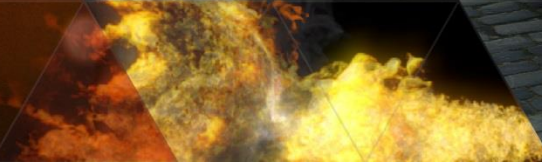




# UNIFIED MEMORY ON P100

Peng Wang

HPC Developer Technology, NVIDIA



# OVERVIEW

How UM works in P100

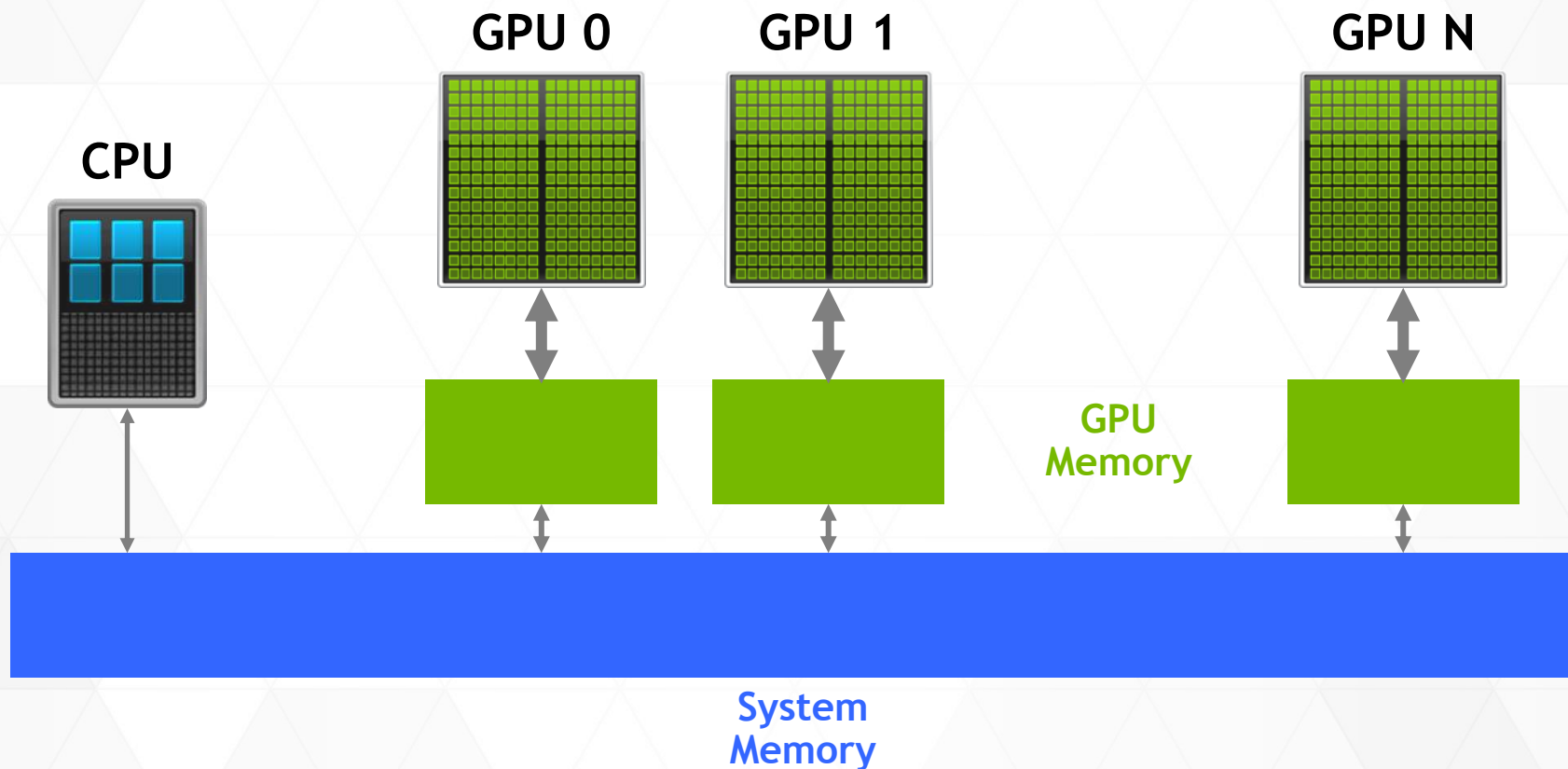
UM optimization

UM and directives

# HOW UNIFIED MEMORY WORKS IN P100

# HETEROGENEOUS ARCHITECTURES

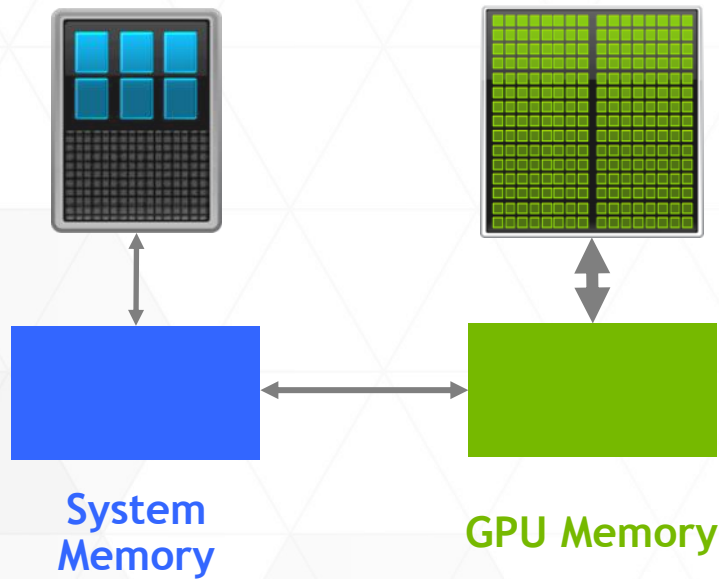
## Memory hierarchy



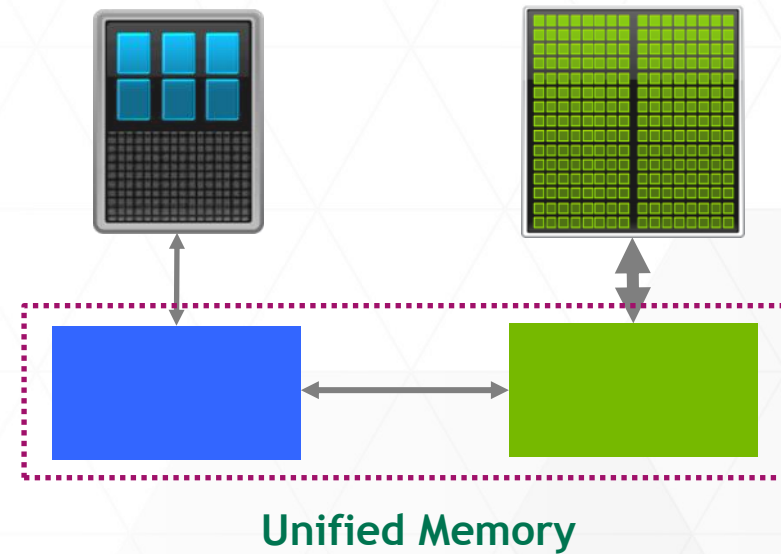
# UNIFIED MEMORY

Starting with Kepler and CUDA 6

