SAGE: Software-based Attestation for GPU Execution

Andrei Ivanov, Benjamin Rothenberger, Arnaud Dethise,
Marco Canini, Torsten Hoefler, Adrian Perrig

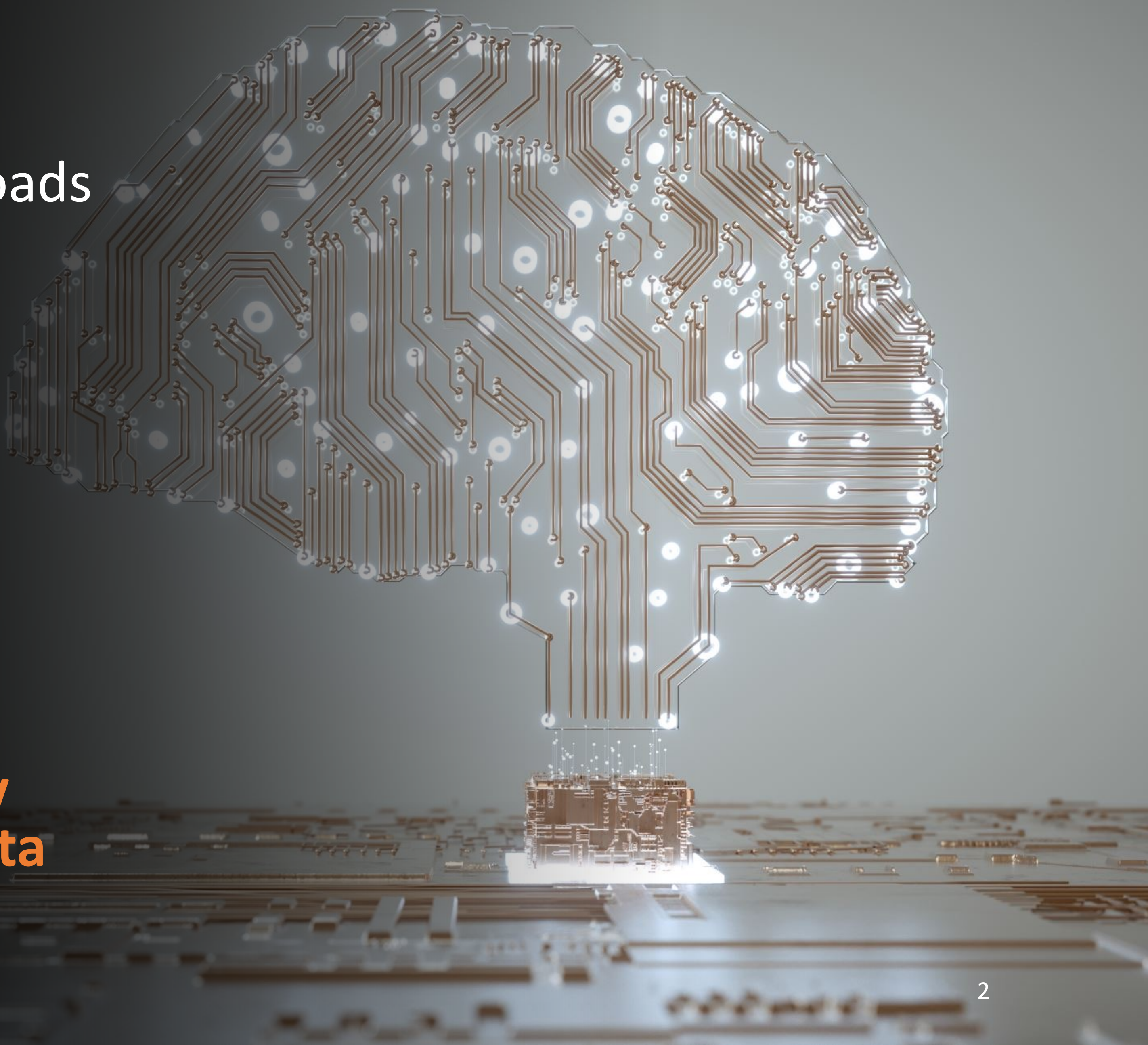
ETH zürich

KAUST

By 2026, majority of Cloud workloads will be DL [Research and Markets, Jul’21 report]

Accelerators (mostly GPUs!) necessary to process vast data volumes of DL applications

DL applied to security-critical and sensitive domains makes **integrity** and **secrecy** for both **code** and **data** within GPUs paramount



How can we execute code securely on GPUs, today?

1. No widespread deployed hardware TEEs, uptake might be a while
2. TEE tech is still a moving target (see SGX)
3. HW-based attestation difficult to secure, impossible to patch

To bridge this gap with a **software-only** approach

SAGE: Software-based Attestation for GPU Execution

SAGE

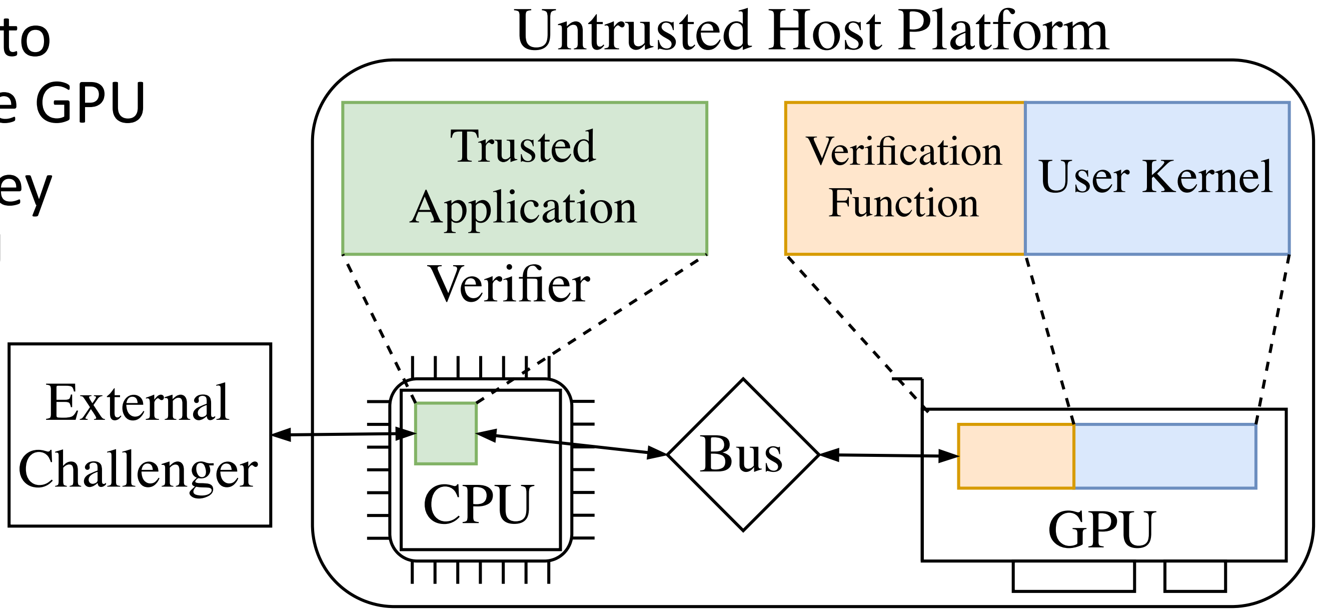
- The first software-based attestation mechanism for GPU execution providing code and data integrity+secrecy for NVIDIA Ampere GPUs

- SAGE **guarantees** that:
  - on the untrusted GPU device …
  - user kernels are unmodified
  - user kernels are invoked for execution
  - user kernels are executed untampered
  - ... despite the potential presence of a malicious actor

SAGE

- The first software-based attestation mechanism for GPU execution providing code and data integrity+secrecy for NVIDIA Ampere GPUs

- CPU enclave (e.g., SGX) serves as local trusted verifier
  - Kicks off a software primitive to establish a root of trust on the GPU
  - Also sets up a shared secret key between verifier and the GPU



Verifiable code execution

Goal: provide verifier with guarantee about what code executed on the GPU

Approach:

1. Verify code integrity through Root-of-Trust attestation
2. Set up untampered code execution environment
3. Execute code

Root-of-Trust (RoT) establishment

Established RoT ensures that:

- state of an untrusted system contains **all and only content** chosen by trusted local verifier, and code begins execution in that state
- or that the verifier discovers the existence of modifications

→ Attestation of code on GPU enables RoT establishment

Software-based attestation for CPU

Basic idea (SWATT [1], PIONEER [2], …)

1. A verification function runs on an untrusted system and computes a checksum over itself
    - Both the checksum value and the time to compute it matter
    - **Noticeably slow down or incorrect if an adversary tampers with the system**
2. A trusted verifier checks for the correct checksum and that value is returned before a threshold time

1 + 2: establish a RoT (or fail), kick off intended code

[1] A. Seshadri, A. Perrig, L. van Doorn and P. Khosla, "SWATT: softWare-based attestation for embedded devices," *IEEE Symposium on Security and Privacy, 2004.*

[2] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. Van Doorn, P. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," *ACM SOSP, 2005*.

