Simplifying GPU Access: A Polyglot Binding for GPUs with GraalVM

Lukas Stadler, Senior Research Manager, Oracle Labs
Rene Mueller, AI Developer Technology Engineer, NVIDIA

GTC Silicon Valley 2020

Safe harbor statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

GraalVM Native Image Early Adopter Status

GraalVM Native Image technology (including SubstrateVM) is Early Adopter technology. It is available only under an early adopter license and remains subject to potentially significant further changes, compatibility testing and certification.

GraalVM™

…and many more

Based on Java Technology
Runs programs more efficiently
Ubiquitous across the Cloud Stack
Makes developers more productive

The CUDA Toolkit (C++)

```
__global__
void inc_kernel(int count, const float *input, float *output) {
  int start_index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = start_index; i < count; i += stride)
    outarr[i] = inarr[i] + 1;
}
```

GPU/device code (kernel)

```
int main() {
  // Allocate Unified Memory (accessible from CPU or GPU)
  float *x, *y;
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  for (int i = 0; i < N; ++i) {  // fill x array
     x[i] = ...
  }
  inc_kernel<<<160,256>>>(N, x, y); // invoke with 160x256 threads
  cudaDeviceSynchronize();

  // use y array (omitted)

  cudaFree(x); // Free memory
  cudaFree(y);
}
```

CPU/host code

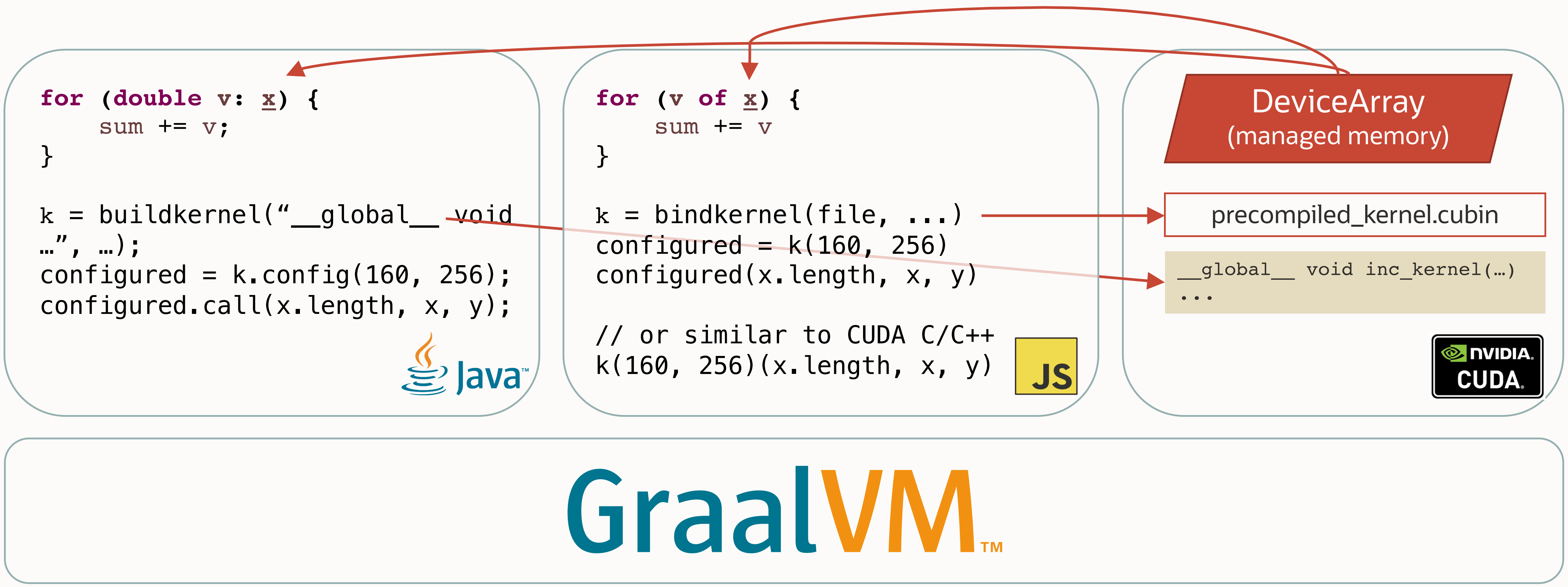
Device code and host code are both written in C/C++

