

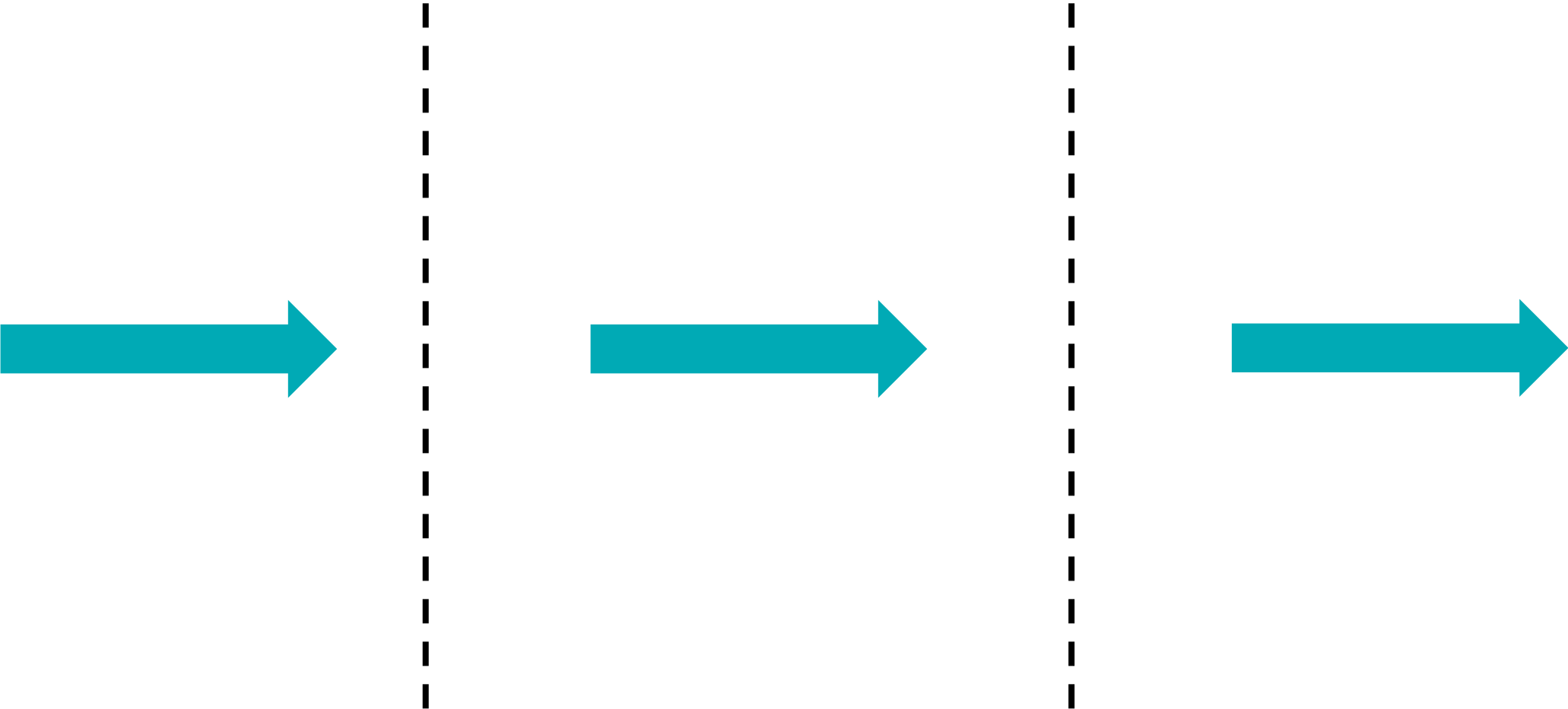
A large teal geometric shape, resembling a stylized 'A' or a series of parallel lines, occupies the left and bottom-left portions of the slide. A smaller teal parallelogram is positioned above the main shape, to the right of the center.

DYNAMIC BUFFER OVERFLOW DETECTION FOR GPGPUS

CHRIS ERB, MIKE COLLINS, JOE GREATHOUSE,
FEBRUARY 6, 2017

CONSEQUENCES OF BUFFER OVERFLOWS

DEGRADING USER EXPERIENCE, AND SECURITY RISKS



CONSEQUENCES OF BUFFER OVERFLOWS

DEGRADING USER EXPERIENCE, AND SECURITY RISKS



Data Corruption



CONSEQUENCES OF BUFFER OVERFLOWS

DEGRADING USER EXPERIENCE, AND SECURITY RISKS



Data Corruption

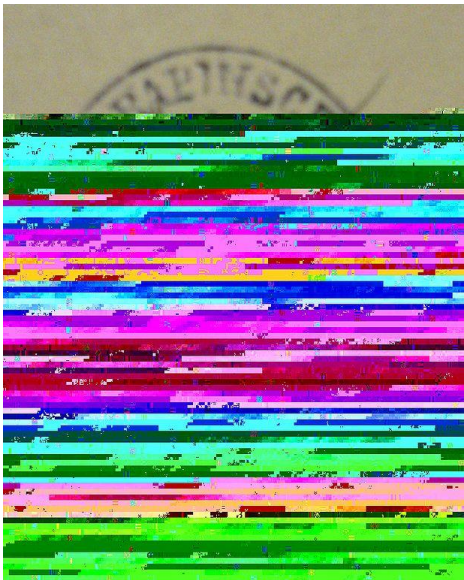


CONSEQUENCES OF BUFFER OVERFLOWS

DEGRADING USER EXPERIENCE, AND SECURITY RISKS



Data Corruption

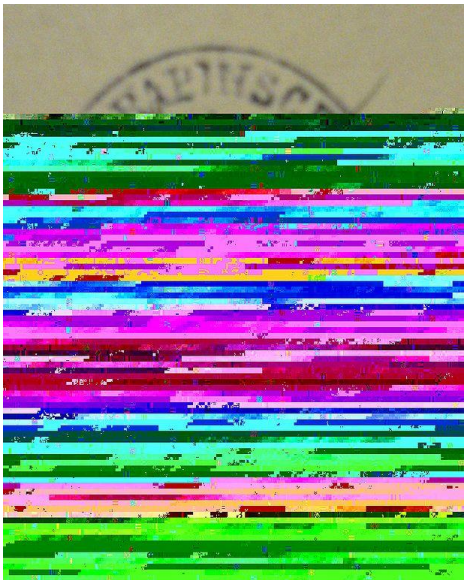


CONSEQUENCES OF BUFFER OVERFLOWS

DEGRADING USER EXPERIENCE, AND SECURITY RISKS



Data Corruption



Segmentation Faults

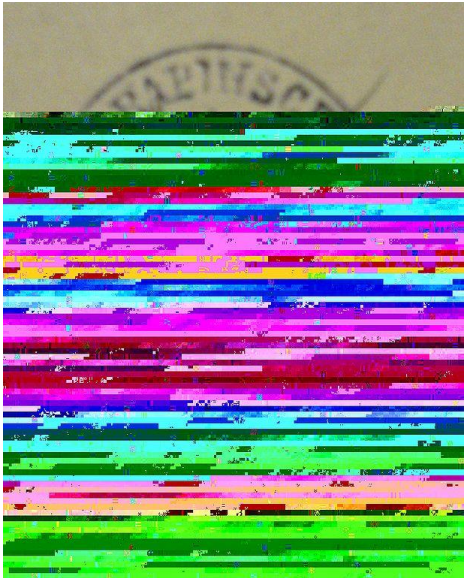


CONSEQUENCES OF BUFFER OVERFLOWS

DEGRADING USER EXPERIENCE, AND SECURITY RISKS



Data Corruption



Segmentation Faults

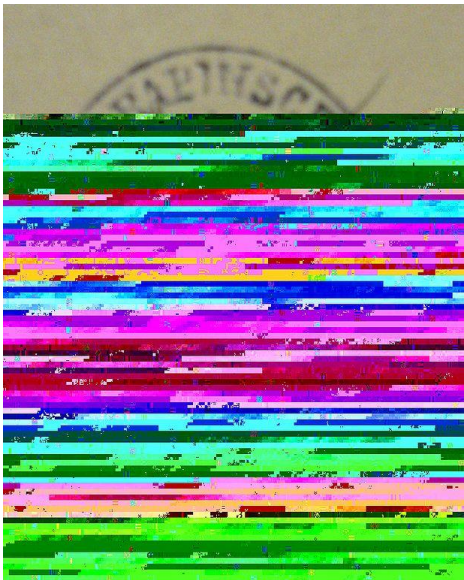


CONSEQUENCES OF BUFFER OVERFLOWS

DEGRADING USER EXPERIENCE, AND SECURITY RISKS



Data Corruption



Segmentation Faults

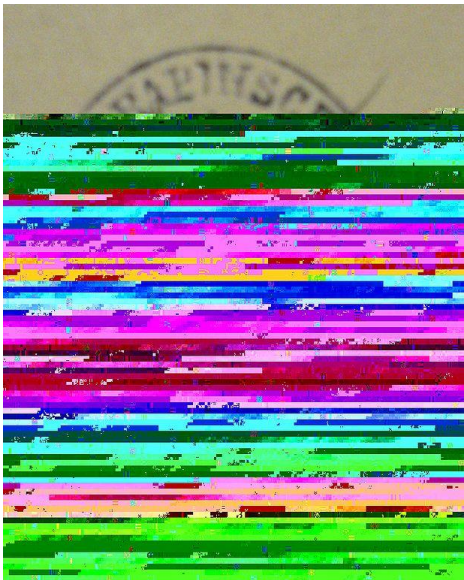


CONSEQUENCES OF BUFFER OVERFLOWS

DEGRADING USER EXPERIENCE, AND SECURITY RISKS



Data Corruption



Segmentation Faults



Altered Control Flow (Security Subversion)

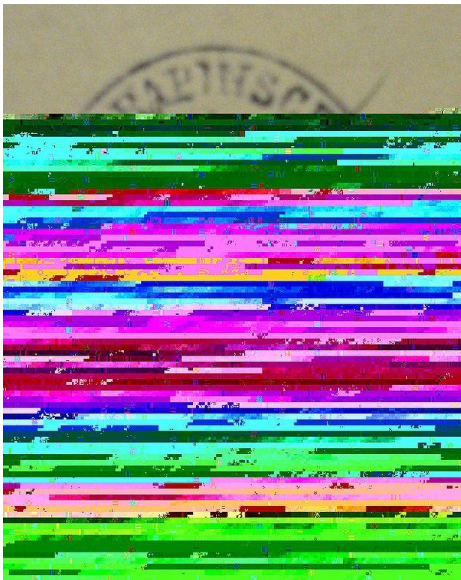


CONSEQUENCES OF BUFFER OVERFLOWS

DEGRADING USER EXPERIENCE, AND SECURITY RISKS



Data Corruption



Segmentation Faults



Altered Control Flow
(Security Subversion)

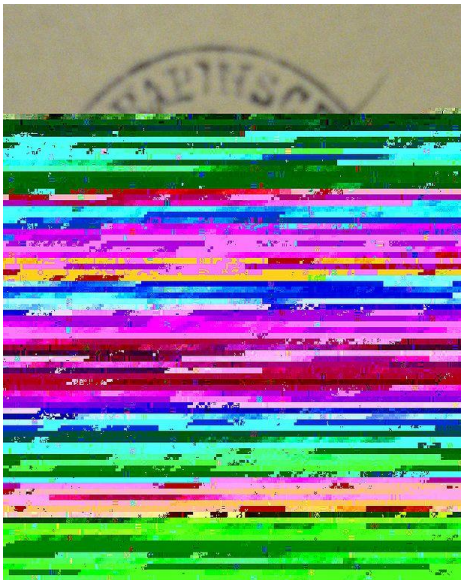


CONSEQUENCES OF BUFFER OVERFLOWS

DEGRADING USER EXPERIENCE, AND SECURITY RISKS



Data Corruption



Segmentation Faults



Altered Control Flow (Security Subversion)



RISK ASSESSMENT —

Elegant 0-day unicorn underscores
“serious concerns” about Linux security

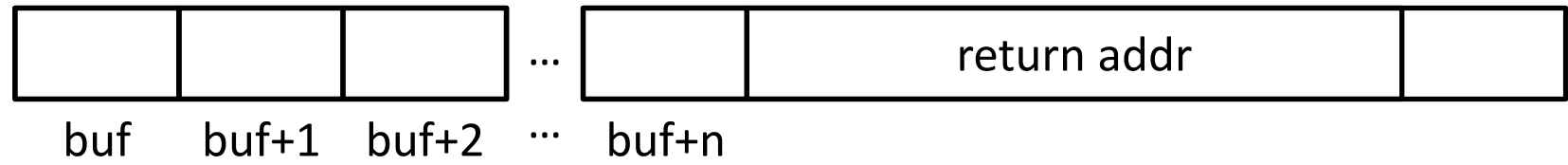
Scriptless exploit bypasses state-of-the-art protections baked into the OS.

DAN GOODIN - 11/22/2016, 3:48 PM

BACKGROUND: NORMAL BUFFER FILL



- ▲ buf[n+1]
- ▲ memcpy(buf, src, n+1)

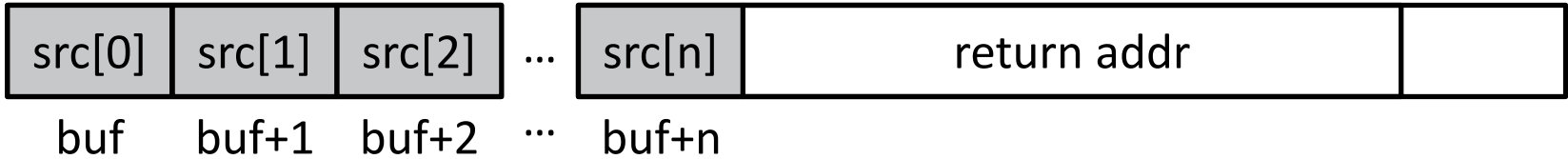


BACKGROUND: NORMAL BUFFER FILL



▲ buf[n+1]

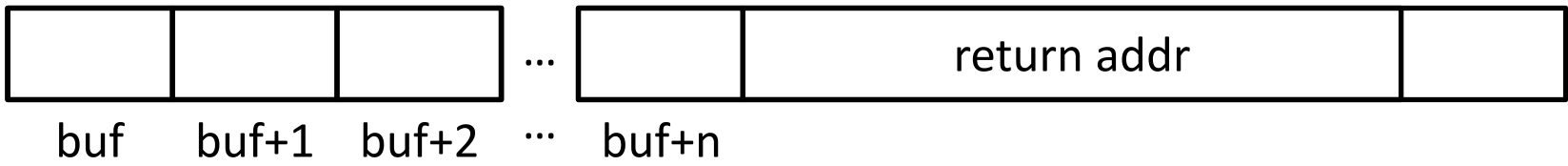
▲ memcpy(buf, src, n+1)



BACKGROUND: BUFFER OVERFLOW



▲ buf[n+1]
▲ memcpy(buf, src, n+5)

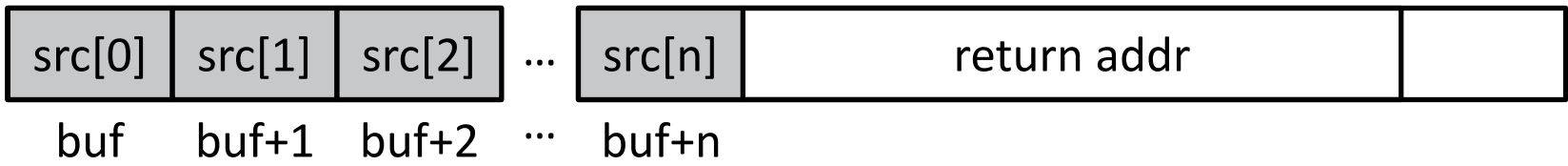


BACKGROUND: BUFFER OVERFLOW



▲ buf[n+1]

▲ memcpy(buf, src, n+5)

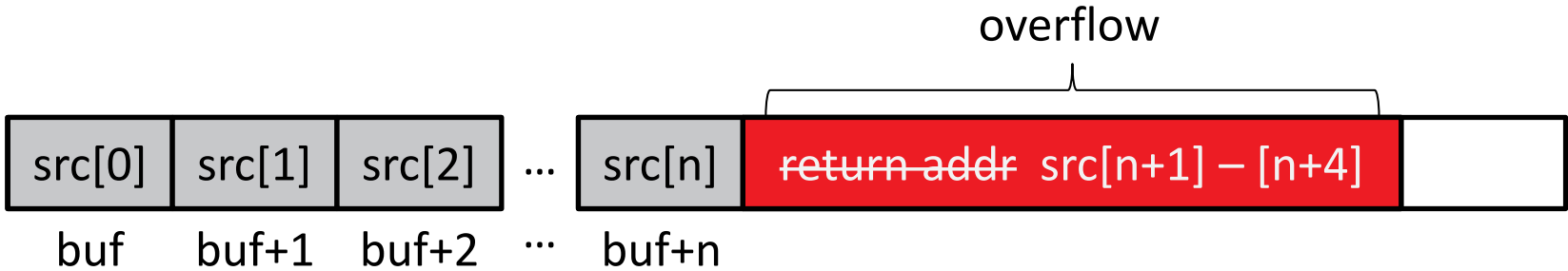


BACKGROUND: BUFFER OVERFLOW



▲ buf[n+1]

▲ memcpy(buf, src, n+5)



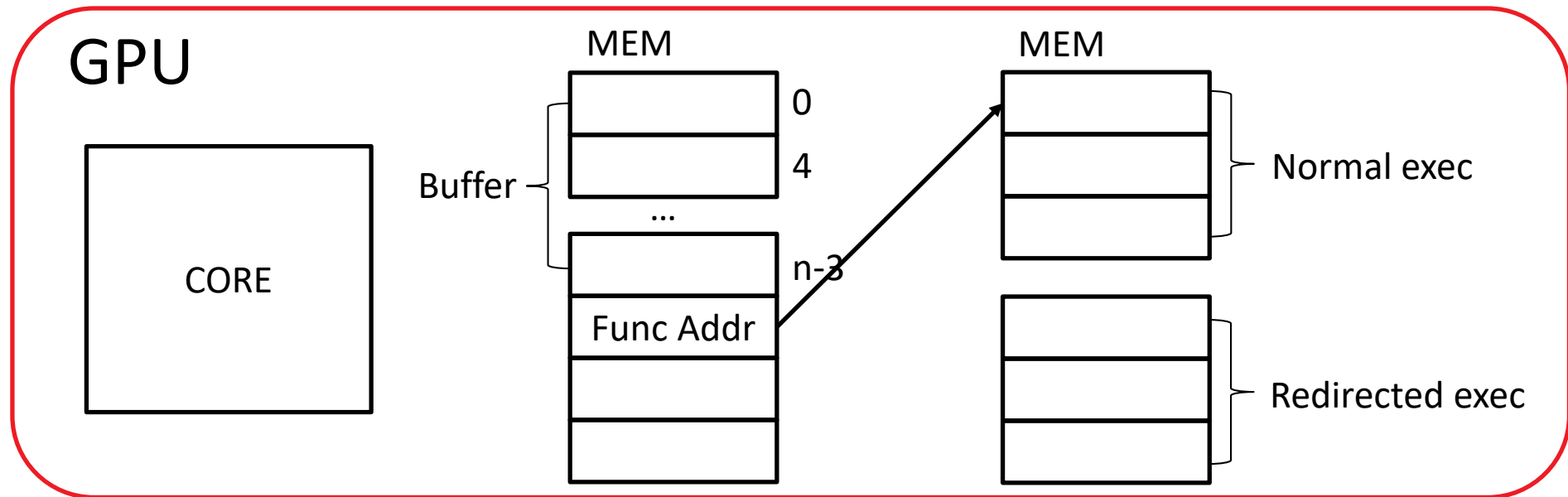
GPU BUFFERS ALSO OVERFLOW



REMOTE CODE EXECUTION ON GPU

▲ Overflows on GPU can cause remote GPU code execution

- A. Miele. *Buffer Overflow Vulnerabilities in CUDA: A Preliminary Analysis*.
- B. Di, J. Sun, and H. Chen. *A Study of Overflow Vulnerabilities on GPUs*.



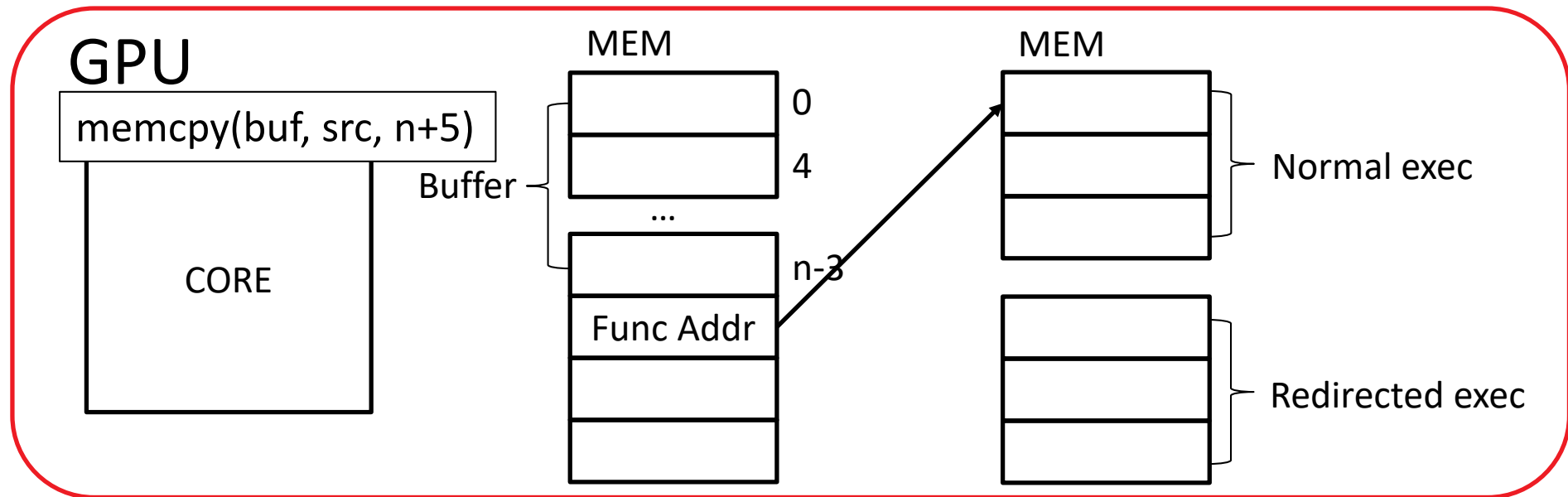
GPU BUFFERS ALSO OVERFLOW



REMOTE CODE EXECUTION ON GPU

▲ Overflows on GPU can cause remote GPU code execution

- A. Miele. *Buffer Overflow Vulnerabilities in CUDA: A Preliminary Analysis*.
- B. Di, J. Sun, and H. Chen. *A Study of Overflow Vulnerabilities on GPUs*.



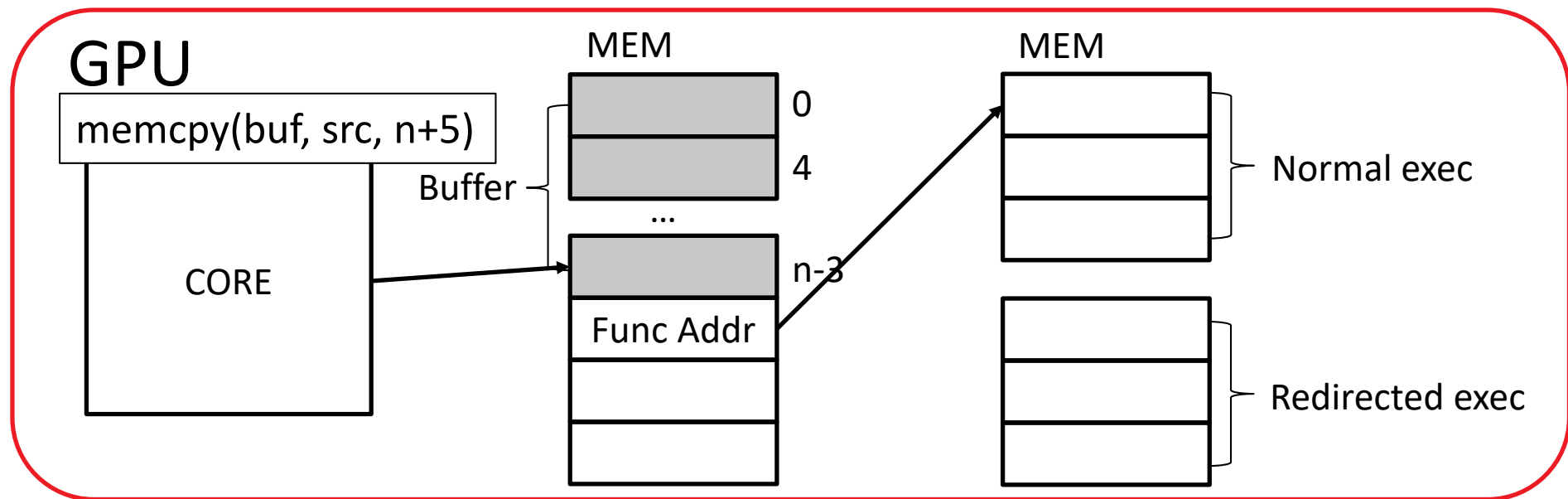
GPU BUFFERS ALSO OVERFLOW



REMOTE CODE EXECUTION ON GPU

▲ Overflows on GPU can cause remote GPU code execution

- A. Miele. *Buffer Overflow Vulnerabilities in CUDA: A Preliminary Analysis*.
- B. Di, J. Sun, and H. Chen. *A Study of Overflow Vulnerabilities on GPUs*.



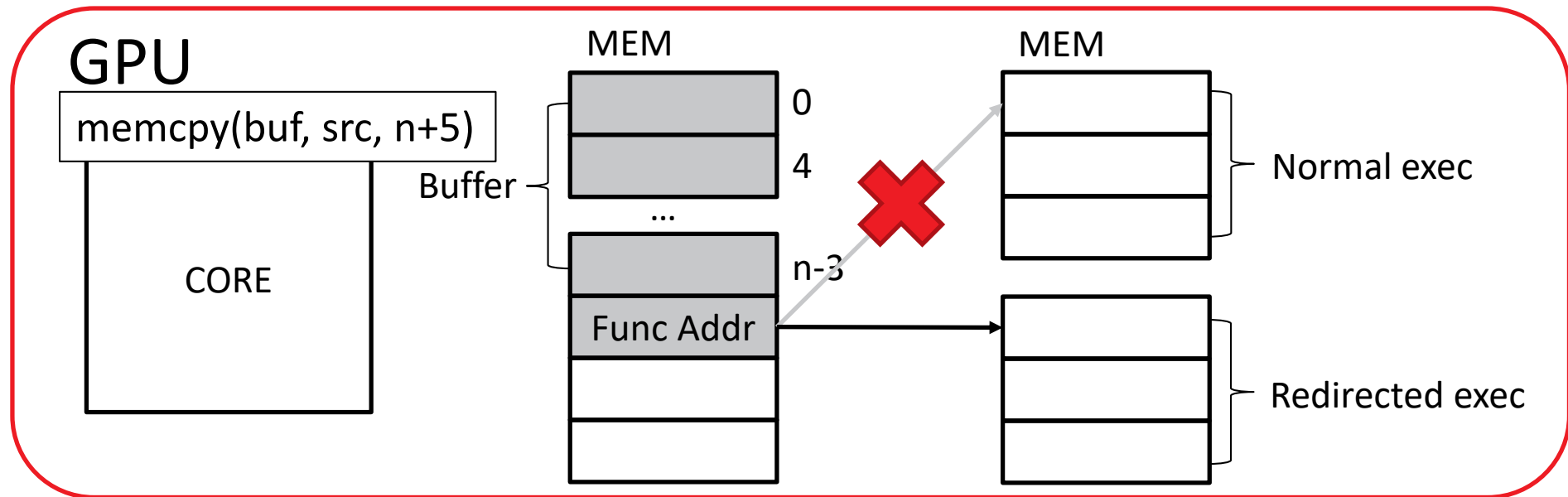
GPU BUFFERS ALSO OVERFLOW



REMOTE CODE EXECUTION ON GPU

▲ Overflows on GPU can cause remote GPU code execution

- A. Miele. *Buffer Overflow Vulnerabilities in CUDA: A Preliminary Analysis*.
- B. Di, J. Sun, and H. Chen. *A Study of Overflow Vulnerabilities on GPUs*.



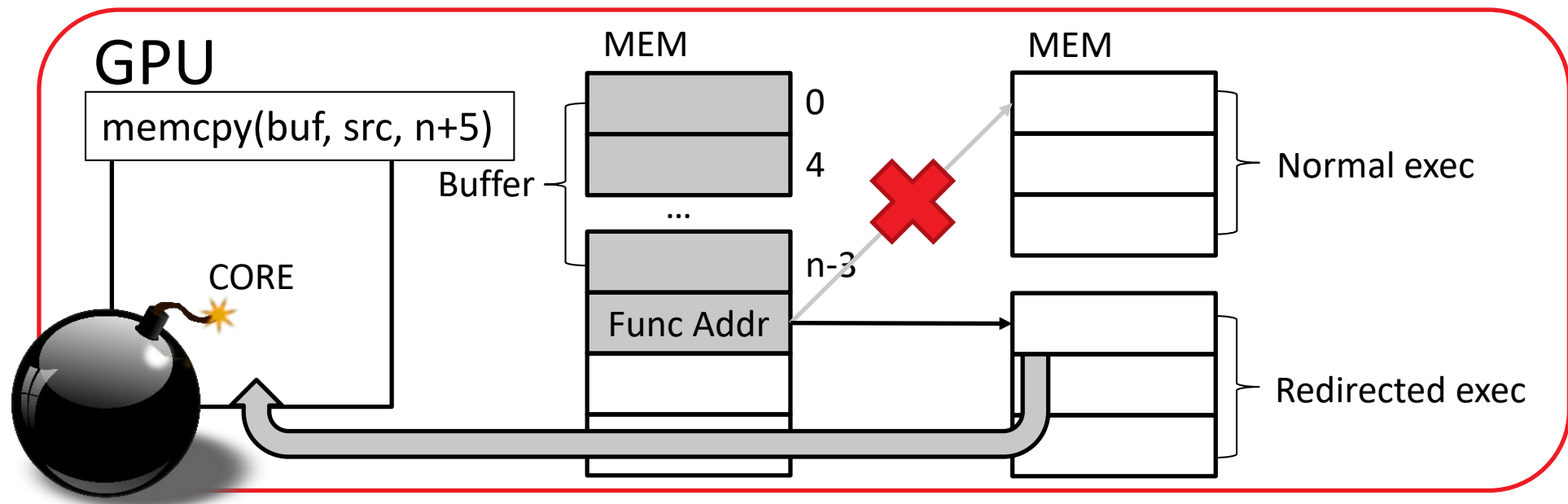
GPU BUFFERS ALSO OVERFLOW



REMOTE CODE EXECUTION ON GPU

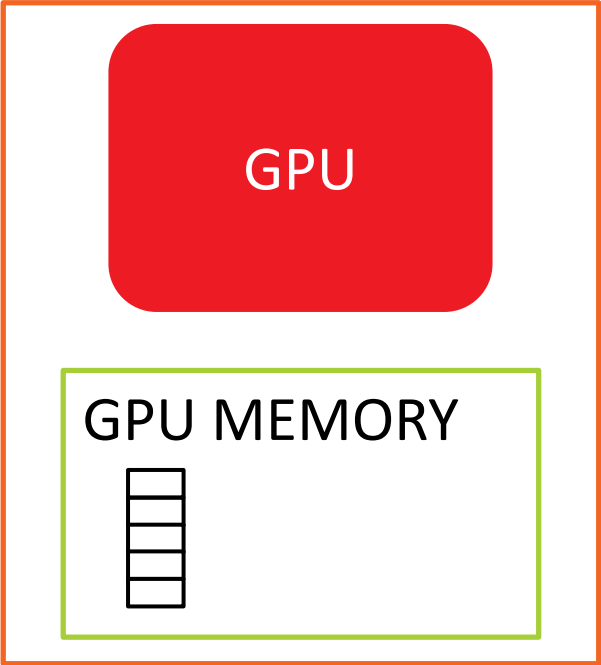
▲ Overflows on GPU can cause remote GPU code execution

- A. Miele. *Buffer Overflow Vulnerabilities in CUDA: A Preliminary Analysis*.
- B. Di, J. Sun, and H. Chen. *A Study of Overflow Vulnerabilities on GPUs*.