Software-based attestation for GPU

Challenges

- Very challenging threat model
  - Data and code secrecy + integrity
  - Malicious code on CPU and/or GPU, snooping interconnect
- Design of **verification function** for GPU
  - Lack of GPU architecture documentation … very hard to:
    - write native GPU code, no toolchain support
    - achieve optimal GPU utilization
    - predictable execution time (verifier must determine correct execution time)
  - No true random number generator (needed for crypto)
  - Fend off subtle attacks (e.g., pre-computation, data substitution)

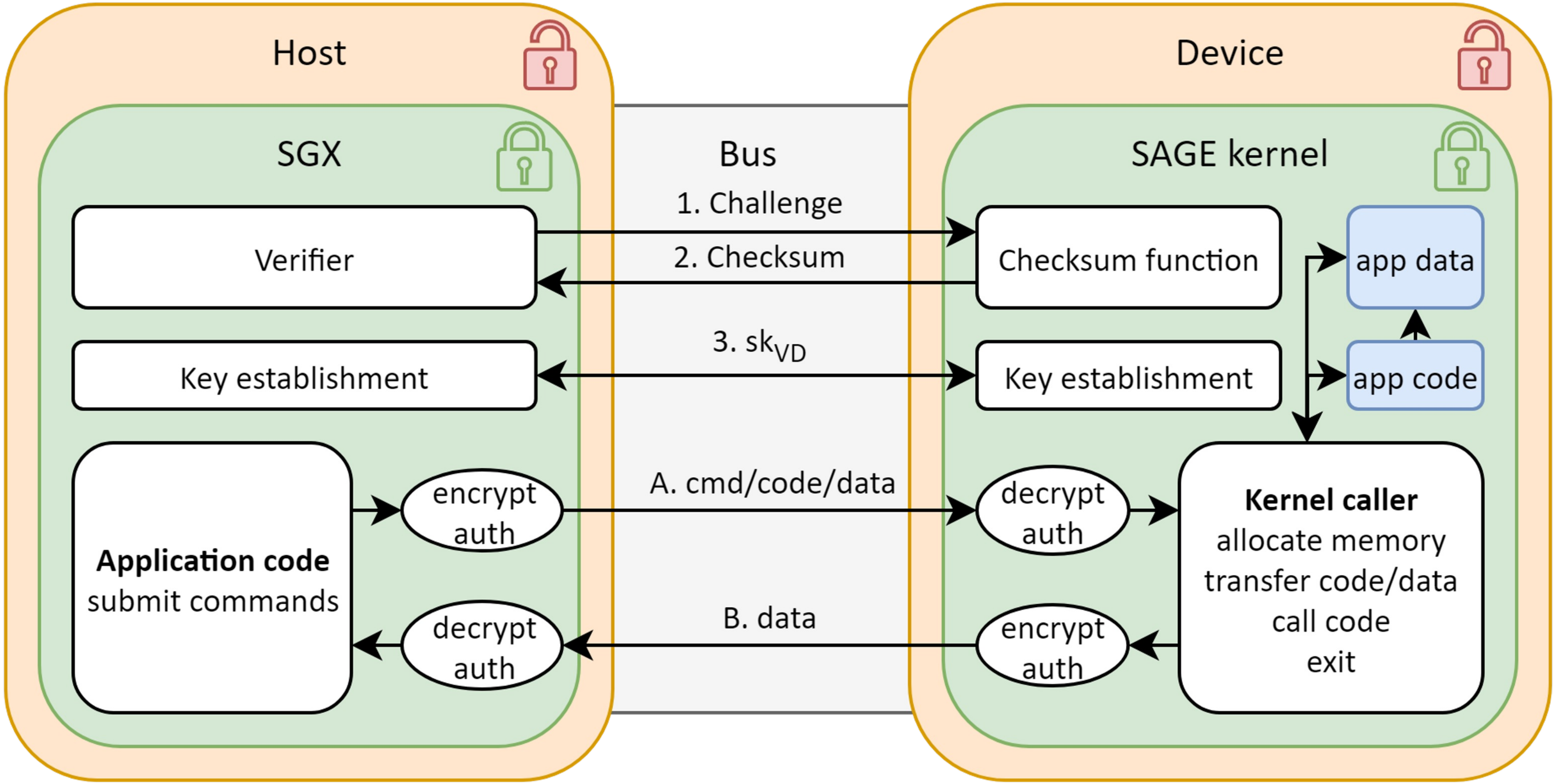
Assumptions

- Verifier and GPU on the same machine
- Verifier is trusted (e.g., SGX)

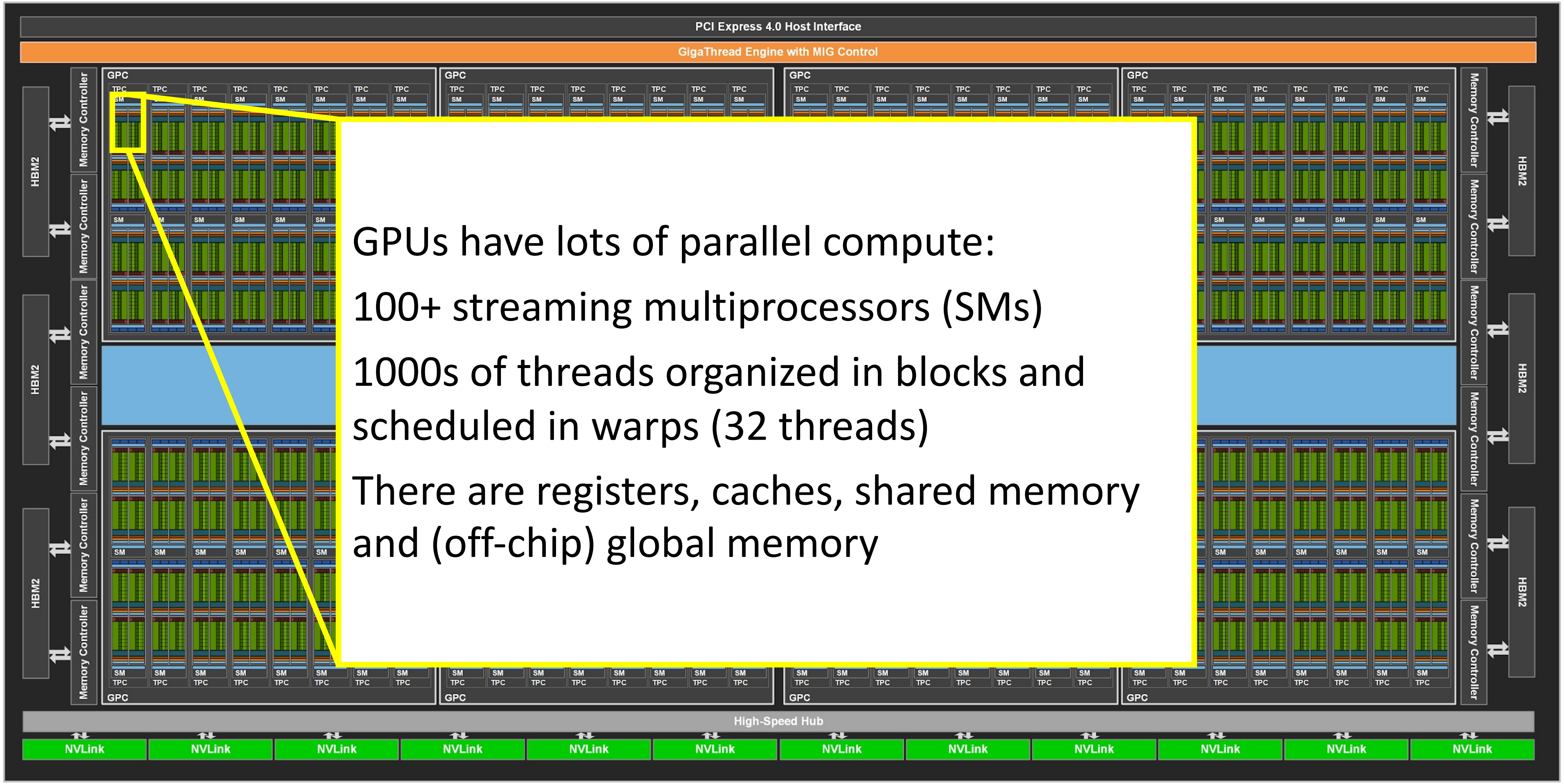
- GPU details are known (model, clock speed, specs.)
- If multi-GPU node, the fastest GPU type is used

