GPU TECHNOLOGY CONFERENCE
April 4-7, 2016 | Silicon Valley

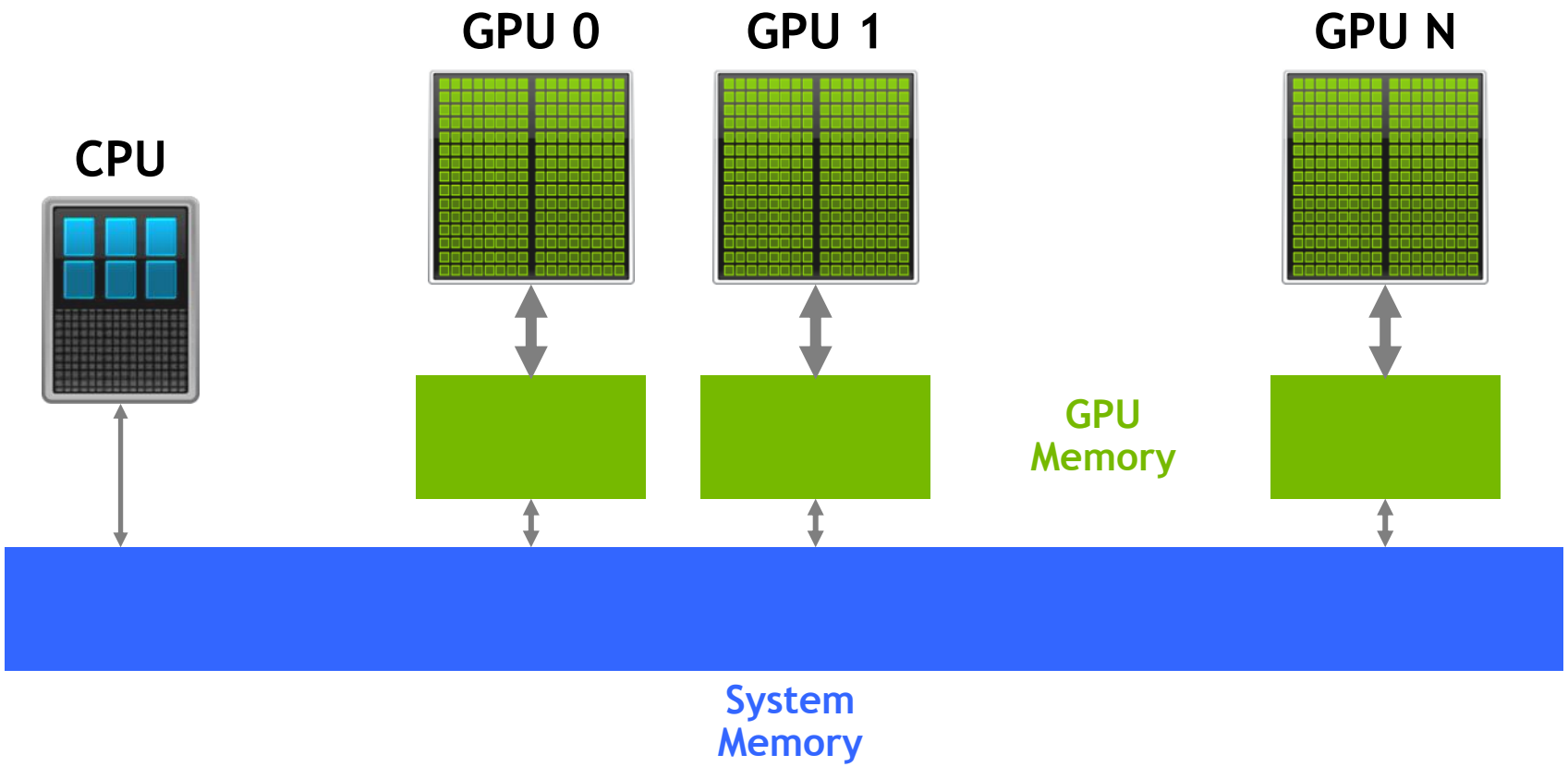
THE FUTURE OF UNIFIED MEMORY

Nikolay Sakharnykh, 4/5/2016

PRESENTED BY
NVIDIA.