GPU BUFFERS ALSO OVERFLOW

SHARED MEMORY CORRUPTION

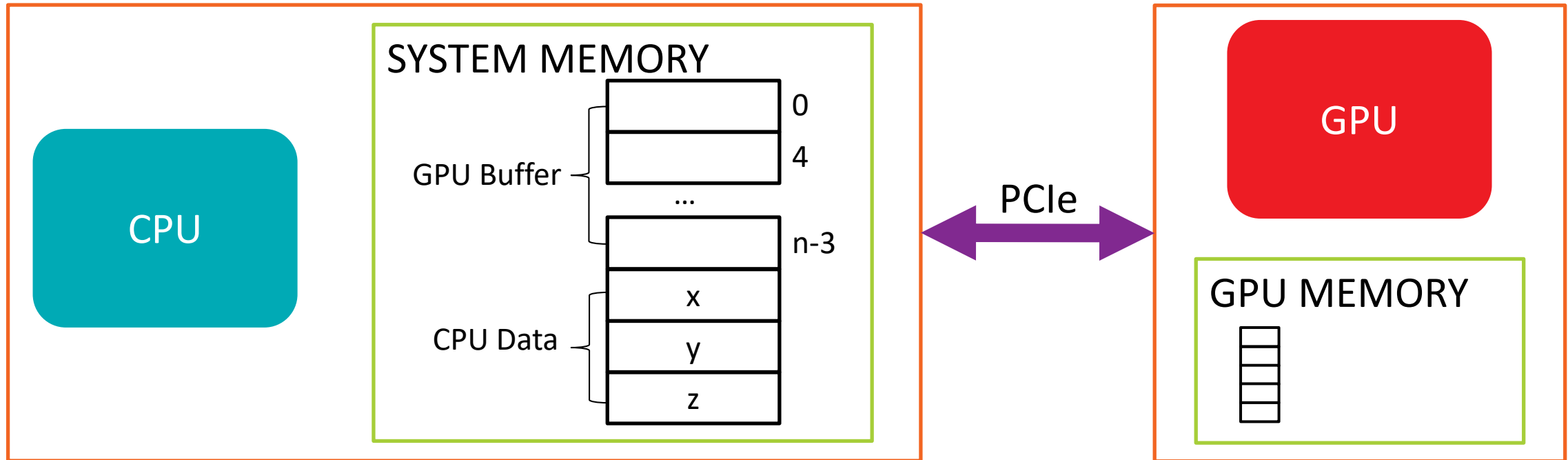


GPU BUFFERS ALSO OVERFLOW

SHARED MEMORY CORRUPTION



- GPU can overflow buffers in system memory
 - Over Interconnects like PCIe[®]

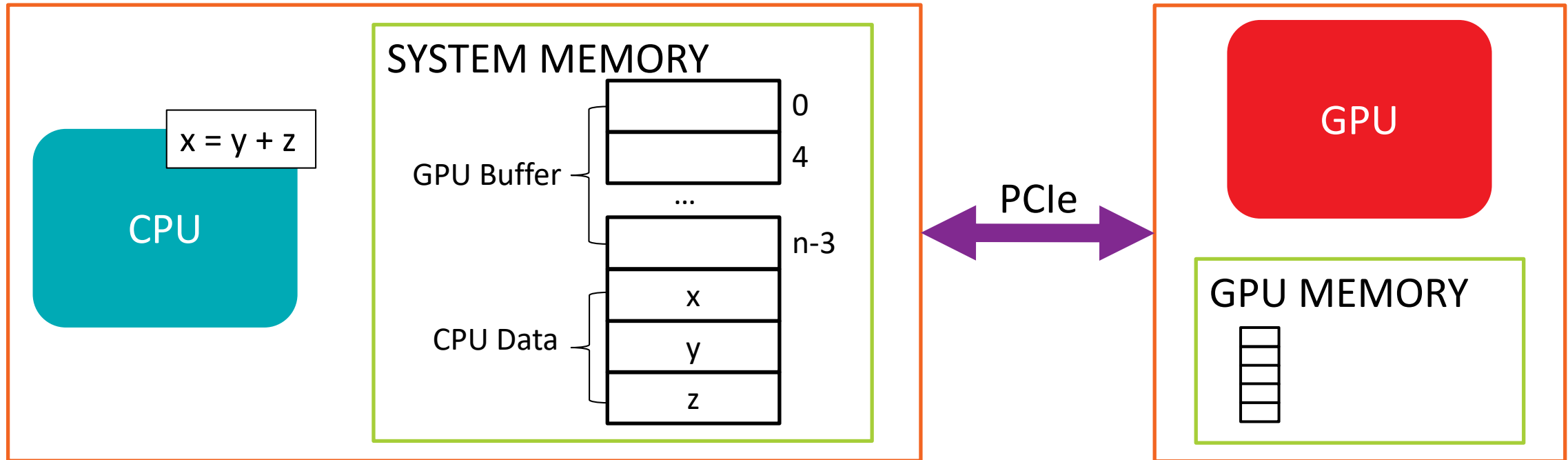


GPU BUFFERS ALSO OVERFLOW

SHARED MEMORY CORRUPTION



- GPU can overflow buffers in system memory
 - Over Interconnects like PCIe[®]

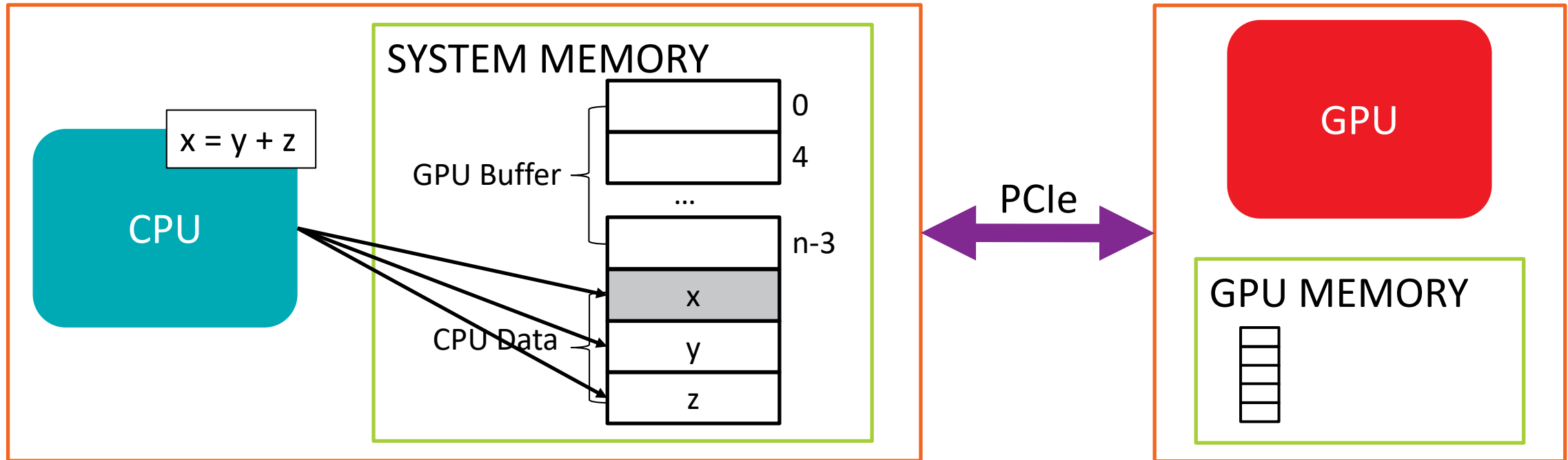


GPU BUFFERS ALSO OVERFLOW

SHARED MEMORY CORRUPTION



- GPU can overflow buffers in system memory
 - Over Interconnects like PCIe[®]

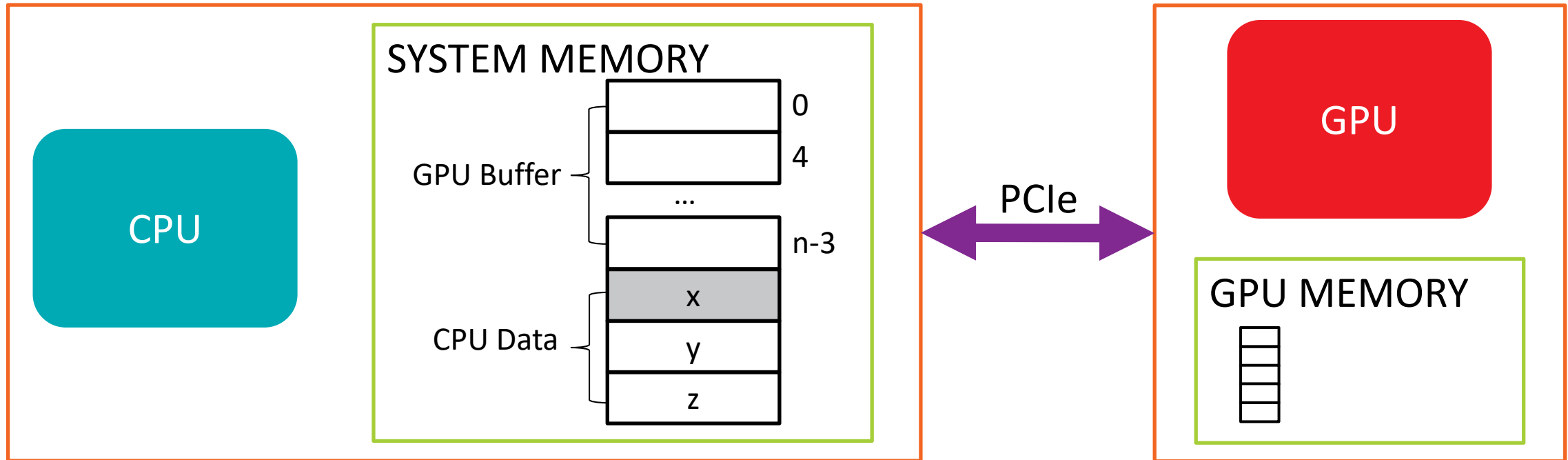


GPU BUFFERS ALSO OVERFLOW

SHARED MEMORY CORRUPTION



- GPU can overflow buffers in system memory
 - Over Interconnects like PCIe[®]

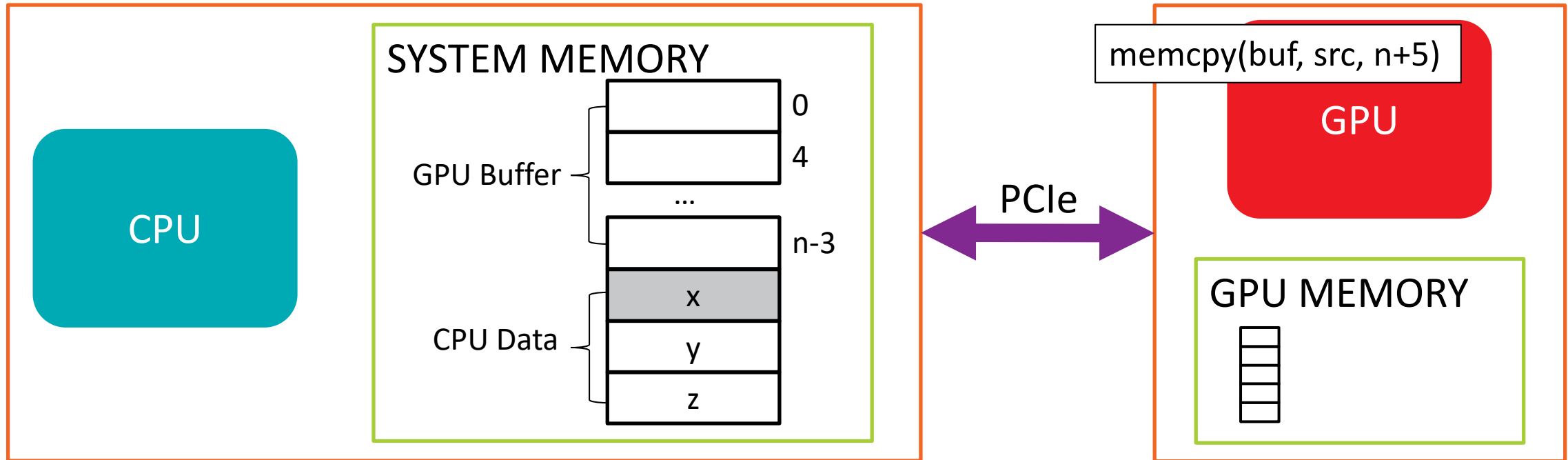


GPU BUFFERS ALSO OVERFLOW

SHARED MEMORY CORRUPTION



- GPU can overflow buffers in system memory
 - Over Interconnects like PCIe[®]

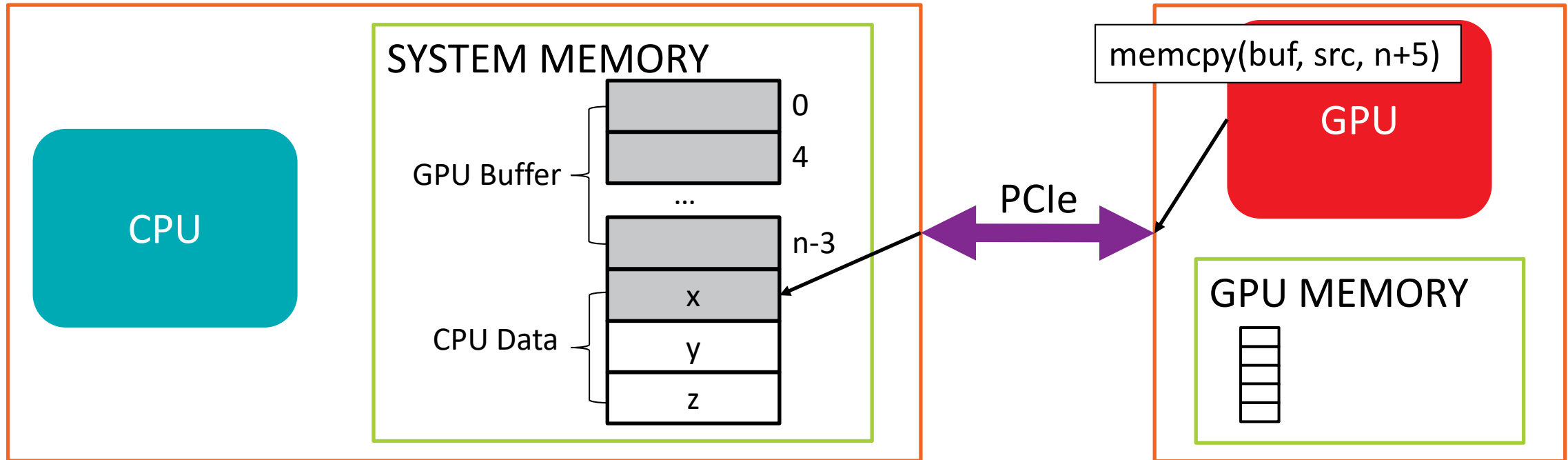


GPU BUFFERS ALSO OVERFLOW

SHARED MEMORY CORRUPTION



- GPU can overflow buffers in system memory
 - Over Interconnects like PCIe[®]

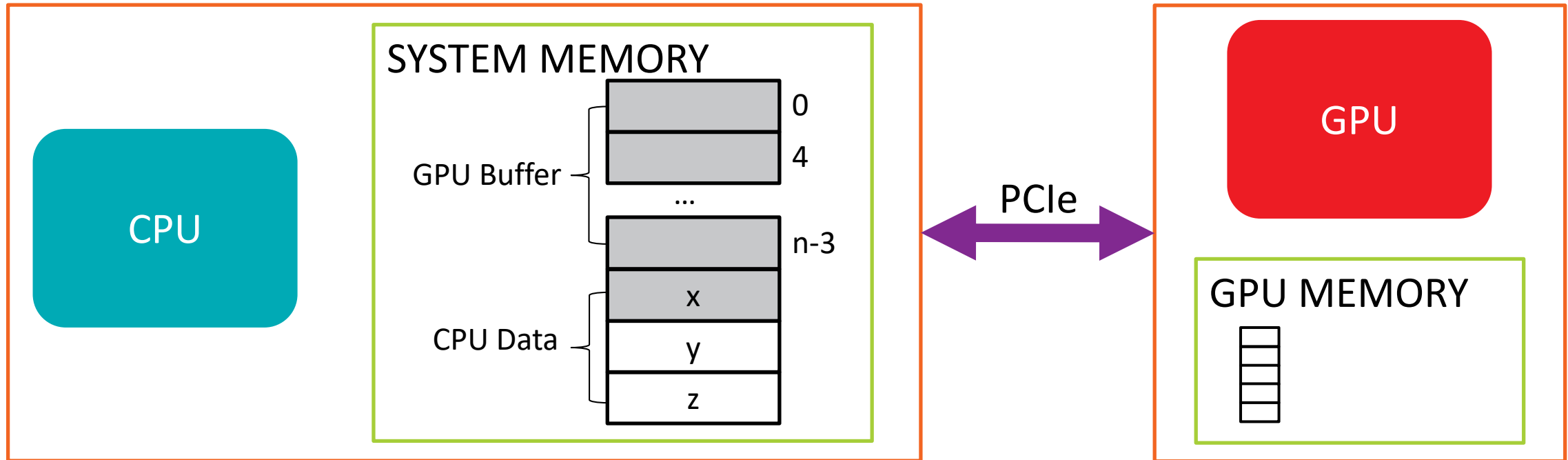


GPU BUFFERS ALSO OVERFLOW

SHARED MEMORY CORRUPTION



- GPU can overflow buffers in system memory
 - Over Interconnects like PCIe[®]

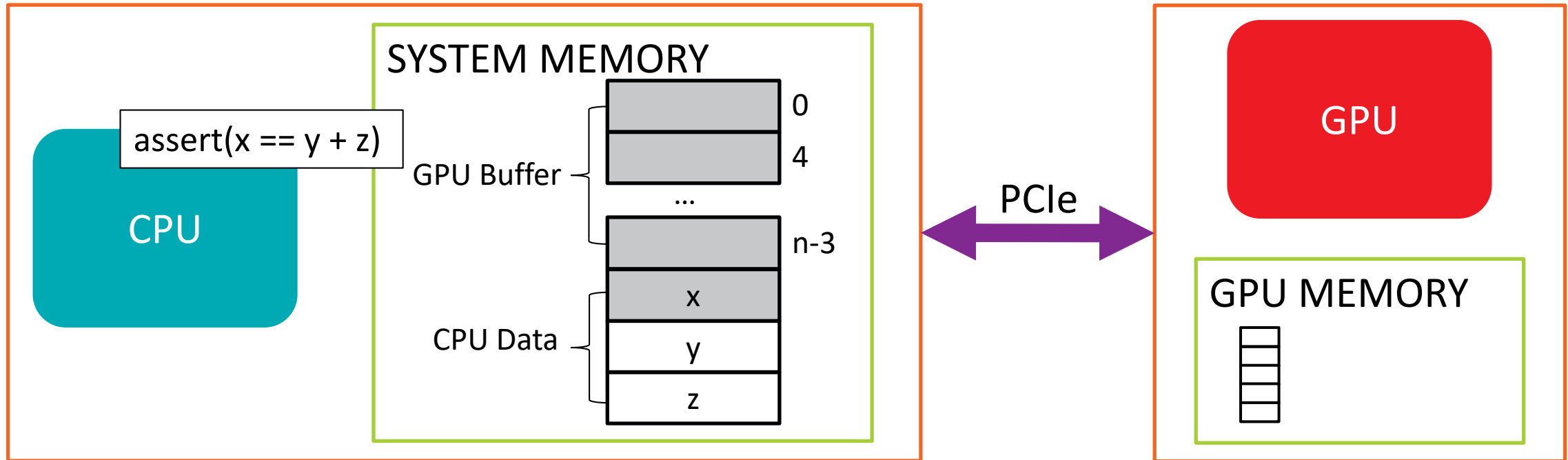


GPU BUFFERS ALSO OVERFLOW

SHARED MEMORY CORRUPTION



- GPU can overflow buffers in system memory
 - Over Interconnects like PCIe[®]

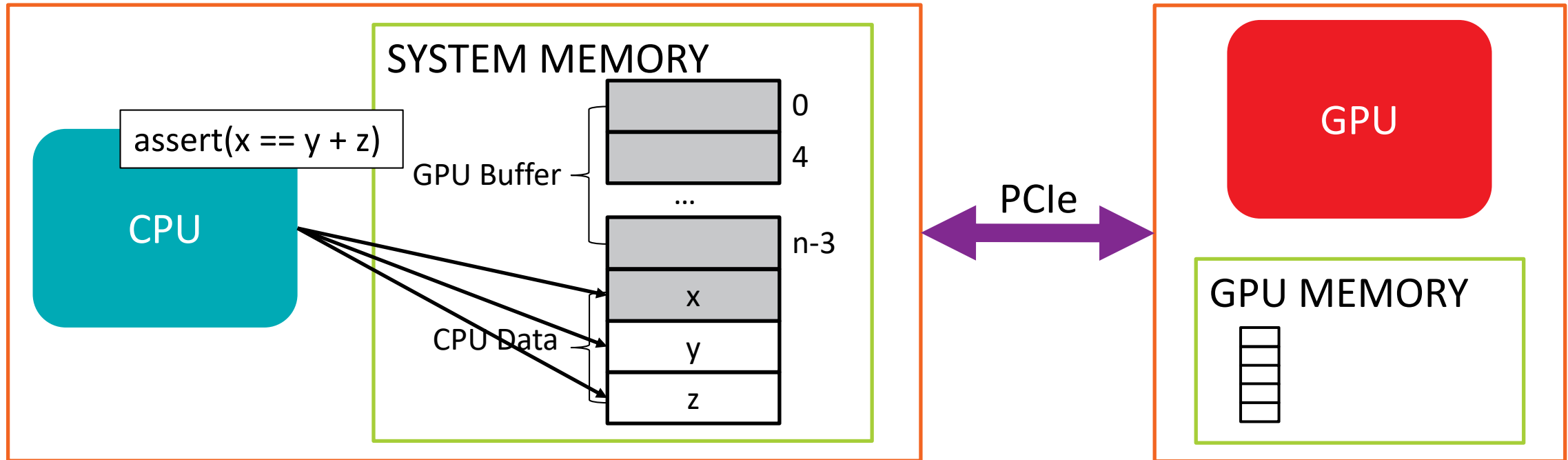


GPU BUFFERS ALSO OVERFLOW

SHARED MEMORY CORRUPTION



- GPU can overflow buffers in system memory
 - Over Interconnects like PCIe[®]

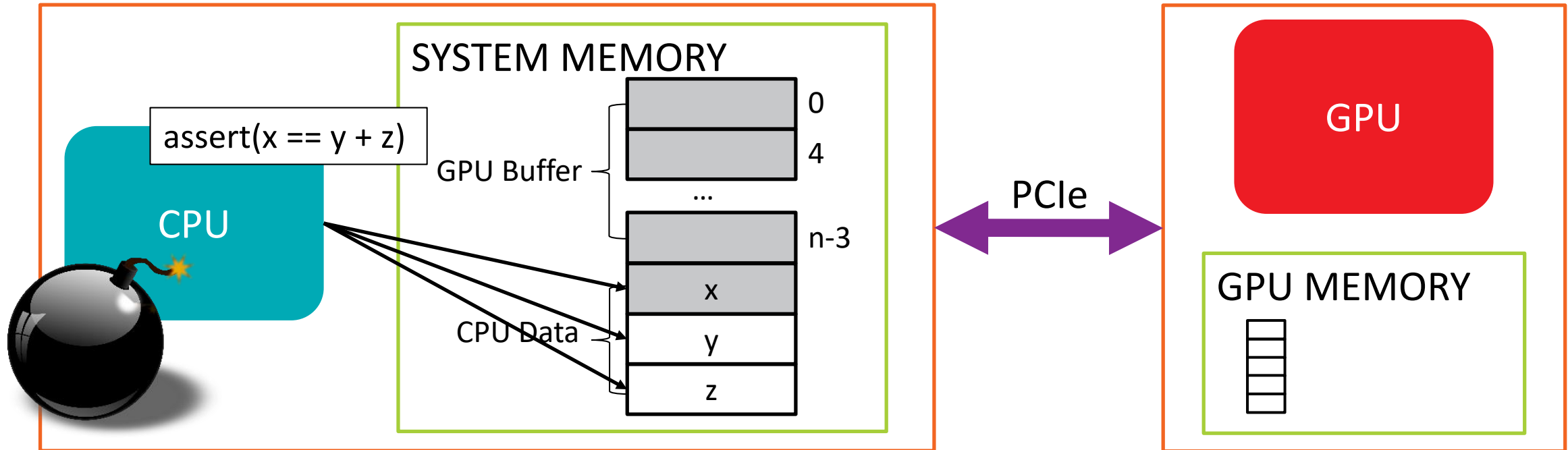


GPU BUFFERS ALSO OVERFLOW

SHARED MEMORY CORRUPTION



- GPU can overflow buffers in system memory
 - Over Interconnects like PCIe[®]