- TCB includes GPU runtime and driver, plus the TCB of Intel SGX

Overview of SAGE



(G)A100 : How to utilize this beast?



GPUs have lots of parallel compute:

100+ streaming multiprocessors (SMs)

1000s of threads organized in blocks and scheduled in warps (32 threads)

There are registers, caches, shared memory and (off-chip) global memory

[Source: NVIDIA GA100 Full GPU 128 SMs]

Verification Function (VF)

Time-optimal implementation

- can't be improved
- additional code makes it slower

Predictable execution time

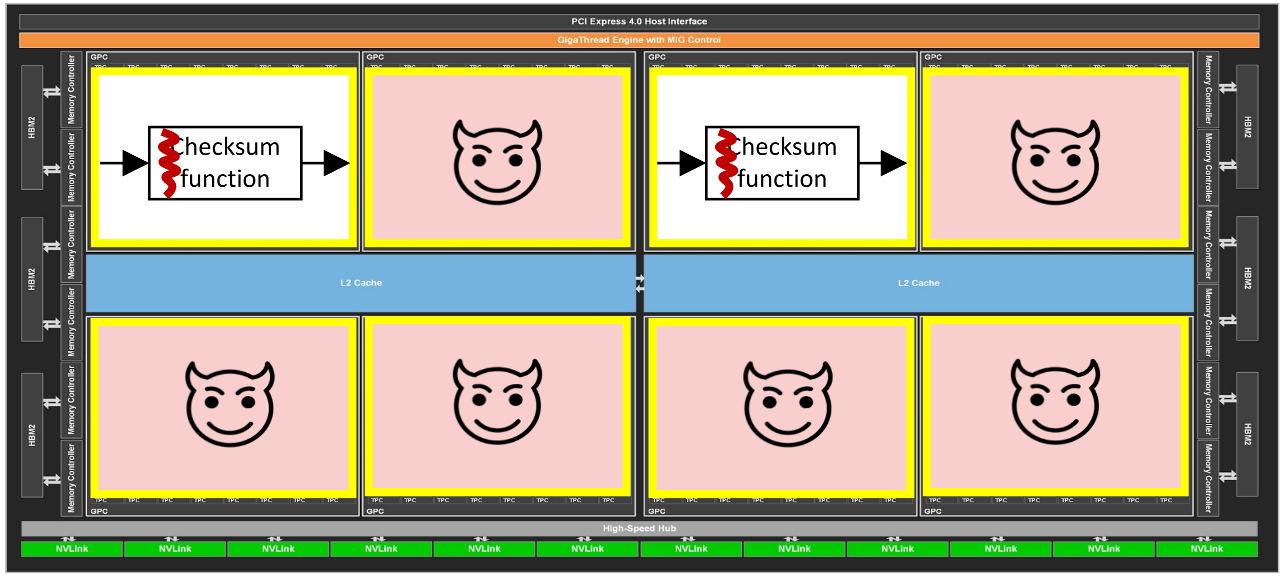
- peak 1 instruction/cycle

Challenge-dependent checksums

- no precomputation

Computation is parallel

- combine values from threads at the end



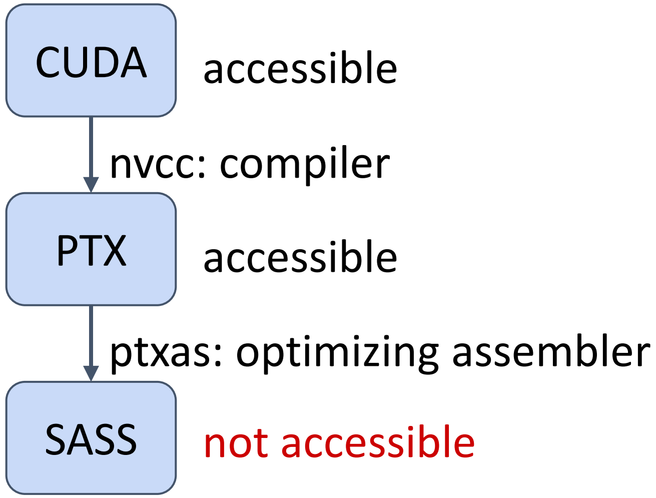
[Source: NVIDIA GA100 Full GPU 128 SMs]

Can’t leave resources to an attacker!

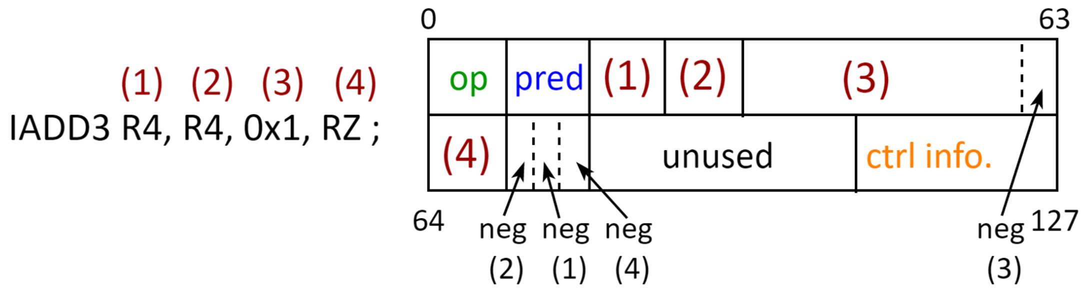
Idea: if an attacker alters the verification function but wants to forge a correct checksum value, needs to do “more work”, causing a time overhead

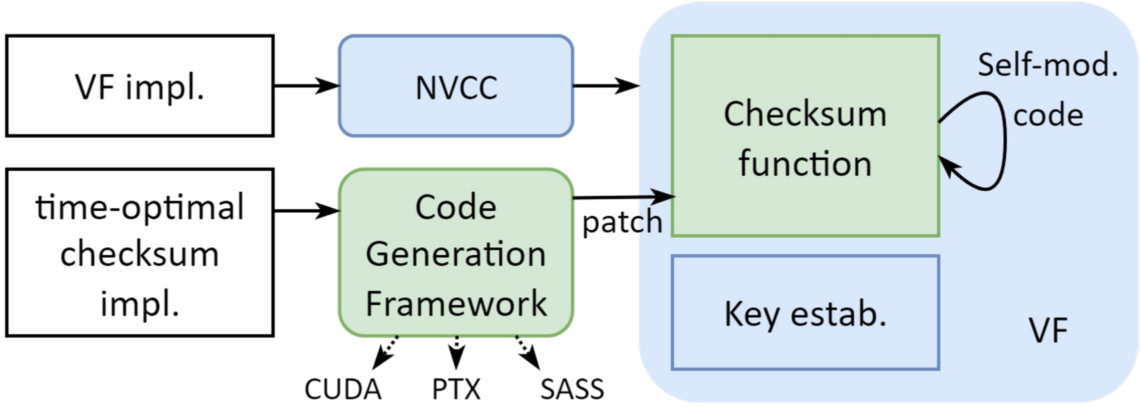
Code generation framework

- Achieving optimal GPU utilization is hard
- Compilers are not optimal
- No access to register allocation and instruction ordering



