GPU TECHNOLOGY CONFERENCE

April 4-7, 2016 | Silicon Valley

# THE FUTURE OF UNIFIED MEMORY

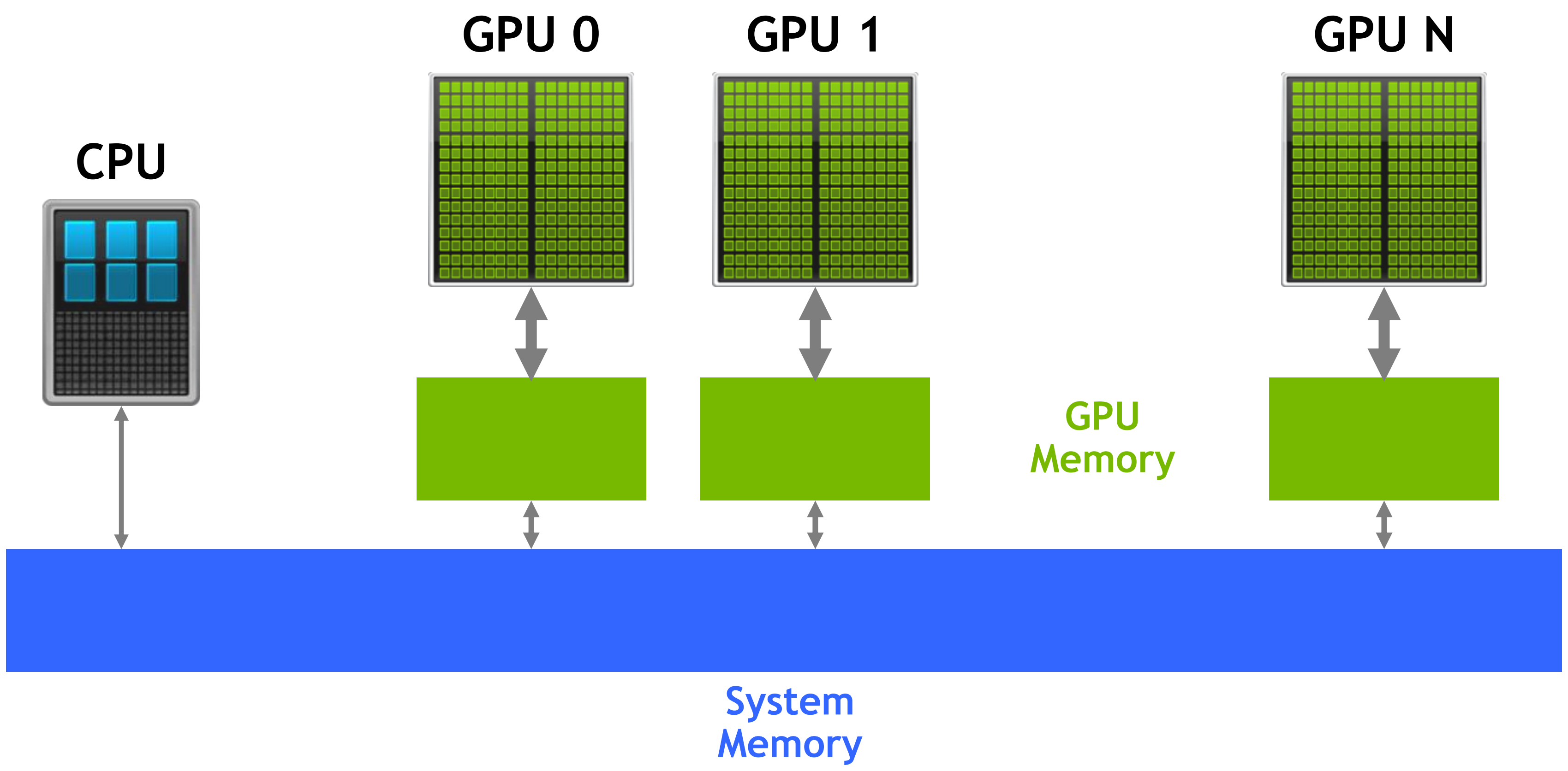
Nikolay Sakharnykh, 4/5/2016

PRESENTED BY

NVIDIA

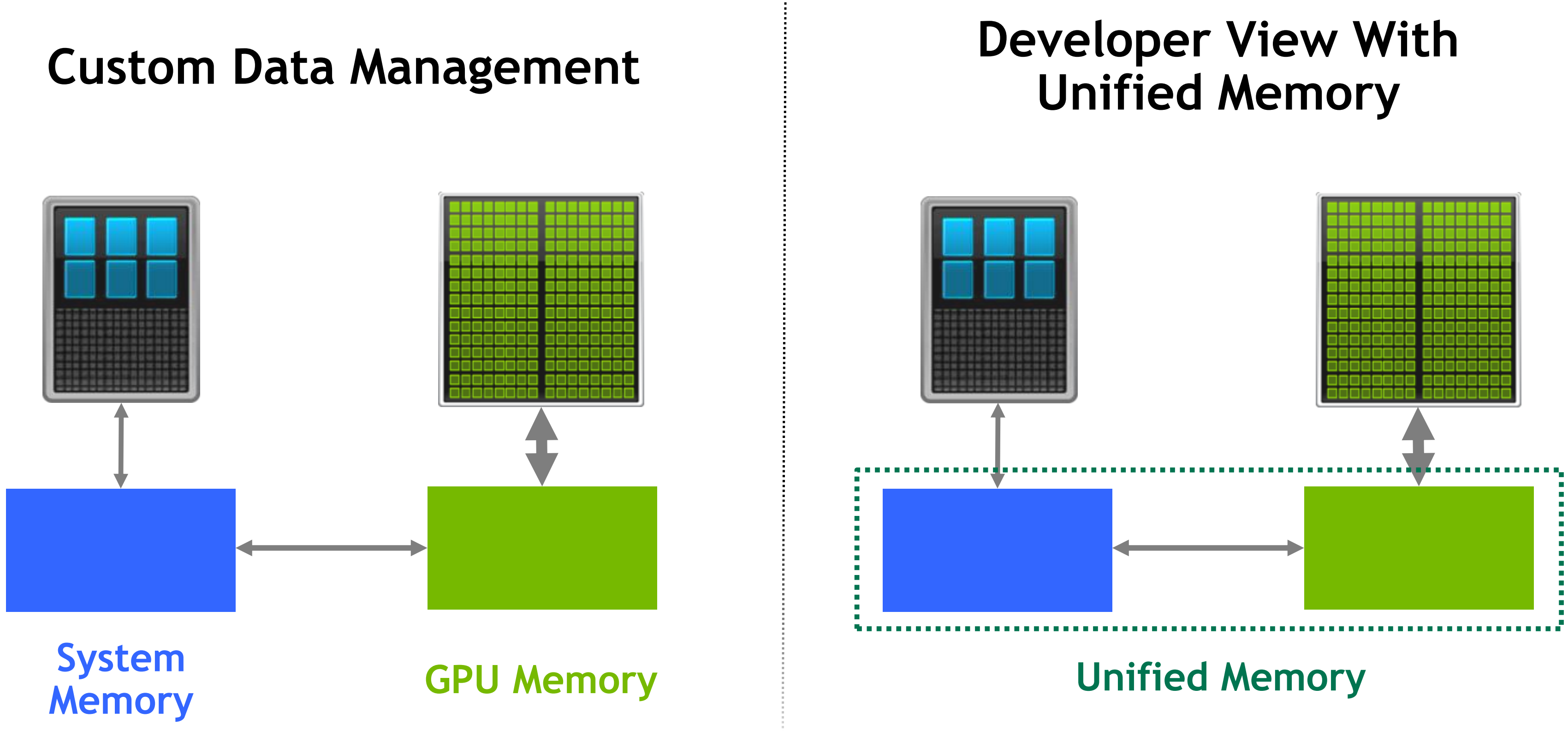
# HETEROGENEOUS ARCHITECTURES

## Memory hierarchy



# UNIFIED MEMORY

## Starting with Kepler and CUDA 6



# UNIFIED MEMORY

## Single pointer for CPU and GPU

### 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);
}
```

### GPU 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 ON PRE-PASCAL

## Code example explained

