



nvidia®

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

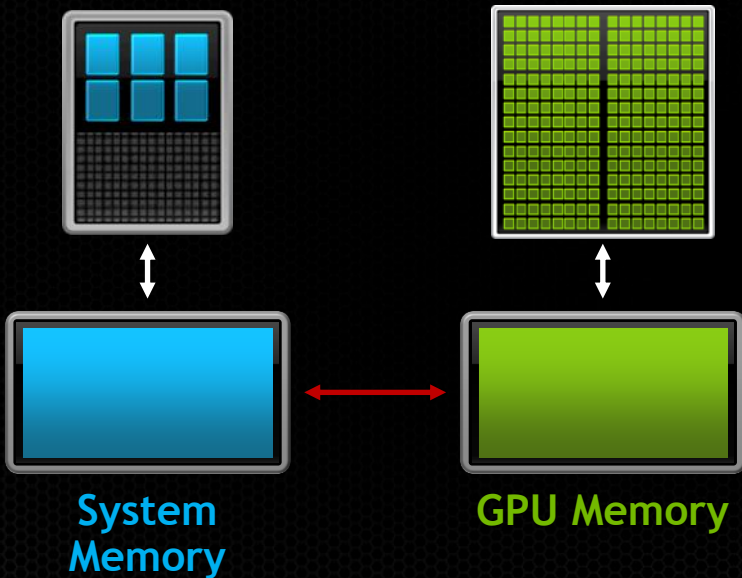
3 GPUDirect RDMA in MPI

4 Developer Tools

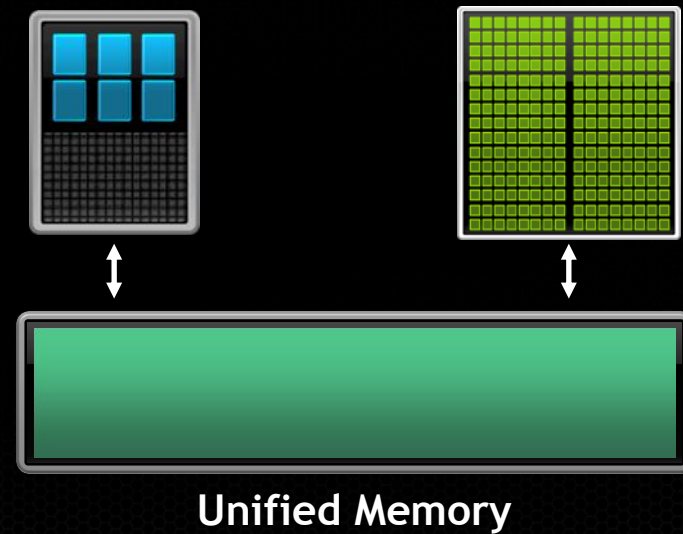
Unified Memory

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

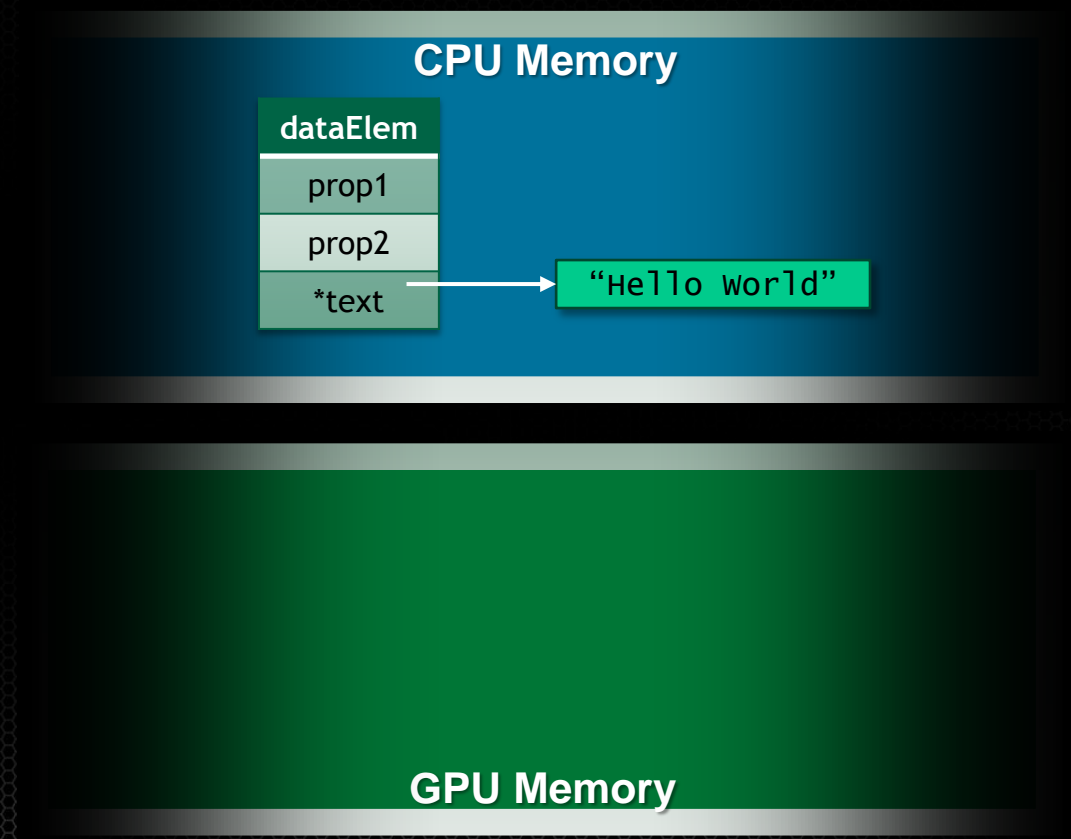
- Single pointer to data, accessible anywhere
- Tight language integration
- Greatly simplifies code porting