We discover SASS instruction <u>encoding</u> and build a code generation <u>framework</u>





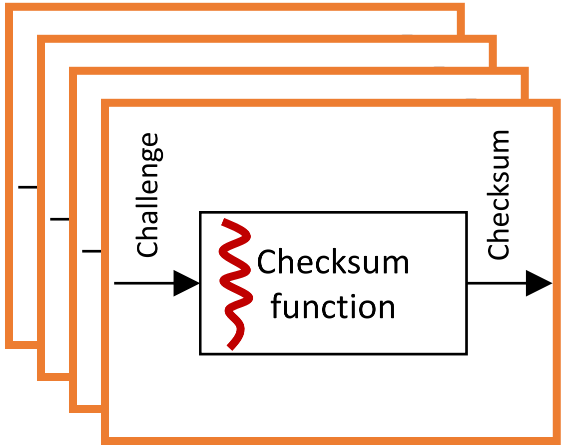
Key establishment

Goal: establish a symmetric key between trusted verifier and the GPU without any prior secret

Approach:

- Rely on SAKE protocol [Seshadri at al., DCOSS’08]
  - DH key exchange + Guy Fawkes for auth. (commitment using hash chains)
  - Exploits the asymmetry between genuine and modified checksum function
- Adapted to SAGE (checksum func., single challenger, crypto primitives)
  - Formally verified w/ Tamarin prover
- Implemented a TRNG (for DH) on the GPU
  - simultaneous memory accesses unpredictably flips bits in shared variables

Checksum function



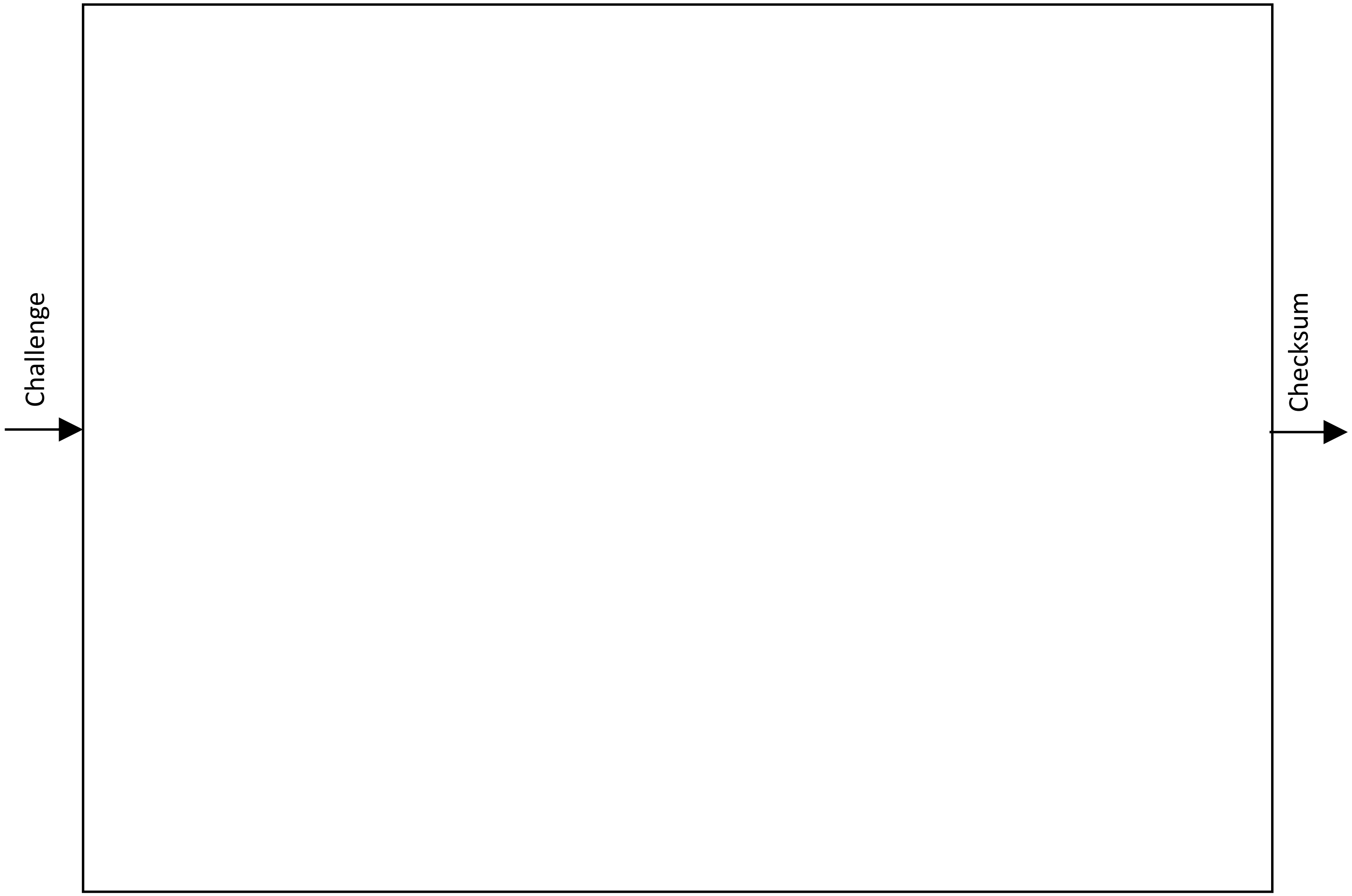
Execute lots of parallel checksum computations
(each with a different seed)

Combine as single value via XOR hierarchy at the end

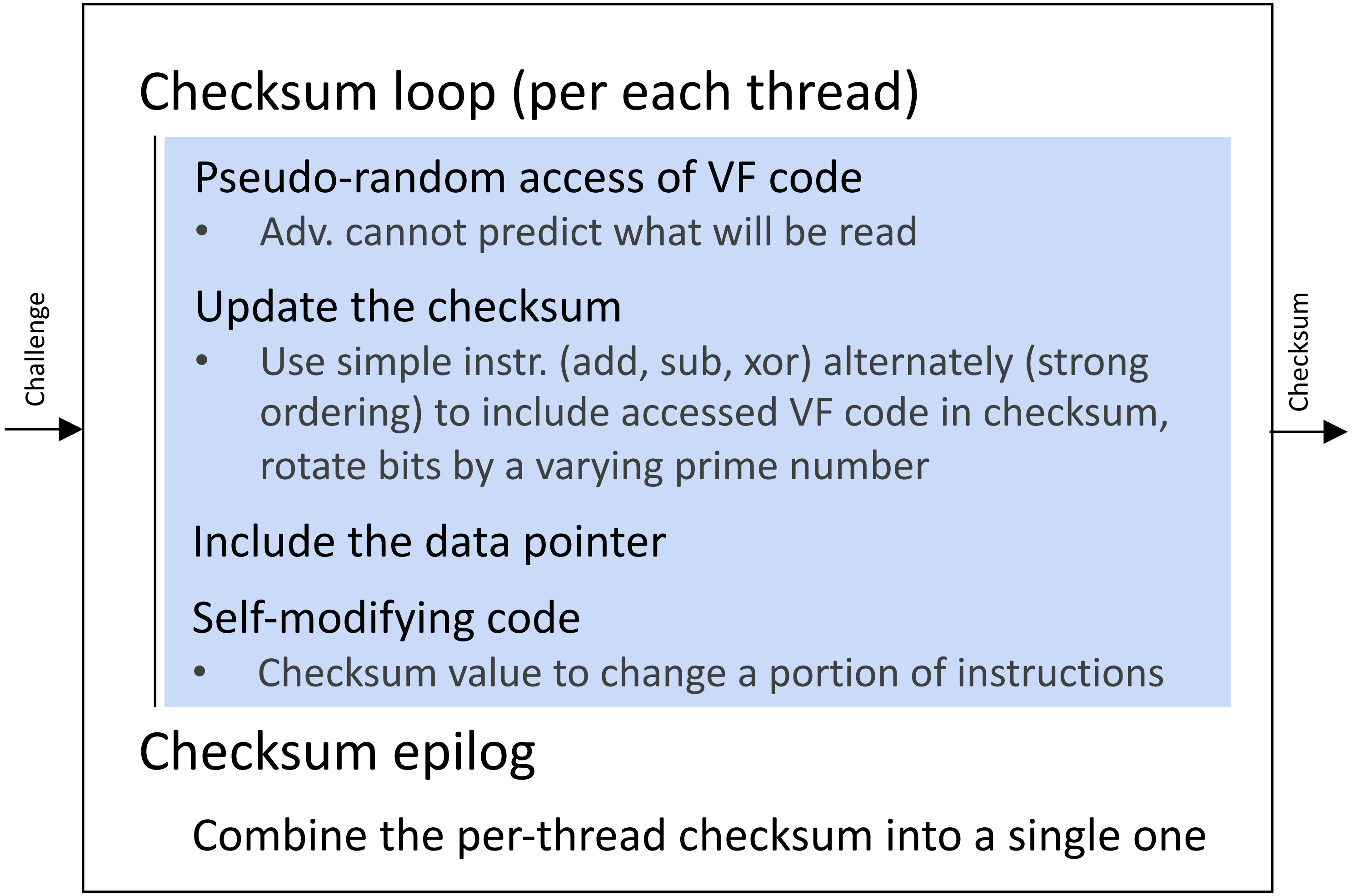
Occupy all SMs, threads
Fill all FMA and ALU pipelines
Occupy all registers

