



NVIDIA®

CUDA 6 OVERVIEW  
Ujval Kapasi

# Supported Platforms

- Windows
  - Windows XP, Vista, 7, 8, 8.1, Server 2008 R2, Server 2012 R2
  - Visual Studio 2008, 2010, 2012, 2012 Express
- Linux
  - Fedora 19
  - RHEL & CentOS 5, 6
  - OpenSUSE 12.3
  - SUSE SLES 11 SP2, SP3
  - Ubuntu 12.04 LTS, 13.04 (including ARM native and cross-compilation tools)
  - ICC 13.1
- Mac
  - OSX 10.8, 10.9

# Deprecations

- Deprecated means:
  - Still supported
  - Not recommended
  - New features may not work with it—check documentation
  - Likely to be completely dropped in the future
- 32-bit applications on x86 Linux (toolkit & driver)
- 32-bit applications on Mac (toolkit & driver)
- sm\_1.0 / G80 architecture (toolkit)

# Dropped Support

- cuSPARSE “Legacy” API (use `cusparse_v2.h` API instead)
- Ubuntu 10.04 LTS (use 12.04 LTS)
- SLES 11 SP1 (use SP2 or SP3)
- Mac OSX 10.7 (use 10.8 or 10.9)
- Mac Models with the MCP79 Chipset (driver)

|                                   |   |
|-----------------------------------|---|
| iMac (20-inch, Early 2009)        | MacBook Pro (15-inch, Mid 2009)         |
| iMac (24-inch, Early 2009)        | MacBook Pro (15-inch 2.53GHz, Mid 2009) |
| iMac (21.5-inch, Late 2009)       | MacBook Pro (13-inch, Mid 2009)         |
| MacBook Pro (15-inch, Late 2008)  | Mac mini (Early 2009)                   |
| MacBook Pro (17-inch, Early 2009) | Mac mini (Late 2009)                    |
| MacBook Pro (17-inch, Mid 2009)   | MacBook Air (Late 2008, Mid 2009)       |

# Maxwell GM107, GM108

## 1. More Efficient Multiprocessors

135% performance/core vs. Kepler

2x performance/watt vs. Kepler

## 2. Larger, Dedicated Shared Memory

## 3. Fast Shared Memory Atomics

## 4. Support for Dynamic Parallelism





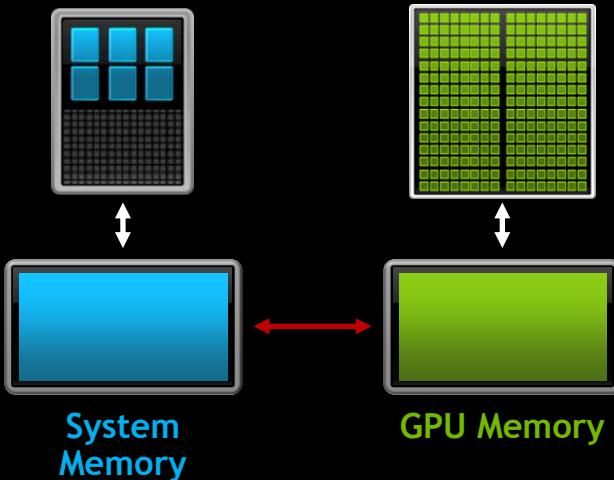
# CUDA 6 Unified Memory



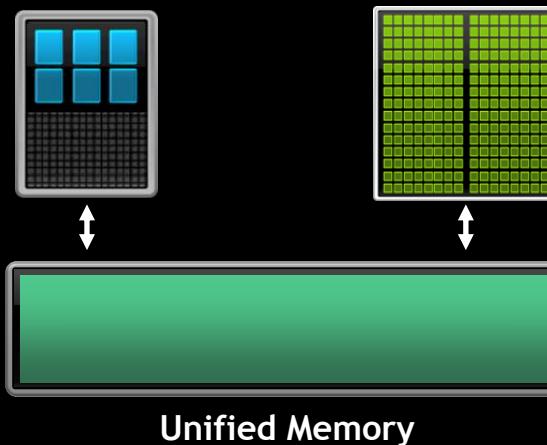
# Unified Memory

## Dramatically Lower Developer Effort

Developer View Today



Developer View With Unified Memory



# Super Simplified Memory Management Code



## CPU Code

```
void sortfile(FILE *fp, int N) {  
    char *data;  
    data = (char *)malloc(N);  
  
    fread(data, 1, N, fp);  
  
    qsort(data, N, 1, compare);  
  
    use_data(data);  
  
    free(data);  
}
```

## CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {  
    char *data;  
    cudaMallocManaged(&data, N);  
  
    fread(data, 1, N, fp);  
  
    qsort<<<...>>>(data,N,1,compare);  
    cudaDeviceSynchronize();  
  
    use_data(data);  
  
    cudaFree(data);  
}
```

# What Is Unified Memory?

## 1. Simpler Programming & Memory Model

- Single pointer to data, accessible anywhere
- Eliminate need for *cudaMemcpy()*
- Greatly simplifies code porting

## 2. Performance Through Data Locality

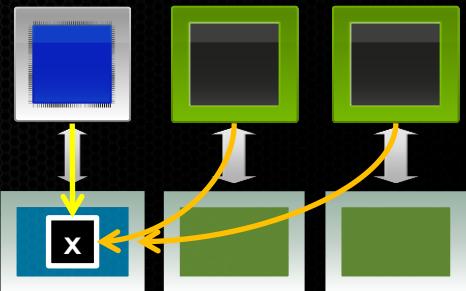
- Migrate data to accessing processor
- Guarantee global coherency
- Still allows *cudaMemcpyAsync()* hand tuning

# System Requirements

|                  |                  |                                       |
|------------------|------------------|---------------------------------------|
| GPU              | Kepler & Maxwell | (GK10x+ or GM10x+, i.e. SM3.0+)       |
| Operating System | 64-bit required  |                                       |
| Linux            | Kernel 2.6.18+   | (all CUDA-supported distros, not ARM) |
| Windows          | Win7 or Win8     | (WDDM & TCC no XP/Vista)              |
| Mac OSX          |                  | Not supported in CUDA 6               |
| Linux on ARM     |                  | Supported with Tegra K1               |

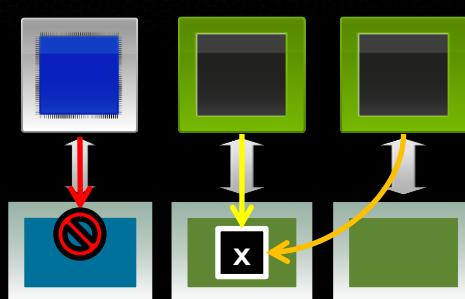
# CUDA Memory Types

Zero-Copy  
`cudaMallocHost(&x, 4)`



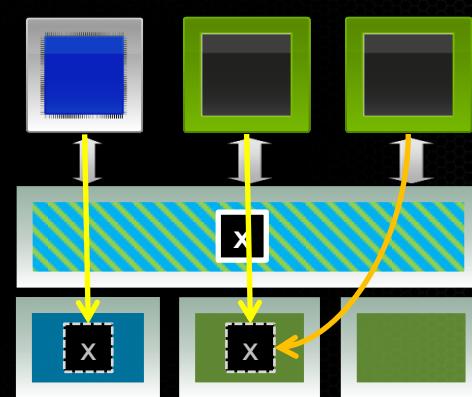
- Allocation fixed in CPU mem
- PCIe access for all GPUs
- Local access for CPU

Unified Virtual Addressing  
`cudaMalloc(&x, 4)`



- Allocation fixed in GPU mem
- Local access for home GPU
- No CPU access
- PCIe access for other GPUs

Unified Memory (CUDA 6)  
`cudaMallocManaged(&x, 4)`



- On-access CPU/GPU migration
- Local access for home GPU
- Local access for CPU
- PCIe access for other GPUs

# Just Three Additions To CUDA

## New API: *cudaMallocManaged()*

- Drop-in replacement for `cudaMalloc()` allocates managed memory
- Returns pointer accessible from both Host and Device

## New API: *cudaStreamAttachMemAsync()*

- Manages concurrency in multi-threaded CPU applications

## New keyword: *\_\_managed\_\_*

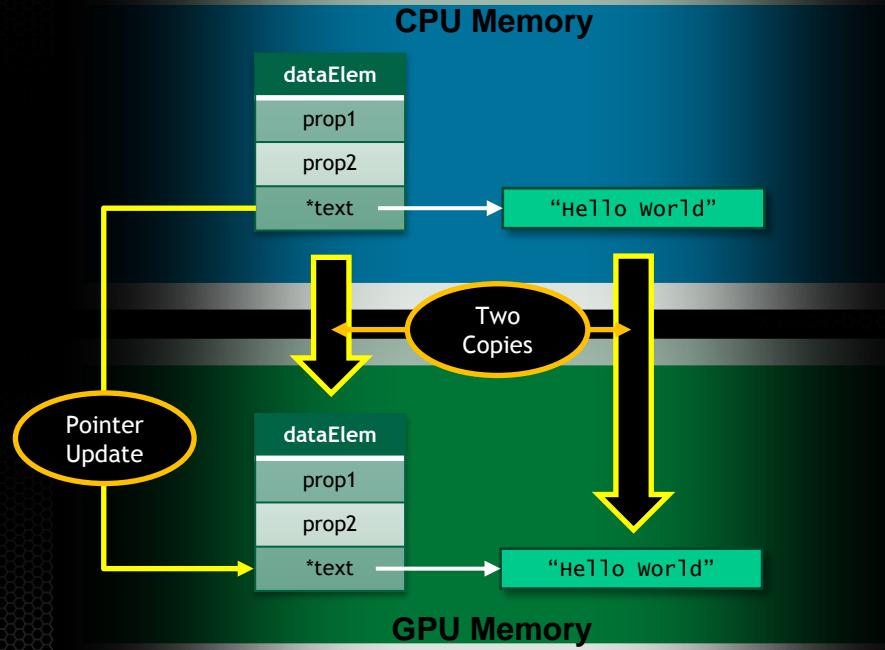
- Global variable annotation combines with `__device__`
- Declares global-scope migratable device variable
- Symbol accessible from both GPU and CPU code

# Using Managed Memory

# Simpler Memory Model

- Eliminate Deep Copying

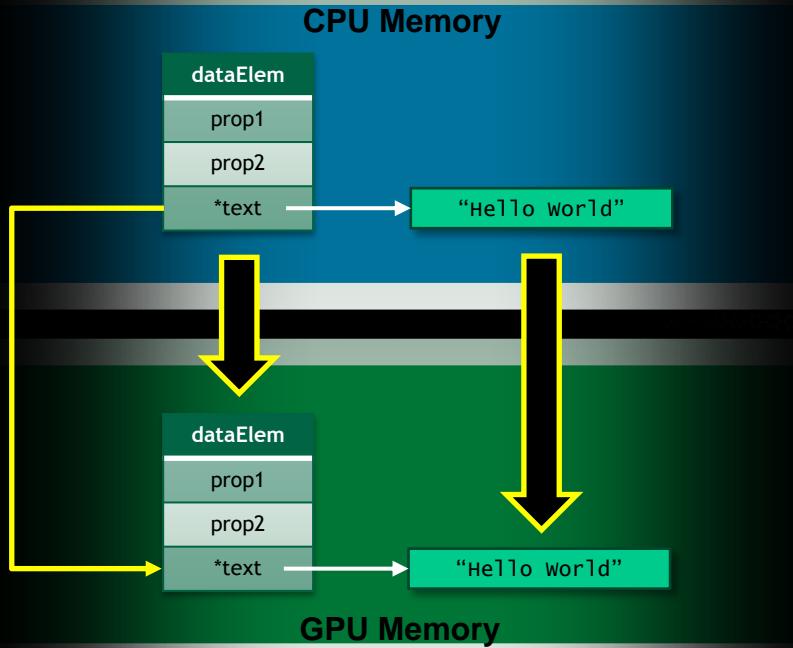
```
struct dataElem {  
    int prop1;  
    int prop2;  
    char *text;  
};
```



# Simpler Memory Model

## Eliminate Deep Copying

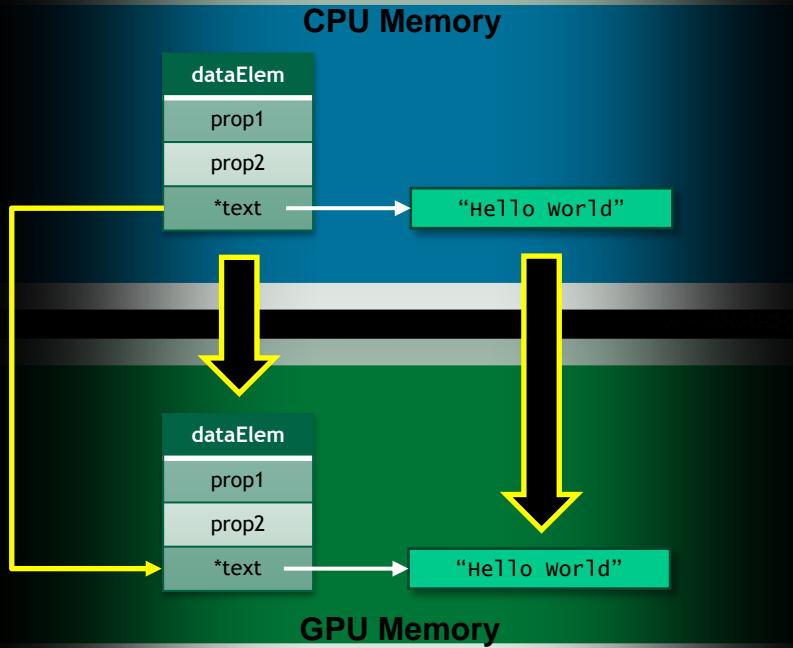
```
void launch(dataElem *elem) {  
    dataElem *g_elem;  
    char *g_text;  
  
