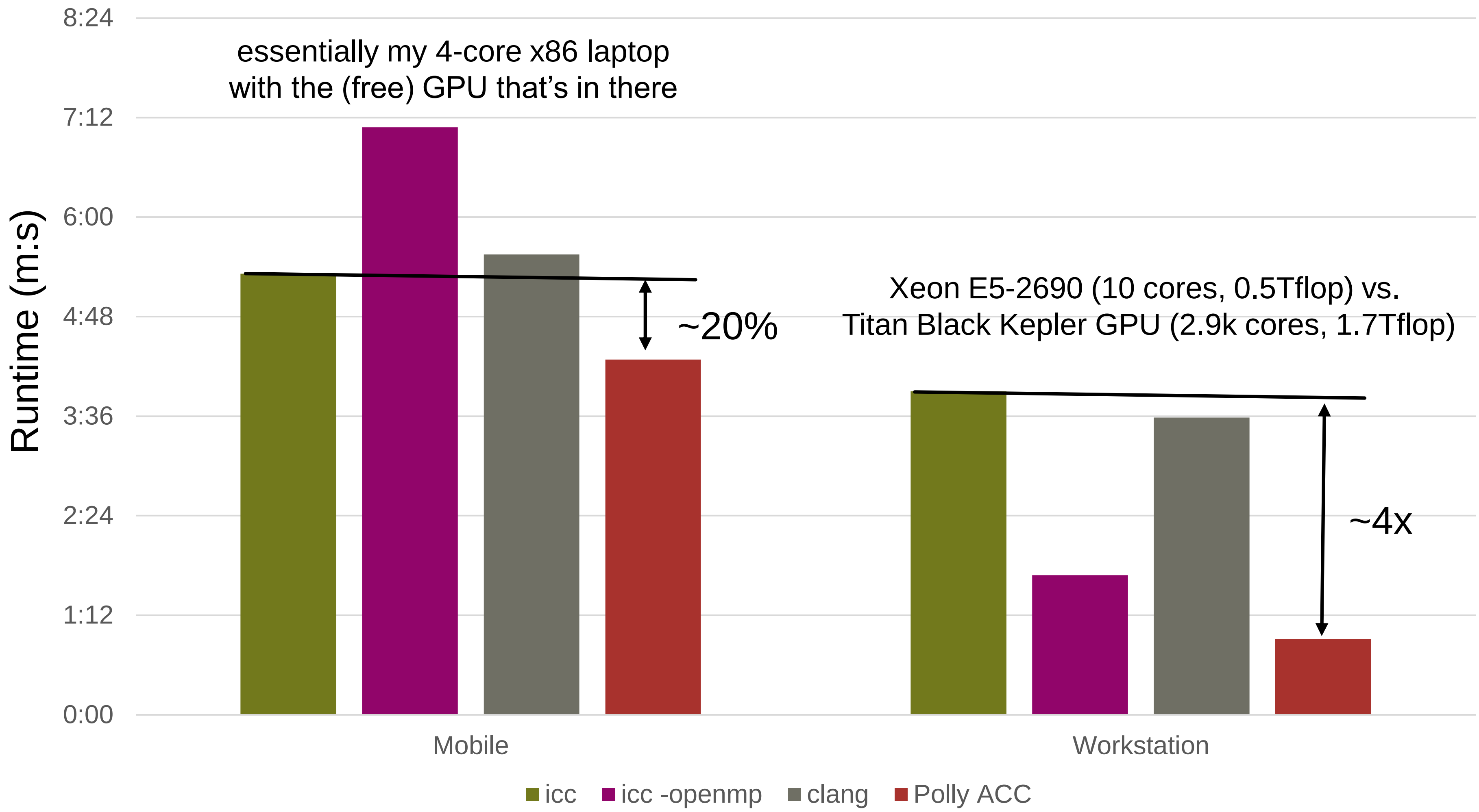
arithmean: ~30x



Xeon E5-2690 (10 cores, 0.5Tflop) vs. Titan Black Kepler GPU (2.9k cores, 1.7Tflop)

T. Grosser, TH: Polly-ACC: Transparent compilation to heterogeneous hardware, ACM ICS'16

Compiles all of SPEC CPU 2006 – Example: LBM

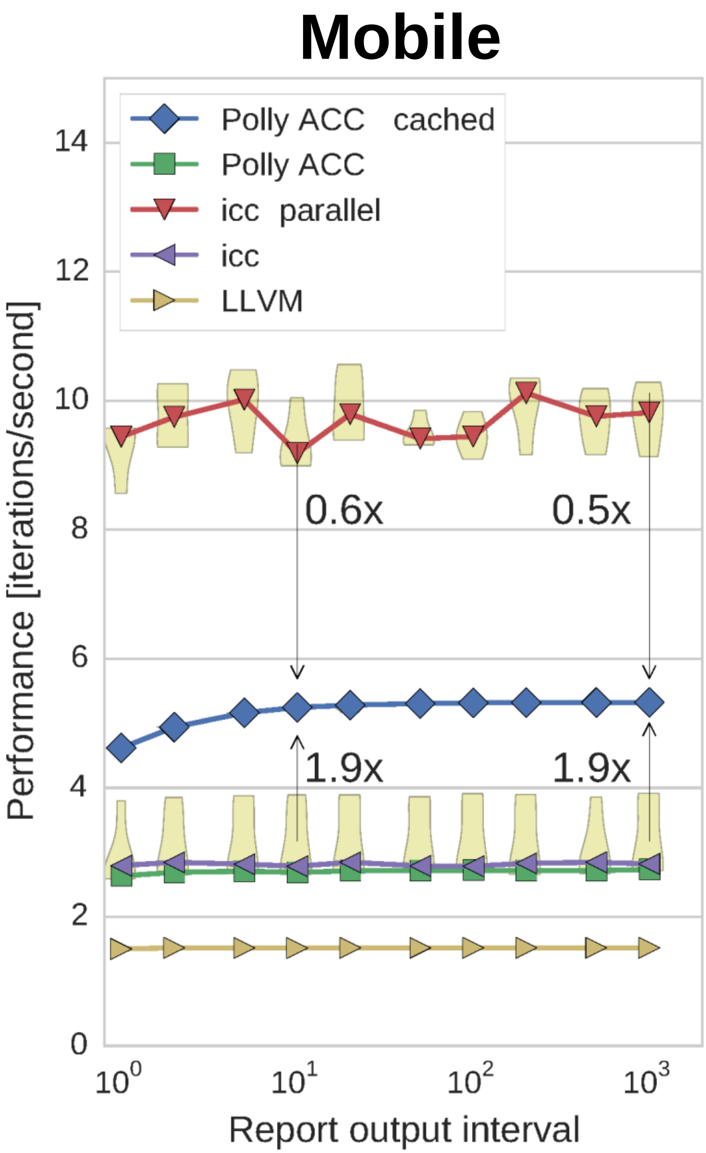


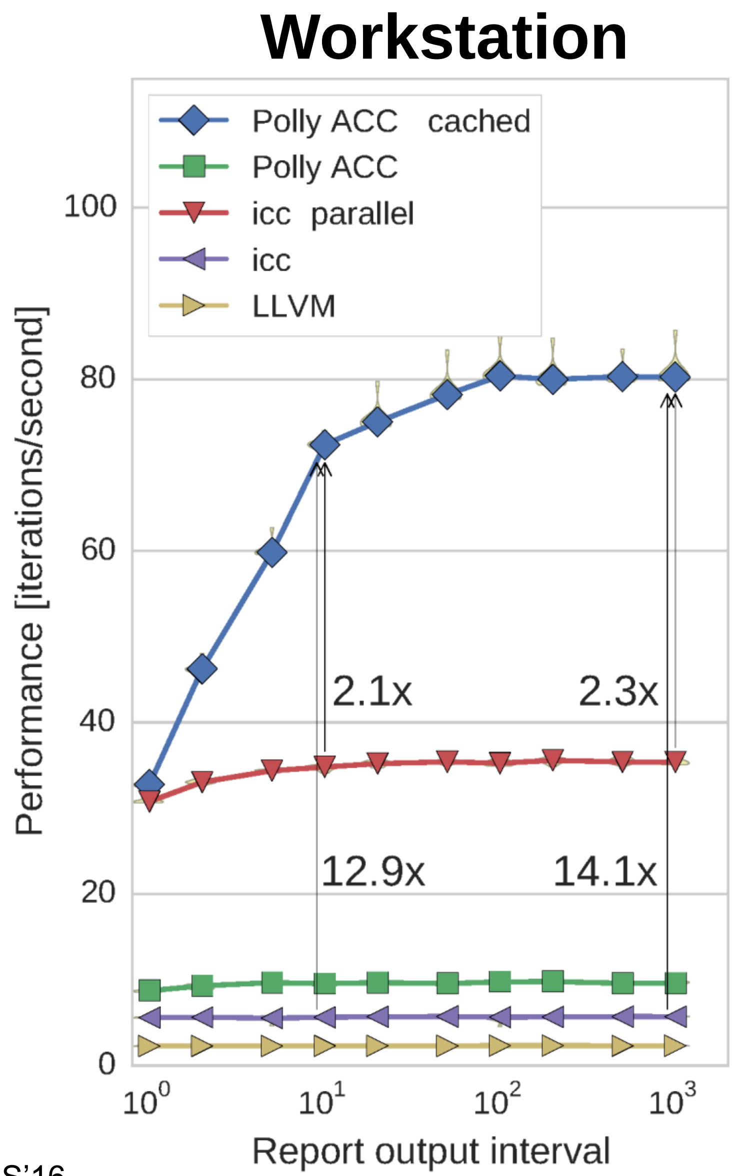
essentially my 4-core x86 laptop
with the (free) GPU that's in there

Xeon E5-2690 (10 cores, 0.5Tflop) vs.
Titan Black Kepler GPU (2.9k cores, 1.7Tflop)

