GPU TECHNOLOGY CONFERENCE

April 4-7, 2016 | Silicon Valley

THE FUTURE OF UNIFIED MEMORY

Nikolay Sakharnykh, 4/5/2016

PRESENTED BY NVIDIA

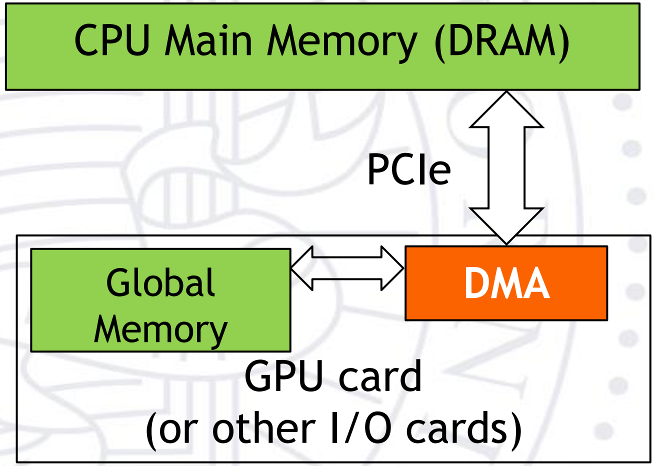
Logistics

- Haven't graded midterm yet, will be finished on Wednesday
- May 22nd – last day to drop without a W or change to S/NS with no fee or penalty
  - https://registrar.ucr.edu/resources/forms
- Lab 2 due Monday May 18th
- Lab 3 due Monday May 25th
- Lab 4 due Friday June 12th
- No lab 5
- Quiz 3 Wednesday May 27th
- Quiz 4 will be a "take home quiz" where it will comprise of your 4 lowest scored questions over the previous 3 quizzes due Monday June 6th
- Final June 3rd or on finals week?