    int textlen = strlen(elem->text);  
  
    // Allocate storage for struct and text  
    cudaMalloc(&g_elem, sizeof(dataElem));  
    cudaMalloc(&g_text, textlen);  
  
    // Copy up each piece separately, including  
    // new "text" pointer value  
    cudaMemcpy(g_elem, elem, sizeof(dataElem));  
    cudaMemcpy(g_text, elem->text, textlen);  
    cudaMemcpy(&(g_elem->text), &g_text,  
              sizeof(g_text));  
  
    // Finally we can launch our kernel, but  
    // CPU & GPU use different copies of "elem"  
    kernel<<< ... >>>(g_elem);  
}
```



# Simpler Memory Model

## Eliminate Deep Copying

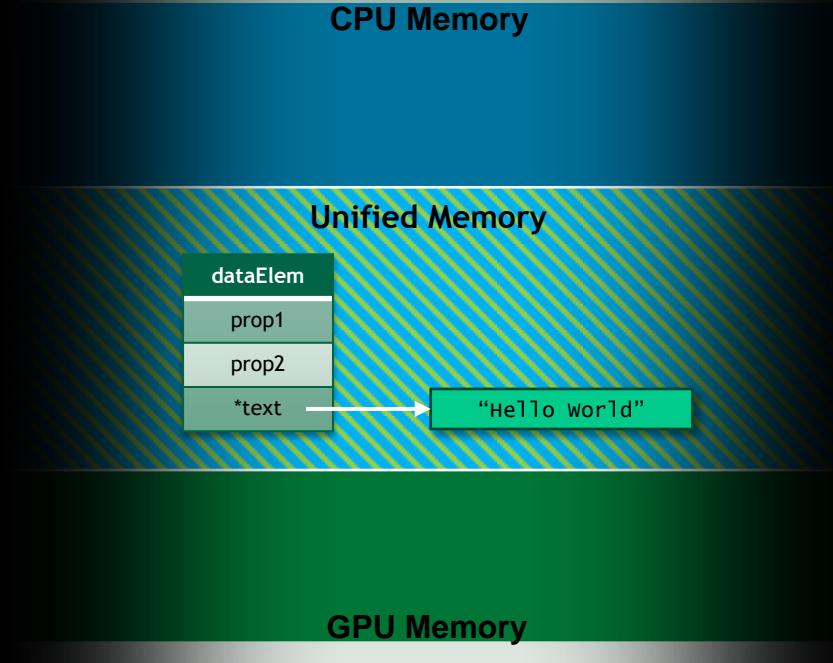
```
void launch(dataElem *elem) {  
    dataElem *g_elem;  
    char *g_text;  
  
    int textlen = strlen(elem->text);  
  
    // Allocate storage for struct and text  
    cudaMalloc(&g_elem, sizeof(dataElem));  
    cudaMalloc(&g_text, textlen);  
  
    // Copy up each piece separately, including  
    // new "text" pointer value  
    cudaMemcpy(g_elem, elem, sizeof(dataElem));  
    cudaMemcpy(g_text, elem->text, textlen);  
    cudaMemcpy(&(g_elem->text), &g_text,  
              sizeof(g_text));  
  
    // Finally we can launch our kernel, but  
    // CPU & GPU use different copies of "elem"  
    kernel<<< ... >>>(g_elem);  
}
```



# Simpler Memory Model

- Eliminate Deep Copying

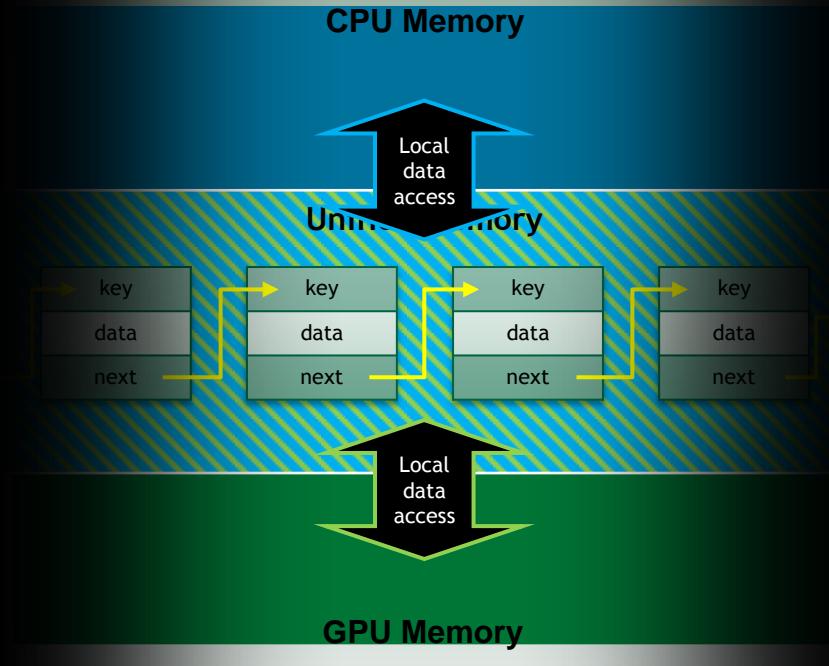
```
void launch(dataElem *elem) {  
    kernel<<< ... >>>(elem);  
}
```



# Simpler Memory Model

## Example: GPU & CPU Shared Linked Lists

- Can pass list elements between Host & Device
- Can insert and delete elements from Host or Device\*
- Single list - no complex synchronization

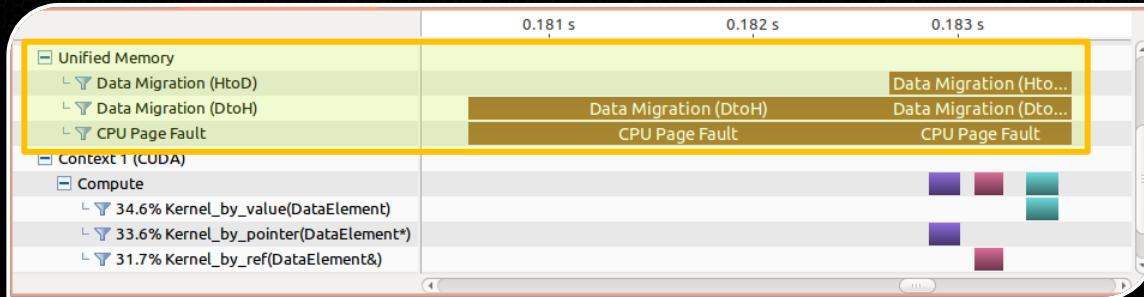


\*Program must still ensure no race conditions.  
Data is coherent between CPU & GPU  
at kernel launch & sync only

# Developer Tools Support Unified Memory



- Visual Profiler, nvprof:



- cuda-memcheck:

```
cuda-memcheck ./dataElem_um
CUDA MEMCHECK
=====
Error: process didn't terminate successfully
=====
The application may have hit an error when dereferencing Unified M
emory from the host. Please rerun the application under cuda-gdb or Nsight Eclipse
Edition to catch host side errors.
=====
internal error (20)
=====
No CUDA-MEMCHECK results found
```

- cuda-gdb:

```
(cuda-gdb) run
Starting program: /home/harrism/src/parallel-forall/code-samples/posts/unified-memo
ry/dataElem_um
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7fffff5f7b700 (LWP 7957)]

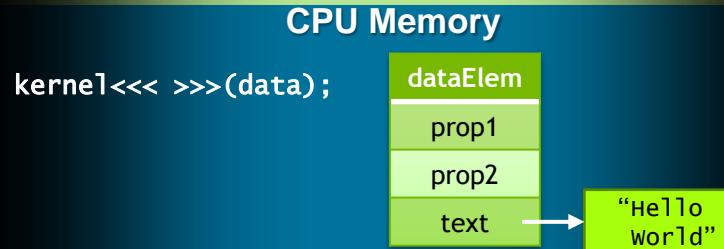
Program received signal CUDA EXCEPTION_15, Invalid Managed Memory Access.
0x000000000004024eb in main ()
```

# Unified Memory with C++

## Host/Device C++ integration has been difficult in CUDA

- Cannot construct GPU class from CPU
- References fail because of no deep copies

```
// Ideal C++ version of
class
class dataElem {
    int prop1;
    int prop2;
    String text;
};
```



```
void kernel(dataElem data) {  
}
```

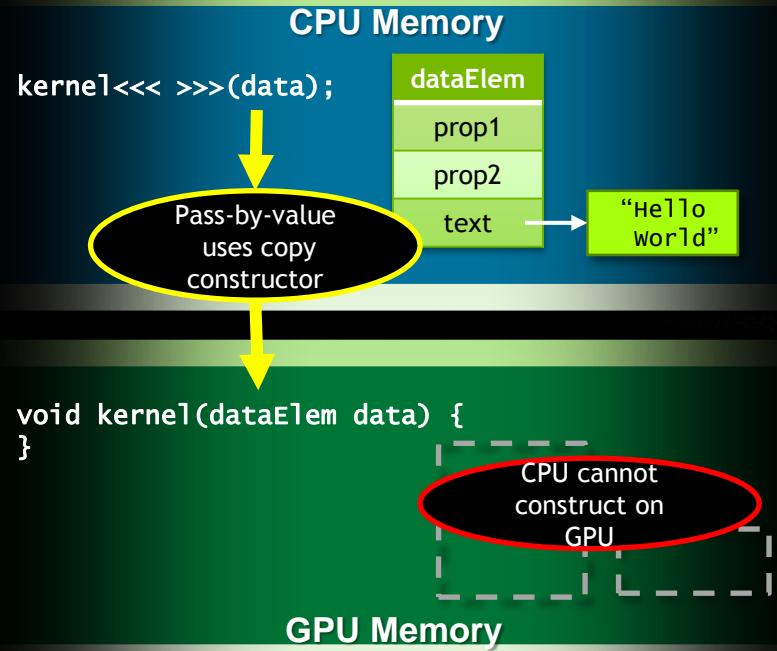
GPU Memory

# Unified Memory with C++

## Host/Device C++ integration has been difficult in CUDA

- Cannot construct GPU class from CPU
- References fail because of no deep copies

```
// Ideal C++ version of  
class dataElem {  
    int prop1;  
    int prop2;  
    String text;  
};
```





# Unified Memory with C++

C++ objects migrate easily when allocated on managed heap

- Overload *new* operator

```
class Managed {  
    void *operator new(size_t len) {  
        void *ptr;  
        cudaMallocManaged(&ptr, len);  
        return ptr;  
    }  
  
    void operator delete(void *ptr) {  
        cudaFree(ptr);  
    }  
};
```

\* (or use placement-new)



# Unified Memory with C++

## Pass-by-reference enabled with *new* overload

```
// Deriving from “Managed” allows pass-by-reference
class String : public Managed {
    int length;
    char *data;

    // Copy constructor using new allocates CPU-only data
    String (const String &s) {
        length = s.length;
        data = new char[length+1];
        strcpy(data, s.data);
    }
};
```

NOTE: CPU/GPU class sharing is restricted to POD-classes only (i.e. no virtual functions)



# Unified Memory with C++

## Pass-by-value enabled by managed memory copy

```
// Deriving from "Managed" allows pass-by-reference
class String : public Managed {
    int length;
    char *data;

    // Unified memory copy constructor allows pass-by-value
    String (const String &s) {
        length = s.length;
        cudaMallocManaged(&data, length+1);
        strcpy(data, s.data);
    }
};
```

NOTE: CPU/GPU class sharing is restricted to POD-classes only (i.e. no virtual functions)

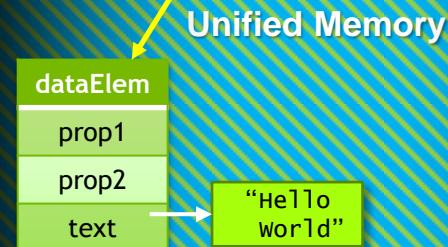
# Unified Memory with C++

## Combination of C++ and Unified Memory is very powerful

- Concise and explicit: let C++ handle deep
- Pass by-value or by-reference without `memcpy shenanigans`

