NEW FEATURES IN CUDA 6 MAKE GPU ACCELERATION EASIER

MARK HARRIS

CUDA 6

1. Unified Memory
2. XT and Drop-in Libraries
3. GPUDirect RDMA in MPI
4. Developer Tools

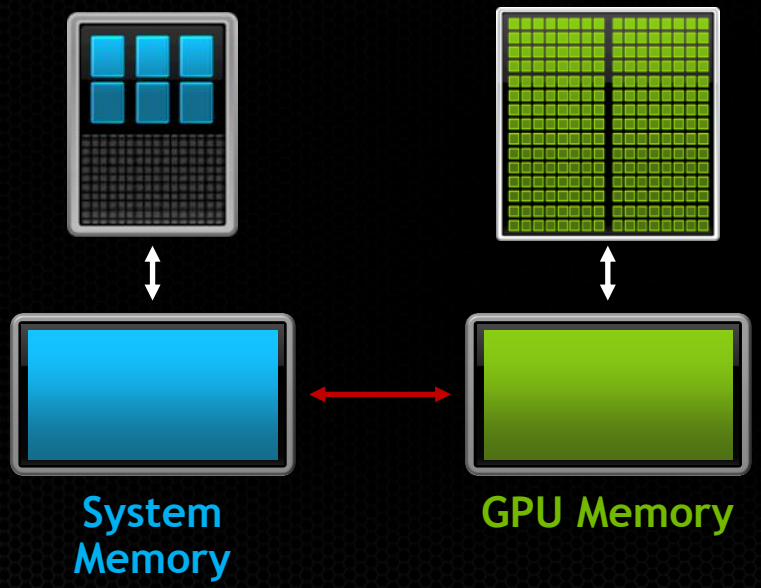
CUDA 6

1. Unified Memory
2. XT and Drop-in Libraries
3. GPUDirect RDMA in MPI
4. Developer Tools

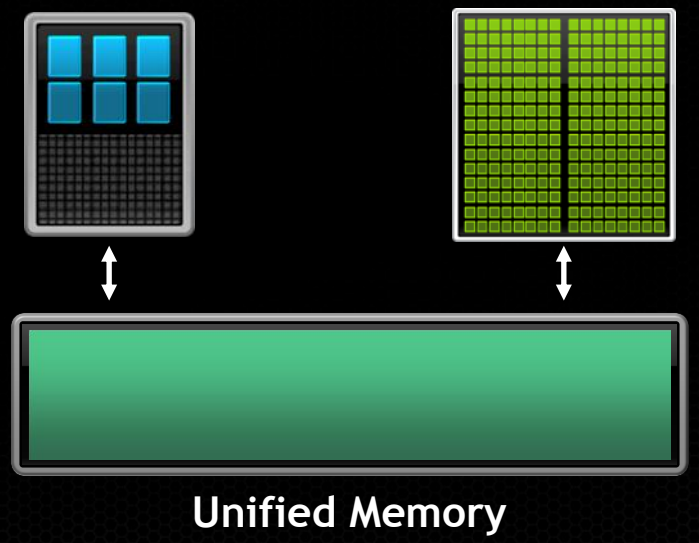
Unified Memory

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);
}
```

Unified Memory Delivers

1. Simpler Programming & Memory Model

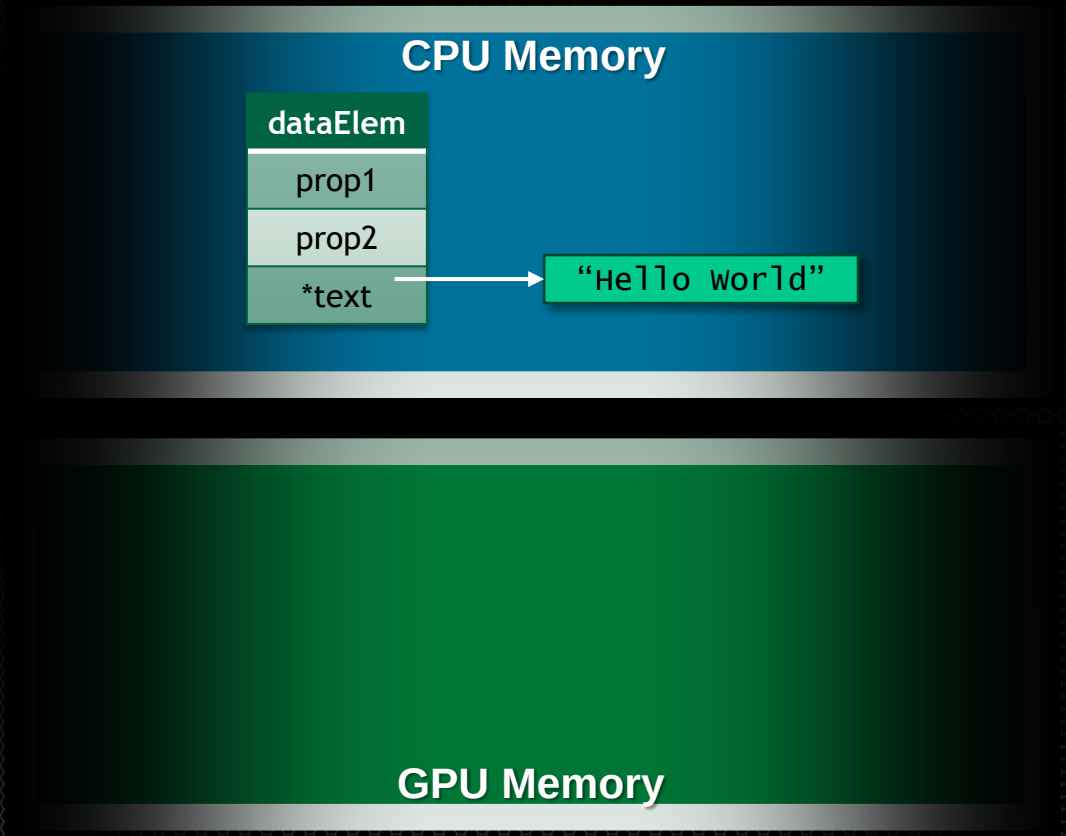