Pinned host memory

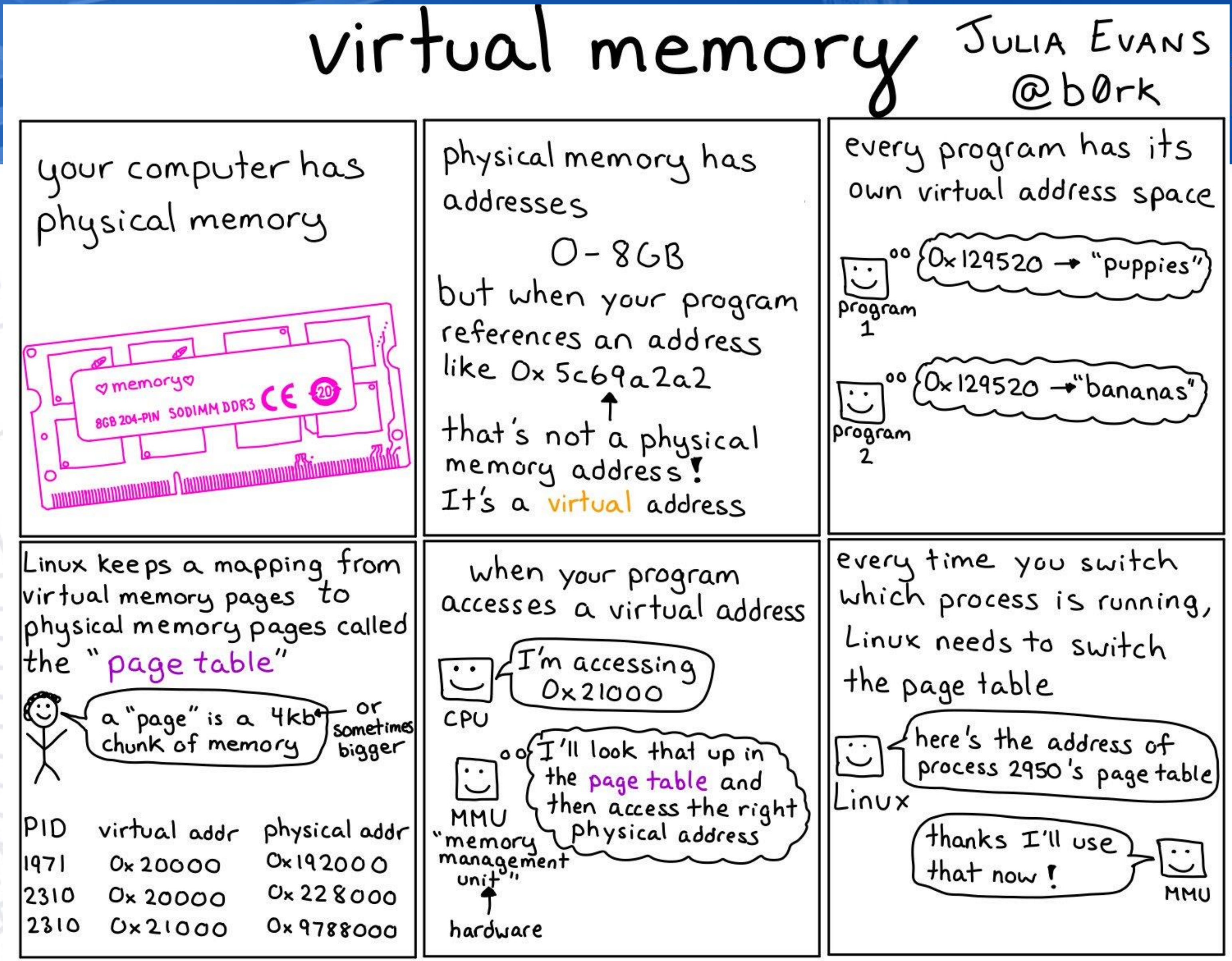
CPU-GPU Data Transfer using DMA

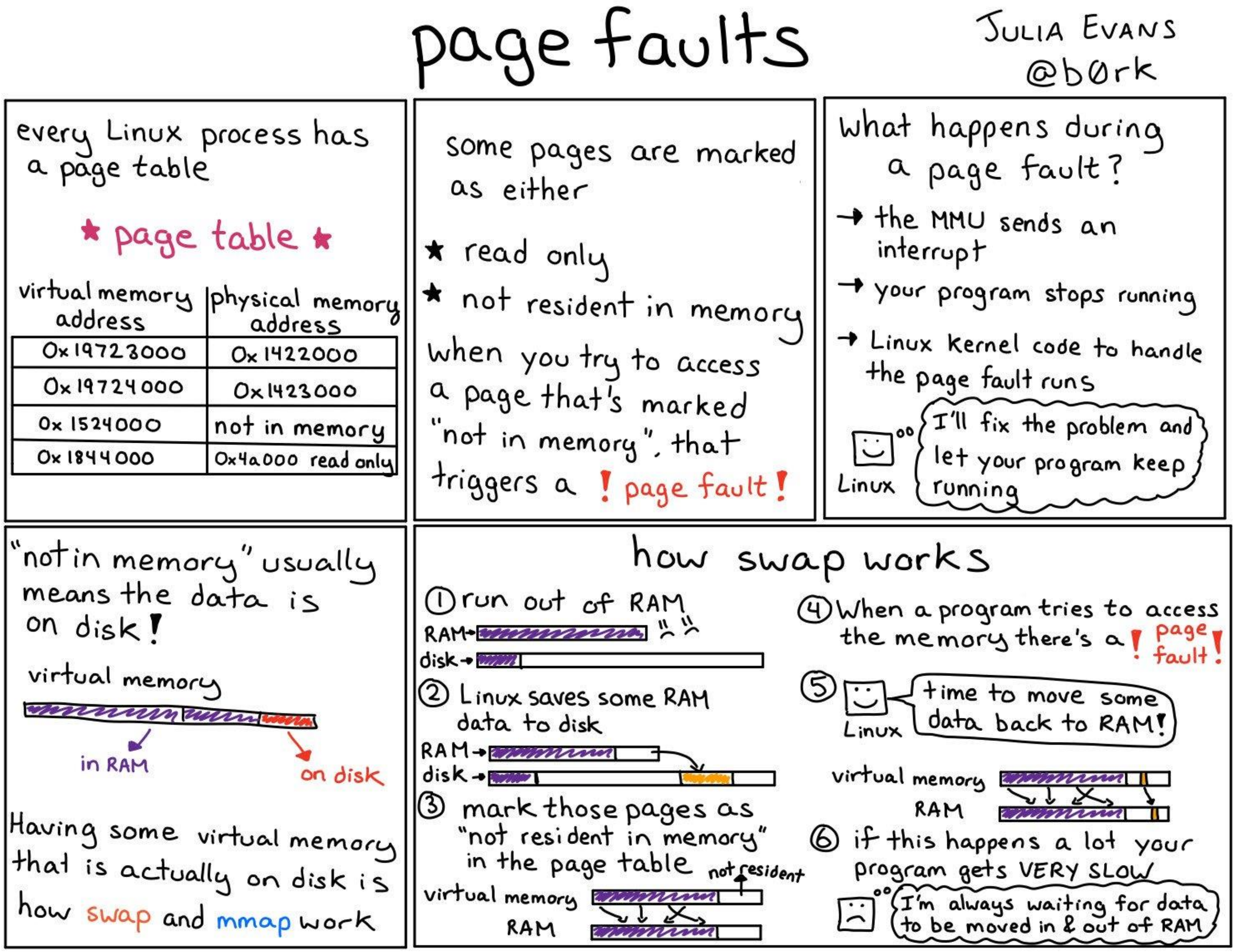
- DMA (Direct Memory Access) hardware is used by `cudaMemcpy()` for better efficiency
  - Frees CPU for other tasks
  - Hardware unit specialized to transfer a number of bytes requested by OS
  - Between physical memory address space regions (some can be mapped I/O memory locations)
  - Uses system interconnect, typically PCIe in today's systems



Virtual Memory Management

- Modern computers use virtual memory management
  - Many virtual memory spaces mapped into a single physical memory
  - Virtual addresses (pointer values) are translated into physical addresses
- Not all variables and data structures are always in the physical memory
  - Each virtual address space is divided into pages that are mapped into and out of the physical memory
  - Virtual memory pages can be mapped out of the physical memory (page-out) to make room
  - Whether or not a variable is in the physical memory is checked at address translation time


virtual memory
JULIA EVANS @b0rk
your computer has physical memory
♡memory♡
8GB 204-PIN SODIMM DDR3
physical memory has addresses
0-8GB
but when your program references an address like 0x5c69a2a2
that's not a physical memory address!
It's a virtual address
every program has its own virtual address space
0x129520 → "puppies"
program 1
0x129520 → "bananas"
program 2
Linux keeps a mapping from virtual memory pages to physical memory pages called the "page table"
a "page" is a 4kb chunk of memory
or sometimes bigger
PID virtual addr physical addr
1971 0x20000 0x192000
2310 0x20000 0x228000
2310 0x21000 0x9788000
when your program accesses a virtual address
I'm accessing 0x21000
CPU
I'll look that up in the page table and then access the right physical address
MMU
"memory management unit"
hardware
every time you switch which process is running, Linux needs to switch the page table
here's the address of process 2950's page table
Linux
thanks I'll use that now!
MMU