```
// Note "Managed" on this class, too.  
// C++ now handles our deep copies  
class dataElem : public Managed {  
    int prop1;  
    int prop2;  
    String text;  
};
```

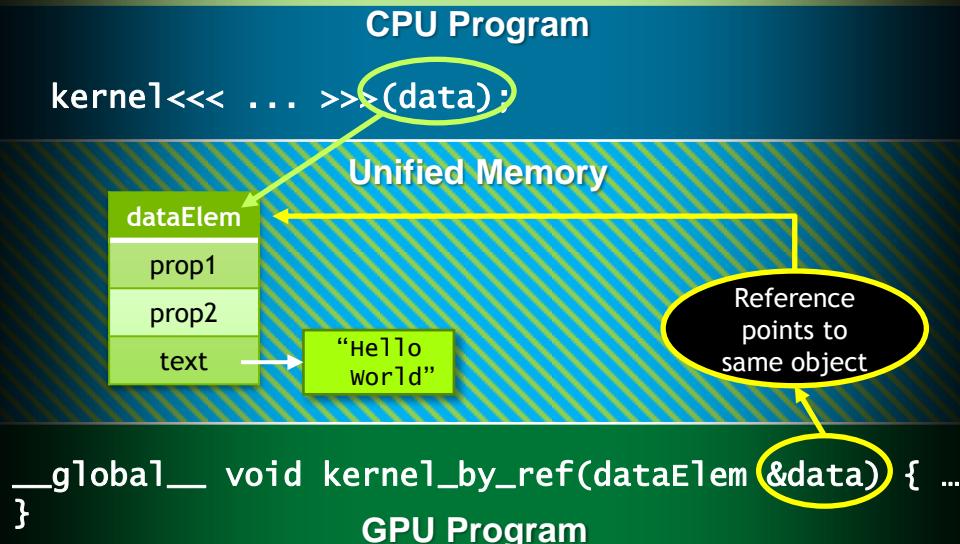
CPU Program  
`dataElem *data = new dataElem;`



GPU Program

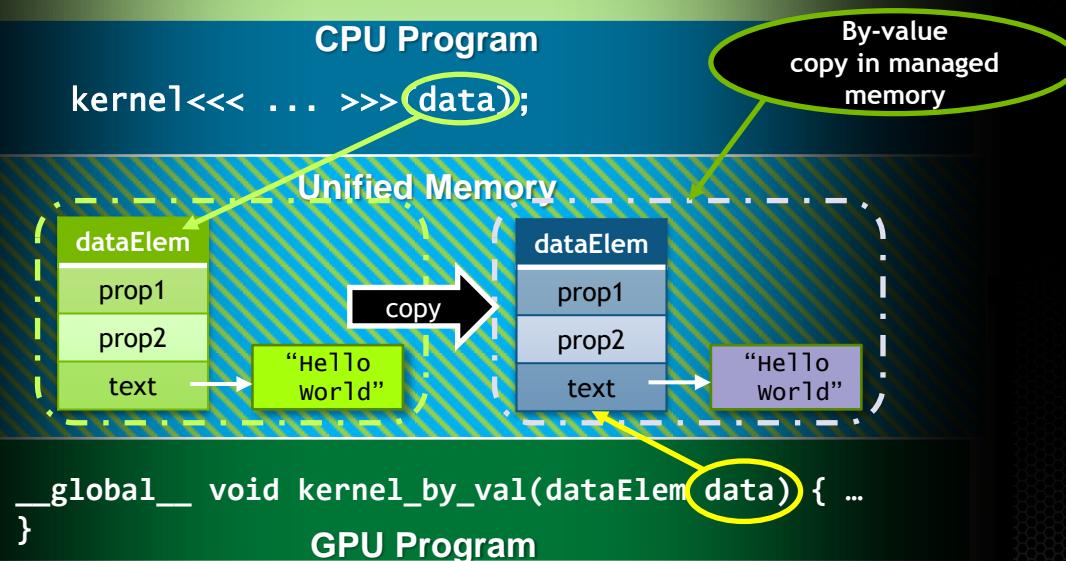
# C++ Pass By Reference

Single pointer to data makes object references just work



# C++ Pass By Value

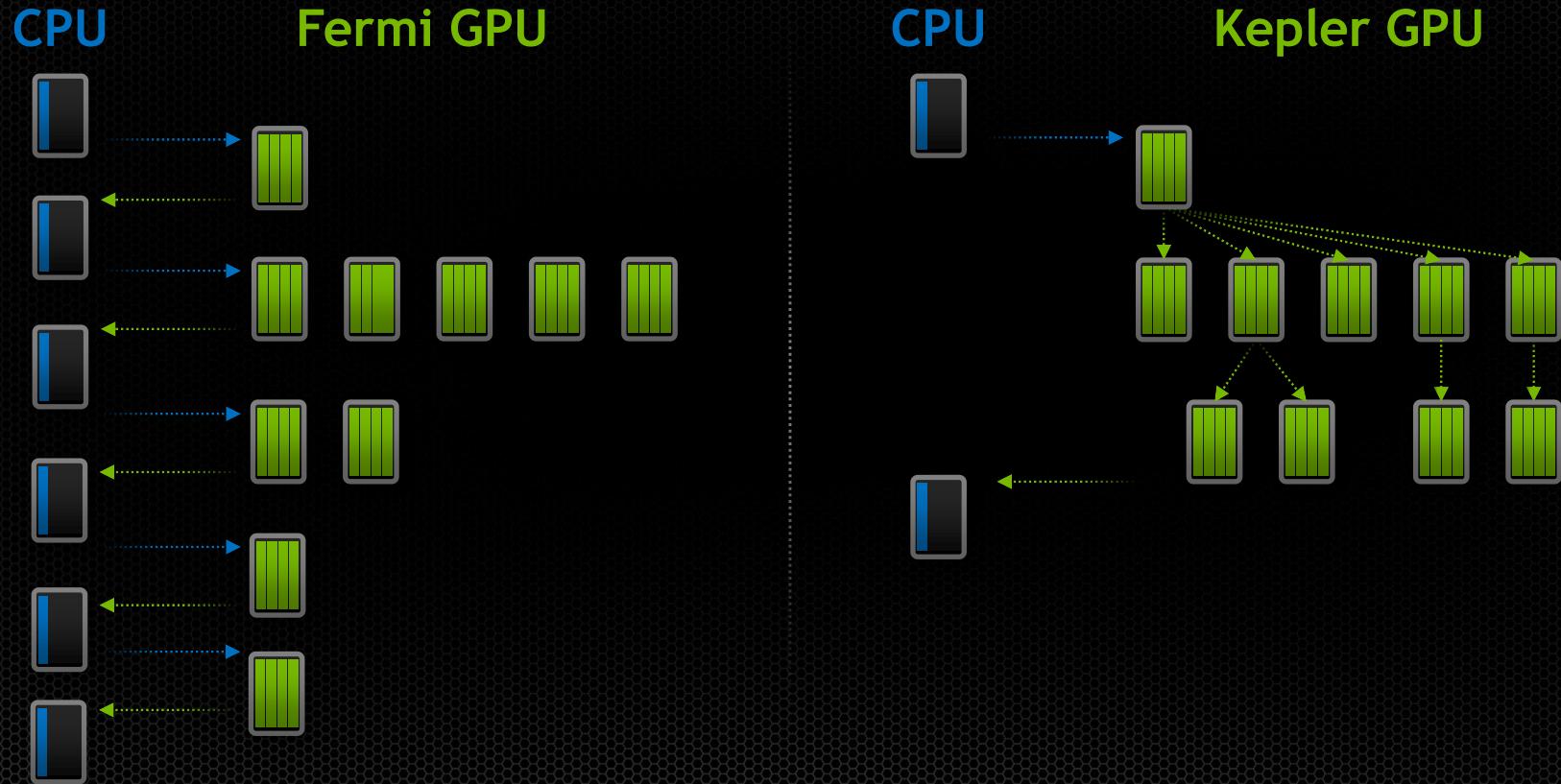
Copy constructors from CPU create GPU-usuable objects



# Dynamic Parallelism Improvements



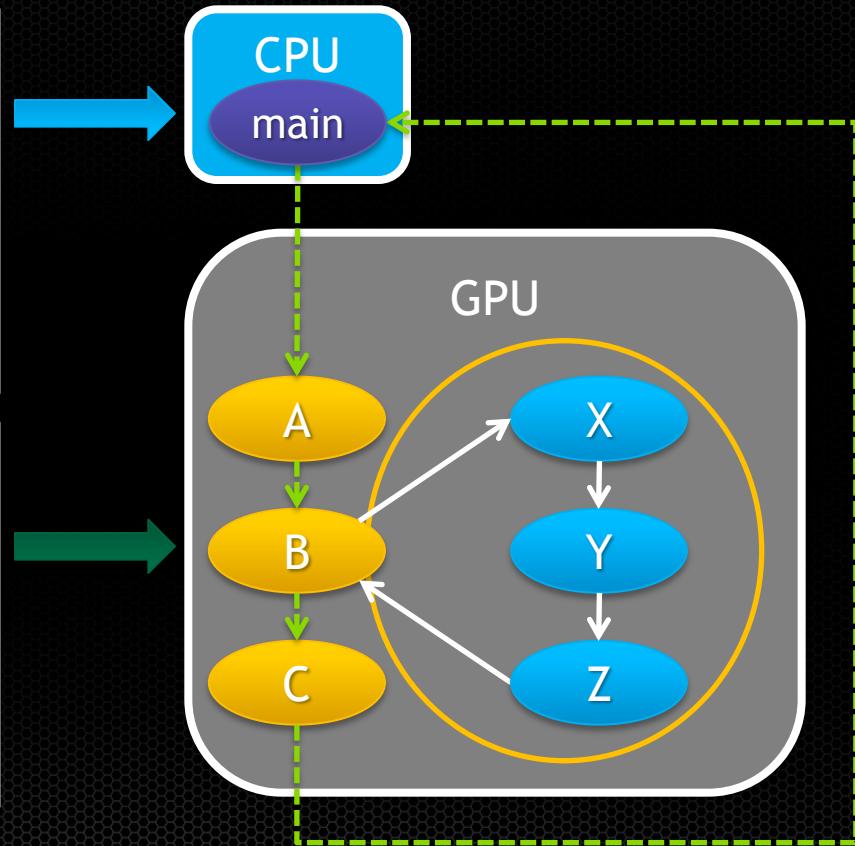
# Dynamic Parallelism - Recap



# Familiar Syntax and Programming Model

```
int main() {  
    float *data;  
    setup(data);  
  
    A <<< ... >>> (data);  
    B <<< ... >>> (data);  
    C <<< ... >>> (data);  
  
    cudaDeviceSynchronize();  
    return 0;  
}
```

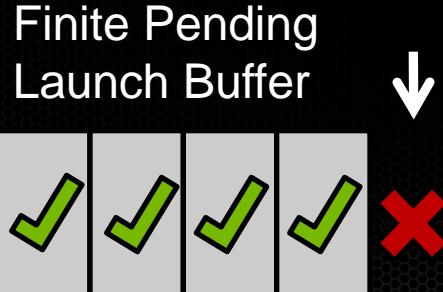
```
__global__ void B(float *data)  
{  
    do_stuff(data);  
  
    X <<< ... >>> (data);  
    Y <<< ... >>> (data);  
    Z <<< ... >>> (data);  
    cudaDeviceSynchronize();  
  
