PANOPTES: A BINARY TRANSLATION FRAMEWORK FOR CUDA

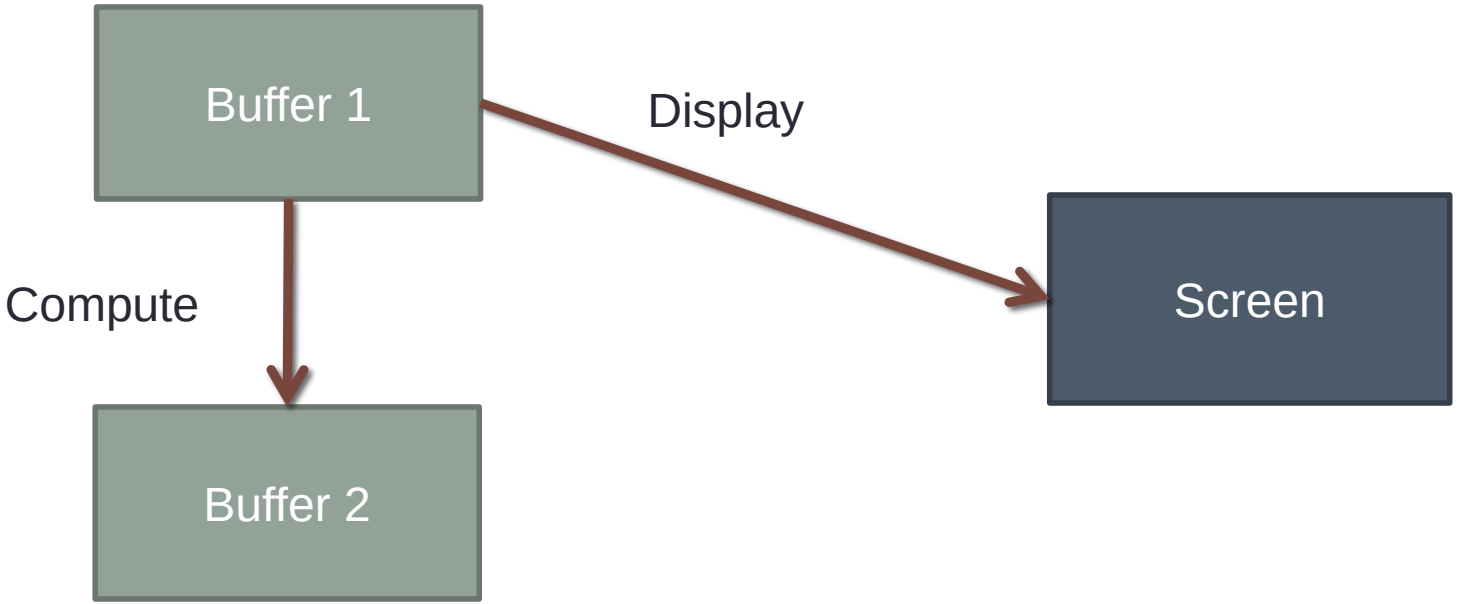
Chris Kennelly

D. E. Shaw Research

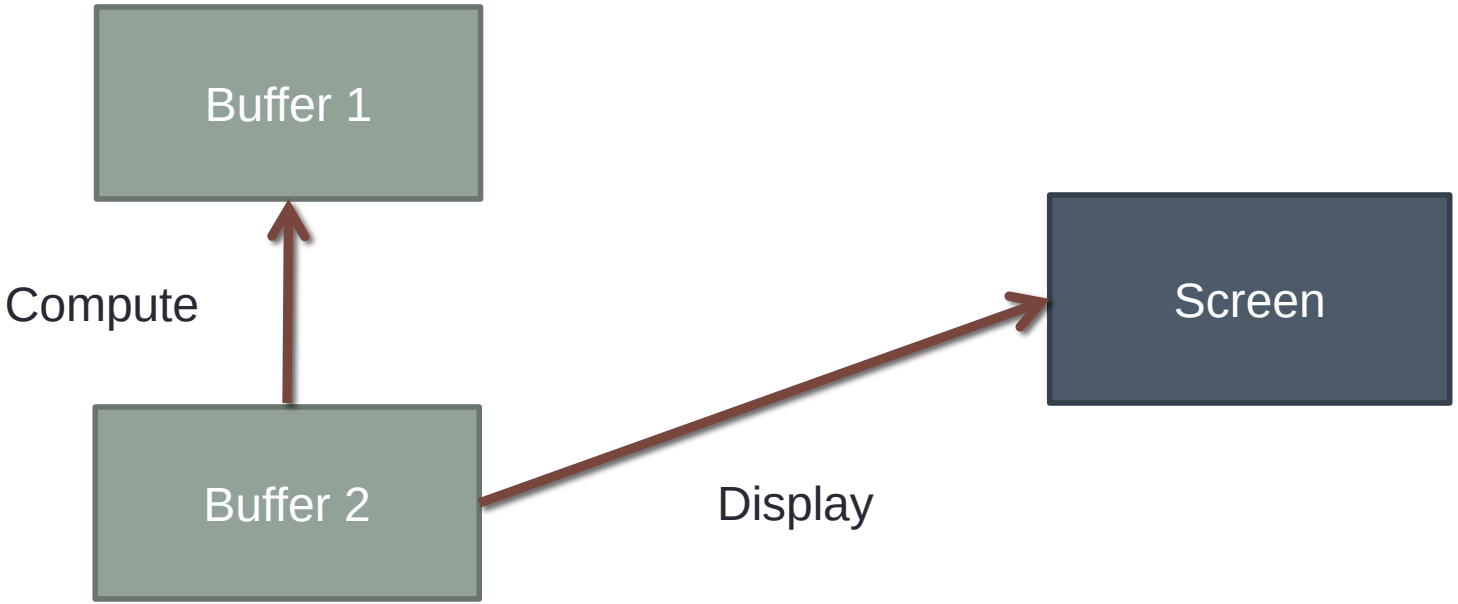
Outline

- The Motivating Problems
- Binary Translation as a Solution
- Results of Panoptes
- Future Work