- Single pointer to data, accessible anywhere
- Tight language integration
- Greatly simplifies code porting

2. Performance Through Data Locality

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

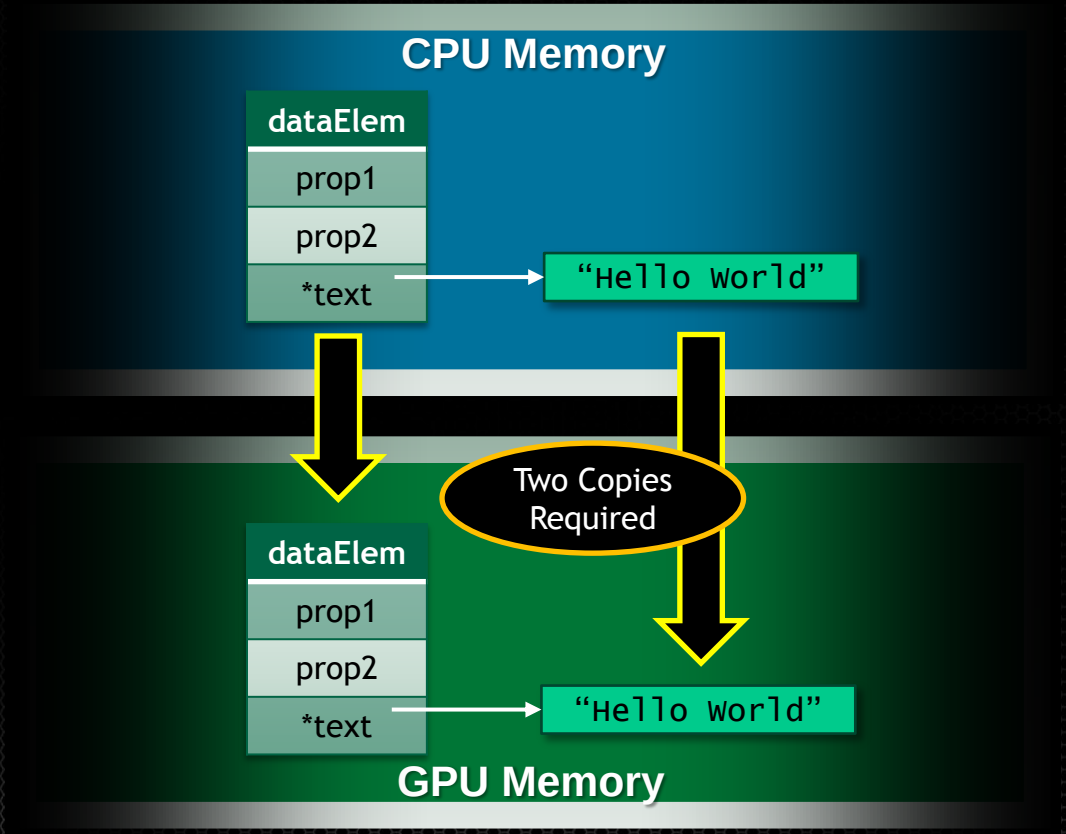
Simpler Memory Model: Eliminate Deep Copies

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



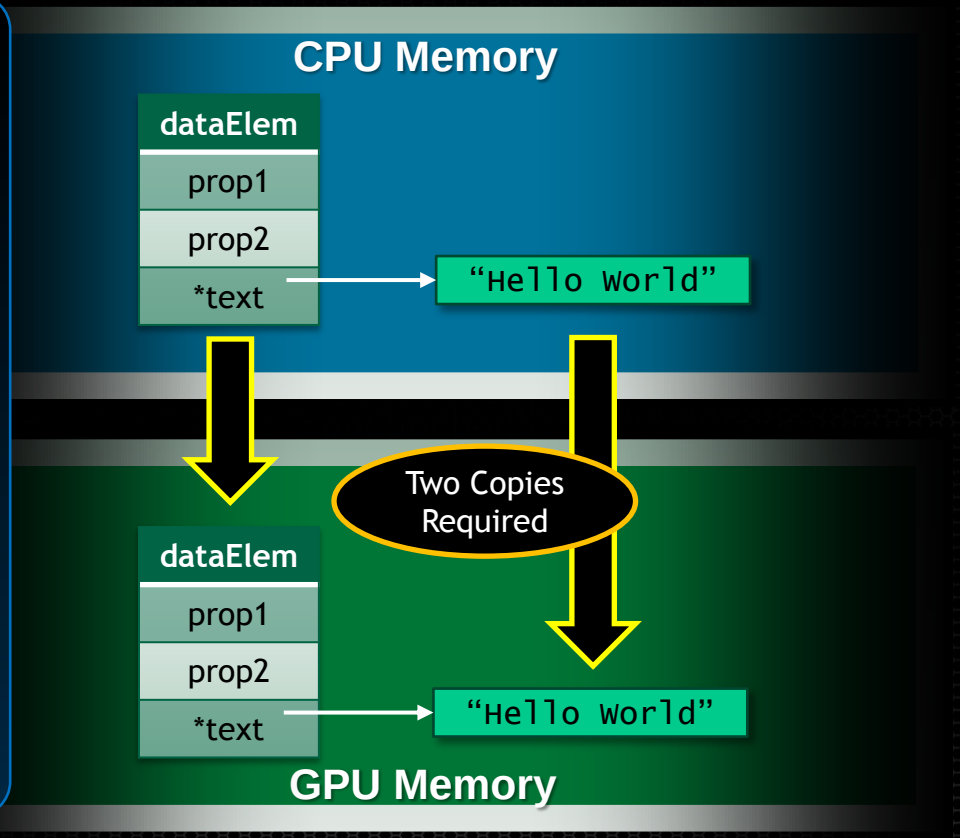
Simpler Memory Model: Eliminate Deep Copies

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



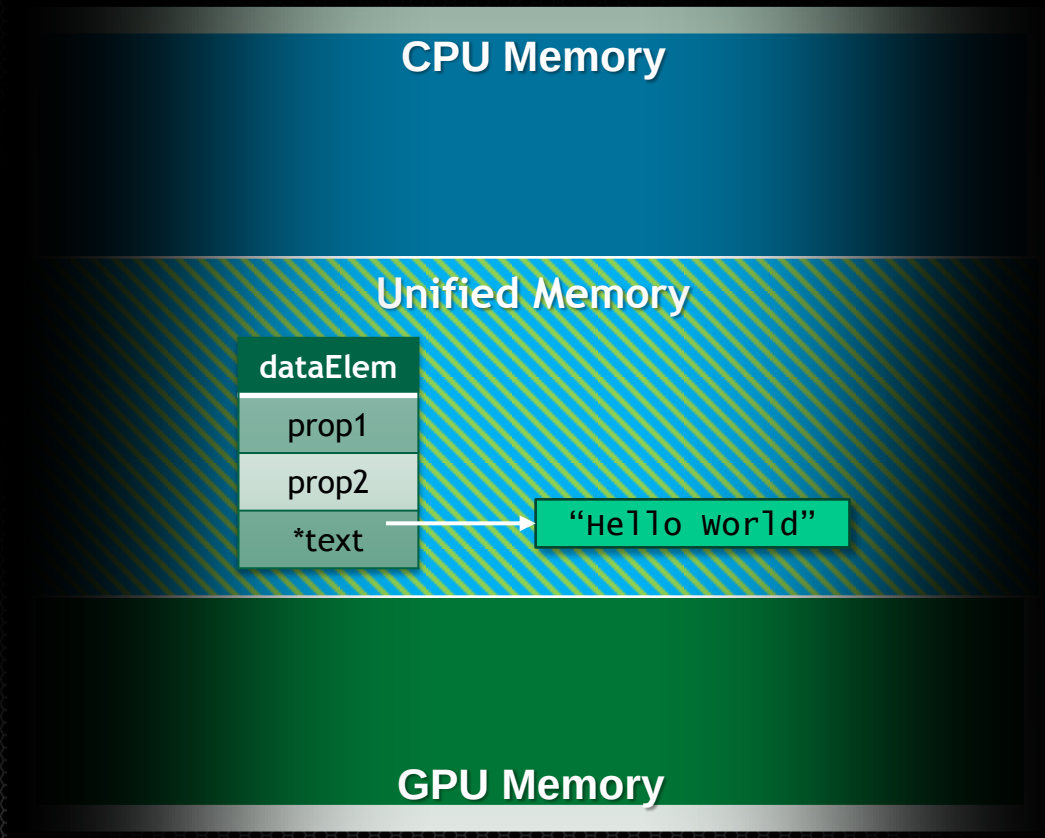
Simpler Memory Model: Eliminate Deep Copies