    do_more_stuff(data);  
}
```



# Dynamic Parallelism

## Pending Launch Buffer Limit

- Dynamic Parallelism applications can launch too many grids and exhaust the pre-allocated pending launch buffer (PLB).
  - Result in launch failures, sometimes intermittent due to scheduling
  - PLB size tuning can fix the problem, but often involves trial-and-error

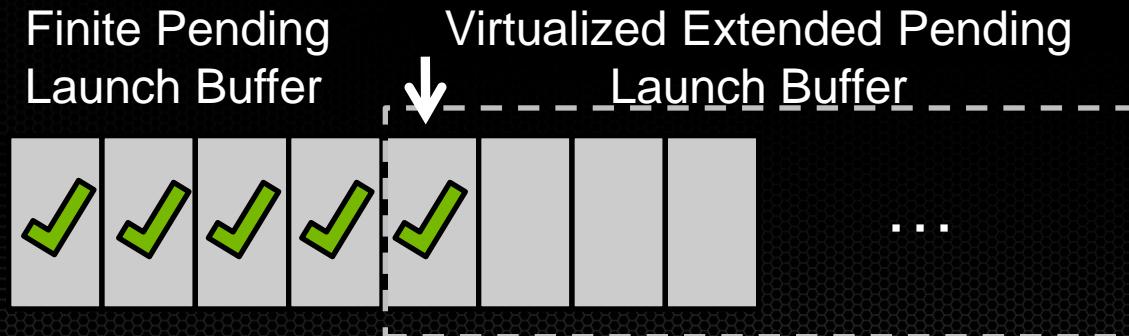


Out-of-memory failure with too  
many concurrent launches.

# Dynamic Parallelism

## Extended Pending Launch Buffer (EPLB)

- EPLB guarantees all launches succeed by using a lower performance virtualized launch buffer, when fast PLB is full.
  - No more launch failures regardless of scheduling
  - PLB size tuning provides direct performance improvement path
  - Enabled by default

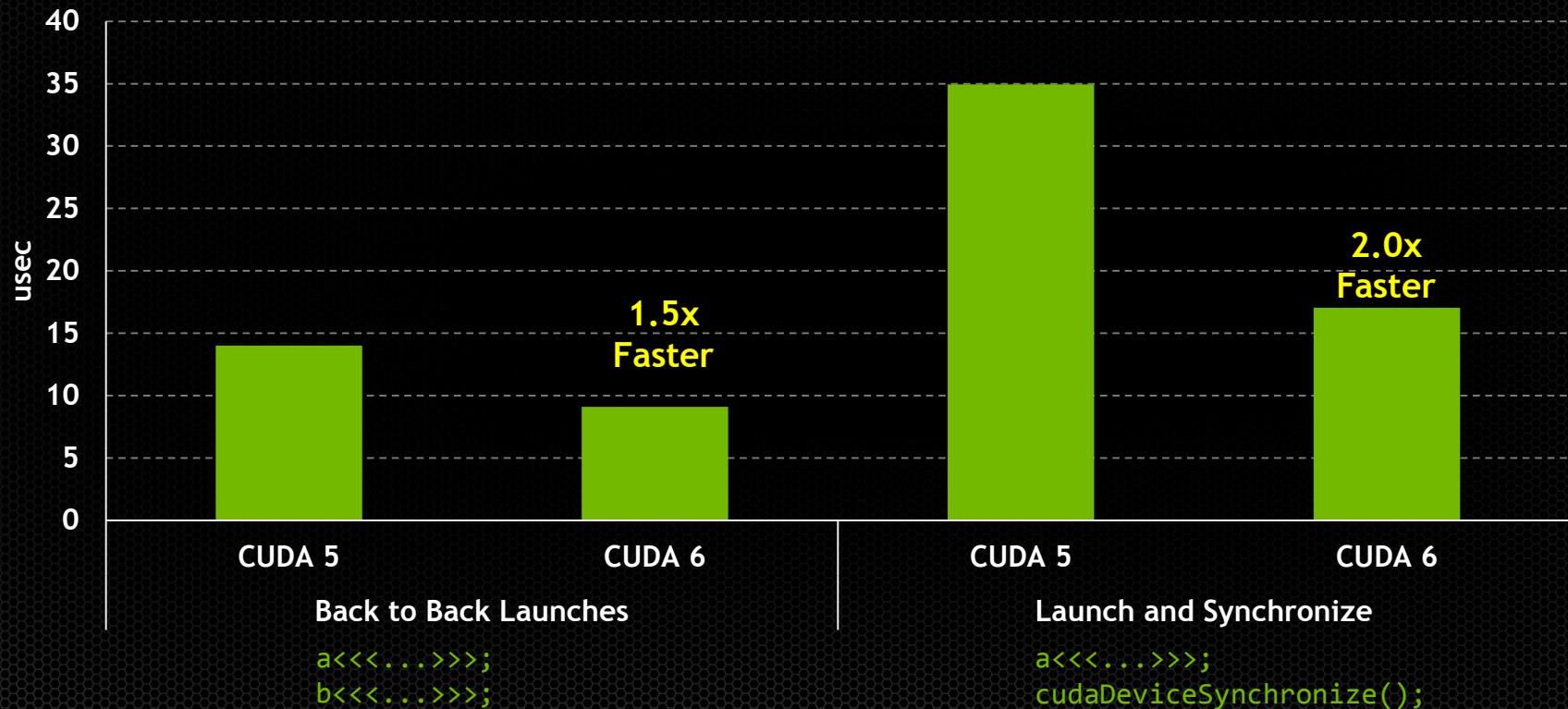


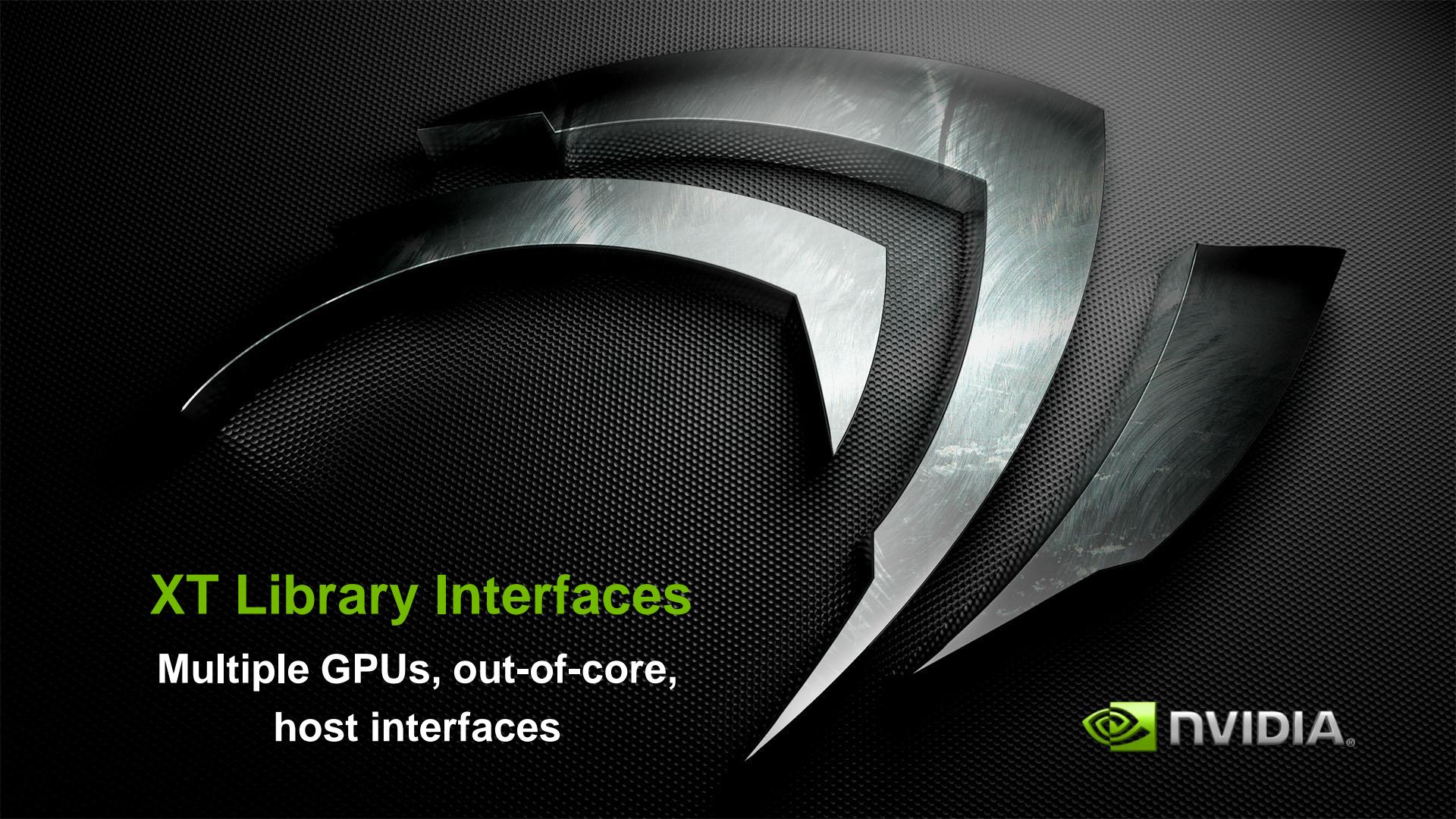
# Dynamic Parallelism Performance Improvements

Performance tuning of key use cases:

- Kernel launch
- Repeated launch of the same set of kernels
- cudaDeviceSynchronize
- Back-to-back grids in a stream

# Dynamic Parallelism Performance Improvements





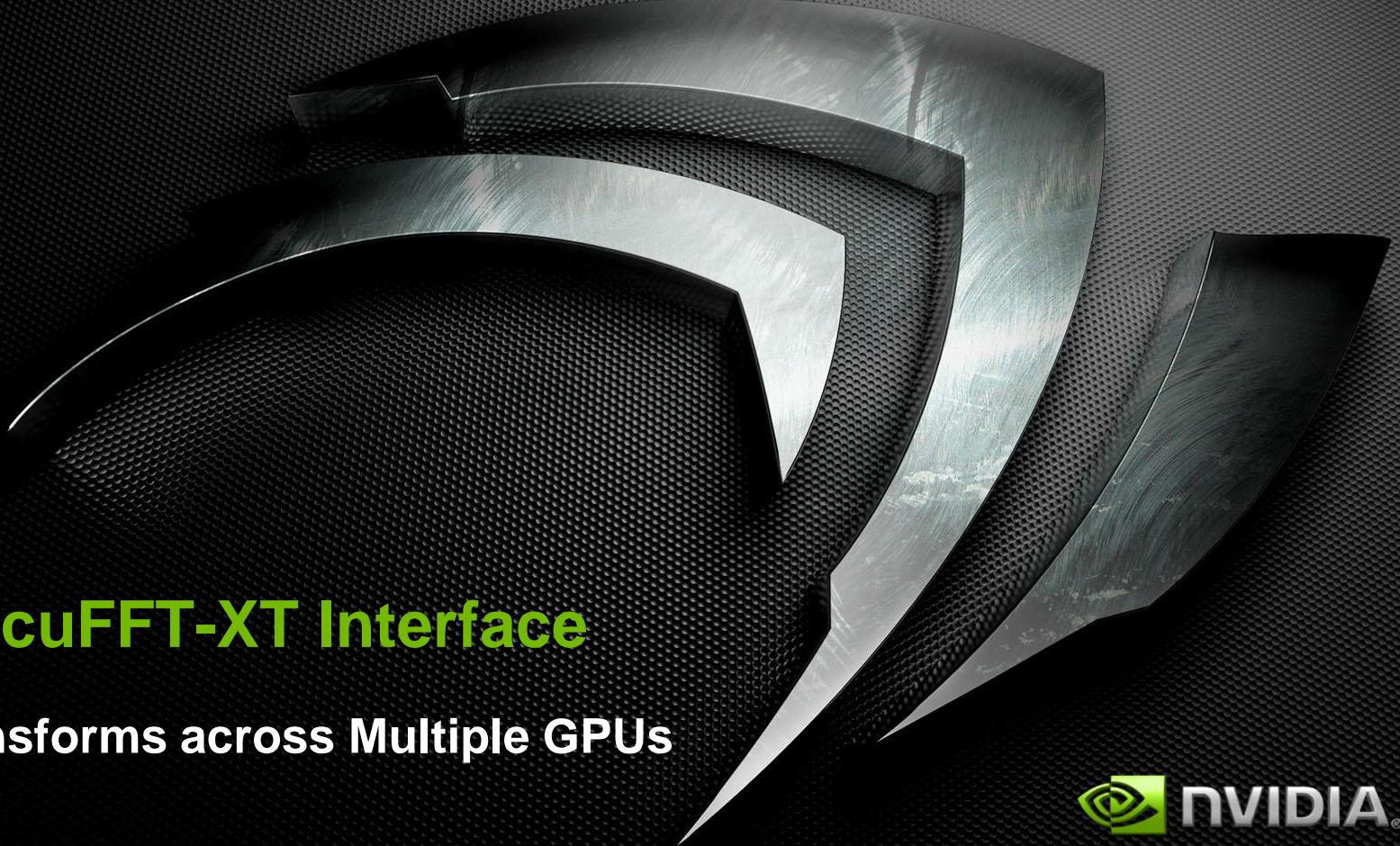
## **XT Library Interfaces**

**Multiple GPUs, out-of-core,  
host interfaces**



# eXTended Interfaces

- cuFFT and cuBLAS support in CUDA 6
- Designed to be interoperable between libraries
- Focus on scaling to >1 GPU per node
- Limiting support to 2 GPUs per node in CUDA 6

A large, metallic, three-dimensional NVIDIA logo watermark is positioned in the center of the slide. It features the iconic green checkmark and the word "NVIDIA" in a serif font, all set against a dark, textured background.

## cuFFT-XT Interface

Transforms across Multiple GPUs



# cuFFT-XT Multi-GPU Feature Overview

- Single Transforms across multiple GPUs (limited to 2 for CUDA 6)
  - C2C/Z2Z Forward and Inverse, in-place only
  - 1D, 2D, 3D
- Single Transform permutes data in place
  - 1D permutation depends on factorization, 2D and 3D data left transposed
  - Use of `cufftXtMemcpy` is advised – properly handles permuted data during copy
  - Must unpermute via `cufftXtMemcpy` D2D *between* fft and ifft calls
- Batched transforms across multiple GPUs (limited to 2 for CUDA 6)
  - All types, strides not allowed
  - Entire transforms performed local to each GPU
  - Data *not* permuted
  - Even number of batches nearly 2x vs. single GPU

# cuFFT-XT Multi-GPU Support

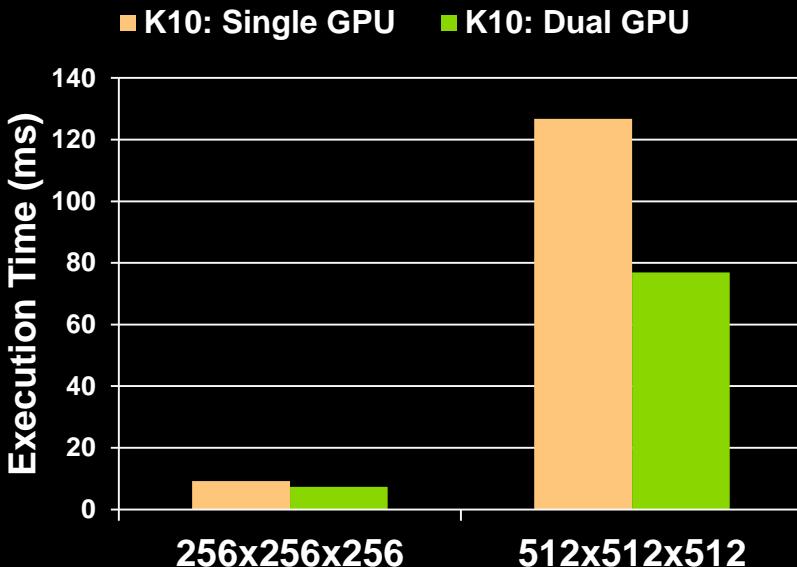
- 1. Create a Blank Plan and Associate GPUs**
- 2. Set Up Plan Parameters (same call as before)**
- 3. Create a Multi-GPU Memory Descriptor, and Allocate Memory**
- 4. Copy from Host to Devices**
- 5. Execute FFT (in-place C2C only)**
- 6. Copy from Devices to Host**