```
cudaMallocManaged(&ptr, ...);
*ptr = 1;
qsort<<<...>>>(ptr);
```

← Pages are populated in GPU memory

← **CPU page fault**: data migrates to CPU

← **Kernel launch**: data migrates to GPU

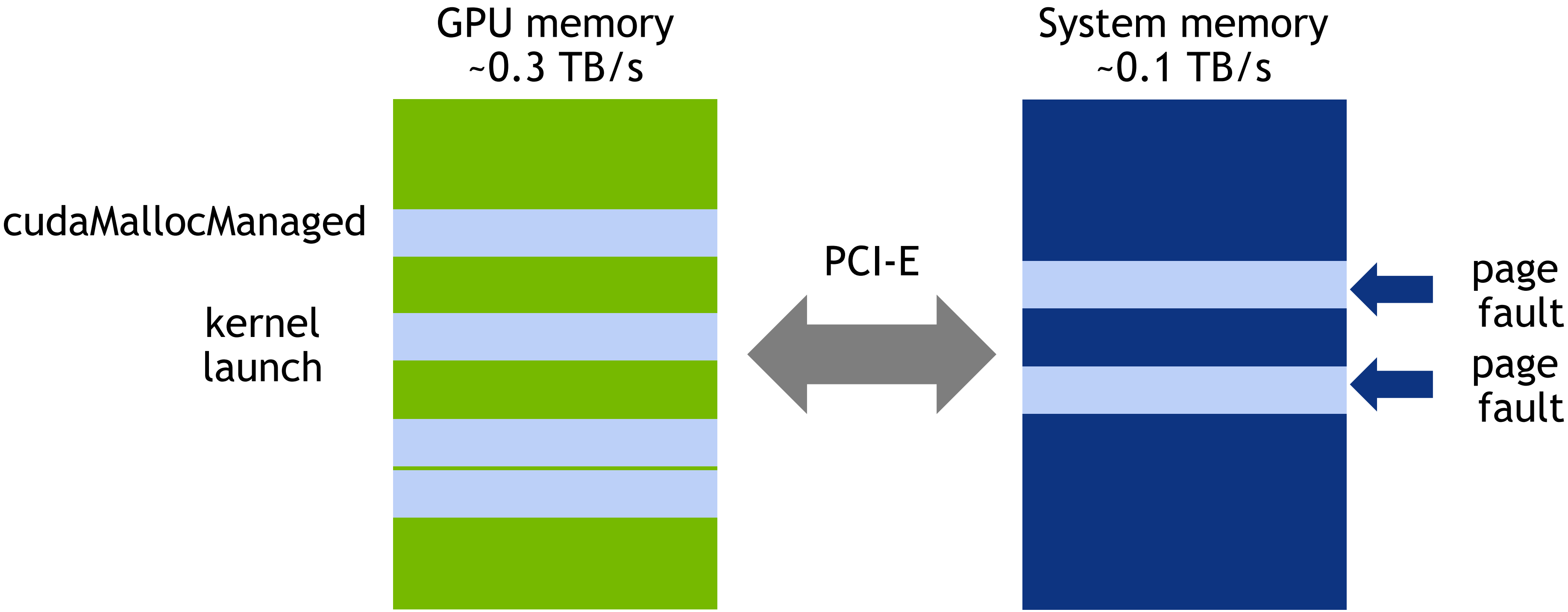
GPU always has address translation during the kernel execution

Pages allocated **before** they are used – cannot oversubscribe GPU

Pages migrate to GPU only on kernel launch – cannot migrate on-demand

# UNIFIED MEMORY ON PRE-PASCAL

## Kernel launch triggers bulk page migrations



# UNIFIED MEMORY ON PASCAL

## Now supports GPU page faults