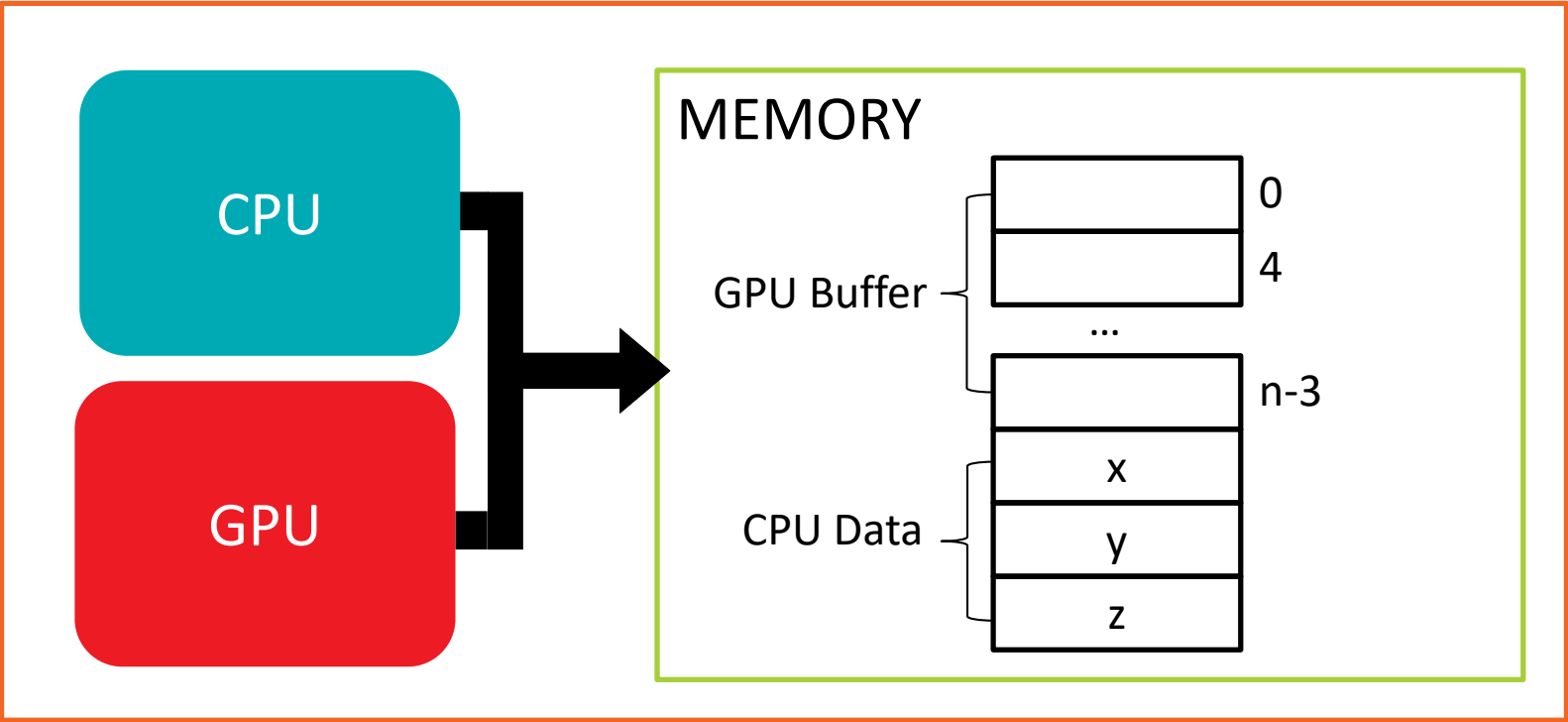


GPU BUFFERS ALSO OVERFLOW

SHARED MEMORY CORRUPTION



▲ CPU and GPU as part of the same package

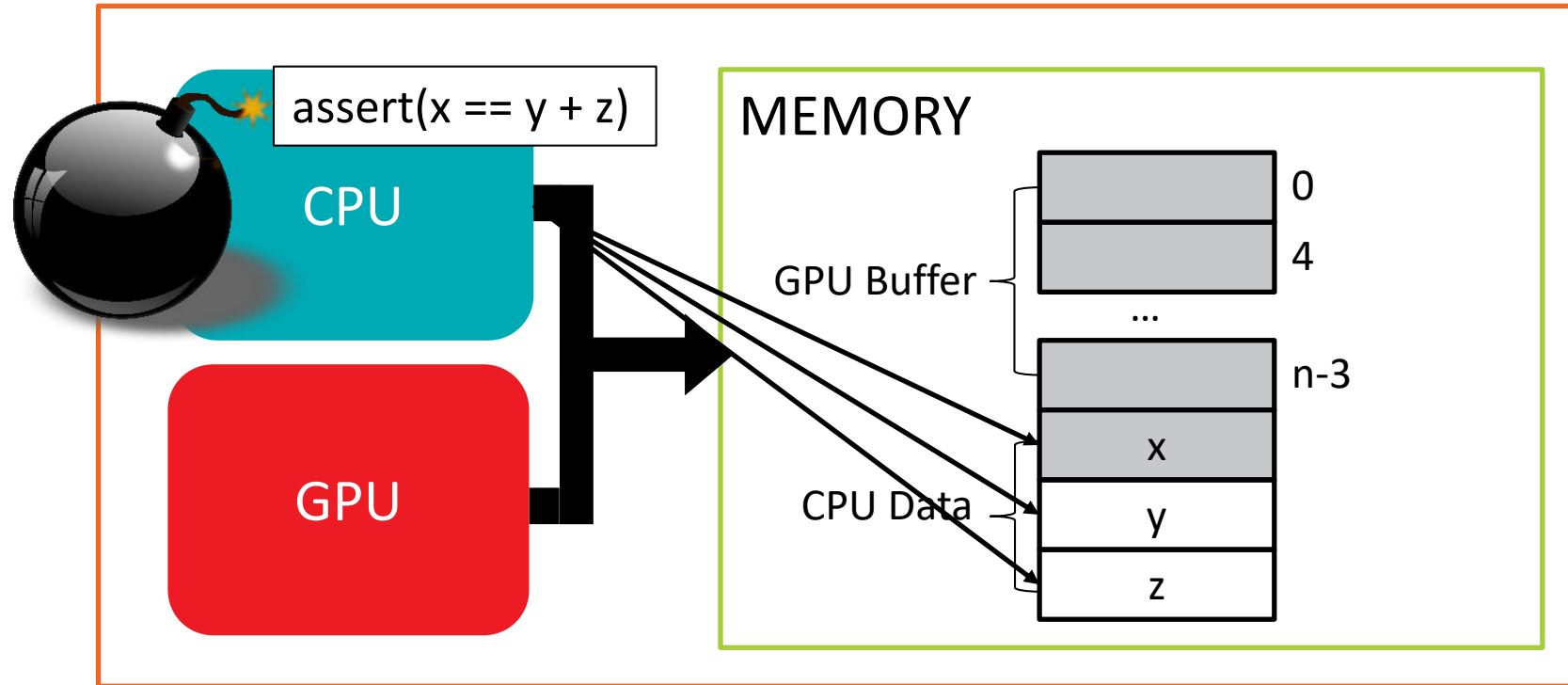


GPU BUFFERS ALSO OVERFLOW

SHARED MEMORY CORRUPTION



- ▲ CPU and GPU as part of the same package

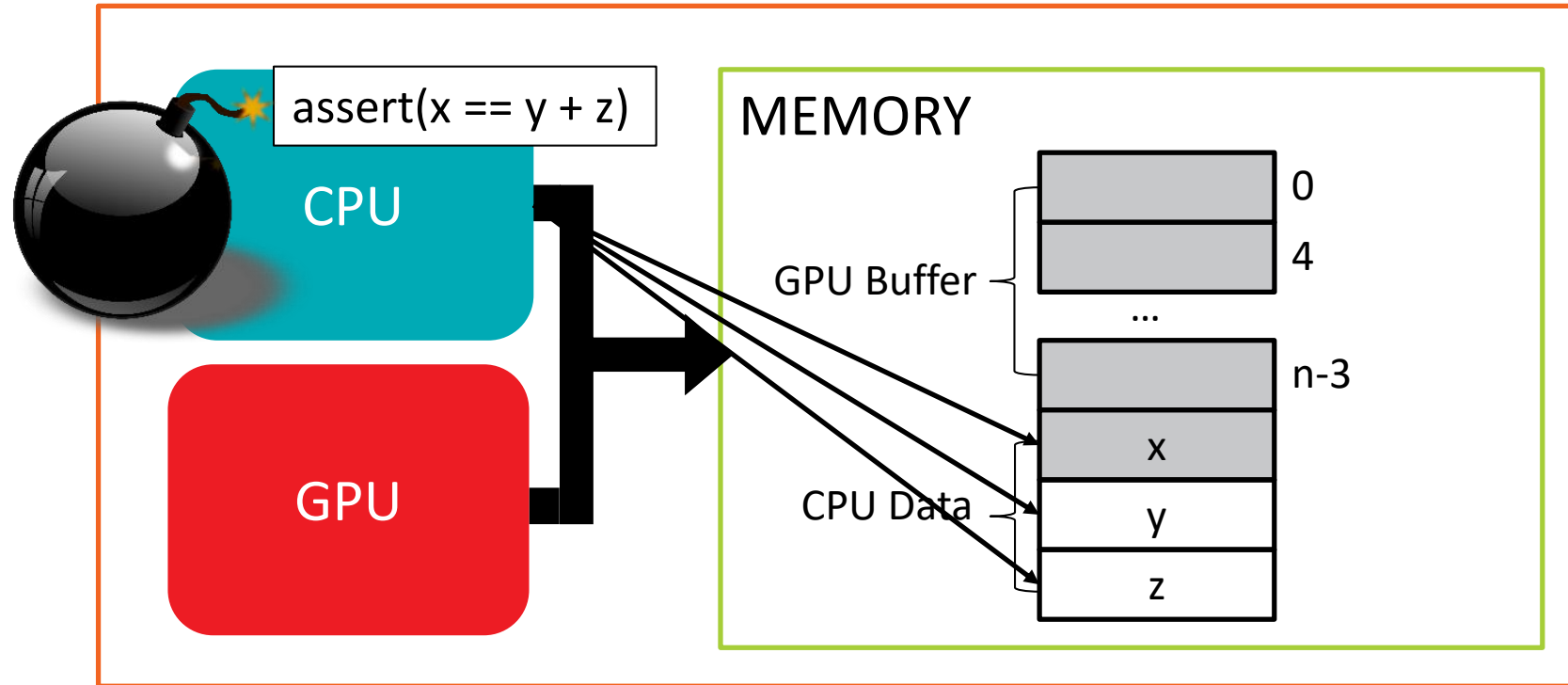


GPU BUFFERS ALSO OVERFLOW

SHARED MEMORY CORRUPTION



- ▲ CPU and GPU as part of the same package
 - Every GPU buffer overflow may affect CPU data



GOALS

BUILDING CLARMOR



▲ Software tool to detect buffer overflows caused by GPU

▲ Runnable with most OpenCL™ applications

▲ Low runtime overhead

GOALS

BUILDING CLARMOR



- ▲ Software tool to detect buffer overflows caused by GPU

 - clARMOR found 13 GPU buffer overflows in 7 programs

- ▲ Runnable with most OpenCL™ applications

- ▲ Low runtime overhead

GOALS

BUILDING CLARMOR



- ▲ Software tool to detect buffer overflows caused by GPU
 - clARMOR found 13 GPU buffer overflows in 7 programs
- ▲ Runnable with most OpenCL™ applications
 - Tested for GPU and CPU device types from multiple vendors
- ▲ Low runtime overhead

GOALS

BUILDING CLARMOR



- ▲ Software tool to detect buffer overflows caused by GPU
 - clARMOR found 13 GPU buffer overflows in 7 programs
- ▲ Runnable with most OpenCL™ applications
 - Tested for GPU and CPU device types from multiple vendors
- ▲ Low runtime overhead
 - 14% overhead across 175 applications in 16 GPU benchmark suites

GOALS

BUILDING CLARMOR



- ▲ Software tool to detect buffer overflows caused by GPU
 - clARMOR found 13 GPU buffer overflows in 7 programs
- ▲ Runnable with most OpenCL™ applications
 - Tested for GPU and CPU device types from multiple vendors
- ▲ Low runtime overhead
 - 14% overhead across 175 applications in 16 GPU benchmark suites

GOALS

BUILDING CLARMOR



- ▲ Software tool to detect buffer overflows caused by GPU
 - clARMOR found 13 GPU buffer overflows in 7 programs
- ▲ Runnable with most OpenCL™ applications
 - Tested for GPU and CPU device types from multiple vendors
- ▲ Low runtime overhead
 - 14% overhead across 175 applications in 16 GPU benchmark suites

BUFFER OVERFLOW DETECTION METHODOLOGY

CANARY-BASED DETECTION



BUFFER OVERFLOW DETECTION METHODOLOGY

CANARY-BASED DETECTION



BUFFER OVERFLOW DETECTION METHODOLOGY

CANARY-BASED DETECTION

▲ Inserting known values around a protected region.



BUFFER OVERFLOW DETECTION METHODOLOGY

CANARY-BASED DETECTION



Inserting known values around a protected region.

buf[n+1]

memcpy(buf, src, n+1)



BUFFER OVERFLOW DETECTION METHODOLOGY

CANARY-BASED DETECTION



▲ Inserting known values around a protected region.

▲ `buf[n+1]`

▲ `memcpy(buf, src, n+1)`



BUFFER OVERFLOW DETECTION METHODOLOGY

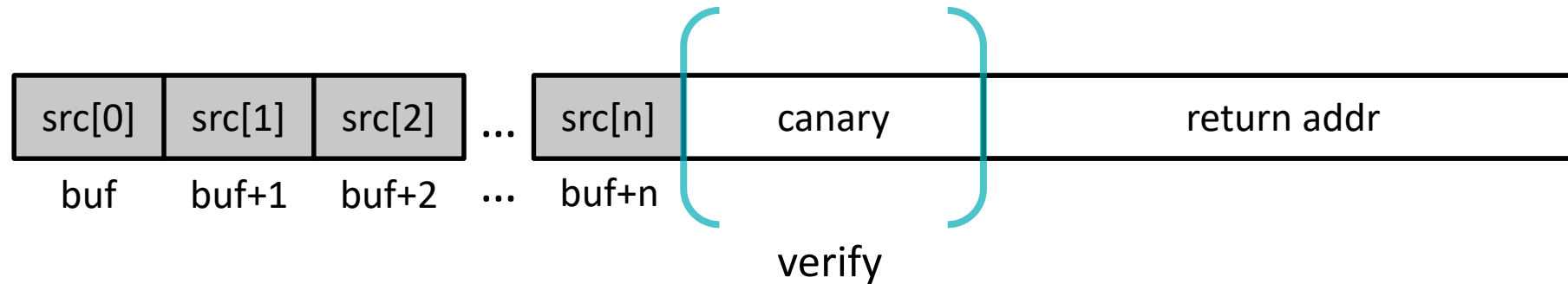
CANARY-BASED DETECTION



▲ Inserting known values around a protected region.

▲ buf[n+1]

▲ memcpy(buf, src, n+1)



BUFFER OVERFLOW DETECTION METHODOLOGY

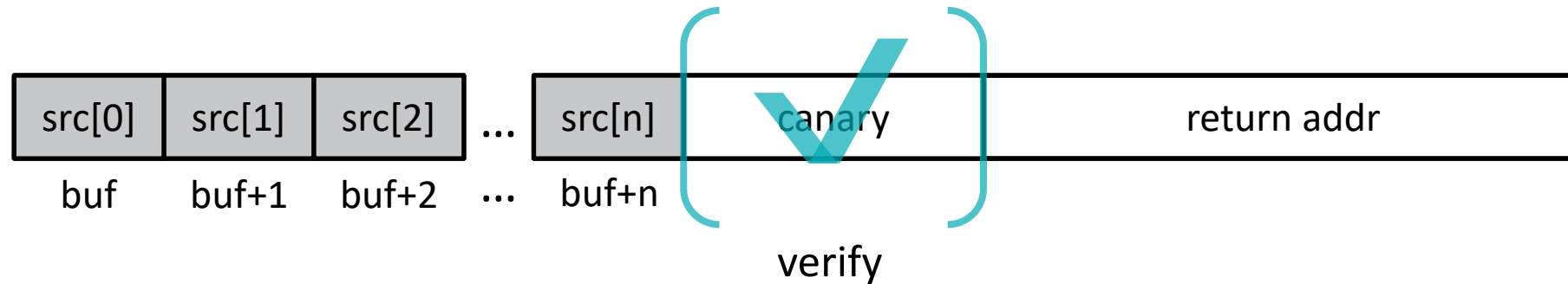
CANARY-BASED DETECTION



▲ Inserting known values around a protected region.

▲ buf[n+1]

▲ memcpy(buf, src, n+1)



BUFFER OVERFLOW DETECTION METHODOLOGY

CANARY-BASED DETECTION



Inserting known values around a protected region.

buf[n+1]

memcpy(buf, src, n+5)



BUFFER OVERFLOW DETECTION METHODOLOGY

CANARY-BASED DETECTION



▲ Inserting known values around a protected region.

▲ `buf[n+1]`

▲ `memcpy(buf, src, n+5)`



BUFFER OVERFLOW DETECTION METHODOLOGY

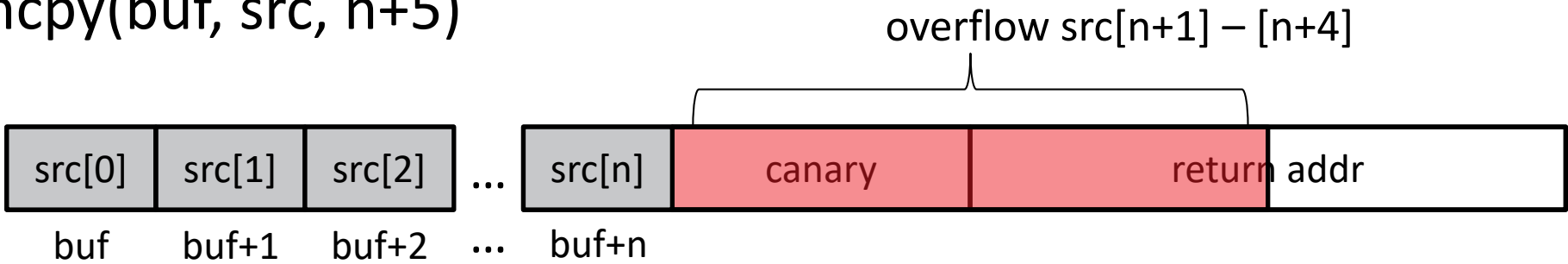
CANARY-BASED DETECTION



Inserting known values around a protected region.

buf[n+1]

memcpy(buf, src, n+5)



BUFFER OVERFLOW DETECTION METHODOLOGY

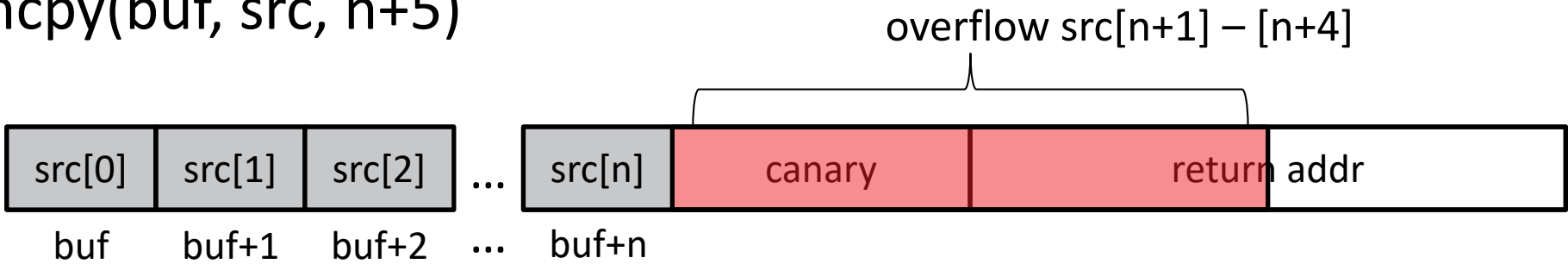
CANARY-BASED DETECTION



Inserting known values around a protected region.

buf[n+1]

memcpy(buf, src, n+5)



BUFFER OVERFLOW DETECTION METHODOLOGY

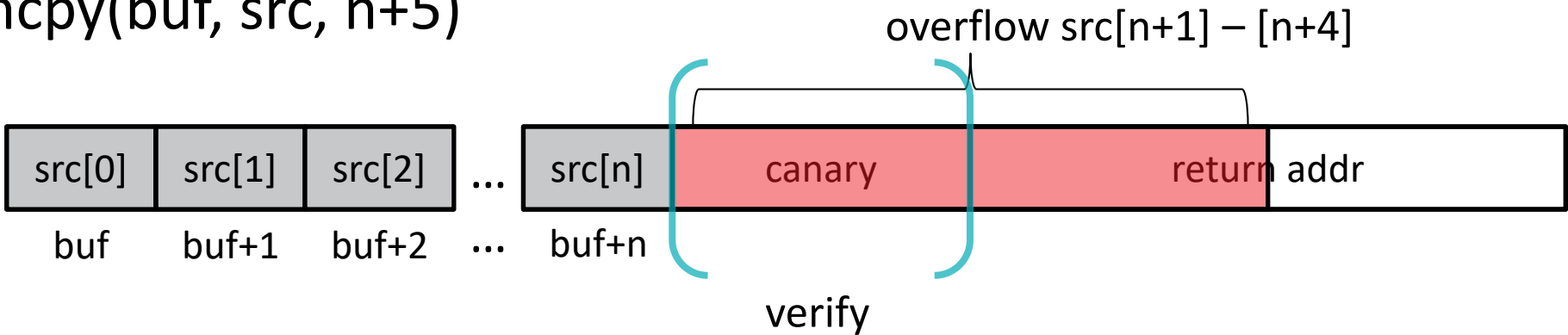


CANARY-BASED DETECTION

Inserting known values around a protected region.

buf[n+1]

memcpy(buf, src, n+5)



BUFFER OVERFLOW DETECTION METHODOLOGY

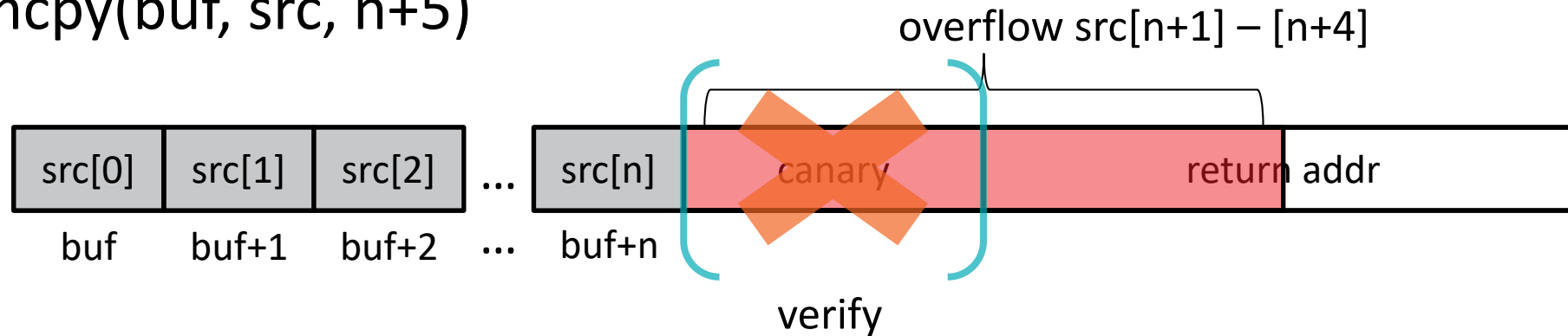
CANARY-BASED DETECTION



▲ Inserting known values around a protected region.

▲ `buf[n+1]`

▲ `memcpy(buf, src, n+5)`



BUFFER OVERFLOW DETECTION METHODOLOGY

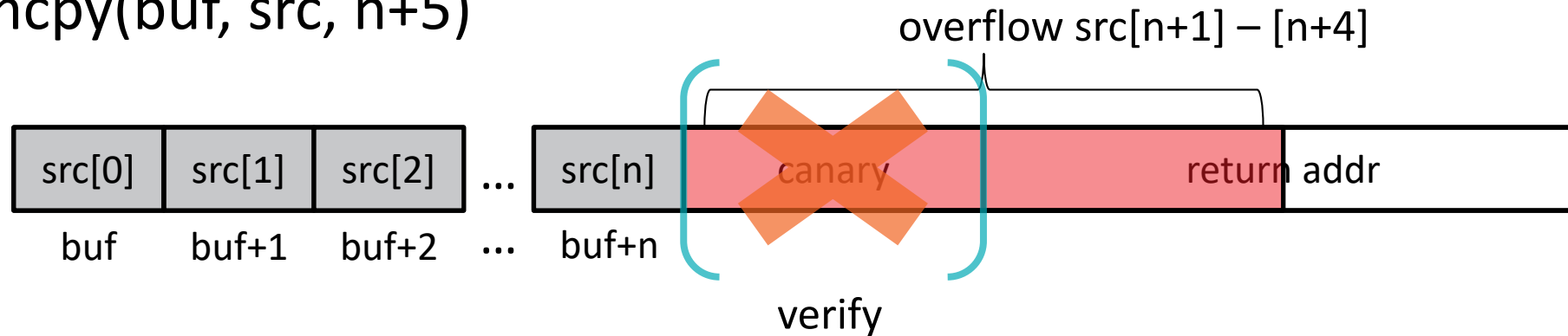
CANARY-BASED DETECTION



▲ Inserting known values around a protected region.

▲ `buf[n+1]`

▲ `memcpy(buf, src, n+5)`



▲ Absence of known canary values alerts to invalid writes.



GOALS

BUILDING CLARMOR



- ▲ Software tool to detect buffer overflows caused by GPU
 - clARMOR found 13 GPU buffer overflows in 7 programs
- ▲ Runnable with most OpenCL™ applications
 - Tested for GPU and CPU device types from multiple vendors
- ▲ Low runtime overhead
 - 14% overhead across 175 applications in 16 GPU benchmark suites

LAUNCHING AN OPENCL™ KERNEL



LAUNCHING AN OPENCL™ KERNEL



Buffer Create



Buffer

LAUNCHING AN OPENCL™ KERNEL



Set Arguments

A solid teal-colored rectangle.

Buffer

A solid purple-colored rectangle.

Kernel

LAUNCHING AN OPENCL™ KERNEL



Set Arguments



LAUNCHING AN OPENCL™ KERNEL



Launch Kernel



LAUNCHING AN OPENCL™ KERNEL



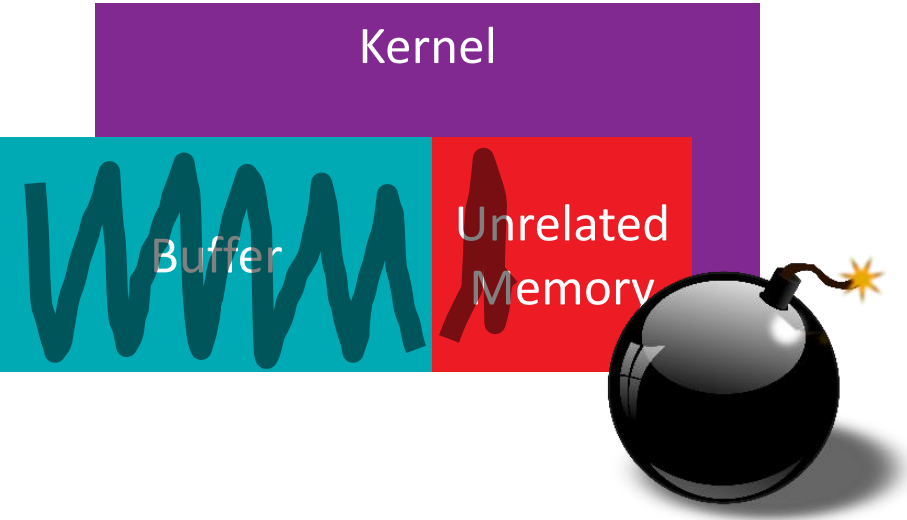
Launch Kernel



LAUNCHING AN OPENCL™ KERNEL



Launch Kernel



LAUNCHING AN OPENCL™ WITH CLARMOR



Buffer Create



Buffer

LAUNCHING AN OPENCL™ WITH CLARMOR



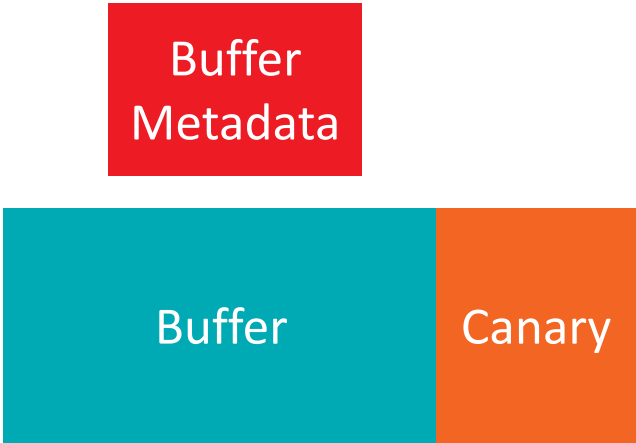
Buffer Create



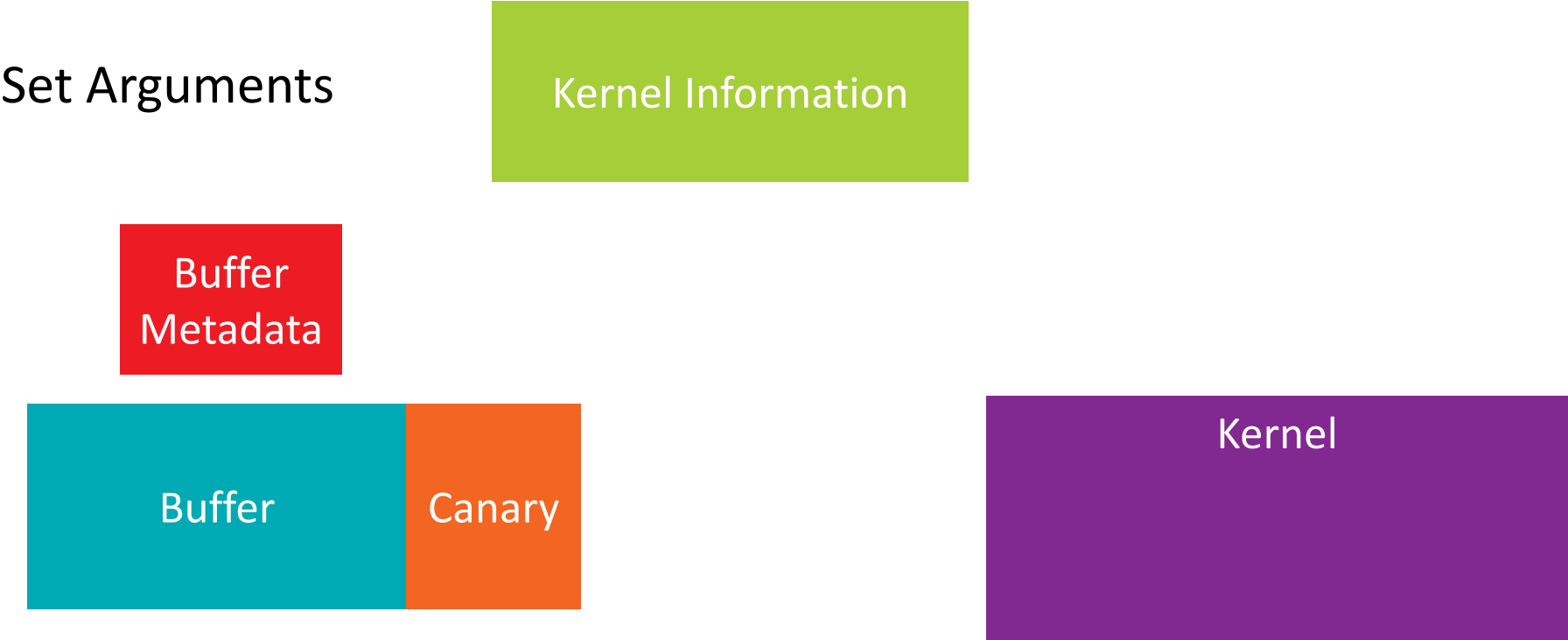
LAUNCHING AN OPENCL™ WITH CLARMOR



Buffer Create



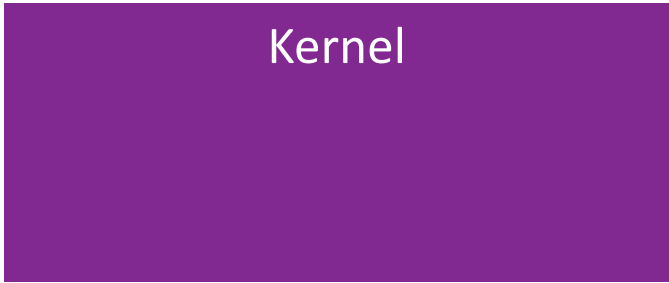
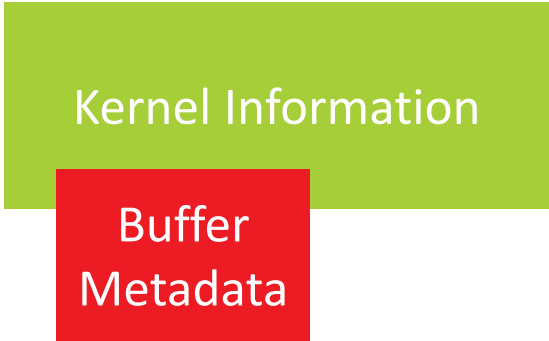
LAUNCHING AN OPENCL™ WITH CLARMOR



