Simultaneous multithreading (SMT)

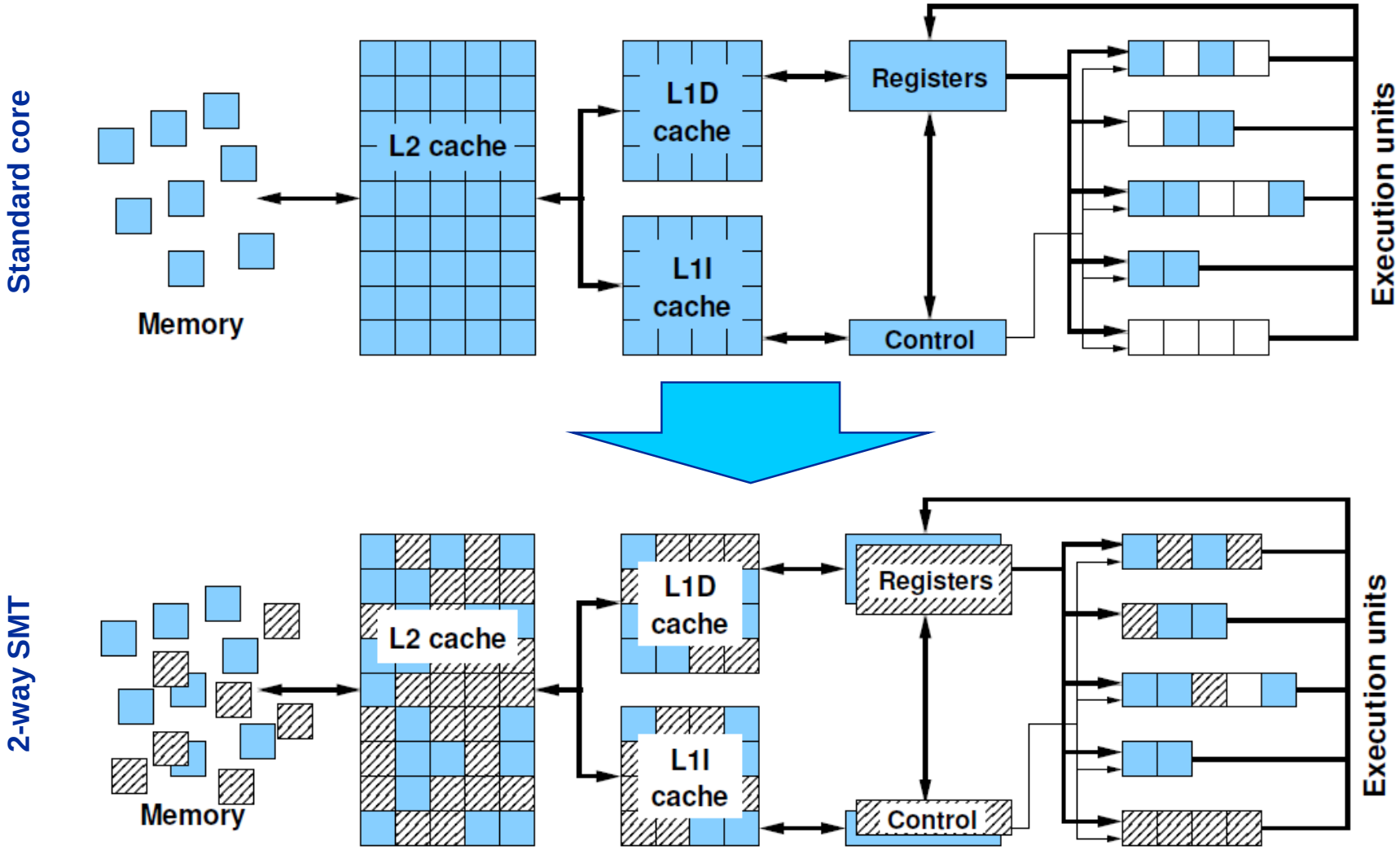
Principles and performance impact

SMT vs. independent instruction streams

Facts and fiction

SMT Makes a single physical core appear as two or more "logical" cores → multiple threads/processes run concurrently