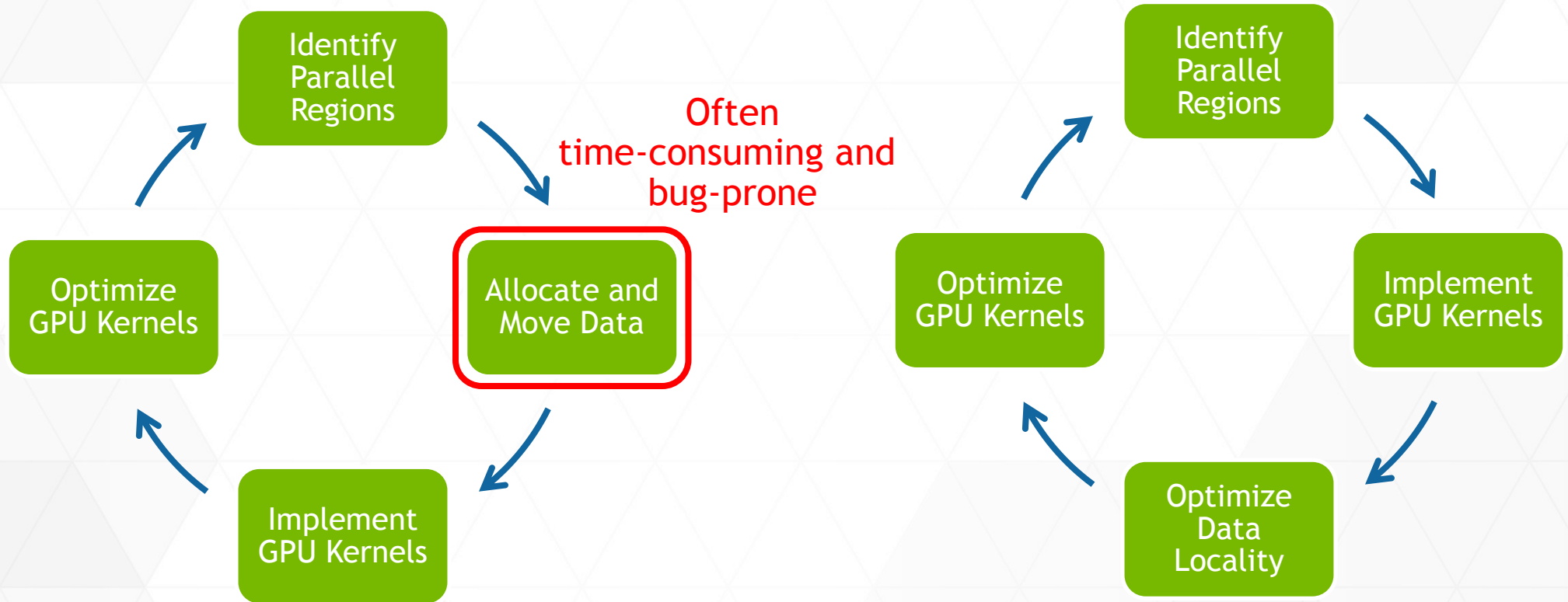
## Custom Data Management



## Developer View With Unified Memory



# UNIFIED MEMORY IMPROVES DEV CYCLE





# 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

<code>cudaMallocManaged(&amp;ptr, ...);</code>	← Pages are populated in GPU memory
<code>*ptr = 1;</code>	← CPU page fault: data migrates to CPU
<code>qsort&lt;&lt;&lt;...&gt;&gt;&gt;(ptr);</code>	← <b>Kernel launch</b> : data migrates to GPU