Using CUDA to Access Nvidia GPUs (without GraalVM)

- Different binding libraries / APIs for CUDA in different programming languages
- Varying set of supported features
- Translation to/from unmanaged environment (in Java, C#, Python, etc.)

Python	Numba, cuPy, PyCUDA
Java	JCuda, jCUDA, CUDA4J
C / C++	CUDA C/C++ (language extension)
R	gpuR, indirectly through Rcpp
JS	gpu.js (WebGL), node-cuda, cuda-ts
C#	Hybridizer, ManagedCUDA, Alea GPU, ILGPU
Ruby	RbCUDA

Using grCUDA to Access Nvidia GPUs with GraalVM



```
for (double v: x) {
     sum += v;
}

k = buildkernel("__global__ void
…", …);
configured = k.config(160, 256);
configured.call(x.length, x, y);
```

```
for (v of x) {
     sum += v
}

k = bindkernel(file, ...)
configured = k(160, 256)
configured(x.length, x, y)

// or similar to CUDA C/C++
k(160, 256)(x.length, x, y)
```

```
__global__ void inc_kernel(…)
...
```

Creating and Using Device Arrays (Java)

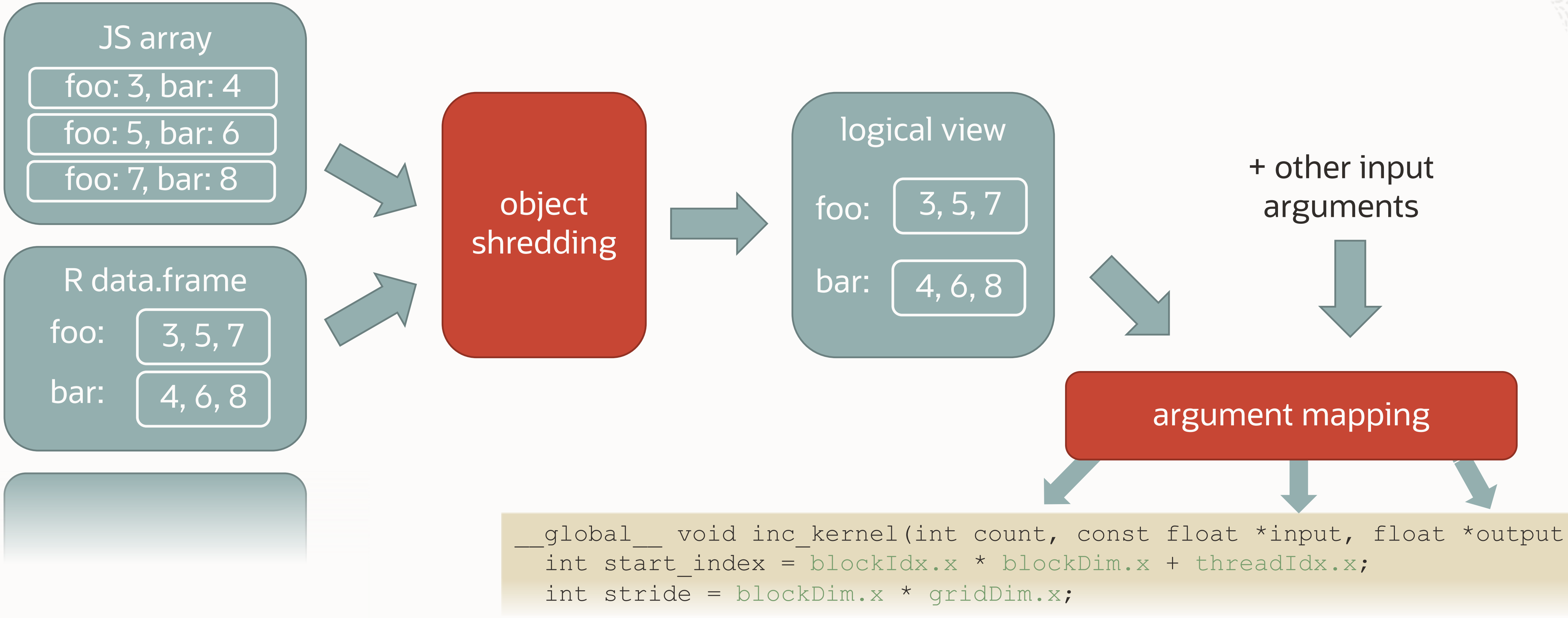
```java
// Get access to CUDA functionality from polyglot environment
CUDA cu = (CUDA) Polyglot.eval("grcuda", "CU")

// Create 1D device array that can hold 1000 int values
int[] arrayInt1D = (int[]) cu.DeviceArray("int", 1000)
// Create 2D device array that can hold 1000 x 100 float values
float[][] arrayFloat2D = (float[][]) cu.DeviceArray("float", 1000, 100)

// Setting array elements
for (int i = 0; i < arrayInt1D.length; i++) {
    arrayInt1D[i] = i;
}
for (int x = 0; x < arrayFloat2D.length; x++) {
    for (int y = 0; y < arrayFloat2D[x].length; y++) {
        arrayFloat2D[x][y] = x + y;
    }
}
```