page faults
JULIA EVANS
@b0rk
every Linux process has a page table
★ page table ★
virtual memory address | physical memory address
0x19723000 | 0x1422000
0x19724000 | 0x1423000
0x1524000 | not in memory
0x1844000 | 0x4a000 read only
some pages are marked as either
★ read only
★ not resident in memory
when you try to access a page that's marked "not in memory", that triggers a ! page fault !
what happens during a page fault?
→ the MMU sends an interrupt
→ your program stops running
→ Linux kernel code to handle the page fault runs
I'll fix the problem and let your program keep running
Linux
"not in memory" usually means the data is on disk!
virtual memory
in RAM
on disk
Having some virtual memory that is actually on disk is how swap and mmap work
how swap works
① run out of RAM
RAM→
disk→
② Linux saves some RAM data to disk
RAM→
disk→
③ mark those pages as "not resident in memory" in the page table
not resident
virtual memory
RAM
④ When a program tries to access the memory there's a ! page fault !
⑤ time to move some data back to RAM!
Linux
virtual memory
RAM
⑥ if this happens a lot your program gets VERY SLOW
I'm always waiting for data to be moved in & out of RAM

Data Transfer and Virtual Memory

- DMA uses physical addresses
  - When `cudaMemcpy()` copies an array, it is implemented as one or more DMA transfers
  - Address is translated and page presence checked for the entire source and destination regions at the beginning of each DMA transfer
  - No address translation for the rest of the same DMA transfer so that high efficiency can be achieved

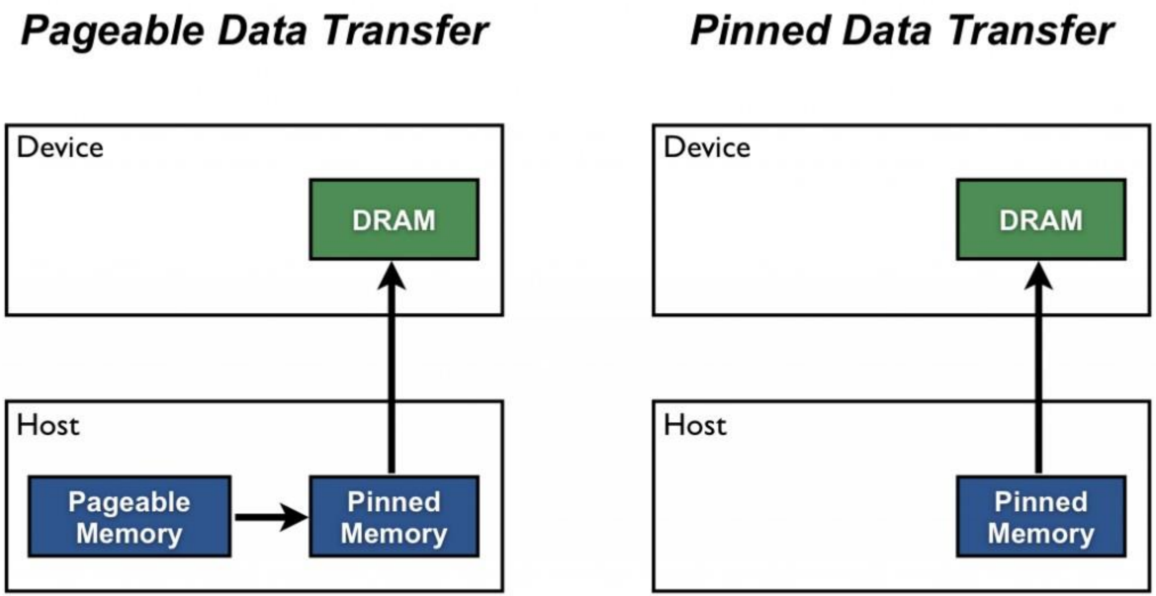
- The OS could accidentally page-out the data that is being read or written by a DMA and page-in another virtual page into the same physical location

Pinned Memory and DMA Data Transfer

- Pinned memory are virtual memory pages that are specially marked so that they cannot be paged out
- Allocated with a special system API function call
- a.k.a. Page Locked Memory, Locked Pages, etc.
- CPU memory that serve as the source or destination of a DMA transfer must be allocated as pinned memory

CUDA data transfer uses pinned memory.