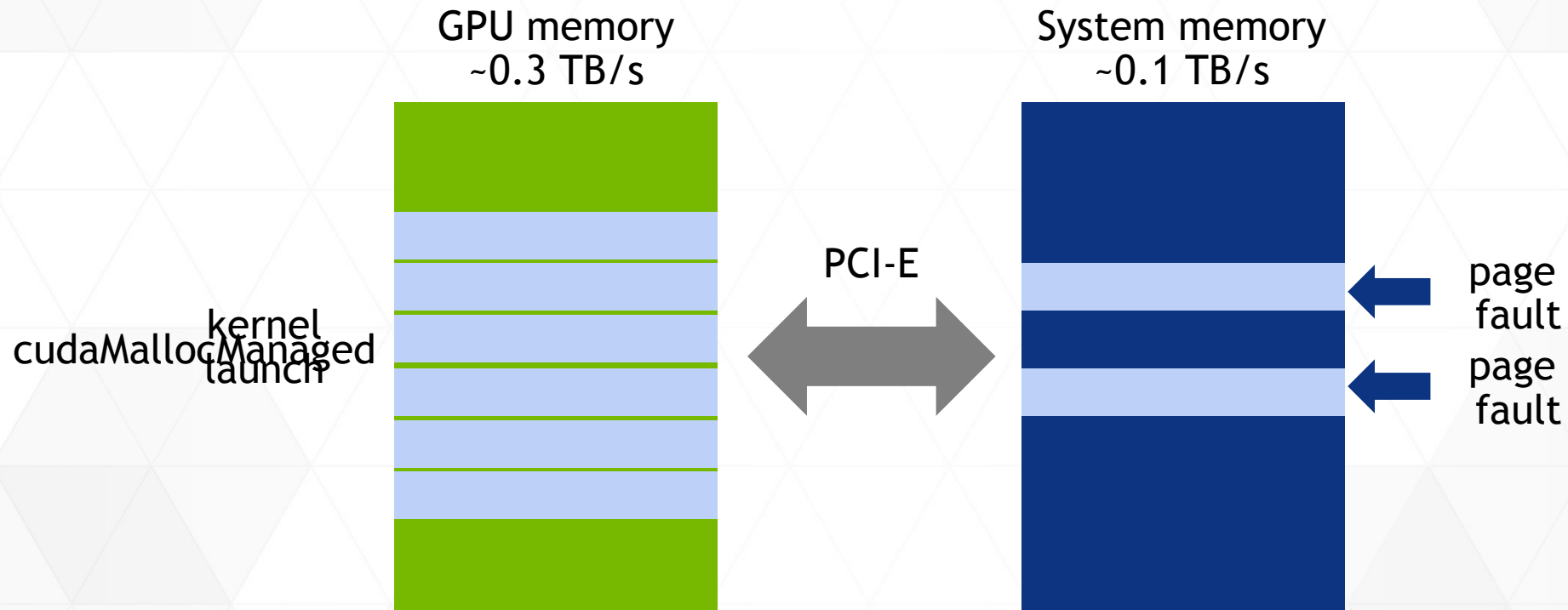
Pages allocated **before** they are used - **cannot oversubscribe GPU**

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



# UNIFIED MEMORY ON PRE-PASCAL

Kernel launch triggers bulk page migrations



# PAGE MIGRATION ENGINE

## Support Virtual Memory Demand Paging

### 49-bit Virtual Addresses

Sufficient to cover 48-bit CPU address + all GPU memory

### GPU page faulting capability

Can handle thousands of simultaneous page faults

### Up to 2 MB page size

Better TLB coverage of GPU memory

# UNIFIED MEMORY ON PASCAL

Now supports GPU page faults

<code>cudaMallocManaged(&amp;ptr, ...);</code>	← Empty, no pages anywhere (similar to malloc)
<code>*ptr = 1;</code>	← CPU page fault: data allocates on CPU
<code>qsort&lt;&lt;&lt;...&gt;&gt;&gt;(ptr);</code>	← GPU page fault: data migrates to GPU