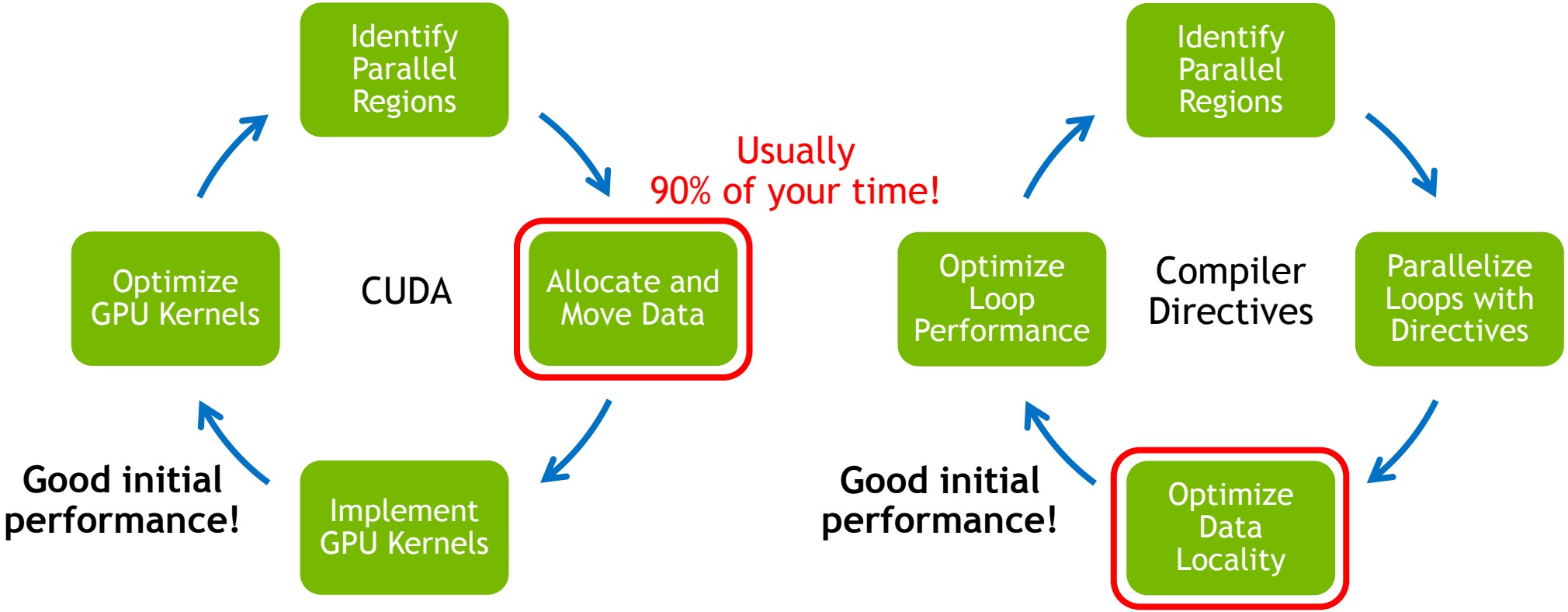
HETEROGENEOUS ARCHITECTURES

Memory hierarchy



BEFORE KEPLER AND CUDA 6

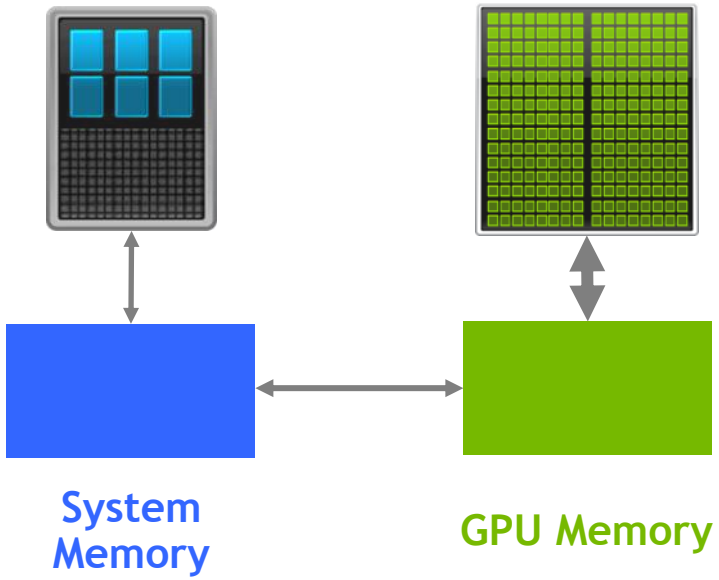
Custom data management



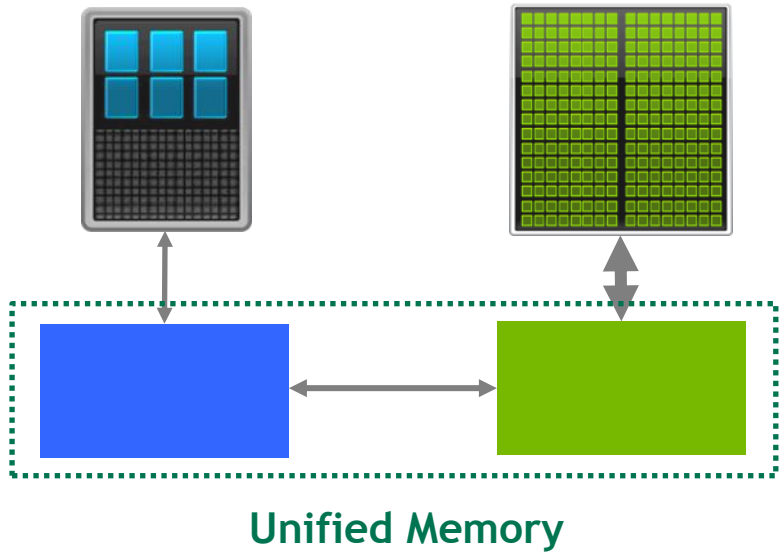
UNIFIED MEMORY

Starting with Kepler and CUDA 6

Custom Data Management

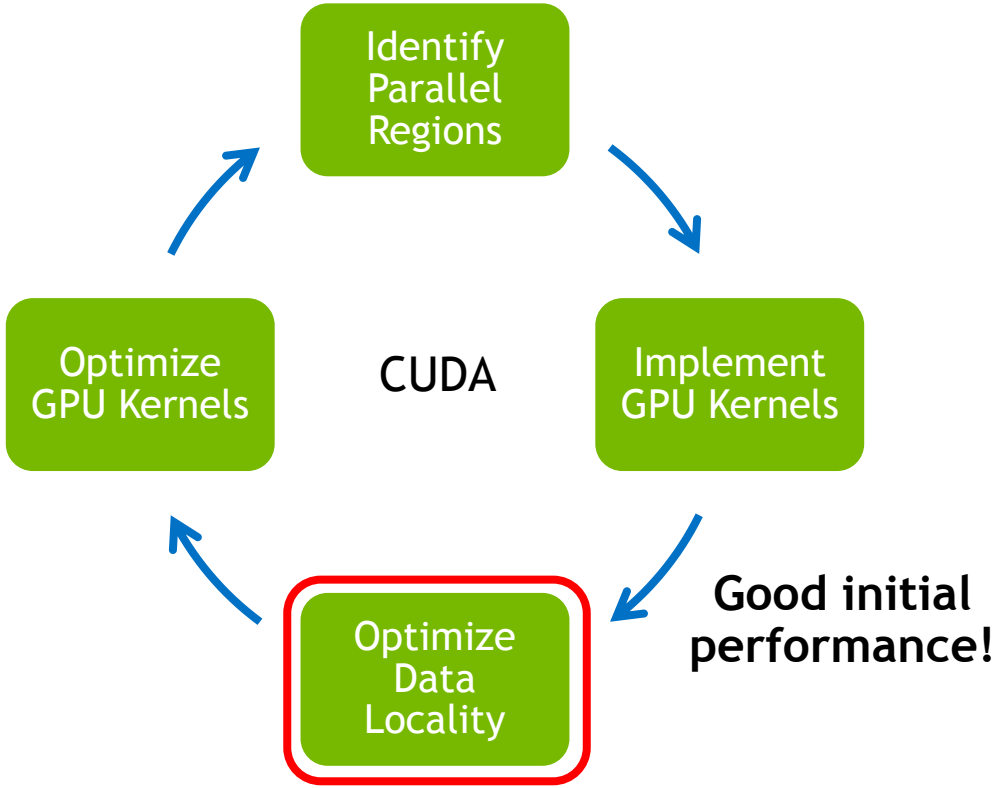


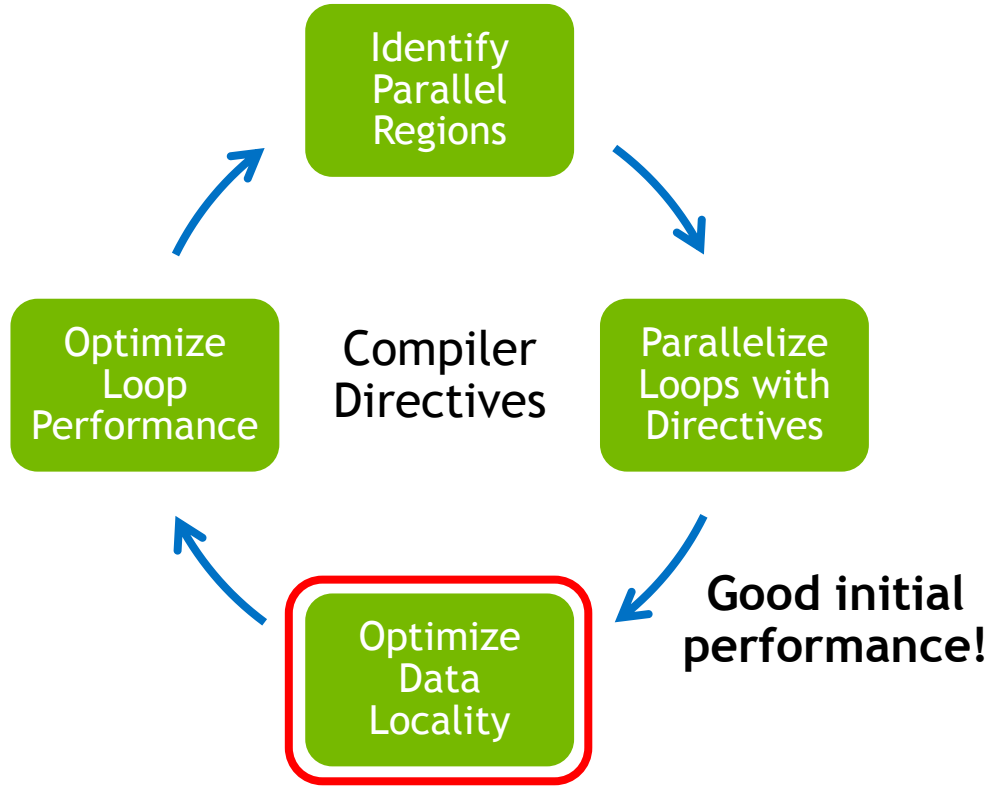
Developer View With Unified Memory



UNIFIED MEMORY