- The DMA used by cudaMemcpy() requires that any source or destination in the host memory is allocated as pinned memory
- If a source or destination of a `cudaMemcpy()` in the host memory is not allocated in pinned memory, it needs to be first copied to a pinned memory – extra overhead
- `cudaMemcpy()` is faster if the host memory source or destination is allocated in pinned memory since no extra copy is needed



Allocate/Free Pinned Memory

- `cudaHostAlloc()`, three parameters
  - Address of pointer to the allocated memory
  - Size of the allocated memory in bytes
  - Option – use `cudaHostAllocDefault` for now

- `cudaFreeHost()`, one parameter
  - Pointer to the memory to be freed

Putting It Together - Vector Addition Host Code Example

```
int main()
{
   float *h_A, *h_B, *h_C;
…
   cudaHostAlloc((void **) &h_A, N* sizeof(float),
      cudaHostAllocDefault);
   cudaHostAlloc((void **) &h_B, N* sizeof(float),
      cudaHostAllocDefault);
   cudaHostAlloc((void **) &h_C, N* sizeof(float),
      cudaHostAllocDefault);
…
   // cudaMemcpy() runs 2X faster
}
```

Using Pinned Memory in CUDA

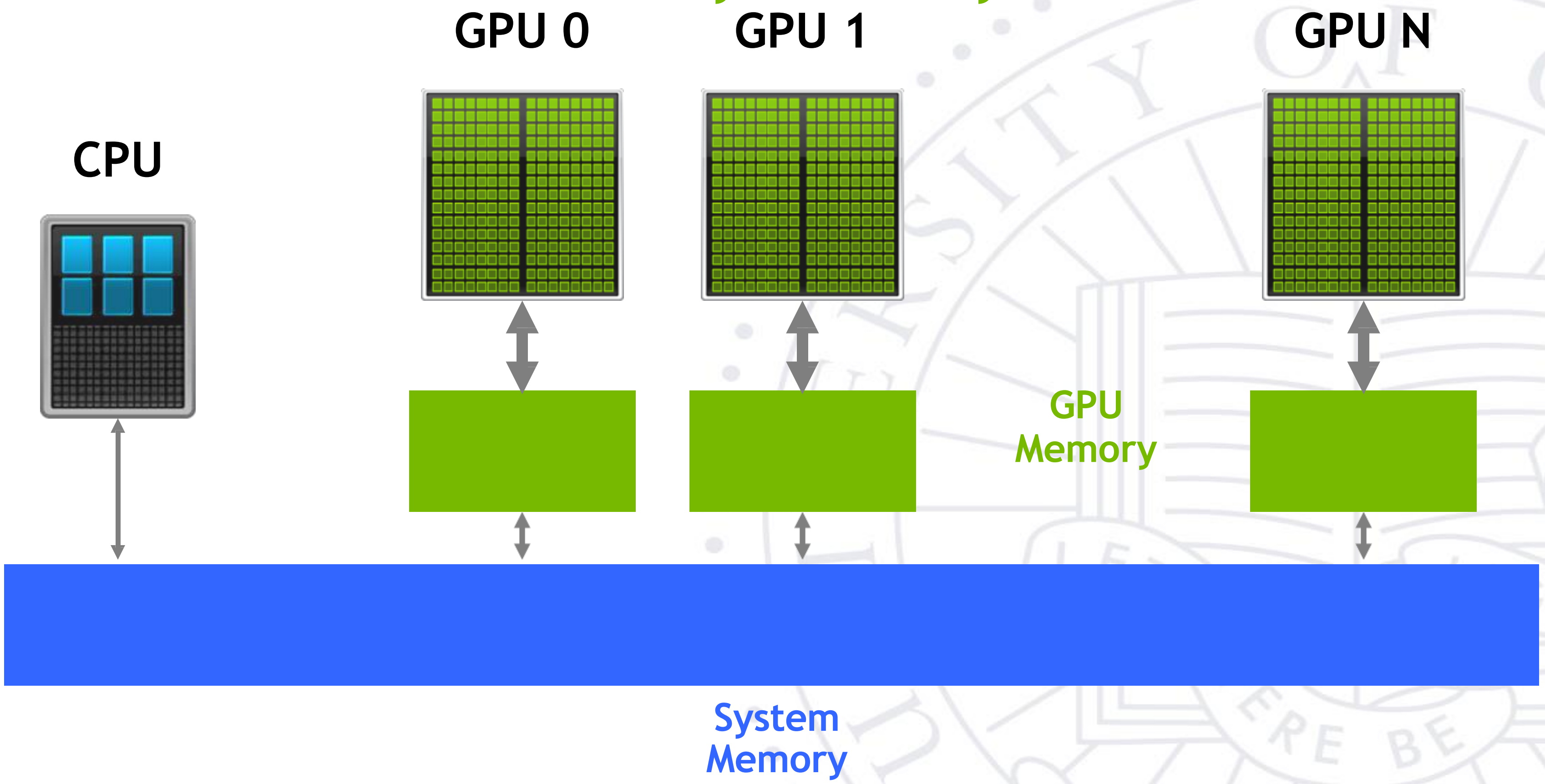
- Use the allocated pinned memory and its pointer the same way as those returned by `malloc();`
- The only difference is that the allocated memory cannot be paged by the OS
- The `cudaMemcpy()` function should be about 2X faster with pinned memory
- Pinned memory is a limited resource
  - over-subscription can have serious consequences