Don’t exceed L0 and L1 instruction caches
Avoid expensive frequent synchronizations

Checksum function – concretely



Checksum function – concretely



Checksum loop (per each thread)

Pseudo-random access of VF code

- Adv. cannot predict what will be read

Update the checksum

- Use simple instr. (add, sub, xor) alternately (strong ordering) to include accessed VF code in checksum, rotate bits by a varying prime number

Include the data pointer

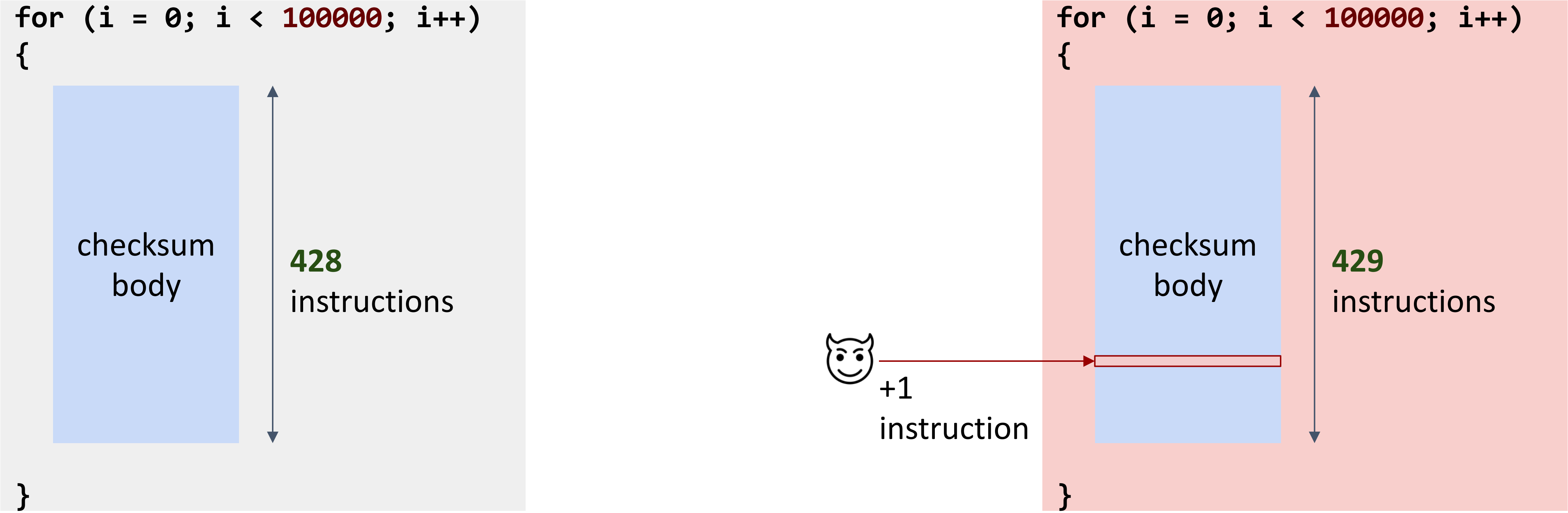
Self-modifying code

- Checksum value to change a portion of instructions

Checksum epilog

Combine the per-thread checksum into a single one

Checksum verification threshold



```
for (i = 0; i < 100000; i++)
{
    checksum body
}
```

428 instructions

😈 +1 instruction

```
for (i = 0; i < 100000; i++)
{
    checksum body
}
```

429 instructions

A100 time: $\mathbf{T_{avg} = 0.4941}$ **s** (99% of peak) σ = **0.0009 s**

A100 time: $\mathbf{T_{min} = 0.4966}$ **s** (98% of peak)

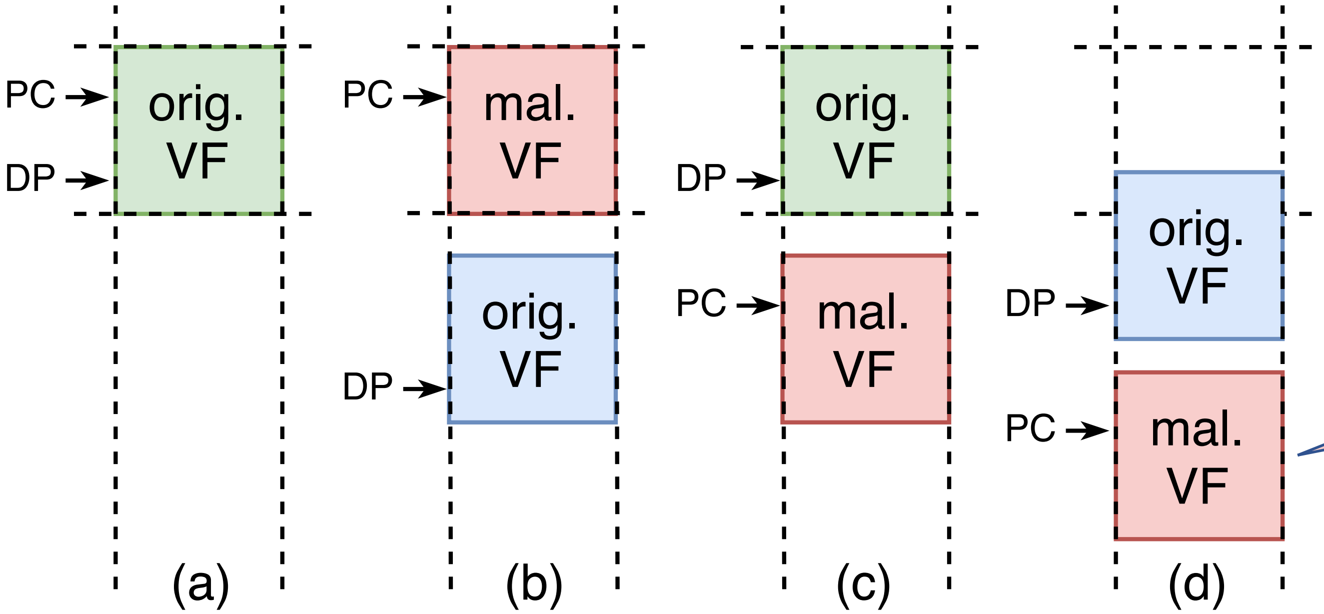
Checksum validation:
AMD EPYC 7742: **21.6 s**
Intel Xeon Gold 6348: **102 s**

$T_{avg} + 2.5\sigma < T_{min}$ ⇨ False positive probability **<1%**

$0.4964 < 0.4966$ ✅

Memory copy attacks

Altered malicious VF runs along side the original VF



Checksum result is the same?