Easier to achieve initial good performance!





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

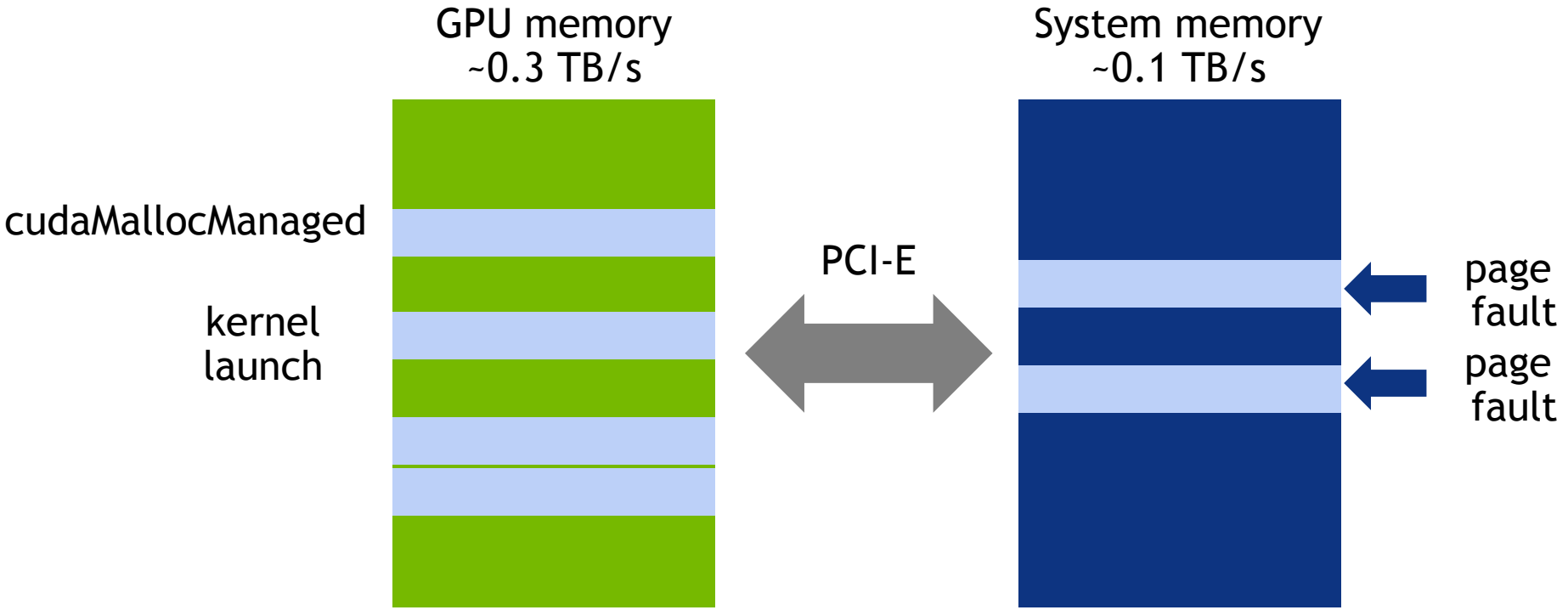
GPU always has address translation during the kernel execution

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



> For at least a decade now, I have been trying to make the case for the value of supporting virtual memory with page faults from the GPU, but without any results.