T. Grosser, TH: Polly-ACC: Transparent compilation to heterogeneous hardware, ACM ICS'16

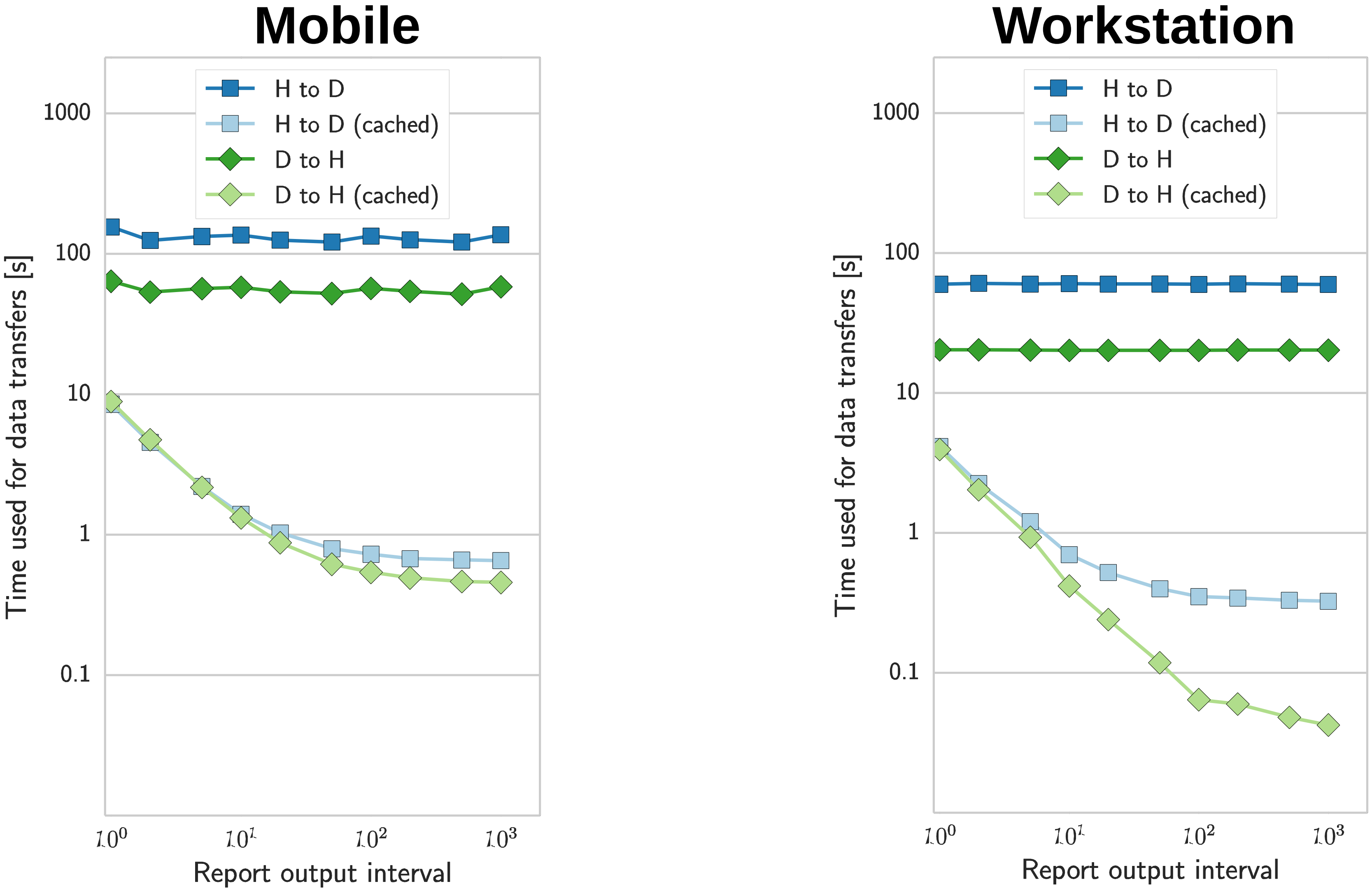
Cactus ADM (SPEC 2006)





T. Grosser, TH: Polly-ACC: Transparent compilation to heterogeneous hardware, ACM ICS'16

Cactus ADM (SPEC 2006) - Data Transfer

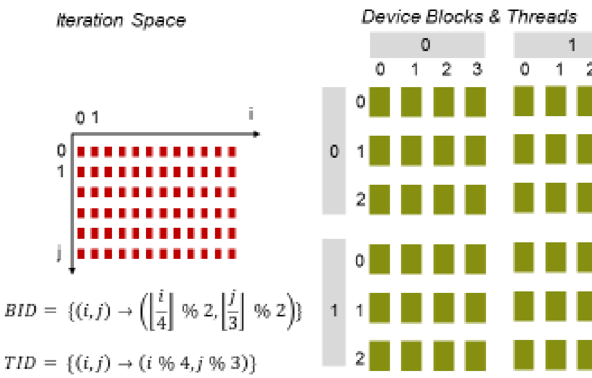


T. Grosser, TH: Polly-ACC: Transparent compilation to heterogeneous hardware, ACM ICS'16

Polly-ACC

http://spcl.inf.ethz.ch/Polly-ACC

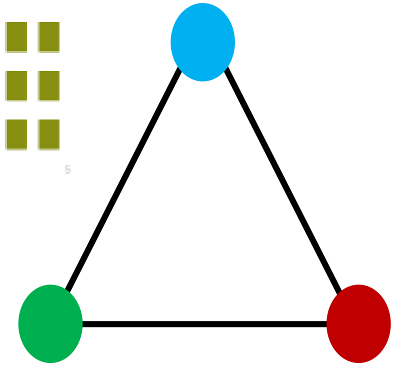
Mapping Computation to Device



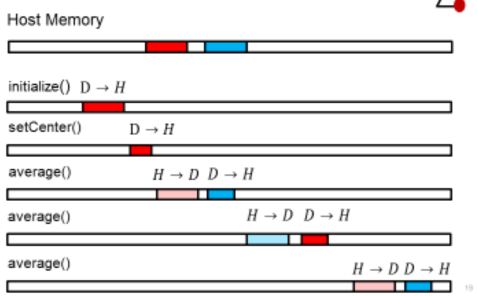
$BID = \{(i,j) \to (\lfloor \frac{i}{4} \rfloor \% 2, \lfloor \frac{j}{3} \rfloor \% 2)\}$

$TID = \{(i,j) \to (i \% 4, j \% 3)\}$

Automatic



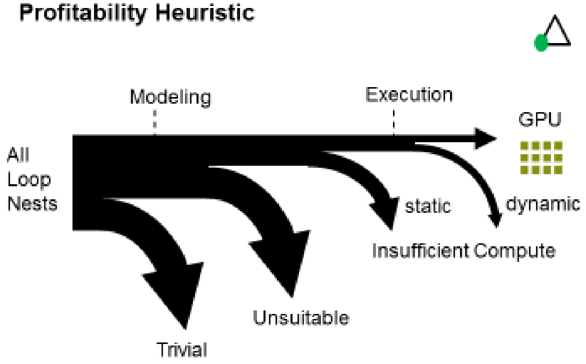
Data Transfer – Per Kernel

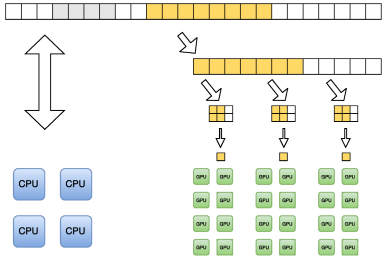


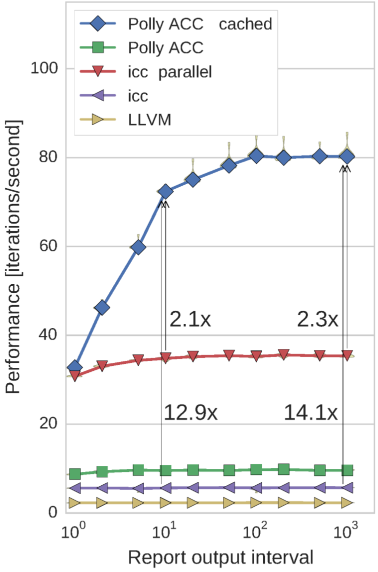
“Regression Free”

High Performance

Profitability Heuristic







T. Grosser, TH: Polly-ACC: Transparent compilation to heterogeneous hardware, ACM ICS’16

Brave new/old compiler world!?

- Unfortunately not …
  - Limited to affine code regions
  - Maybe generalizes to oblivious (data-independent control) programs
  - No distributed anything!!
- Good news:
  - Much of traditional HPC fits that model
  - Infrastructure is coming along

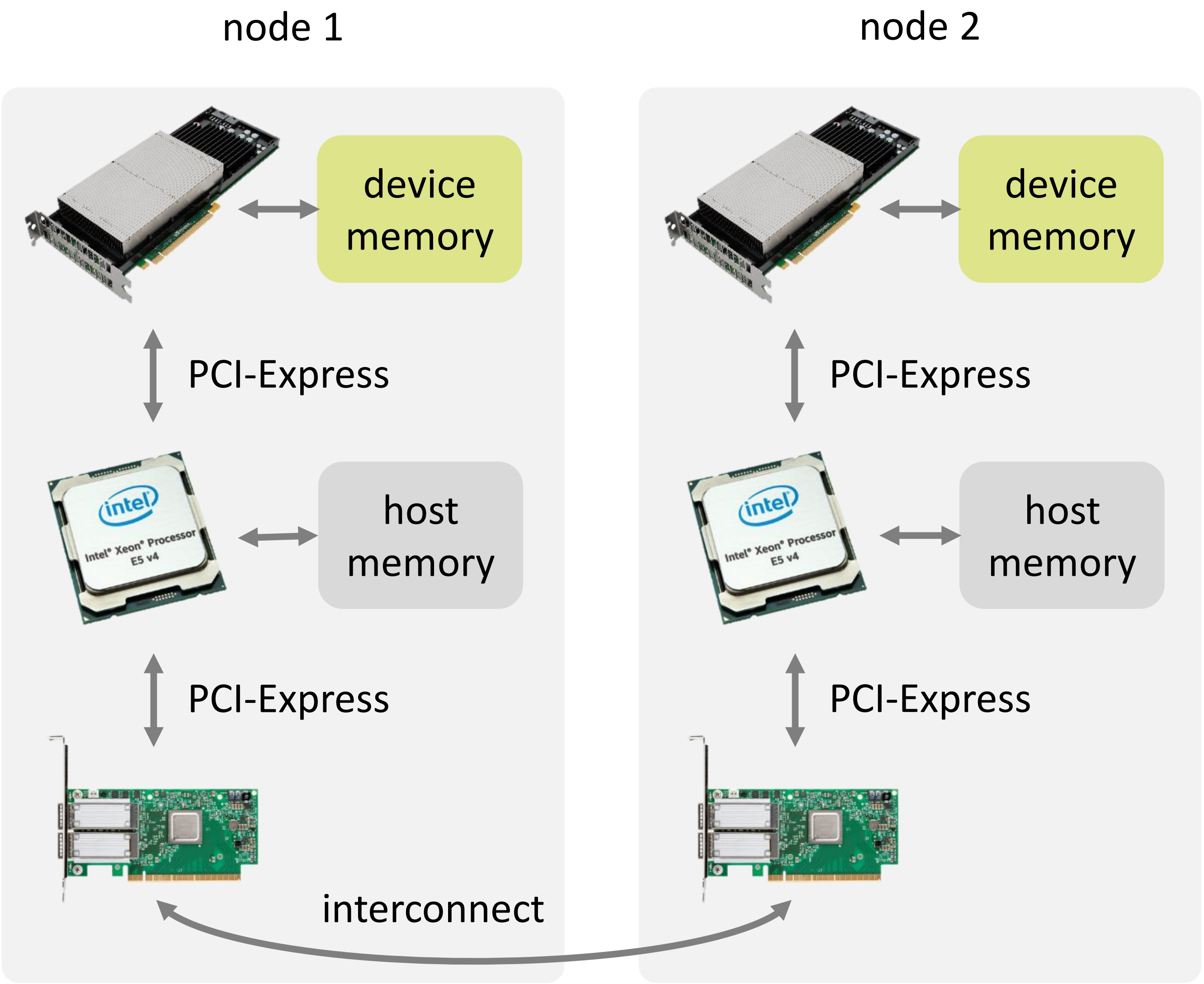
LLVM COMPILER INFRASTRUCTURE

pasc Platform for Advanced Scientific Computing

- Bad news:
  - Modern data-driven HPC and Big Data fits less well
  - Need a programming model for **distributed** heterogeneous machines!