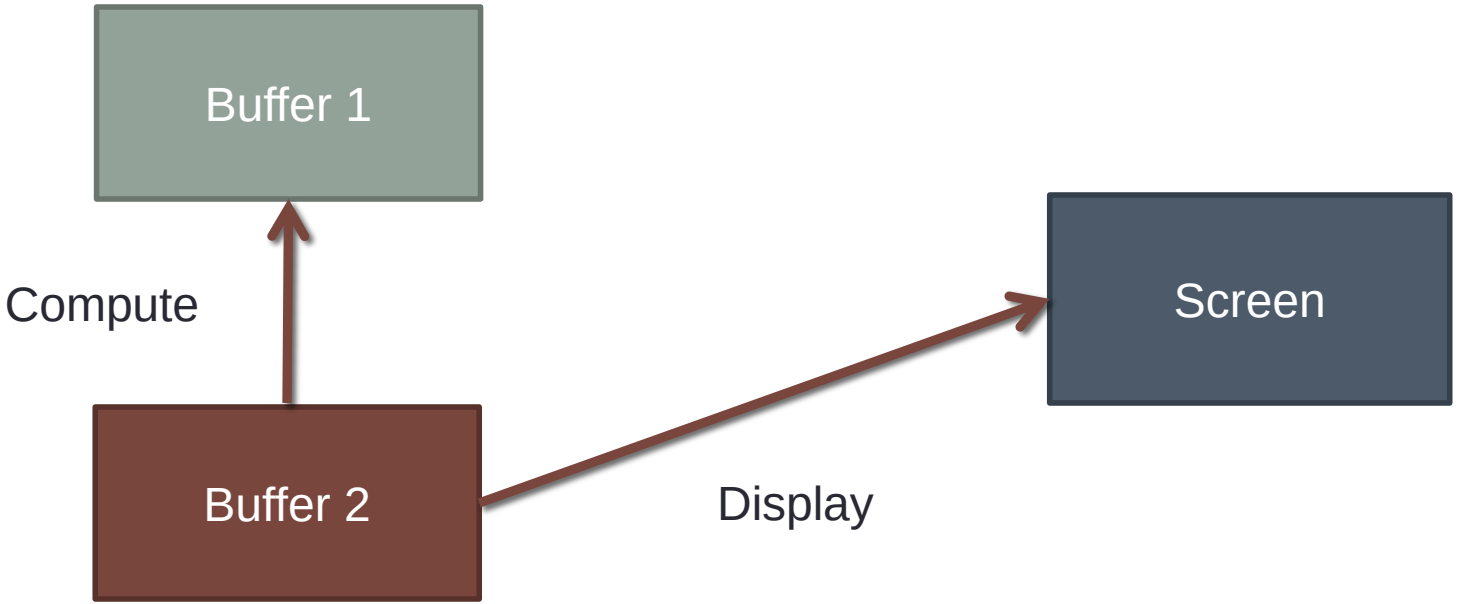
My Story: Buffer Ping-Ponging



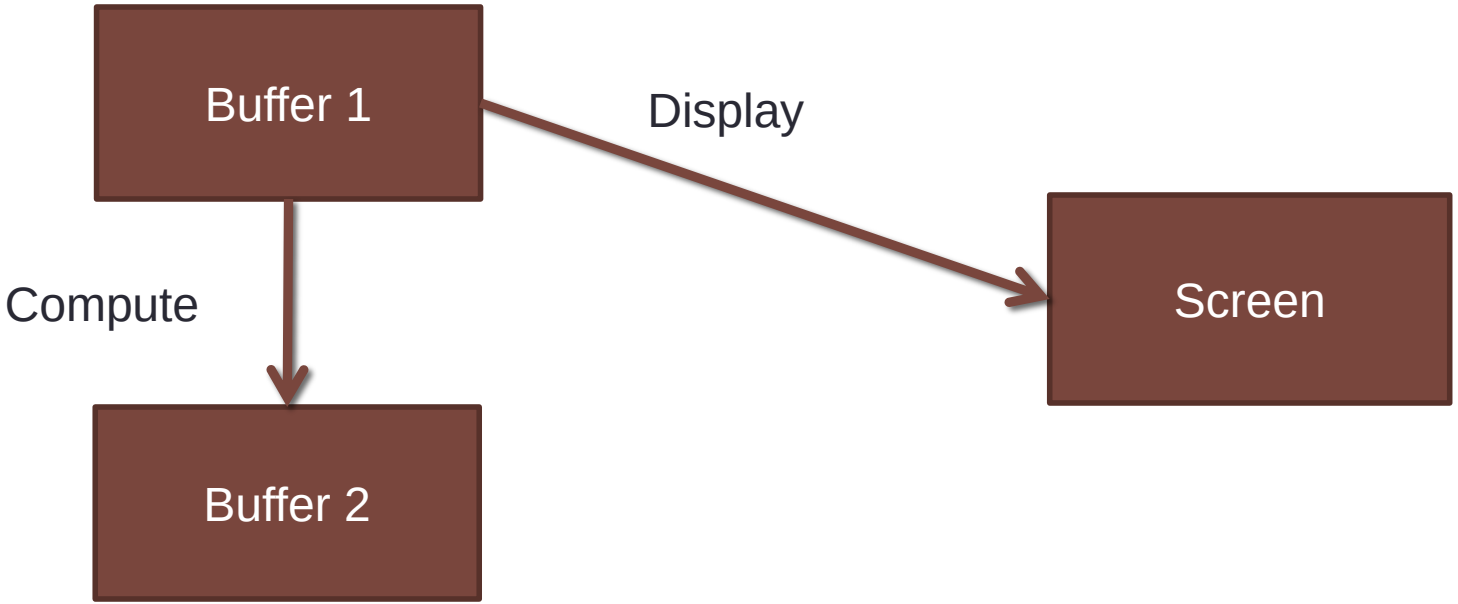
My Story: Buffer Ping-Ponging



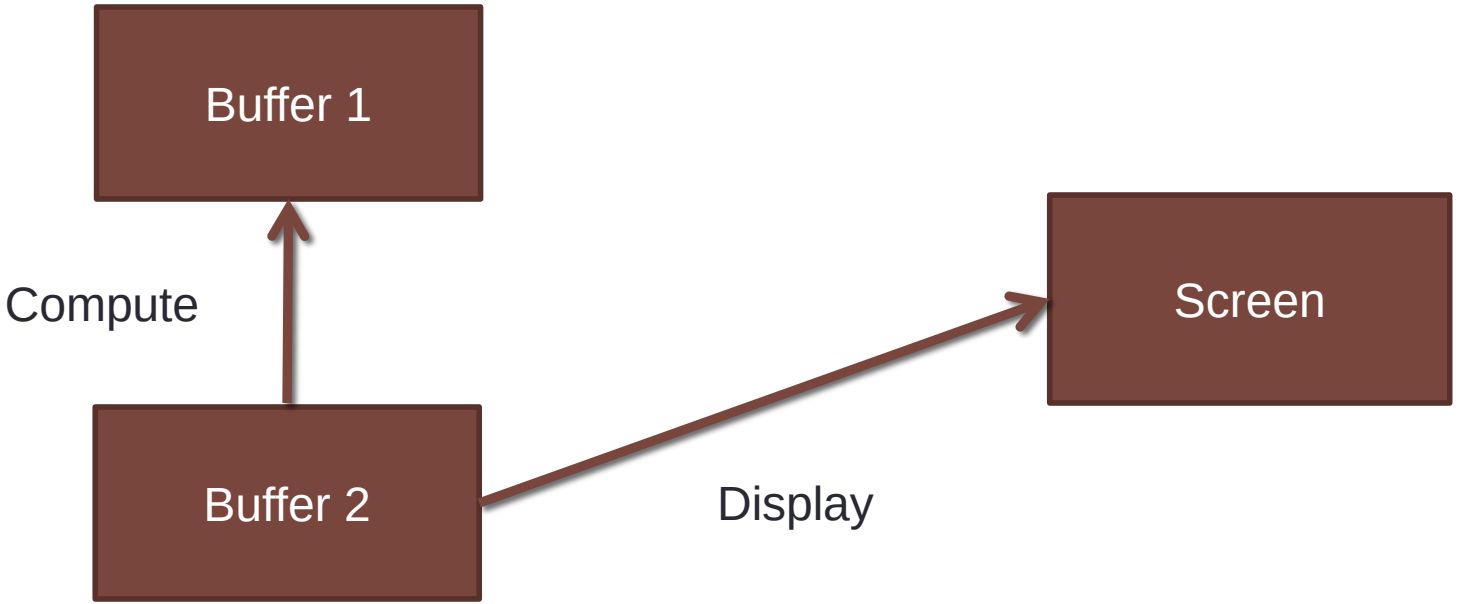
My Story: Buffer Ping-Ponging



My Story: Buffer Ping-Ponging



My Story: Buffer Ping-Ponging



Memory Overruns

```
k_simple(int * d, int n) {
 int tid = threadIdx.x + blockIdx.x * blockDim.x;
 for (int i = tid; i <= n;
     i += blockDim.x * gridDim.x){
  d[i] = i;
 }
}
```

Oops

- We didn’t mean to use `i <= n`.
- cuda-memcheck runs successfully.
  - cudaMalloc may have overallocated a buffer.
  - We used our own buffer management and overran an unrelated memory location

Inserting Bounds Checks

- Tedious
- Correctness
- Maintainability

Uninitialized Values

- Even more tedious
- Harder to maintain

When To Worry…

- Upon allocation? No.
- Upon access? No.