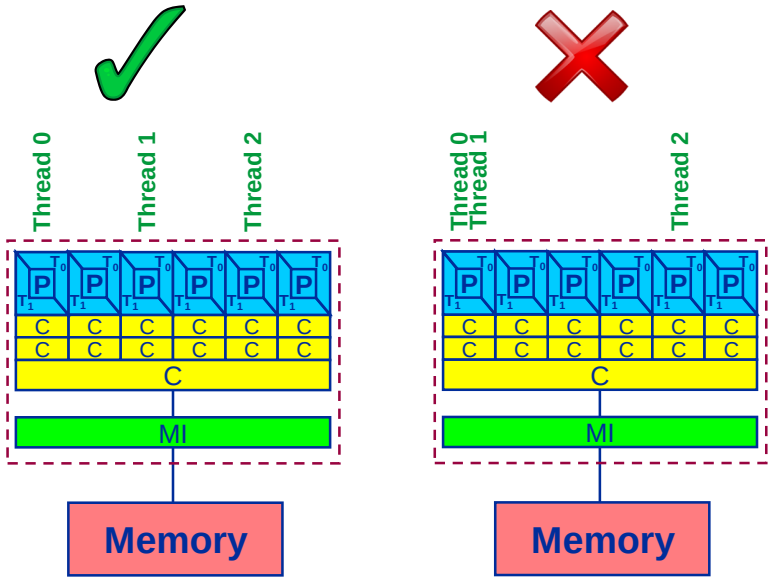
- SMT principle (2-way example):



SMT impact

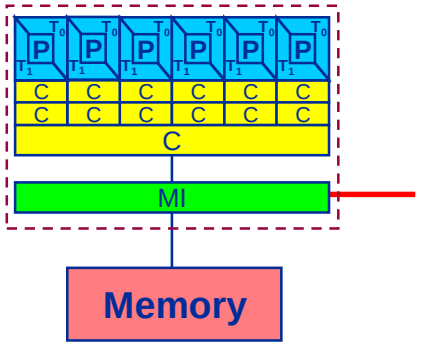
- SMT is primarily suited for increasing processor throughput
  - With multiple threads/processes running concurrently
- Scientific codes tend to utilize chip resources quite well
  - Standard optimizations (loop fusion, blocking, …)
  - High data and instruction-level parallelism
  - Exceptions do exist

- SMT is an important topology issue
  - SMT threads share almost all core resources
    - Pipelines, caches, data paths
  - Affinity matters!
  - If SMT is not needed
    - pin threads to physical cores
    - or switch it off via BIOS etc.



SMT impact

- Caveat: SMT threads share all caches!



- Possible benefit: Better pipeline throughput
  - Filling otherwise unused pipelines
  - Filling pipeline bubbles with other thread's executing instructions:

Thread 0:

```
do i=1,N
  a(i) = a(i-1)*c
enddo
```

Dependency → pipeline stalls until previous MULT is over

Thread 1:

```
do i=1,N
  b(i) = s*b(i-2)+d
enddo
```

Unrelated work in other thread can fill the pipeline bubbles

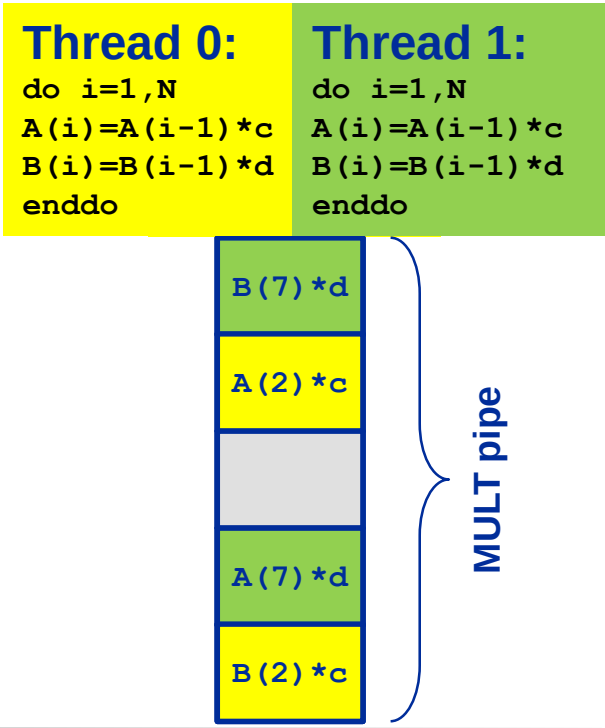
  - Beware: Executing it all in a single thread (if possible) may reach the same goal without SMT:

```
do i=1,N
  a(i) = a(i-1)*c
  b(i) = s*b(i-2)+d
enddo
```

The ideal workload for SMT

Simple loop-carried dependency benchmark `A(i) = s*A(i-1)`

- Bottleneck: MULT pipeline latency
  - Haswell CPU: 5 cy/it best case
- Running 2 threads via SMT: expect 2.5 cy/it if no other bottlenecks turn up
- Further improvement?
  - Multiple independent streams of instructions per thread
- What about the data transfer?