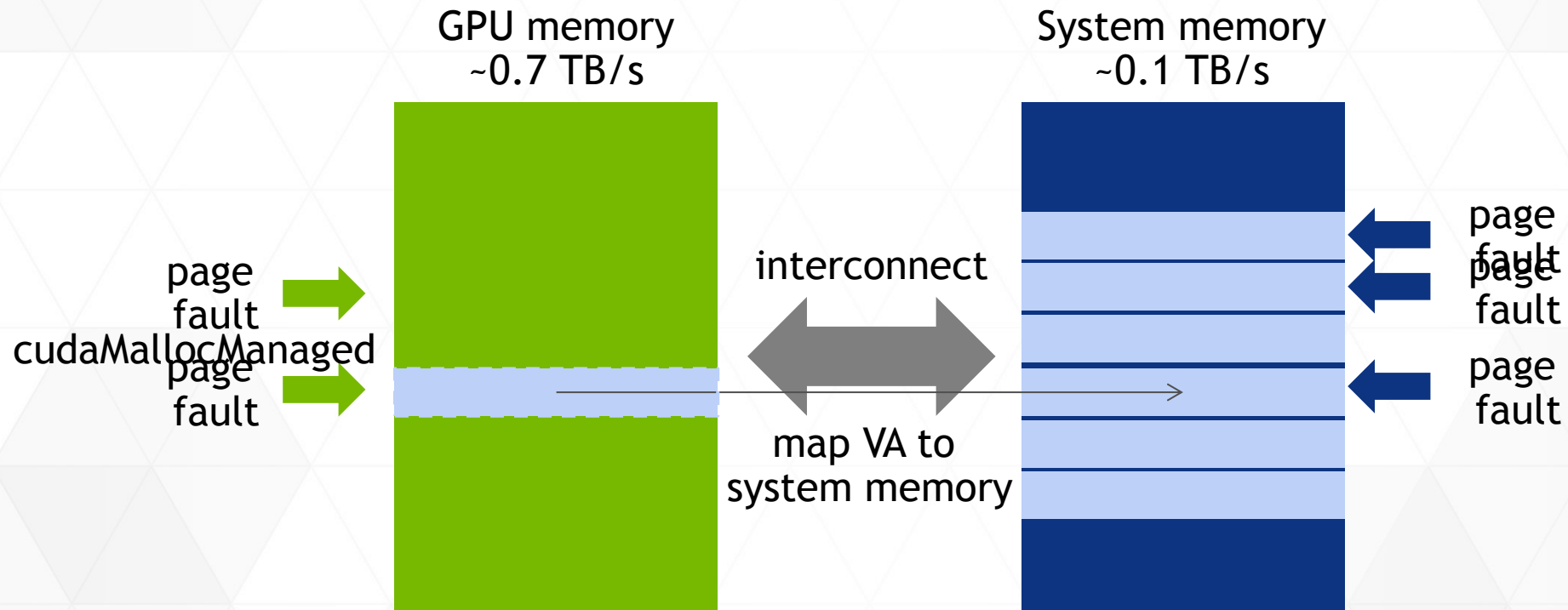
If GPU does not have a VA translation, it issues an interrupt to CPU

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

# UM USE CASE: HPGMG

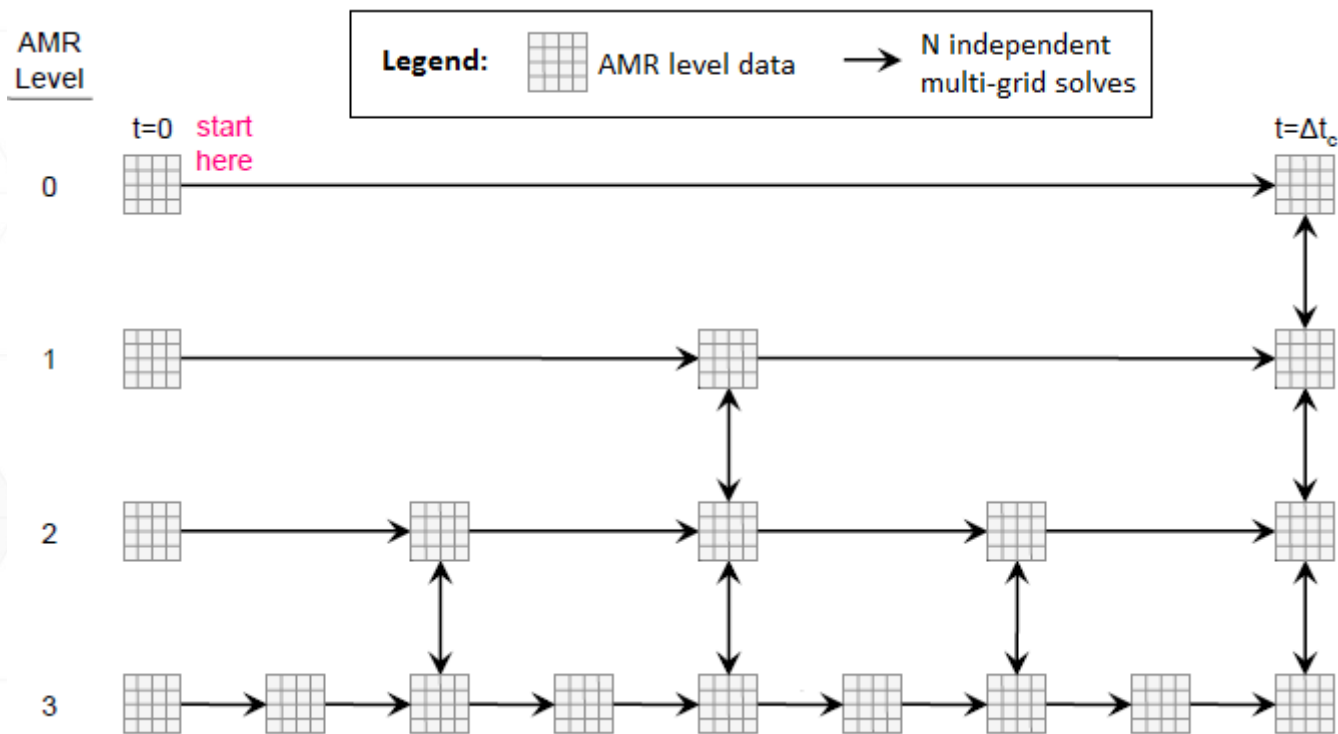
Fine grids are offloaded to GPU (TOC), coarse grids are processed on CPU (LOC)