Why is pinned memory a limited resource? What might be the consequences of over-subscription?

Unified Memory

HETEROGENEOUS ARCHITECTURES

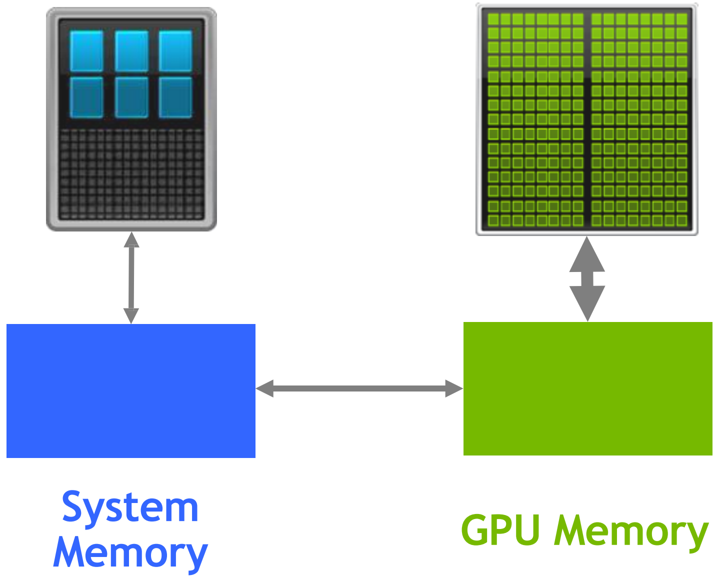
Memory hierarchy



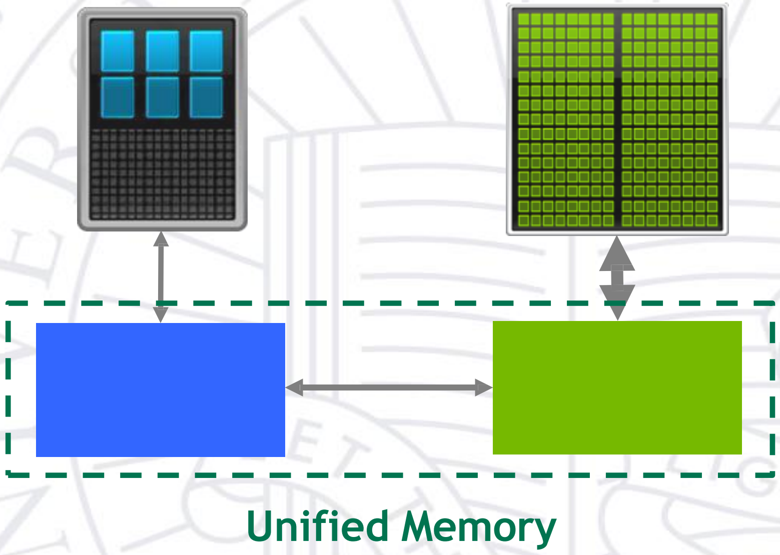
UNIFIED MEMORY

Starting with Kepler and CUDA 6

Custom Data Management



Developer View With Unified Memory



