GPU TECHNOLOGY CONFERENCE

April 4-7, 2016 | Silicon Valley

CUDA 8 AND BEYOND

Mark Harris, April 5, 2016

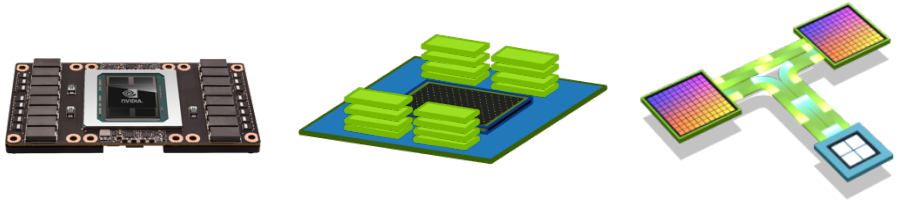
PRESENTED BY

NVIDIA.

INTRODUCING CUDA 8

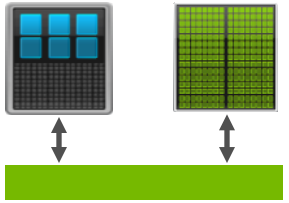
Pascal Support

New Architecture, Stacked Memory, NVLINK



Unified Memory

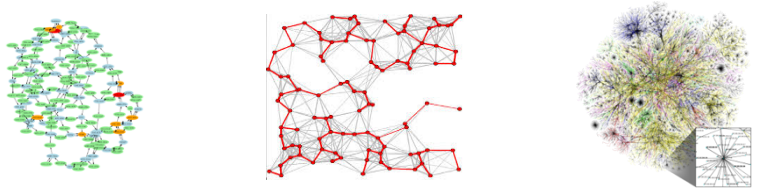
Simple Parallel Programming with large virtual memory



Libraries

nvGRAPH – library for accelerating graph analytics apps

FP16 computation to boost Deep Learning workloads

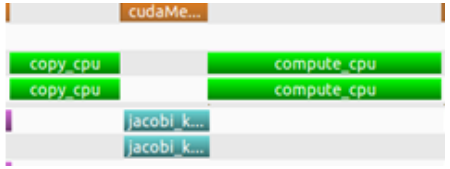


Developer Tools

Critical Path Analysis to speed overall app tuning

OpenACC profiling to optimize directive performance

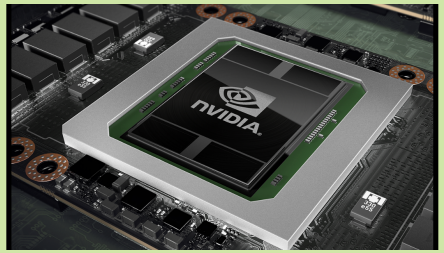
Single GPU debugging on Pascal



INTRODUCING TESLA P100

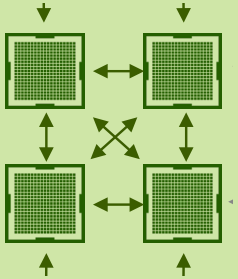
New GPU Architecture to Enable the World's Fastest Compute Node

Pascal Architecture



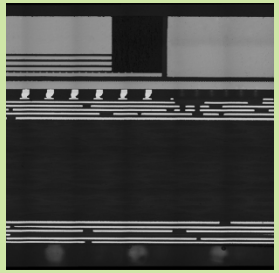
Highest Compute Performance

NVLink



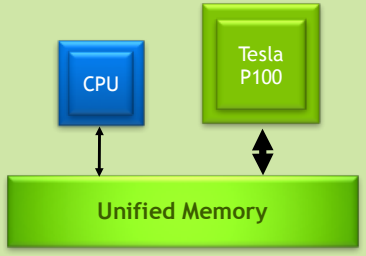
GPU Interconnect for Maximum Scalability

HBM2 Stacked Memory

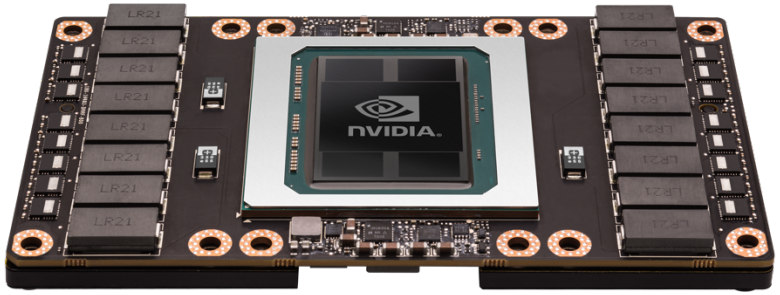


Unifying Compute & Memory in Single Package

Page Migration Engine



Simple Parallel Programming with 512 TB of Virtual Memory

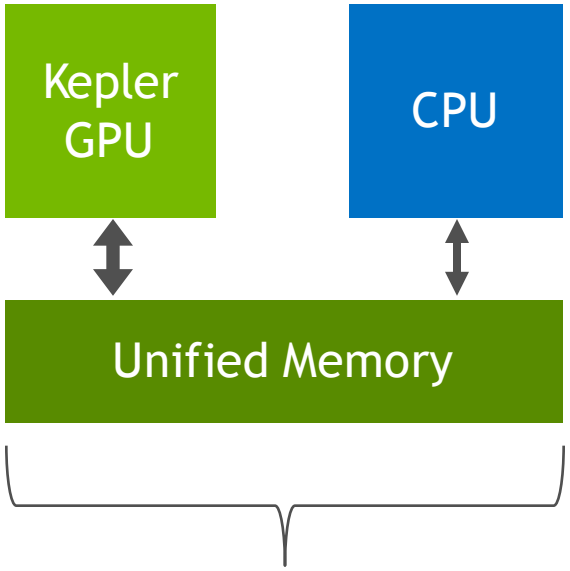


UNIFIED MEMORY

UNIFIED MEMORY

Dramatically Lower Developer Effort

CUDA 6+



Allocate Up To GPU Memory Size

Simpler Programming & Memory Model

- Single allocation, single pointer, accessible anywhere
- Eliminate need for *explicit copy*
- Greatly simplifies code porting

Performance Through Data Locality

- Migrate data to accessing processor
- Guarantee global coherence
- Still allows explicit hand tuning