Worry when it affects program behavior.

Uninitialized Variable Example

```
k_simple(int * out, int * in, int * mask,
     int n) {
 int sum = 0;
 for (int i = threadIdx.x; i < n; i++){
  if (mask[i]) { sum += in[i]; }
 }
 out[threadIdx.x] = sum;
}
```

Sentinel Values

- Fill memory regions after allocation with a sentinel value to indicate it is not initialized.
- The chosen value must have no legitimate use.
- Every possible use point must check for the sentinel.

```
__global__ void
k_sum(int * sum, const int * in, int n) {
   int tsum = 0;
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   for (int i = tid; i < n; i += blockDim.x * gridDim.x) {
      tsum += in[i];
   }
   sum[threadIdx.x] = tsum;
}
```

```
__global__ void
k_scale(int * out, const int * in, int n) {
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   for (int i = tid; i < n; i += blockDim.x * gridDim.x) {
      out = in[n] >> 1;
   }
}
```

Checks for sentinel values are now lost.

Emulation?

- Compile
- Emulate CUDA device with GPU Ocelot
- Use Valgrind to track validity bits

The Pitfalls of Emulation

- Imperfect emulation
- Performance
  - GPUs brought us massive parallelism
  - Emulation drags us back to the CPU
  - Valgrind hampers performance further

Source Translation

- Automatically instrument source code
  - Requires hooking into the build process
  - Obvious to compiler optimization level
- Creates a new compilation artifact to test

Binary Translation as a Solution

- Rewrite existing, compiled CUDA programs on the fly *for the GPU*.
- Retain the parallelism that necessitated GPUs in the first place.

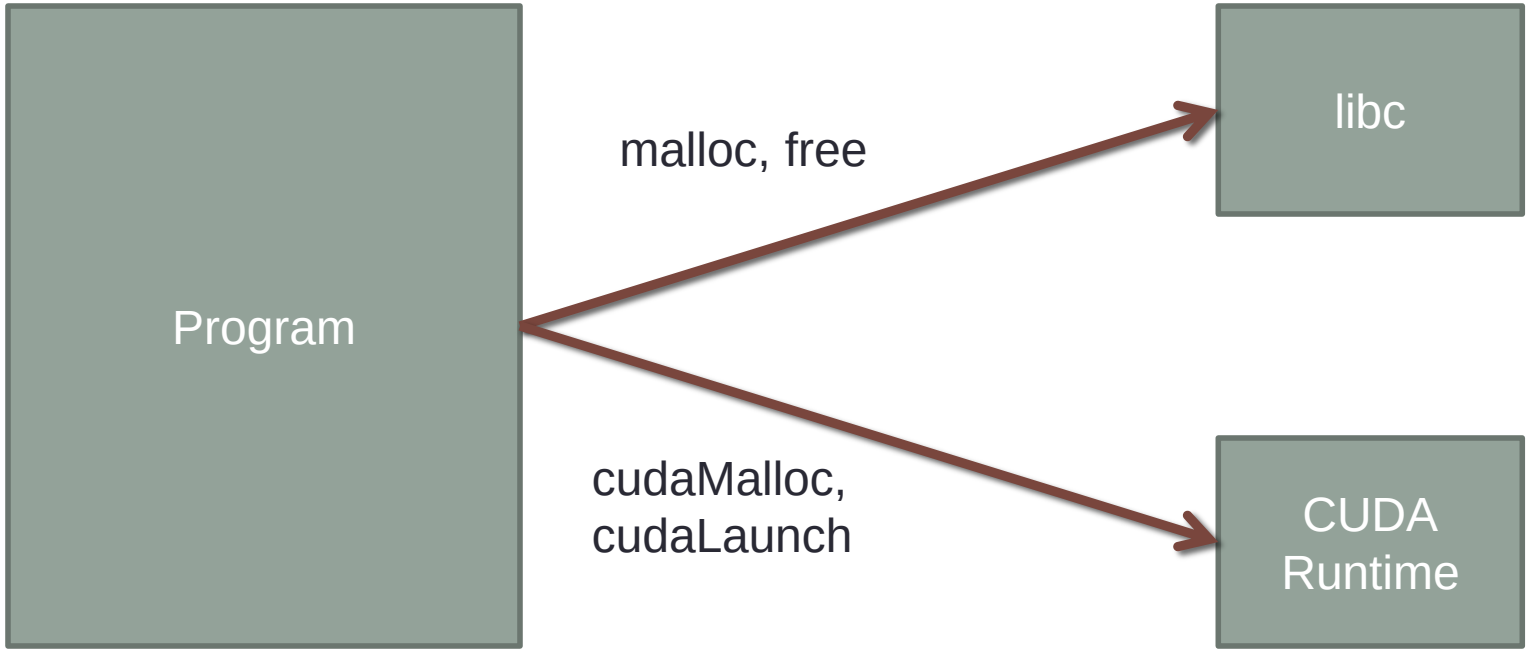
Panoptes

- Provides a framework for capturing calls to the CUDA library and translating device code on the fly
- Demonstrates this capability with a CUDA-centric version of Valgrind’s memcheck