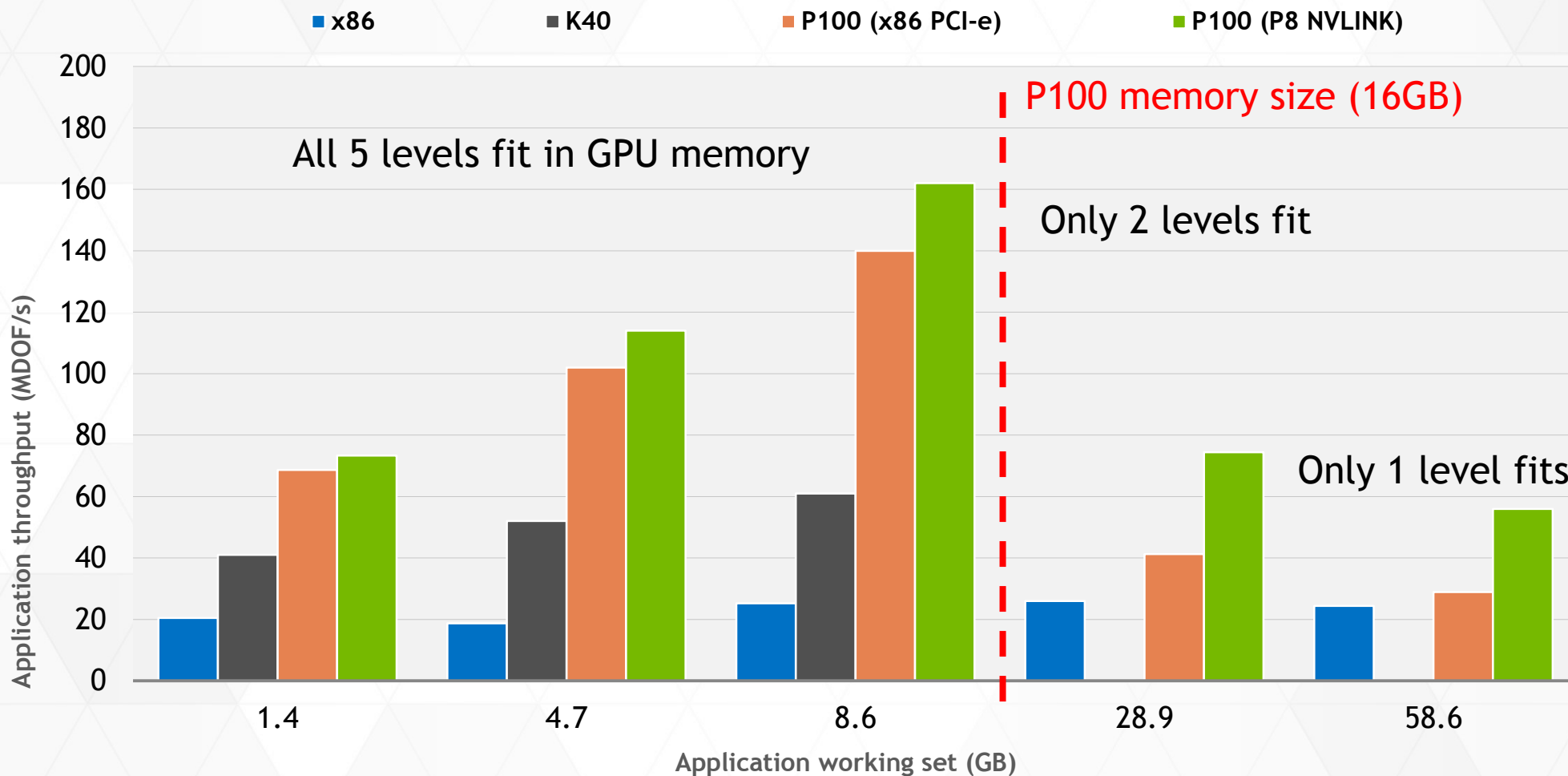


# HPGMG AMR PROXY

Data locality and reuse of AMR levels



# OVERSUBSCRIPTION RESULTS



# 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 (both CPU-GPU and GPU-GPU)

PCI-E will have software-assisted atomics (only CPU-GPU)

# UNIFIED MEMORY ATOMICS

## System-Wide Atomics

```
__global__ void mykernel(int *addr) {  
    atomicAdd_system(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

# 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

# UNIFIED MEMORY OPTIMIZATION



# GENERAL GUIDELINES

## UM overhead

**Migration:** move the data, limited by CPU-GPU interconnect bandwidth

**Page fault:** update page table, ~10s of  $\mu$ s per page, while execution stalls.

Solution: prefetch

## Redundant transfer for read-only data

Solution: duplication

## Thrashing: infrequent access, migration overhead can exceed locality benefits

Solution: mapping

# NEW HINTS API IN CUDA 8

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

Migrate data to destDevice: overlap with compute

Update page table: much lower overhead than page fault in kernel

Async operation that follows CUDA stream semantics

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

Specifies allocation and usage policy for memory region

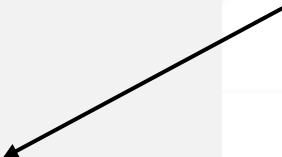
User can set and unset at any time

# PREFETCH


## Simple code example