2. Performance Through Data Locality

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

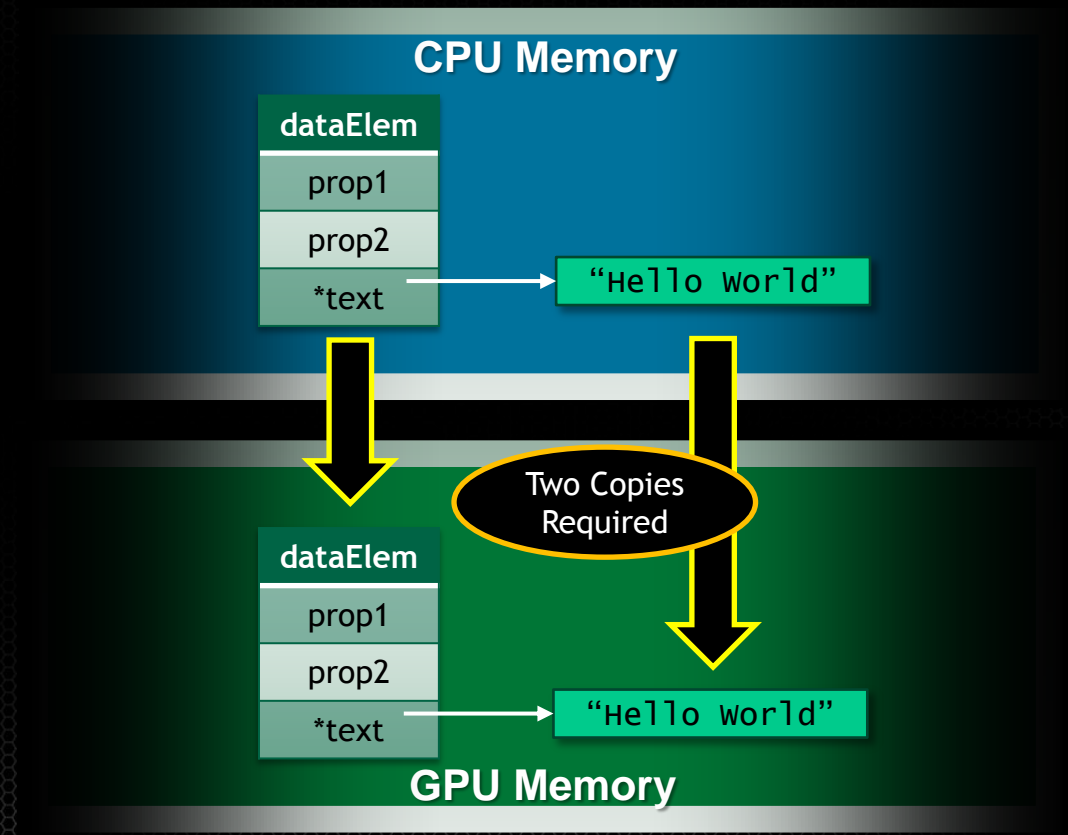
Simpler Memory Model: Eliminate Deep Copies

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



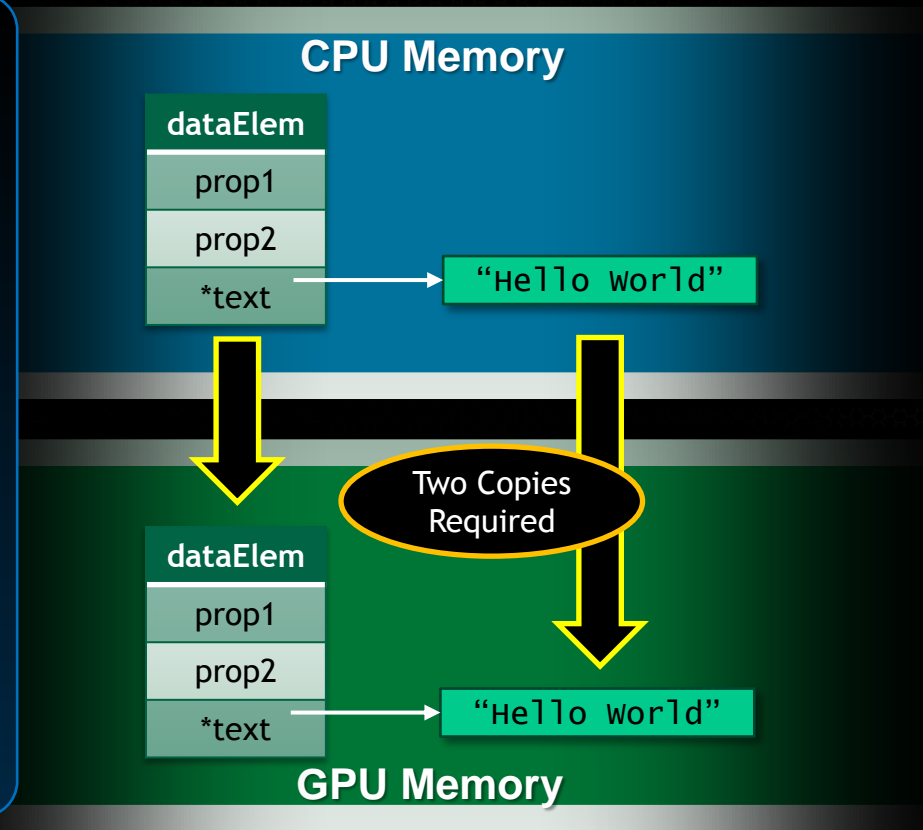
Simpler Memory Model: Eliminate Deep Copies

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



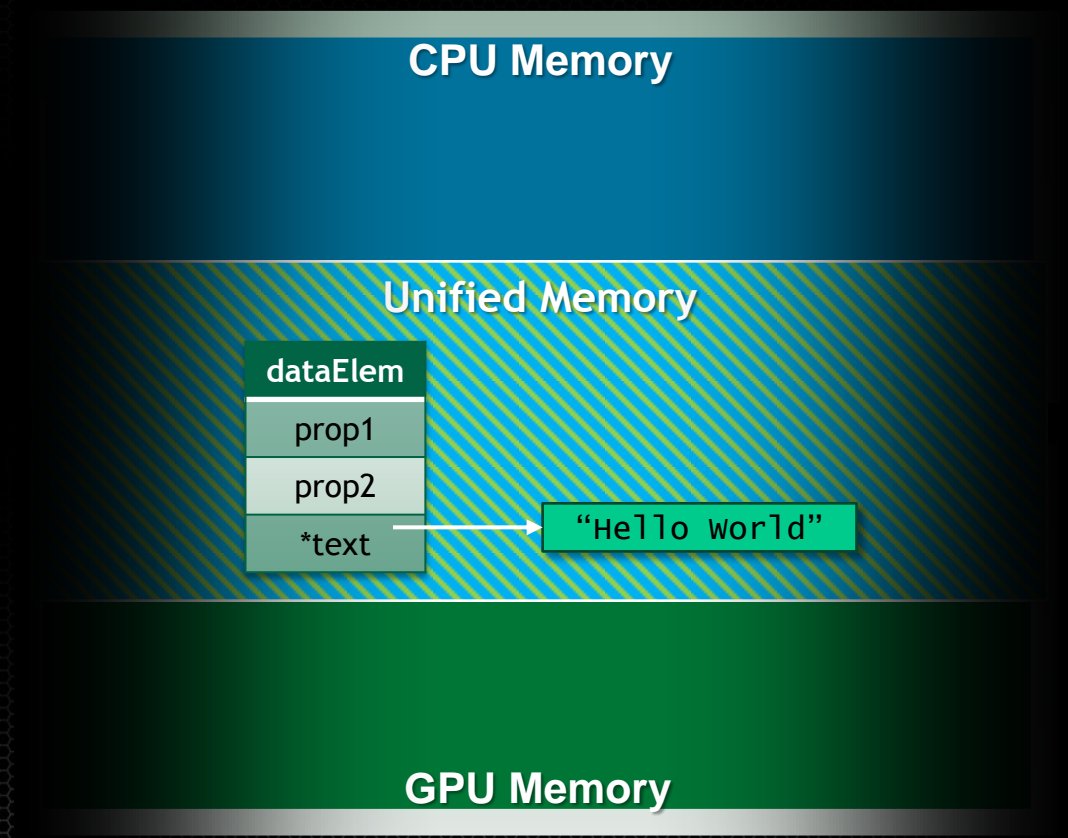
Simpler Memory Model: Eliminate Deep Copies

```
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);  
}
```