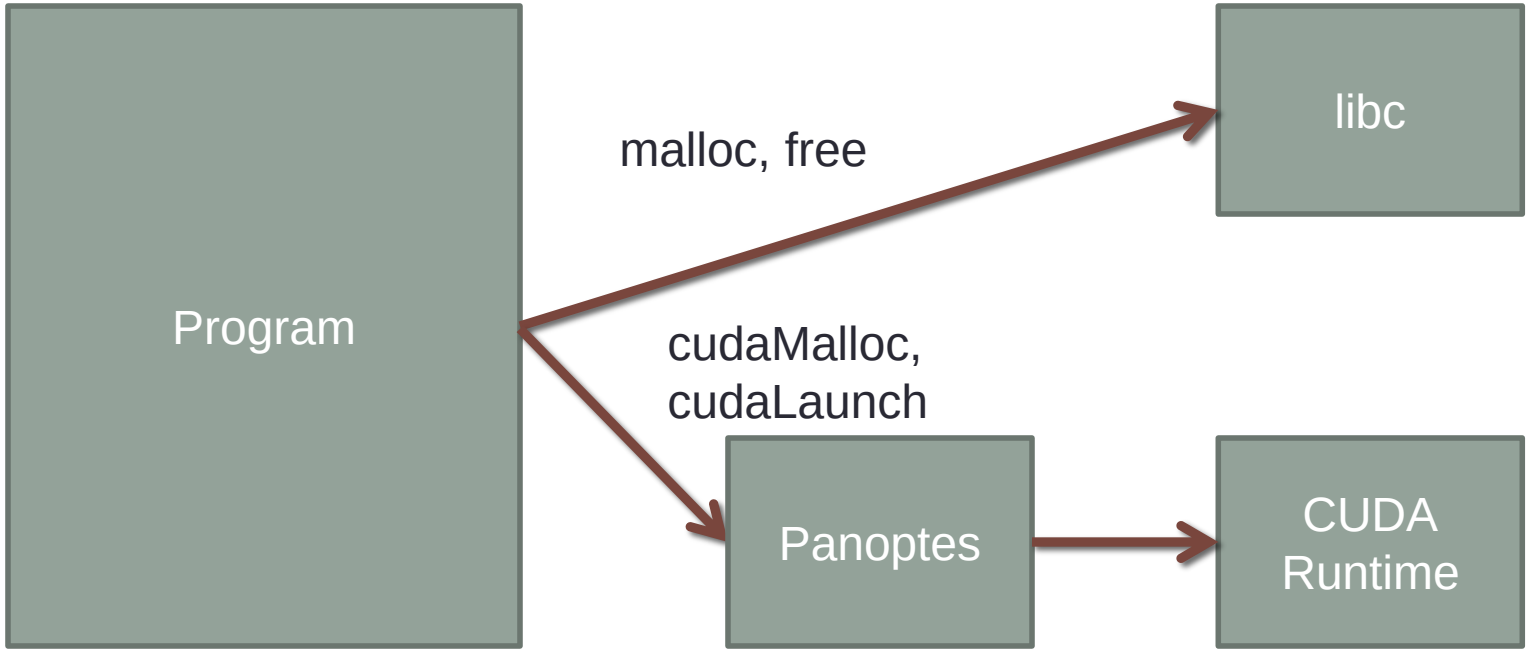
The Translation Process

- Use library interposition to intercept calls to the CUDA Runtime API
- Startup
  - Parse compiled PTX
  - Instrument
  - Pass into CUDA Driver API
- Execution: Track further calls into the runtime

Library Interposition



Library Interposition



```
$ LD_PRELOAD=“./libpanoptes.so” ./my_cuda_program
```

Illustrative Examples

- Logical Errors (API Misuse)
- Buffer overruns
- Uninitialized memory

API Misuse

- Since Panoptes sees every API call, it keeps a copy of the state of various resources
  - It needs to know the size given to cudaMalloc (for addressability purposes)
  - …but we can also warn when cudaBindTexture spills over the allocated region.
- The library may return cudaSuccess, but errors can appear downstream.

Identical Behavior

- Program behavior under Panoptes should be nearly identical to behavior without Panoptes.
- Most of the test suite is built around verifying this property.
- There are a number of places where the CUDA Runtime segfaults; Panoptes deliberately does so as well.

Asynchronous Memory Copy

```
int * device;
cudaMalloc((void **) &device, sizeof(*device));

int host;
cudaMemcpyAsync(&host, device, sizeof(host),
    cudaMemcpyDeviceToHost);
```

==12703== cudaMemcpyAsync must use pinned host memory: 0x7fefffaf4 is not pinned at offset 0.