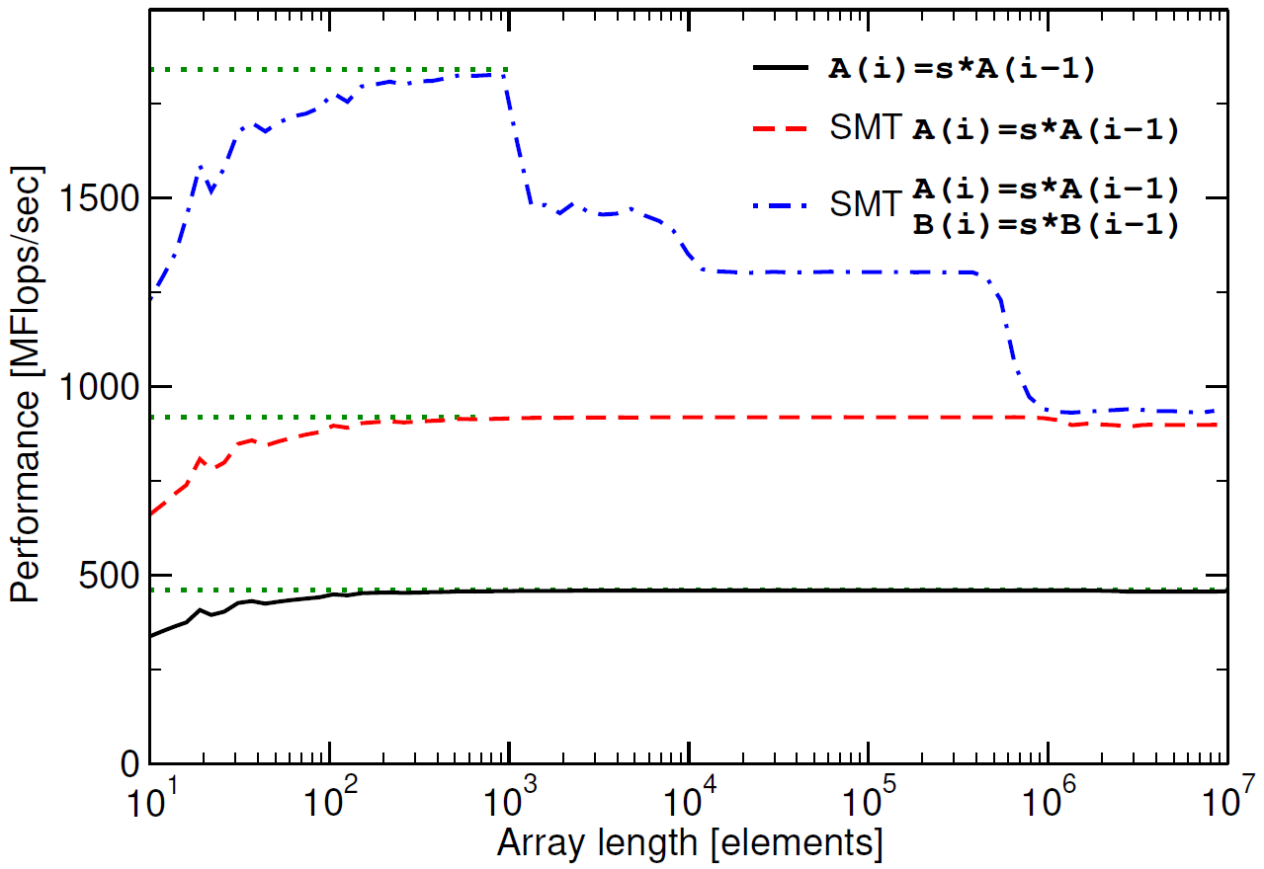


The ideal workload for SMT

Simple loop-carried dependency benchmark `A(i) = s*A(i-1)`

- Bottleneck: MULT pipeline latency
- Performance insensitive to problem size w/o SMT
- Fill bubbles via
  - SMT
  - Multiple indep. streams

Intel Xeon “Haswell” E5-2695v3, 2.3GHz



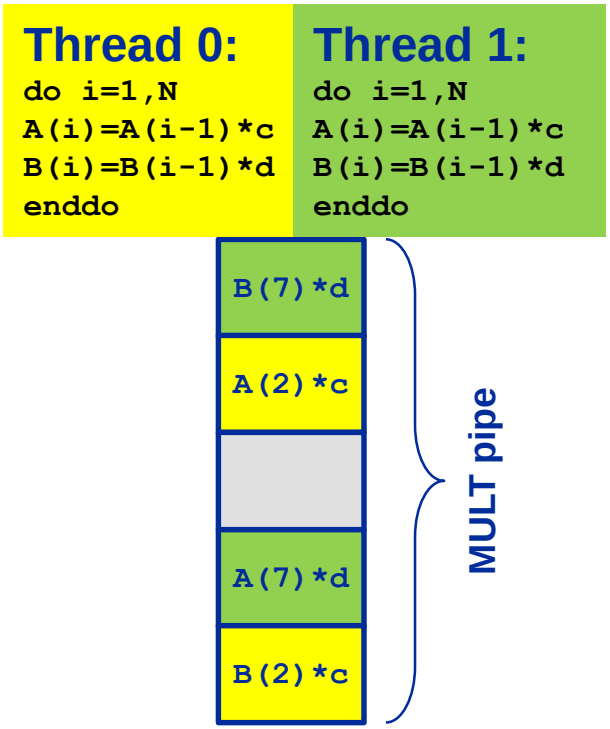
SMT myths: Facts and fiction (1)

Myth: “If the code is compute-bound, then the functional units should be saturated and SMT should show no improvement.”