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);
}
```

GREAT PERFORMANCE WITH UNIFIED MEMORY

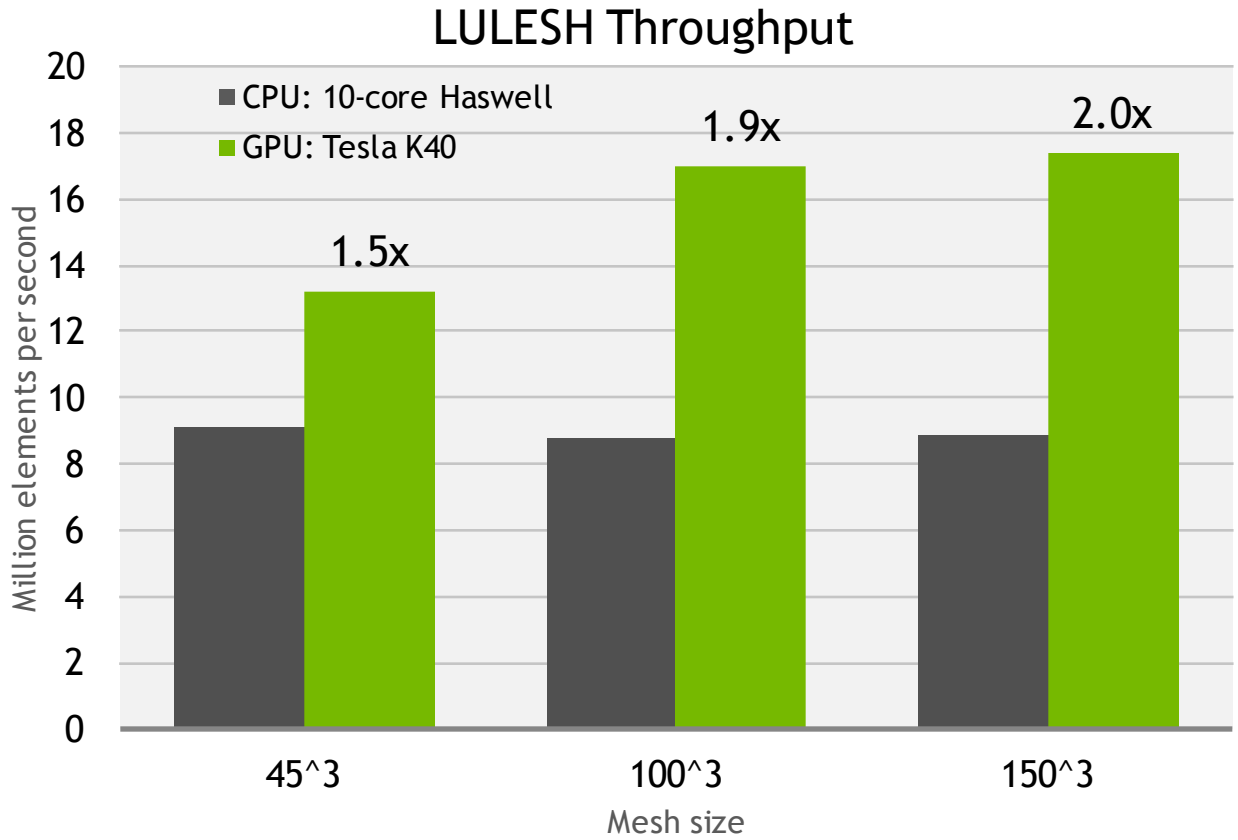
RAJA: Portable C++ Framework for parallel-for style programming

RAJA uses Unified Memory for heterogeneous array allocations

Parallel forall loops run on device

"Excellent performance considering this is a "generic" version of LULESH with no architecture-specific tuning."

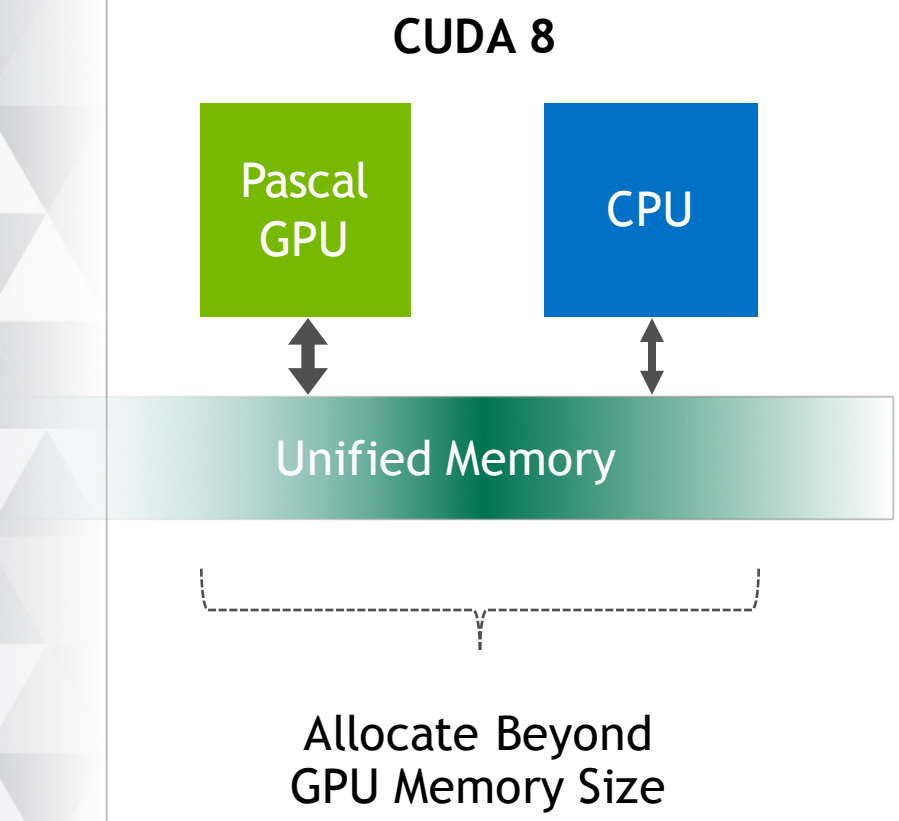
-Jeff Keasler, LLNL



GPU: NVIDIA Tesla K40, CPU: Intel Haswell E5-2650 v3 @ 2.30GHz, single socket 10-core

CUDA 8: UNIFIED MEMORY

Large datasets, simple programming, High Performance



Enable Large Data Models	Oversubscribe GPU memory Allocate up to system memory size
Simpler Data Accesss	CPU/GPU Data coherence Unified memory atomic operations
Tune Unified Memory Performance	Usage hints via cudaMemAdvise API Explicit prefetching API

UNIFIED MEMORY EXAMPLE

On-Demand Paging

```
__global__
void setValue(int *ptr, int index, int val)
{
  ptr[index] = val;
}


void foo(int size) {
  char *data;
  cudaMallocManaged(&data, size);

  memset(data, 0, size);

  setValue<<<...>>>(data, size/2, 5);
  cudaDeviceSynchronize();

  useData(data);

  cudaFree(data);
}
```

Unified Memory allocation

Access all values on CPU

Access one value on GPU

HOW UNIFIED MEMORY WORKS IN CUDA 6

Servicing CPU page faults

GPU Code