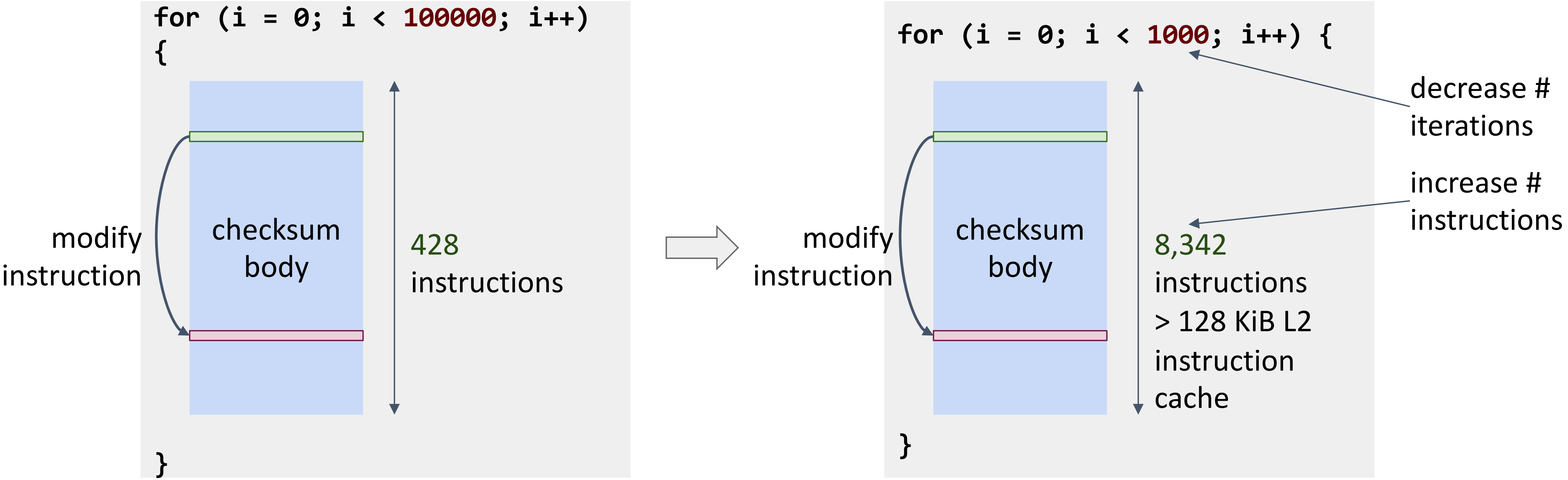
⇨ **Successful attack!**

Variants of relative placement of original and malicious VF

Seshadri et al., [1], [2]

How to defend against memory copy attacks? Self-modifying code

Self-modifying code – 1st attempt



```
for (i = 0; i < 100000; i++)
{
}
```

modify instruction

checksum body

428 instructions

```
for (i = 0; i < 1000; i++) {
}
```

modify instruction

checksum body

8,342 instructions

> 128 KiB L2 instruction cache

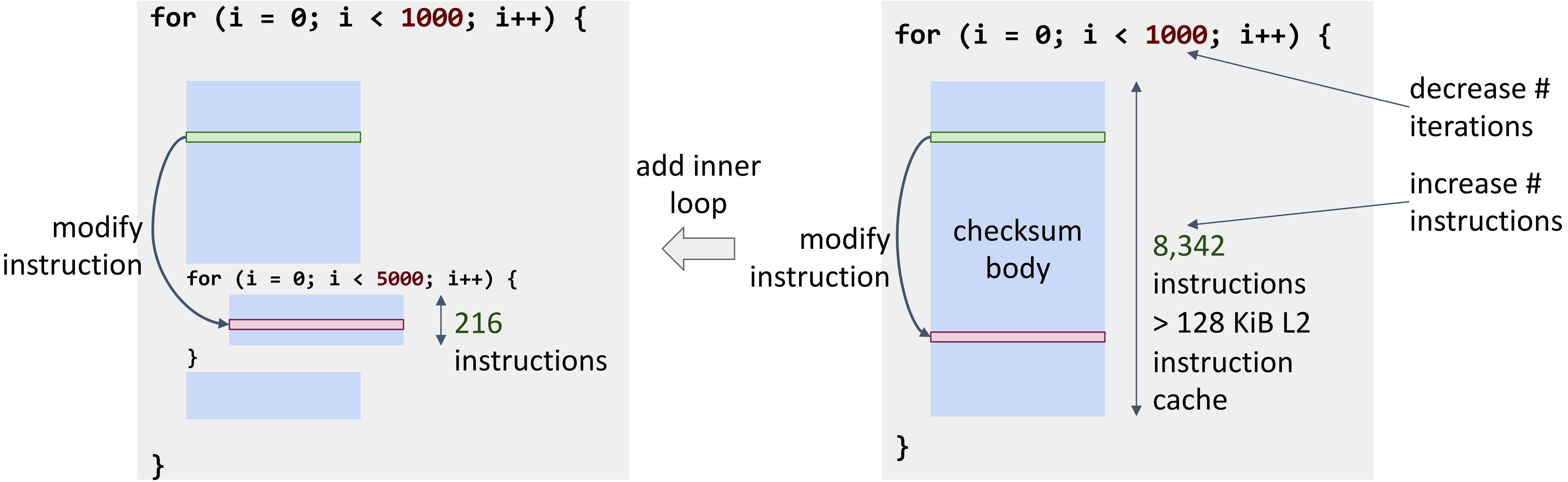
decrease # iterations

increase # instructions

Modified instruction is not visible!

Only 75% of peak performance due to overheads from cache misses

Self-modifying code



```
for (i = 0; i < 1000; i++) {
  for (i = 0; i < 5000; i++) {
  }
}
```

```
for (i = 0; i < 1000; i++) {
}
```

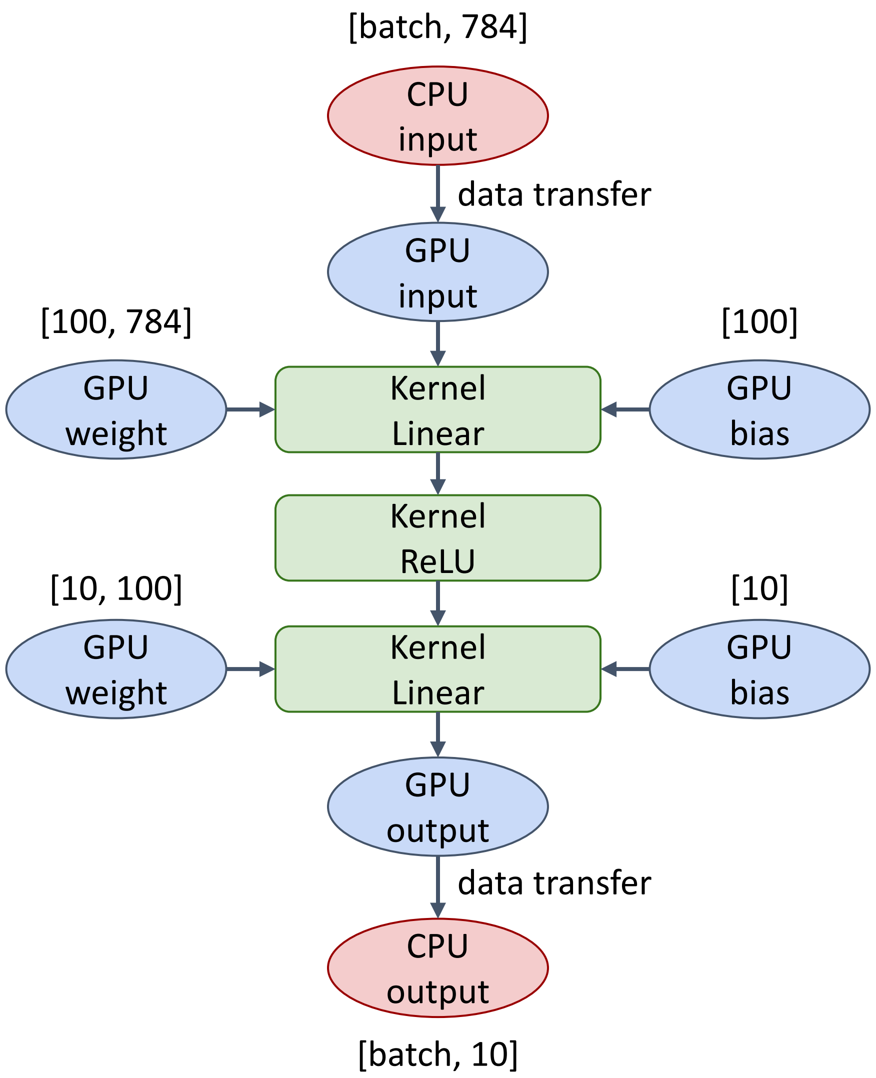
A100 time: **T_{avg} = 12.40 s** (100% of peak)