```
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);
}
```



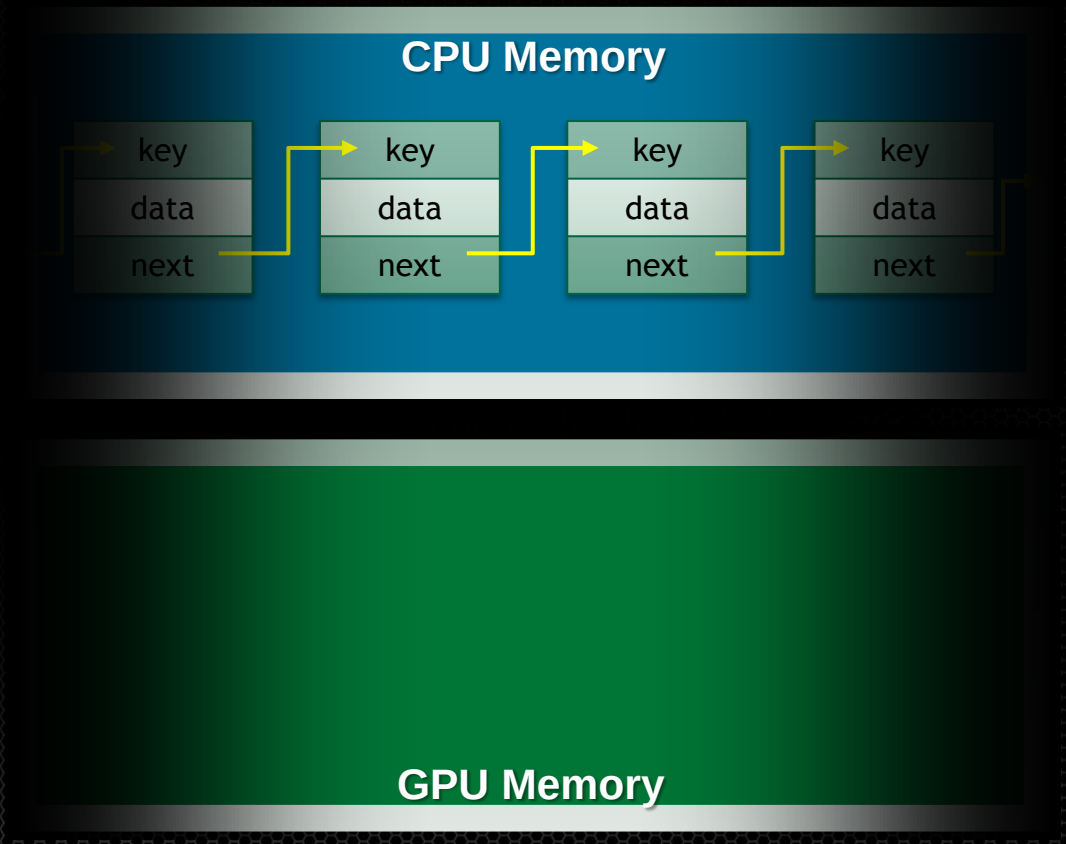
Simpler Memory Model: Eliminate Deep Copies

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



Simpler Memory Model

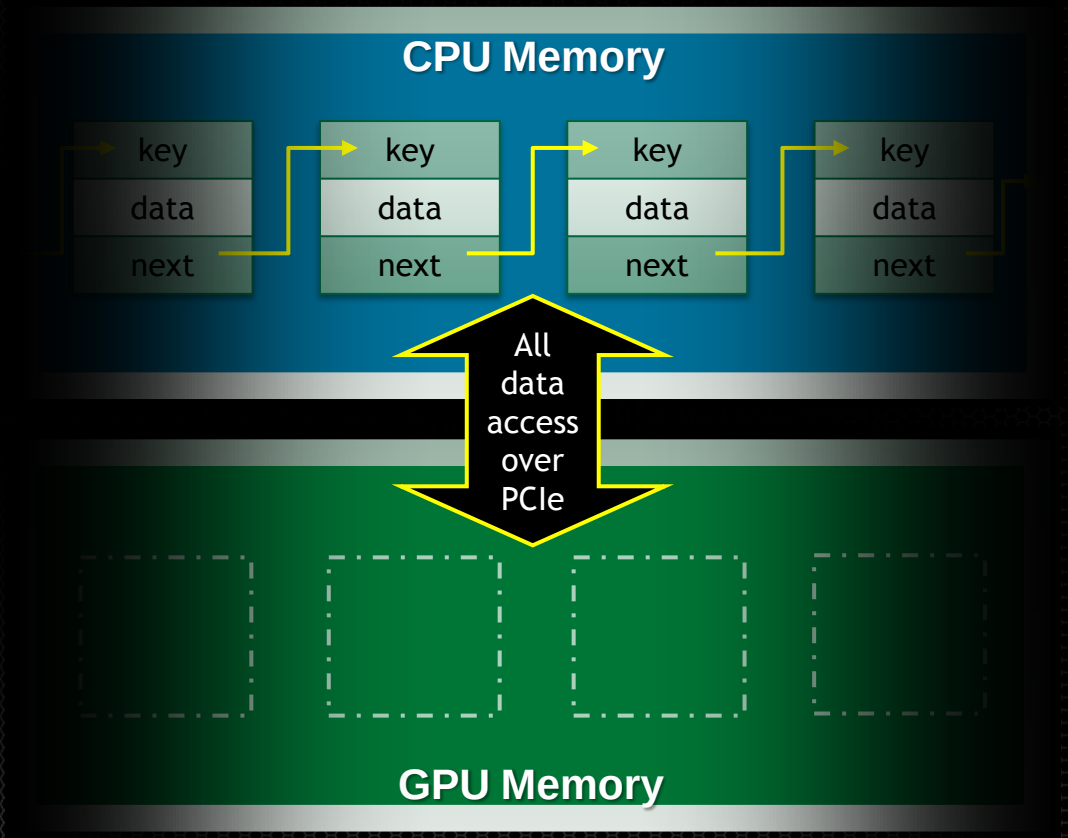
Example: GPU & CPU Shared Linked Lists



Simpler Memory Model

Example: GPU & CPU Shared Linked Lists

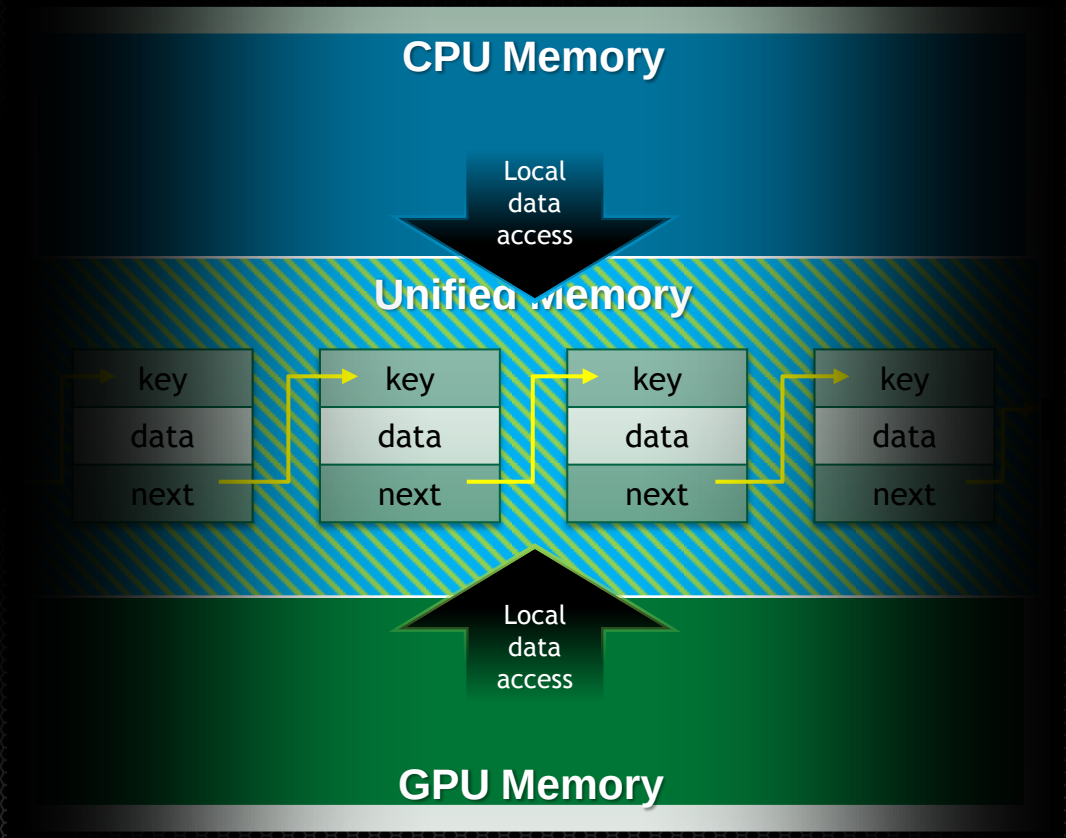
- Only practical option is to use zero-copy (pinned system) memory
- GPU accesses at PCIe bandwidth
- GPU accesses at very high latency



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