Distributed GPU Computing

GPU cluster programming using MPI and CUDA



code

```
// run compute kernel
__global__
void mykernel( … ) { }
```

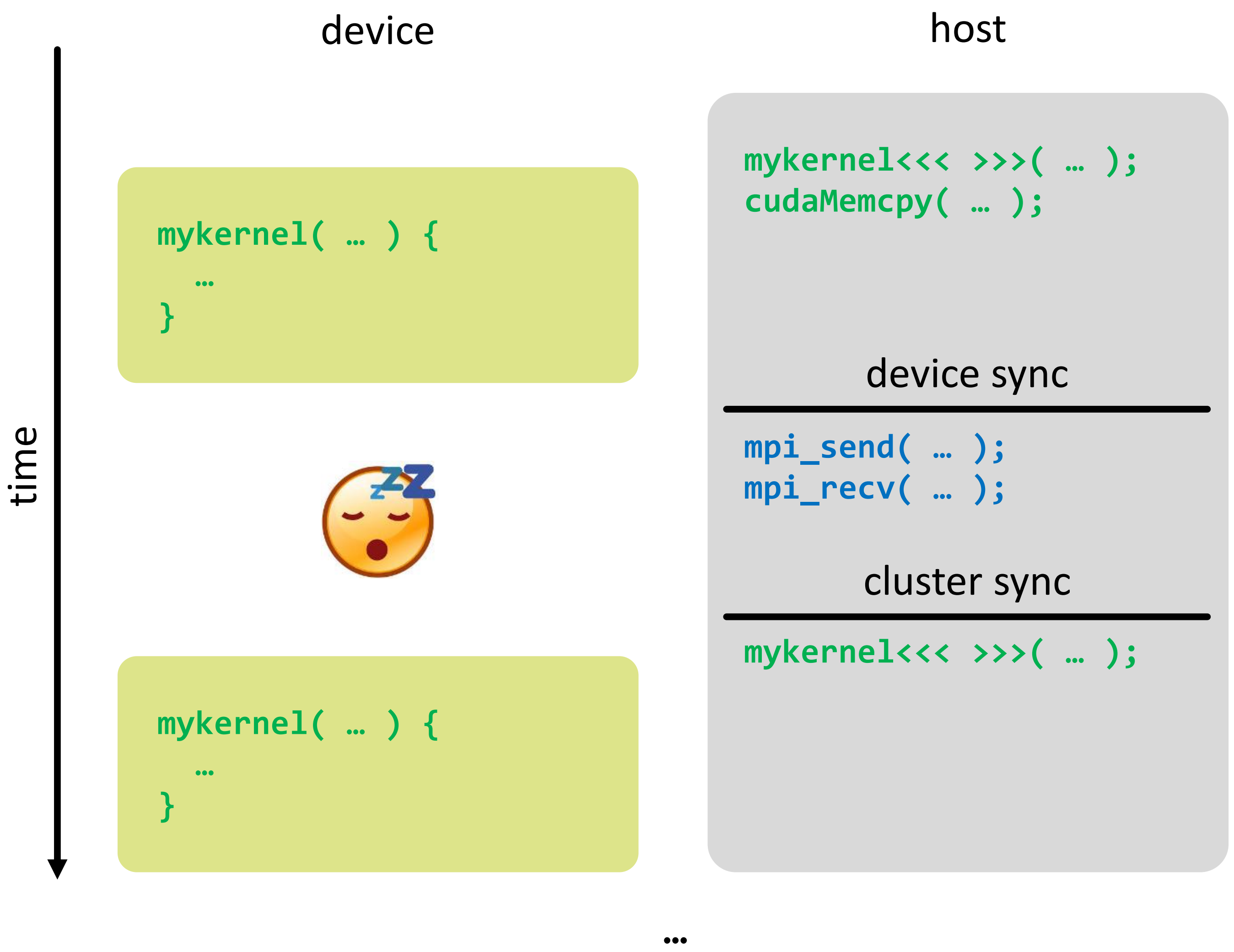
```
// launch compute kernel
mykernel<<<64,128>>>( … );

// on-node data movement
cudaMemcpy(
  psize, &size,
  sizeof(int),
  cudaMemcpyDeviceToHost);

// inter-node data movement
mpi_send(
  pdata, size,
  MPI_FLOAT, … );
mpi_recv(
  pdata, size,
  MPI_FLOAT, … );
```

NVIDIA CUDA

NVIDIA CUDA

MPI

Disadvantages of the MPI-CUDA approach



```
mykernel( … ) {
  …
}
```

```
mykernel<<< >>>( … );
cudaMemcpy( … );
```

```
mpi_send( … );
mpi_recv( … );
```

```
mykernel<<< >>>( … );
```

```
mykernel( … ) {
  …
}
```

complexity

- two programming models
- duplicated functionality

copy

sync

performance

- encourages sequential execution
- low utilization of the costly hardware

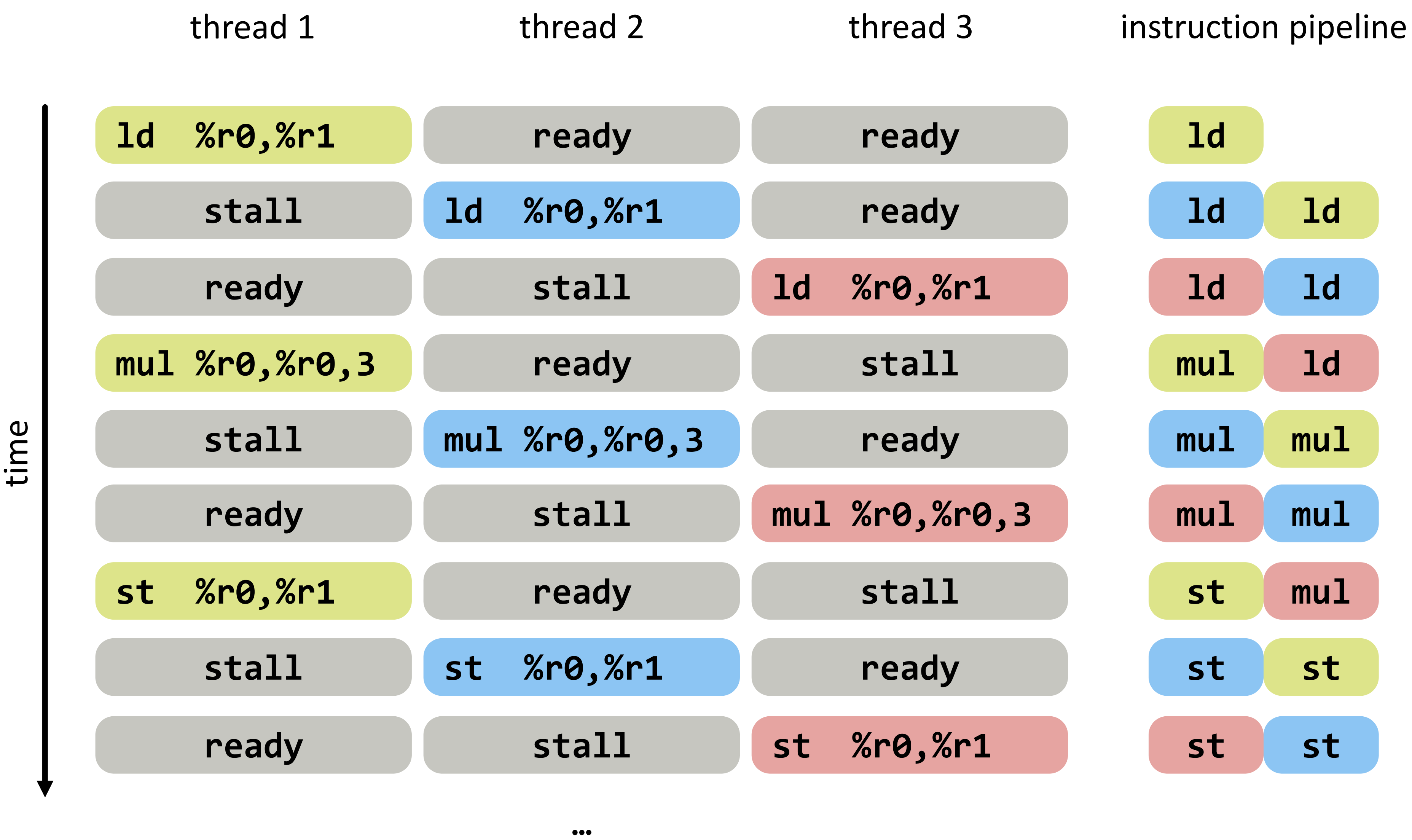
T. Gysi, J. Baer, TH: dCUDA: Hardware Supported Overlap of Computation and Communication, SC16