```
cudaMallocManaged(&ptr, ...);
*ptr = 1;
qsort<<<...>>>(ptr);
```

← Empty, no pages anywhere (similar to malloc)

← **CPU page fault**: data allocates on CPU

← **GPU page fault**: data migrates to GPU

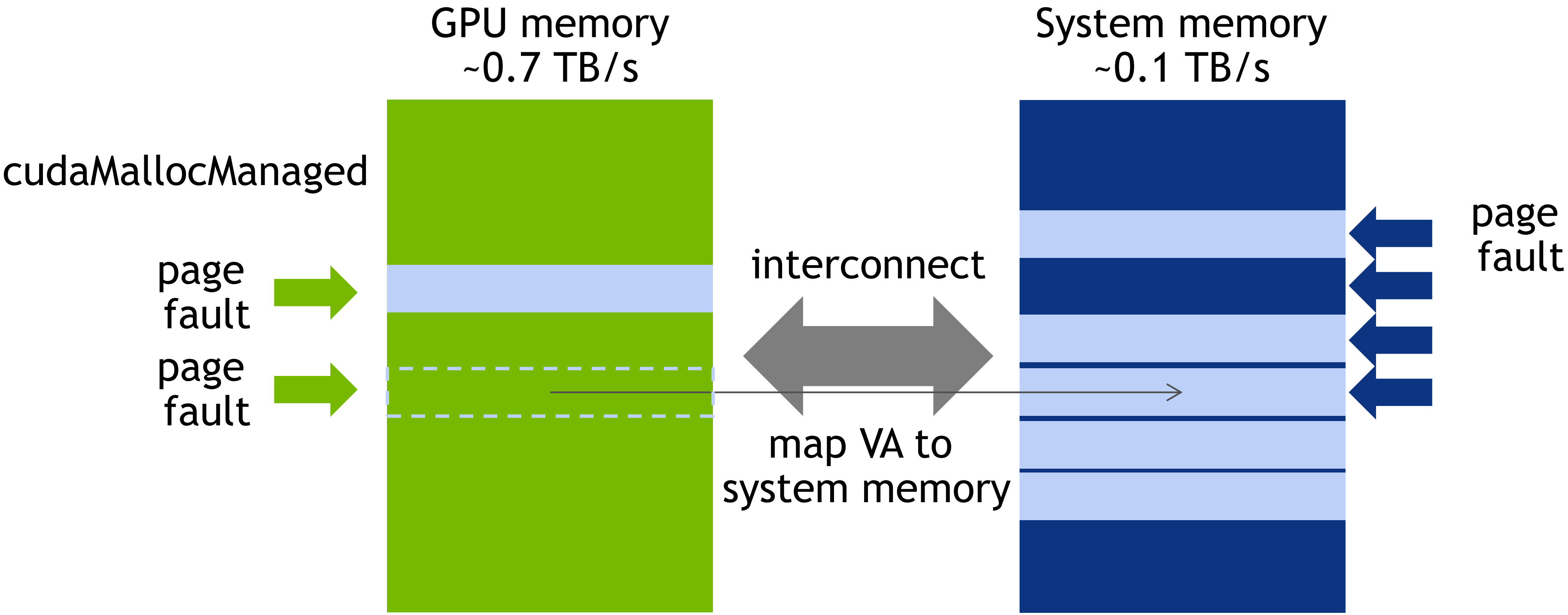
If GPU does not have a VA translation, it issues an interrupt to CPU

Unified Memory driver could decide to map or migrate depending on heuristics

Pages populated and data migrated **on first touch**

# UNIFIED MEMORY ON PASCAL

## True on-demand page migrations



# UNIFIED MEMORY ON PASCAL

## Improvements over previous GPU generations

On-demand page migration

GPU memory oversubscription is now practical (*)

Concurrent access to memory from CPU and GPU (page-level coherency)

Can access OS-controlled memory on supporting systems

(*) on pre-Pascal you can use zero-copy but the data will always stay in system memory