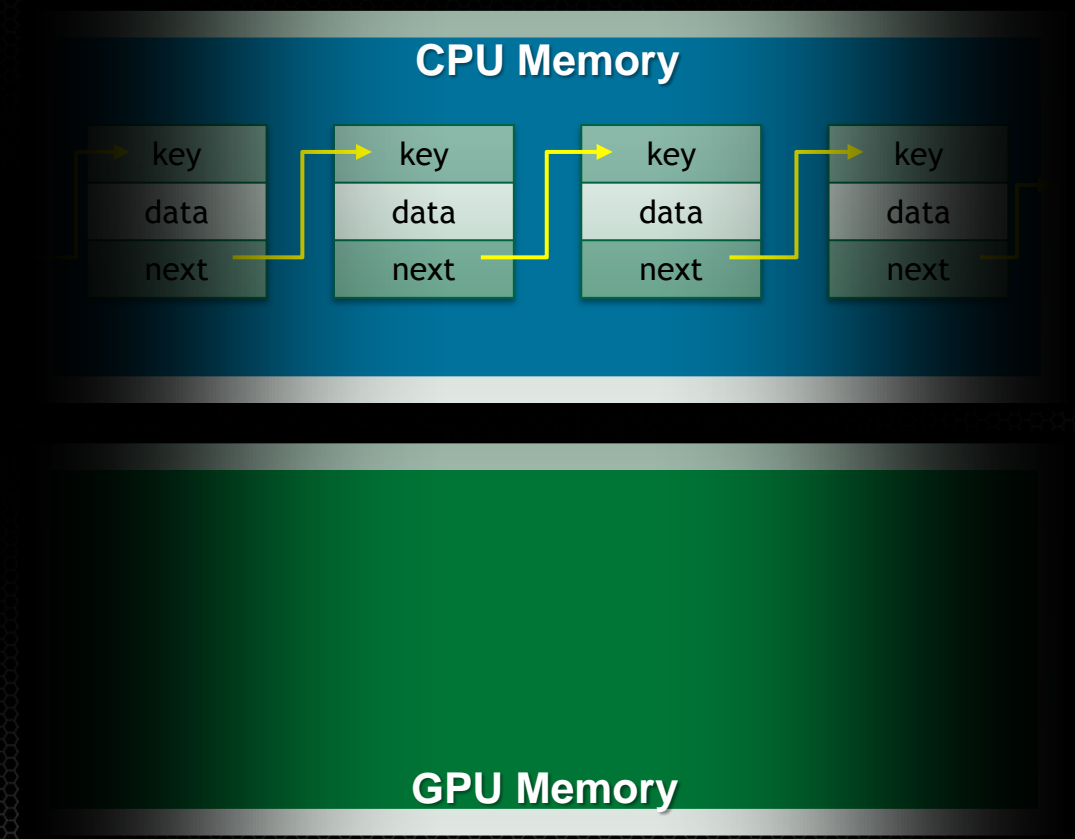
Simpler Memory Model: Eliminate Deep Copies