==12703==

==12703== at 0x4f56fe7: cudaMemcpyAsync+0x29 (../libpanoptes.so)

==12703== by 0x4047f9: ./vtest_memcpyasync

Texture Overrun

```
const textureReference * texref;
cudaChannelFormatDesc desc;

void * p;
cudaMalloc(&p, 1 << 22 /* 4MB */);
cudaBindTexture(NULL, texref, p, &desc,
  1 << 23 /* 8MB */);
```

```
==10988== Texture bound 4194304 bytes beyond allocation.
==10988== Address 0x801a00000 is 0 bytes after a block of
size 0 alloc'd
==10988==
==10988==
==10988== at 0x4f5572e: cudaBindTexture+0x2a
(../libpanoptes.so)
==10988== by 0x4048f5: ./vtest_bindtexture
```

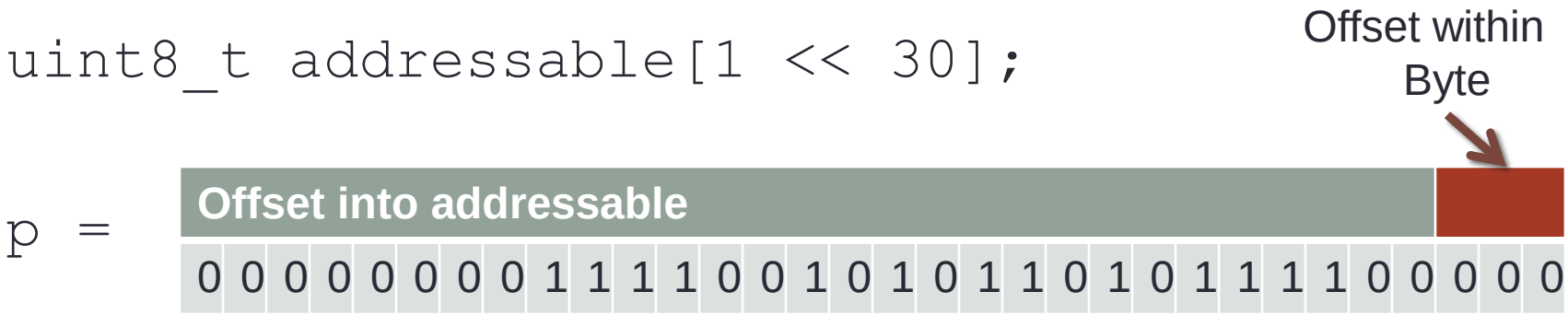
Memory Check Instrumentation

- How do we instrument a memory load?

```
unsigned i = *p;

ld.u32 %r1,[%rd1];
```

```
uint8_t addressable[1 << 30];
```



Shadow Every Byte

```
uint8_t addressable[1 << 32];
uint8_t a = (addressable[p >> 3]
     >> (p & 0x7)) & 0x0F;
unsigned i;
if (a == 0x0F) { i = *p; }
else { /* invalid */ }
```

Shadow Every Byte

```
uint8_t addressable[1 << 32];
uint8_t a = (addressable[p >> 3]
     >> (p & 0x7)) & 0x0F;
unsigned i;
if (a == 0x0F) { i = *p; }
else { /* invalid */ }
```

- `addressable` is quite large!

Shadow Every Byte

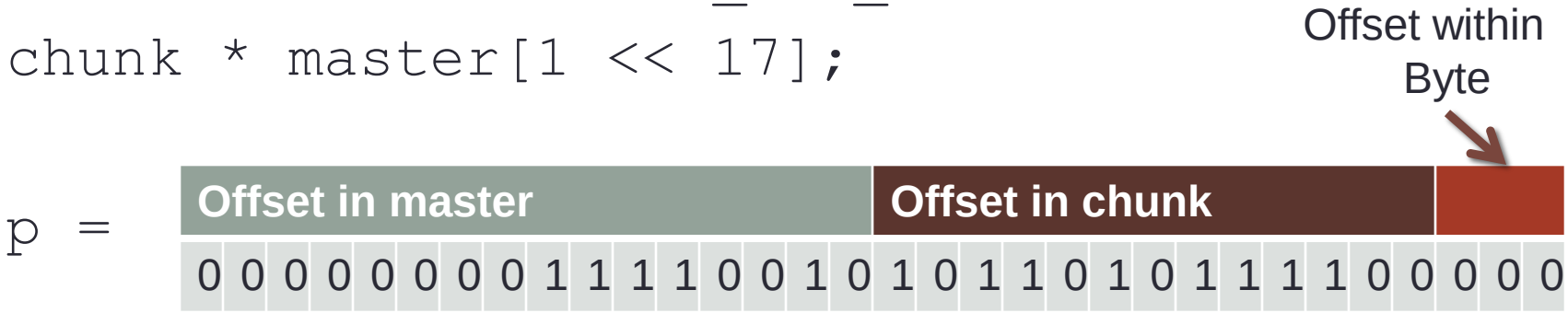
```
uint8_t addressable[1 << 32];
uint8_t a = (addressable[p >> 3]
     >> (p & 0x7)) & 0x0F;
unsigned i;
if (a == 0x0F) { i = *p; }
else { /* invalid */ }
```