LAUNCHING AN OPENCL™ WITH CLARMOR



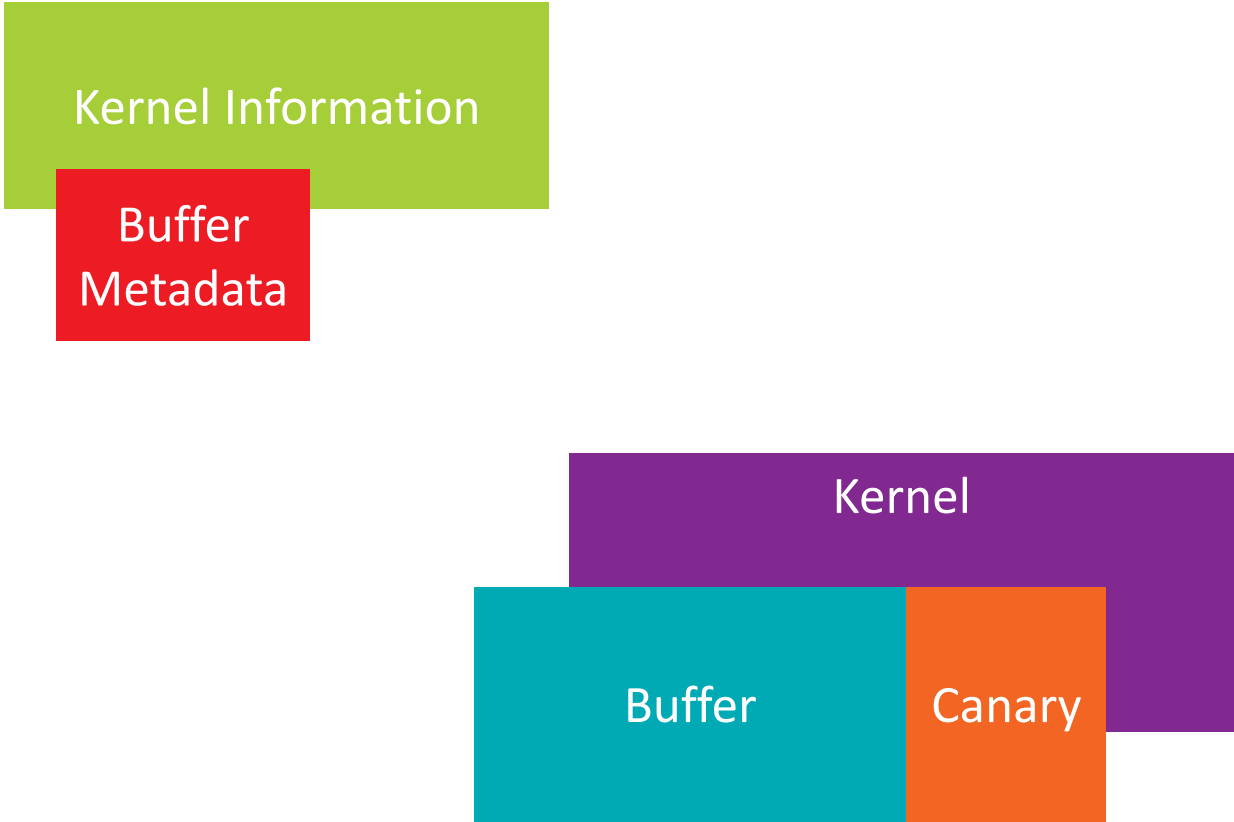
Set Arguments



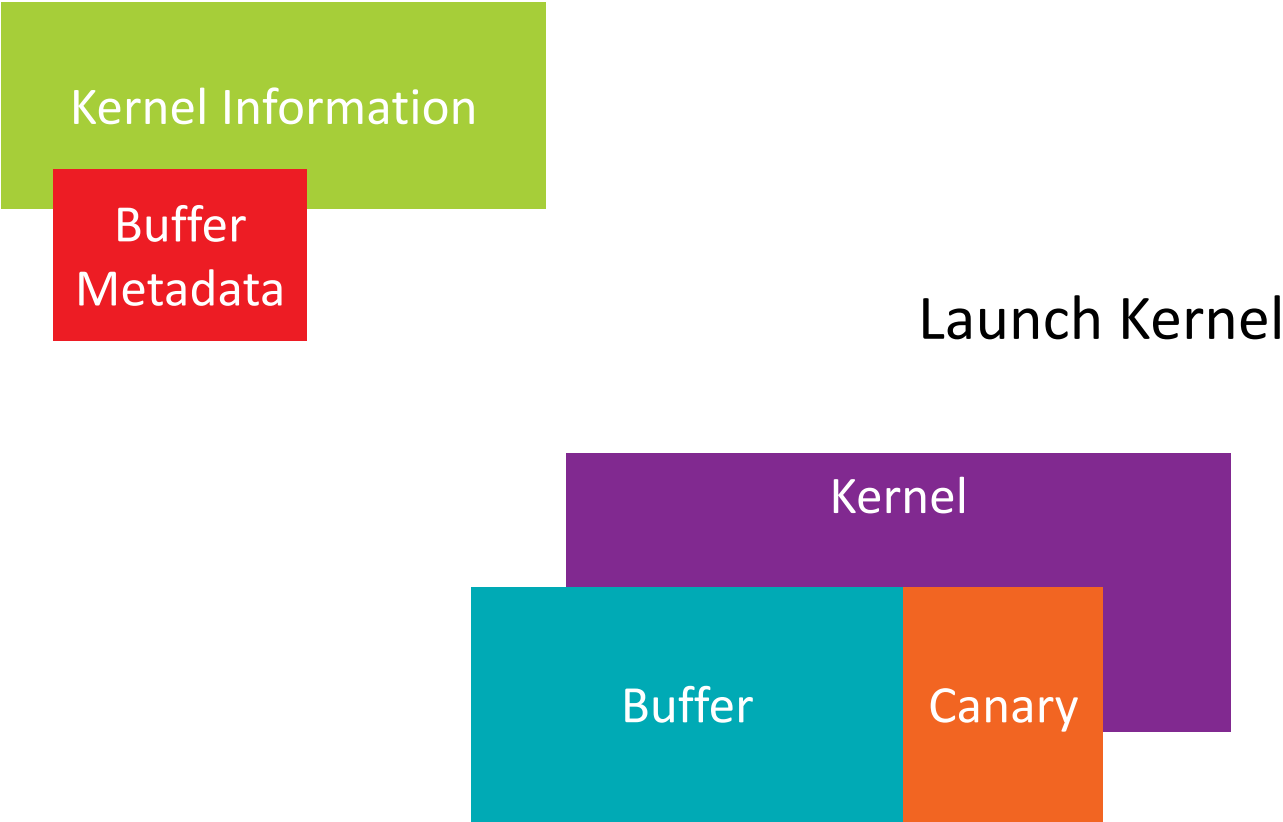
LAUNCHING AN OPENCL™ WITH CLARMOR



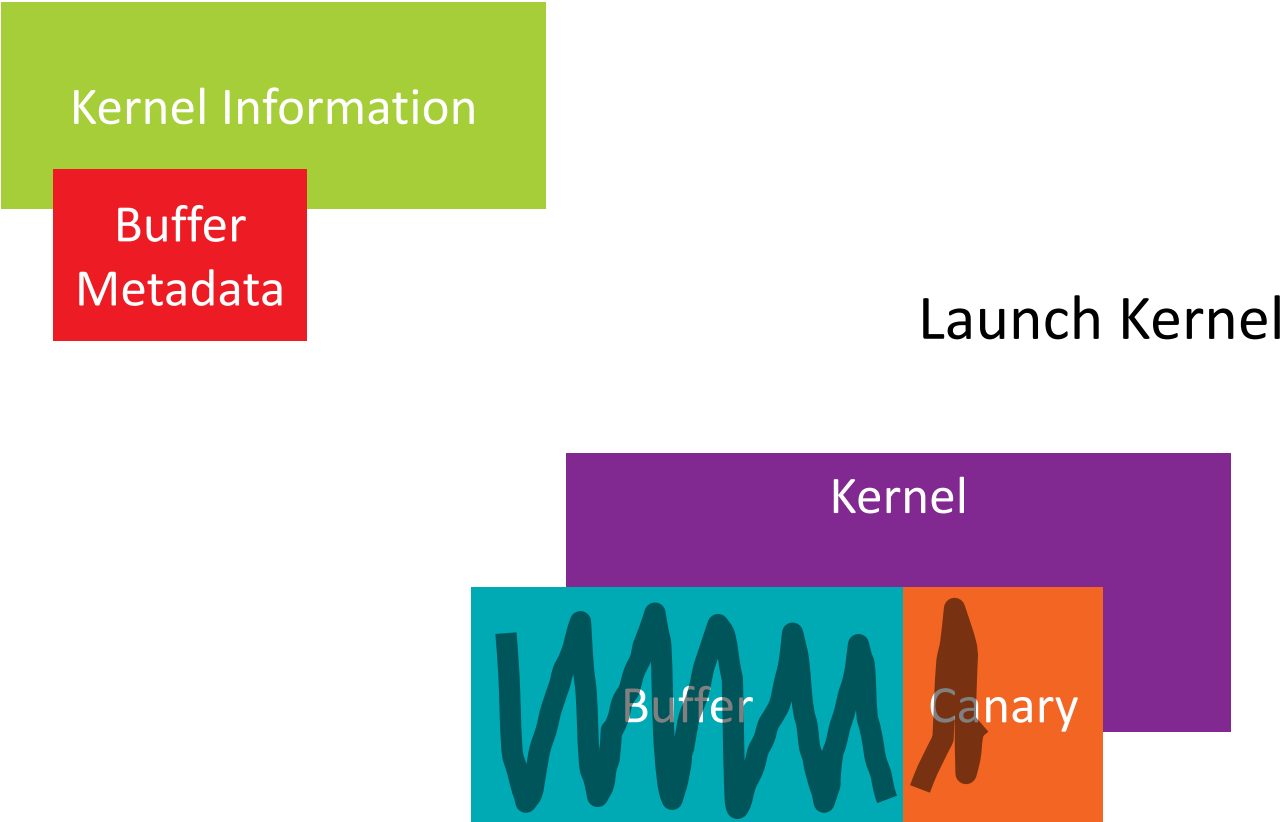
Set Arguments



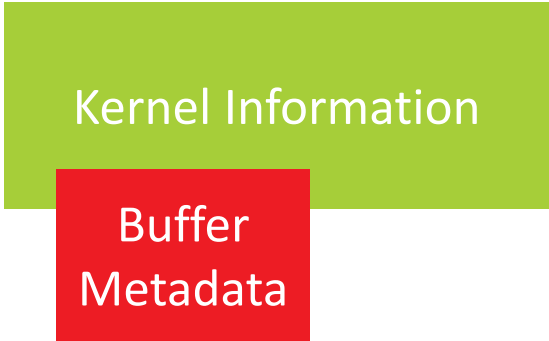
LAUNCHING AN OPENCL™ WITH CLARMOR



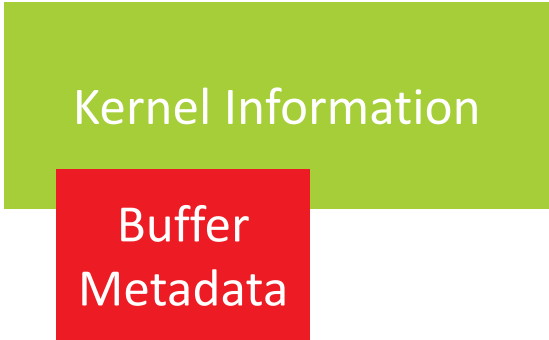
LAUNCHING AN OPENCL™ WITH CLARMOR



LAUNCHING AN OPENCL™ WITH CLARMOR



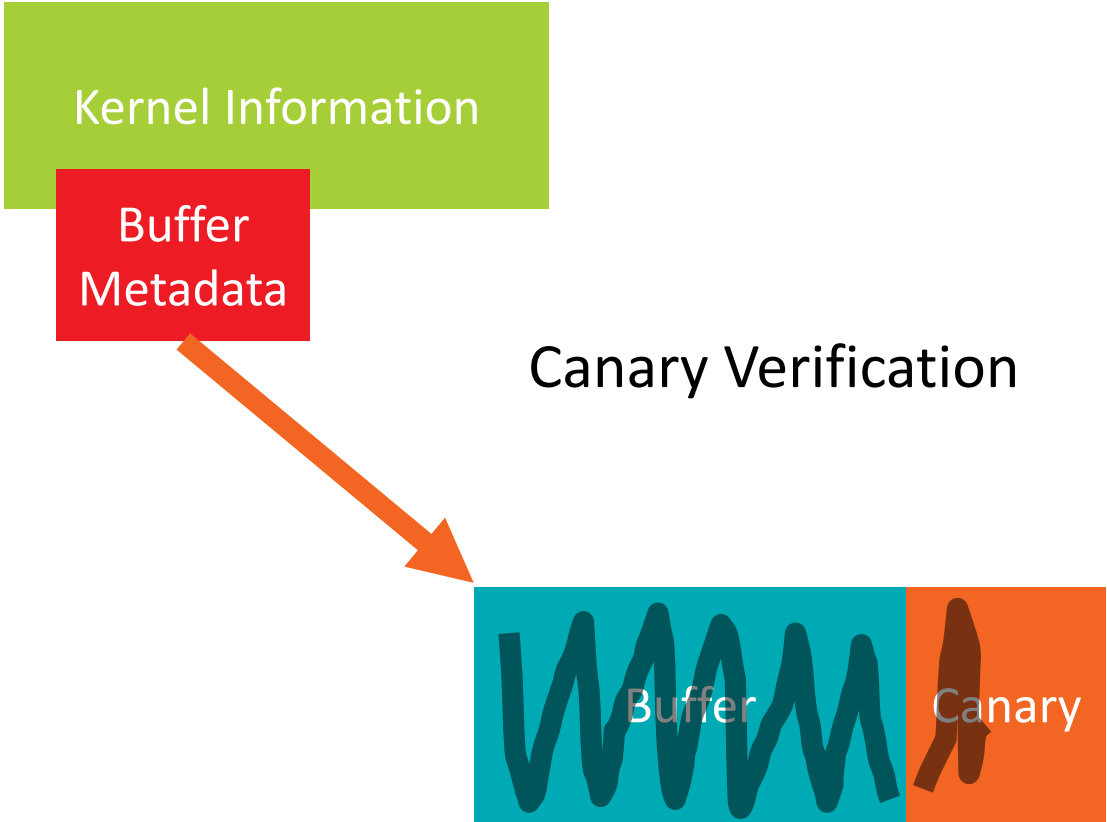
LAUNCHING AN OPENCL™ WITH CLARMOR



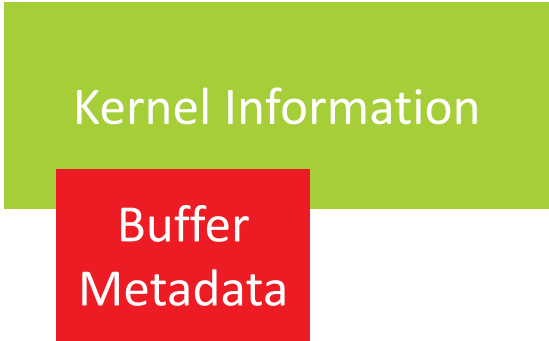
Canary Verification



LAUNCHING AN OPENCL™ WITH CLARMOR



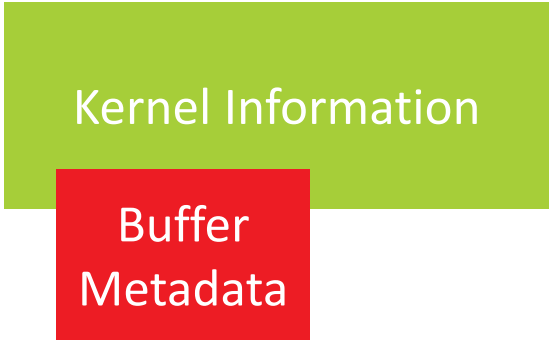
LAUNCHING AN OPENCL™ WITH CLARMOR



Canary Verification



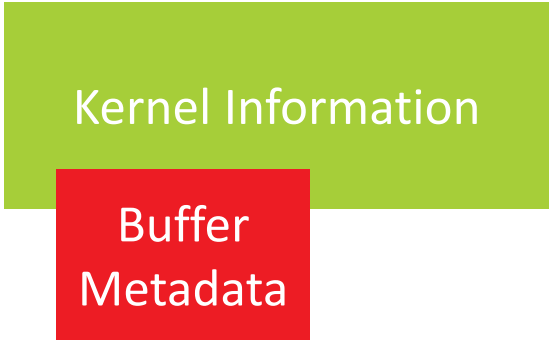
LAUNCHING AN OPENCL™ WITH CLARMOR



Canary Verification



LAUNCHING AN OPENCL™ WITH CLARMOR



Canary Verification

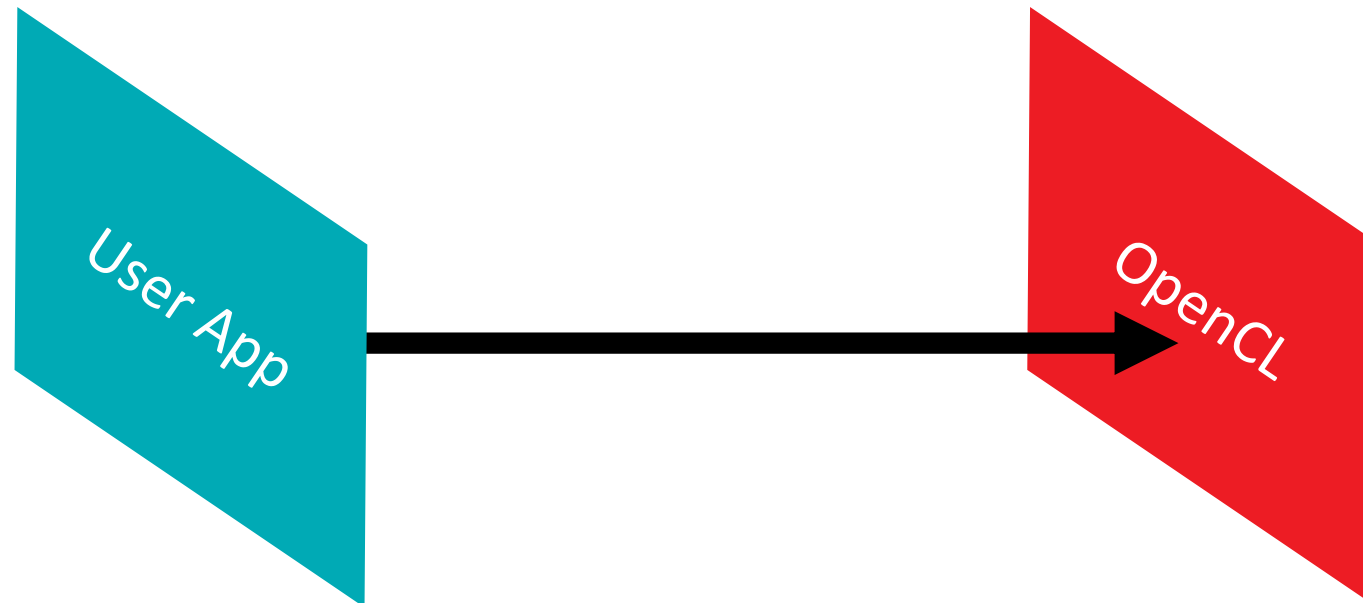


WRAPPING OPENCL™



CLARMOR BETWEEN YOUR APPLICATION AND OPENCL

- ▲ clARMOR is a Linux® library that uses LD_PRELOAD to wrap OpenCL™ library calls
- ▲ Call Wrapping
 - Buffer and Image creates
 - Argument setters
 - Kernel launches
 - Information functions

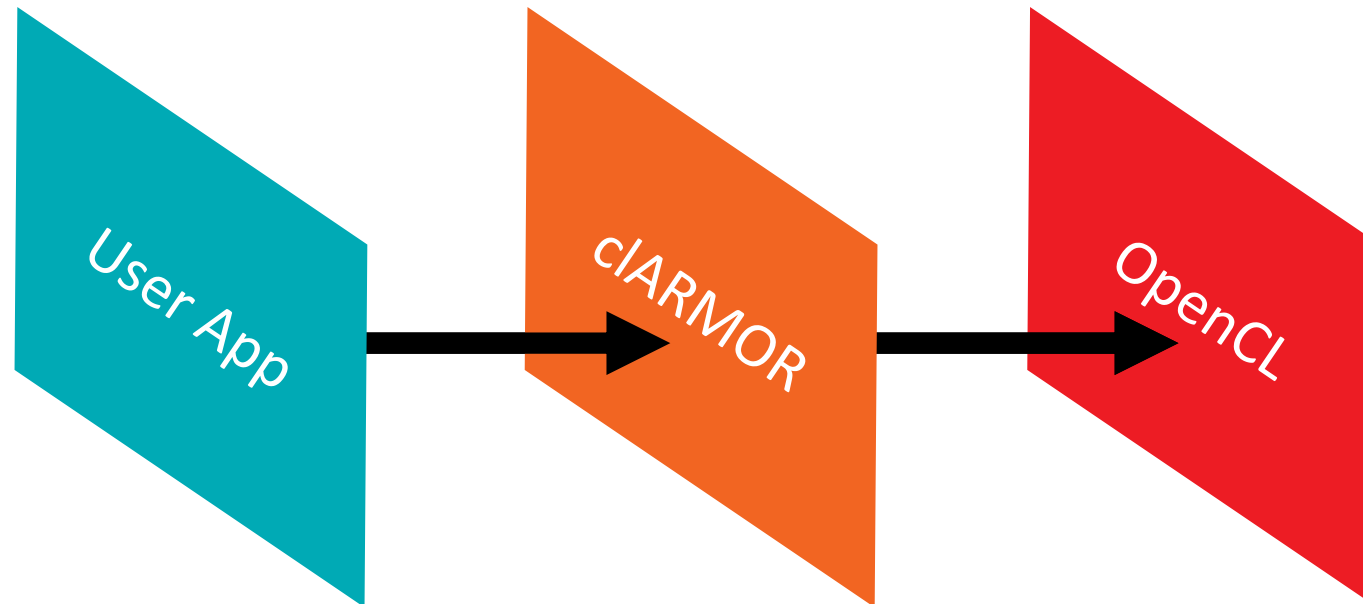


WRAPPING OPENCL™



CLARMOR BETWEEN YOUR APPLICATION AND OPENCL

- ▲ clARMOR is a Linux® library that uses LD_PRELOAD to wrap OpenCL™ library calls
- ▲ Call Wrapping
 - Buffer and Image creates
 - Argument setters
 - Kernel launches
 - Information functions



WRAPPING THE OPENCL™ API

BUFFER AND IMAGE CREATION



▲ Attach canaries to memory objects

▲ Buffer Creation

- Calls to ***clCreateBuffer***, ***clCreateSubBuffer*** or ***clSVMAlloc***
- Increase space requested, fill end with canary



WRAPPING THE OPENCL™ API

BUFFER AND IMAGE CREATION



▲ Attach canaries to memory objects

▲ Buffer Creation

- Calls to ***clCreateBuffer***, ***clCreateSubBuffer*** or ***clSVMAlloc***
- Increase space requested, fill end with canary



WRAPPING THE OPENCL™ API

BUFFER AND IMAGE CREATION



▲ Attach canaries to memory objects

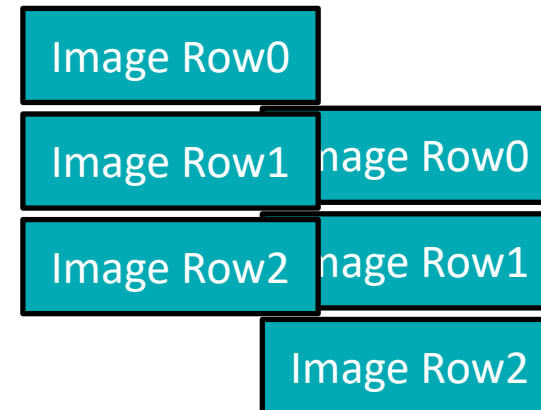
▲ Buffer Creation

- Calls to ***clCreateBuffer***, ***clCreateSubBuffer*** or ***clSVMAlloc***
- Increase space requested, fill end with canary



▲ Image Creation

- Calls to ***clCreateImage***, ***clCreateImage2D***, or ***clCreateImage3D***
- Potential for multi dimensional overflow
- Add canary regions to each dimension



WRAPPING THE OPENCL™ API

BUFFER AND IMAGE CREATION



▲ Attach canaries to memory objects

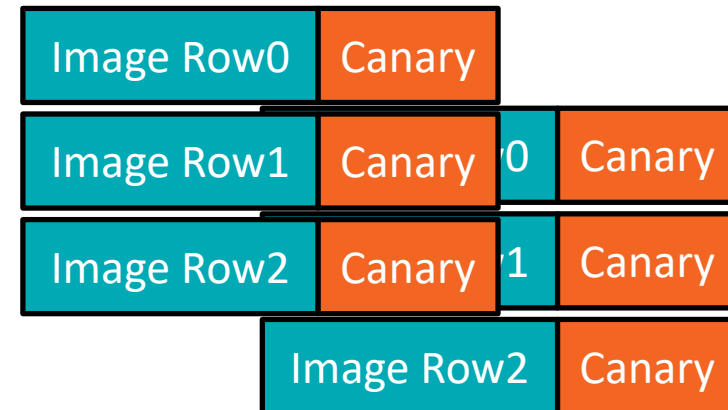
▲ Buffer Creation

- Calls to ***clCreateBuffer***, ***clCreateSubBuffer*** or ***clSVMAlloc***
- Increase space requested, fill end with canary



▲ Image Creation

- Calls to ***clCreateImage***, ***clCreateImage2D***, or ***clCreateImage3D***
- Potential for multi dimensional overflow
- Add canary regions to each dimension



WRAPPING THE OPENCL™ API

BUFFER AND IMAGE CREATION



▲ Attach canaries to memory objects

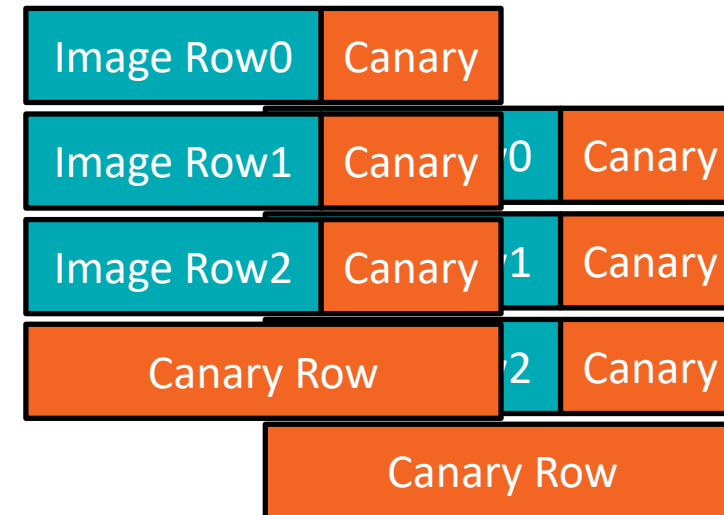
▲ Buffer Creation

- Calls to ***clCreateBuffer***, ***clCreateSubBuffer*** or ***clSVMAlloc***
- Increase space requested, fill end with canary



▲ Image Creation

- Calls to ***clCreateImage***, ***clCreateImage2D***, or ***clCreateImage3D***
- Potential for multi dimensional overflow
- Add canary regions to each dimension



WRAPPING THE OPENCL™ API

BUFFER AND IMAGE CREATION



▲ Attach canaries to memory objects

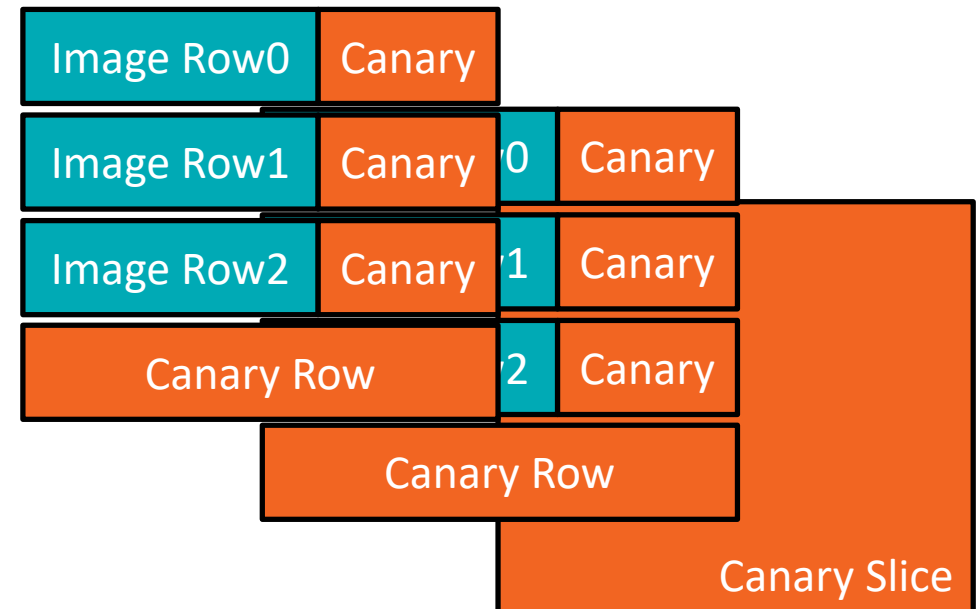
▲ Buffer Creation

- Calls to ***clCreateBuffer***, ***clCreateSubBuffer*** or ***clSVMAlloc***
- Increase space requested, fill end with canary



▲ Image Creation

- Calls to ***clCreateImage***, ***clCreateImage2D***, or ***clCreateImage3D***
- Potential for multi dimensional overflow
- Add canary regions to each dimension

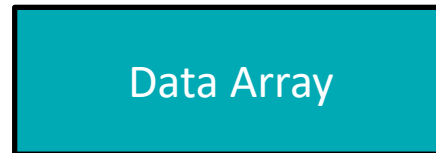


WRAPPING THE OPENCL™ API

BUFFER CREATION FROM EXISTING ALLOCATIONS



- ▲ OpenCL allows buffer creation using an existing memory allocation



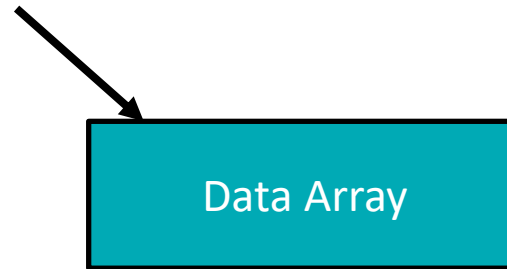
WRAPPING THE OPENCL™ API

BUFFER CREATION FROM EXISTING ALLOCATIONS



- ▲ OpenCL allows buffer creation using an existing memory allocation

Make this a buffer.