```
void foo(cudaStream_t s) {  
    char *data;  
    cudaMallocManaged(&data, N);  
  
    init_data(data, N);  
  
    cudaMemPrefetchAsync(data, N, myGpuId, s);  
    // potentially other compute ...  
    mykernel<<<..., s>>>(data, N, 1, compare);  
    cudaMemPrefetchAsync(data, N, cudaCpuDeviceId, s);  
    // potentially other compute ...  
    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



# PREFETCH EXPERIMENT

```
__global__ void inc(float *a, int n)
{
    int gid = blockIdx.x*blockDim.x + threadIdx.x;
    a[gid] += 1;
}
```

```
cudaMallocManaged(&a, n*sizeof(float));
for (int i = 0; i < n; i++)
    a[i] = 1;
timer.Start();
cudaMemPrefetchAsync(a, n*sizeof(float), 0, NULL);
inc<<<grid, block>>>(a, n);
cudaDeviceSynchronize();
timer.Stop();
```

n=128M

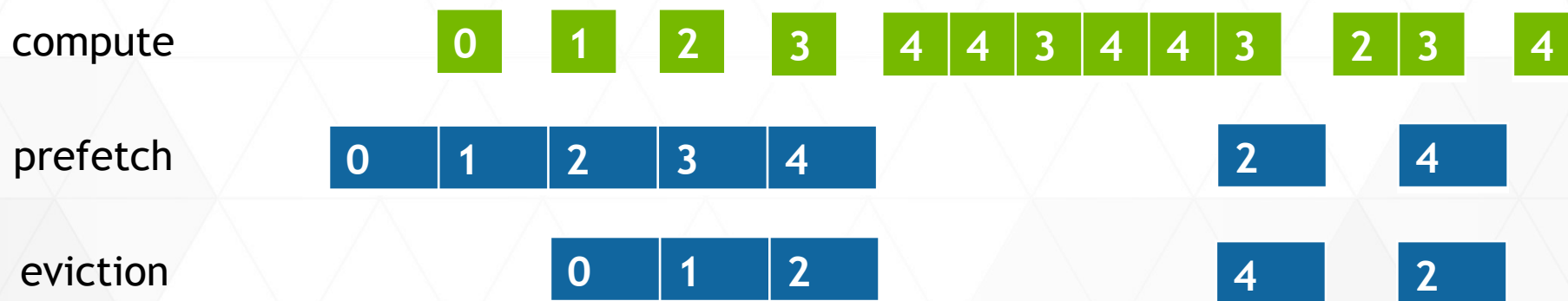
ms (GB/s)	Total time	Kernel time	Prefetch time
No prefetch	476 (2.3)	476 (2.3)	N/A
Prefetch	49 (22)	2 (536)	47 (11)

Page fault within kernel are expensive.