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



Simpler Memory Model

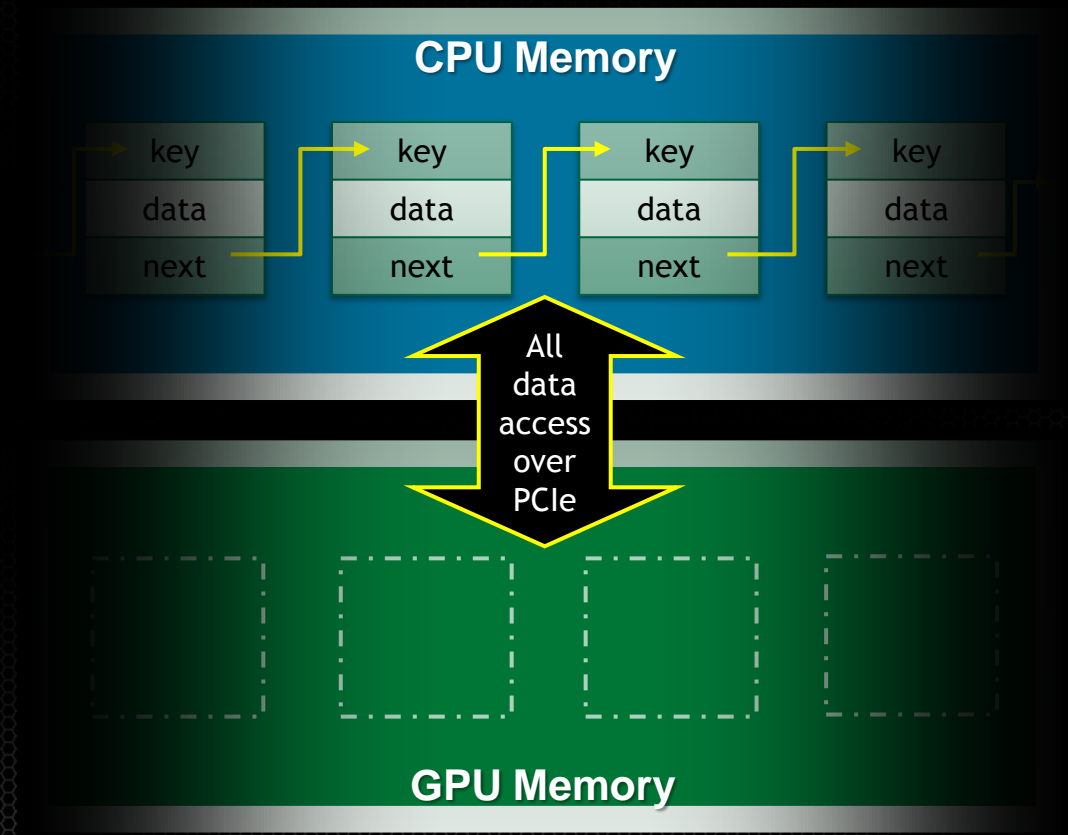
Example: GPU & CPU Shared
Linked Lists



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

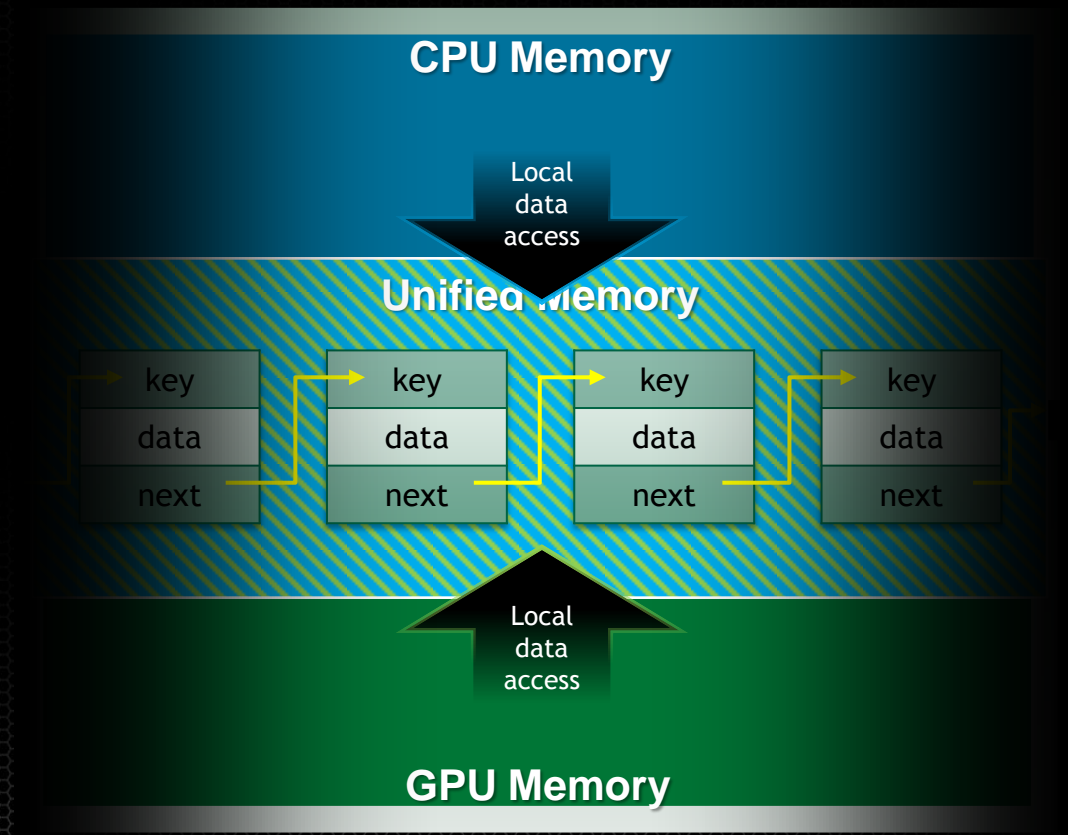


Simpler Memory Model

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++

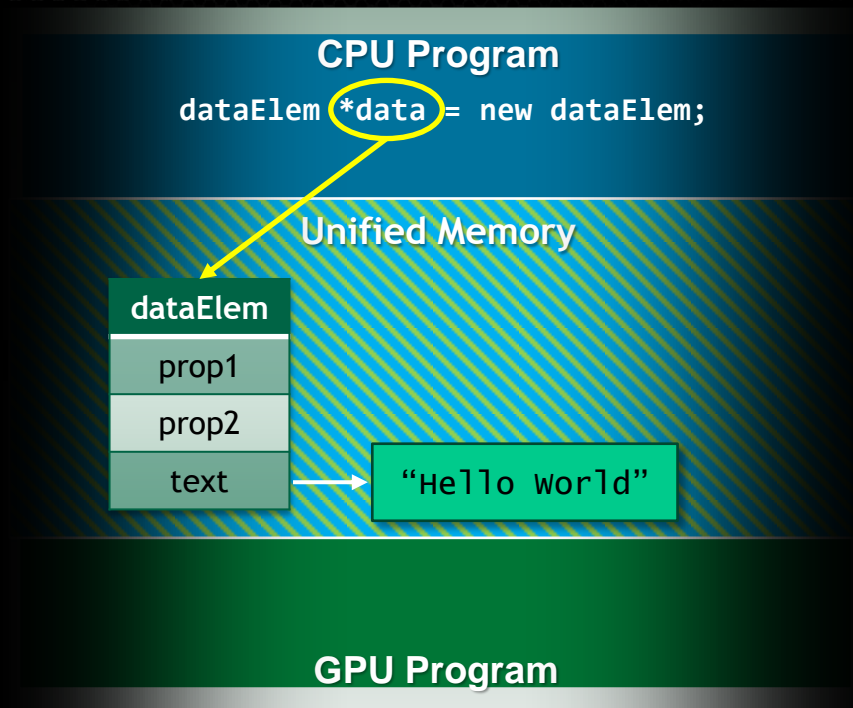
A Powerful Combination

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

CUDA 6: Ease of Use

Single Pointer to Data
No Memcopy Required
Coherence @ launch &
sync

Shared C/C++ Data
Structures

Next: Optimizations

Prefetching
Migration Hints
Additional OS Support