Unified Memory with C++

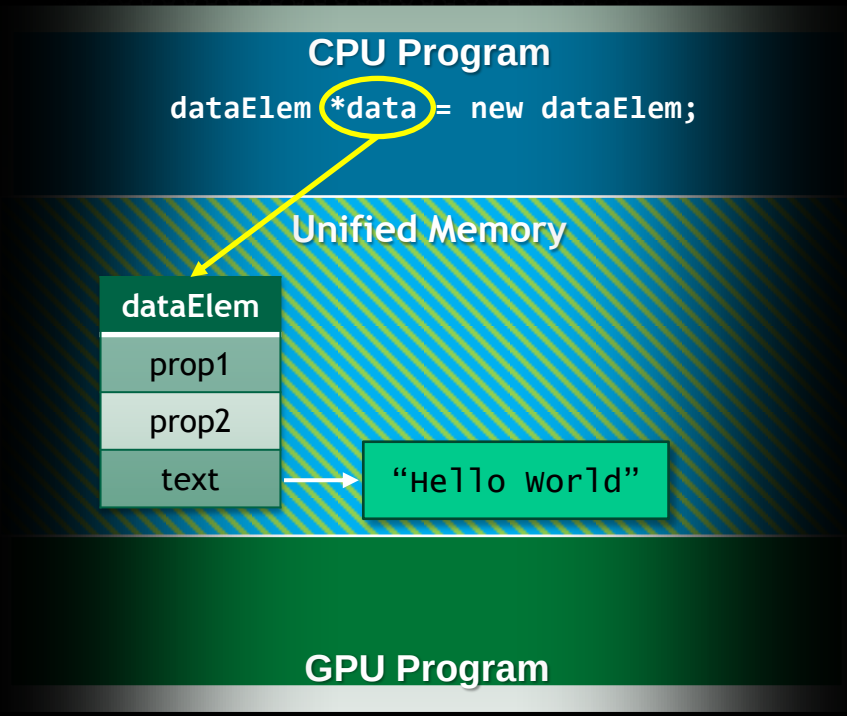
A Powerful Combination

C++ objects migrate easily when allocated on managed heap

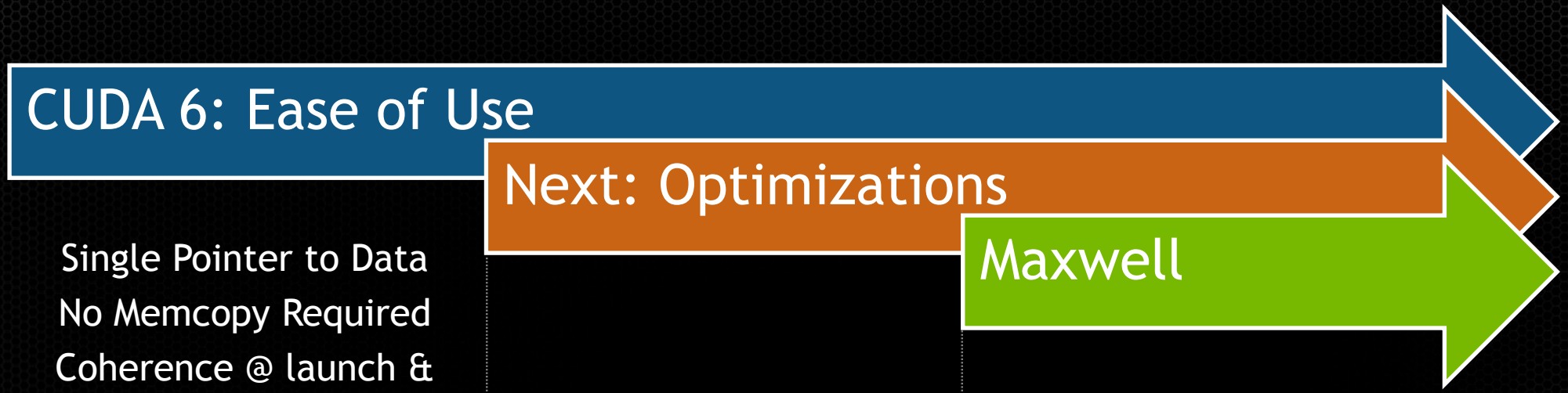
- Overload *new* operator to use C++ in unified memory region
- Deep copies, pass-by-value, pass-by-reference: JUST WORKS

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

    void operator delete(void *ptr) {
        cudaFree(ptr);
    }
};
```

```
// Inherit from "Managed",
// C++ now handles our deep copies
class dataElem : public Managed {
        int prop1;
        int prop2;
        String text;
};
```