Achieve high resource utilization using oversubscription & hardware threads

code

```
ld  %r0,%r1
mul %r0,%r0,3
st  %r0,%r1
```

GPU cores use “parallel slack” to hide instruction pipeline latencies

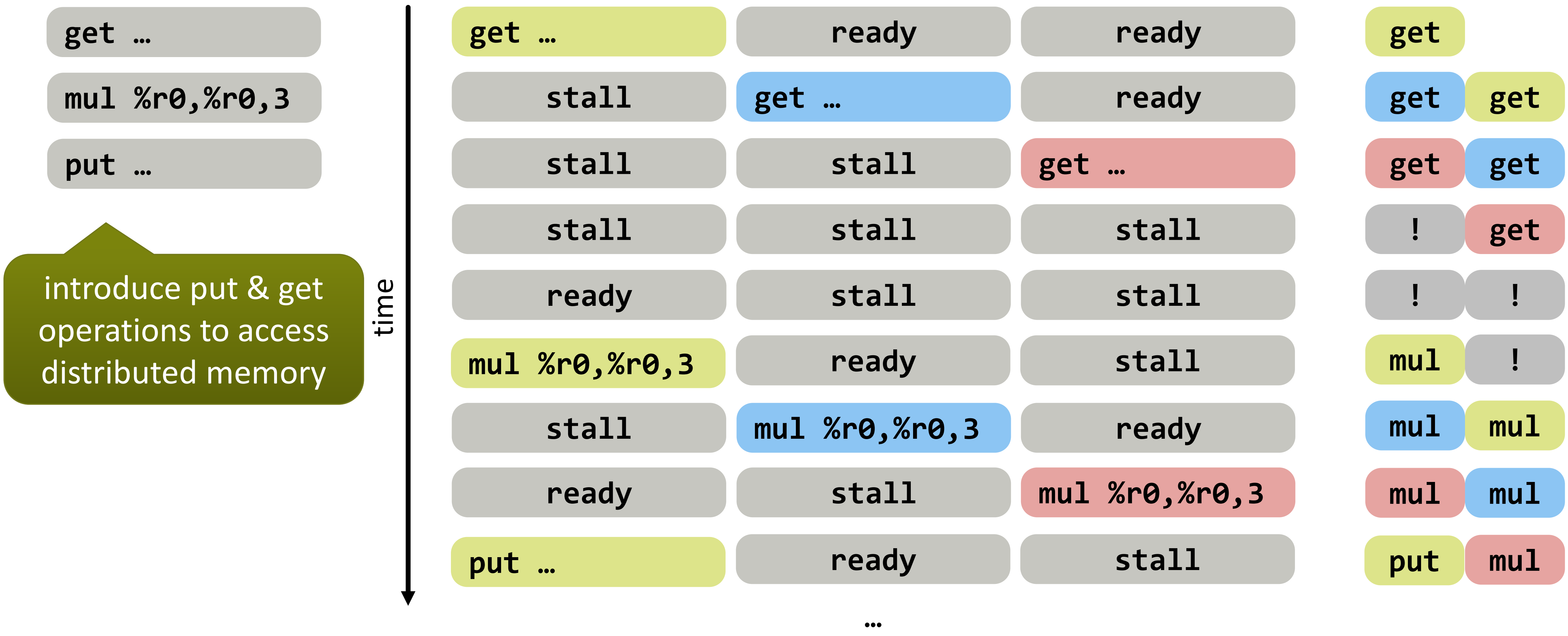


thread 1	thread 2	thread 3	instruction pipeline	
`ld  %r0,%r1`	ready	ready	ld	
stall	`ld  %r0,%r1`	ready	ld	ld
ready	stall	`ld  %r0,%r1`	ld	ld
`mul %r0,%r0,3`	ready	stall	mul	ld
stall	`mul %r0,%r0,3`	ready	mul	mul
ready	stall	`mul %r0,%r0,3`	mul	mul
`st  %r0,%r1`	ready	stall	st	mul
stall	`st  %r0,%r1`	ready	st	st
ready	stall	`st  %r0,%r1`	st	st

…

T. Gysi, J. Baer, TH: dCUDA: Hardware Supported Overlap of Computation and Communication, SC16

Use oversubscription & hardware threads to hide remote memory latencies



code	thread 1	thread 2	thread 3	instruction pipeline	
get …	get …	ready	ready	get	
mul %r0,%r0,3	stall	get …	ready	get	get
put …	stall	stall	get …	get	get
	stall	stall	stall	!	get
	ready	stall	stall	!	!
	mul %r0,%r0,3	ready	stall	mul	!
	stall	mul %r0,%r0,3	ready	mul	mul
	ready	stall	mul %r0,%r0,3	mul	mul
	put …	ready	stall	put	mul

introduce put & get operations to access distributed memory

…

T. Gysi, J. Baer, TH: dCUDA: Hardware Supported Overlap of Computation and Communication, SC16

How much "parallel slack" is necessary to fully utilize the interconnect?

Little's law

$$concurrency = latency \; * throughput$$

	device memory
latency	1µs
bandwidth	200GB/s
concurrency	200kB
#threads	~12000

>>

T. Gysi, J. Baer, TH: dCUDA: Hardware Supported Overlap of Computation and Communication, SC16

dCUDA (distributed CUDA) extends CUDA with MPI-3 RMA and notifications

```
for (int i = 0; i < steps; ++i) {
  for (int idx = from; idx < to; idx += jstride)
    out[idx] = -4.0 * in[idx] +
      in[idx + 1] + in[idx - 1] +
      in[idx + jstride] + in[idx - jstride];

  if (lsend)
    dcuda_put_notify(ctx, wout, rank - 1,
      len + jstride, jstride, &out[jstride], tag);
  if (rsend)
    dcuda_put_notify(ctx, wout, rank + 1,
      0, jstride, &out[len], tag);

  dcuda_wait_notifications(ctx, wout,
    DCUDA_ANY_SOURCE, tag, lsend + rsend);

  swap(in, out);
  swap(win, wout);
}
```

computation

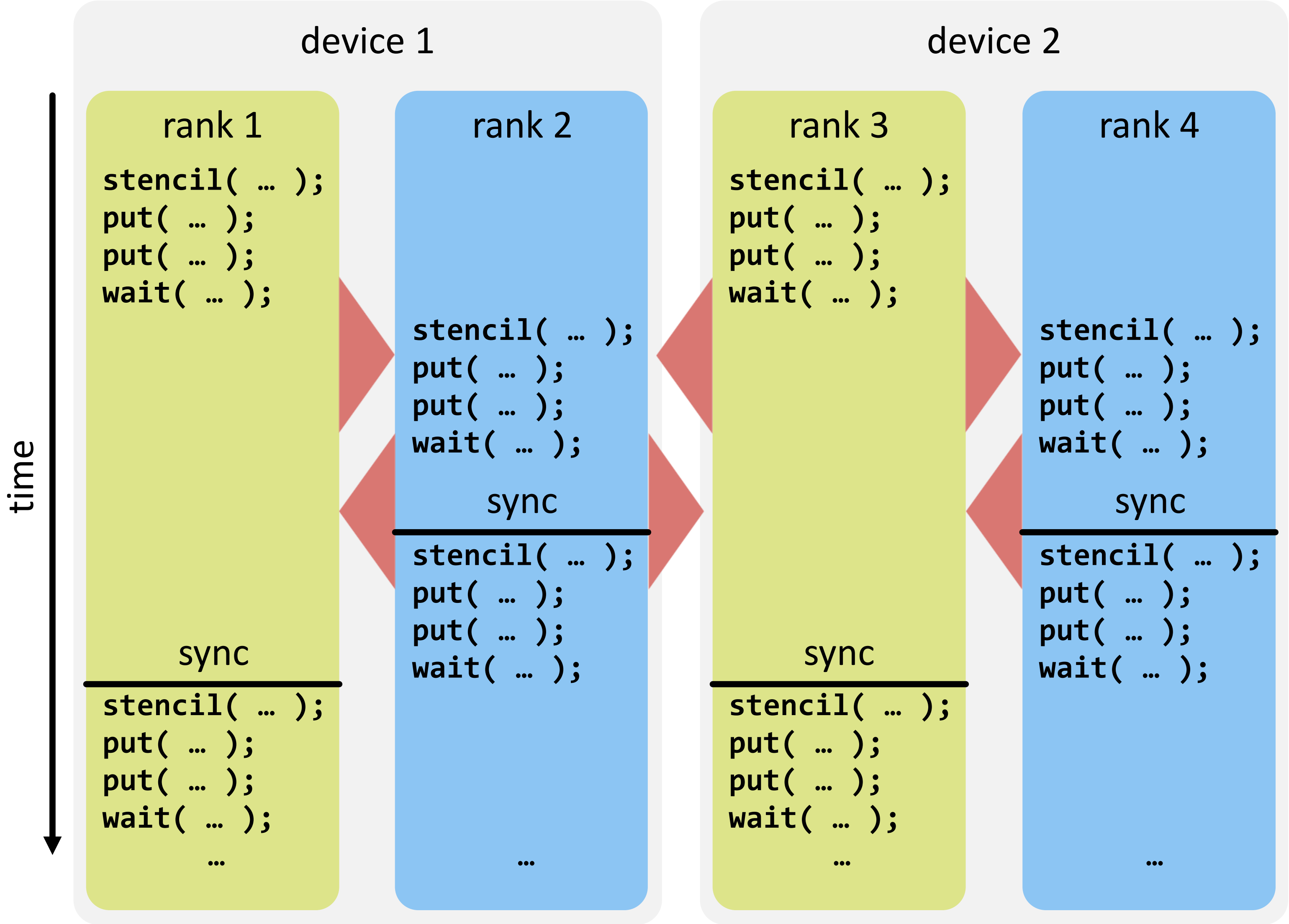
communication [2]

- iterative stencil kernel
- thread specific idx

- map ranks to blocks
- device-side put/get operations
- notifications for synchronization
- shared and distributed memory

[1] T. Gysi, J. Baer, TH: dCUDA: Hardware Supported Overlap of Computation and Communication, SC16
[2] R. Belli, T. Hoefler: Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization, IPDPS'15

Advantages of the dCUDA approach



rank 1:

```
stencil( … );
put( … );
put( … );
wait( … );
sync
stencil( … );
put( … );
put( … );
wait( … );
…
```

rank 2:

```
stencil( … );
put( … );
put( … );
wait( … );
sync
stencil( … );
put( … );
put( … );
wait( … );
…
```

rank 3:

```
stencil( … );
put( … );
put( … );
wait( … );
sync
stencil( … );
put( … );
put( … );
wait( … );
…
```

rank 4:

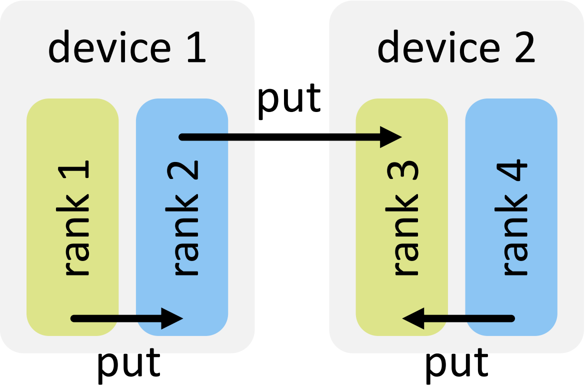
```
stencil( … );
put( … );
put( … );
wait( … );
sync
stencil( … );
put( … );
put( … );
wait( … );
…
```

performance

- avoid device synchronization
- latency hiding at cluster scale

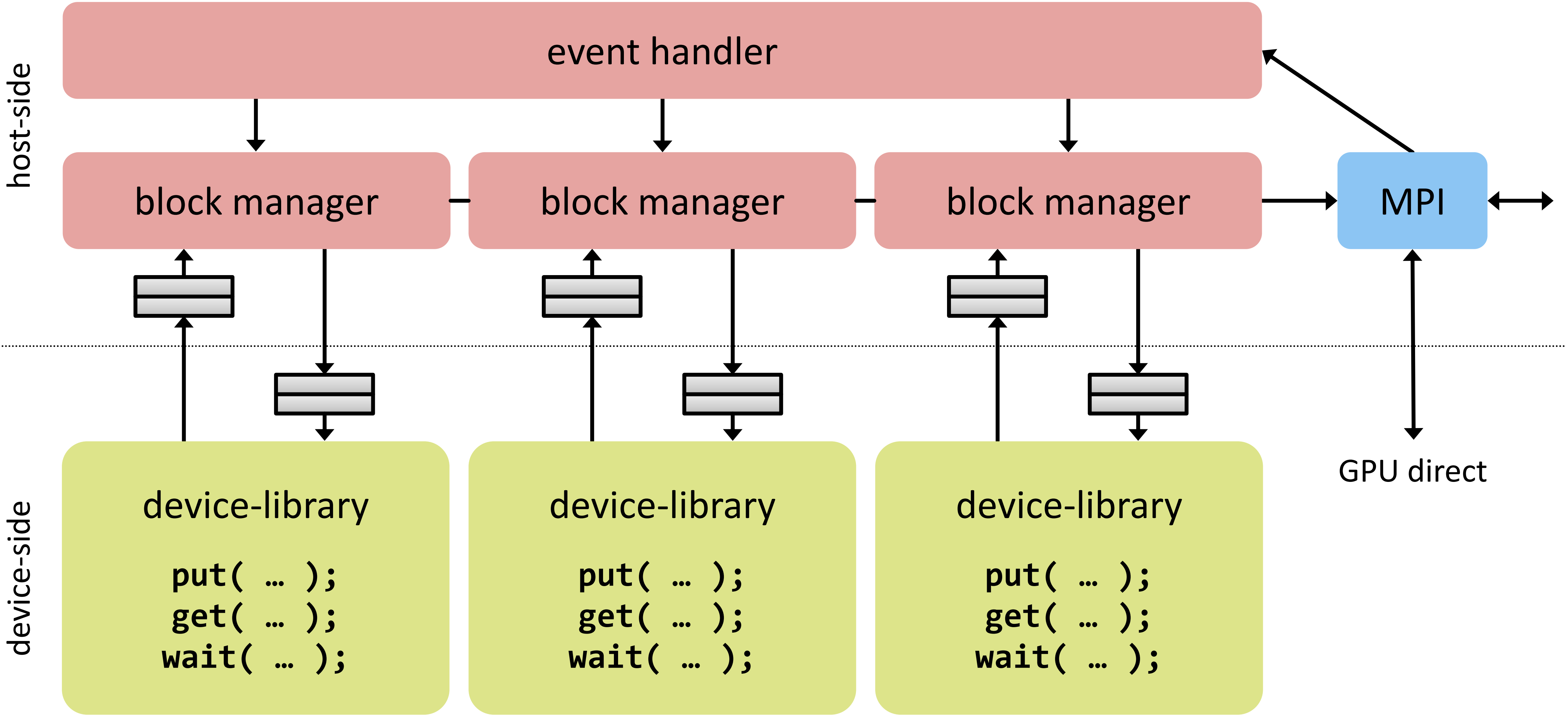
complexity

- unified programming model
- one communication mechanism



T. Gysi, J. Baer, TH: dCUDA: Hardware Supported Overlap of Computation and Communication, SC16

Implementation of the dCUDA runtime system



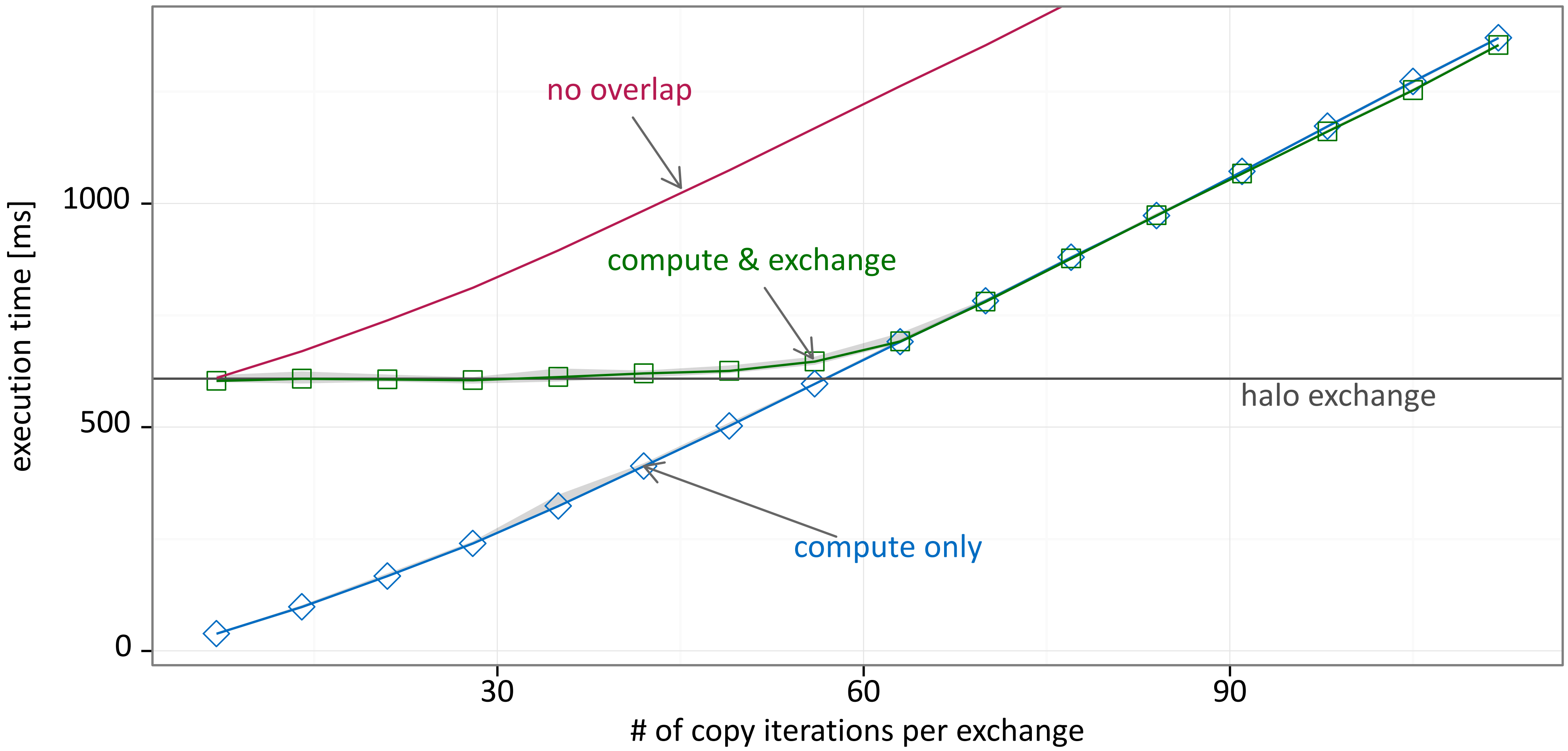
T. Gysi, J. Baer, TH: dCUDA: Hardware Supported Overlap of Computation and Communication, SC16

Evaluation

Cluster: 8 Haswell nodes, 1x Tesla K80 per node

Overlap of a copy kernel with halo exchange communication

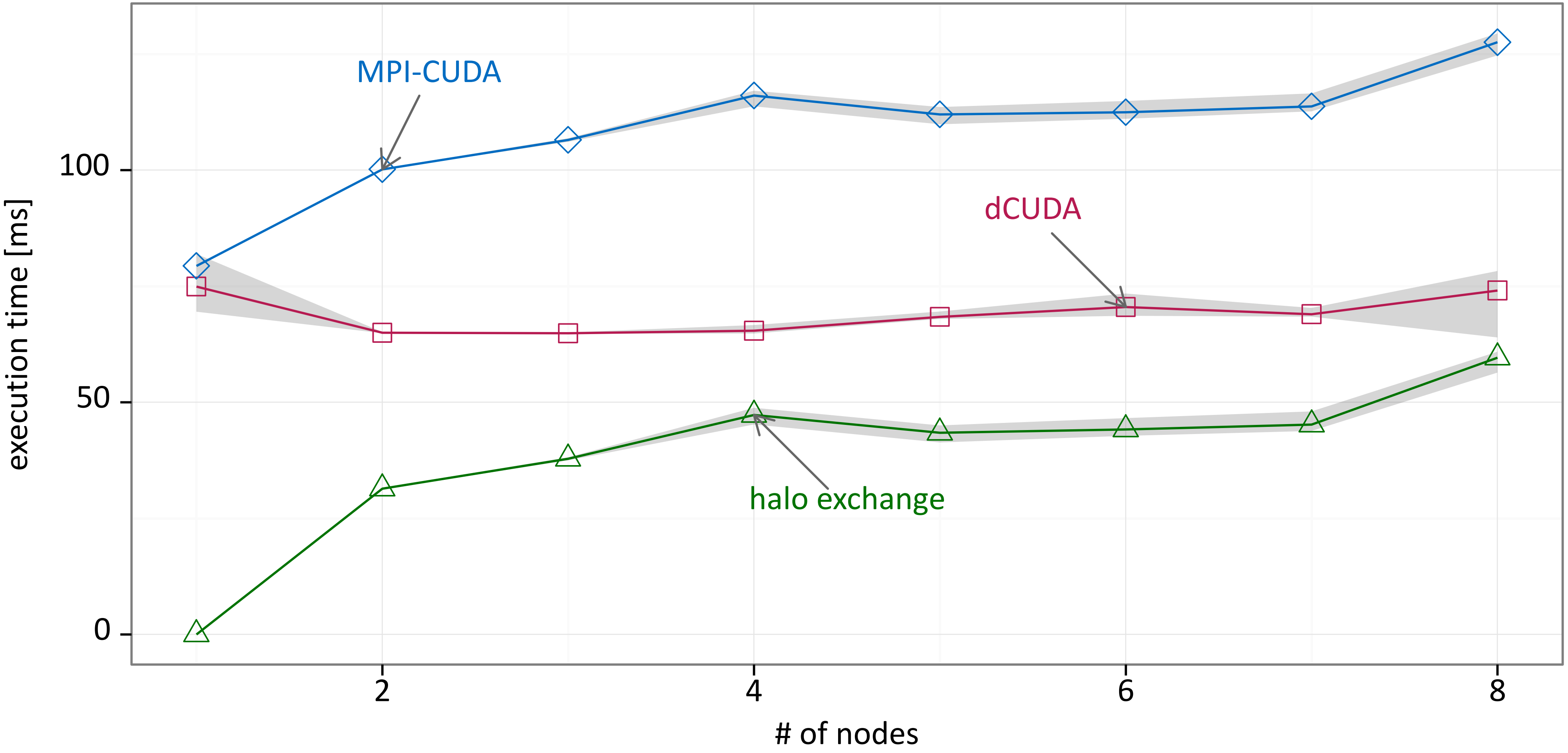
benchmarked on Greina (8 Haswell nodes with 1x Tesla K80 per node)