Maxwell

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

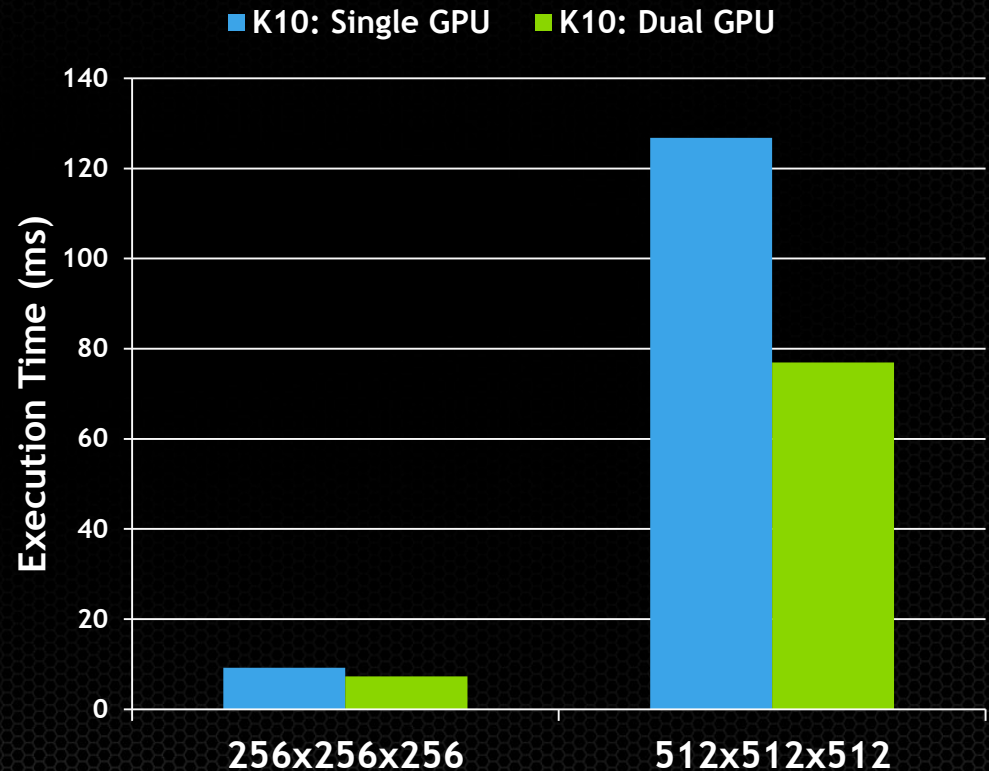
Extended (XT) Library Interfaces

- 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

cuFFT 3D Performance on 2 GPUs*

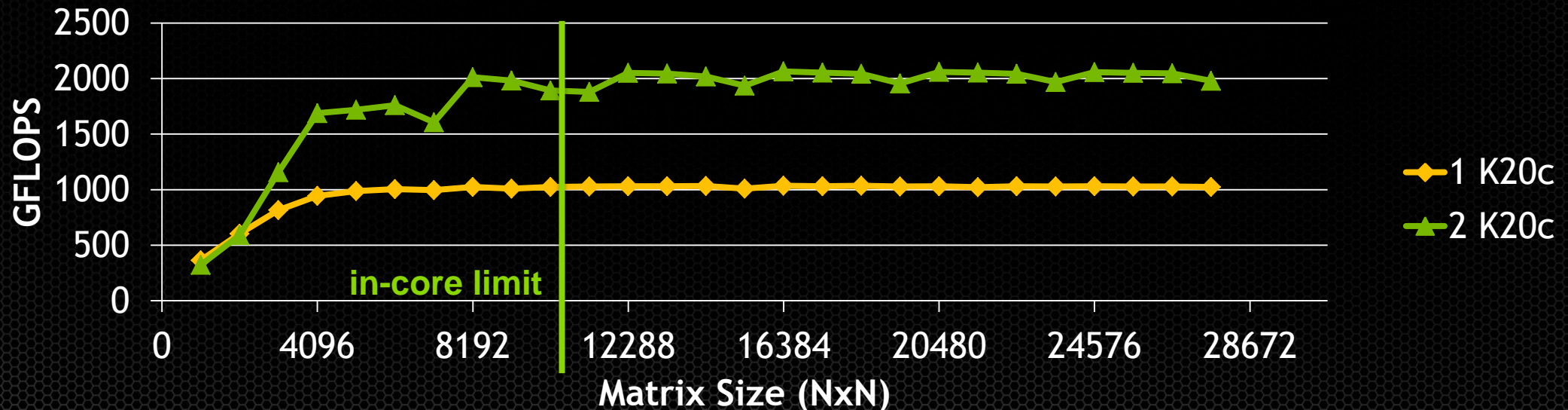


*Does not include memcpy time

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

cuBLAS ZGEMM Performance on 2 GPUS



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

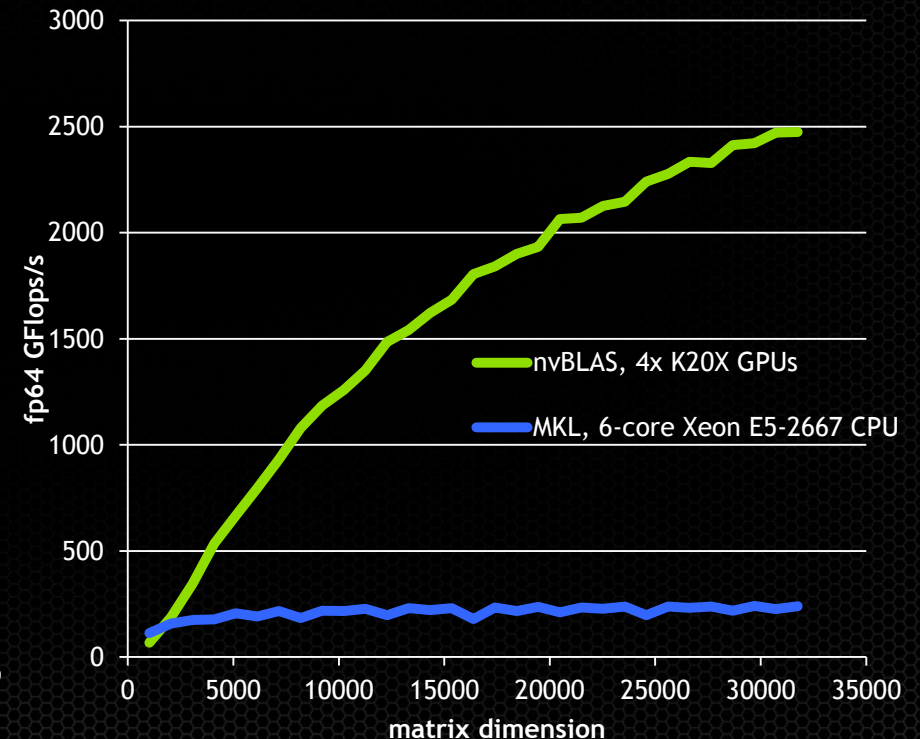
- Drop-in replacement for CPU-only BLAS
 - Automatically route BLAS3 calls to cuBLAS

- 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.