— John Carmack on 11-12-2010
http://media.armadilloaerospace.com/misc/gpuDataPaging.htm

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

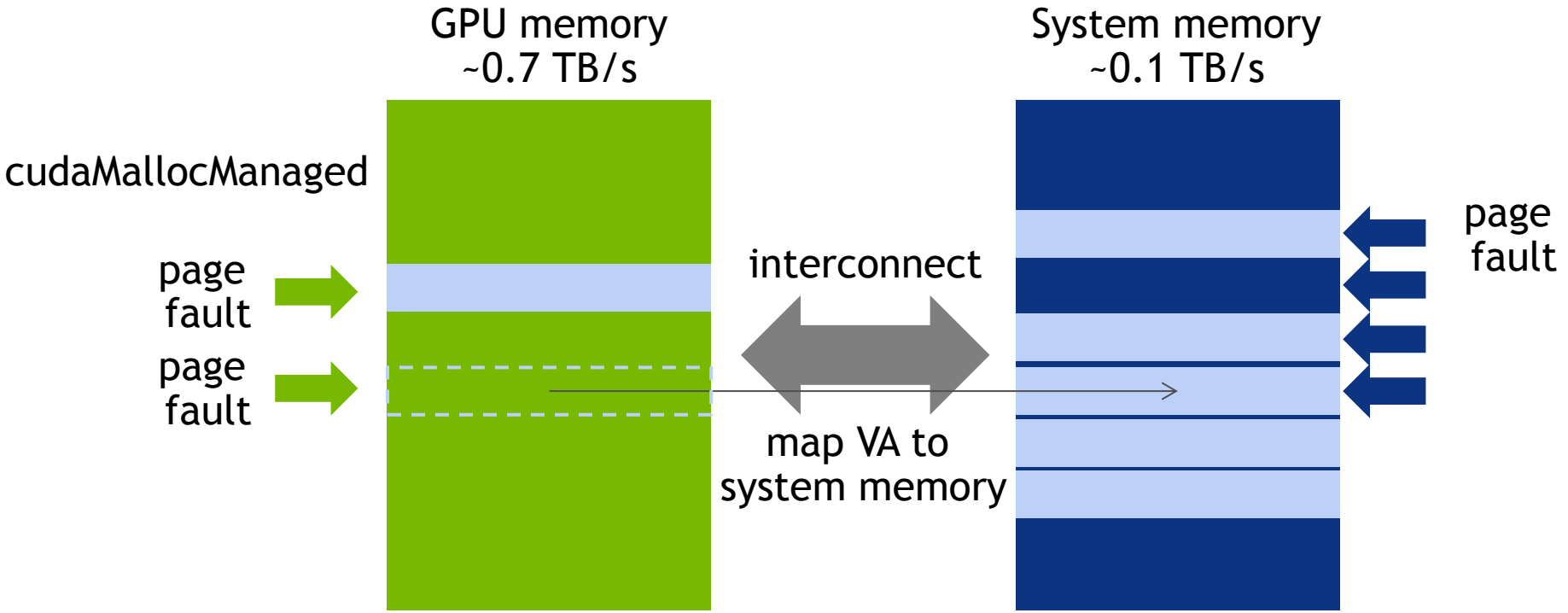
If GPU does not have a VA translation, it issues an interrupt to CPU

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

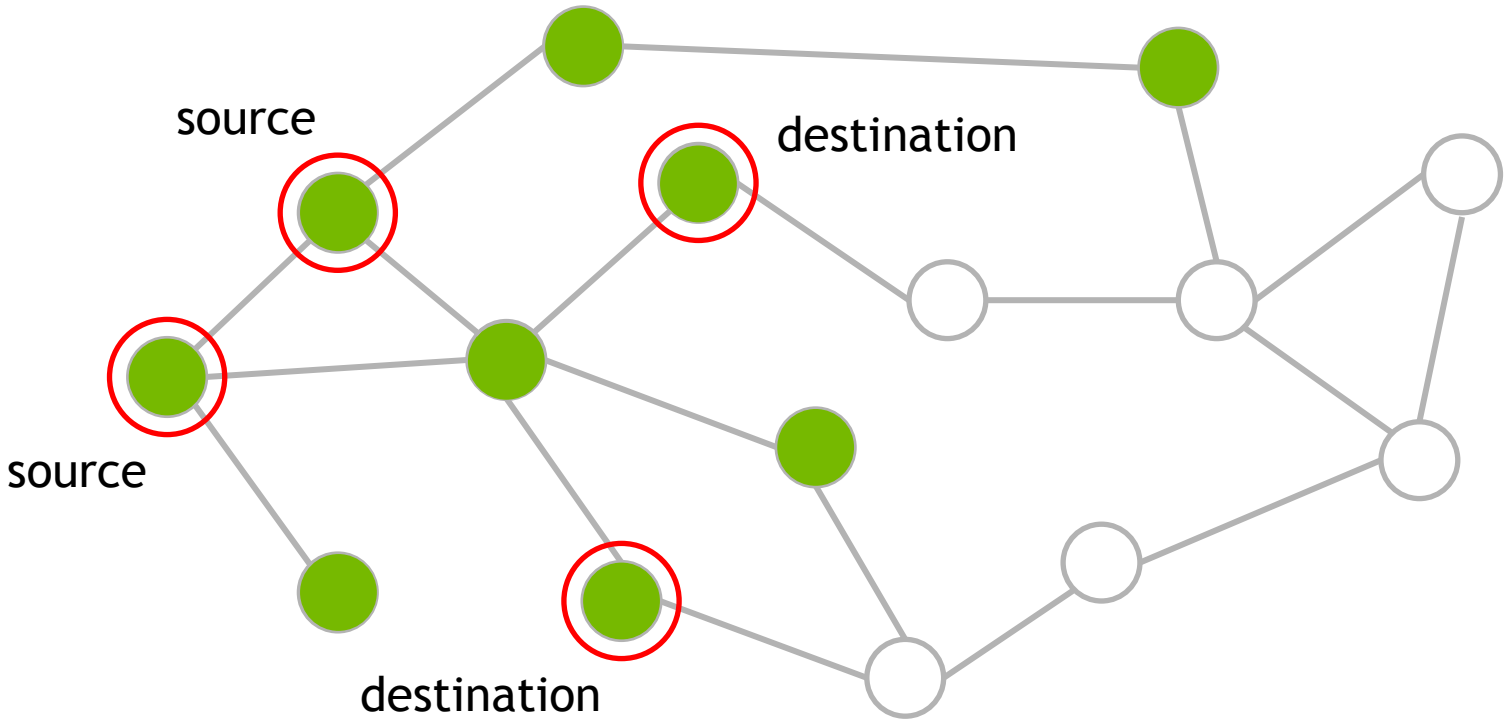
Can decide to migrate from GPU B to GPU A, or map GPU A