```
__global__
void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```

CPU Code

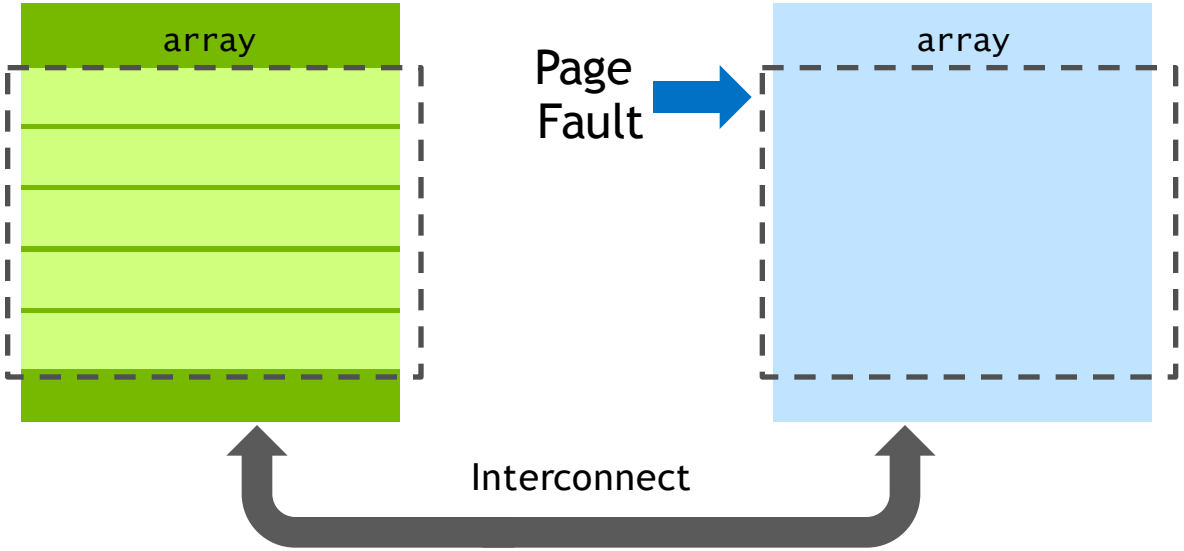
```
cudaMallocManaged(&array, size);

memset(array, size);

setValue<<<...>>>(array, size/2, 5);
```

GPU Memory Mapping

CPU Memory Mapping



HOW UNIFIED MEMORY WORKS ON PASCAL

Servicing CPU *and* GPU Page Faults

GPU Code

```
__global__
void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```

CPU Code

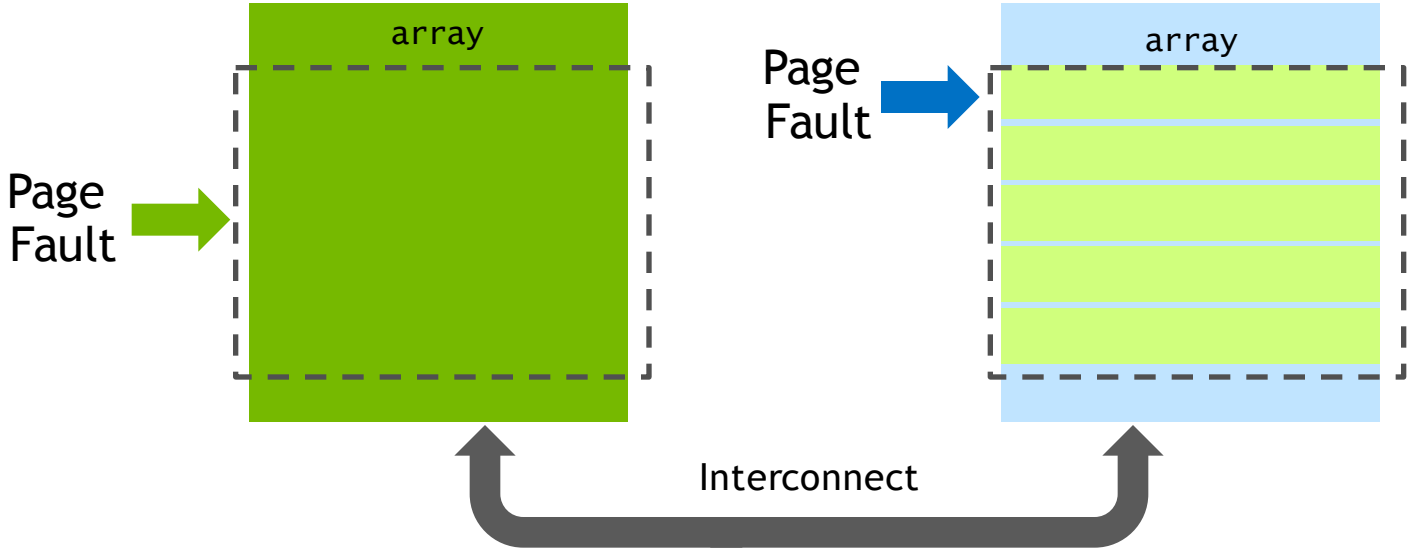
```
cudaMallocManaged(&array, size);

memset(array, size);

setValue<<<...>>>(array, size/2, 5);
```

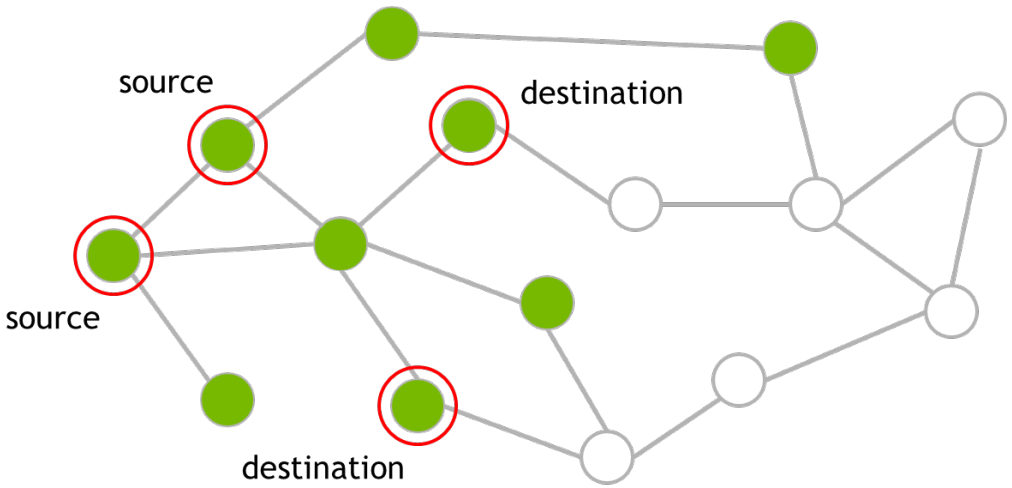
GPU Memory Mapping

CPU Memory Mapping

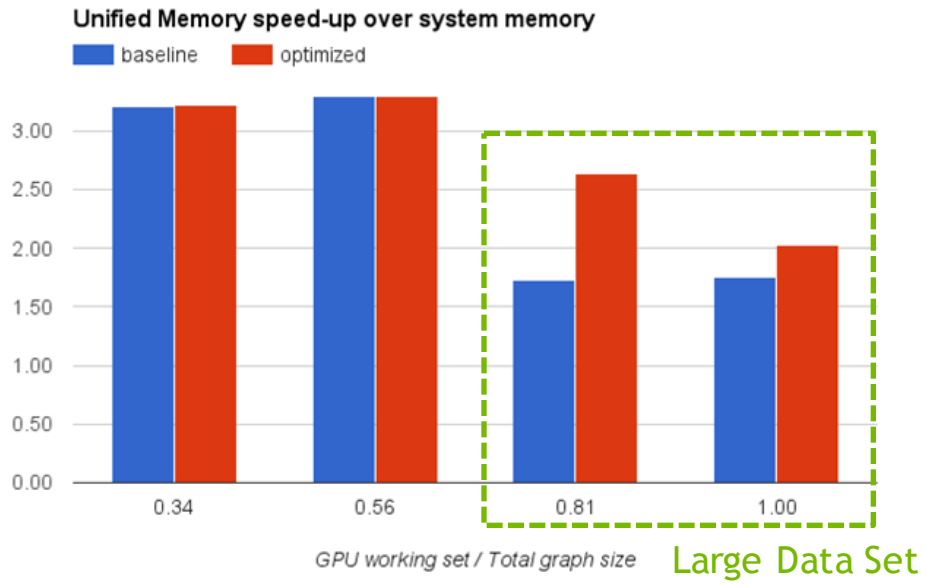


USE CASE: ON-DEMAND PAGING

Graph Algorithms



Performance over GPU directly accessing host memory (zero-copy)



Baseline: migrate on first touch
Optimized: best placement in memory

UNIFIED MEMORY ON PASCAL

GPU memory oversubscription