Unified Memory Roadmap



Single Pointer to Data

No Memcopy Required

Coherence @ launch & sync

Shared C/C++ Data Structures

Prefetching

Migration Hints

Additional OS Support

System Allocator Unified

Stack Memory Unified

HW-Accelerated Coherence

CUDA 6

1 Unified Memory

2 XT and Drop-in Libraries

3 GPUDirect RDMA in MPI

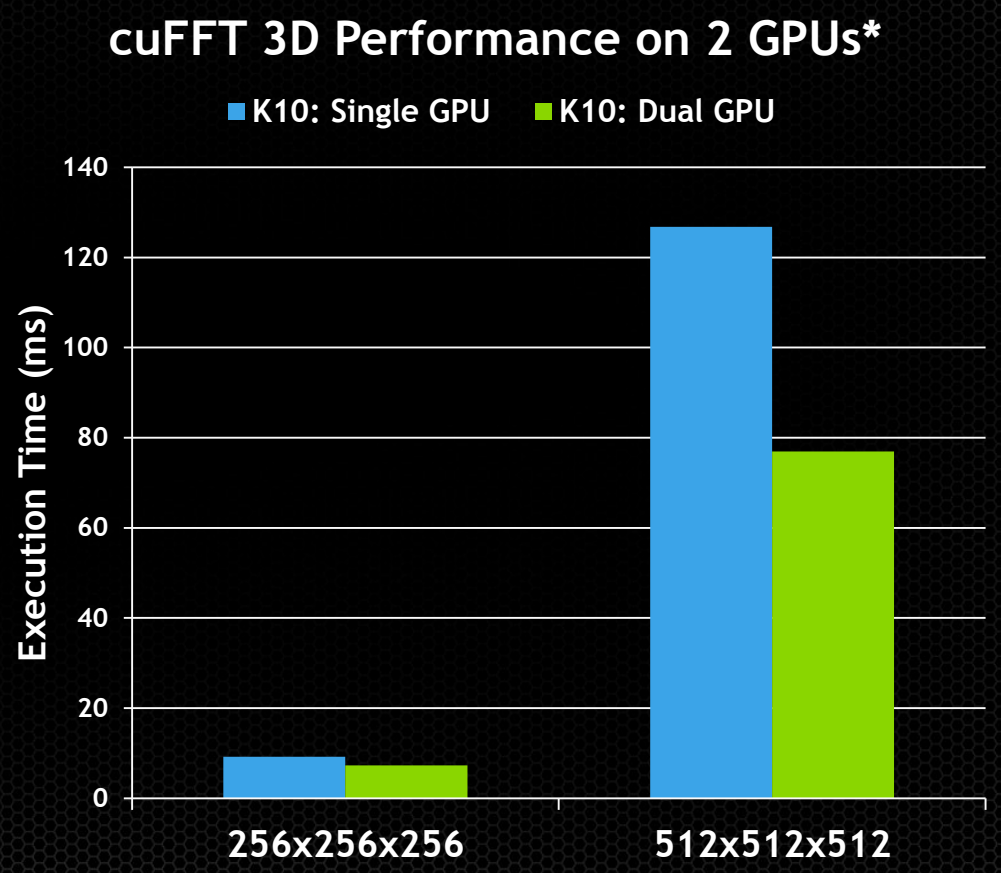
4 Developer Tools

Extended (XT) Library Interfaces

- Automatic Scaling to >1 GPU per node
- cuFFT and cuBLAS level 3
- Out-of-core operations: e.g. very large GEMM
- BLAS 3 Host Interfaces: automatically overlaps memory transfers

Multi-GPU cuFFT

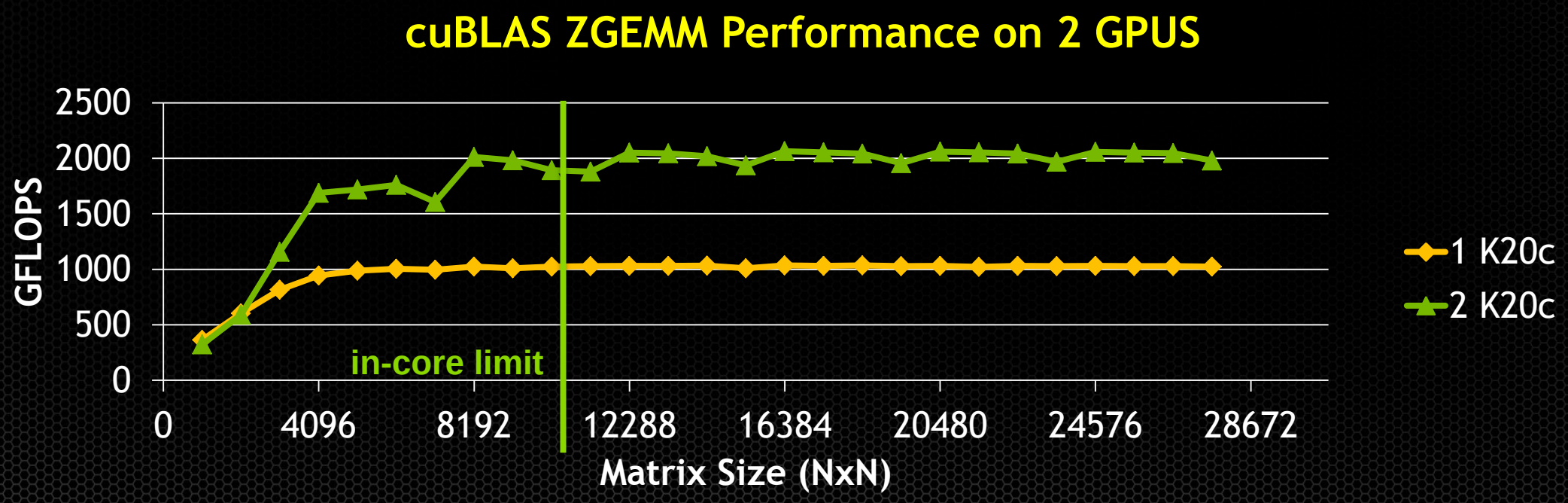
- Single & Batch Transforms across multiple GPUs (max 2 in CUDA 6)
- Tuned for multi-GPU cards (K10)
  - Better scaling for larger transforms



*Does not include memcpy time

Multi-GPU cuBLAS

- Single function call automatically spreads work across two GPUs
- Source and result data in system memory
- Supports matrices > size of memory (out-of-core)
- All BLAS Level-3 routines



New Drop-in NVBLAS Library

- Drop-in replacement for CPU-only BLAS
  - Automatically routes standard BLAS3 calls to cuBLAS
  - Optionally configure which routines and matrix sizes are accelerated
  - User provides CPU-only BLAS dynamic library location

- Simply re-link or change library load order

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

- or -

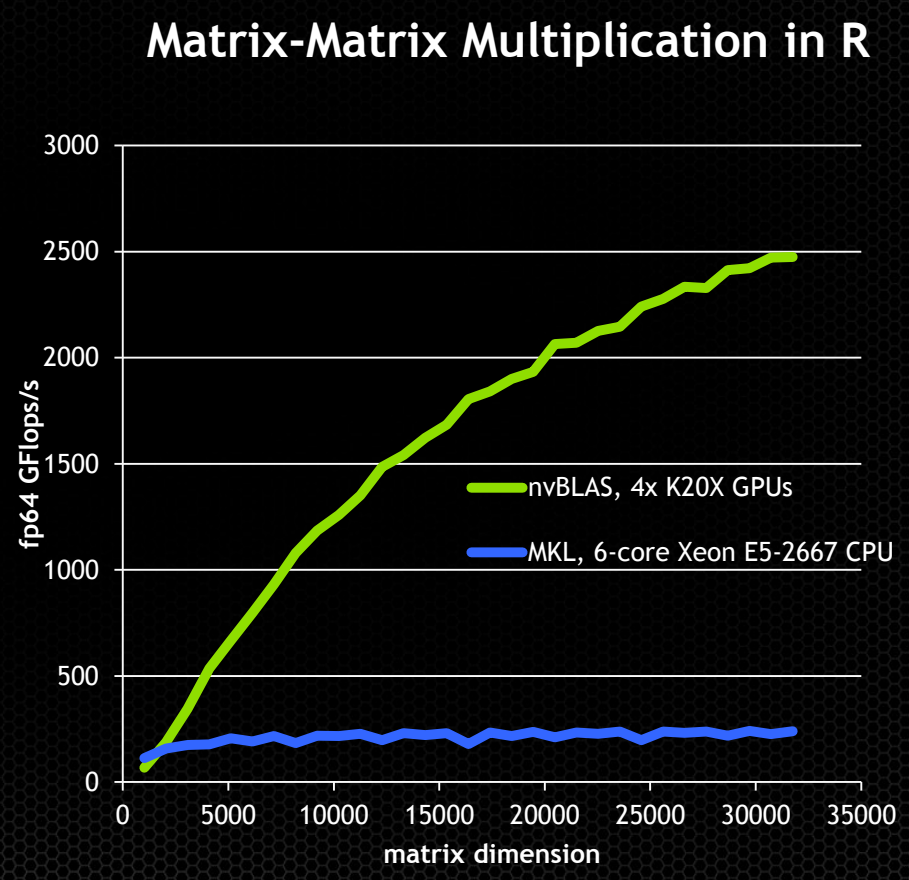
```
env LD_PRELOAD=libnvblas.so myapp
```

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.