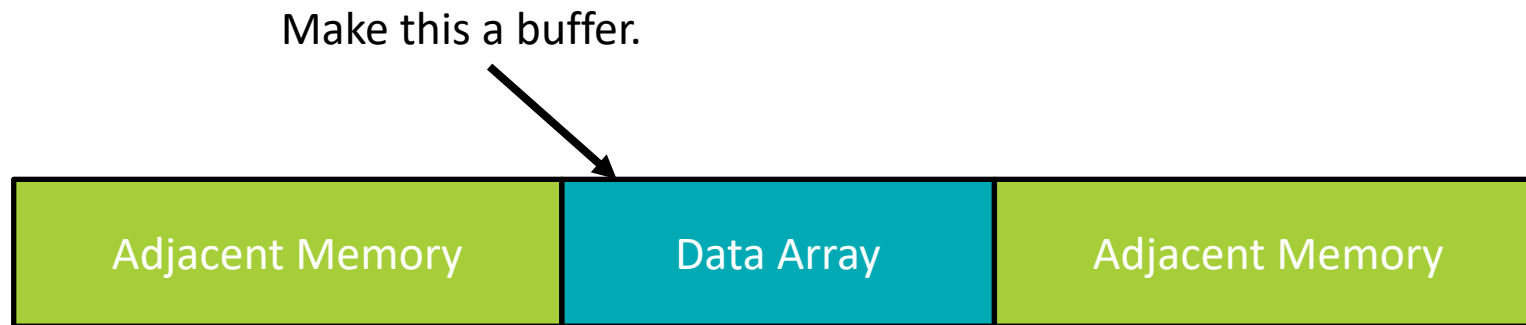


WRAPPING THE OPENCL™ API

BUFFER CREATION FROM EXISTING ALLOCATIONS



- ▲ OpenCL allows buffer creation using an existing memory allocation

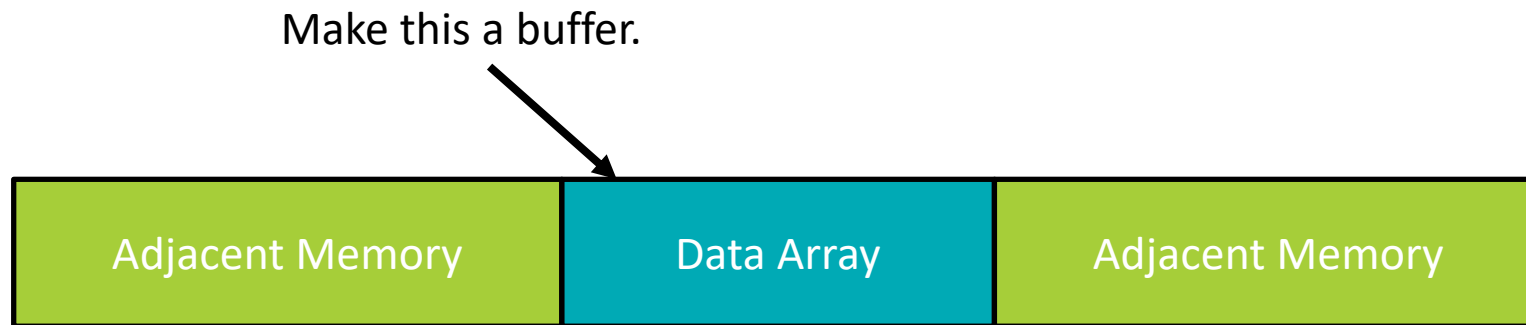


WRAPPING THE OPENCL™ API

BUFFER CREATION FROM EXISTING ALLOCATIONS



- ▲ OpenCL allows buffer creation using an existing memory allocation
 - Cannot extend buffer

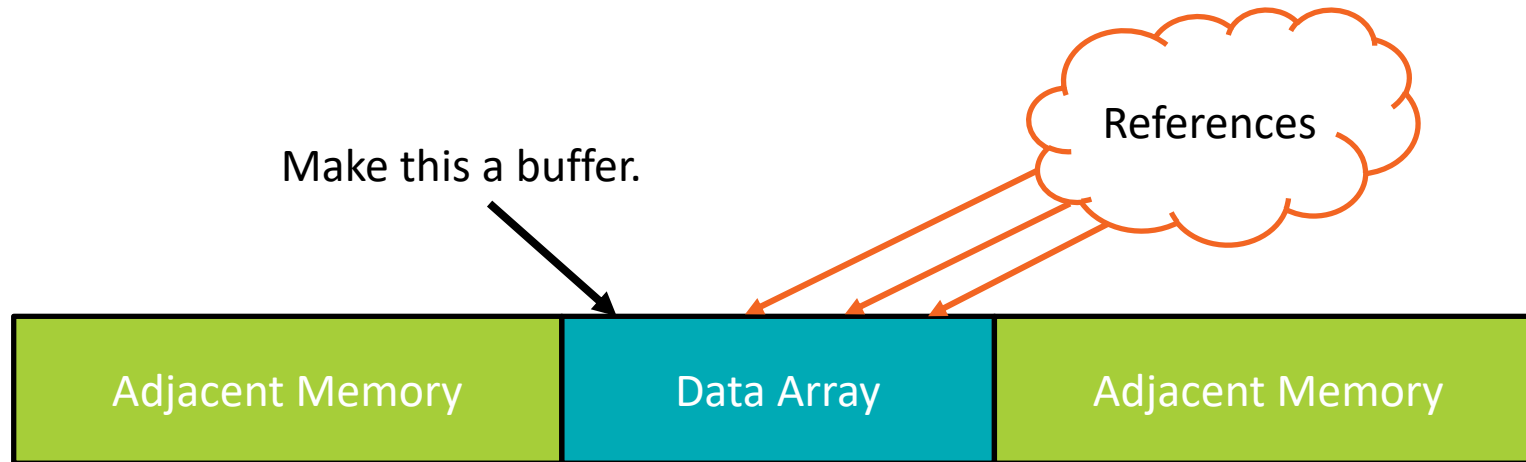


WRAPPING THE OPENCL™ API

BUFFER CREATION FROM EXISTING ALLOCATIONS



- ▲ OpenCL allows buffer creation using an existing memory allocation
 - Cannot extend buffer



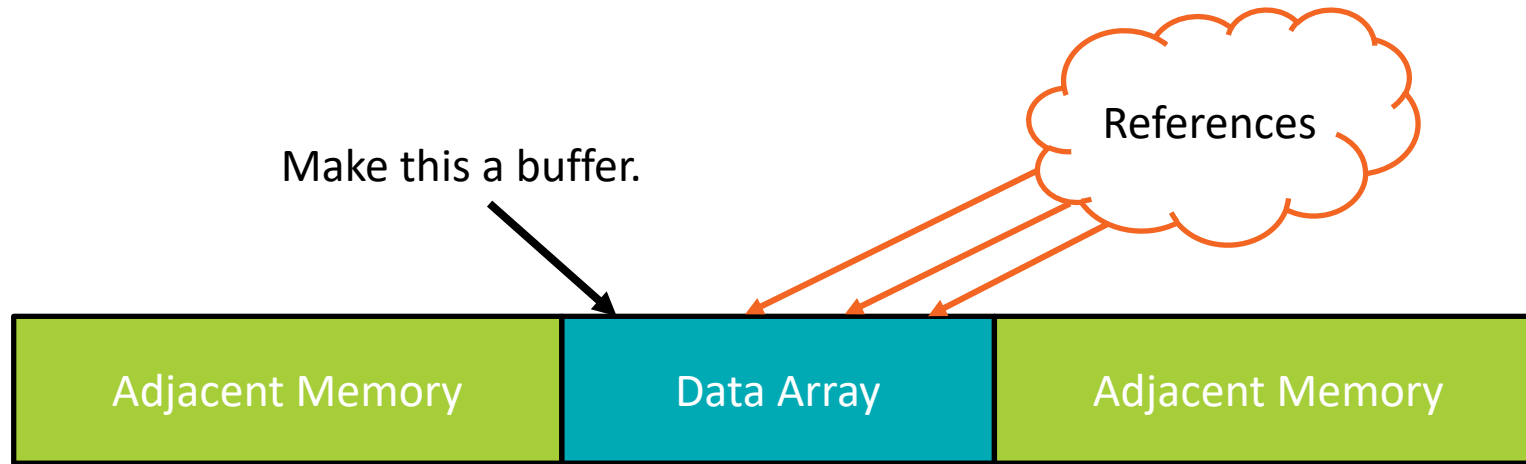
WRAPPING THE OPENCL™ API

BUFFER CREATION FROM EXISTING ALLOCATIONS



▲ OpenCL allows buffer creation using an existing memory allocation

- Cannot extend buffer
- Cannot move buffer



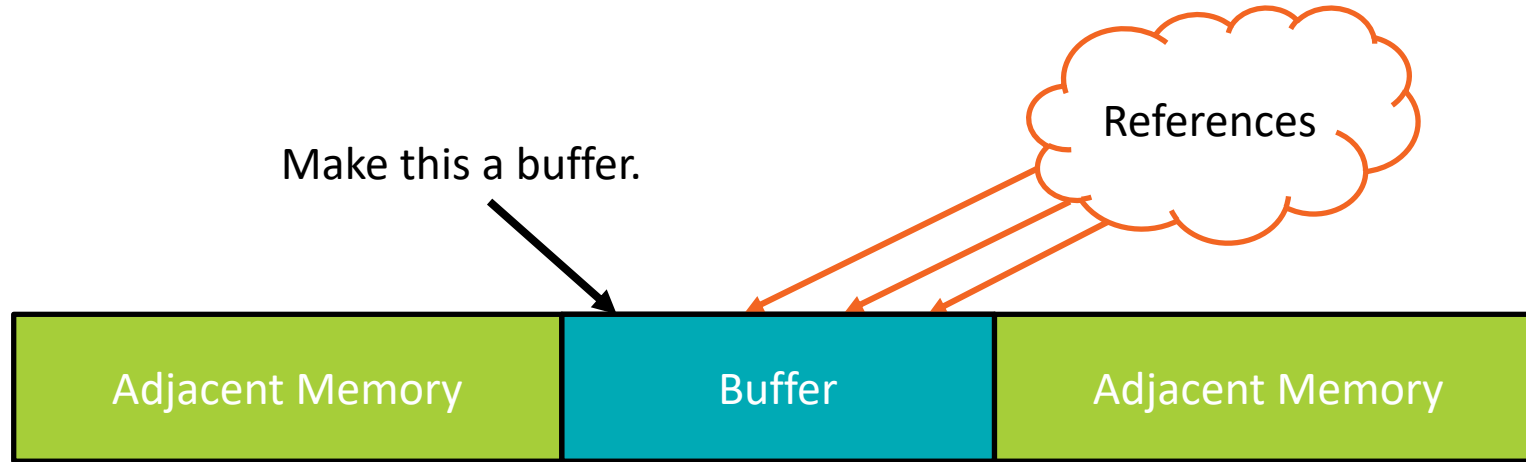
WRAPPING THE OPENCL™ API

BUFFER CREATION FROM EXISTING ALLOCATIONS



▲ OpenCL allows buffer creation using an existing memory allocation

- Cannot extend buffer
- Cannot move buffer



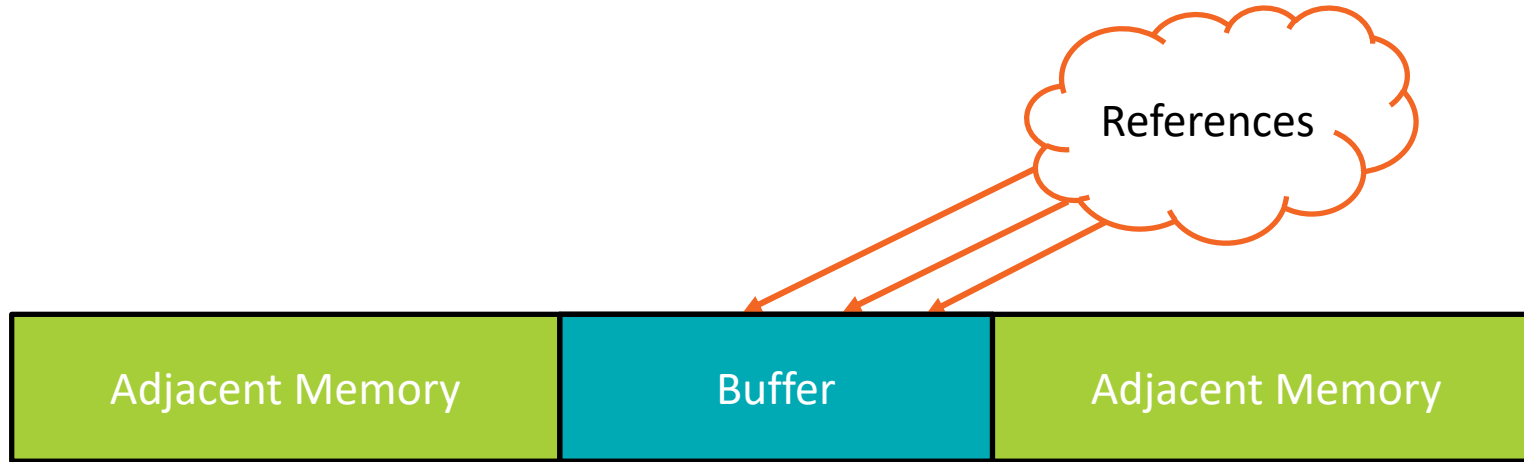
WRAPPING THE OPENCL™ API

BUFFER CREATION FROM EXISTING ALLOCATIONS



▲ OpenCL allows buffer creation using an existing memory allocation

- Cannot extend buffer
- Cannot move buffer



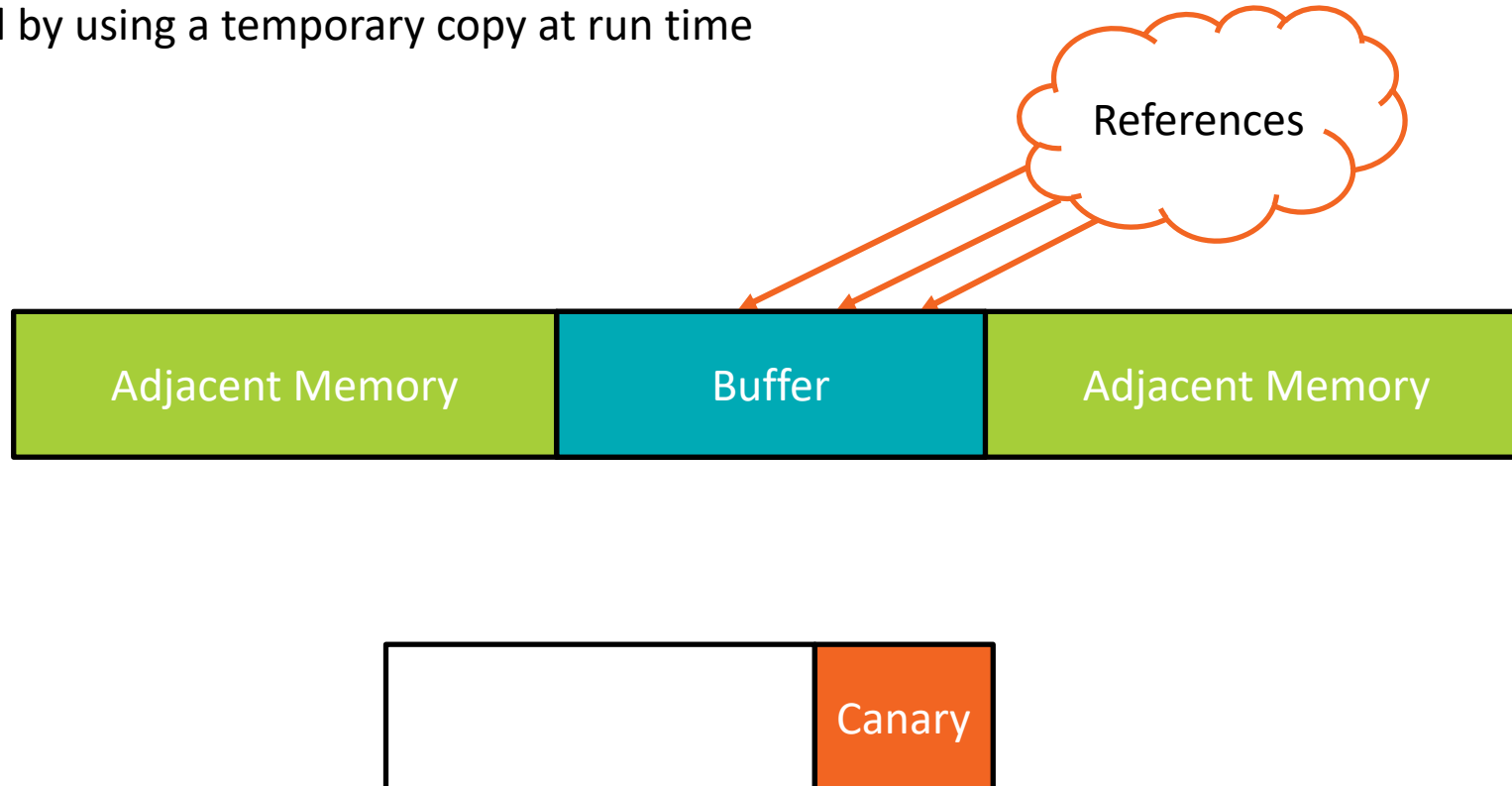
WRAPPING THE OPENCL™ API

BUFFER CREATION FROM EXISTING ALLOCATIONS



▲ OpenCL allows buffer creation using an existing memory allocation

- Cannot extend buffer
- Cannot move buffer
- Work around by using a temporary copy at run time



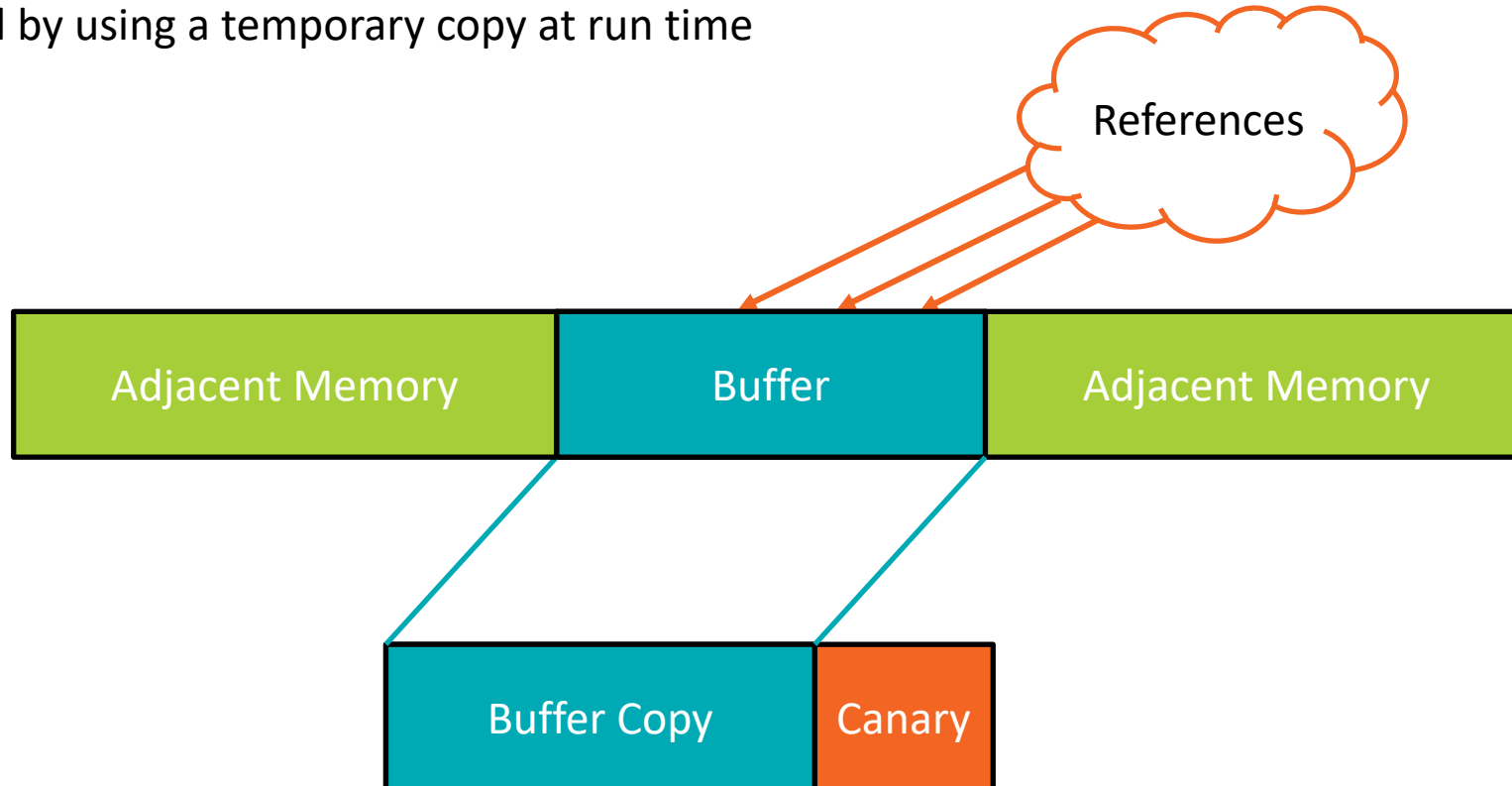
WRAPPING THE OPENCL™ API

BUFFER CREATION FROM EXISTING ALLOCATIONS



▲ OpenCL allows buffer creation using an existing memory allocation

- Cannot extend buffer
- Cannot move buffer
- Work around by using a temporary copy at run time



WRAPPING THE OPENCL™ API

SET ARGUMENTS



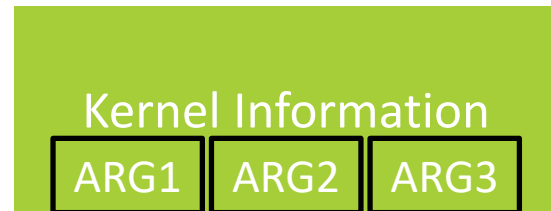
- ▲ clARMOR needs to know which buffers/images to check for overflows
- ▲ Kernel information object
 - map kernel argument number to buffer information
- ▲ Update on call to ***clSetKernelArg*** or ***clSetKernelArgSVMPointer***

WRAPPING THE OPENCL™ API

SET ARGUMENTS



- ▲ clARMOR needs to know which buffers/images to check for overflows
- ▲ Kernel information object
 - map kernel argument number to buffer information
- ▲ Update on call to ***clSetKernelArg*** or ***clSetKernelArgSVMPointer***

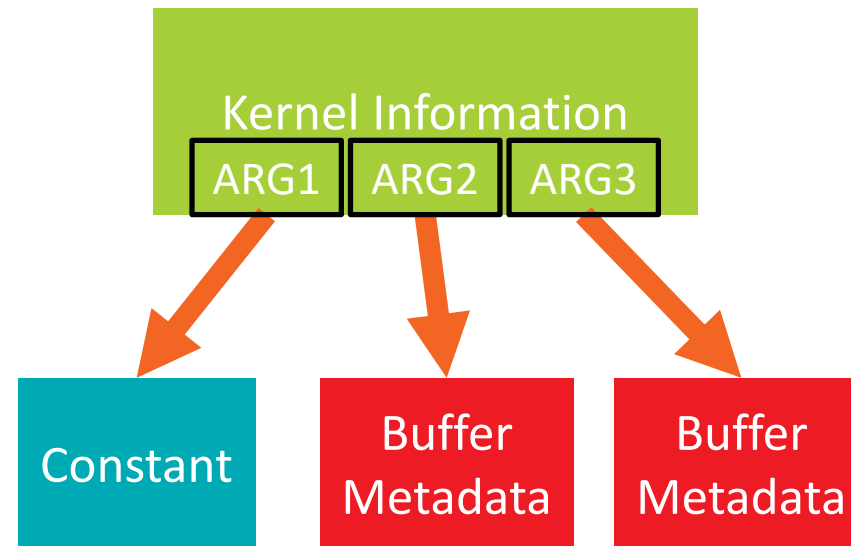


WRAPPING THE OPENCL™ API

SET ARGUMENTS



- ▲ clARMOR needs to know which buffers/images to check for overflows
- ▲ Kernel information object
 - map kernel argument number to buffer information
- ▲ Update on call to ***clSetKernelArg*** or ***clSetKernelArgSVMPointer***



WRAPPING THE OPENCL™ API

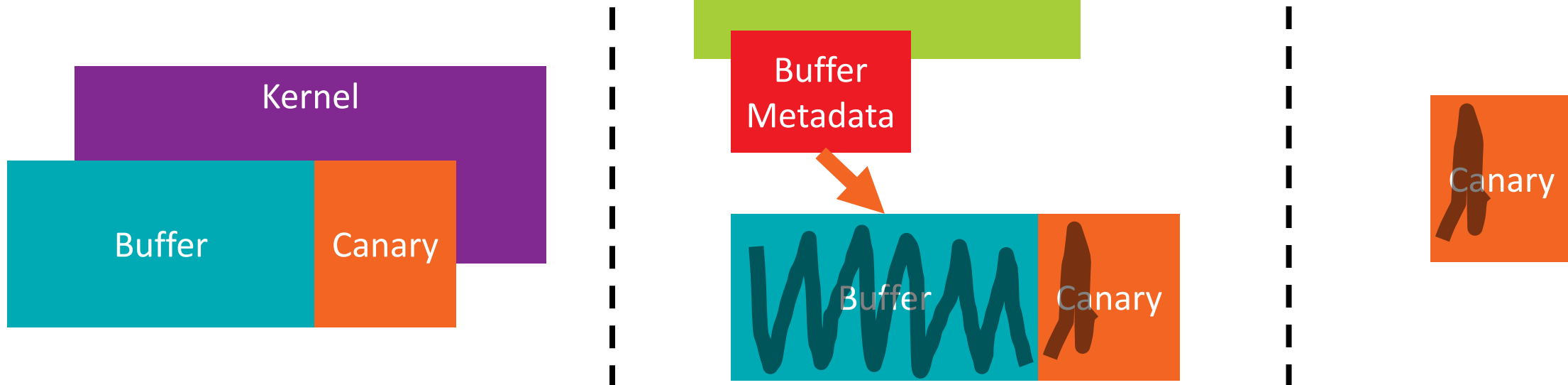
KERNEL LAUNCH



▲ Do the work of detecting buffer overflows

▲ On call to ***clEnqueueNDRangeKernel***

- Enqueue the kernel
- Retrieve affected buffers
- Run the canary check
- Report errors



WRAPPING THE OPENCL™ API

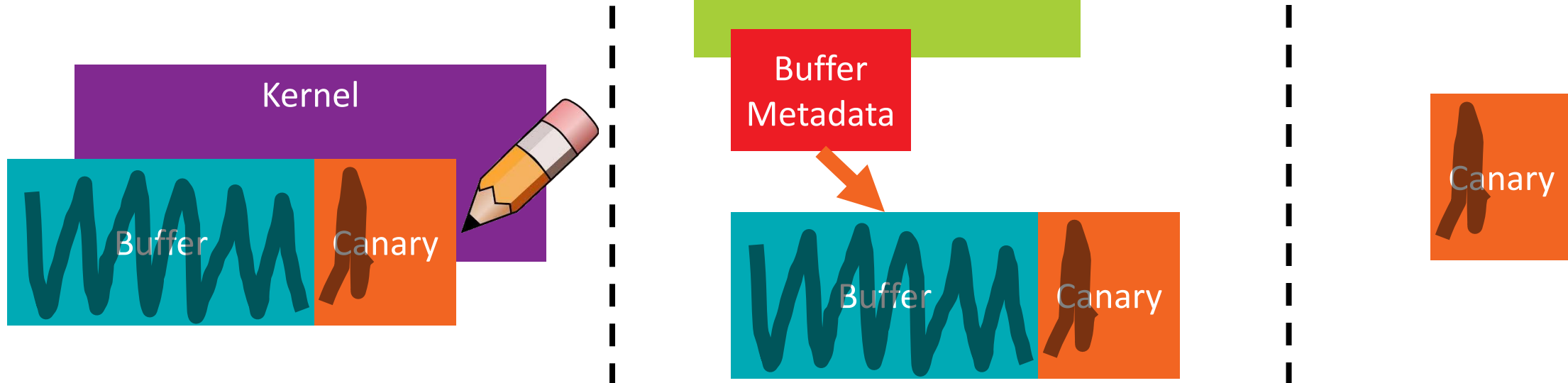
KERNEL LAUNCH



▲ Do the work of detecting buffer overflows

▲ On call to ***clEnqueueNDRangeKernel***

- Enqueue the kernel
- Retrieve affected buffers
- Run the canary check
- Report errors



WRAPPING THE OPENCL™ API

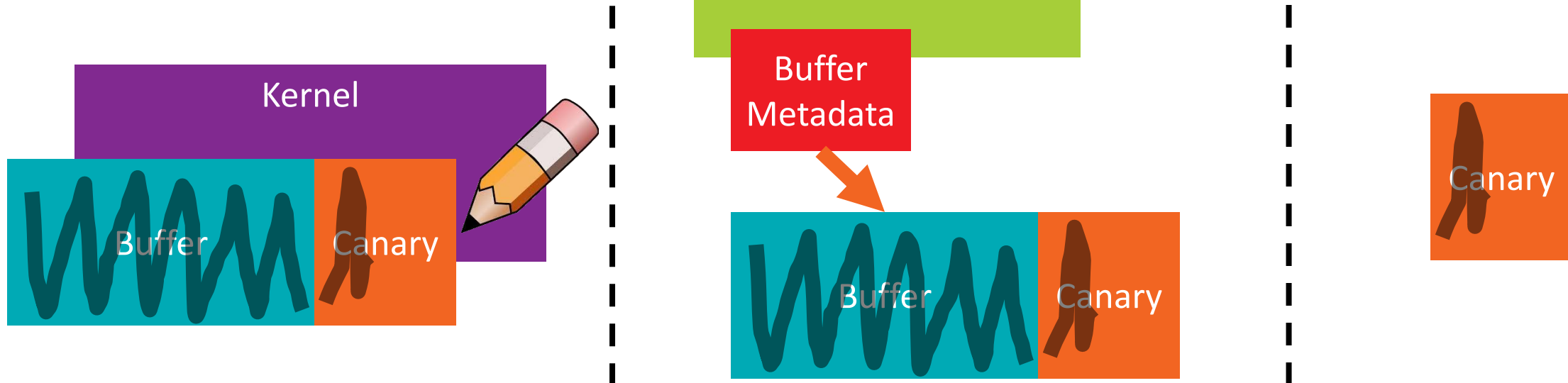
KERNEL LAUNCH



▲ Do the work of detecting buffer overflows

▲ On call to ***clEnqueueNDRangeKernel***

- Enqueue the kernel
- Retrieve affected buffers
- Run the canary check
- Report errors



WRAPPING THE OPENCL™ API

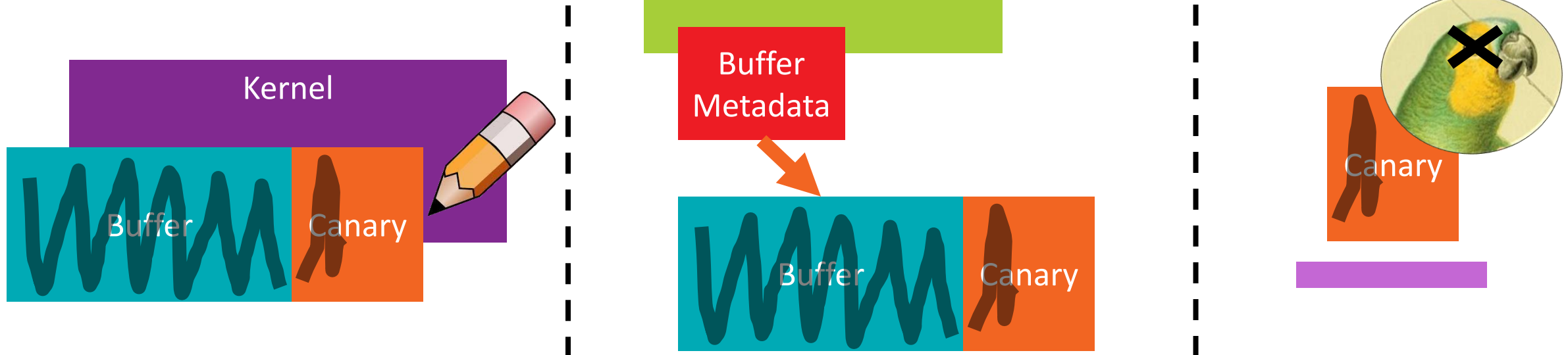
KERNEL LAUNCH



▲ Do the work of detecting buffer overflows

▲ On call to ***clEnqueueNDRangeKernel***

- Enqueue the kernel
- Retrieve affected buffers
- Run the canary check
- Report errors