T. Gysi, J. Baer, TH: dCUDA: Hardware Supported Overlap of Computation and Communication, SC16

Weak scaling of MPI-CUDA and dCUDA for a stencil program

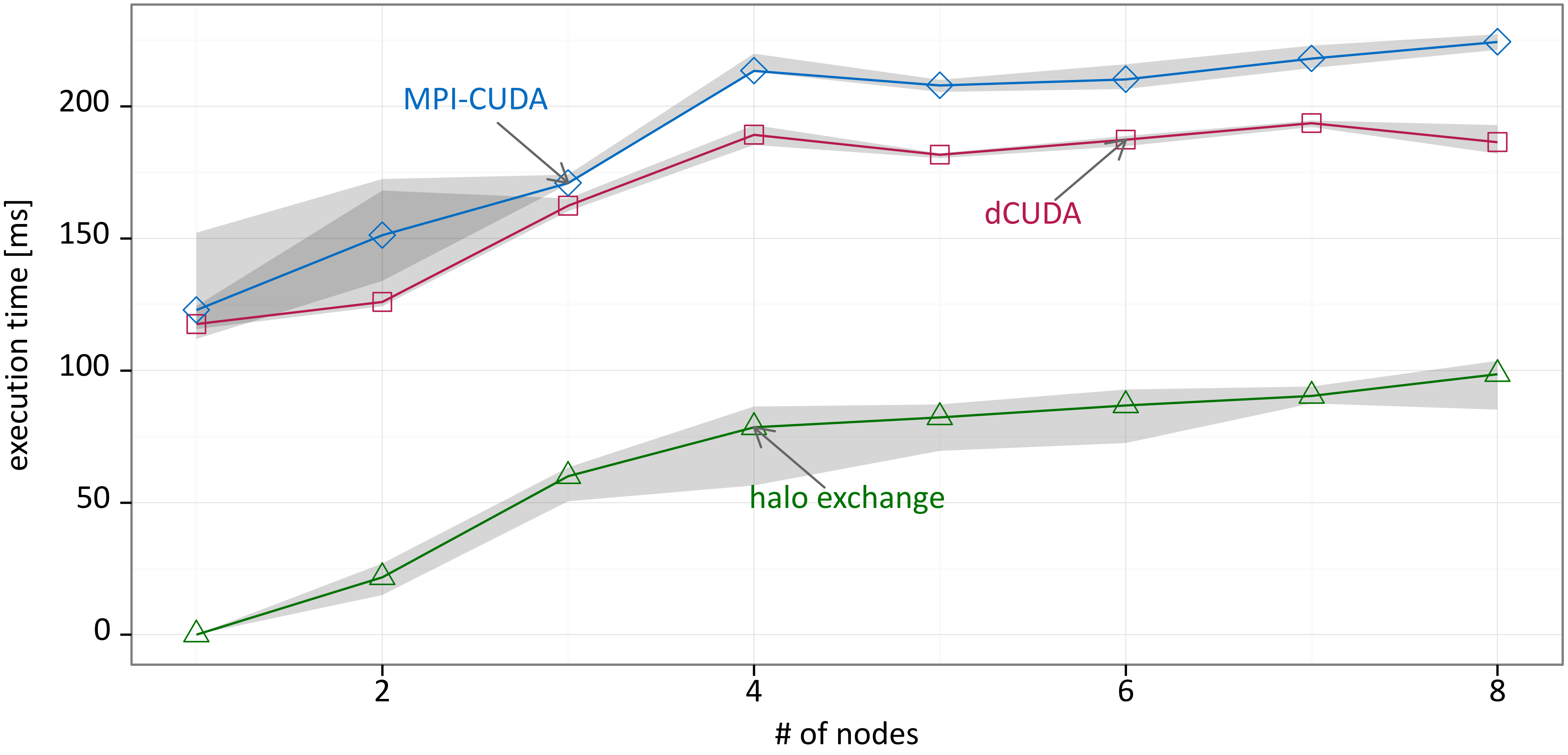
benchmarked on Greina (8 Haswell nodes with 1x Tesla K80 per node)



T. Gysi, J. Baer, TH: dCUDA: Hardware Supported Overlap of Computation and Communication, SC16

Weak scaling of MPI-CUDA and dCUDA for a particle simulation

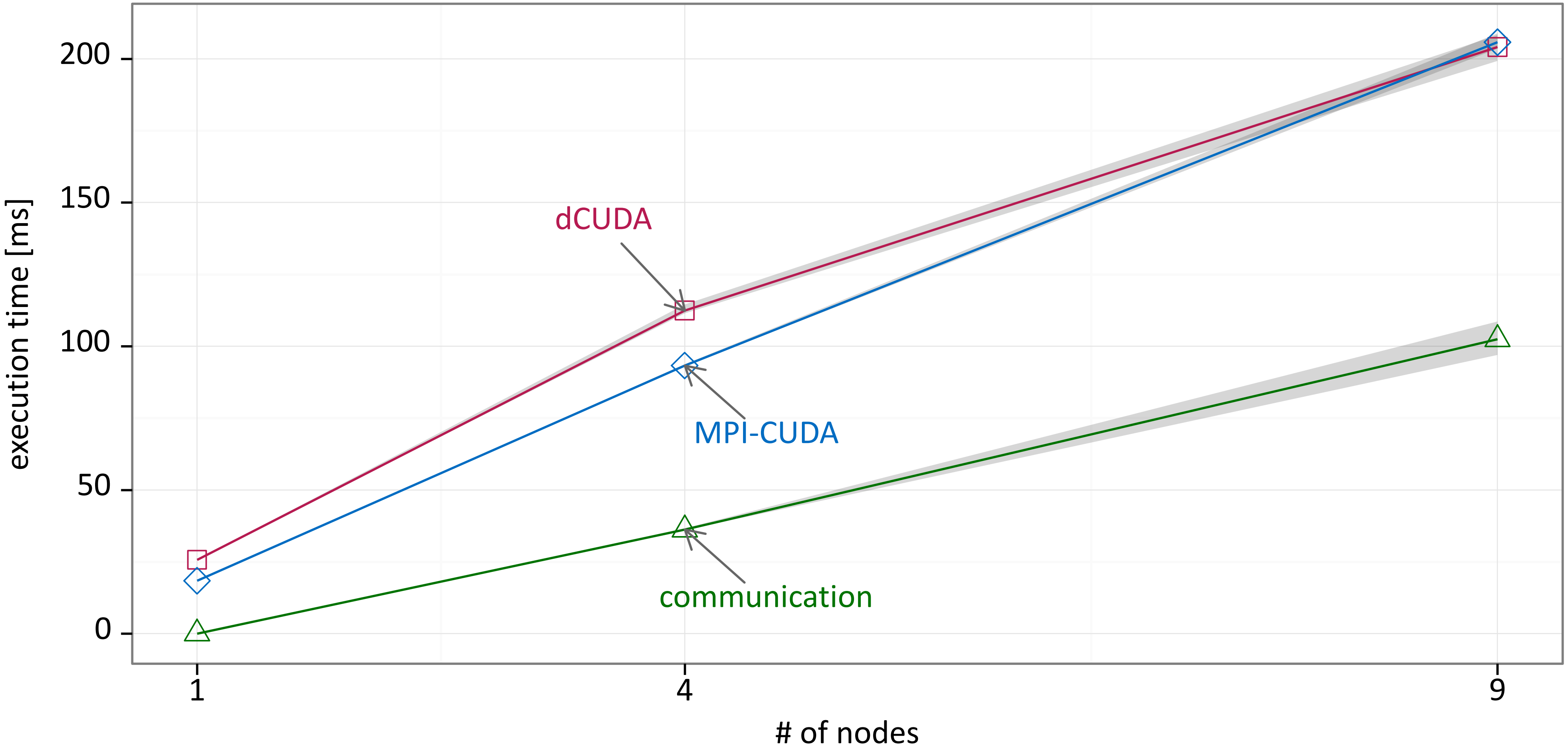
benchmarked on Greina (8 Haswell nodes with 1x Tesla K80 per node)



T. Gysi, J. Baer, TH: dCUDA: Hardware Supported Overlap of Computation and Communication, SC16

Weak scaling of MPI-CUDA and dCUDA for sparse-matrix vector multiplication

benchmarked on Greina (8 Haswell nodes with 1x Tesla K80 per node)