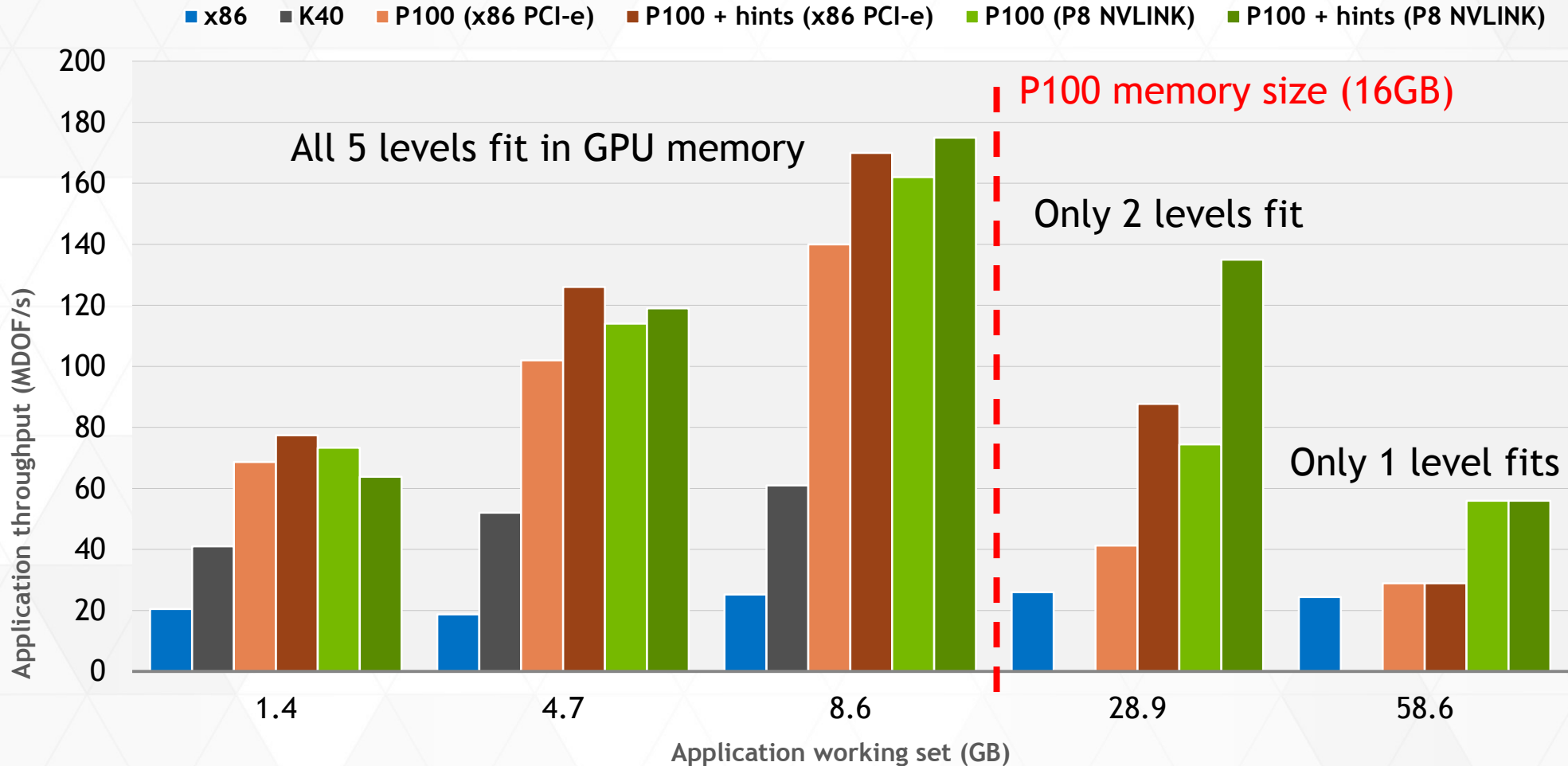
# HPGMG: DATA PREFETCHING

Prefetch next level while performing computations on current level

Use `cudaMemPrefetchAsync` with non-blocking stream to overlap with default stream



# RESULTS WITH USER HINTS





# 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

# DIRECT MAPPING

CUDA 8 performance hints (works with cudaMallocManaged)

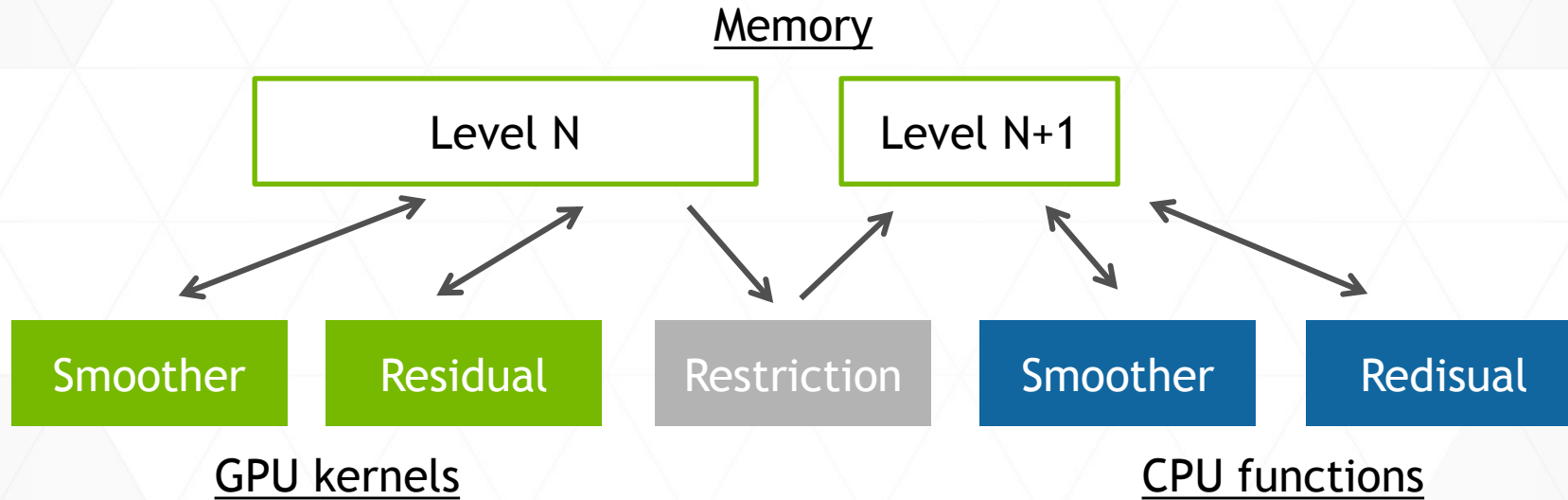
```
// set preferred location to CPU to avoid migrations
cudaMemAdvise(ptr, size, cudaMemAdviseSetPreferredLocation, cudaCpuDeviceId);

// 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);
```

# HYBRID IMPLEMENTATION

Data sharing between CPU and GPU



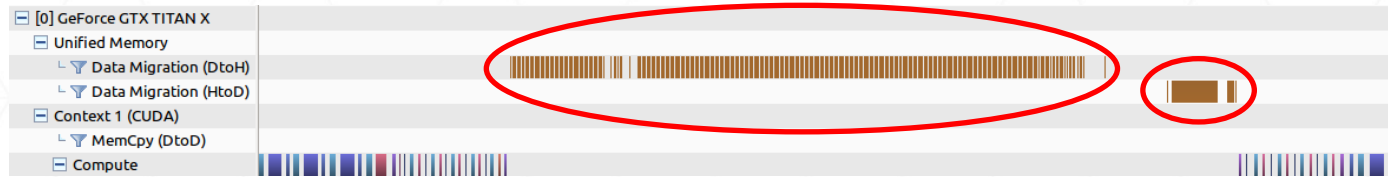
Level N+1 (small) is shared between CPU and GPU