GOALS

BUILDING CLARMOR



- ▲ Software tool to detect buffer overflows caused by GPU
 - clARMOR found 13 GPU buffer overflows in 7 programs
- ▲ Runnable with most OpenCL™ applications
 - Tested for GPU and CPU device types from multiple vendors
- ▲ Low runtime overhead
 - 14% overhead across 175 applications in 16 GPU benchmark suites

ACCELERATION



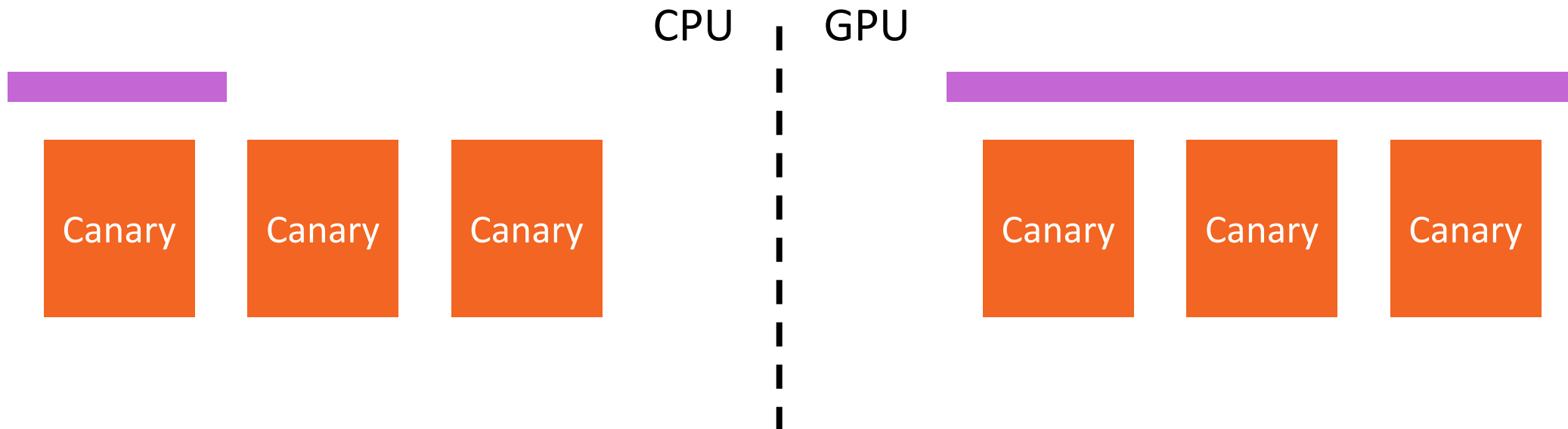
SELECTING A DEVICE FOR PERFORMING CANARY VERIFICATION

▲ CPU is faster

- small / few canary regions (latency advantage)

▲ GPU is faster

- large / many canary regions (throughput advantage with embarrassingly parallel workload)
- reduced transfers over PCIe[®] by keeping on GPU

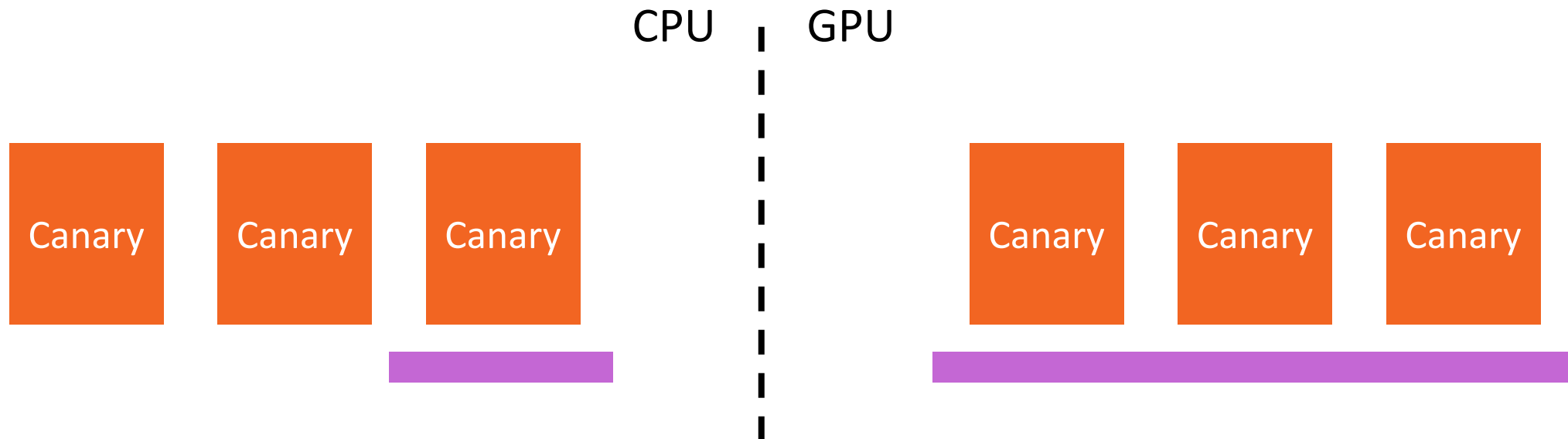


ACCELERATION



SELECTING A DEVICE FOR PERFORMING CANARY VERIFICATION

- ▲ CPU is faster
 - small / few canary regions (latency advantage)
- ▲ GPU is faster
 - large / many canary regions (throughput advantage with embarrassingly parallel workload)
 - reduced transfers over PCIe[®] by keeping on GPU



ACCELERATION

USING OPENCL™ EVENTS TO INCREASE THROUGHPUT



▲ Maximizing asynchrony

- Event-based programming wherever possible
- GPU check kernels enqueue behind work kernels and wait on completion
- Evaluation of check kernel results is done with call-backs



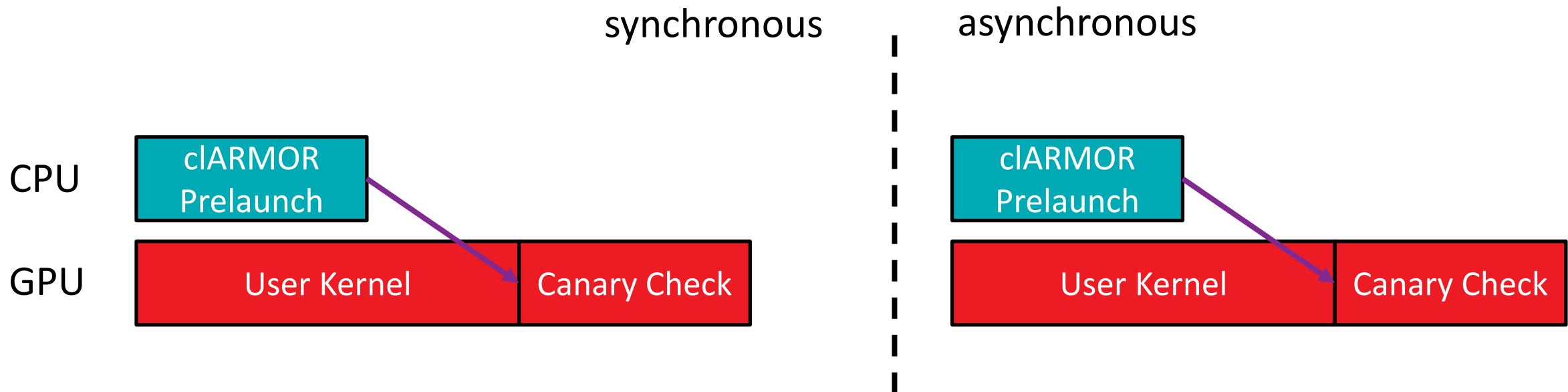
ACCELERATION



USING OPENCL™ EVENTS TO INCREASE THROUGHPUT

▲ Maximizing asynchrony

- Event-based programming wherever possible
- GPU check kernels enqueue behind work kernels and wait on completion
- Evaluation of check kernel results is done with call-backs



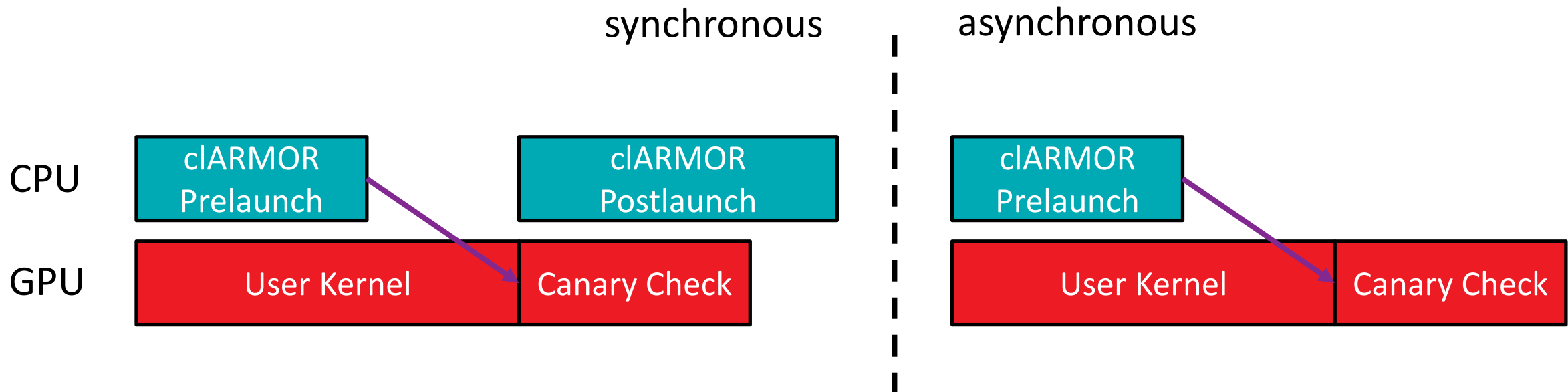
ACCELERATION



USING OPENCL™ EVENTS TO INCREASE THROUGHPUT

▲ Maximizing asynchrony

- Event-based programming wherever possible
- GPU check kernels enqueue behind work kernels and wait on completion
- Evaluation of check kernel results is done with call-backs



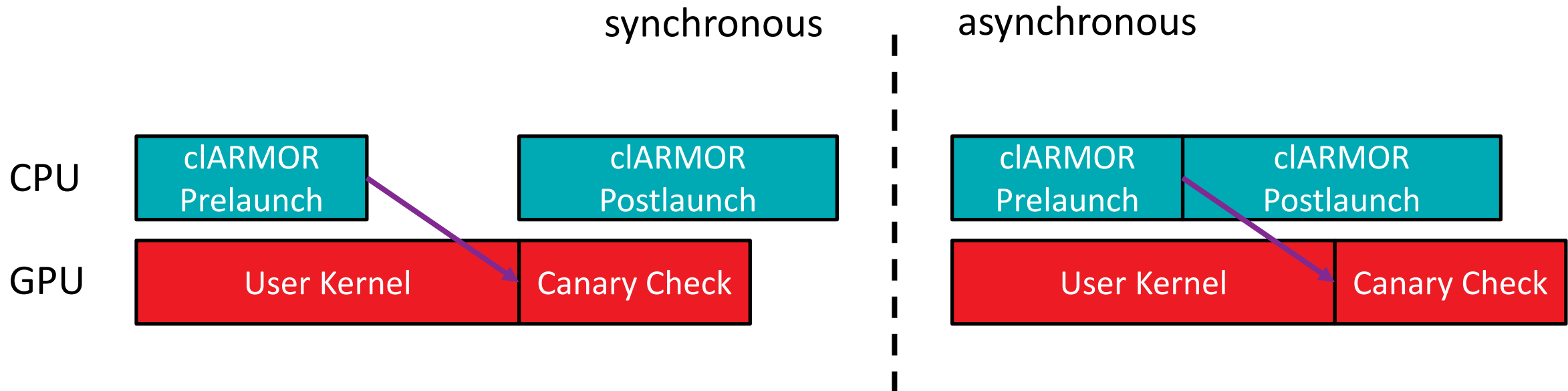
ACCELERATION



USING OPENCL™ EVENTS TO INCREASE THROUGHPUT

▲ Maximizing asynchrony

- Event-based programming wherever possible
- GPU check kernels enqueue behind work kernels and wait on completion
- Evaluation of check kernel results is done with call-backs



TEST SETUP

HARDWARE SPECIFICATIONS AND BENCHMARKS SUITES

- ▲ 3.7 GHz AMD A10-7850K CPU
 - 32 GB of DDR3-1866
- ▲ AMD FirePro™ W9100 discrete GPU
 - 930 MHz core frequency
 - 320 GB/s of memory bandwidth
 - 16 GB of GDDR5 memory
- ▲ 3rd Generation PCIe® x8 CPU–GPU connection
- ▲ 175 benchmarks in 16 benchmark suites



Suite	Num Benchmarks
AMDAPP	46
FINANCEBENCH	2
GPUSTREAM	1
HETEROMARK	14
MANTEVO	4
NPB_OCL	8
OPENDWARFS	7
PANNOTIA	6
PARBOIL	9
PHORONIX	4
POLYBENCH	21
PROXYAPPS	6
RODINIA	21
SHOC	14
STREAMMR	4
VIENNACL	8

TEST SETUP

HARDWARE SPECIFICATIONS AND BENCHMARKS SUITES



- ▲ 3.7 GHz AMD A10-7850K CPU
 - 32 GB of DDR3-1866
- ▲ AMD FirePro™ W9100 discrete GPU
 - 930 MHz core frequency
 - 320 GB/s of memory bandwidth
 - 16 GB of GDDR5 memory
- ▲ 3rd Generation PCIe® x8 CPU–GPU connection
- ▲ 175 benchmarks in 16 benchmark suites

☐ Detect GPU Buffer Overflows

☐ Compatible With Most OpenCL™

☐ Low Runtime Overhead

Suite	Num Benchmarks
AMDAPP	46
FINANCEBENCH	2
GPUSTREAM	1
HETEROMARK	14
MANTEVO	4
NPB_OCL	8
OPENDWARFS	7
PANNOTIA	6
PARBOIL	9
PHORONIX	4
POLYBENCH	21
PROXYAPPS	6
RODINIA	21
SHOC	14
STREAMMR	4
VIENNACL	8

TEST SETUP

HARDWARE SPECIFICATIONS AND BENCHMARKS SUITES



- ▲ 3.7 GHz AMD A10-7850K CPU
 - 32 GB of DDR3-1866
- ▲ AMD FirePro™ W9100 discrete GPU
 - 930 MHz core frequency
 - 320 GB/s of memory bandwidth
 - 16 GB of GDDR5 memory
- ▲ 3rd Generation PCIe® x8 CPU–GPU connection
- ▲ 175 benchmarks in 16 benchmark suites

☐ Detect GPU Buffer Overflows

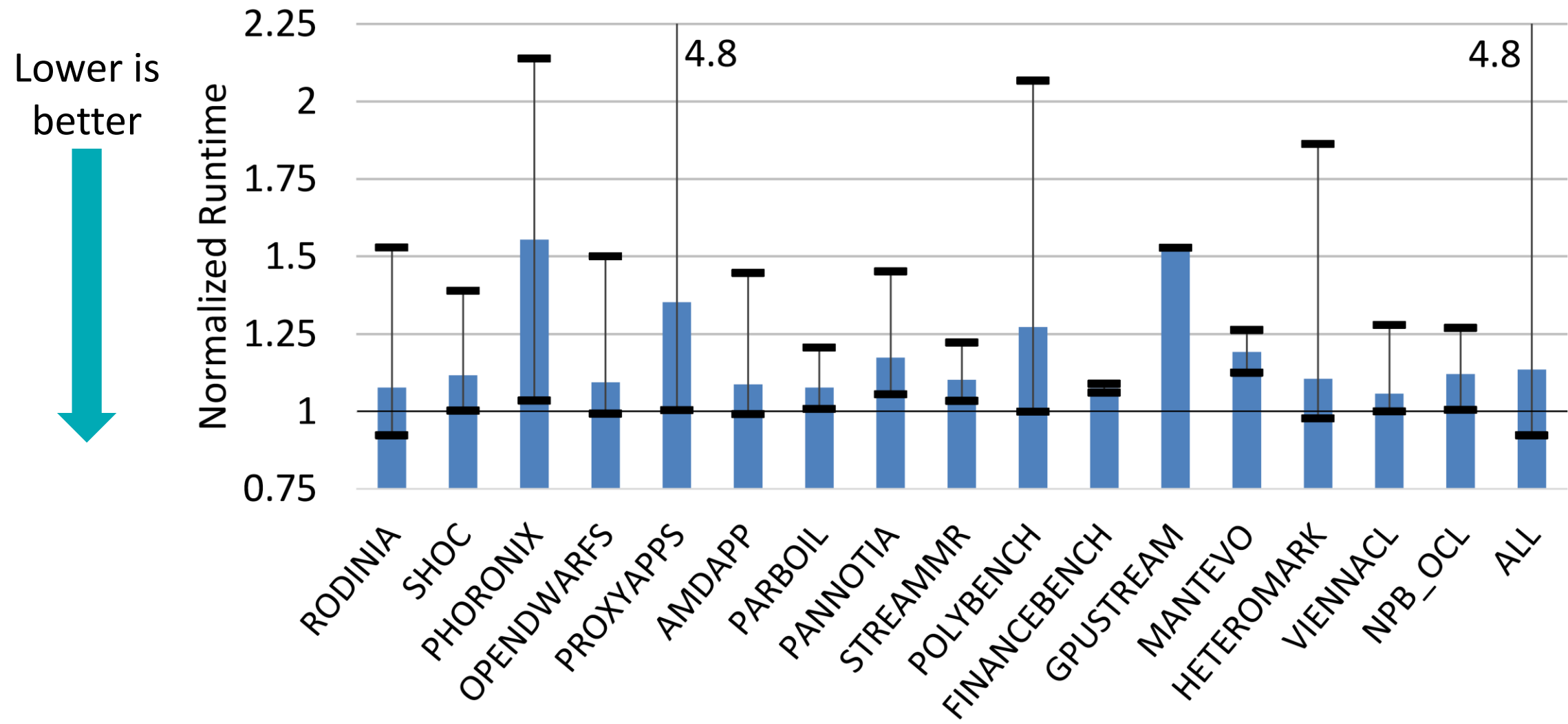
☒ Compatible With Most OpenCL™

☐ Low Runtime Overhead

Suite	Num Benchmarks
AMDAPP	46
FINANCEBENCH	2
GPUSTREAM	1
HETEROMARK	14
MANTEVO	4
NPB_OCL	8
OPENDWARFS	7
PANNOTIA	6
PARBOIL	9
PHORONIX	4
POLYBENCH	21
PROXYAPPS	6
RODINIA	21
SHOC	14
STREAMMR	4
VIENNACL	8

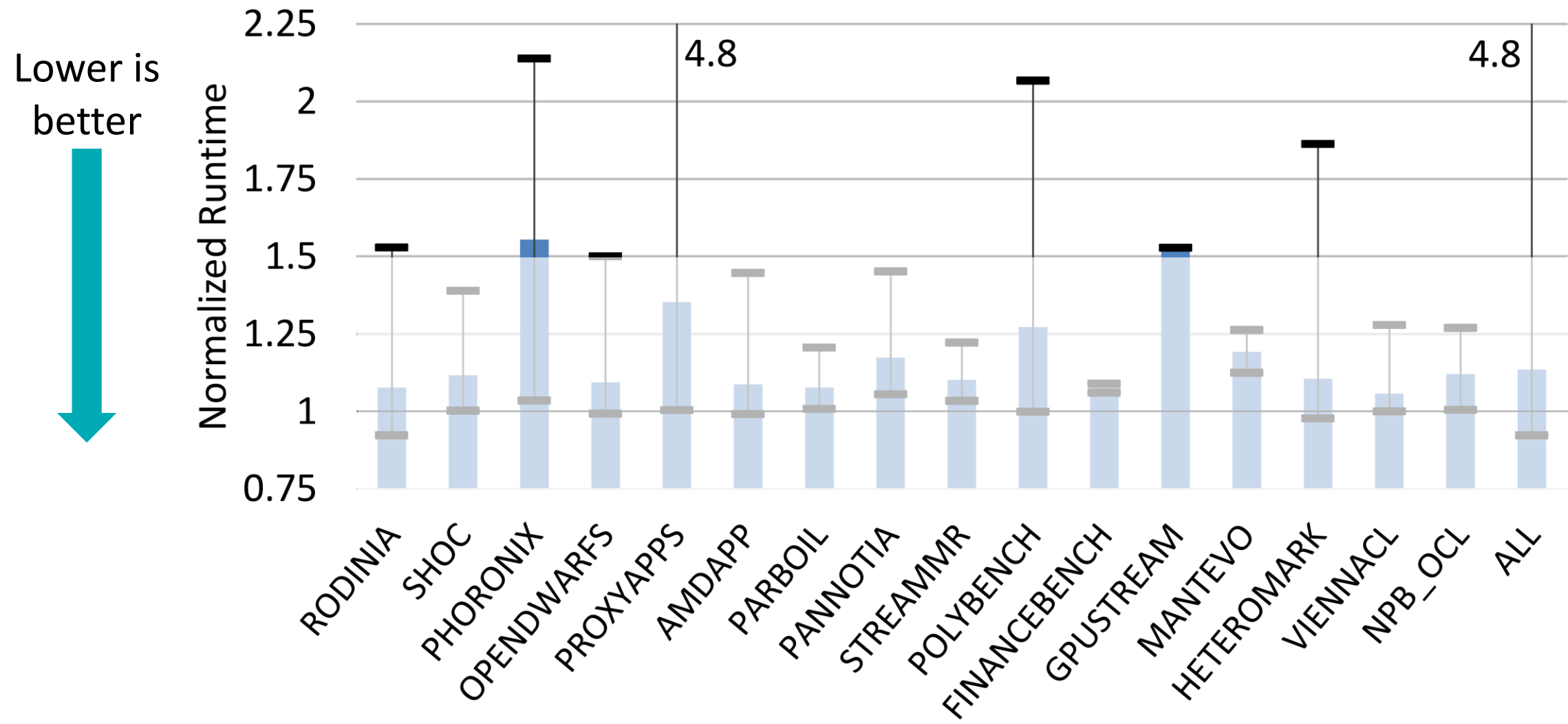
PERFORMANCE EVALUATION

APPLICATION RUNTIME: WITH / WITHOUT TOOL



PERFORMANCE EVALUATION

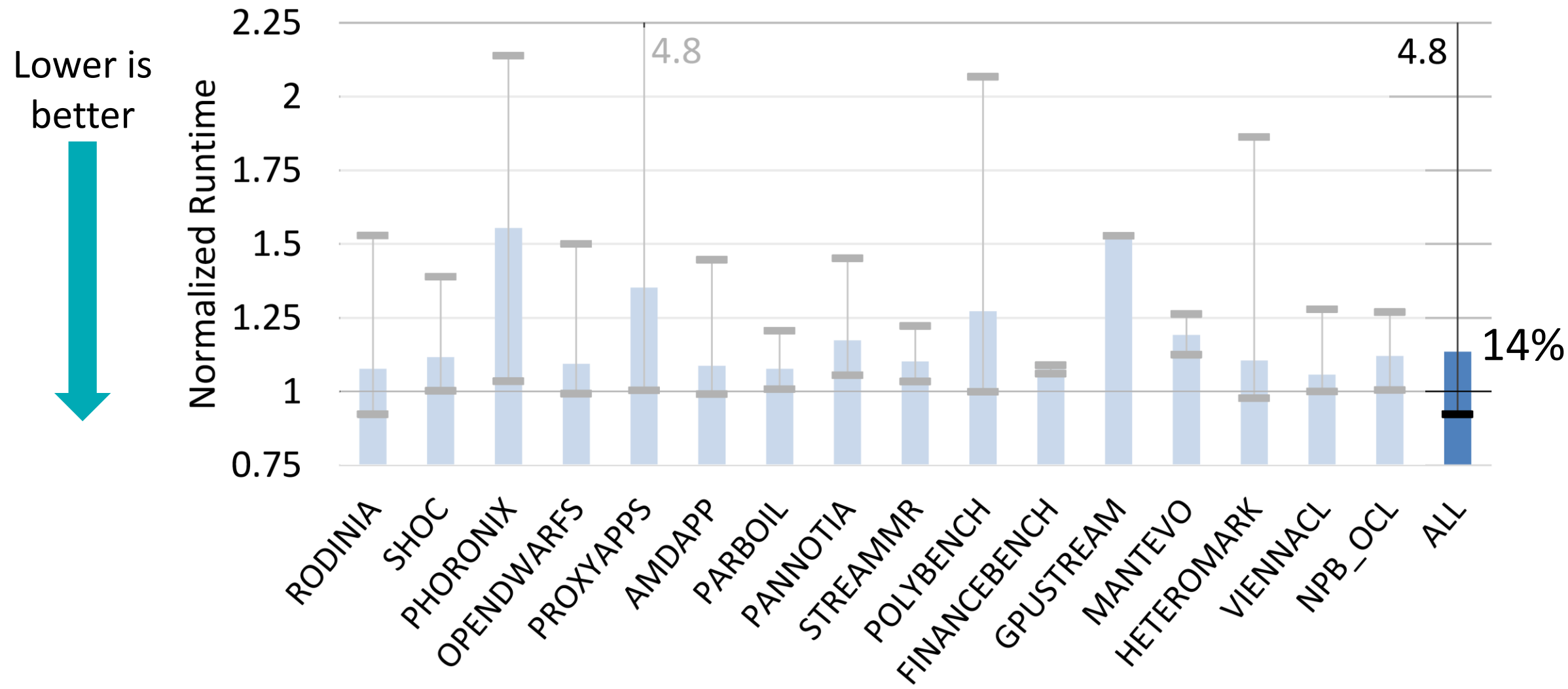
APPLICATION RUNTIME: WITH / WITHOUT TOOL



PERFORMANCE EVALUATION



APPLICATION RUNTIME: WITH / WITHOUT TOOL



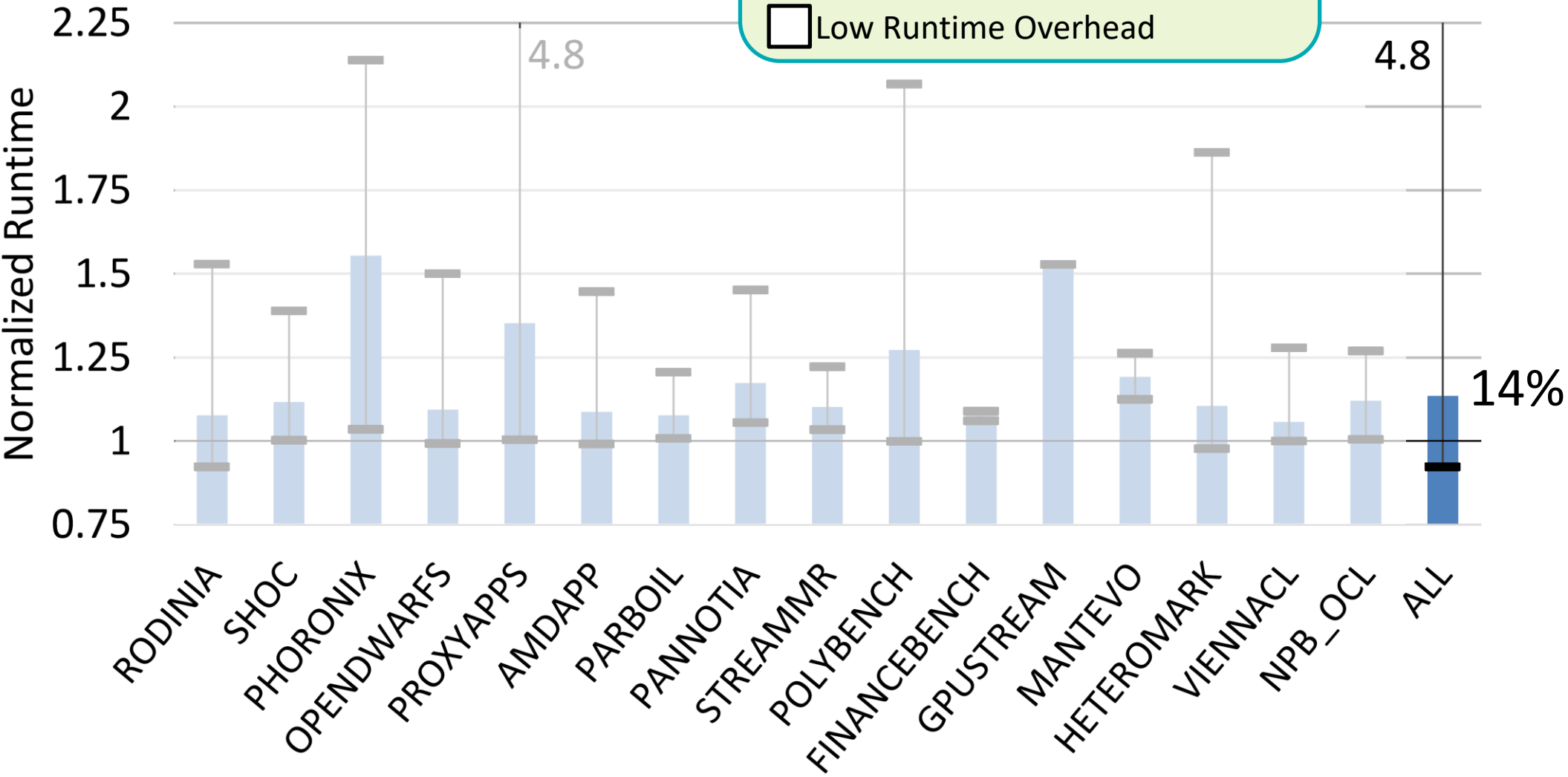
PERFORMANCE EVALUATION

APPLICATION RUNTIME: WITH / WITHOUT TOOL



- ☐ Detect GPU Buffer Overflows
- ☒ Compatible With Most OpenCL™
- ☐ Low Runtime Overhead