# UNIFIED MEMORY: ATOMICS

**Pre-Pascal:** atomics from the GPU are atomic only for *that GPU*

GPU atomics to peer memory are **not** atomic for remote GPU

GPU atomics to CPU memory are **not** atomic for CPU operations

**Pascal:** Unified Memory enables wider scope for atomic operations

NVLINK supports native atomics in hardware

PCI-E will have software-assisted atomics

# UNIFIED MEMORY: MULTI-GPU

**Pre-Pascal:** direct access requires P2P support, otherwise falls back to sysmem

Use `CUDA_MANAGED_FORCE_DEVICE_ALLOC` to mitigate this

**Pascal:** Unified Memory works very similar to CPU-GPU scenario

GPU A accesses GPU B memory: GPU A takes a page fault

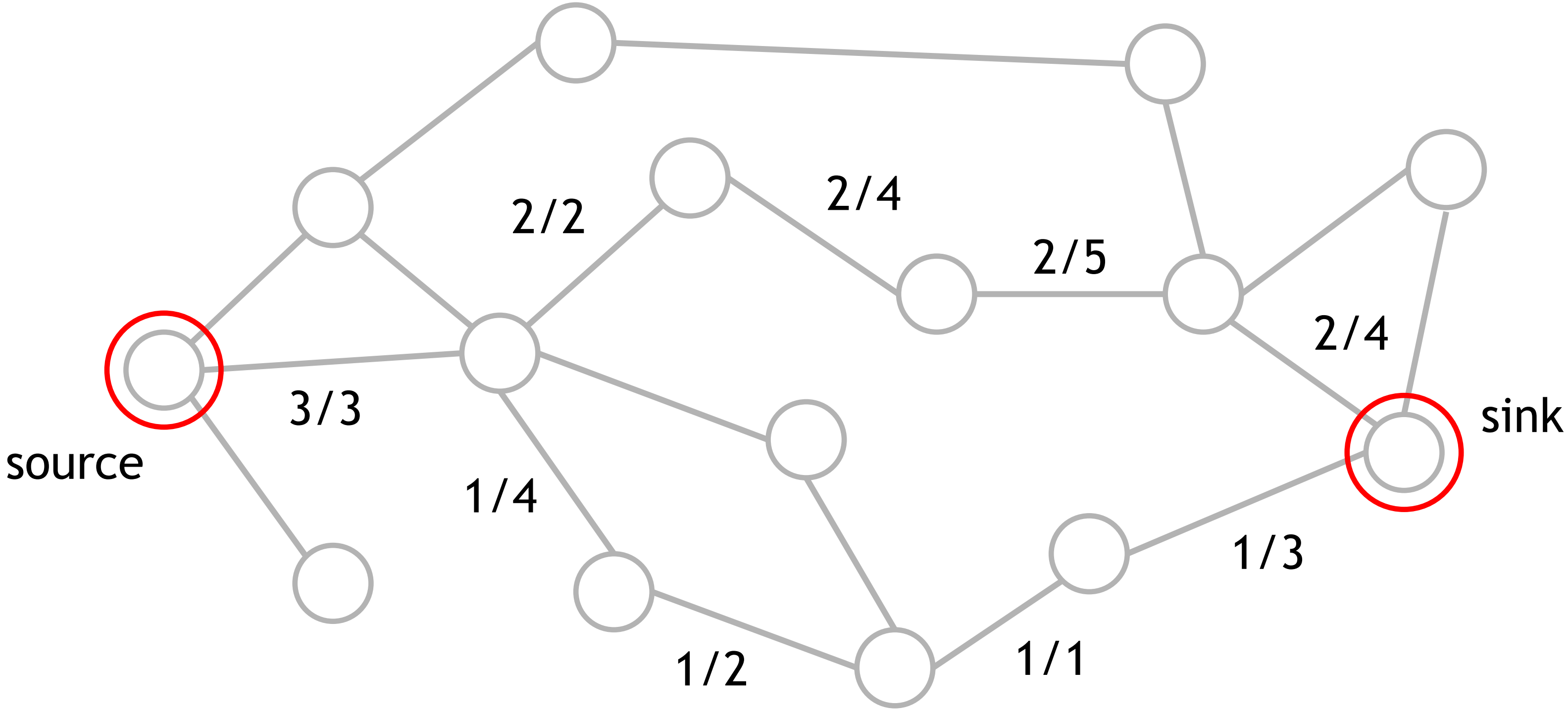
Can decide to migrate from GPU B to GPU A, or map GPU A

GPUs can map each other's memory, but CPU cannot access GPU memory directly

# NEW APPLICATION USE CASES

# ON-DEMAND PAGING

## Maximum flow



# ON-DEMAND PAGING

## Maximum flow

Edmonds-Karp algorithm pseudo-code:

```
while (augmented path exists)
{
  run BFS to find augmented path
  backtrack and update flow graph
}
```

← Parallel: run on GPU
← Serial: run on CPU

Implementing this algorithm without Unified Memory is just **painful**

Hard to predict what edges will be touched on GPU or CPU, very data-driven

# ON-DEMAND PAGING

## Maximum flow with Unified Memory

**Pre-Pascal:**

The whole graph has to be migrated to GPU memory

Significant **start-up time**, and graph size **limited to GPU memory size**

**Pascal:**

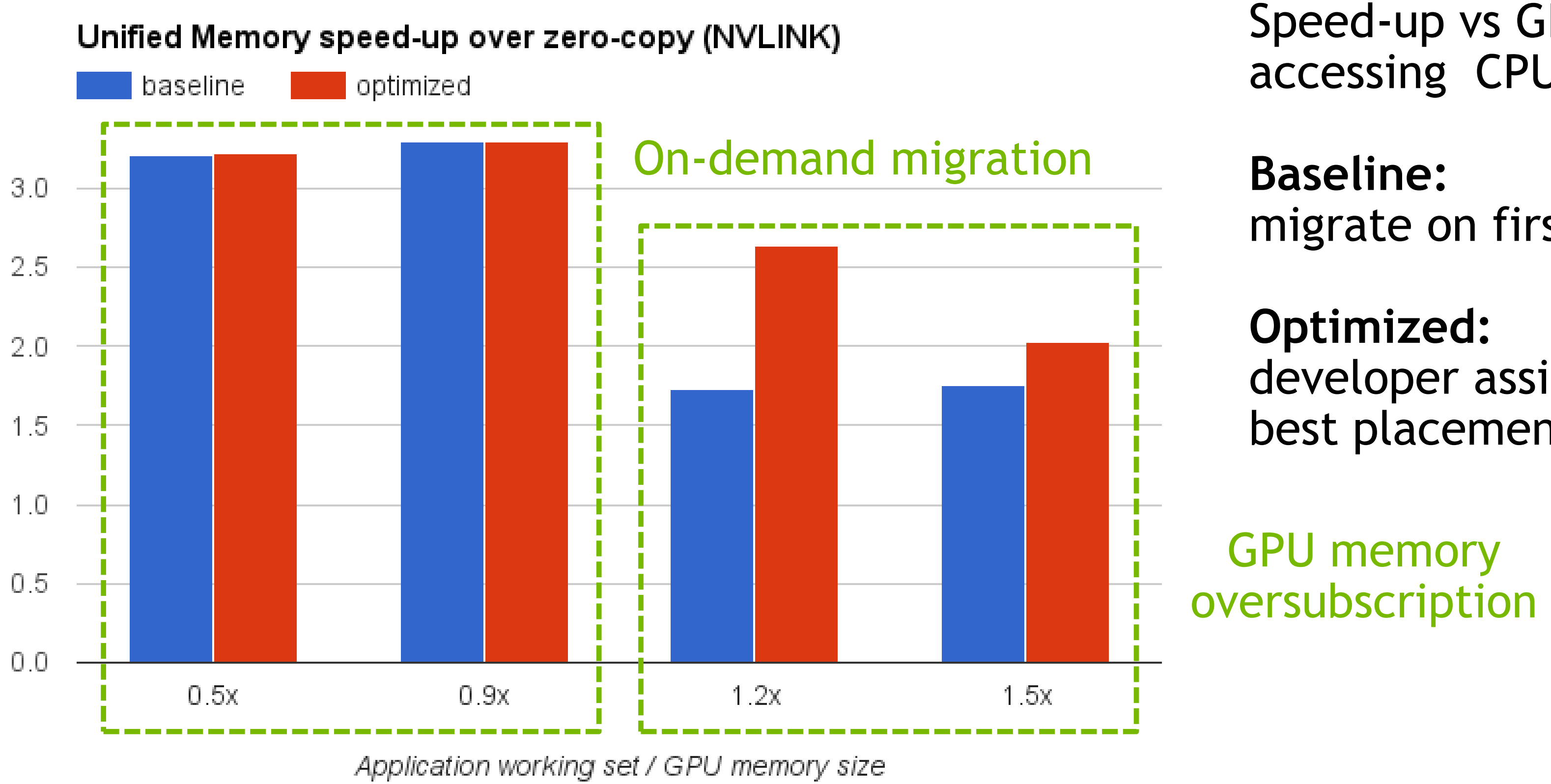
Both CPU and GPU bring only necessary vertices/edges on-demand

Can work on very large graphs that cannot fit into GPU memory

Multiple BFS iterations can amortize the cost of page migration

# ON-DEMAND PAGING

## Maximum flow performance projections




Speed-up vs GPU directly accessing CPU memory (zero-copy)

**Baseline:**
migrate on first touch

**Optimized:**
developer assists with hints for best placement in memory

# GPU OVERSUBSCRIPTION

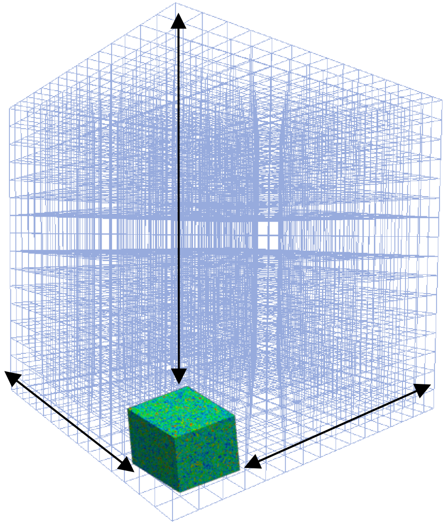
## Now possible with Pascal

Many domains would benefit from GPU memory oversubscription:

**Combustion** – many species to solve for

**Quantum chemistry** – larger systems

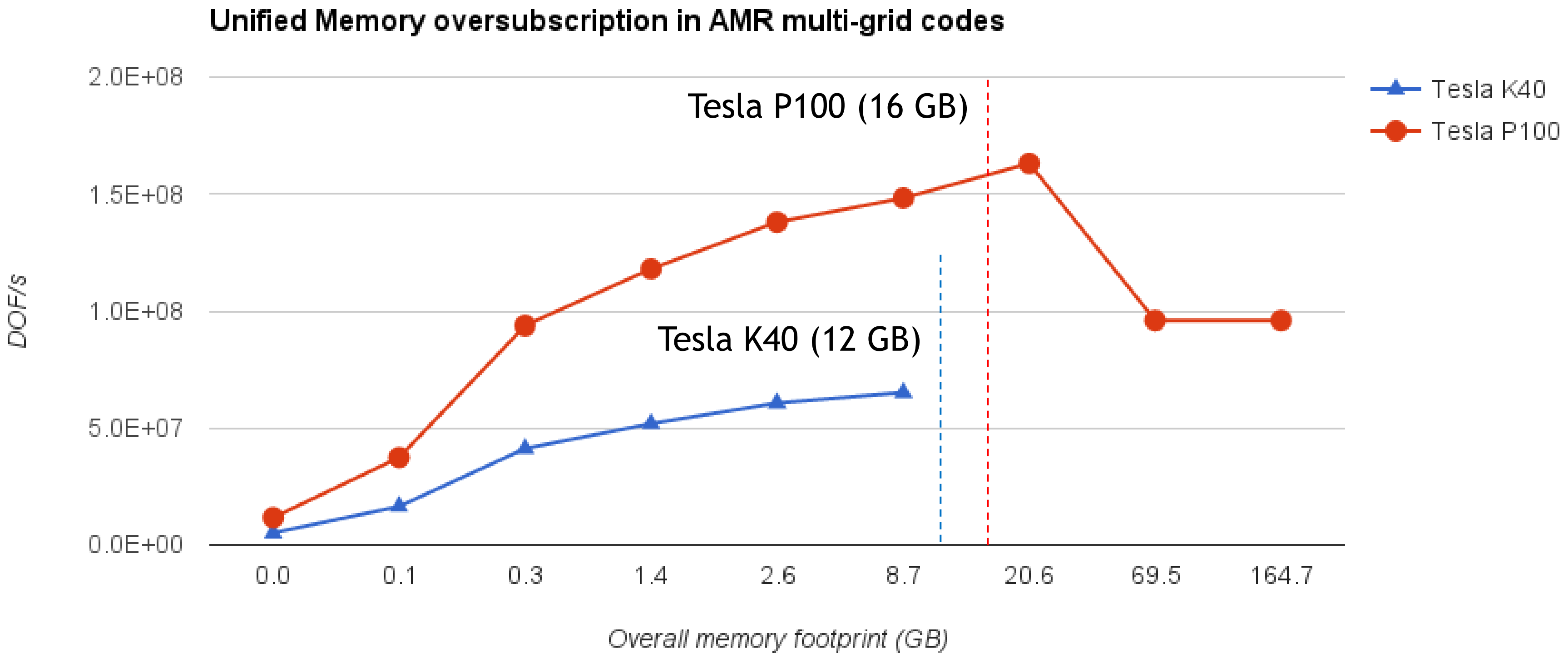