```
cufftCreate(&plan);  
cufftXtSetGPUs(plan, nGPUs, whichGPUs);  
cufftMakePlanMany(plan, rank, ...);  
cufftXtMalloc(plan, &descriptor, ...);  
cufftXtMemcpy(plan, &descriptor,  
&hostData, CUFFT_COPY_HOST_TO_DEVICE)  
cufftXtExecDescriptorC2C(plan,  
&descriptor, &descriptor,CUFFT_FORWARD)  
cufftXtMemcpy(plan, &hostData,  
&descriptor, CUFFT_COPY_DEVICE_TO_HOST)
```

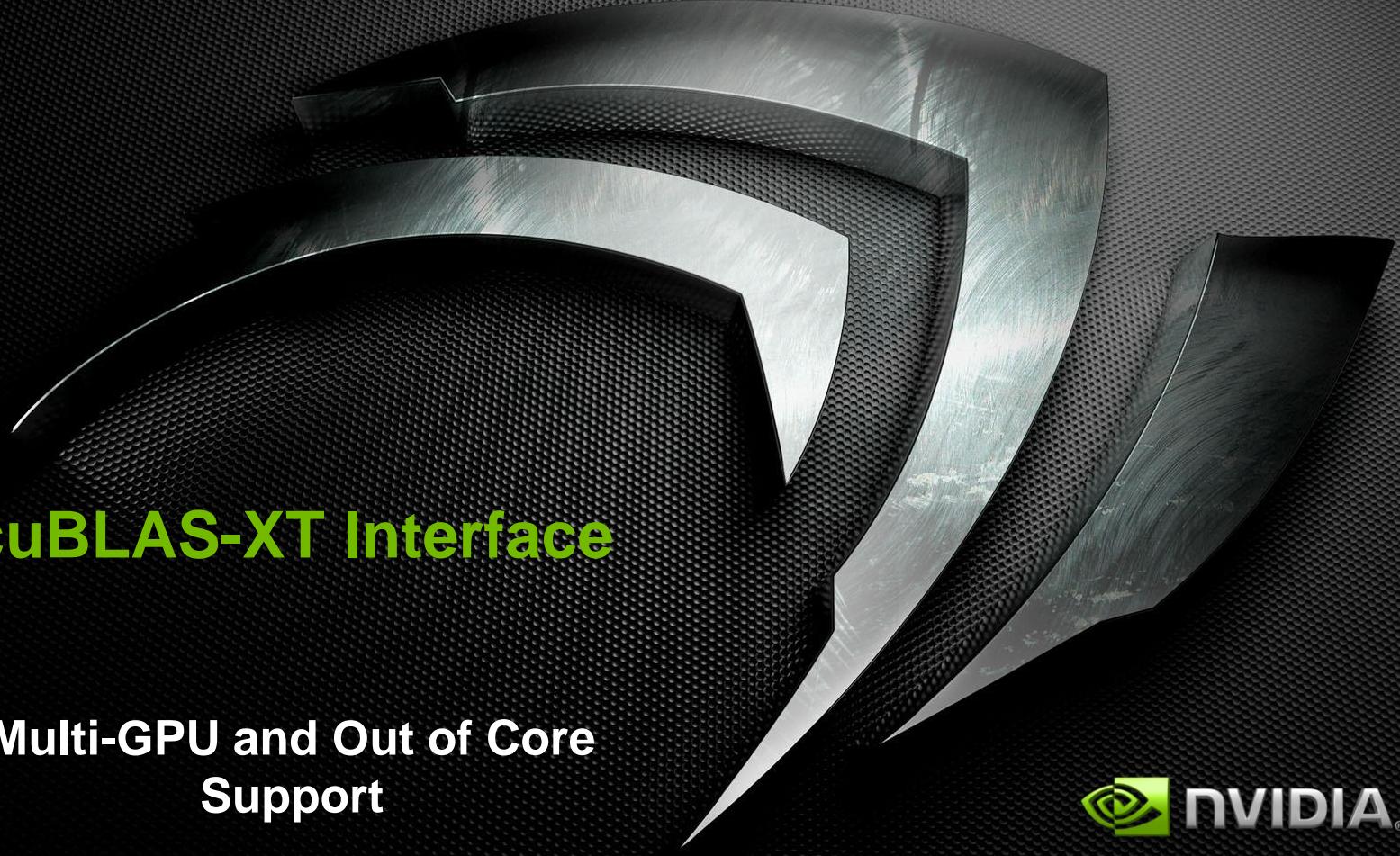
# cuFFT-XT Performance

- Tuned for Multi-GPUs
- Scaling improves for larger transforms

cuFFT 3D Performance on 2 GPUs\*



\*Does not include memcpy time

A large, metallic, three-dimensional NVIDIA logo watermark is positioned diagonally across the slide, from the top left towards the bottom right. It features the iconic green checkmark and the word "NVIDIA" in its signature font.

## cuBLAS-XT Interface

Multi-GPU and Out of Core  
Support



# cuBLAS-XT multi-GPU Feature Overview

- Most BLAS3 routines supported: GEMM,TRSM,SYRK,HERK,SYMM....
- Host interface - source & result in system memory, automatically overlaps PCIE copy
- No size limit – data must fit into system memory, no GPU size limit
- Tile-based scheme – tile size trades PCIE utilization vs. GPU utilization
- Caller pins host memory, or use autopin option
- Limited to 2 GPUs per node for CUDA 6



# cuBLAS Example: Single GPU

```
cublasCreate( &handle);  
cudaSetDevice(deviceId);  
cudaMalloc( &A, sizeA );  
cudaMalloc( &B, sizeB );  
cudaMalloc( &C, sizeC );
```

```
cudaMemcpy ( devA, A, sizeA, cudaMemcpyHostToDevice);  
cudaMemcpy( devB, B, sizeB, cudaMemcpyHostToDevice);  
cudaMemcpy( devC, C, sizeC, cudaMemcpyHostToDevice);
```