Lower is better



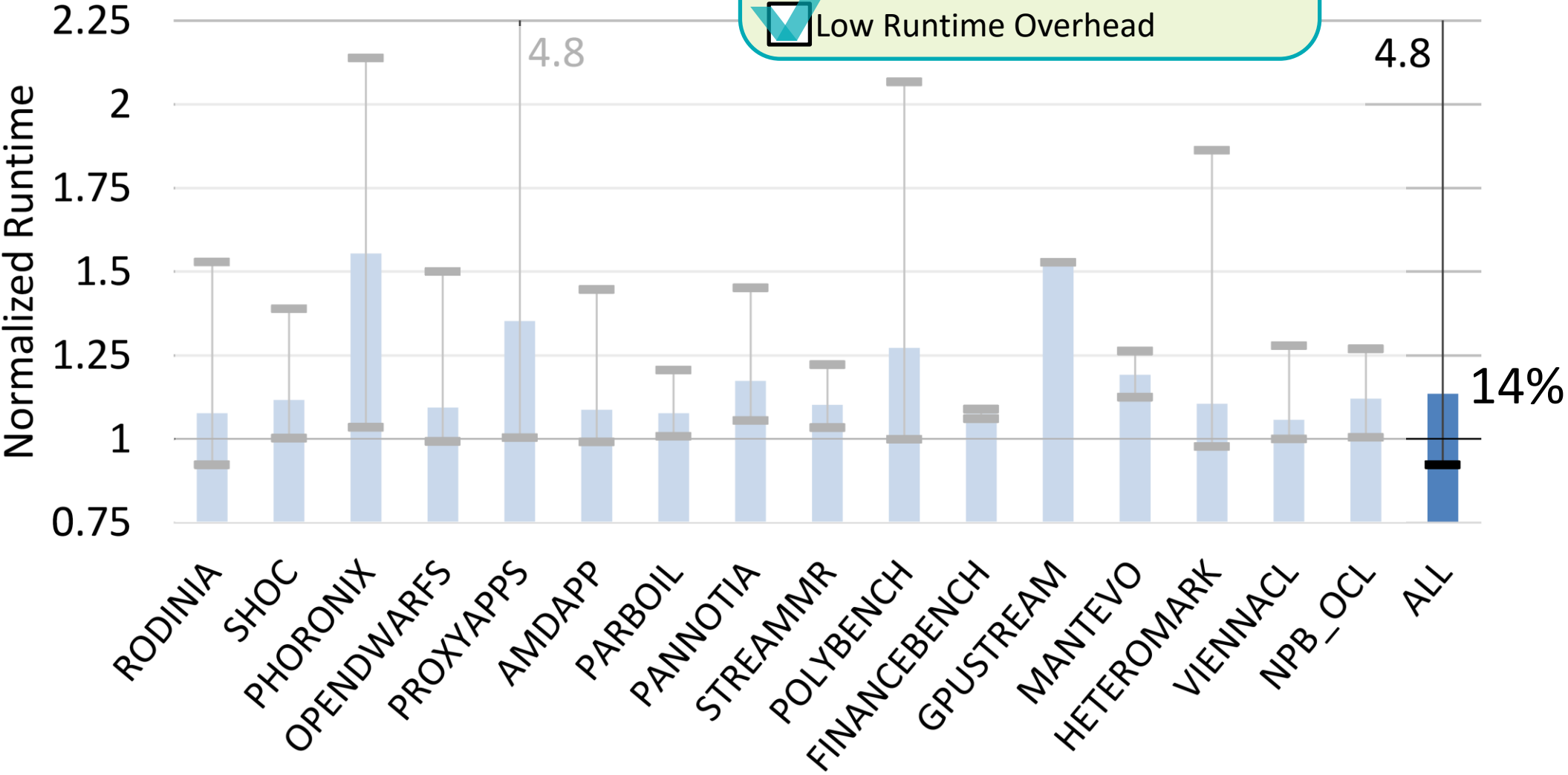
PERFORMANCE EVALUATION

APPLICATION RUNTIME: WITH / WITHOUT TOOL



- ☐ Detect GPU Buffer Overflows
- ☒ Compatible With Most OpenCL™
- ☒ Low Runtime Overhead

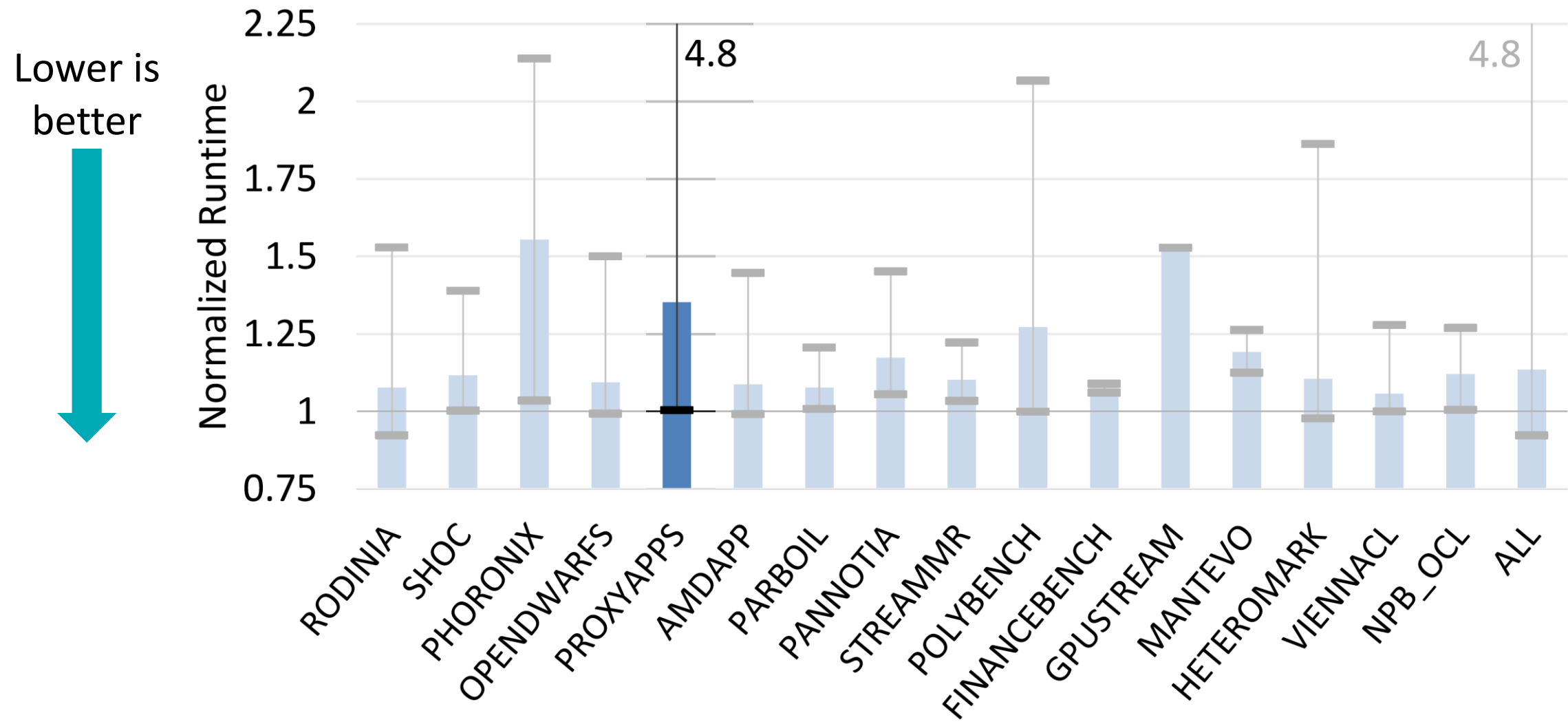
Lower is better



PERFORMANCE EVALUATION



APPLICATION RUNTIME: WITH / WITHOUT TOOL



ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



Example SNAP_MPI kernel launch

ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



Example SNAP_MPI kernel launch

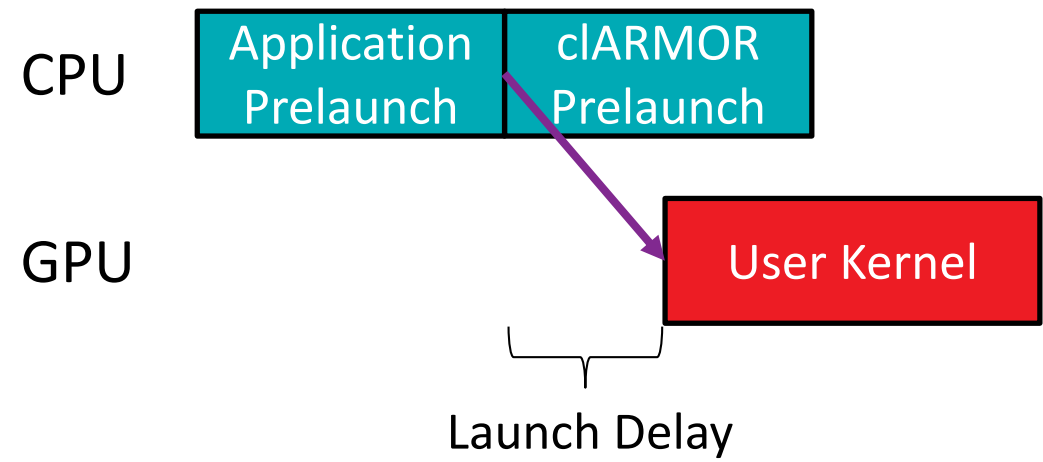
CPU Application
Prelaunch

GPU

ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



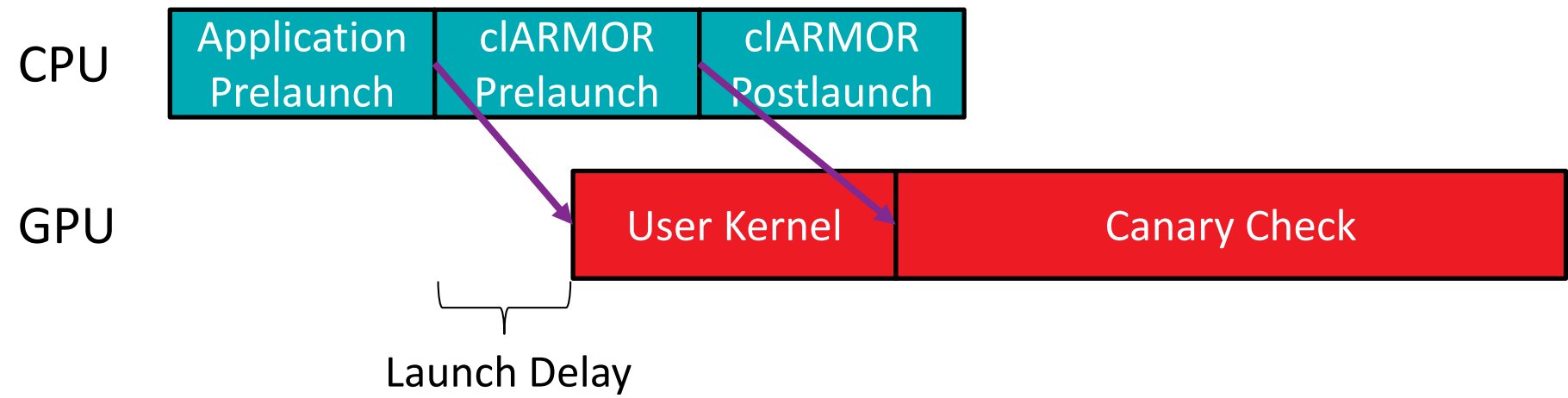
Example SNAP_MPI kernel launch



ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



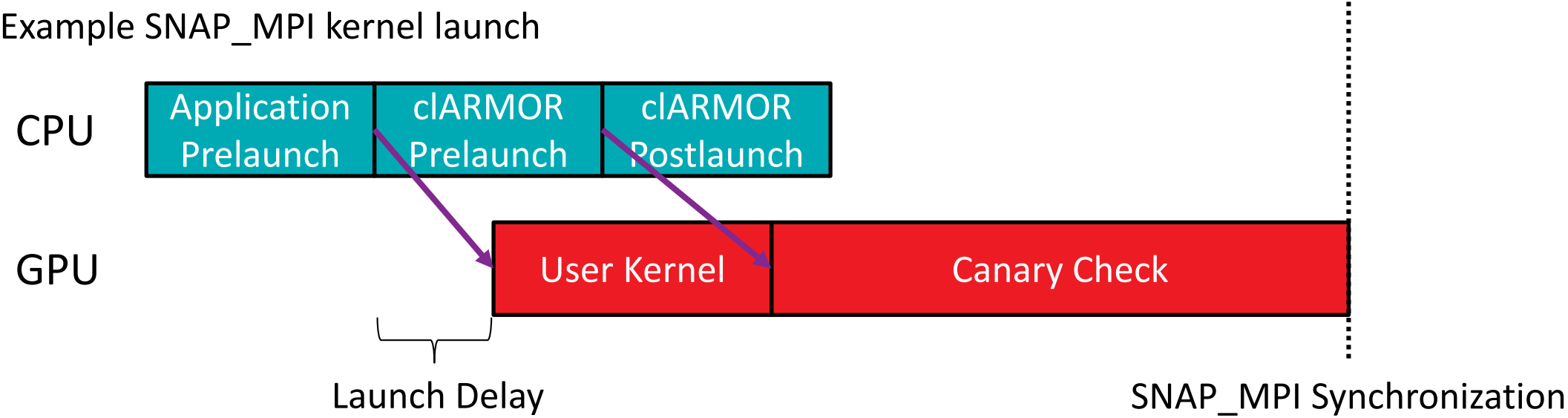
Example SNAP_MPI kernel launch



ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



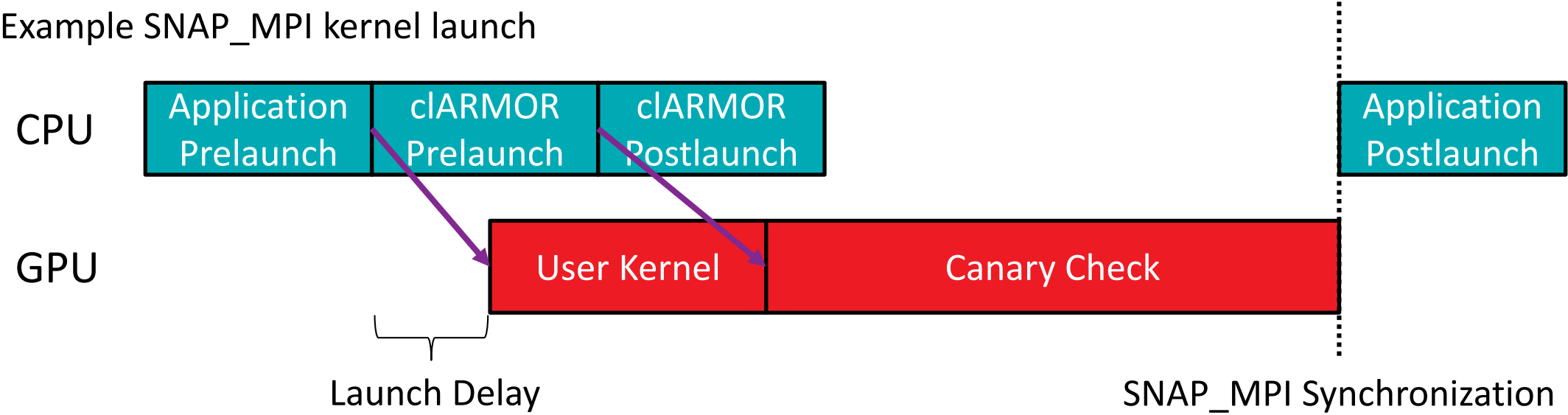
Example SNAP_MPI kernel launch



ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



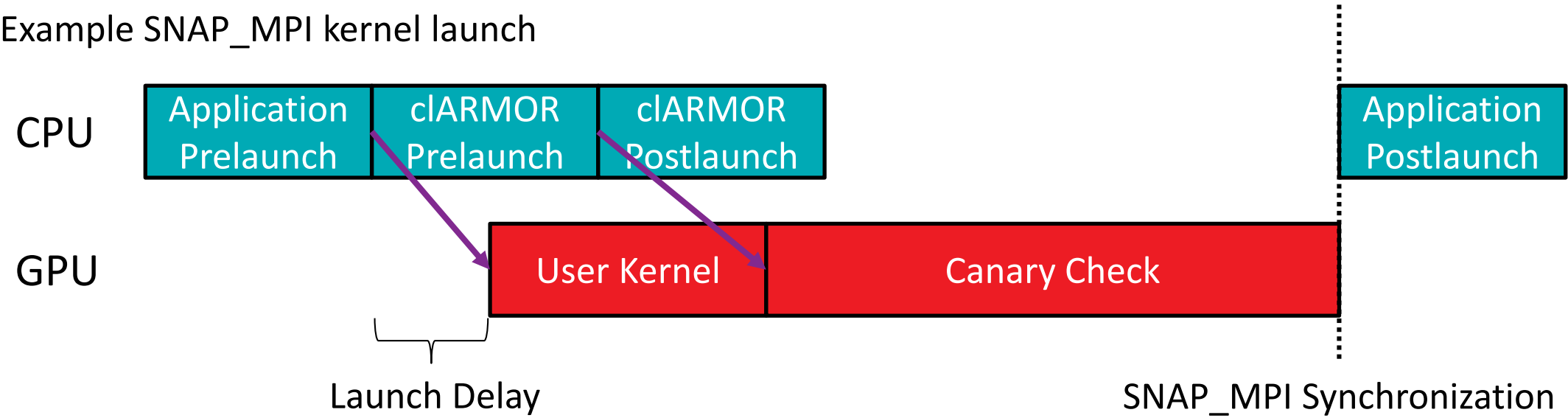
Example SNAP_MPI kernel launch



ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



Example SNAP_MPI kernel launch

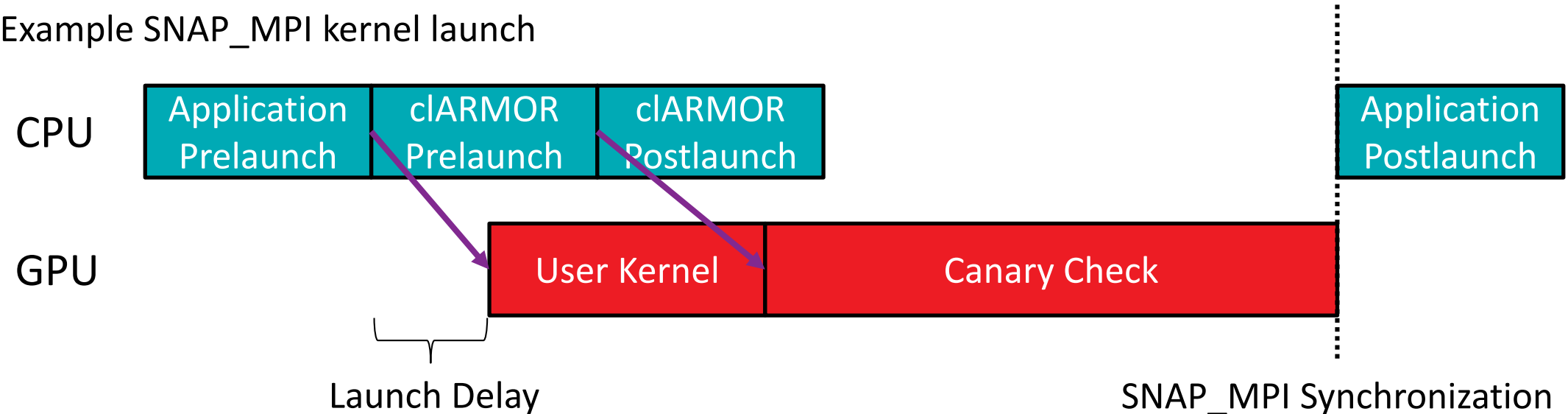


Possible improvement for SNAP_MPI kernel launch

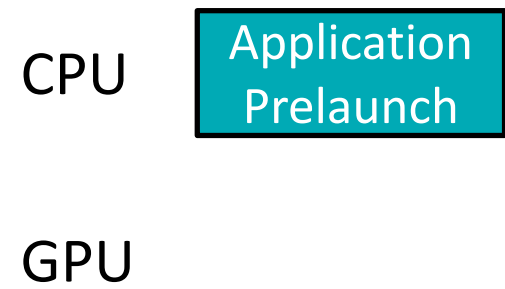
ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



Example SNAP_MPI kernel launch



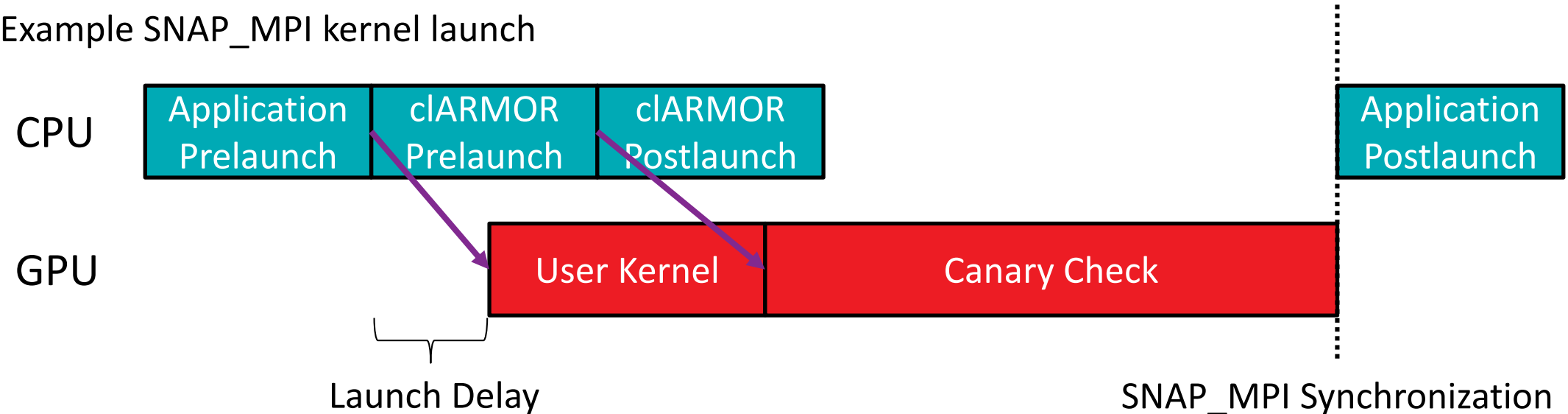
Possible improvement for SNAP_MPI kernel launch



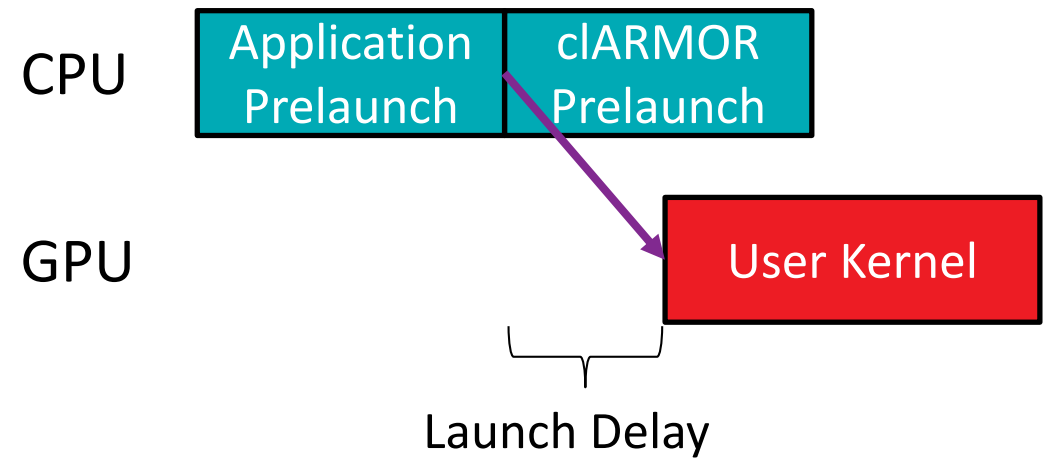
ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



Example SNAP_MPI kernel launch



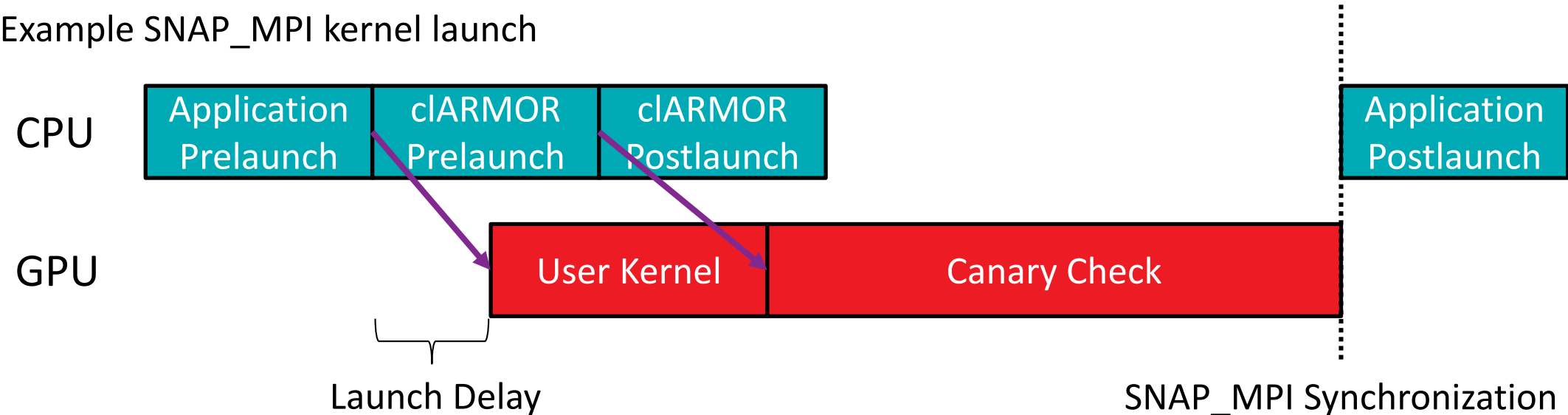
Possible improvement for SNAP_MPI kernel launch



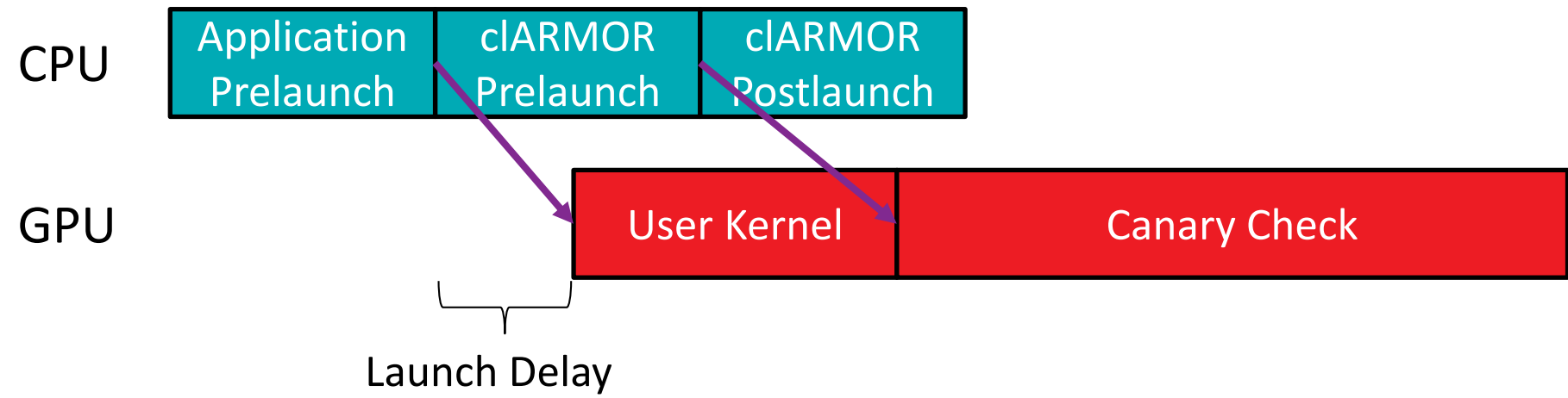
ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



Example SNAP_MPI kernel launch



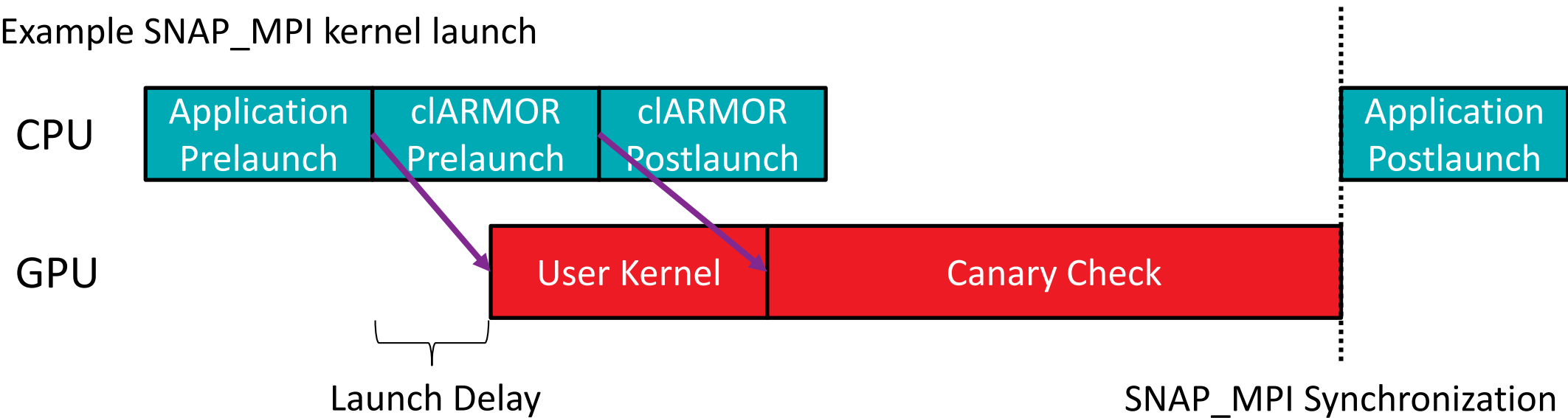
Possible improvement for SNAP_MPI kernel launch