Launching GPU Kernels (R)

```
# Get access to CUDA functionality from polyglot environment
CU <- eval.polyglot("grcuda", "CU")

count <- 1000000000
input <- CU$DeviceArray("float", count)
output <- CU$DeviceArray("float", count)

code <- "__global__ void inc_kernel(int count, const float *input, float *output) ..."

# Create/compile kernel from source and link with given signature
incKernel <- CU$buildkernel(code, "inc_kernel", "sint32, pointer, pointer")

# Launch kernel in grid consisting of 160
# blocks with 256 threads each
incKernel(160, 256)(count, input, output)
```

Device arrays `input` and `output` can be passed to GPU kernel

Shredding of Objects: Structs-of-Arrays vs. Arrays-of-Structs



```
__global__ void inc_kernel(int count, const float *input, float *output
  int start_index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
```

Shredding of Objects: Structs-of-Arrays vs. Arrays-of-Structs (JS)

```
// create mapping that ”shreds” its inputs – extracts property “foo” as array
toFloatDeviceArray = CU.DeviceArray.map("float")
shreddedArg = CU.Map.arg("x").shred().foo.map(toFloatDeviceArray)
mapped = CU.Map(incKernel(160, 256),
                CU.Map.size(shreddedAr),  // arg 1: length of shredded input
                shreddedArg,              // arg 2: shredded input as DeviceArray
                CU.Map.arg("y")           // arg 3: output DeviceArray
             )
len = 10000000
output = CU.DeviceArray("float", len)

// create and initialize array-of-structs and struct-of-arrays
arrayOfStructs = Array.from(new Array(len), (x, i) => ({foo: i})) // [{foo:0},{foo:1},…
structOfArrays = {foo: Array.from(new Array(len).keys())}         // {foo:[0,1,2,3,4,…

mapped(arrayOfStructs, output)
mapped(structOfArrays, output)
```