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

NEW APPLICATION USE CASES

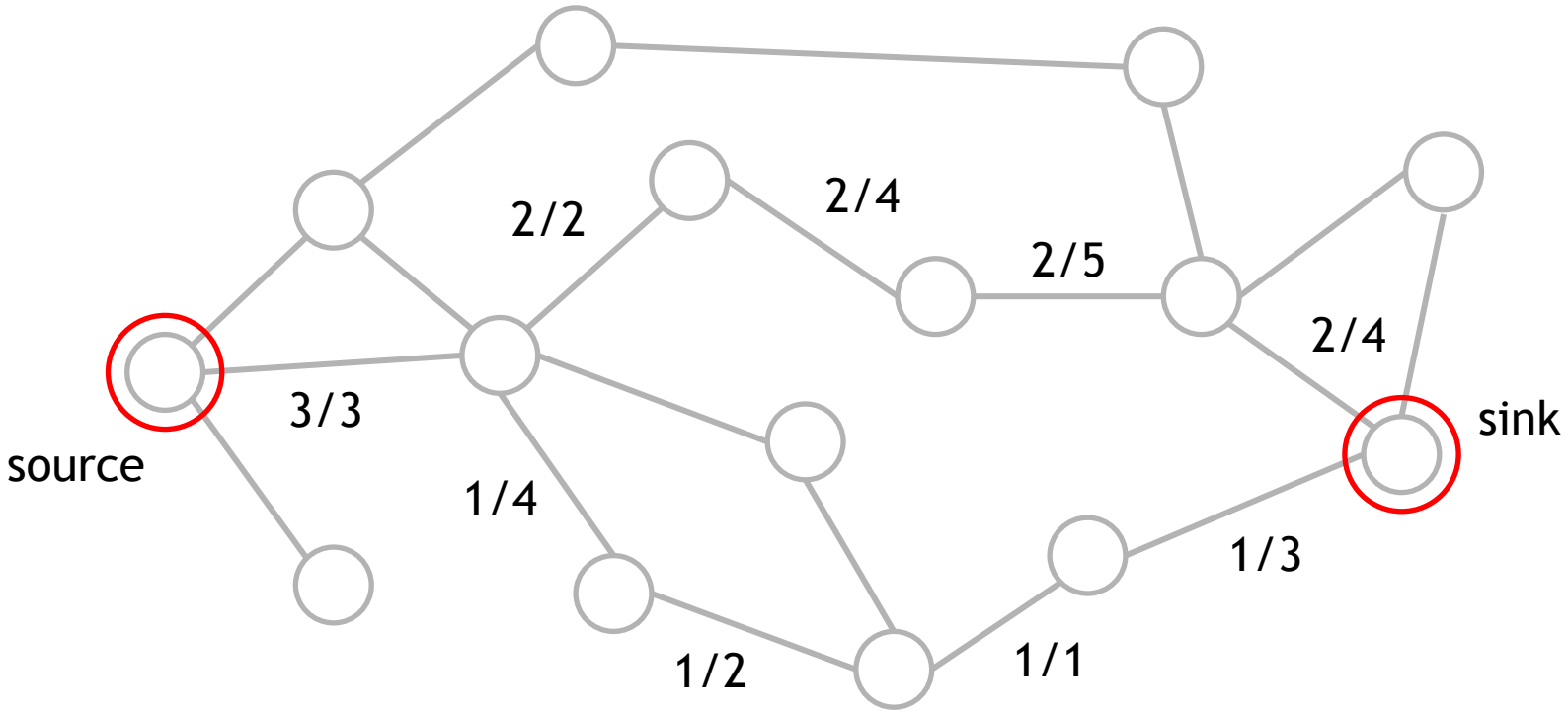
ON-DEMAND PAGING

Graph algorithms



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        <-- Parallel: run on GPU
  backtrack and update flow graph       <-- 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:

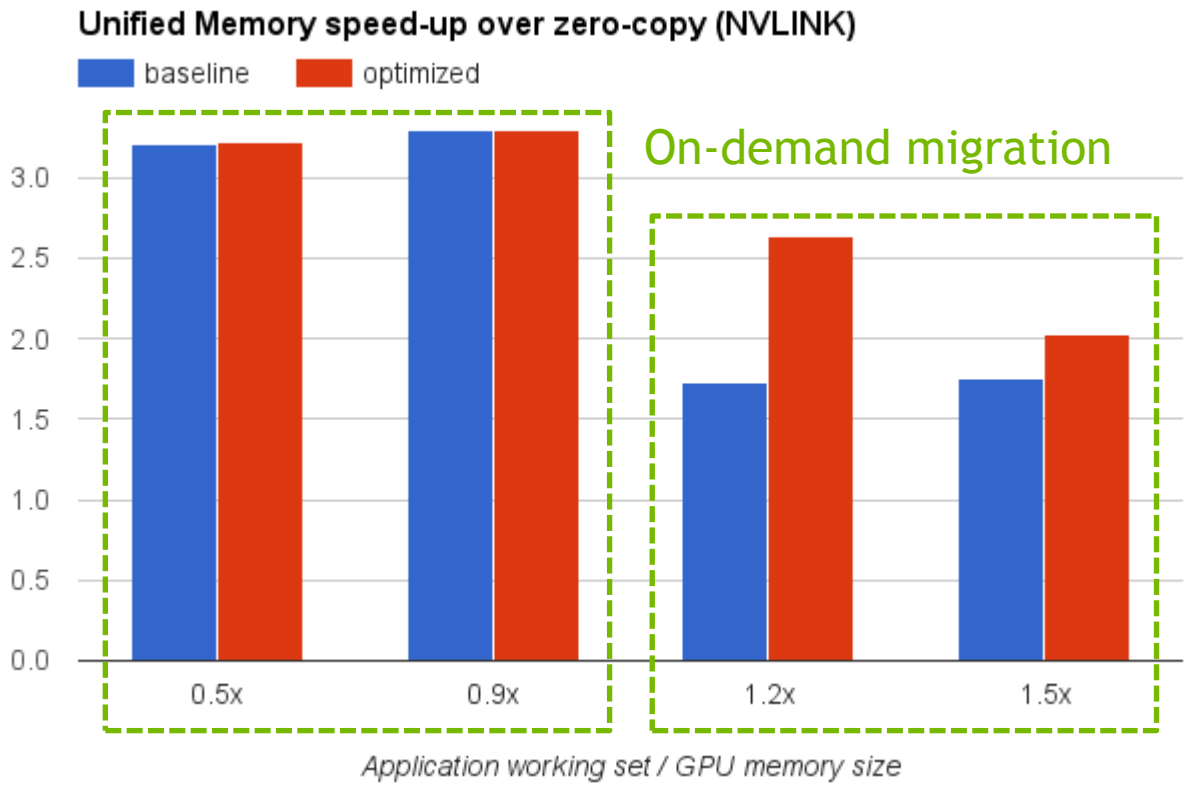
Both CPU and GPU bring only necessary vertices/edges on-demand