To avoid frequent migrations allocate N+1: pin page in CPU memory

# DIRECT MAPPING

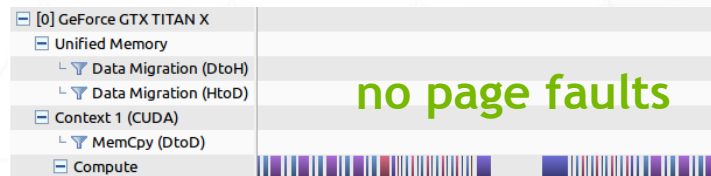
Use case: HPGMG

**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`



~5% performance improvement on Tesla P100



# MPI AND UNIFIED MEMORY

Using Unified Memory with CUDA-aware MPI needs explicit support from the MPI implementation:

- Check with your MPI implementation of choice for their support

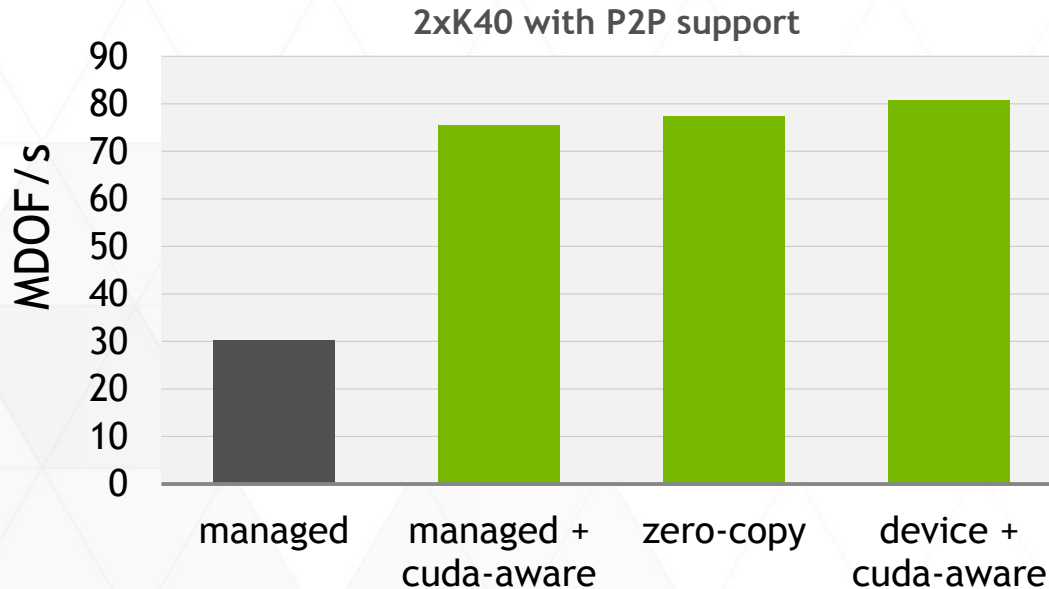
- Unified Memory is supported in OpenMPI since 1.8.5 and MVAPICH2-GDR since 2.2b

Set preferred location may help improve performance of CUDA-aware MPI using managed pointers

# MULTI-GPU PERFORMANCE: HPGMG

MPI buffers can be allocated with `cudaMalloc`, `cudaMallocHost`, `cudaMallocManaged`

CUDA-aware MPI can stage managed buffers through system or device memory



# UNIFIED MEMORY AND DIRECTIVES

## W/o UM

```
module.F90:
module particle_array
real,dimension(:,,:),allocatable :: zelectron
!$acc declare create(zelectron)

setup.F90:
allocate(zelectron(6,me))
call init(zelectron)
!$acc update device(zelectron)

pushe.F90:
real mpgc(4,me)
!$acc declare create(mpgc)
!$acc parallel loop
do i=1,me
    zelectron(1,m)=mpgc(1,m)
enddo
!$acc end parallel

!$acc update host(zelectron)
call foo(zelectron)
```

## W/ UM

```
module.F90:
module particle_array
real,dimension(:,,:),allocatable :: zelectron

setup.F90:
allocate(zelectron(6,me))
call init(zelectron)

pushe.F90:
real mpgc(4,me)
!$acc parallel loop
do i=1,me
    zelectron(1,me)=...
enddo
!$acc end parallel
call foo(zelectron)
```

Remove all data directives.  
Add “-ta=managed” to compile

# UM OPENACC USE CASE: GTC

Plasma code for fusion science

10X slowdown in key compute routine when turning on Unified Memory

Problem: automatic array.

Allocated/deallocated EACH time entering the routine

Paying page fault cost EACH time entering the routine.

Even though no migration.

```
module.F90:
```

```
module particle_array  
real,dimension(:,:),allocatable :: zelectron
```

```
setup.F90:
```

```
allocate(zelectron(6,me))  
call init(zelectron)
```

```
pushe.F90:
```

```
real mpgc(4,me)  
!$acc parallel loop  
do i=m,me  
    zelectron(1,m)=mpgc(1,m)  
enddo  
!$acc end parallel  
call foo(zelectron)
```

# UNIFIED MEMORY SUMMARY

On-demand page migration

Performance overhead: migration, page fault

Optimization: prefetch, duplicate, mapping