CUDA 6

1 Unified Memory

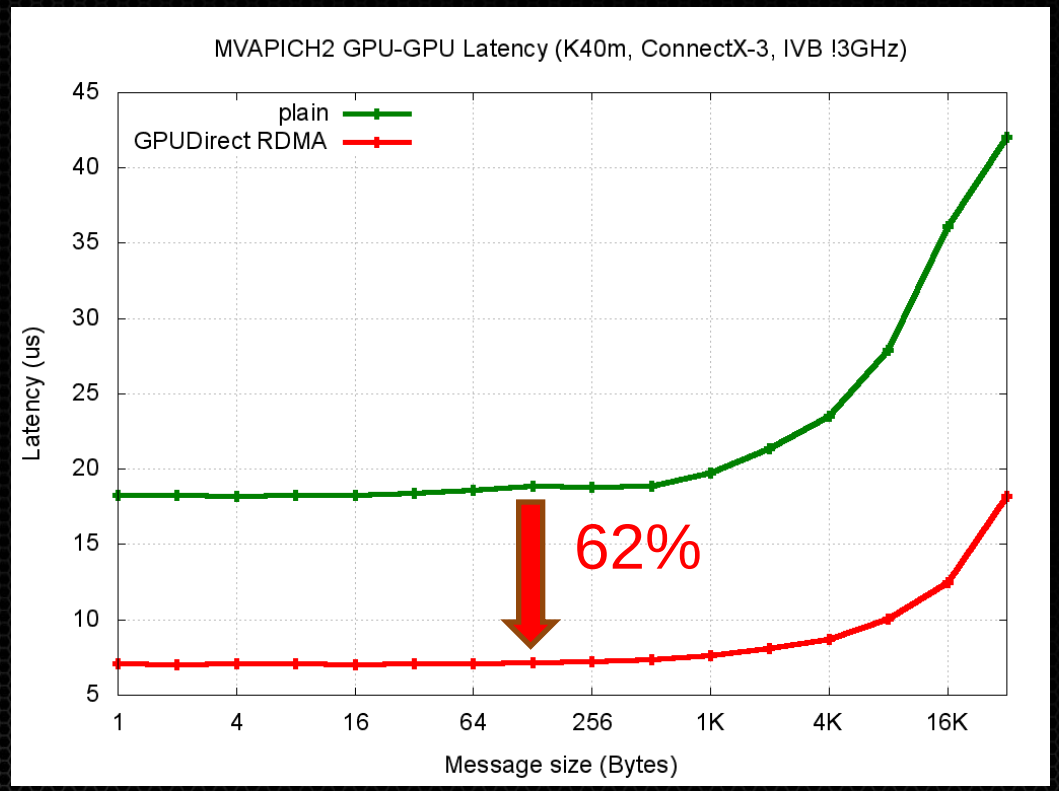
2 XT and Drop-in Libraries

3 GPUDirect RDMA in MPI

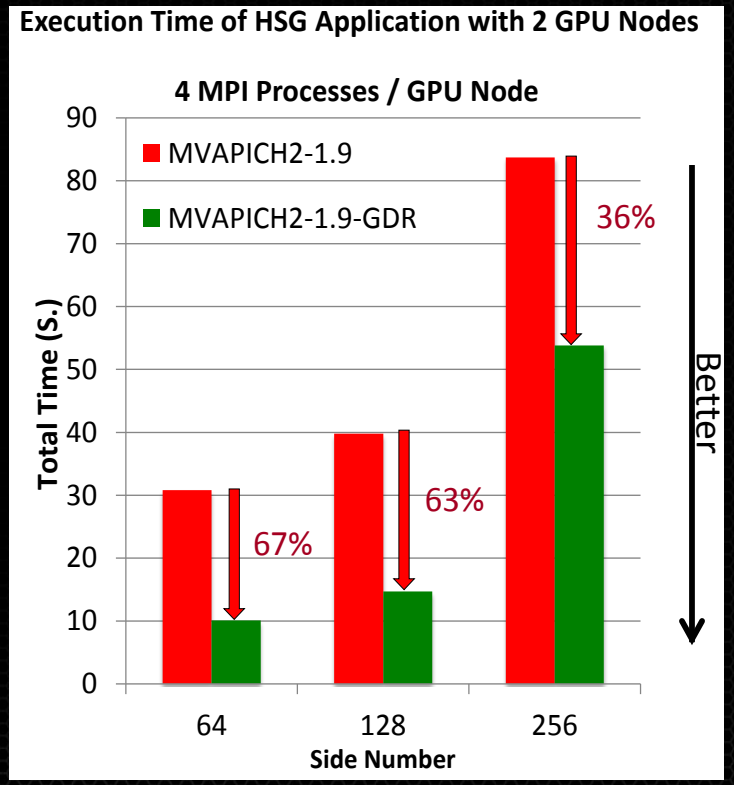
4 Developer Tools

GPUDirect RDMA in MVAPICH2 & OpenMPI

- Reduced inter-node latency



- Better MPI Application Scaling



Visit Mellanox booth #2722 for a Demo

CUDA 6

1 Unified Memory

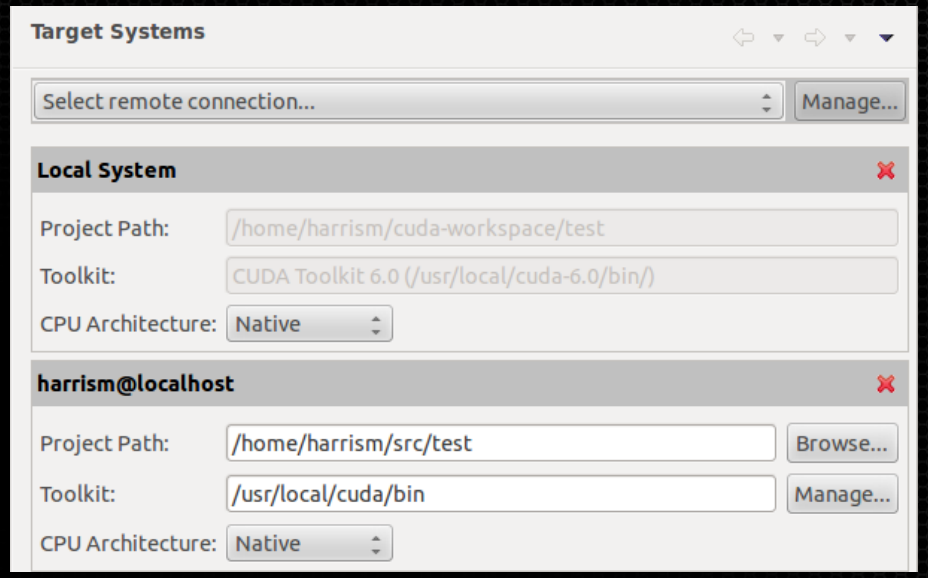
2 XT and Drop-in Libraries

