Supported Platforms

**Windows**
- Windows XP, Vista, 7, 8, 8.1, Server 2008 R2, Server 2012 R2

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

System Memory

GPU Memory

Developer View With Unified Memory

Unified Memory
### Super Simplified Memory Management Code

#### CPU Code

```c
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

```c
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
<table>
<thead>
<tr>
<th>System Requirements</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPU</strong></td>
<td>Kepler &amp; Maxwell</td>
<td>(GK10x+ or GM10x+, i.e. SM3.0+)</td>
</tr>
<tr>
<td><strong>Operating System</strong></td>
<td>64-bit required</td>
<td></td>
</tr>
<tr>
<td><strong>Linux</strong></td>
<td>Kernel 2.6.18+</td>
<td>(all CUDA-supported distros, not ARM)</td>
</tr>
<tr>
<td><strong>Windows</strong></td>
<td>Win7 or Win8</td>
<td>(WDDM &amp; TCC no XP/Vista)</td>
</tr>
<tr>
<td><strong>Mac OSX</strong></td>
<td>Not supported in CUDA 6</td>
<td></td>
</tr>
<tr>
<td><strong>Linux on ARM</strong></td>
<td>Supported with Tegra K1</td>
<td></td>
</tr>
</tbody>
</table>
CUDA Memory Types

Zero-Copy
\texttt{cudaMallocHost}(&x, 4)
- Allocation fixed in CPU mem
- PCIe access for all GPUs
- Local access for CPU

Unified Virtual Addressing
\texttt{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)
\texttt{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
Eliminate Deep Copying

```c
struct dataElem {
    int prop1;
    int prop2;
    char *text;
};
```
Eliminate Deep Copying

```c
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

```c
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);
}
```
Eliminate Deep Copying

```c
#include<...>

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

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

```cpp
void kernel(dataElem data) {
    // Code...
}
```
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

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

```cpp
void kernel(dataElem data) {
}
```
C++ objects migrate easily when allocated on managed heap

Overload `new` operator to use C++ in unified memory region

```c++
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

```cpp
// 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 constructors.

```cpp
// 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;
        cudaMemcpyManaged(&data, length+1);
        strcpy(data, s.data);
    }
};
```

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

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

Single pointer to data makes object references just work

```cpp
__global__ void kernel_by_ref(dataElem &data) { ...

```

“Hello World”
C++ Pass By Value

Copy constructors from CPU create GPU Usable objects

```cpp
__global__ void kernel_by_val(dataElem data) {
    "Hello World"
}
```

By-value copy in managed memory
Dynamic Parallelism Improvements
Dynamic Parallelism - Recap

CPU → Fermi GPU

CPU → Kepler GPU
Familiar Syntax and Programming Model

```c
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

Finite Pending Launch Buffer

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

Back to Back Launches
a<<<...>>>(CUDA 5); b<<<...>>>(CUDA 6);

Launch and Synchronize
a<<<...>>>(CUDA 5); cudaDeviceSynchronize();

CUDA 5: 1.5x Faster
CUDA 6: 2.0x Faster

CUDA 5
CUDA 6
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
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
   - `cufftCreate(&plan);`
   - `cufftXtSetGPUs(plan, nGPUs, whichGPUs);`
   - `cufftMakePlanMany(plan, rank, ...);`
   - `cufftXtMalloc(plan, &descriptor, ...);`
   - `cufftXtMemcpy(plan, &descriptor, &hostData, CUFFT_COPY_HOST_TO_DEVICE)`

2. Set Up Plan Parameters (same call as before)
   - `cufftXtExecDescriptorC2C(plan, &descriptor, &descriptor, CUFFT_FORWARD)`
   - `cufftXtMemcpy(plan, &hostData, &descriptor, CUFFT_COPY_DEVICE_TO_HOST)`

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
cuFFT-XT Performance

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

**cuFFT 3D Performance on 2 GPUs**

*K10: Single GPU  K10: Dual GPU*

*Does not include memcpy time*
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 ZGEMM Performance on 2 GPUS**

![Performance Graph]

GFLOPS vs. Matrix Size (NxN) for 1 K20c and 2 K20c GPUs.
cuBLAS Example: Single GPU

DGEMM Usage Example comparison

Using cublas API on 1 device

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);

Goes away with Unified Memory
cuBLAS-XT Example: Multi-GPU

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);
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

```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.
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
Tools
Advanced Kernel Optimization Tools

Corresponding Assembly

Instruction histogram can identify execution unit bottlenecks
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

![Diagram showing remote development process with Nsight Eclipse Edition](image)
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

MVAPICH2 GPU-GPU Latency (K40m, ConnectX-3, 1x 13GHz)

Latency (us)

Message size (Bytes)

62%

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

Parallel Forall: devblogs.nvidia.com/parallelforall
  CUDACasts at bit.ly/cudacasts

Self-paced labs: nvidia.qwiklab.com
  90-minute labs, simply need a supported web browser

Documentation: docs.nvidia.com

Technical Questions:
  NVIDIA Developer forums devtalk.nvidia.com
  Search or ask on stackoverflow.com/tags/cuda