Truth

1. A compute-bound loop does not necessarily saturate the pipelines; dependencies can cause a lot of bubbles, which may be filled by SMT threads.

2. If a pipeline is already full, SMT will not improve its utilization



SMT myths: Facts and fiction (2)

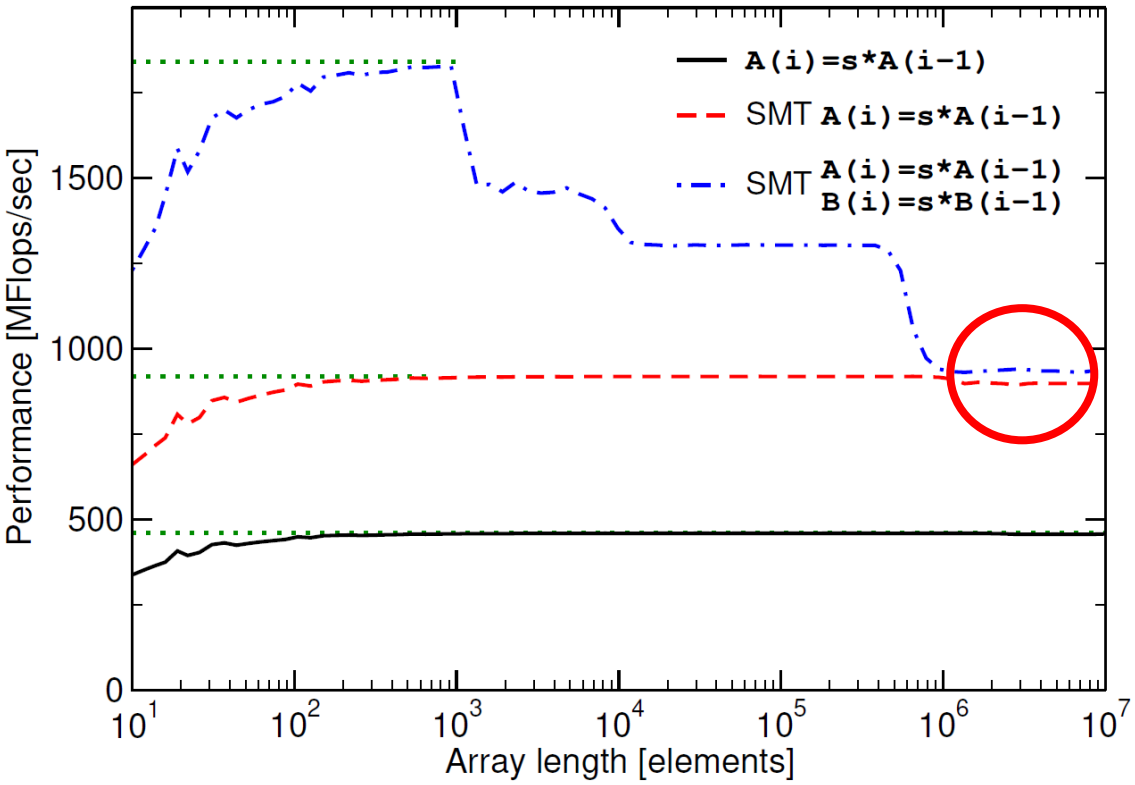
Myth: “If the code is memory-bound, SMT should help because it can fill the bubbles left by waiting for data from memory.”

Truth:

1. If the maximum memory bandwidth is already reached, SMT will not help since the relevant resource (bandwidth) is exhausted.

2. If the relevant bottleneck is not exhausted, SMT may help since it can fill bubbles in the LOAD pipeline.

This applies also to other “relevant bottlenecks!”



SMT myths: Facts and fiction (3)

Myth: “SMT can help bridge the latency to memory (more outstanding references).”

Truth:
Outstanding references may or may not be bound to SMT threads; they may be a resource of the memory interface and shared by all threads. The benefit of SMT with memory-bound code is usually due to better utilization of the pipelines so that less time gets “wasted” in the cache hierarchy.

See also the “ECM Performance Model” later on.