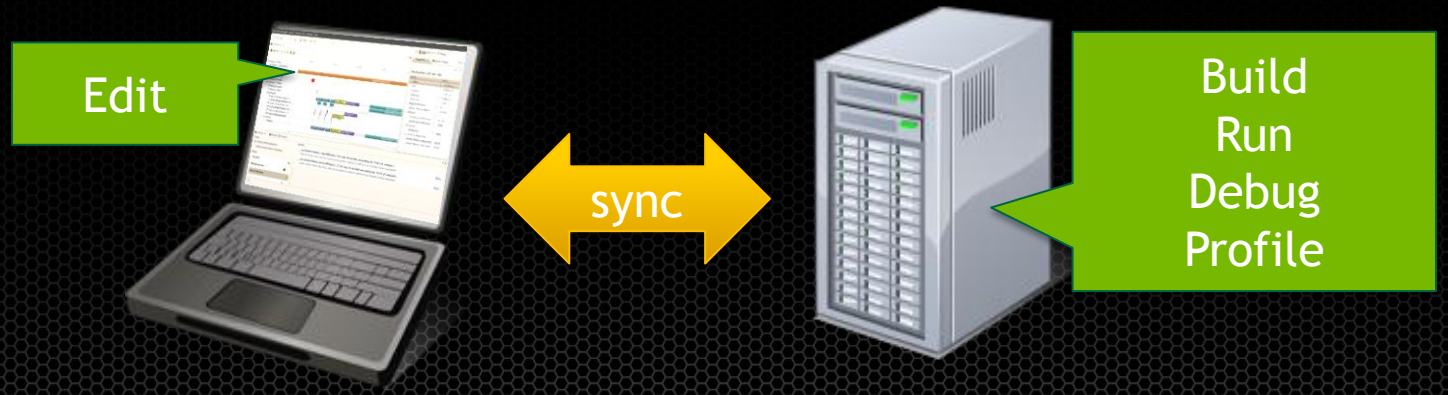
3 GPUDirect RDMA in MPI

4 Developer Tools

Remote Development with Nsight Eclipse Edition

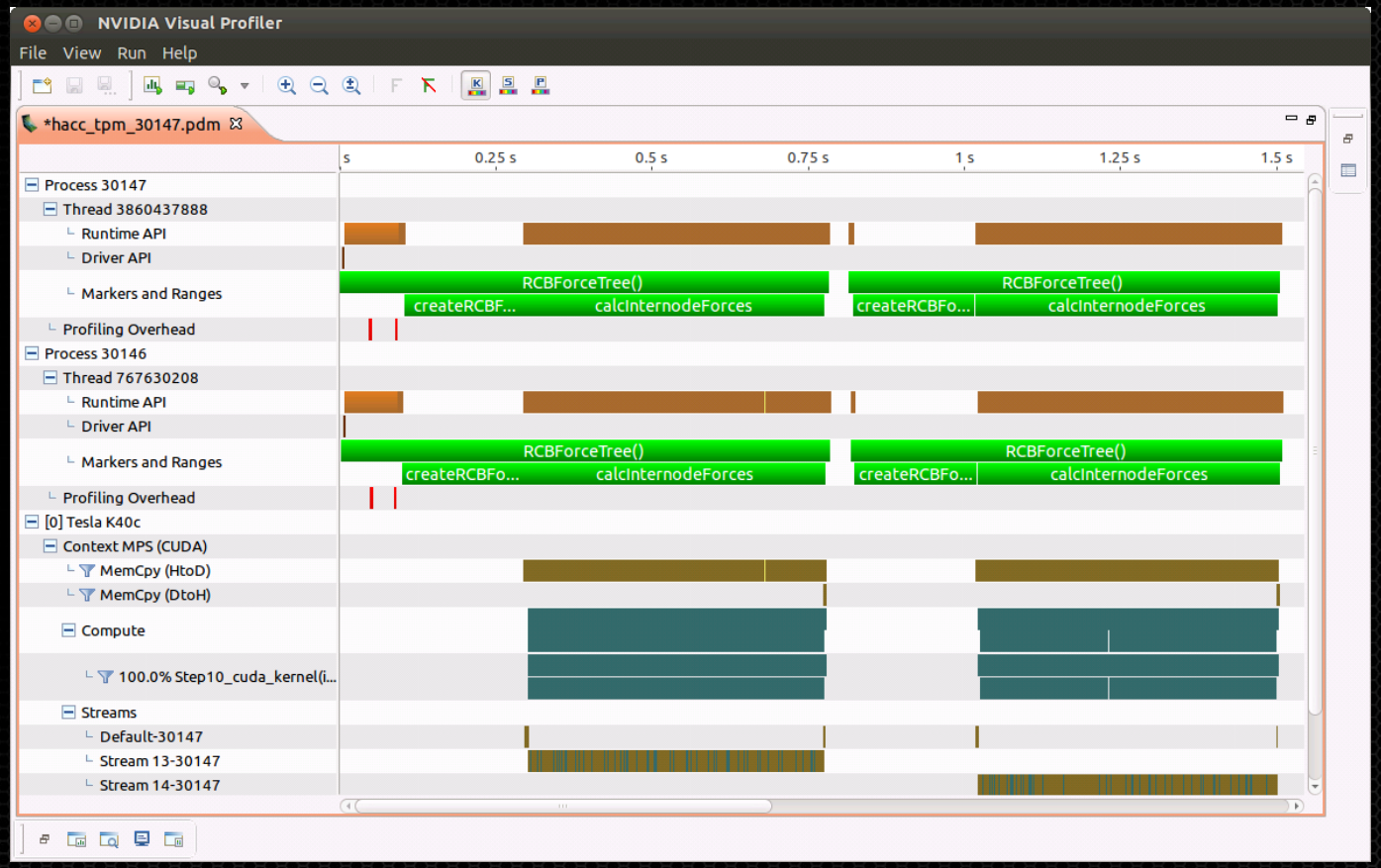
- Local IDE, remote application
  - Edit locally, build & run remotely
  - Automatic sync via ssh
  - Cross-compilation to ARM
- Full debugging & profiling via remote connection





CUDA tools for MPS (Multi-Process Server)