```
void foo() {
  // Assume GPU has 16 GB memory
  // Allocate 32 GB
  char *data;
  size_t size = 32*1024*1024*1024;
  cudaMallocManaged(&data, size);
}
```

32 GB allocation

Pascal supports allocations where only a subset of pages reside on GPU. Pages can be migrated to the GPU when “hot”.

Fails on Kepler/Maxwell

GPU OVERSUBSCRIPTION

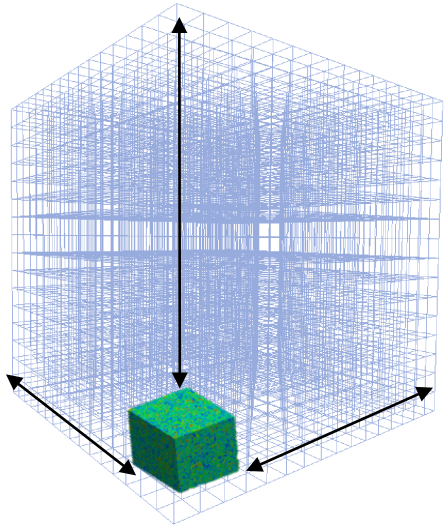
Now possible with Pascal

Many domains would benefit from GPU memory oversubscription:

Combustion – many species to solve for

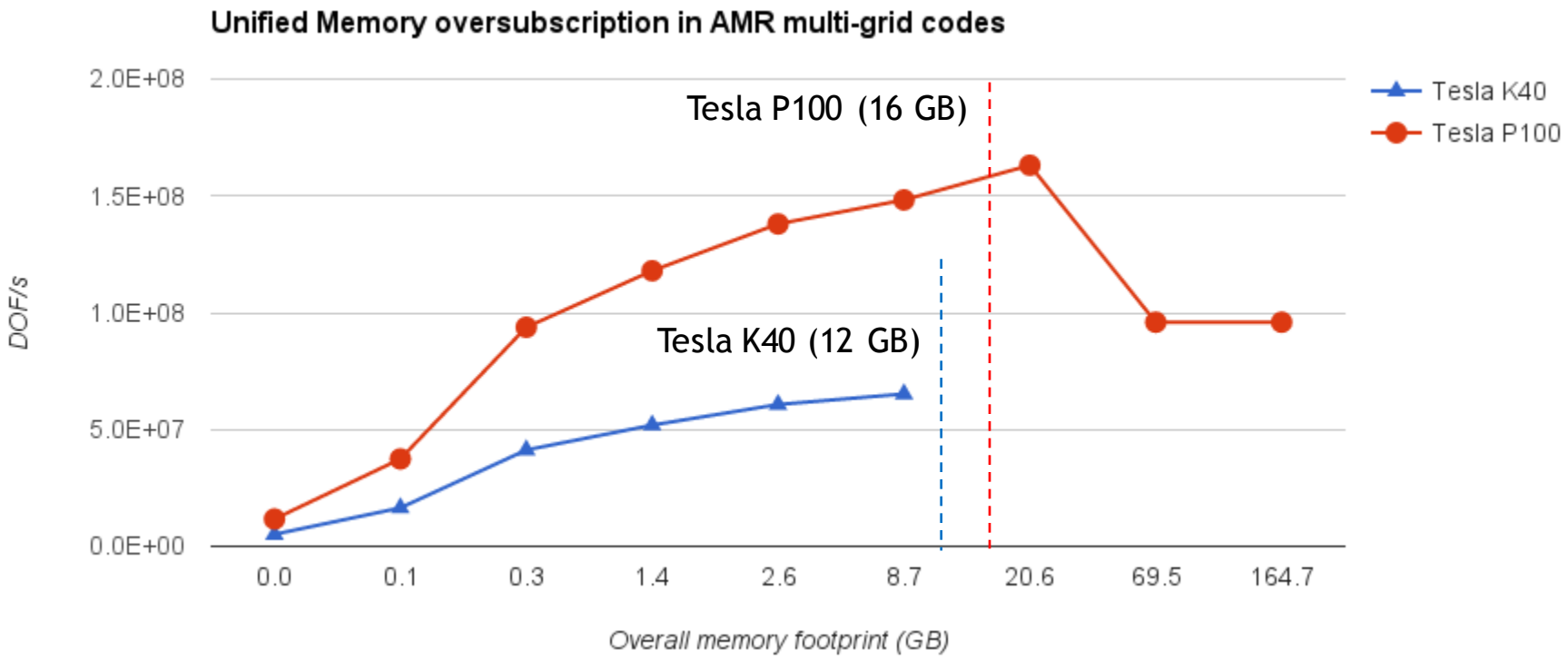
Quantum chemistry – larger systems

Ray tracing - larger scenes to render



GPU OVERSUBSCRIPTION

HPGMG: high-performance multi-grid



*Tesla P100 performance is very early modelling results

UNIFIED MEMORY ON PASCAL

Concurrent CPU/GPU access to managed memory

```
__global__ void mykernel(char *data) {
  data[1] = 'g';
}

void foo() {
  char *data;
  cudaMallocManaged(&data, 2);

  mykernel<<<...>>>(data);
  // no synchronize here
  data[0] = 'c';

  cudaFree(data);
}
```

OK on Pascal: just a page fault

Concurrent CPU access to 'data' on previous GPUs caused a fatal segmentation fault

UNIFIED MEMORY ON PASCAL

System-Wide Atomics

```
__global__ void mykernel(int *addr) {
  atomicAdd(addr, 10);
}

void foo() {
  int *addr;
  cudaMallocManaged(addr, 4);
  *addr = 0;