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 memory oversubscription

GPU OVERSUBSCRIPTION

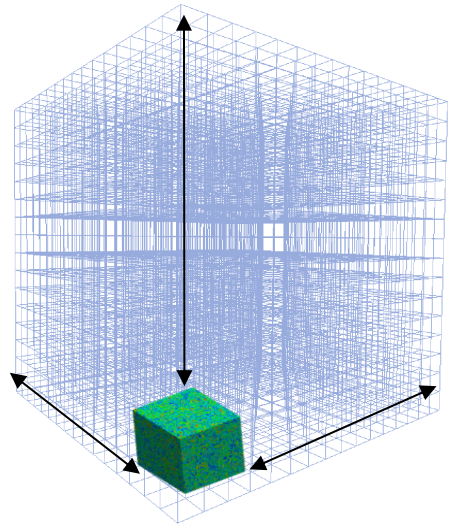
Now possible with Pascal

Many domains would benefit from GPU memory oversubscription:

Combustion – many species to solve for

Quantum chemistry – larger systems

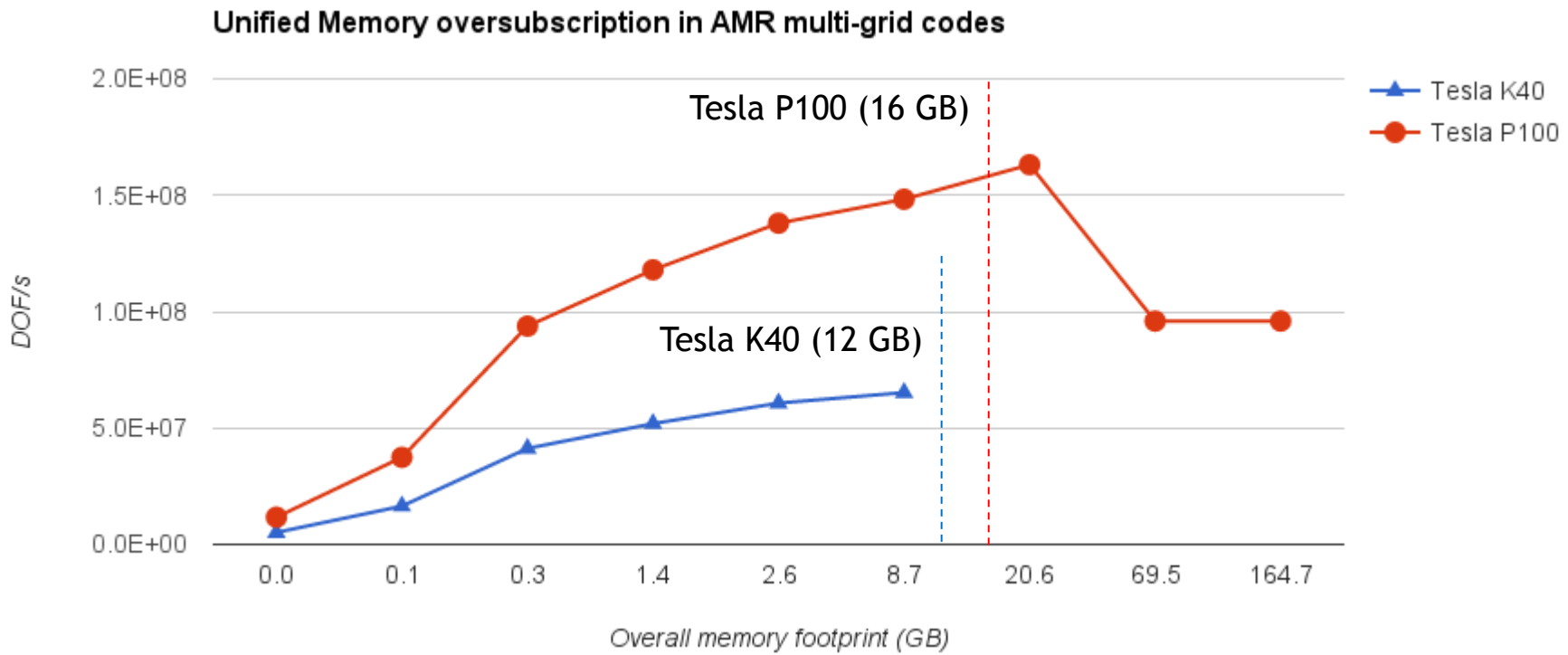
Ray-tracing - larger scenes to render



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

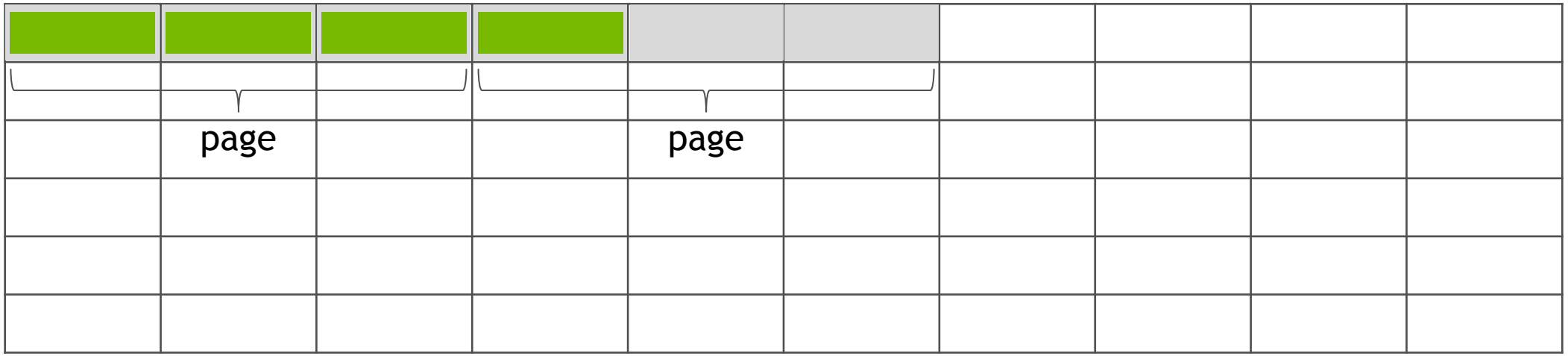
Problem: GPU populates queues with unknown size, need to overallocate