```
double alpha=1.0, beta=1.0;  
cublasDgemm( handle, CUBLAS_OP_N, CUBLAS_OP_N, m,n,k,&alpha, devA, m, devB, k, &beta, devC, m);
```

```
cudaMemcpy( A, devA, sizeA, cudaMemcpyDeviceToHost);  
cudaMemcpy( B, devB, sizeB, cudaMemcpyDeviceToHost);  
cudaMemcpy( C, devC, sizeC, cudaMemcpyDeviceToHost);  
cublasDestroy( handle);
```

## DGEMM Usage Example comparison

- Using cublas API on 1 device

Goes away with  
Unified Memory

# cuBLAS-XT Example: Multi-GPU

## DGEMM Usage Example comparison

- Using cuBLAS-XT API on 2 devices

```
cublasXtCreate( &handle);  
int device[] = { 0,3};  
cublasXtSelectDevice( handle, 2, device);  
double alpha=1.0, beta=1.0;  
cublasXtDgemm( handle, CUBLAS_OP_N, CUBLAS_OP_N, m,n,k, &alpha, A, m, B, k, &beta, C, m);  
cublasXtDestroy( handle);
```

A large, metallic, three-dimensional NVIDIA logo watermark is positioned in the center of the slide. It features the iconic interlocking 'G' and 'N' shapes, with a dark, brushed metal texture and sharp edges. The logo is set against a dark, textured background.

**NVBLAS**

“Drop-in” Level 3 BLAS



# New Drop-in NVBLAS Library

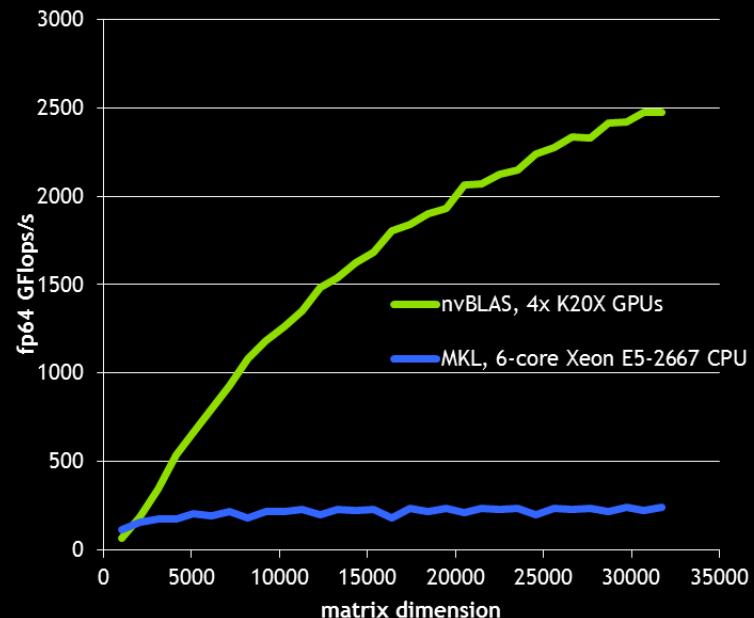
- Drop-in replacement for CPU-only BLAS
  - Automatically route BLAS3 calls to cuBLAS

- Example: Drop-in Speedup for R

```
> LD_PRELOAD=/usr/local/cuda/lib64/libnvblas.so R  
  
> A <- matrix(rnorm(4096*4096), nrow=4096, ncol=4096)  
> B <- matrix(rnorm(4096*4096), nrow=4096, ncol=4096)  
> system.time(C <- A %*% B)  
  
 user  system elapsed  
0.348  0.142  0.289
```

- Use in any app that uses standard BLAS3
  - Octave, Scilab, etc.

Matrix-Matrix Multiplication in R



# NVBLAS

- Drop-in replacement of CPU BLAS
  - Intercept standard BLAS3 calls, route to cuBLAS-XT or CPU BLAS
  - User-configurable using config file defined with environment variable NVBLAS\_CONFIG\_FILE
  - User provides CPU BLAS dynamic library location
  - BLAS3 calls are sent to GPU when appropriate

# NVBLAS – 2 Use Cases

- Relink applications with `libnvblas.so` before CPU BLAS on link line:

```
gcc myapp.c -lnvblas -lmkl_rt -o myapp
```

- On Linux, use `LD_PRELOAD` env variable

```
env LD_PRELOAD=libnvblas.so myapp
```

A large, metallic, three-dimensional NVIDIA logo watermark is positioned in the center of the slide. It features the iconic interlocking 'V' shape in a dark, polished metal finish, set against a dark, textured background.

# Tools



# Advanced Kernel Optimization Tools