Matrix-Matrix Multiplication in R



CUDA 6

1 Unified Memory

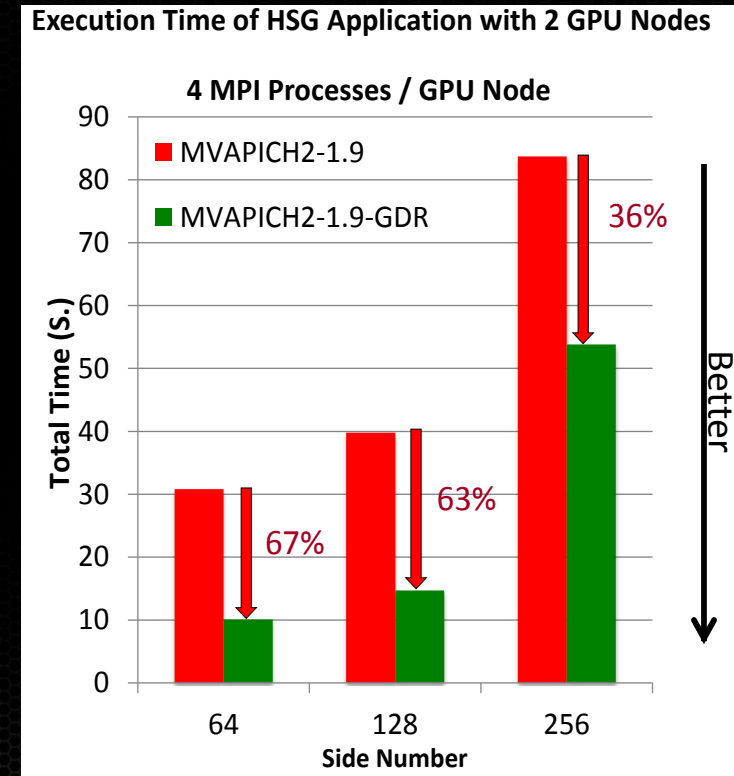
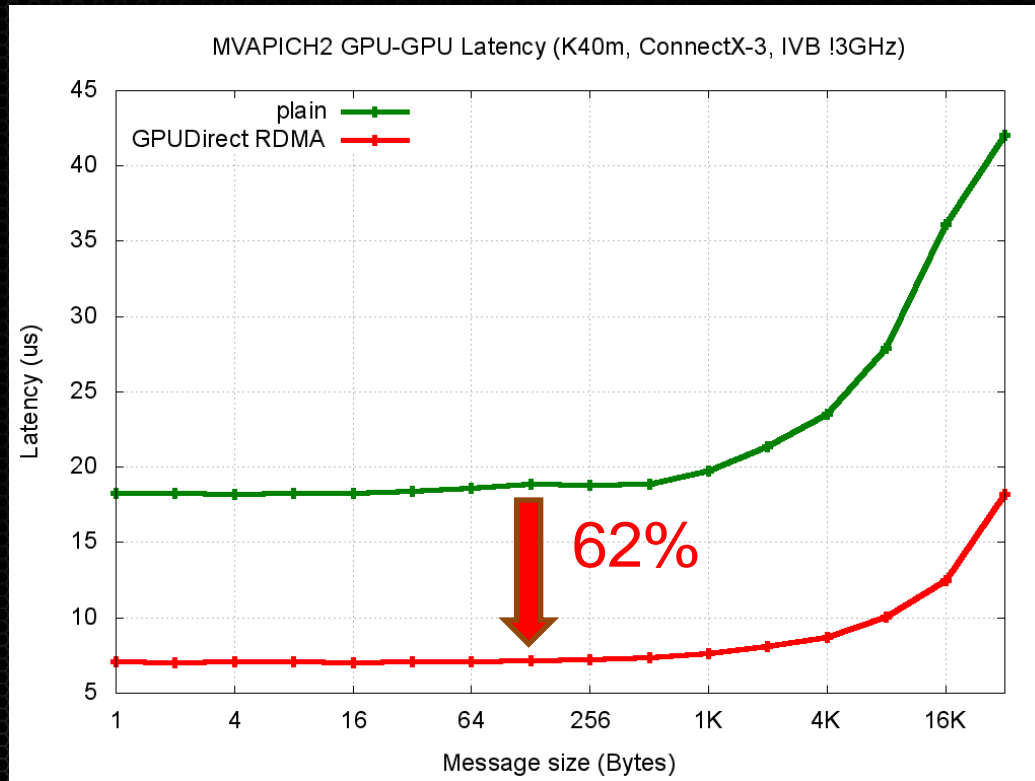
2 XT and Drop-in Libraries