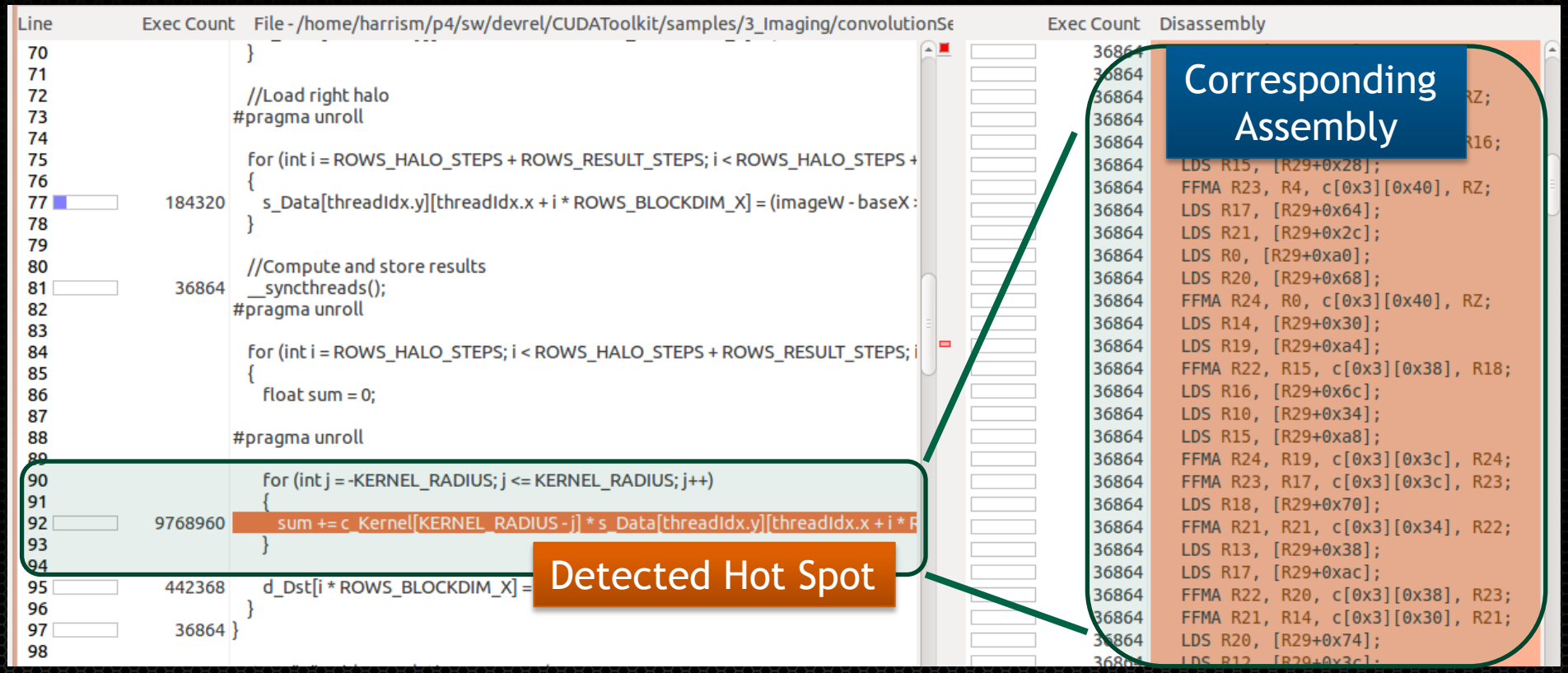
- Profile MPI apps on MPS using nvprof
- Import multi-process MPI ranks into Visual Profiler
- Run CUDA-MEMCHECK on apps running on MPS



Detailed Kernel Profiling

Visual Profiler and NSight EE

Instruction counts automatically locate hot spots in your code

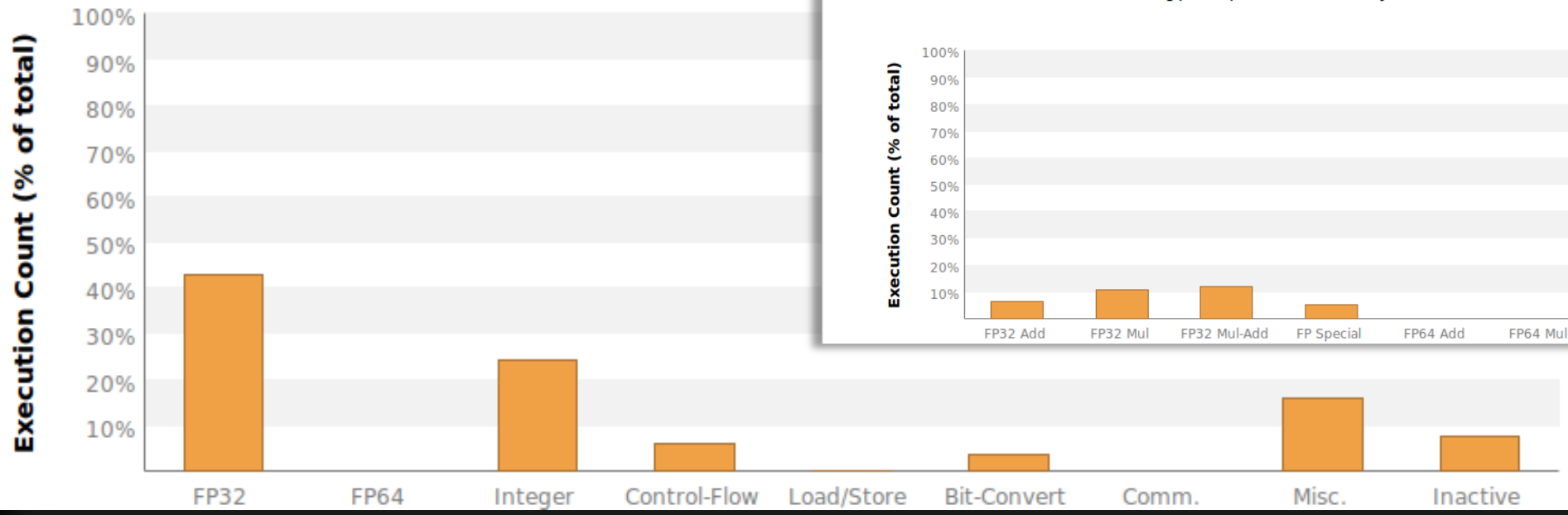


Detailed Instruction Mix Visualization

Visual Profiler and NSight EE

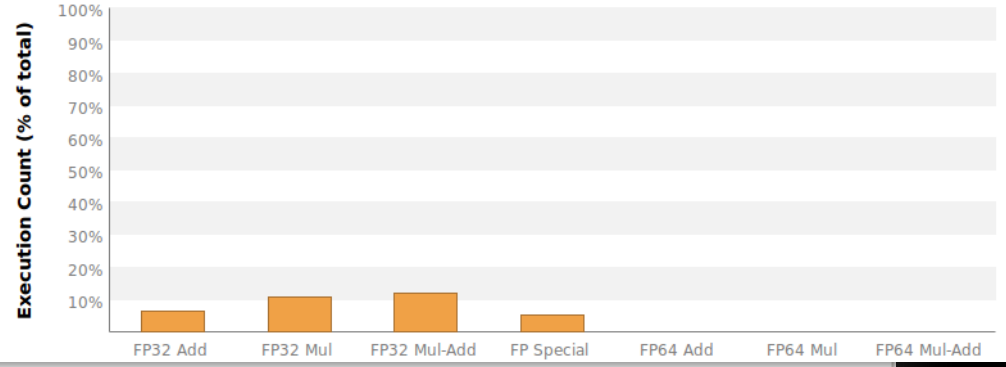
Instruction Execution Counts

The following chart shows the mix of instructions executed by the kernel. The instructions are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing instructions in that class. The "Inactive" result shows the thread executions that did not execute any instruction because the thread was predicated or inactive due to divergence.



Floating-Point Operation Counts

The following chart shows the mix of floating-point operations executed by the kernel. The operations are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing operations in that class. The results do not sum to 100% because non-floating-point operations executed by the kernel are not shown in this chart.



CUDA 6

Dramatically Simplifies Parallel Programming with Unified Memory

Sign up for CUDA Registered Developer Program

https://developer.nvidia.com/cuda-toolkit