Transparently converts different object layouts

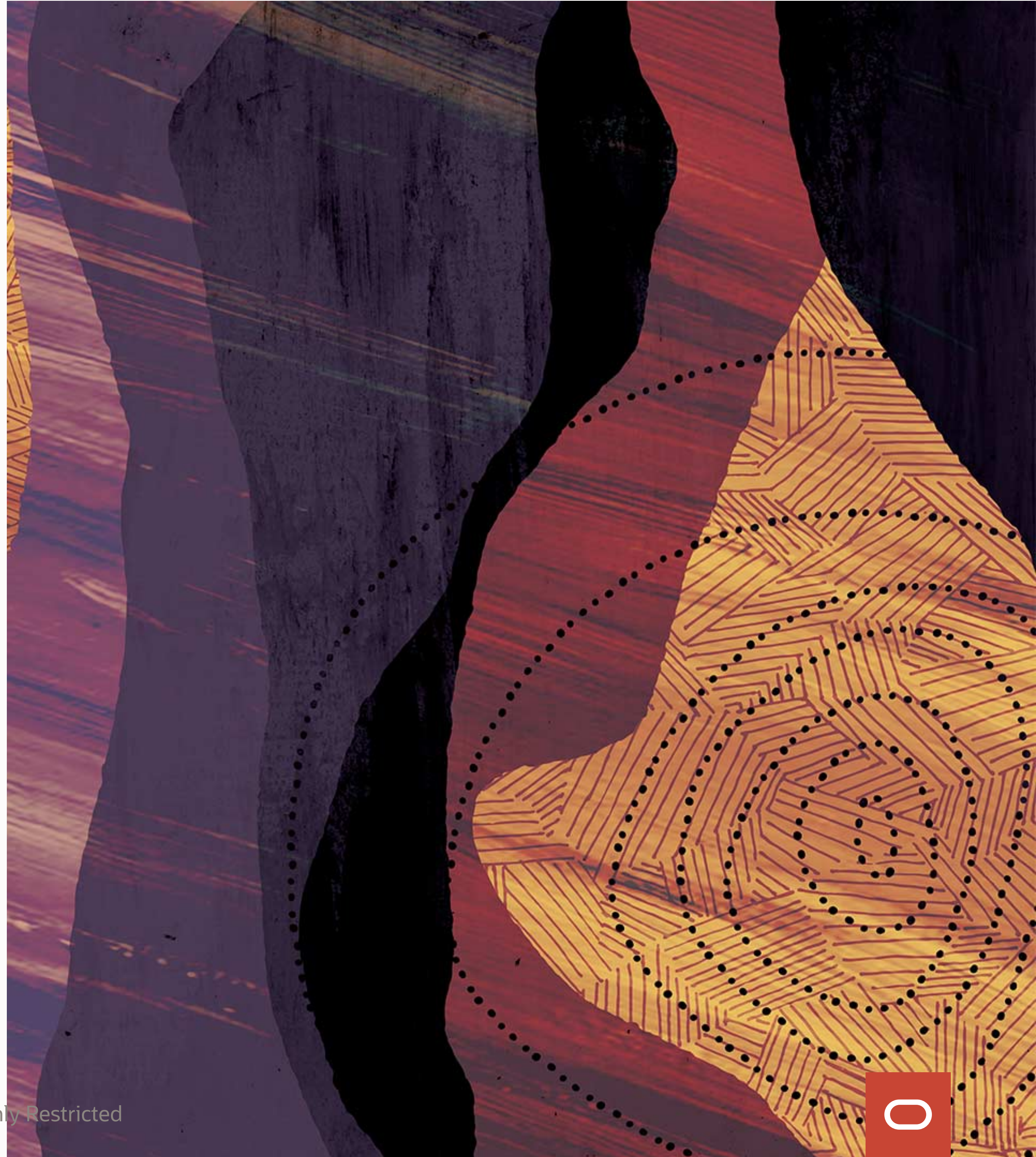
Demo

grCUDA

- … and a lot more, e.g.:
  - Use existing GPU-accelerated libraries with DeviceArray objects
  - Load pre-compiled kernels from .ptx and .cubin files
  - Query GPU device properties, e.g., grid and memory sizes
- More features planned, e.g.:
  - Device Memory managed by grCUDA using explicit transfers.
  - Asynchronous execution of copy and kernel launches.

Get it from https://github.com/NVIDIA/grcuda
More info on Nvidia's Blog: https://devblogs.nvidia.com/

Thank you



ORACLE