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

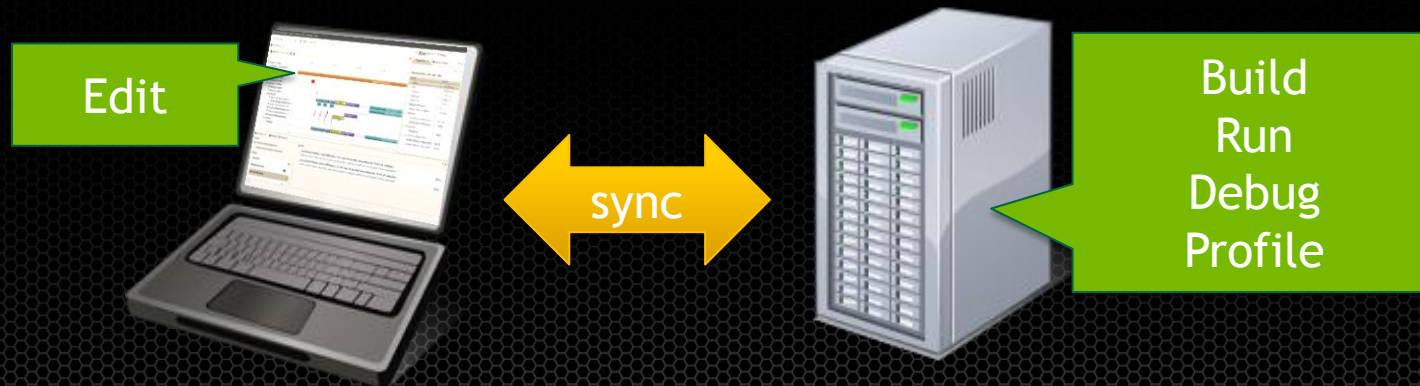
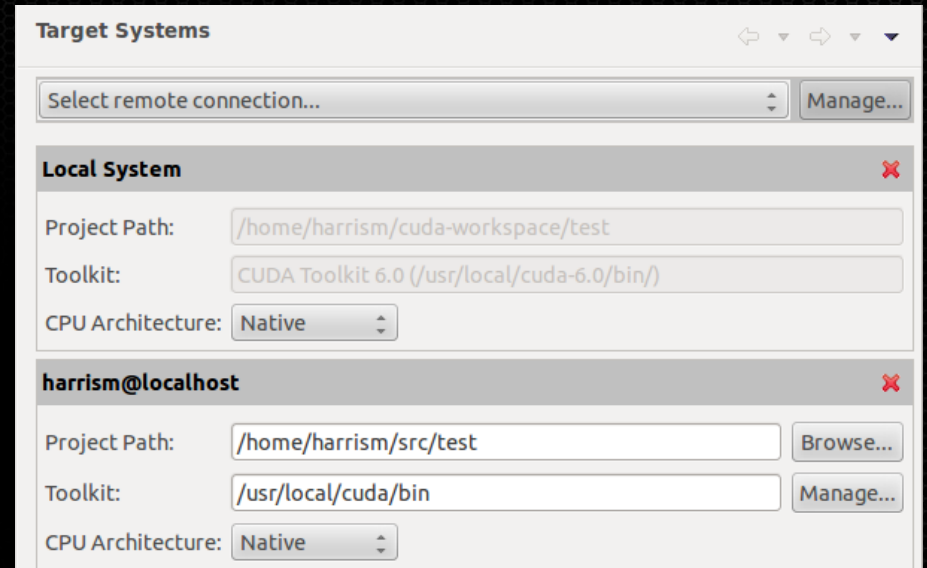
2 XT and Drop-in Libraries

3 GPUDirect RDMA in MPI

4 Developer Tools

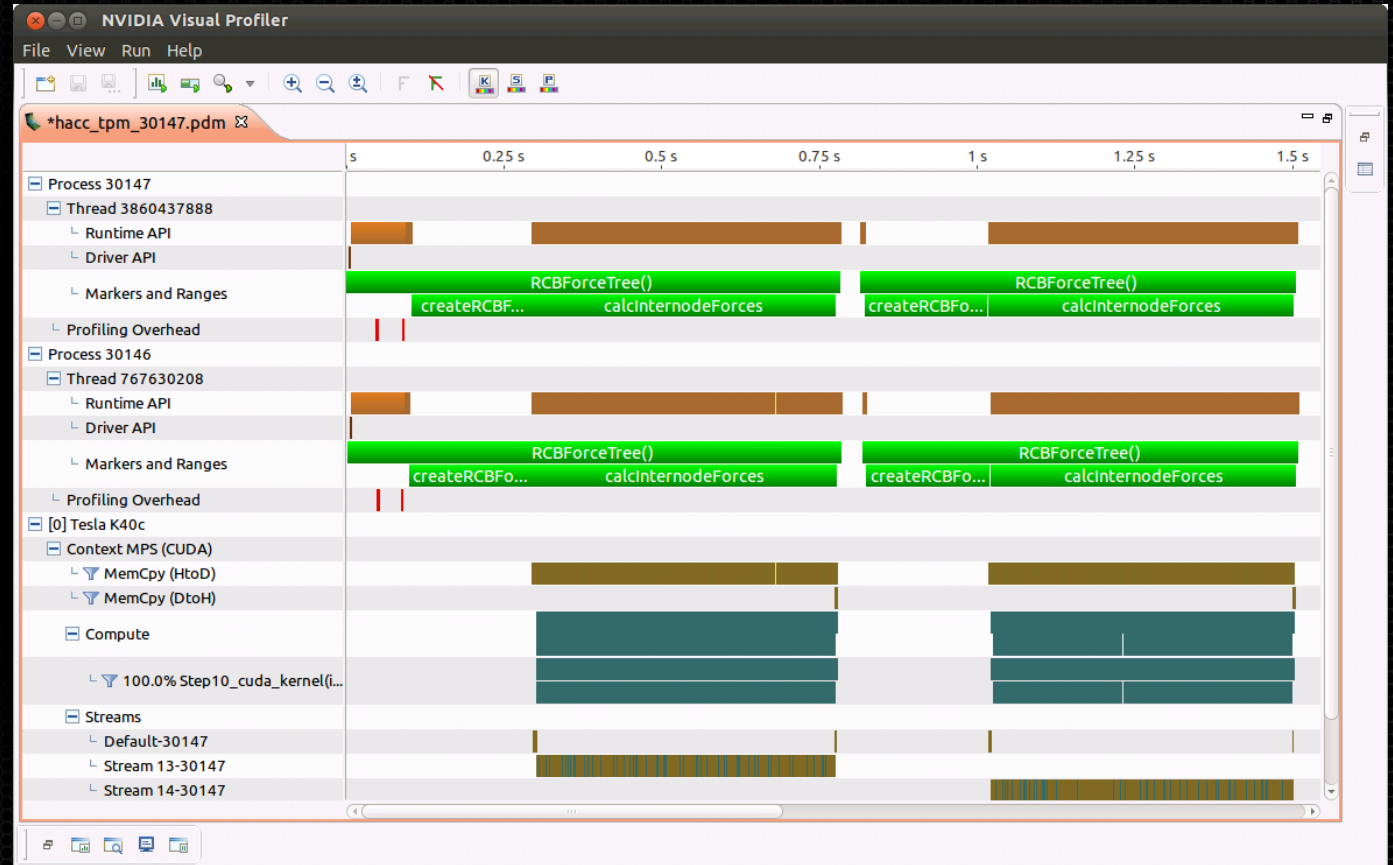
Remote Development with Nsight Eclipse Edition

- 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

The screenshot displays the Visual Profiler interface for a CUDA kernel. The main window is divided into two panes. The left pane shows the C++ source code with columns for Line, Exec Count, and File. The right pane shows the corresponding assembly code with columns for Exec Count and Disassembly.

Detected Hot Spot: A green box highlights the inner loop of the kernel, specifically the line where the sum is updated: `sum += c_Kernel[KERNEL_RADIUS - j] * s_Data[threadIdx.y][threadIdx.x + i * ROWS_BLOCKDIM_X];`. This line has an execution count of 9768960.