**Ray-tracing** - larger scenes to render



Unified Memory on Pascal will provide oversubscription by default!

# GPU OVERSUBSCRIPTION

## HPGMG: high-performance multi-grid




*Tesla P100 performance is very early modelling results

# ON-DEMAND ALLOCATION

## Dynamic queues

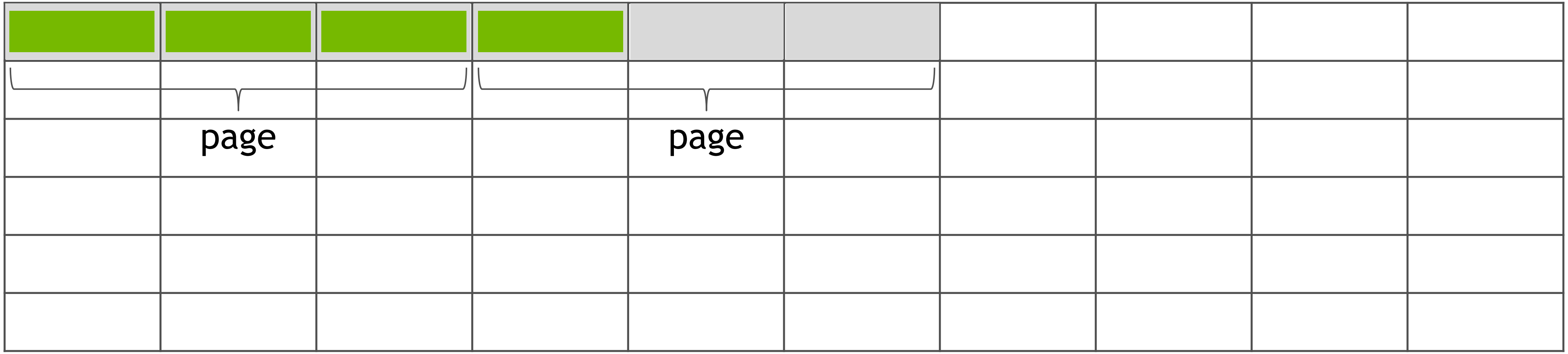
Problem: GPU populates queues with unknown size, need to overallocate


Solution: use Unified Memory for allocations (on Pascal)

# ON-DEMAND ALLOCATION

## Dynamic queues

Memory is allocated on-demand so we don't waste resources




All translations from a given SM stall on page fault on Pascal

# PERFORMANCE TUNING

# PERFORMANCE TUNING

## General guidelines

Minimize page fault overhead:

Fault handling can take **10s of µs**, while execution stalls

Keep data local to the accessing processor:

Higher bandwidth, lower latency

Minimize thrashing:

Migration overhead can exceed locality benefits

# PERFORMANCE TUNING

## New hints in CUDA 8

**`cudaMemPrefetchAsync`**`(ptr, length, destDevice, stream)`

Unified Memory alternative to cudaMemcpyAsync

Async operation that follows CUDA stream semantics

**`cudaMemAdvise`**`(ptr, length, advice, device)`

Specifies allocation and usage policy for memory region

User can set and unset advices at any time

# PREFETCHING

## Simple code example