Only 75% of peak performance due to overheads from cache misses

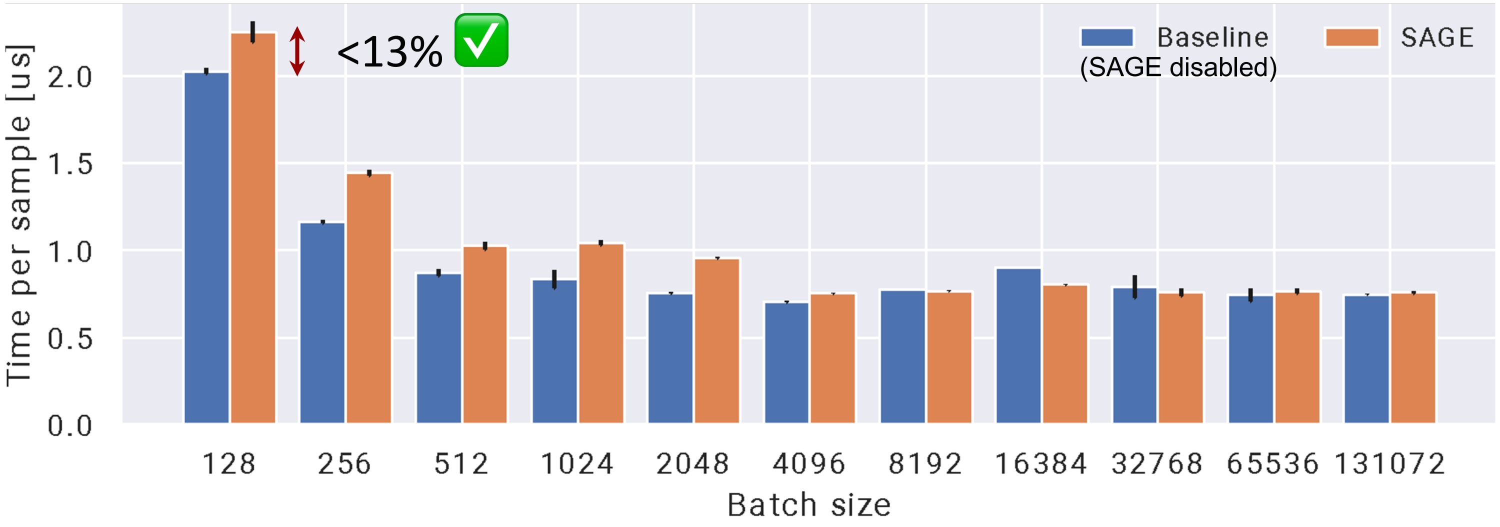
Checksum validation:
AMD EPYC 7742: **497 s**
Intel Xeon Gold 6348: **2337 s**

Quite slow!
Conjecture: GPU vendor could help get better performance

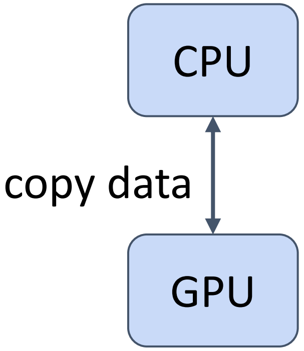
Example: Multilayer Perceptron

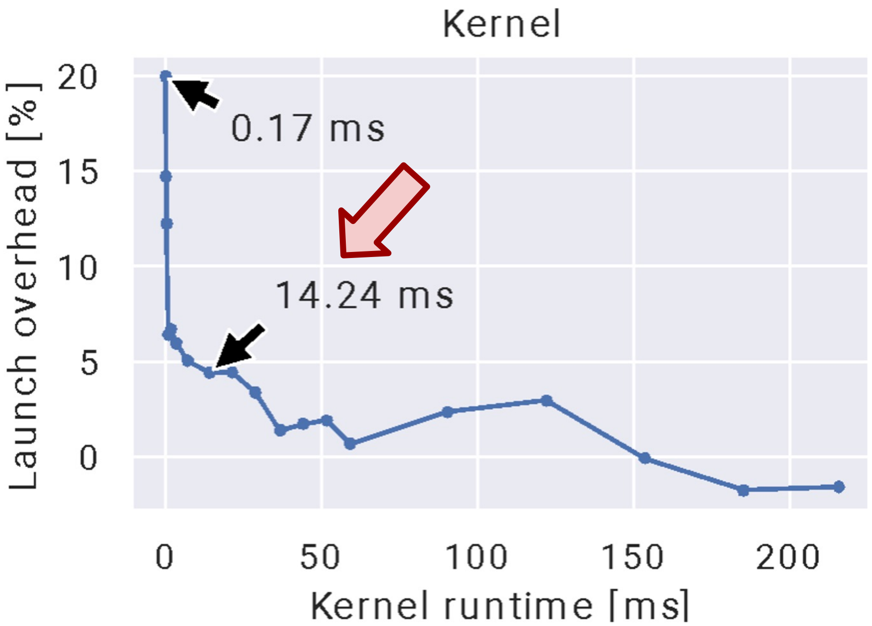


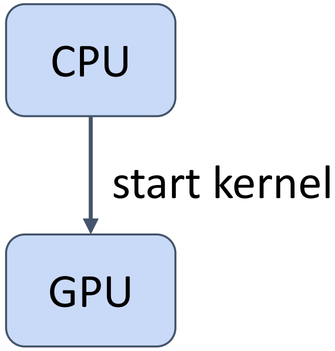
Runtime including data transfer and kernel launch overheads



Overheads of SAGE







A100 40 GB memory bandwidth: **1,555 GB/s**

SAGE overhead: $\frac{1000\ [\text{ms/s}]}{2.21\ [\text{ms/GB}]} = 452\ \text{GB/s}$ **< 30%** ✅

<5% for kernels with duration >14.24 ms ✅

Conclusion

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

https://github.com/spcl/sage

SAGE: software-only RoT establishment for GPU guaranteeing code and data integrity+secrecy even in presence of an adversary

- Concrete VF design as a proof-of-concept
  - GPU vendors natural incentives to develop improved VFs
- Technical demonstration for NVIDIA Ampere GPUs