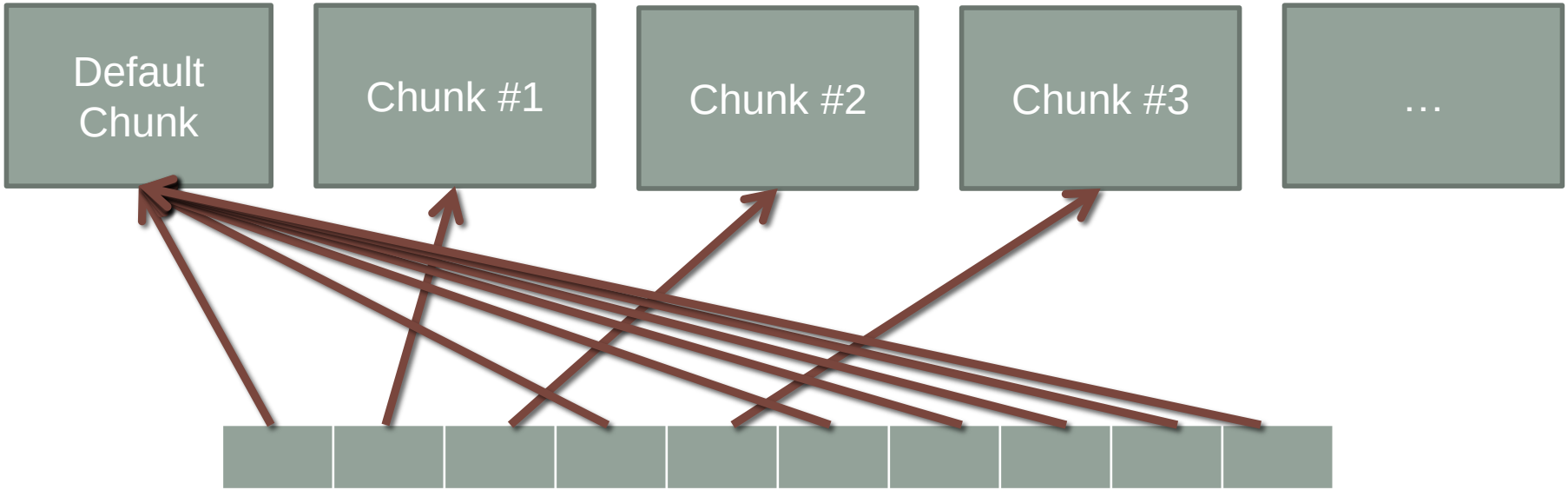
- Introduces a branch on every load.

Improvements

- Chunk memory regions into smaller (64K) blocks
  - Eliminates need to allocate 1GB of contiguous memory on startup
  - Permits reuse of identical chunks: All 64K blocks of RAM with no allocations have the same, all-zero chunk.

```
struct chunk { uint8_t a_data[1 << 13]; };
chunk * master[1 << 17];
```




Default Chunk
Chunk #1
Chunk #2
Chunk #3
…

Branch Elimination

- Branching on the value of `a` is expensive
- Panoptes always performs a memory load
  - For invalid dereferences, we load from a chosen, known-to-be-valid memory location.
  - Selecting a pointer and always loading is faster than conditionally loading.

Validity Tracking Instrumentation

- Validity bits shadow real memory allocations
- Comparatively costly
  - Validity bits should have a 1 to 1 mapping to actual data
  - Compression and packing schemes increase complexity

- Addressability bits are paired with validity bits.
- We have a pointer to the chunk from our address lookup.

```
struct metadata_chunk {
    uint8_t a_data[1 << 13];
    uint8_t v_data[1 << 16];
};
```

Panoptes in Action

```
__global__ void ksum(int * out, const int * in, int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) { sum += in[i]; }
  if (sum == 0) {
     out[1] = sum;
  } else {
     out[0] = sum;
  }
}
```

Generated PTX

```
… (%r5 is the value of sum after the loop)
mov.u32 %r8, 0;
setp.ne.s32 %p3, %r5, %r8;
@%p3 bra $Lt_1_3330;
ld.param.u64 %rd2, [__cudaparm__Z5k_sumPiPKii_out];
st.global.s32 [%rd2+4], %r5;
bra.uni $Lt_1_3074;
$Lt_1_3330:
ld.param.u64 %rd2, [__cudaparm__Z5k_sumPiPKii_out];
st.global.s32 [%rd2+0], %r5;
$Lt_1_3074:
…
```

Host Code

```
int n = 64;
int *out;
cudaMalloc((void **) &out, sizeof(*out) * 2);
int * in;
cudaMalloc((void **) &in, sizeof(*in) * n);
cudaMemset(in, 0x01, sizeof(*in) * (n - 1));
ksum<<<1, 1, 0>>>(out, in, n);
cudaDeviceSynchronize();
int sum;
cudaMemcpy(&sum,out, sizeof(sum), cudaMemcpyDeviceToHost);
```