Corresponding Assembly: A blue box highlights the assembly code corresponding to the hot spot, showing instructions like `LDS R15, [R29+0x28];` and `FFMA R23, R4, c[0x3][0x40], RZ;`.

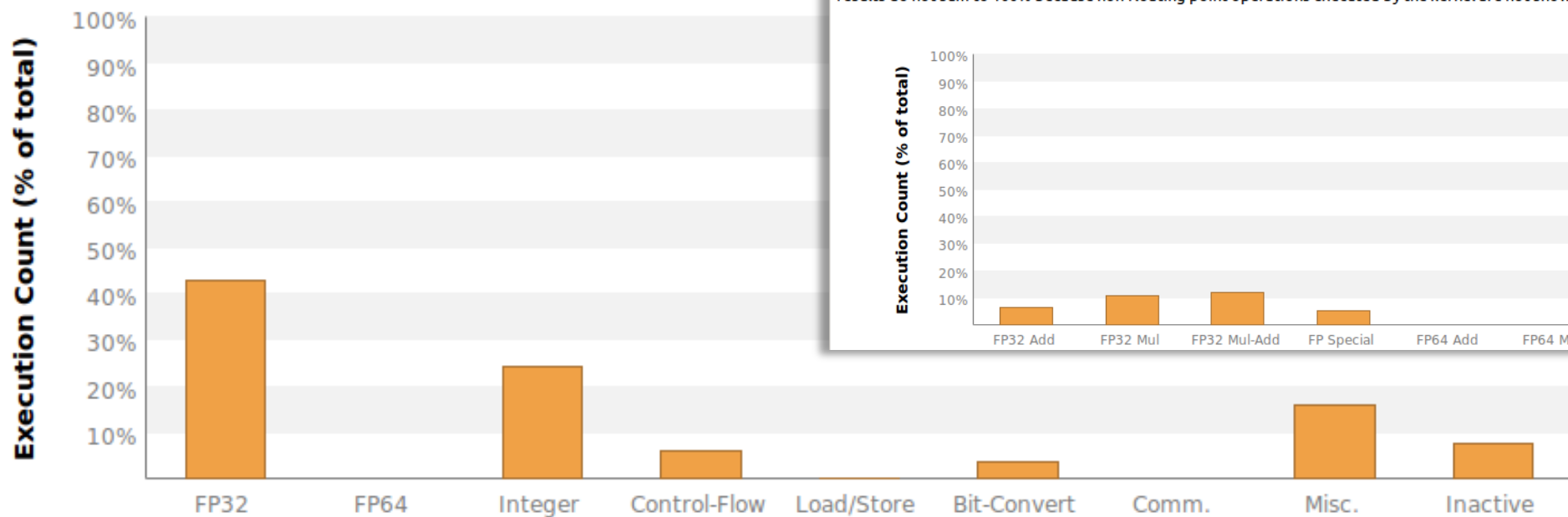
Line	Exec Count	File	Exec Count	Disassembly
70		}/td> <td>36864</td> <td></td>	36864	
71			36864	
72		//Load right halo	36864	
73		#pragma unroll	36864	
74			36864	
75		for (int i = ROWS_HALO_STEPS + ROWS_RESULT_STEPS; i < ROWS_HALO_STEPS +	36864	
76		{	36864	
77	184320	s_Data[threadIdx.y][threadIdx.x + i * ROWS_BLOCKDIM_X] = (imageW - baseX :	36864	LDS R15, [R29+0x28];
78		}	36864	FFMA R23, R4, c[0x3][0x40], RZ;
79			36864	LDS R17, [R29+0x64];
80		//Compute and store results	36864	LDS R21, [R29+0x2c];
81	36864	_syncthreads();	36864	LDS R0, [R29+0xa0];
82		#pragma unroll	36864	LDS R20, [R29+0x68];
83			36864	FFMA R24, R0, c[0x3][0x40], RZ;
84		for (int i = ROWS_HALO_STEPS; i < ROWS_HALO_STEPS + ROWS_RESULT_STEPS; i	36864	LDS R14, [R29+0x30];
85		{	36864	LDS R19, [R29+0xa4];
86		float sum = 0;	36864	FFMA R22, R15, c[0x3][0x38], R18;
87			36864	LDS R16, [R29+0x6c];
88		#pragma unroll	36864	LDS R10, [R29+0x34];
89			36864	LDS R15, [R29+0xa8];
90		for (int j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; j++)	36864	FFMA R24, R19, c[0x3][0x3c], R24;
91		{	36864	FFMA R23, R17, c[0x3][0x3c], R23;
92	9768960	sum += c_Kernel[KERNEL_RADIUS - j] * s_Data[threadIdx.y][threadIdx.x + i * R	36864	LDS R18, [R29+0x70];
93		}	36864	FFMA R21, R21, c[0x3][0x34], R22;
94			36864	LDS R13, [R29+0x38];
95	442368	d_Dst[i * ROWS_BLOCKDIM_X] =	36864	LDS R17, [R29+0xac];
96		}	36864	FFMA R22, R20, c[0x3][0x38], R23;
97	36864	}	36864	FFMA R21, R14, c[0x3][0x30], R21;
98			36864	LDS R20, [R29+0x74];

Detailed Instruction Mix Visualization

Visual Profiler and NSight EE

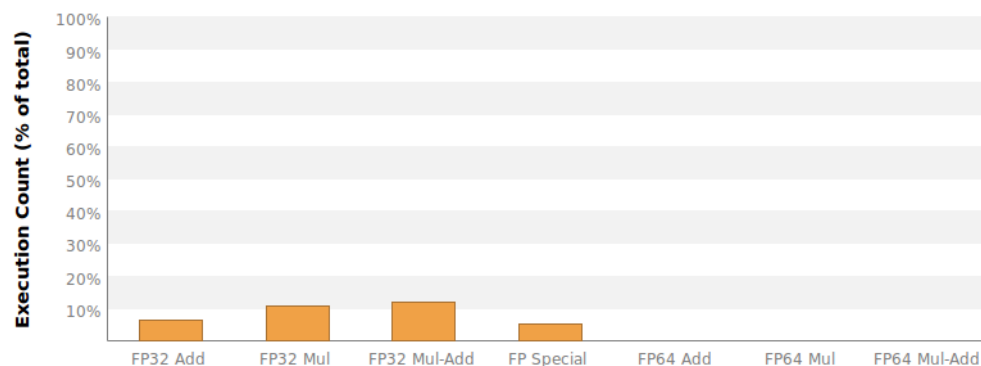
i Instruction Execution Counts

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.



i 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>