T. Gysi, J. Baer, TH: dCUDA: Hardware Supported Overlap of Computation and Communication, SC16

Conclusions

- unified programming model for GPU clusters
  - device-side remote memory access operations with notifications
  - transparent support of shared and distributed memory
- extend the latency hiding technique of CUDA to the full cluster
  - inter-node communication without device synchronization
  - use oversubscription & hardware threads to hide remote memory latencies
- automatic overlap of computation and communication
  - synthetic benchmarks demonstrate perfect overlap
  - example applications demonstrate the applicability to real codes
- https://spcl.inf.ethz.ch/Research/Parallel_Programming/dCUDA/

pasc Platform for Advanced Scientific Computing

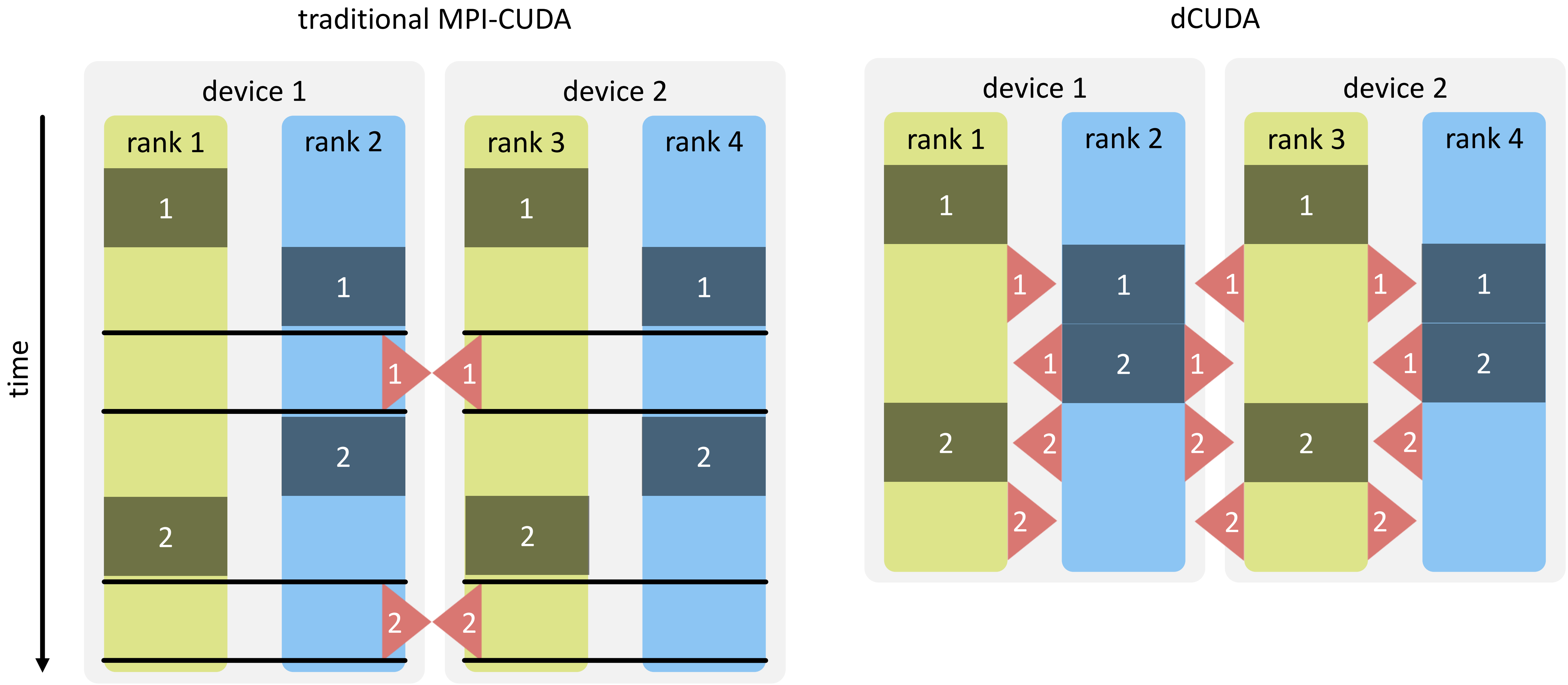
suk cus Swiss university conference

ETH-RAT

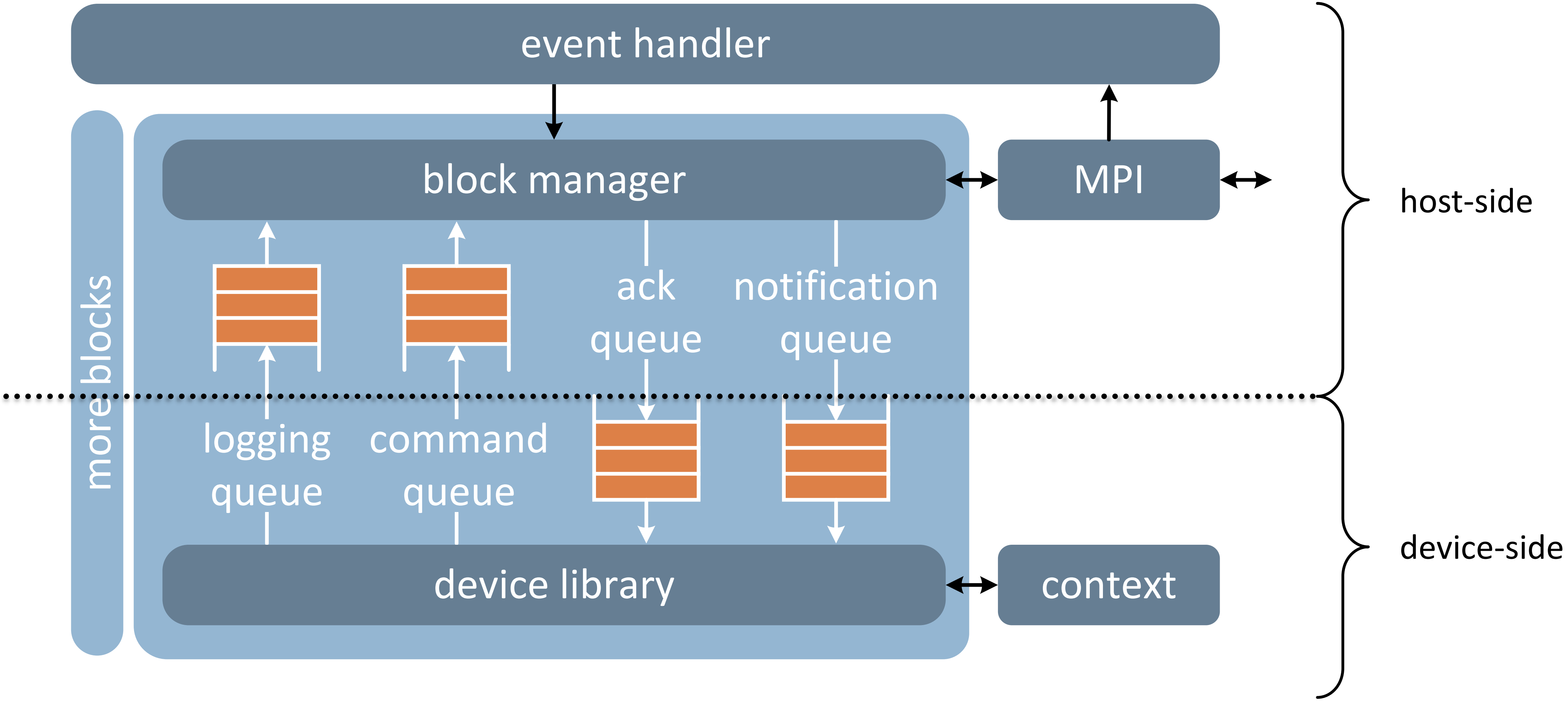
CSCS

Backup slides

Hardware utilization of dCUDA compared to MPI-CUDA

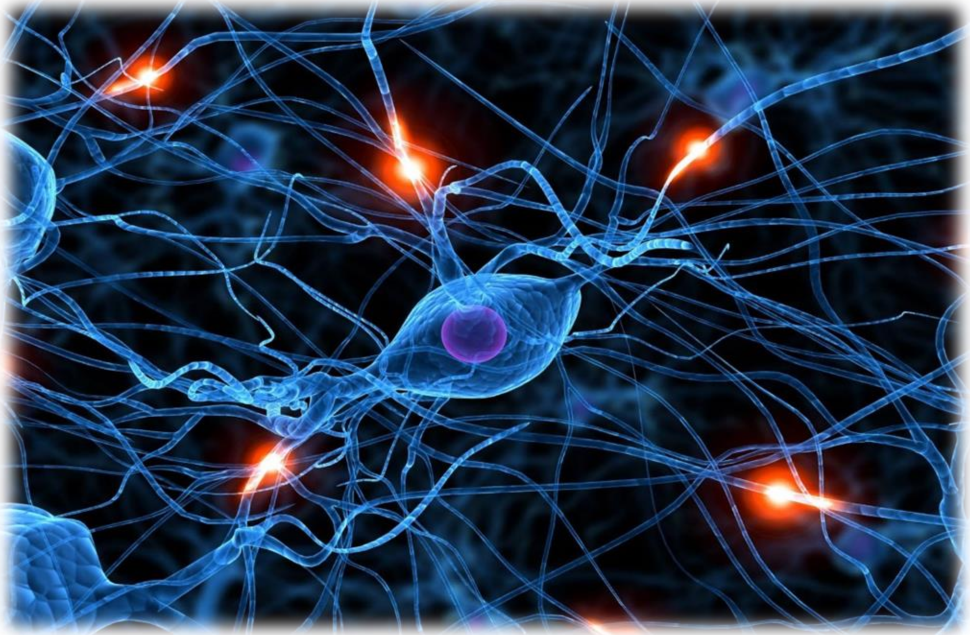


Implementation of the dCUDA runtime system



GPU clusters gained a lot of popularity in various application domains

machine learning

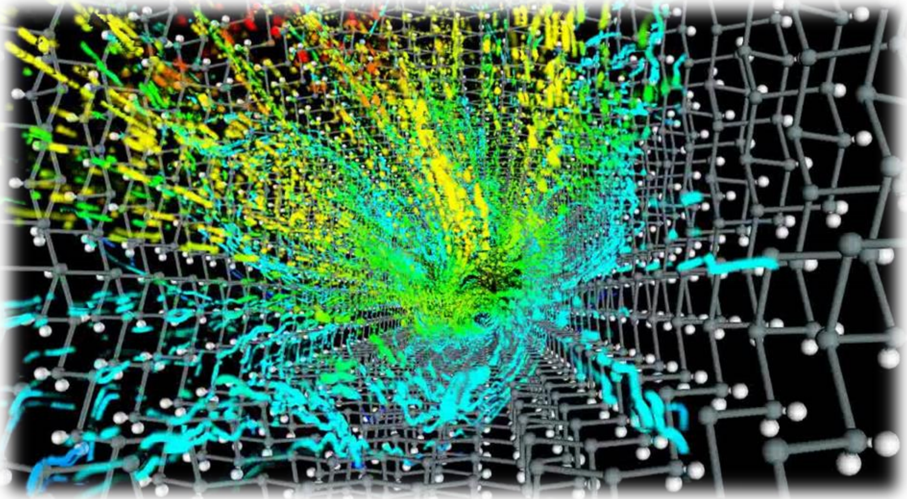