```
void foo(cudaStream_t s) {
  char *data;
  cudaMallocManaged(&data, N);

  init_data(data, N);

  cudaMemPrefetchAsync(data, N, myGpuId, s);
  mykernel<<<..., s>>>(data, N, 1, compare);
  cudaMemPrefetchAsync(data, N, cudaCpuDeviceId, s);
  cudaStreamSynchronize(s);

  use_data(data, N);

  cudaFree(data);
}
```

GPU faults are expensive
prefetch to avoid excess faults

CPU faults are less expensive
may still be worth avoiding

# READ DUPLICATION

**`cudaMemAdviseSetReadMostly`**

Use when data is *mostly read* and occasionally written to

```
init_data(data, N);

cudaMemAdvise(data, N, cudaMemAdviseSetReadMostly, myGpuId);

mykernel<<<...>>>(data, N);

use_data(data, N);
```

Read-only copy will be created on GPU page fault

CPU reads will not page fault

# READ DUPLICATION

Prefetching creates read-duplicated copy of data and avoids page faults

Note: writes are allowed but will generate page fault and remapping

```
init_data(data, N);

cudaMemAdvise(data, N, cudaMemAdviseSetReadMostly, myGpuId);
cudaMemPrefetchAsync(data, N, myGpuId, cudaStreamLegacy);
mykernel<<<...>>>(data, N);

use_data(data, N);
```

Read-only copy will be created during prefetch

CPU and GPU reads will not fault

# DIRECT MAPPING

## Preferred location and direct access

**`cudaMemAdviseSetPreferredLocation`**

Set preferred location to avoid migrations

First access will page fault and establish mapping

**`cudaMemAdviseSetAccessedBy`**

Pre-map data to avoid page faults

First access will not page fault

Actual data location can be anywhere

# INTERACTION WITH OPERATING SYSTEM

# LINUX AND UNIFIED MEMORY

## ANY memory will be available for GPU*

### 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);
}
```

### GPU code with Unified Memory

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

  fread(data, 1, N, fp);

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

  use_data(data);

  free(data);
}
```

*on supported operating systems

# HETEROGENEOUS MEMORY MANAGER

## HMM

HMM will manage a GPU page table and keep it **synchronize** with the CPU page table

Also handle DMA mapping on behalf of the device

HMM allows **migration** of process memory to device memory

CPU access will trigger fault that will migrate memory back

HMM is **not only for GPUs**, network devices can use it as well

Mellanox has on-demand paging mechanism, so RDMA will work in future

# TAKEAWAYS

Use Unified Memory now! Your programs will work even better on Pascal

Think about new use cases to take advantage of Pascal capabilities

Performance hints will provide more flexibility for advanced developers

Even more powerful on supported OS platforms