```

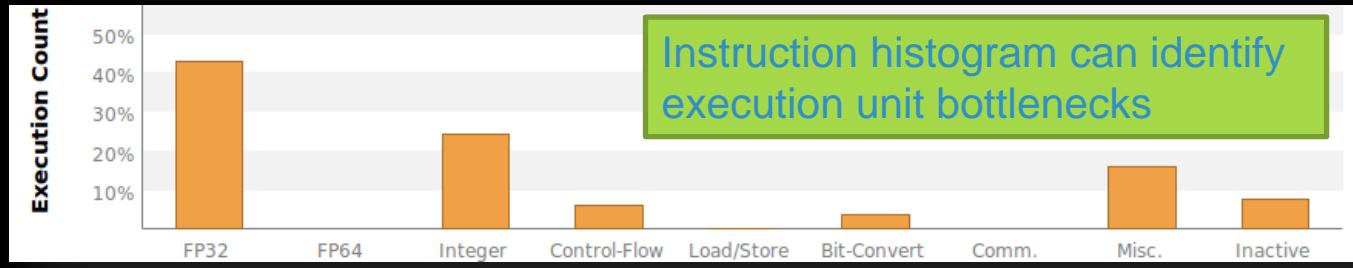
Line   Exec Count File - /home/harrism/p4/sw/devrel/CUDAToolkit/samples/3_Imaging/convolutionSe
70      } 
71      //Load right halo
72      #pragma unroll
73
74      for (int i = ROWS_HALO_STEPS + ROWS_RESULT_STEPS; i < ROWS_HALO_STEPS +
75      { 
76          s_Data[threadIdx.y][threadIdx.x + i * ROWS_BLOCKDIM_X] = (imageW - baseX:
77      }
78
79
80      //Compute and store results
81      __syncthreads();
82      #pragma unroll
83
84      for (int i = ROWS_HALO_STEPS; i < ROWS_HALO_STEPS + ROWS_RESULT_STEPS; i++)
85      {
86          float sum = 0;
87
88          #pragma unroll
89
90          for (int j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; j++)
91          {
92              sum += c_Kernel[KERNEL_RADIUS - j] * s_Data[threadIdx.y][threadIdx.x + i * R
93          }
94
95          d_Dst[i * ROWS_BLOCKDIM_X] = sum;
96
97      }
98      36864 }

```

Detected Hot Spot

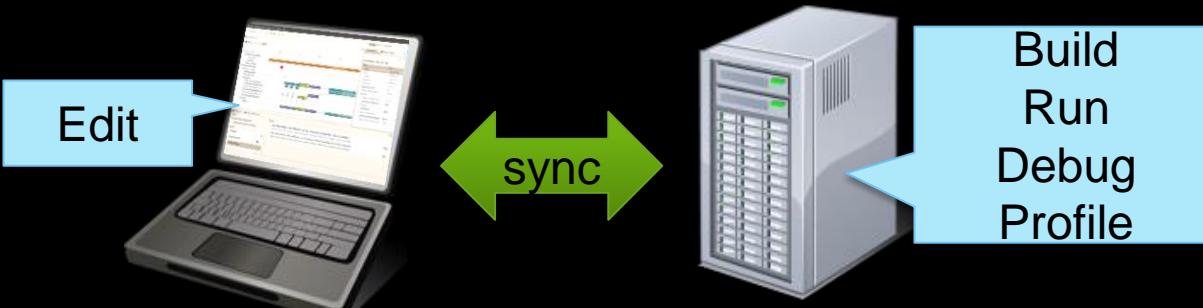
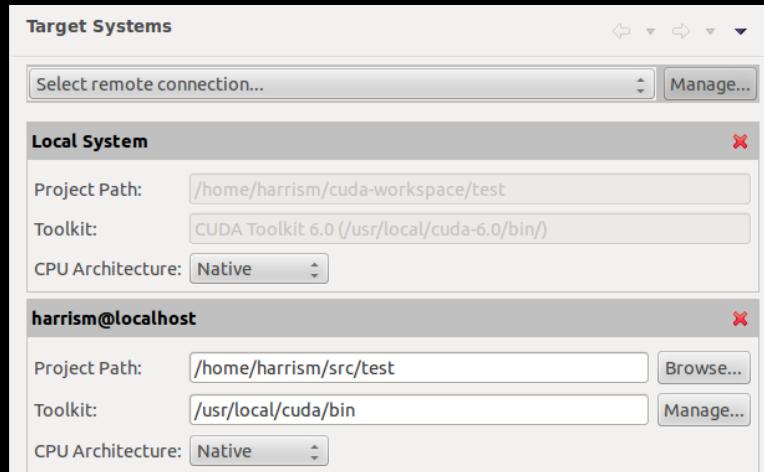
| Exec Count | Disassembly                       |
|------------|-----------------------------------|
| 36864      | LDS R0, [R29+0x20];               |
| 36864      | LDS R6, [R29+0x24];               |
| 36864      | FFMA R16, R0, c[0x3][0x40], R1;   |
| 36864      | LDS R4, [R29+0x60];               |
| 36864      | FFMA R18, R6, c[0x3][0x3c], R2;   |
| 36864      | LDS R15, [R29+0x28];              |
| 36864      | FFMA R23, R4, c[0x3][0x40], R3;   |
| 36864      | LDS R17, [R29+0x64];              |
| 36864      | LDS R21, [R29+0x2c];              |
| 36864      | LDS R0, [R29+0xa0];               |
| 36864      | LDS R20, [R29+0x68];              |
| 36864      | FFMA R24, R0, c[0x3][0x40], RZ;   |
| 36864      | LDS R14, [R29+0x30];              |
| 36864      | LDS R19, [R29+0xa4];              |
| 36864      | FFMA R22, R15, c[0x3][0x38], R18; |
| 36864      | LDS R16, [R29+0x6c];              |
| 36864      | LDS R10, [R29+0x34];              |
| 36864      | LDS R15, [R29+0xa8];              |
| 36864      | FFMA R24, R19, c[0x3][0x3c], R24; |
| 36864      | FFMA R23, R17, c[0x3][0x3c], R23; |
| 36864      | LDS R18, [R29+0x70];              |
| 36864      | FFMA R21, R21, c[0x3][0x34], R22; |
| 36864      | LDS R13, [R29+0x38];              |
| 36864      | LDS R17, [R29+0xac];              |
| 36864      | FFMA R22, R20, c[0x3][0x38], R23; |
| 36864      | FFMA R21, R14, c[0x3][0x30], R21; |
| 36864      | LDS R20, [R29+0x74];              |
| 36864      | LDS R12, [R29+0x3c];              |

Corresponding Assembly



# Remote Development with Nsight Eclipse Edition

- Local IDE, remote build & run via ssh
- Ideal for developers targeting:
  - HPC cluster environments
  - Embedded systems
- Full debugging & profiling via remote connection

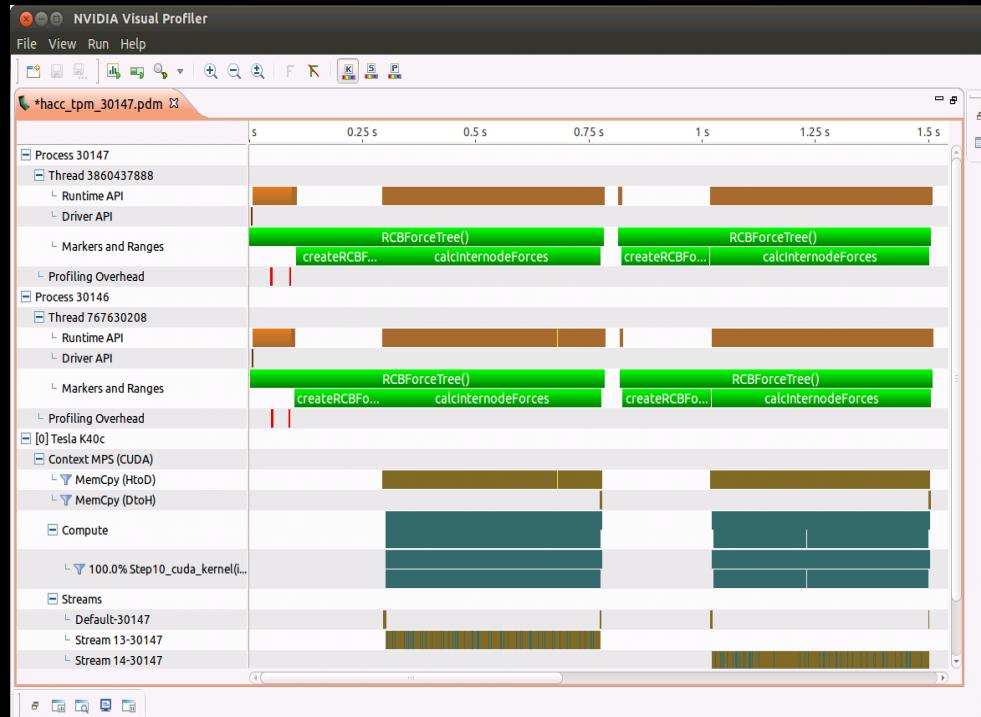


# CUDA tools for MPS (Multi-Process Server)



## Profiler

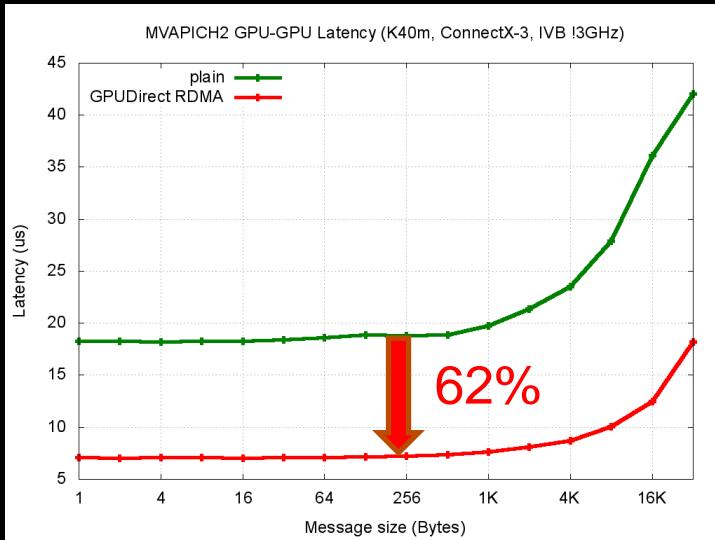
- Verify GPU concurrency among multiple MPI ranks
- Identify node-level performance bottlenecks due to GPU scheduling
- Run CUDA-MEMCHECK on apps running on MPS



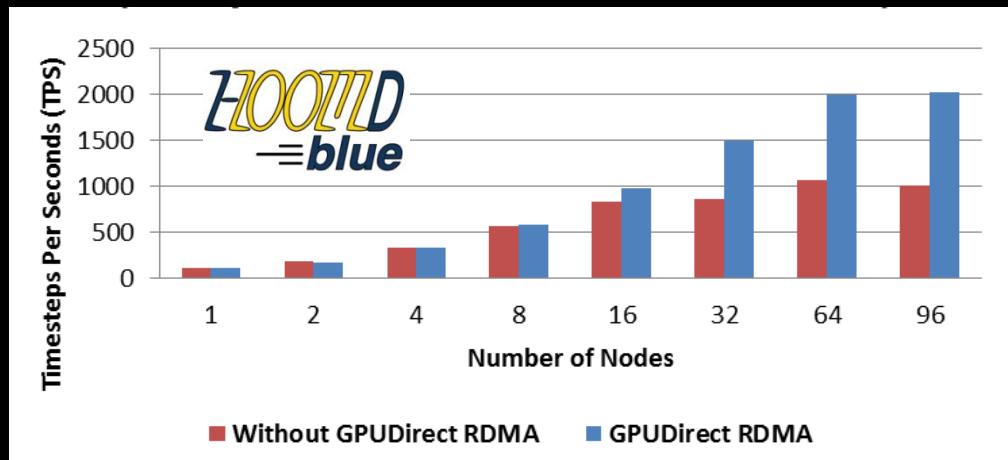
# Improving Cross-Cluster Communication

## GPUDirect RDMA

Reduced inter-node latency



Better MPI Application Scaling



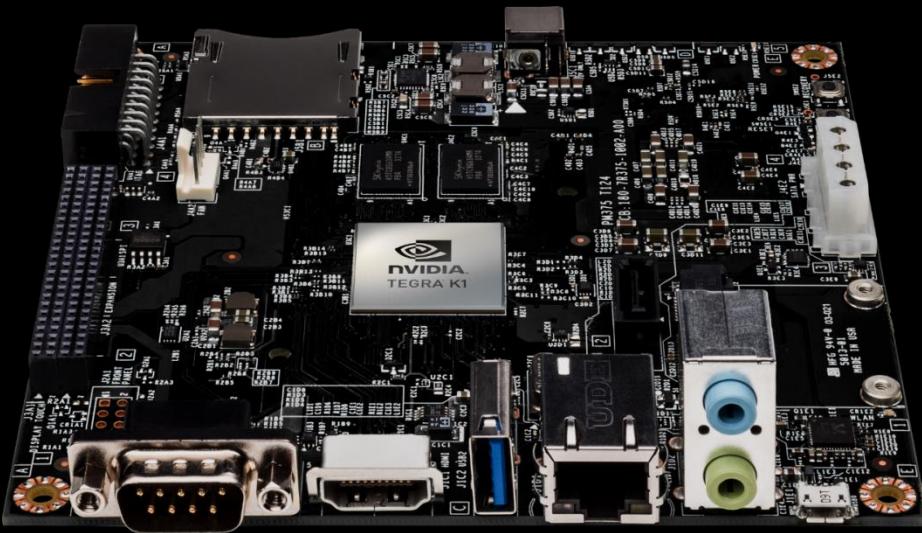
HOOMD-blue

[http://hpcadvisorycouncil.com/pdf/HOOMDblue\\_Analysis\\_and\\_Profiling.pdf](http://hpcadvisorycouncil.com/pdf/HOOMDblue_Analysis_and_Profiling.pdf)

# CUDA 6 for Embedded Applications



## JETSON TK1: THE WORLD'S 1st EMBEDDED SUPERCOMPUTER



Development Platform for Embedded  
Computer Vision, Robotics, Medical

192 Cores · 300+ GFLOPS  
CUDA 6 (including Unified Memory)  
OpenGL 4.4, DX 11 & OpenGL ES 3.0

Available Now

# Resources: [developer.nvidia.com/cudazone](https://developer.nvidia.com/cudazone)

- Parallel Forall: [devblogs.nvidia.com/parallelforall](https://devblogs.nvidia.com/parallelforall)
  - CUDAcasts at [bit.ly/cudacasts](https://bit.ly/cudacasts)
- Self-paced labs: [nvidia.qwiklab.com](https://nvidia.qwiklab.com)
  - 90-minute labs, simply need a supported web browser
- Documentation: [docs.nvidia.com](https://docs.nvidia.com)
- Technical Questions:
  - NVIDIA Developer forums [devtalk.nvidia.com](https://devtalk.nvidia.com)
  - Search or ask on [stackoverflow.com/tags/cuda](https://stackoverflow.com/tags/cuda)