  mykernel<<<...>>>(addr);
  __sync_fetch_and_add(addr, 10);
}
```

Pascal enables system-wide atomics

- Direct support of atomics over NVLink
- Software-assisted over PCIe

System-wide atomics not available on Kepler / Maxwell

PERFORMANCE TUNING ON PASCAL

Explicit Memory Hints and Prefetching

Advise runtime on known memory access behaviors with `cudaMemAdvise()`

- `cudaMemAdviseSetReadMostly`: Specify read duplication
- `cudaMemAdviseSetPreferredLocation`: suggest best location
- `cudaMemAdviseSetAccessedBy`: initialize a mapping

Explicit prefetching with `cudaMemPrefetchAsync(ptr, length, destDevice, stream)`

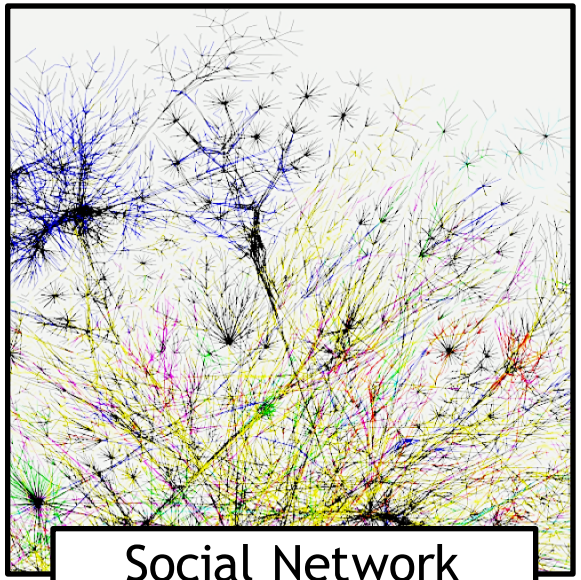
- Unified Memory alternative to cudaMemcpyAsync
- Asynchronous operation that follows CUDA stream semantics

To Learn More:
S6216 "The Future of Unified Memory" by Nikolay Sakharnykh
Tuesday, 4pm

GRAPH ANALYTICS

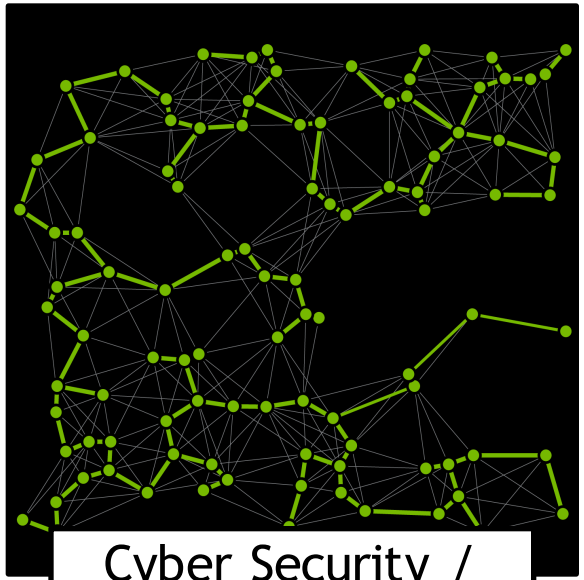
GRAPH ANALYTICS

Insight from Connections in Big Data

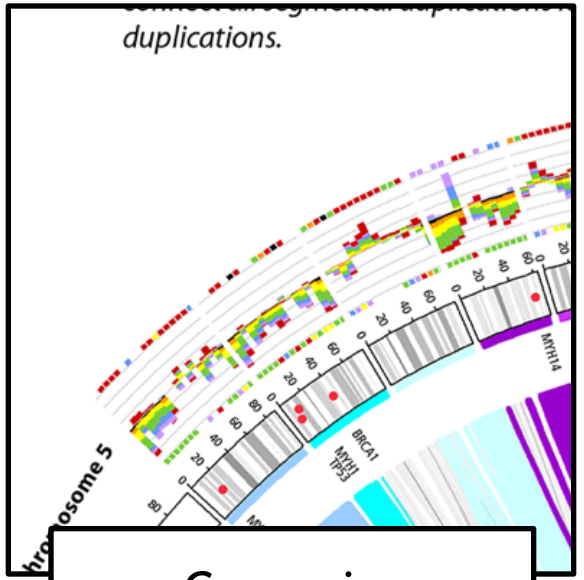


Wikimedia Commons

Social Network Analysis



Cyber Security / Network Analytics



Circos.ca

Genomics

… and much more: Parallel Computing, Recommender Systems, Fraud Detection, Voice Recognition, Text Understanding, Search

nvGRAPH

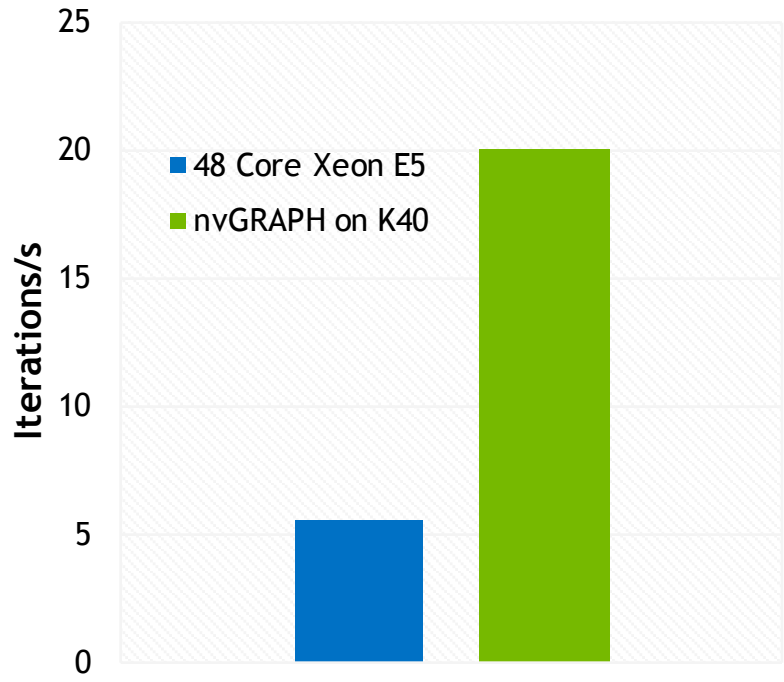
Accelerated Graph Analytics

Process graphs with up to 2.5 Billion edges on a single GPU (24GB M40)

Accelerate a wide range of applications:

PageRank	Single Source Shortest Path	Single Source Widest Path
Search	Robotic Path Planning	IP Routing
Recommendation Engines	Power Network Planning	Chip Design / EDA
Social Ad Placement	Logistics & Supply Chain Planning	Traffic sensitive routing

nvGRAPH: 4x Speedup



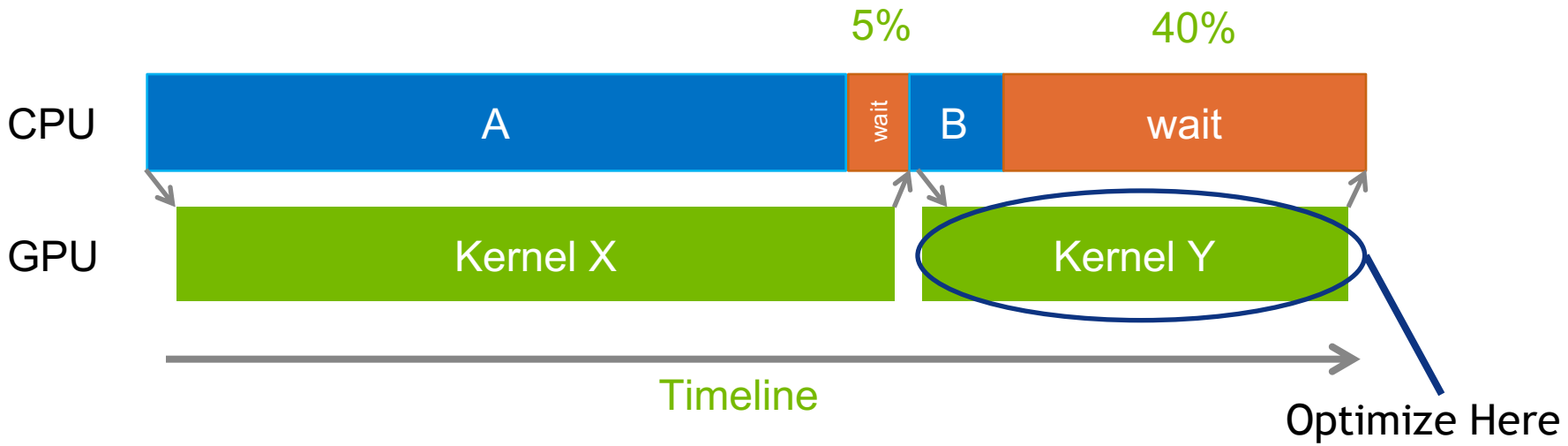
PageRank on Wikipedia 84 M link dataset

developer.nvidia.com/nvgraph

ENHANCED PROFILING

DEPENDENCY ANALYSIS

Easily Find the Critical Kernel To Optimize



The longest running kernel is not always the most critical optimization target

DEPENDENCY ANALYSIS

Visual Profiler

Unguided Analysis

Generating critical path

Analysis | GPU Details | Console | Settings

Reset All | Analyze All

To enable kernel analysis stages select a host-launched kernel instance in the timeline.

Application

- Data Movement And Concurrency
- Compute Utilization
- Kernel Performance
- Dependency Analysis
- NVLink

Results

Dependency Analysis

The following table shows metrics collected from a dependency analysis of the program execution. The data is summarized per function type. Use the "Dependency Analysis" menu on the main toolbar to visualize analysis results on the timeline. More...

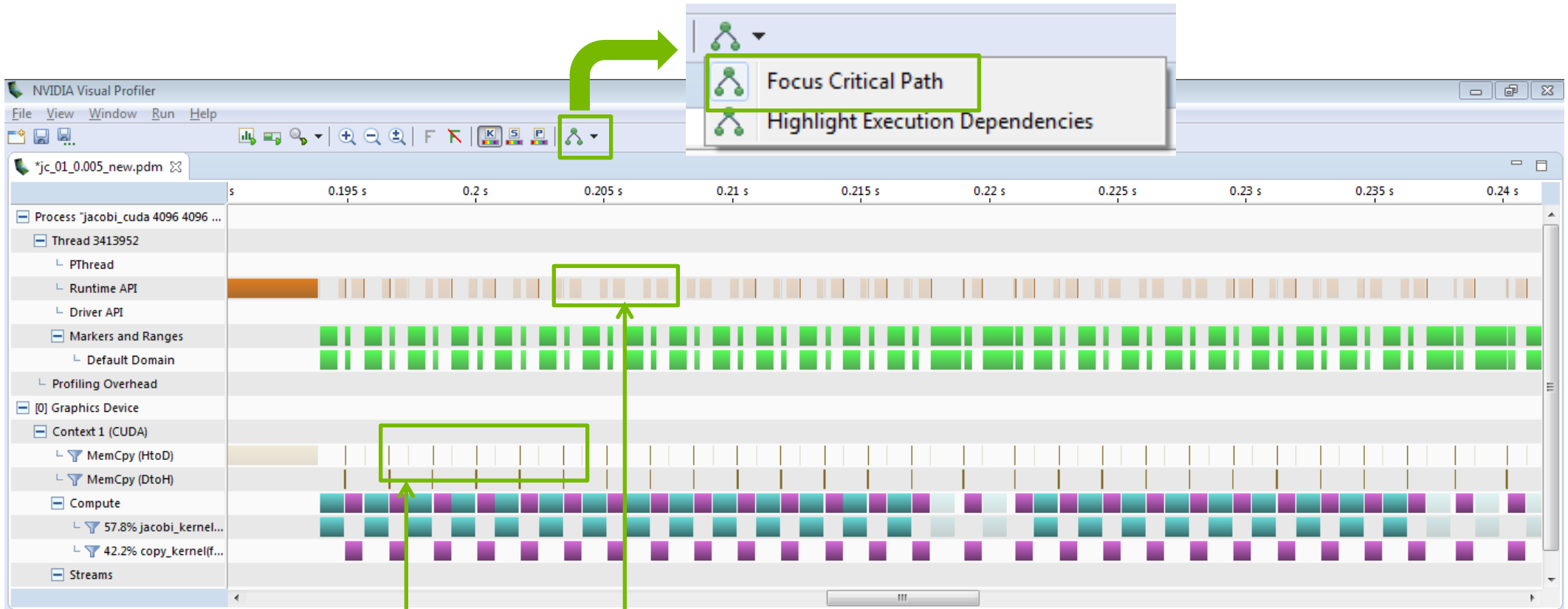
Function Name	Time on Critical Path (%)	Time on Critical Path	Waiting time
cudaMalloc	32.72 %	127.392 ms	0 ns
jacobi_kernel(float const *, float*, int, int, float*)	20.61 %	80.248 ms	0 ns
copy_kernel(float*, float const *, int, int)	17.46 %	68.004 ms	0 ns