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(&ptr, ...);`	← Pages are populated in GPU memory
`*ptr = 1;`	← **CPU page fault:** data migrates to CPU
`qsort<<<...>>>(ptr);`	← **Kernel launch**: data migrates to GPU

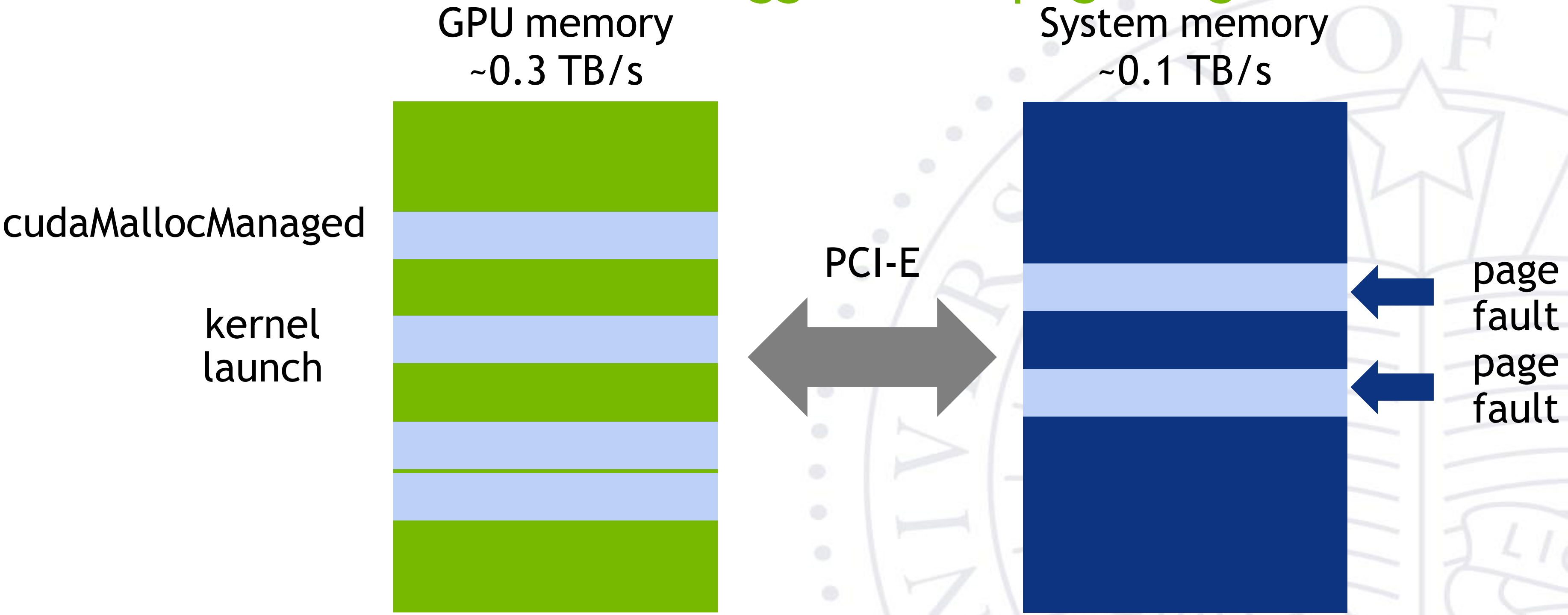
GPU always has address translation during the kernel execution