TensorFlow

weather & climate

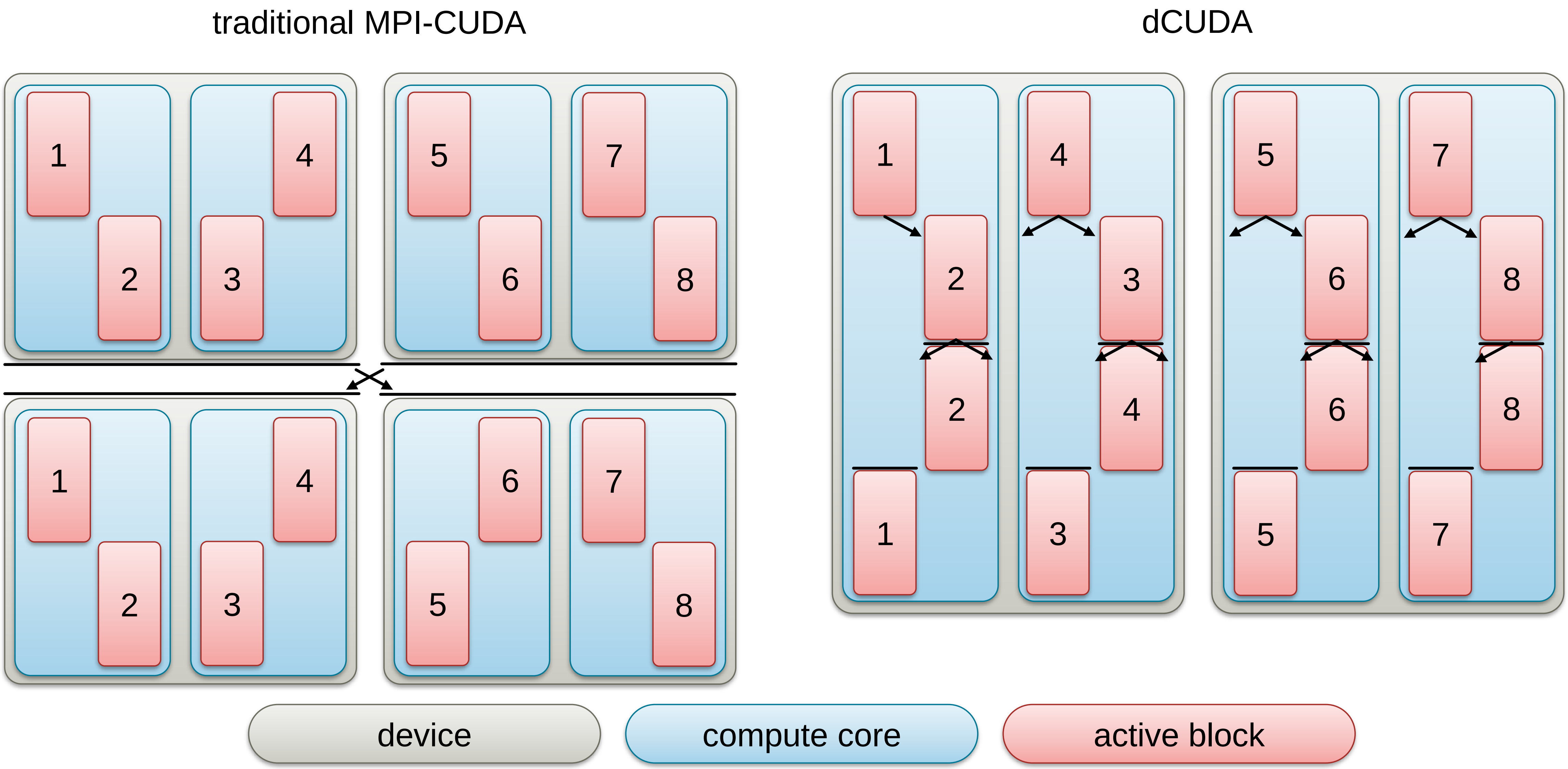
CONSORTIUM FOR SMALL SCALE MODELING
COSMO

molecular dynamics

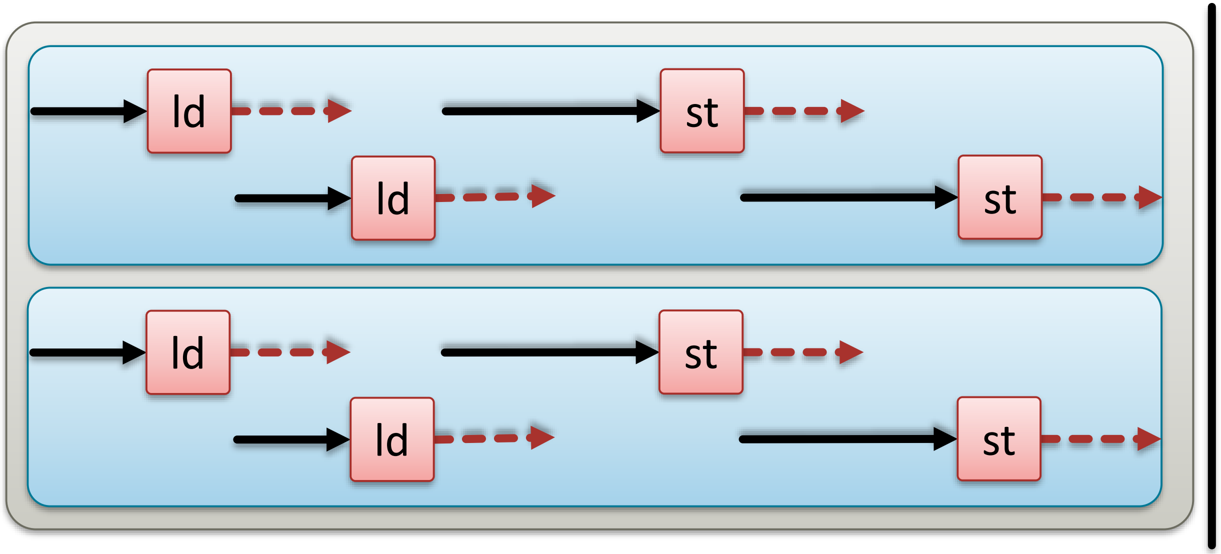


CP2K

Hardware supported overlap of computation & communication



Traditional GPU cluster programming using MPI and CUDA



intel
Intel® Xeon® Processor E5 v4
…

CUDA

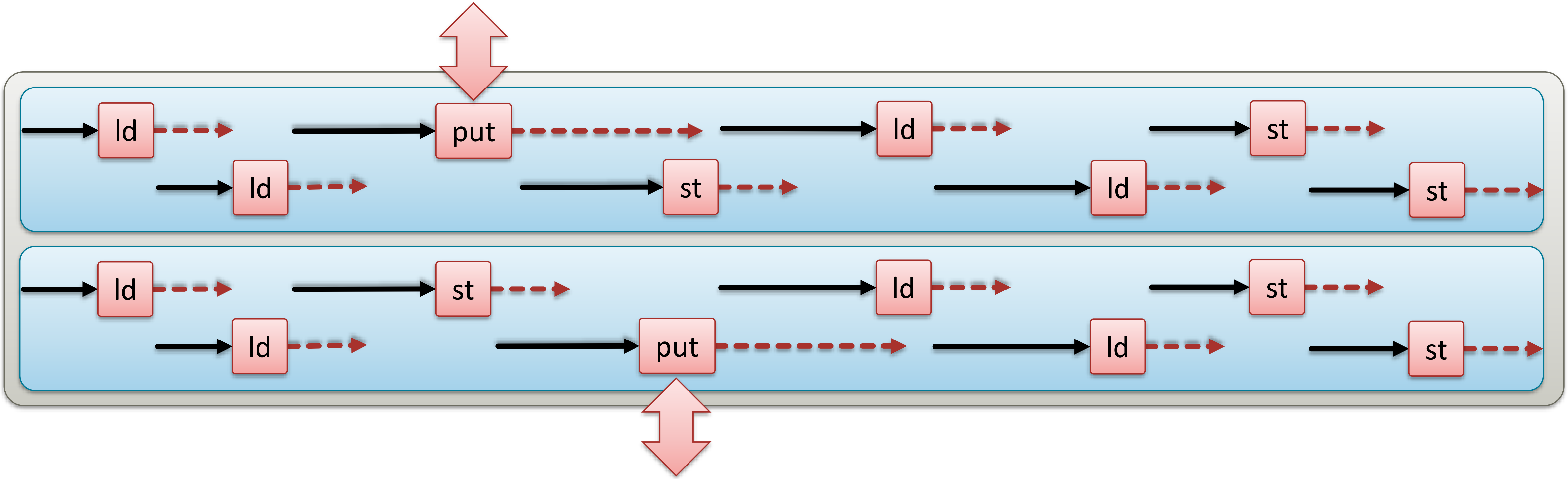
- over-subscribe hardware
- use spare parallel slack for latency hiding

MPI

- host controlled
- full device synchronization

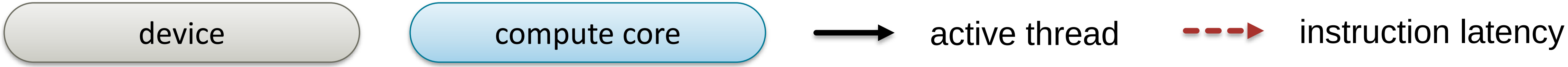
device　　compute core　　→ active thread　　- - -> instruction latency

How can we apply latency hiding on the full GPU cluster?



dCUDA (distributed CUDA)

- unified programming model for GPU clusters
- avoid unnecessary device synchronization to enable system wide latency hiding



Questions

- relation to NV link
  - NV link is a transport layer in the first place
  - it should enable a faster implementation
- synchronized programming in NV link
  - single kernel on machine with multiple GPUs connected by NV link
  - single node at the moment
  - will probably not scale to 1000s of nodes as there is no explicit communication