<Other>	12.61 %	49.113 ms	0 ns
cudaMemcpy	10.75 %	41.844 ms	20.181 ms
[CUDA memcpy DtoH]	5.18 %	20.181 ms	0 ns
cudaSetupArgument	0.14 %	534.684 µs	0 ns
cudaFree	0.11 %	424.883 µs	0 ns
[CUDA memcpy HtoD]	0.10 %	400.25 µs	0 ns
cuDeviceGetAttribute	0.09 %	336.781 µs	0 ns
cudaGetDeviceProperties	0.08 %	319.677 µs	0 ns
cudaLaunch	0.05 %	192.598 µs	0 ns
cudaConfigureCall	0.05 %	186.452 µs	0 ns
cuDeviceTotalMem_v2	0.05 %	182.833 µs	0 ns
cuDeviceGetName	0.00 %	18.022 µs	0 ns
cudaSetDevice	0.00 %	12.933 µs	0 ns

Dependency Analysis

Functions on critical path

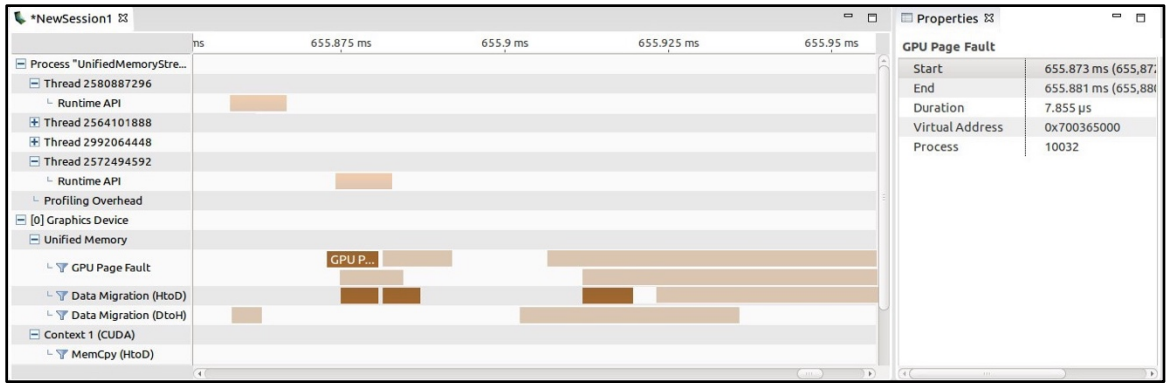
DEPENDENCY ANALYSIS

Visual Profiler

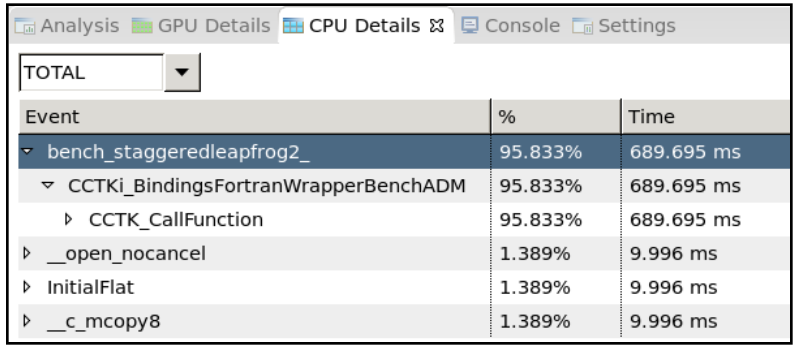


APIs, GPU activities not in critical path are greyed out

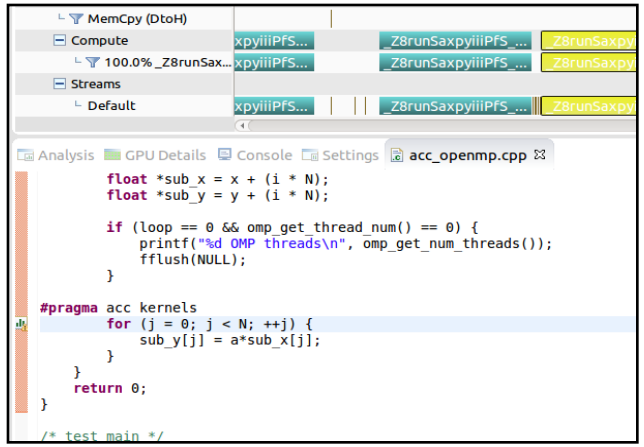
MORE CUDA 8 PROFILER FEATURES



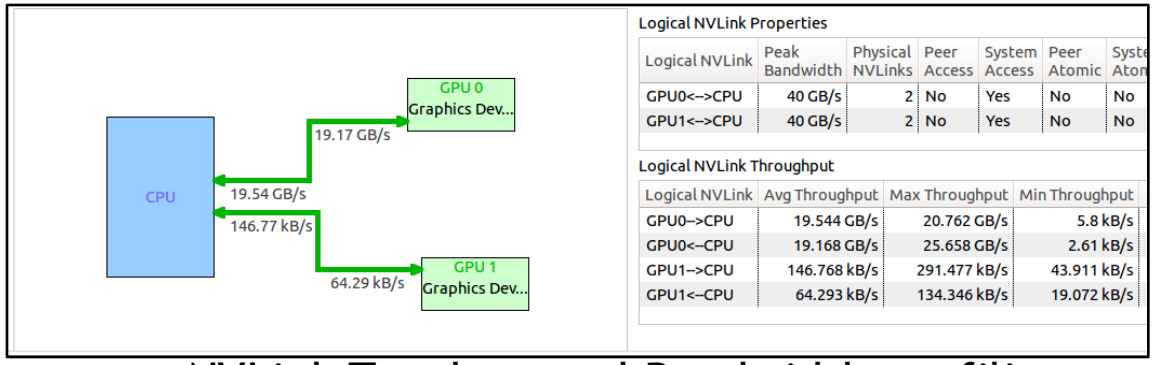
Unified Memory Profiling



CPU Profiling



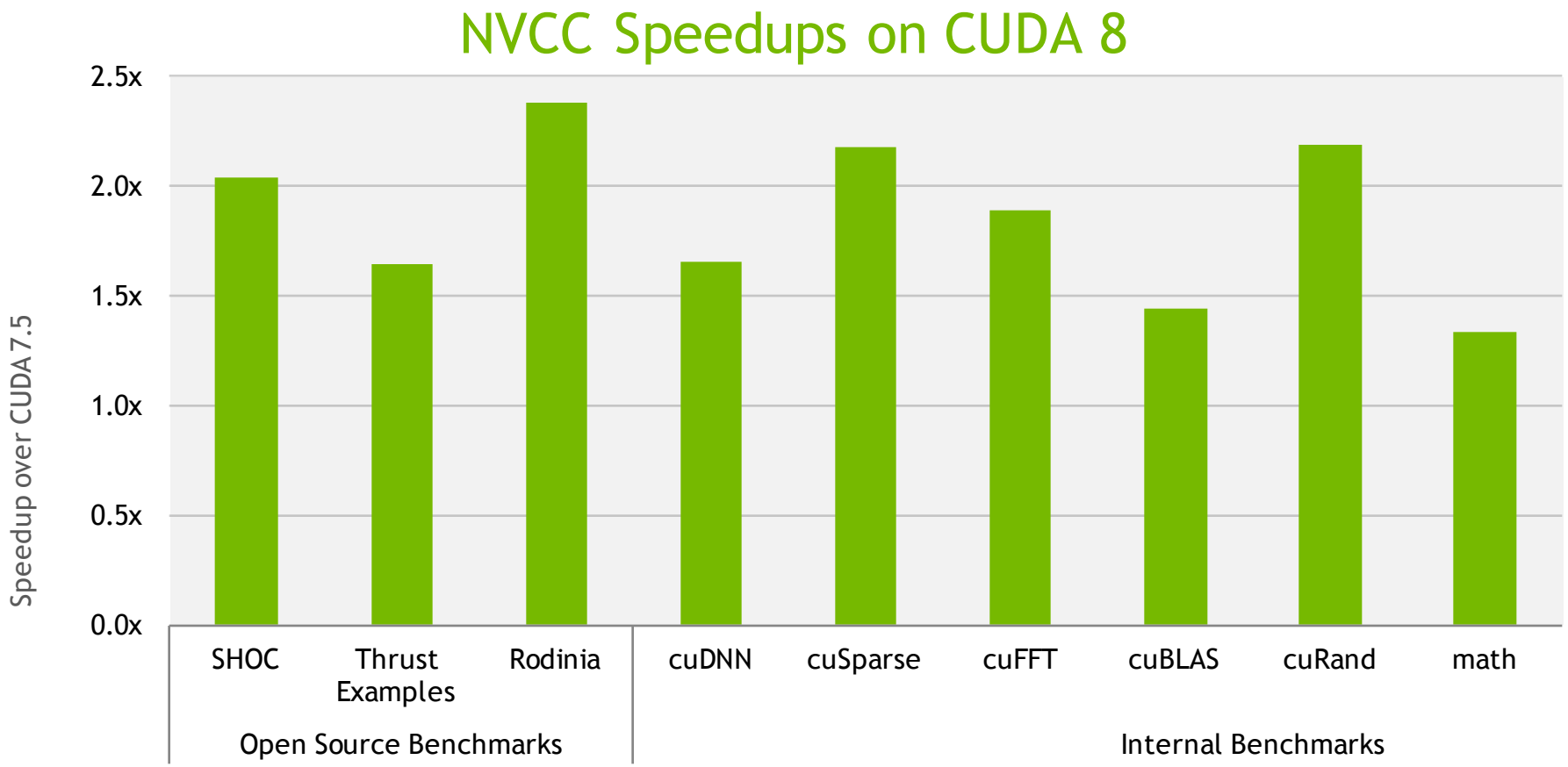
OpenACC Profiling



NVLink Topology and Bandwidth profiling

COMPILER IMPROVEMENTS

2X FASTER COMPILE TIME ON CUDA 8



QUDA increase
1.54x

- Average total compile times (per translation unit)
- Intel Core i7-3930K (6-cores) @ 3.2GHz
- CentOS x86_64 Linux release 7.1.1503 (Core) with GCC 4.8.3 20140911
- GPU target architecture sm_52

Performance may vary based on OS and software versions, and motherboard configuration

HETEROGENEOUS C++ LAMBDA

Combined CPU/GPU lambda functions