Wild Branch

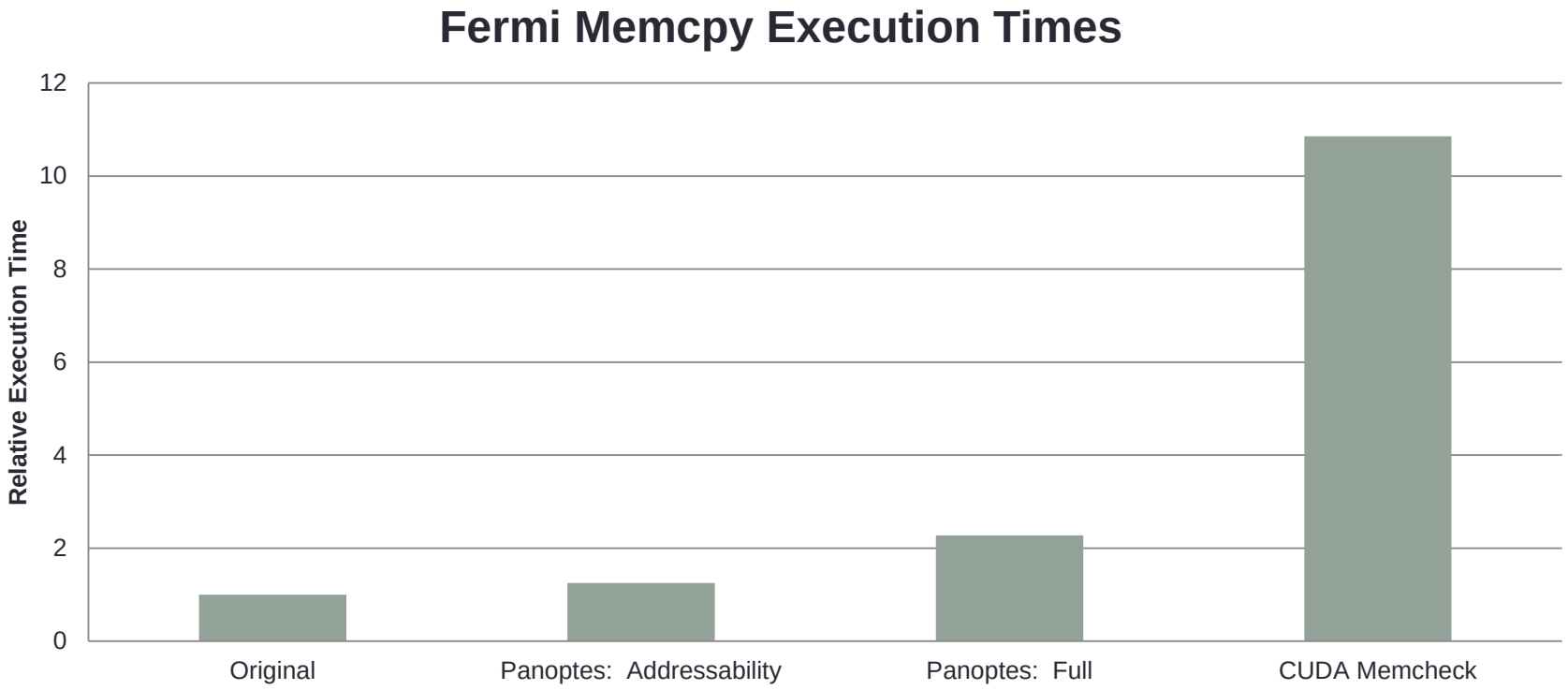
```
==12805== Encountered 1 errors in `k_sum(int*, int const*, int)'.
==12805== Error 0: Wild branch at @%p3 bra $Lt_1_3330;
==12805==
==12805== at 0x4f57b3e: cudaStreamSynchronize+0x19
(../libpanoptes.so)
==12805== by 0x40398d: ./vtest_k_validity
…
==12805==
==12805== Kernel launched by:
==12805==
==12805== at 0x4f577da: cudaLaunch+0x19 (../libpanoptes.so)
==12805== by 0x404ce6: ./vtest_k_validity
…
```

Device to Host Validity Transfer

- If run under Valgrind and Panoptes, we see the invalid bits on the host as well:

```
==12805== Conditional jump or move depends on
uninitialised value(s)
==12805== at 0x403CCE:
ValidityTransfer_Summation_Test::TestBody()
(vtest_k_validity.cu:162)
```

Performance



Future Work

- Optimized Instrumentation
- Reduced memory requirements

Optimized Instrumentation

- Current implementation aims to be unobtrusive, but some operations can be redundant.
- Mapping instructions at a higher level than single PTX operations would enable constant propagation and simplifications.

Reduced Memory Requirements

- Panoptes currently imposes steep memory requirements
  - Addressability checks (12.5% overhead)
  - Validity tracking (100% overhead)
- Reducing memory requirements broadens applications that can be run with Panoptes.

Alternative Instrumentation

- Virtual CUDA Devices
- Data Race Detection

- Source Code: http://www.github.com/ckennelly/panoptes
- Open Sourced under the GPLv3.

- Email: chris@ckennelly.com