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

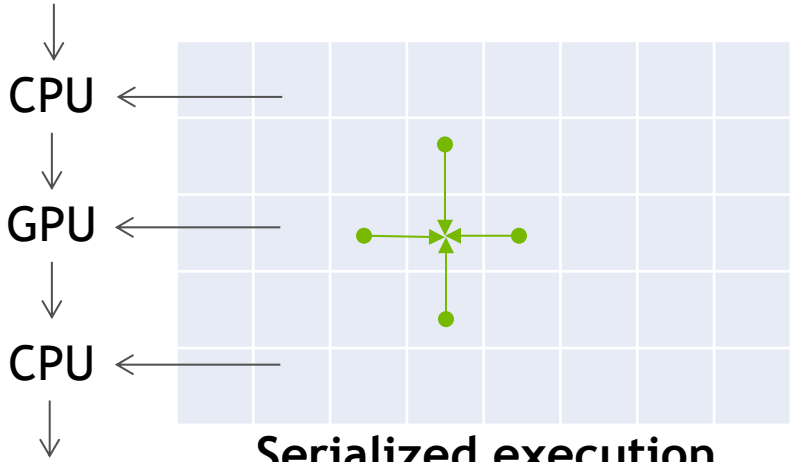
CPU and GPU reads will not fault

READ DUPLICATION

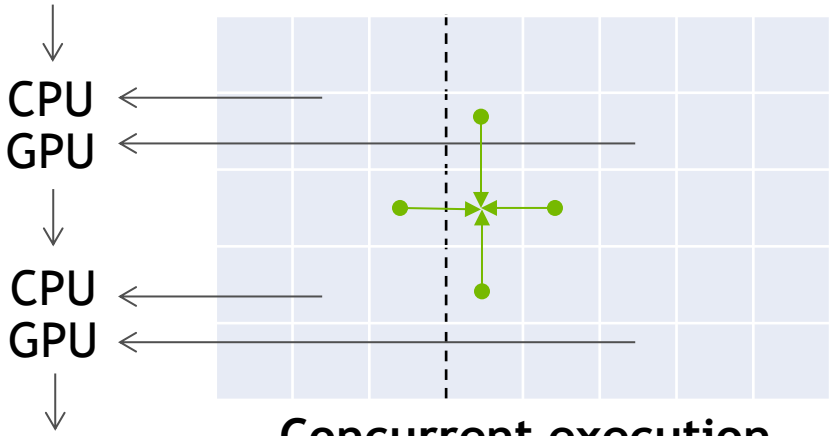
Use cases

Useful during initial stages of porting – lots of CPU code using the same structures

Other examples: mesh connectivity, matrix coefficients, control state



Serialized execution
(whole grid is duplicated)



Concurrent execution
(only halos are duplicated)

DIRECT MAPPING

Preferred location and direct access

`cudaMemAdviseSetPreferredLocation`

Set preferred location to avoid migrations

First access will page fault and establish mapping

`cudaMemAdviseSetAccessedBy`

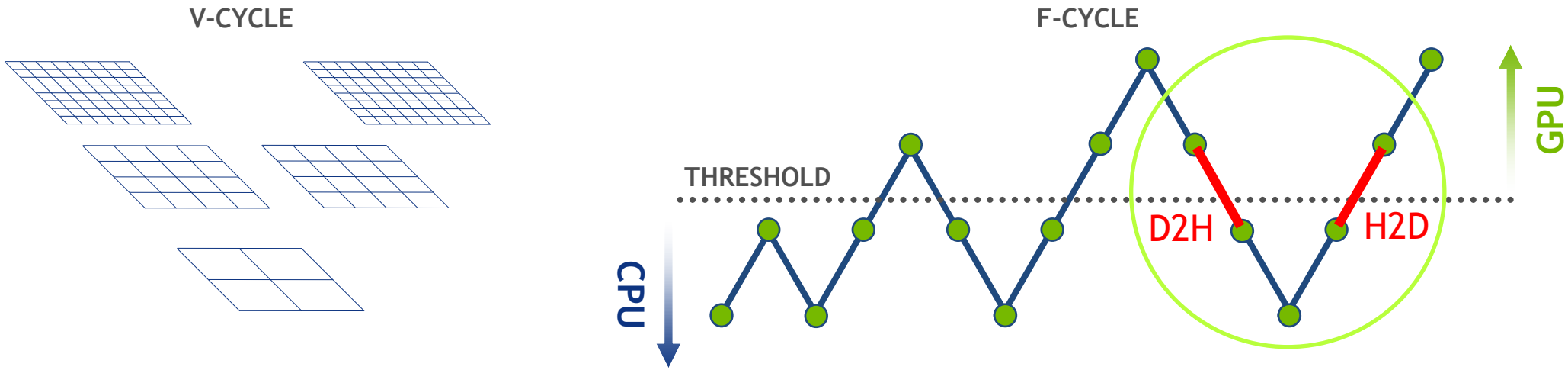
Pre-map data to avoid page faults

First access will not page fault

Actual data location can be anywhere

DIRECT MAPPING

Use case: HPGMG



Hybrid implementation with Unified Memory: **fine** grids on **GPU**, **coarse** grids on **CPU**

Implicit CPU<->GPU communication during restriction and interpolation phases

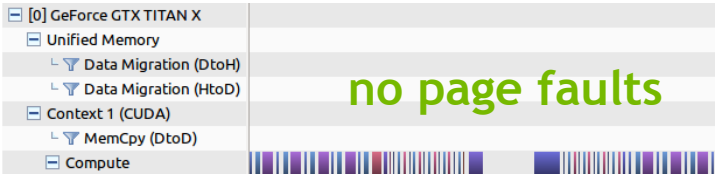
DIRECT MAPPING

Use case: HPGMG

Problem: excessive faults and migrations at CPU-GPU crossover point

Solution: pin coarse levels to CPU and map them to GPU page tables

Pre-Pascal: allocate data with cudaMallocHost or malloc + cudaHostRegister



DIRECT MAPPING

Use case: HPGMG

CUDA 8 solution with performance hints (works with cudaMallocManaged)

```
// set preferred location to CPU to avoid migrations
cudaMemAdvise(ptr, size, cudaMemAdviseSetPreferredLocation, cudaCpuDeviceId);

// keep this region mapped to my GPU to avoid page faults
cudaMemAdvise(ptr, size, cudaMemAdviseSetAccessedBy, myGpuId);

// prefetch data to CPU and establish GPU mapping
cudaMemPrefetchAsync(ptr, size, cudaCpuDeviceId, cudaStreamLegacy);
```

20-30% estimated performance improvement on Tesla P100

UNIFIED MEMORY AND DIRECTIVES

OPENACC

Explicit data management

```
float *a = (float*)malloc(sizeof(float) * n);
for(int i = 0; i < n; i++) a[i] = i;
#pragma acc kernels
{
  for(int i = 0; i < n; i++) a[i] *= 2;
}
printf("%f %f %f\n", a[0], a[1], a[2]);
```

OpenACC compiler looks for parallelism within this region

But we only access 3 elements!