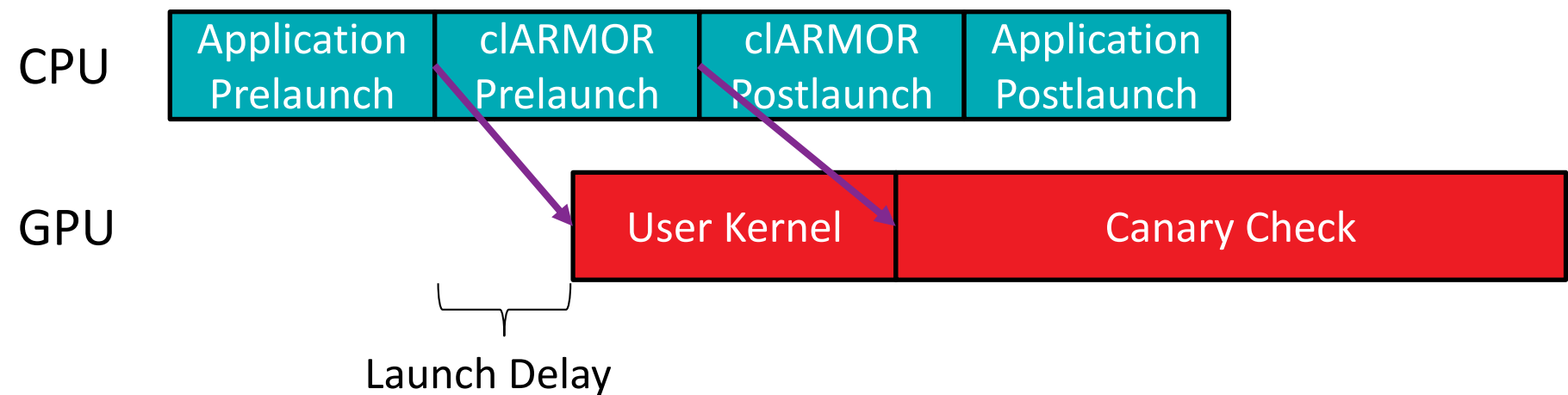
ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



Example SNAP_MPI kernel launch



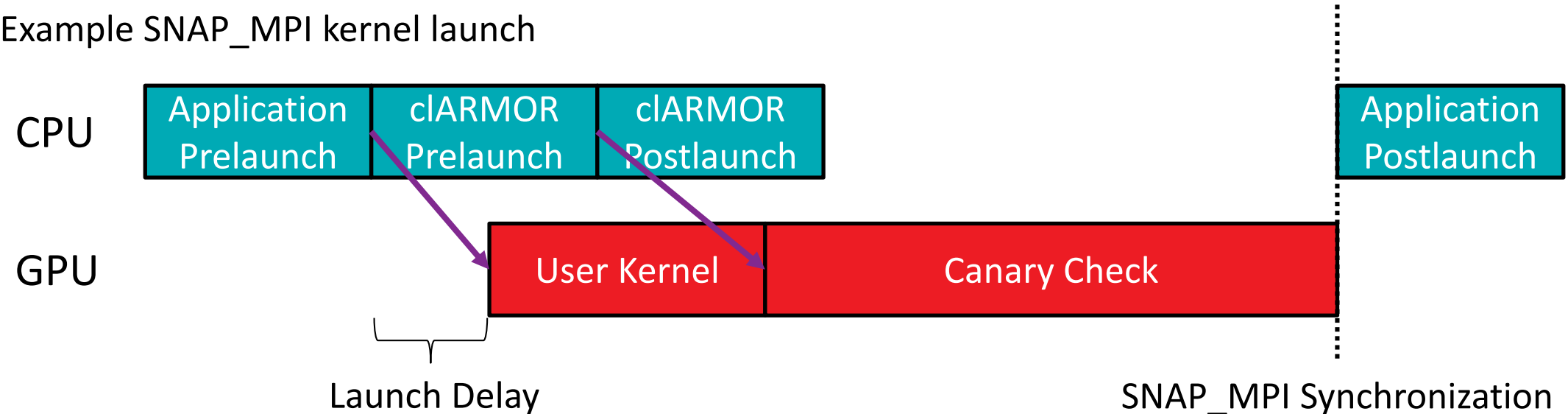
Possible improvement for SNAP_MPI kernel launch



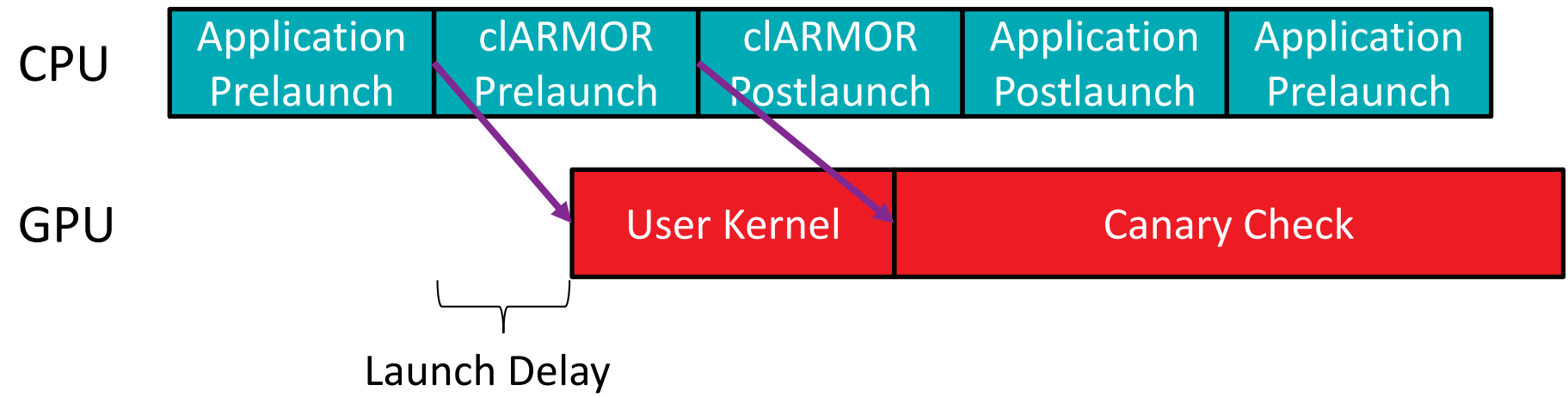
ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



Example SNAP_MPI kernel launch



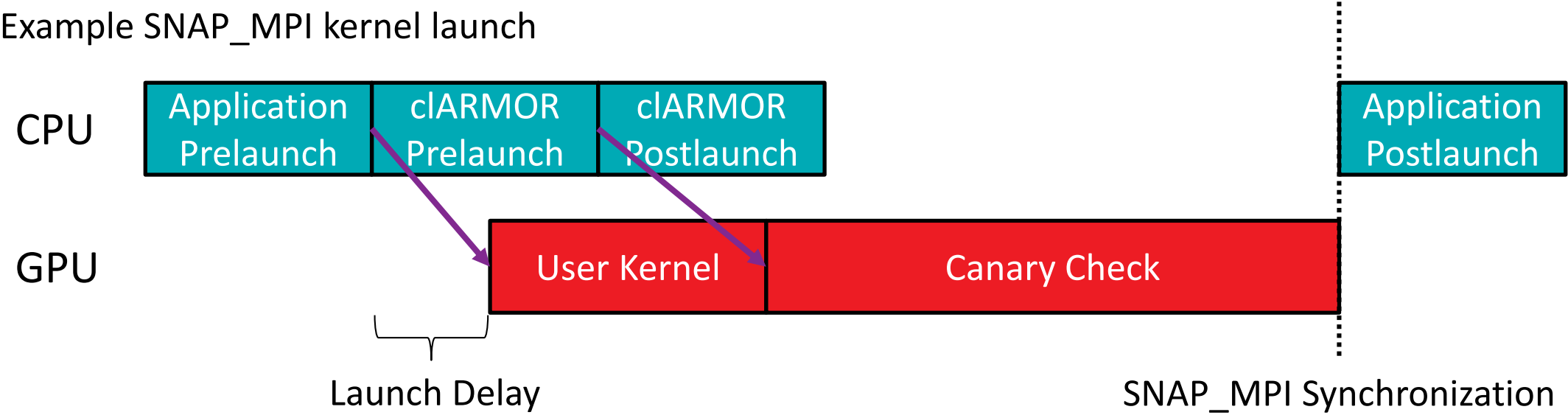
Possible improvement for SNAP_MPI kernel launch



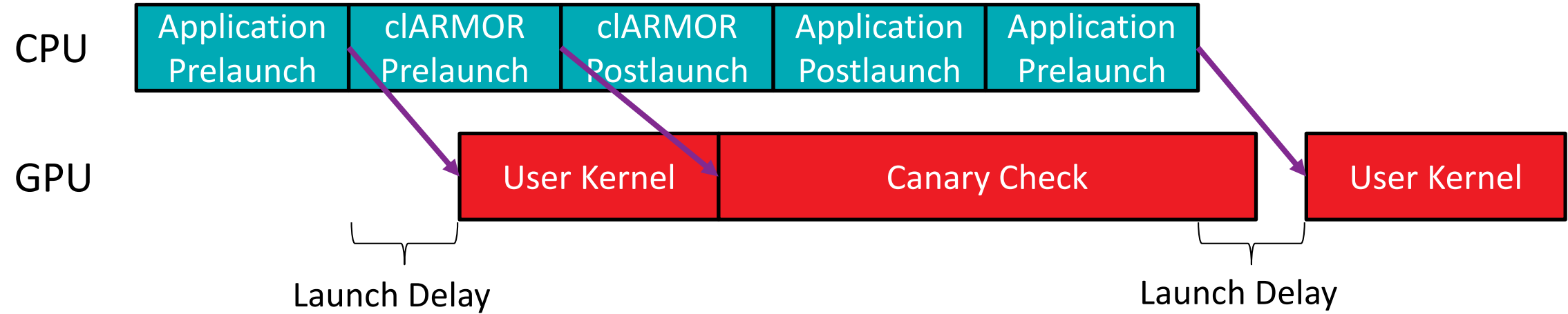
ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



Example SNAP_MPI kernel launch



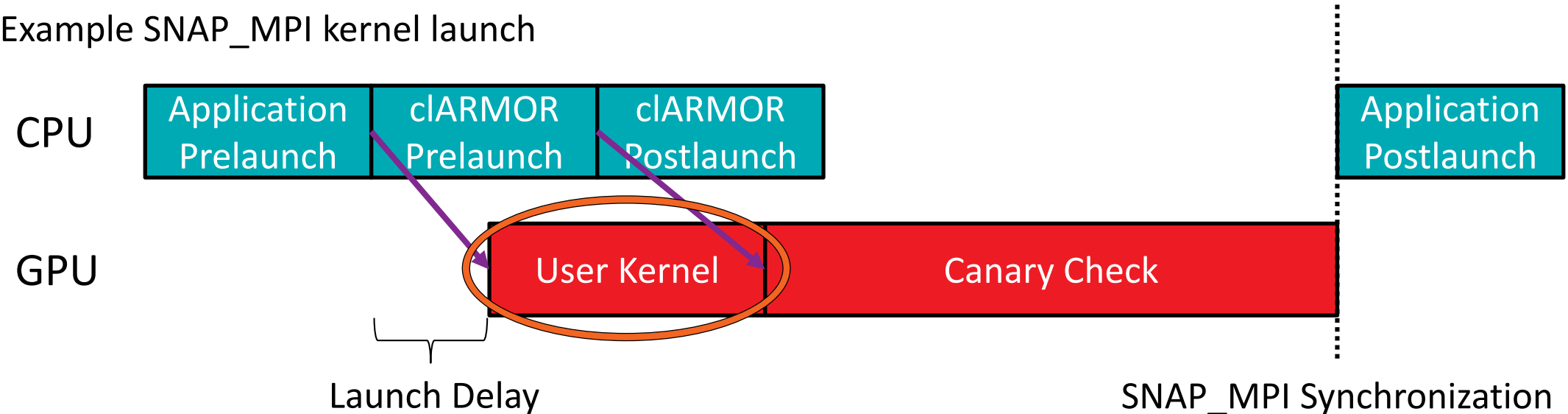
Possible improvement for SNAP_MPI kernel launch



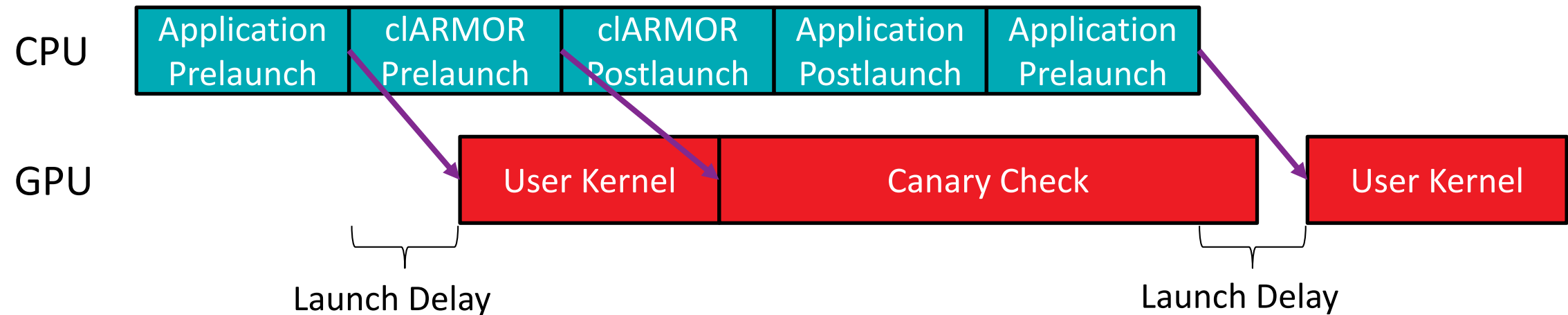
ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



Example SNAP_MPI kernel launch



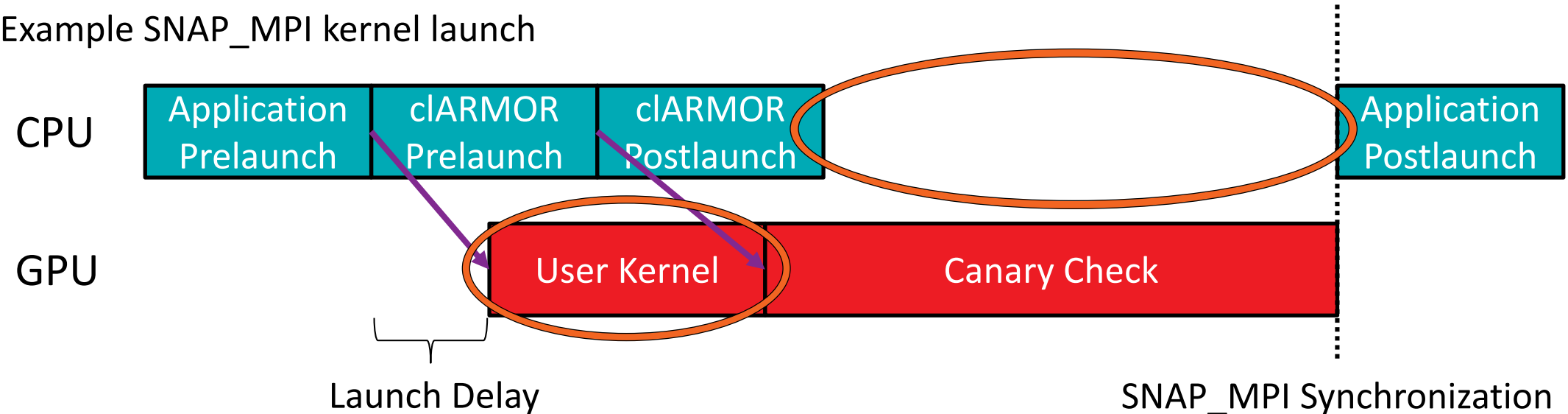
Possible improvement for SNAP_MPI kernel launch



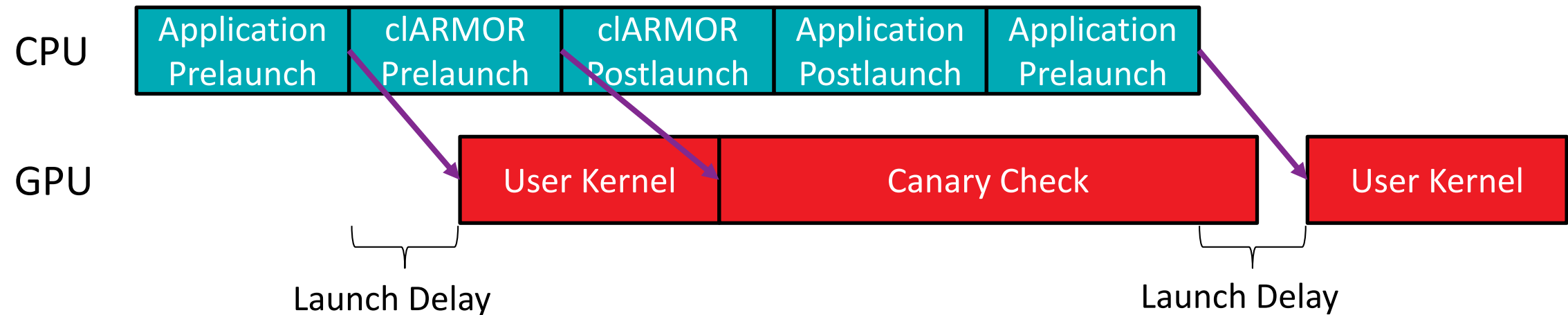
ANALYSIS OF TOOL OVERHEAD WITH SNAP_MPI



Example SNAP_MPI kernel launch



Possible improvement for SNAP_MPI kernel launch



CLARMOR DETECTION RESULTS

LIST OF BENCHMARKS WITH BUFFER OVERFLOWS



▲ Parboil

- mri-gridding

▲ StreamMR

- kmeans
- wordcount

▲ Hetero-Mark

- OpenCL™ 1.2 kmeans
- OpenCL 2.0 kmeans
- OpenCL 1.2 sw, 4 errors
- OpenCL 2.0 sw, 4 errors

CLARMOR DETECTION RESULTS

LIST OF BENCHMARKS WITH BUFFER OVERFLOWS



▲ Parboil

- mri-gridding

▲ StreamMR

- kmeans
- wordcount

▲ Hetero-Mark

- OpenCL™ 1.2 kmeans
- OpenCL 2.0 kmeans
- OpenCL 1.2 sw, 4 errors
- OpenCL 2.0 sw, 4 errors

- ☐ Detect GPU Buffer Overflows
- ☒ Compatible With Most OpenCL™
- ☒ Low Runtime Overhead

CLARMOR DETECTION RESULTS

LIST OF BENCHMARKS WITH BUFFER OVERFLOWS



▲ Parboil

- mri-gridding

▲ StreamMR

- kmeans
- wordcount

▲ Hetero-Mark

- OpenCL™ 1.2 kmeans
- OpenCL 2.0 kmeans
- OpenCL 1.2 sw, 4 errors
- OpenCL 2.0 sw, 4 errors

- ☒ Detect GPU Buffer Overflows
- ☒ Compatible With Most OpenCL™
- ☒ Low Runtime Overhead

CLARMOR DETECTION RESULTS

LIST OF BENCHMARKS WITH BUFFER OVERFLOWS



▲ Parboil

- mri-gridding

▲ StreamMR

- kmeans
- wordcount

▲ Hetero-Mark

- OpenCL™ 1.2 kmeans
- OpenCL 2.0 kmeans
- OpenCL 1.2 sw, 4 errors
- OpenCL 2.0 sw, 4 errors



- ☒ Detect GPU Buffer Overflows
- ☒ Compatible With Most OpenCL™
- ☒ Low Runtime Overhead

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...
    const unsigned M_LEN,
    ...
    __global double *cu,
    ... ) {
    int x = get_global_id(0);
    int y = get_global_id(1);
    cu[(y + 1) * M_LEN + x] = <input_equation>
    ...
}
```

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;
...
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,
    sizeInBytes, NULL, &err);
...
const size_t globalSize[2] = {m_len_, n_len_};
...
err |= clSetKernelArg(kernel_sw_compute0_, 6,
    sizeof(cl_mem),
    reinterpret_cast<void *>(&cu_));
...
err = clEnqueueNDRangeKernel(cmdQueue_,
    kernel_sw_compute0_, 2, NULL, globalSize,
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```


Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

x = m - 1

y = n - 1

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```


Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

x = m - 1

y = n - 1

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...
    const unsigned M_LEN,
    ...
    __global double *cu,
    ... ) {
    int x = get_global_id(0);
    int y = get_global_id(1);
    cu[(y + 1) * M_LEN + x] = <input_equation>
    ...
}
```

$(y + 1) * M_LEN + x$

$x = m - 1$

$y = n - 1$

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;
...
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,
    sizeInBytes, NULL, &err);
...
const size_t globalSize[2] = {m_len_, n_len_};
...
err |= clSetKernelArg(kernel_sw_compute0_, 6,
    sizeof(cl_mem),
    reinterpret_cast<void *>(&cu_));
...
err = clEnqueueNDRangeKernel(cmdQueue_,
    kernel_sw_compute0_, 2, NULL, globalSize,
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...
    const unsigned M_LEN,
    ...
    __global double *cu,
    ... ) {
    int x = get_global_id(0);
    int y = get_global_id(1);
    cu[(y + 1) * M_LEN + x] = <input_equation>
    ...
}
```

$(y + 1) * M_LEN + x$

$x = m - 1$

$y = n - 1$

➡ $m == M_LEN$

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;
...
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,
    sizeInBytes, NULL, &err);
...
const size_t globalSize[2] = {m_len_, n_len_};
...
err |= clSetKernelArg(kernel_sw_compute0_, 6,
    sizeof(cl_mem),
    reinterpret_cast<void *>(&cu_));
...
err = clEnqueueNDRangeKernel(cmdQueue_,
    kernel_sw_compute0_, 2, NULL, globalSize,
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

$(y + 1) * M_LEN + x$

$(y + 1) * m + x$

$x = m - 1$

$y = n - 1$

➡ $m == M_LEN$

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...
    const unsigned M_LEN,
    ...
    __global double *cu,
    ... ) {
    int x = get_global_id(0);
    int y = get_global_id(1);
    cu[(y + 1) * M_LEN + x] = <input_equation>
    ...
}
```

$(y + 1) * M_LEN + x$

$(y + 1) * m + x$

$x = m - 1$

→ $y = n - 1$

$m == M_LEN$

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;
...
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,
    sizeInBytes, NULL, &err);
...
const size_t globalSize[2] = {m_len_, n_len_};
...
err |= clSetKernelArg(kernel_sw_compute0_, 6,
    sizeof(cl_mem),
    reinterpret_cast<void *>(&cu_));
...
err = clEnqueueNDRangeKernel(cmdQueue_,
    kernel_sw_compute0_, 2, NULL, globalSize,
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

$(y + 1) * M_LEN + x$

$(y + 1) * m + x$

$(n) * m + x$

$x = m - 1$

→ $y = n - 1$

$m == M_LEN$

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

$(y + 1) * M_LEN + x$

$(y + 1) * m + x$

$(n) * m + x$

➡ $x = m - 1$
 $y = n - 1$
 $m == M_LEN$

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...
    const unsigned M_LEN,
    ...
    __global double *cu,
    ... ) {
    int x = get_global_id(0);
    int y = get_global_id(1);
    cu[(y + 1) * M_LEN + x] = <input_equation>
    ...
}
```

$(y + 1) * M_LEN + x$

$(y + 1) * m + x$

$(n) * m + x$

$n * m + m - 1$

➡ $x = m - 1$
 $y = n - 1$
 $m == M_LEN$

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;
...
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,
    sizeInBytes, NULL, &err);
...
const size_t globalSize[2] = {m_len_, n_len_};
...
err |= clSetKernelArg(kernel_sw_compute0_, 6,
    sizeof(cl_mem),
    reinterpret_cast<void *>(&cu_));
...
err = clEnqueueNDRangeKernel(cmdQueue_,
    kernel_sw_compute0_, 2, NULL, globalSize,
    localSize, 0, NULL, NULL);
```


Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

$(y + 1) * M_LEN + x$

$(y + 1) * m + x$

$(n) * m + x$

$n * m + m - 1$

$x = m - 1$

$y = n - 1$

$m == M_LEN$

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

$(y + 1) * M_LEN + x$

$(y + 1) * m + x$

$(n) * m + x$

$n * m + m - 1$

$m*n - 1 + m > m*n - 1$

$x = m - 1$

$y = n - 1$

$m == M_LEN$

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

$(y + 1) * M_LEN + x$

$(y + 1) * m + x$

$(n) * m + x$

$n * m + m - 1$

~~$m * n - 1$~~ + m > ~~$m * n - 1$~~

$x = m - 1$

$y = n - 1$

$m == M_LEN$

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...  
    const unsigned M_LEN,  
    ...  
    __global double *cu,  
    ... ) {  
  
    int x = get_global_id(0);  
    int y = get_global_id(1);  
    cu[(y + 1) * M_LEN + x] = <input_equation>  
    ...  
}
```

$(y + 1) * M_LEN + x$

$(y + 1) * m + x$

$(n) * m + x$

$n * m + m - 1$

~~$m * n - 1 + m > m * n - 1$~~

$m > 0$

$x = m - 1$

$y = n - 1$

$m == M_LEN$

Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;  
...  
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,  
    sizeInBytes, NULL, &err);  
...  
const size_t globalSize[2] = {m_len_, n_len_};  
...  
err |= clSetKernelArg(kernel_sw_compute0_, 6,  
    sizeof(cl_mem),  
    reinterpret_cast<void *>(&cu_));  
...  
err = clEnqueueNDRangeKernel(cmdQueue_,  
    kernel_sw_compute0_, 2, NULL, globalSize,  
    localSize, 0, NULL, NULL);
```

Hetero-Mark OpenCL™ 1.2 SW Overflow Error



Kernel

```
__kernel void sw_compute0(...
    const unsigned M_LEN,
    ...
    __global double *cu,
    ... ) {

    int x = get_global_id(0);
    int y = get_global_id(1);
    cu[(y + 1) * M_LEN + x] = <input_equation>
    ...
}
```

$(y + 1) * M_LEN + x$

$(y + 1) * m + x$

$(n) * m + x$

$n * m + m - 1$

~~$m * n - 1$~~ + m > ~~$m * n - 1$~~

$m > 0$

$x = m - 1$

$y = n - 1$

$m == M_LEN$



Host

```
size_t sizeInBytes = sizeof(double) * m_len_ * n_len_;
...
cu_ = clCreateBuffer(context_, CL_MEM_READ_WRITE,
    sizeInBytes, NULL, &err);
...
const size_t globalSize[2] = {m_len_, n_len_};
...
err |= clSetKernelArg(kernel_sw_compute0_, 6,
    sizeof(cl_mem),
    reinterpret_cast<void *>(&cu_));
...
err = clEnqueueNDRangeKernel(cmdQueue_,
    kernel_sw_compute0_, 2, NULL, globalSize,
    localSize, 0, NULL, NULL);
```

CONCLUSION

CLARMOR IS READY FOR YOU TO USE



- ▲ Canary-based detection scheme finds GPU write overflows
 - 13 GPU buffer overflows in 7 programs
- ▲ Works for most OpenCL™ applications
 - Running on GPU or CPU, not vendor specific
- ▲ Near real time detection
 - 14% overhead across 175 applications in 16 GPU benchmark suites
- ▲ Open Sourced
 - <https://github.com/GPUOpen-ProfessionalCompute-Tools/clARMOR>
 - Branch available for reproducing paper measurements

DISCLAIMER & ATTRIBUTION



The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

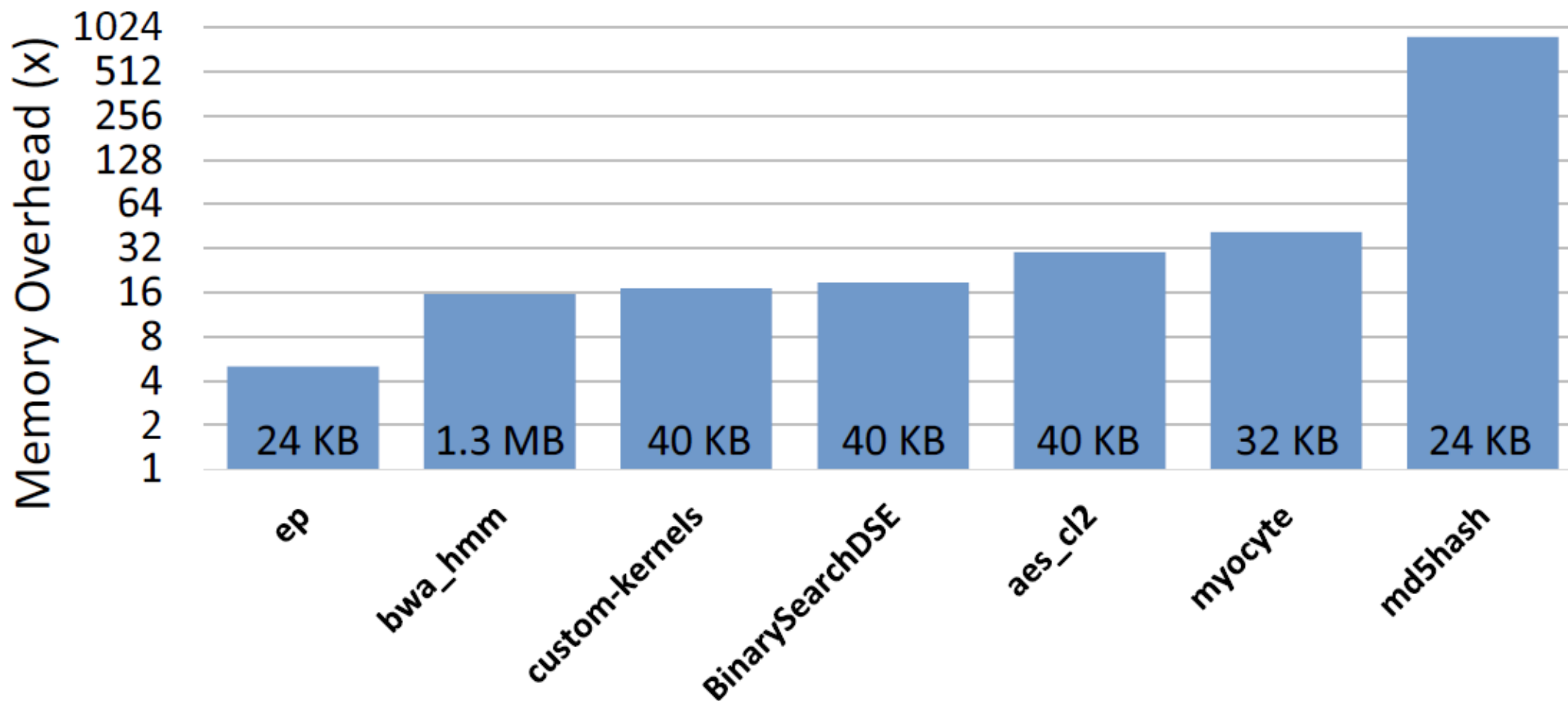
ATTRIBUTION

© 2017 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, AMD FirePro, and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. OpenCL is a trademark of Apple Inc. used by permission by Khronos. PCIe is a registered trademark of PCI-SIG Corporation. Linux is a registered trademark of Linus Torvalds. Other names are for informational purposes only and may be trademarks of their respective owners.



AMD





EXAMPLE ERROR



```
clARMOR: Loaded CL_WRAPPER
clARMOR:
clARMOR: ATTENTION:
clARMOR: ***** Buffer overflow detected *****
clARMOR: Kernel: sw_compute0, Buffer: cu
clARMOR:     First observed writing 1 byte(s) past the end.
clARMOR:
clARMOR: Exiting application because of buffer overflow.
```