```
__global__ template <typename F, typename T>
void apply(F function, T *ptr) {
  *ptr = function(ptr);
}

int main(void) {
  float *x;
  cudaMallocManaged(&x, 2);

  auto square =
    [=] __host__ __device__ (float x) { return x*x; };

  apply<<<1, 1>>>(square, &x[0]);

  ptr[1] = square(&x[1]);

  cudaFree(x);
}
```

Call lambda from device code

__host__ __device__ lambda

Pass lambda to CUDA kernel

… or call it from host code

Experimental feature in CUDA 8.
`nvcc --expt-extended-lambda`

HETEROGENEOUS C++ LAMBDA

Usage with Thrust

```
void saxpy(float *x, float *y, float a, int N) {
    using namespace thrust;
    auto r = counting_iterator(0);

    auto lambda = [=] __host__ __device__ (int i) {
      y[i] = a * x[i] + y[i];
    };

    if(N > gpuThreshold)
      for_each(device, r, r+N, lambda);
    else
      for_each(host, r, r+N, lambda);
}
```

__host__ __device__ lambda

Use lambda in thrust::for_each on host or device

Experimental feature in CUDA 8.
`nvcc --expt-extended-lambda`

BEYOND

FUTURE: UNIFIED SYSTEM ALLOCATOR

Allocate unified memory using standard malloc

CUDA 8 Code with System Allocator

```
void sortfile(FILE *fp, int N) {
  char *data;

// Allocate memory using any standard allocator
  data = (char *) malloc(N * sizeof(char));

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);

  use_data(data);

// Free the allocated memory
  free(data);
}
```

Removes CUDA specific allocator restrictions

Data movement is transparently handled

Requires operating system support

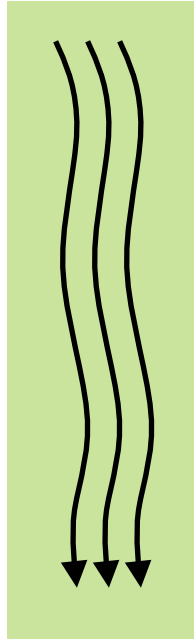
COOPERATIVE GROUPS

A Programming Model for Coordinating Groups of Threads

Support clean composition across software boundaries (e.g. Libraries)

Optimize for hardware fast-path using safe, flexible synchronization

A programming model that can scale from Kepler to future platforms



COOPERATIVE GROUPS SUMMARY

Flexible, Explicit Synchronization

Thread groups are explicit objects in the program

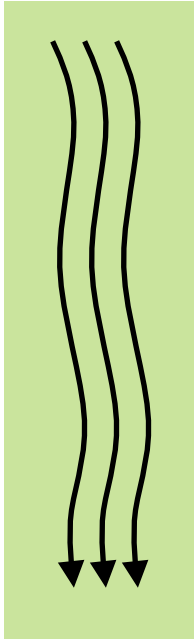
```
thread_group group = this_thread_block();
```

Collectives, such as barriers, operate on thread groups

```
sync(group);
```

New groups are constructed by partitioning existing groups

```
thread_group tiled_partition(thread_group base, int size);
```



MOTIVATING EXAMPLE

Optimizing for Warp Size

```
__device__
int warp_reduce(int val) {
  extern __shared__ int smem[];
  const int tid = threadIdx.x;

  #pragma unroll
  for (int i = warpSize/2; i > 0; i /= 2) {
    smem[tid] = val;      __syncthreads();
    val += smem[tid ^ i]; __syncthreads();
  }
  return val;
}
```

__syncthreads() is too expensive when sharing is only within warps

MOTIVATING EXAMPLE

Implicit Warp-Synchronous Programming is Tempting…

```
__device__
int warp_reduce(int val) {
  extern __shared__ int smem[];
  const int tid = threadIdx.x;

  #pragma unroll
  for (int i = warpSize/2; i > 0; i /= 2) {
    smem[tid] = val;
    val += smem[tid ^ i];
  }
  return val;
}
```

Barriers separating steps removed.
UNSAFE!

MOTIVATING EXAMPLE

Safe, Explicit Programming for Performance

Approximately equal performance to unsafe warp programming

```
__device__
int warp_reduce(int val) {
  extern __shared__ int smem[];
  const int tid = threadIdx.x;

  #pragma unroll
  for (int i = warpSize/2; i > 0; i /= 2) {
    smem[tid] = val;        sync(this_warp());
    val += smem[tid ^ i];   sync(this_warp());
  }
  return val;
}
```

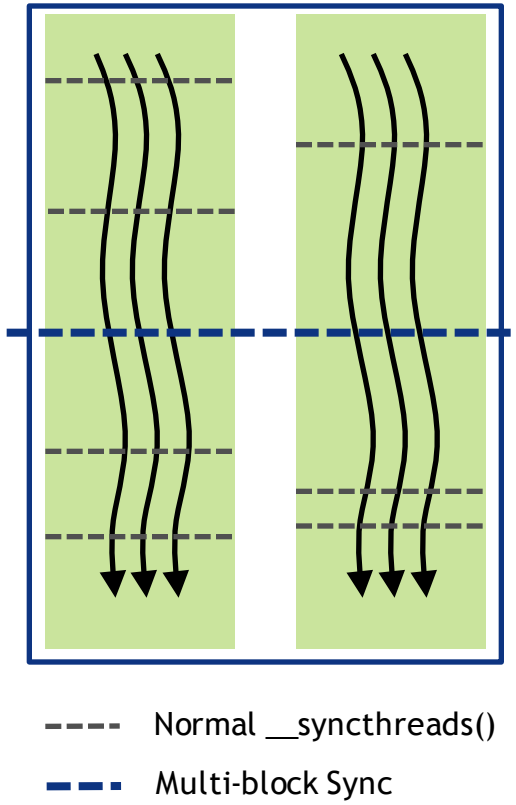
Safe *and* Fast!

PASCAL: MULTI-BLOCK COOPERATIVE GROUPS

Provide a new launch mechanism for multi-block groups

Cooperative Groups collective operations like `sync(group)` work across all threads in the group

Save bandwidth and latency compared to multi-kernel approach required on Kepler GPUs



GPU TECHNOLOGY CONFERENCE

April 4-7, 2016 | Silicon Valley

CUDA 8 AND BEYOND

JOIN THE CONVERSATION

#GTC16

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

http://parallelforall.com

mharris@nvidia.com

@harrism

PRESENTED BY

NVIDIA.