We don't need to copy all of them

```
$ pgc++ -acc -ta=tesla -Minfo=all test.cpp
main:
     11, Generating copy(a[:n])
     12, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         12, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

n elements are copied before and after the loop.

OPENACC

Explicit data management

```
float *a = (float*)malloc(sizeof(float) * n);
for(int i = 0; i < n; i++) a[i] = i;
#pragma acc kernels
{
  for(int i = 0; i < n/m; i++)
    for(int j = 0; j < m; j++) a[i*m+j] *= 2;
}
printf("%f %f %f\n", a[0], a[1], a[2]);
```

2D indexing, same computation

```
$ pgc++ -acc -ta=tesla -Minfo=all test.cpp
main:
     13, Accelerator restriction: size of the GPU copy is unknown
```

The new indexing prevented compiler from generating any device code!

OPENACC

Unified Memory

```
float *a = (float*)malloc(sizeof(float) * n);
for(int i = 0; i < n; i++) a[i] = i;
#pragma acc kernels
{
  for(int i = 0; i < n/m; i++)
    for(int j = 0; j < m; j++) a[i*m+j] *= 2;
}
printf("%f %f %f\n", a[0], a[1], a[2]);
```

Data management is offloaded to Unified Memory, so explicit copies are not required!

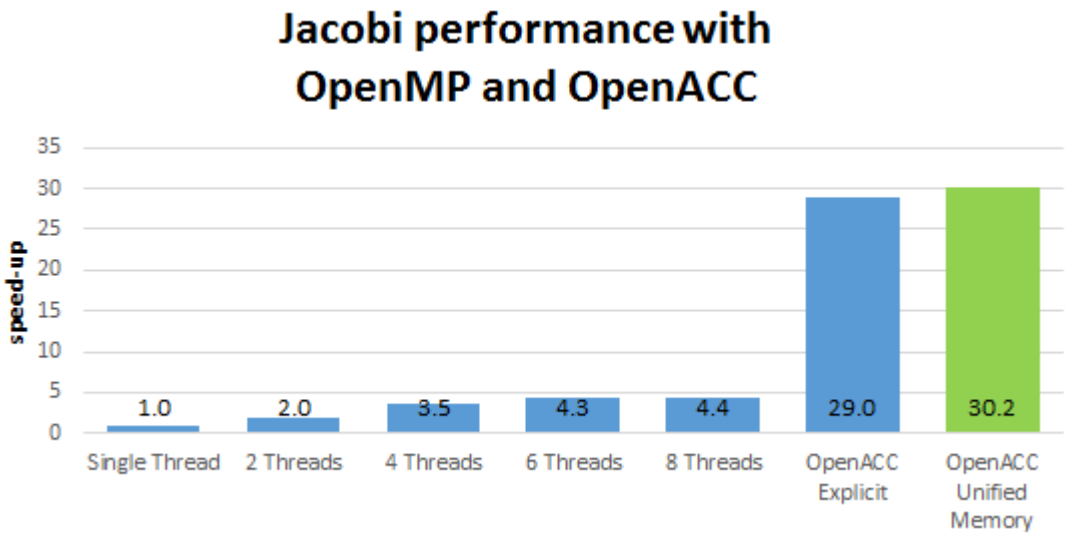
```
$ pgc++ -acc -ta=tesla:managed -Minfo=all test.cpp
```

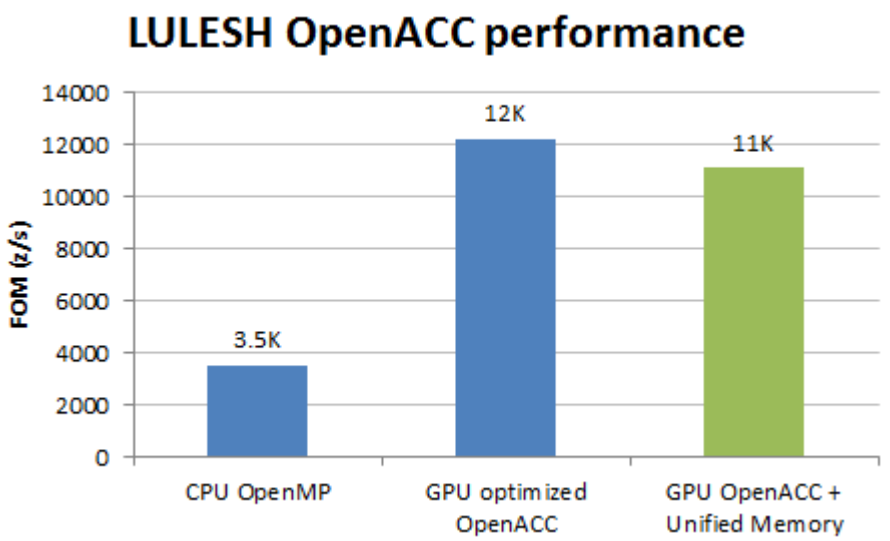
Also more efficient: only a single 4KB page is copied back from GPU

Focus on parallel loops instead of memory management

OPENACC + UNIFIED MEMORY

Learn more





S6134 - High Performance and Productivity with Unified Memory and OpenACC

Thursday @ 10:00, Marriott Salon 1

http://devblogs.nvidia.com/parallelforall/combine-openacc-unified-memory-productivity-performance/

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

THANK YOU

JOIN THE CONVERSATION
#GTC16

L6126 - Tips and Tricks for Unified Memory on NVIDIA Kepler and Maxwell
Wednesday @ 9:30, Room 210C

S6810 - Optimizing application performance with CUDA® profiling tools
Thursday @ 10:00, Room 211B