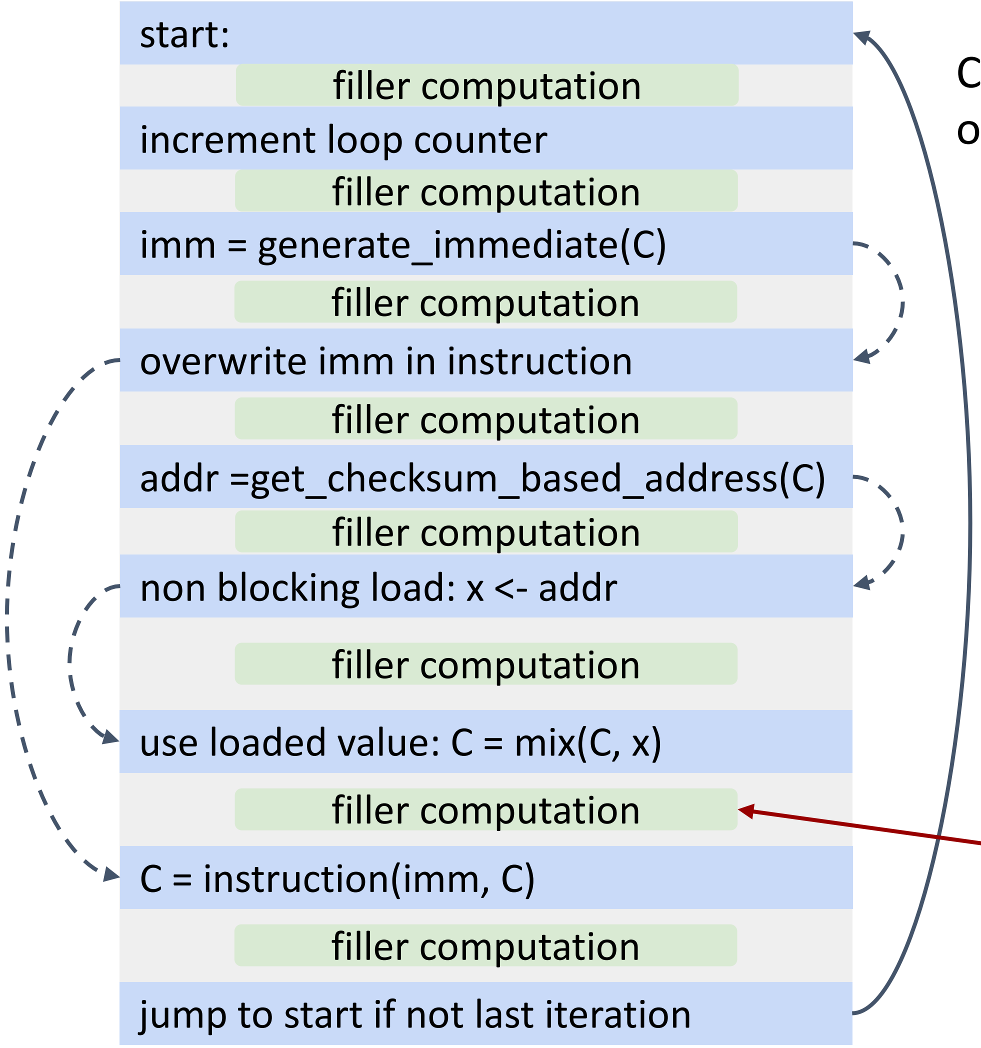
HW solutions? NVIDIA Hopper intros confidential computing

SW + HW together:

  - multiple layers of security, defense in depth
  - no reliance on embedded keys, lower TCB
  - less overall trust required

Backup

Checksum loop implementation

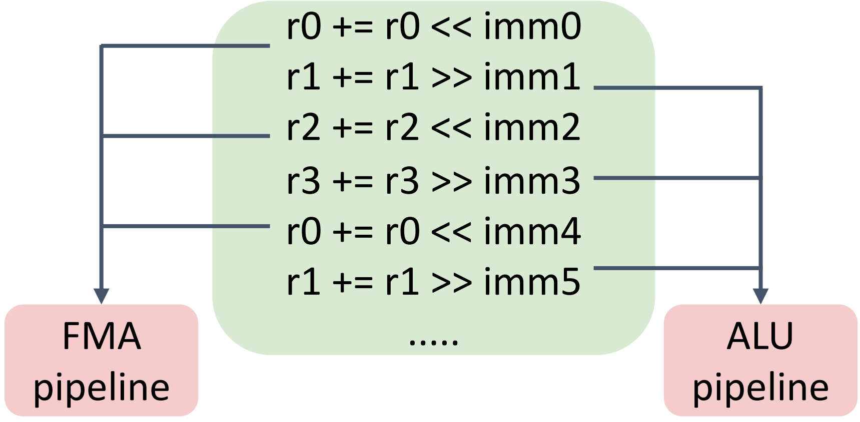


```
start:
    filler computation
increment loop counter
    filler computation
imm = generate_immediate(C)
    filler computation
overwrite imm in instruction
    filler computation
addr =get_checksum_based_address(C)
    filler computation
non blocking load: x <- addr
    filler computation
use loaded value: C = mix(C, x)
    filler computation
C = instruction(imm, C)
    filler computation
jump to start if not last iteration
```

C - checksum value on current thread

Need to hide latency of long instructions

filler computation: 1 instruction/cycle



SHF.R.U32.HI REG, RZ, IMM, REG

IMAD.U32 REG, REG, 2**IMM, REG

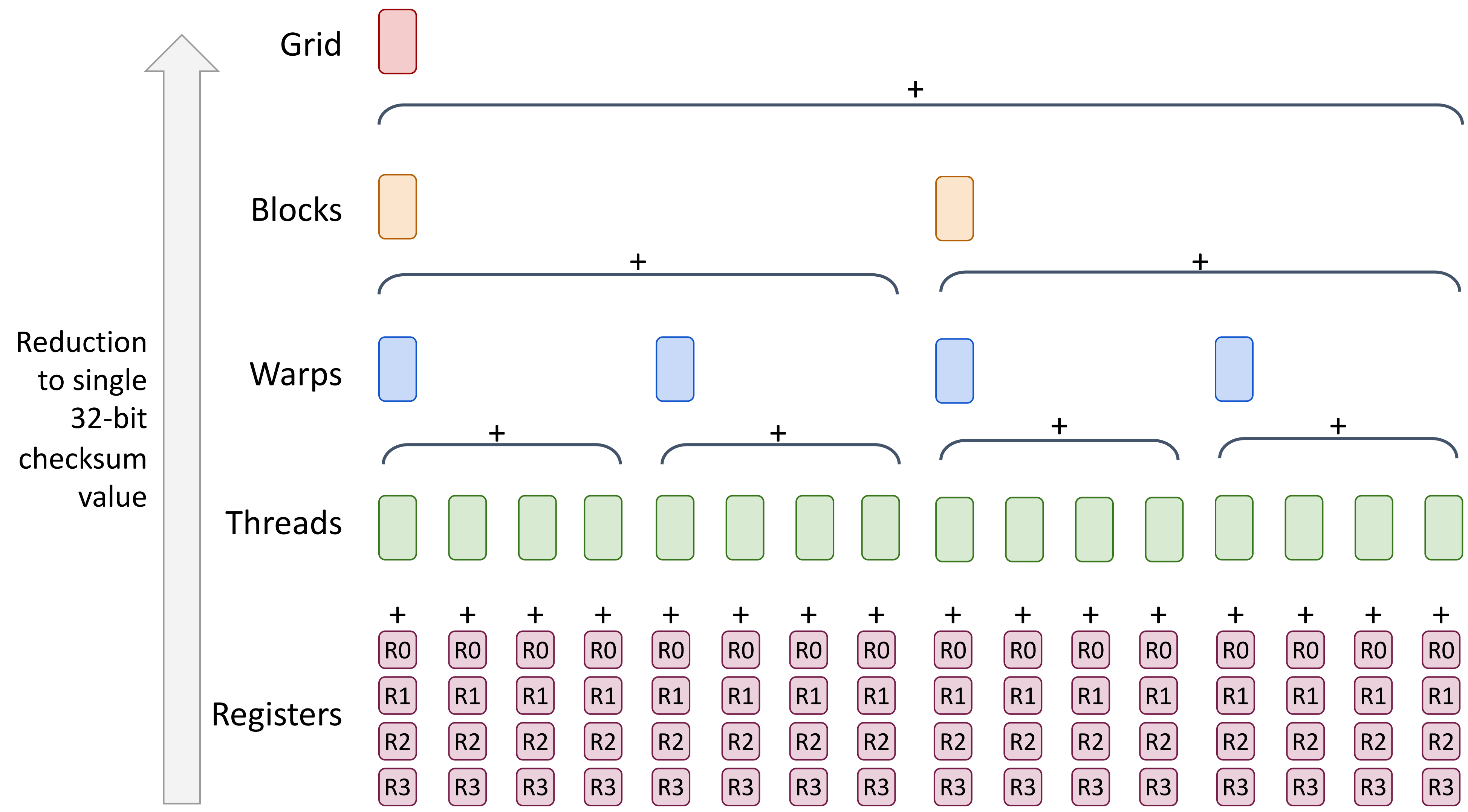
Number of iterations: **2,500,000**

Verifiable memory region: **524,288** × 32-bit

Probability that certain location is skipped:

$$\left(1-\frac{1}{524288}\right)^{2500000}=0.0085$$ ✅

Checksum epilog



grid-level reduce
in **global** memory:
atomicAdd

block-level reduce
in **shared** memory:
atomicAdd_block

warp-level reduce
in **registers**:
__shfl_xor_sync