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);
```

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

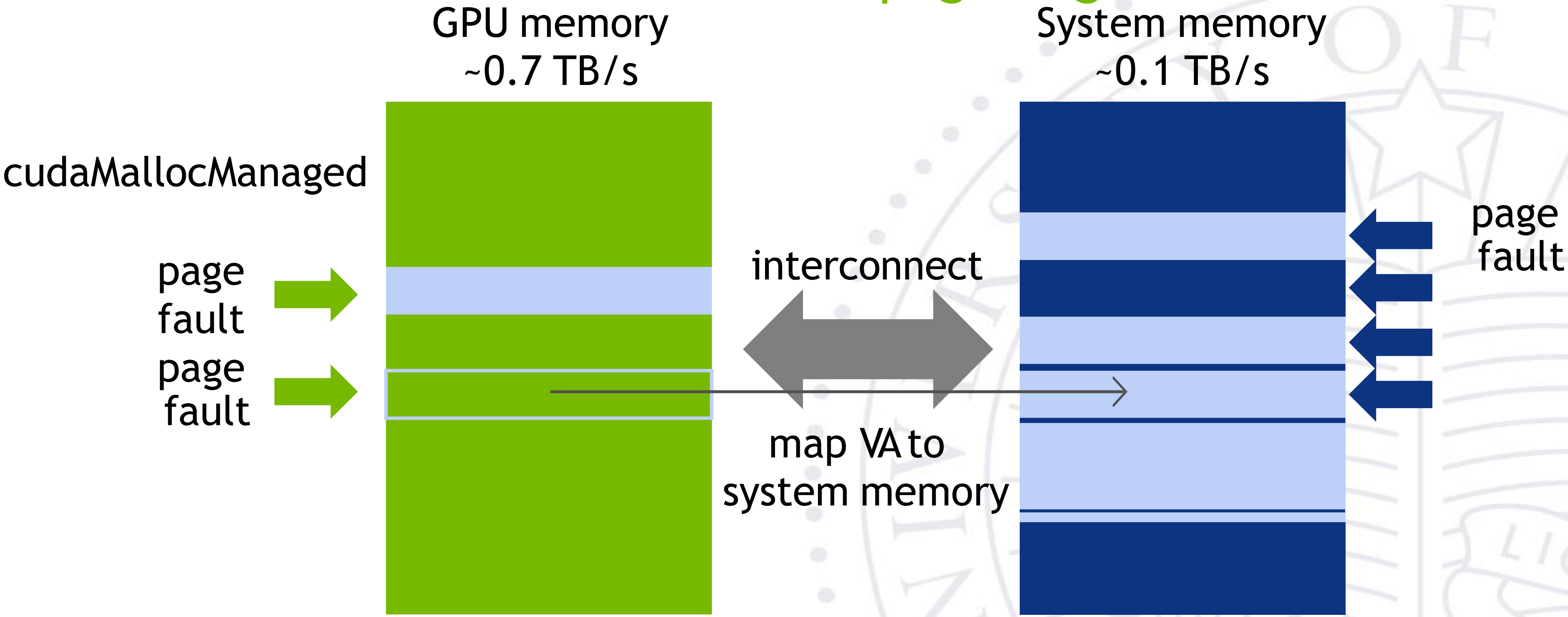
If GPU does not have a VA translation, it issues an interrupt to CPU

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

When poll is active, respond at **PollEv.com/marcuschow119**

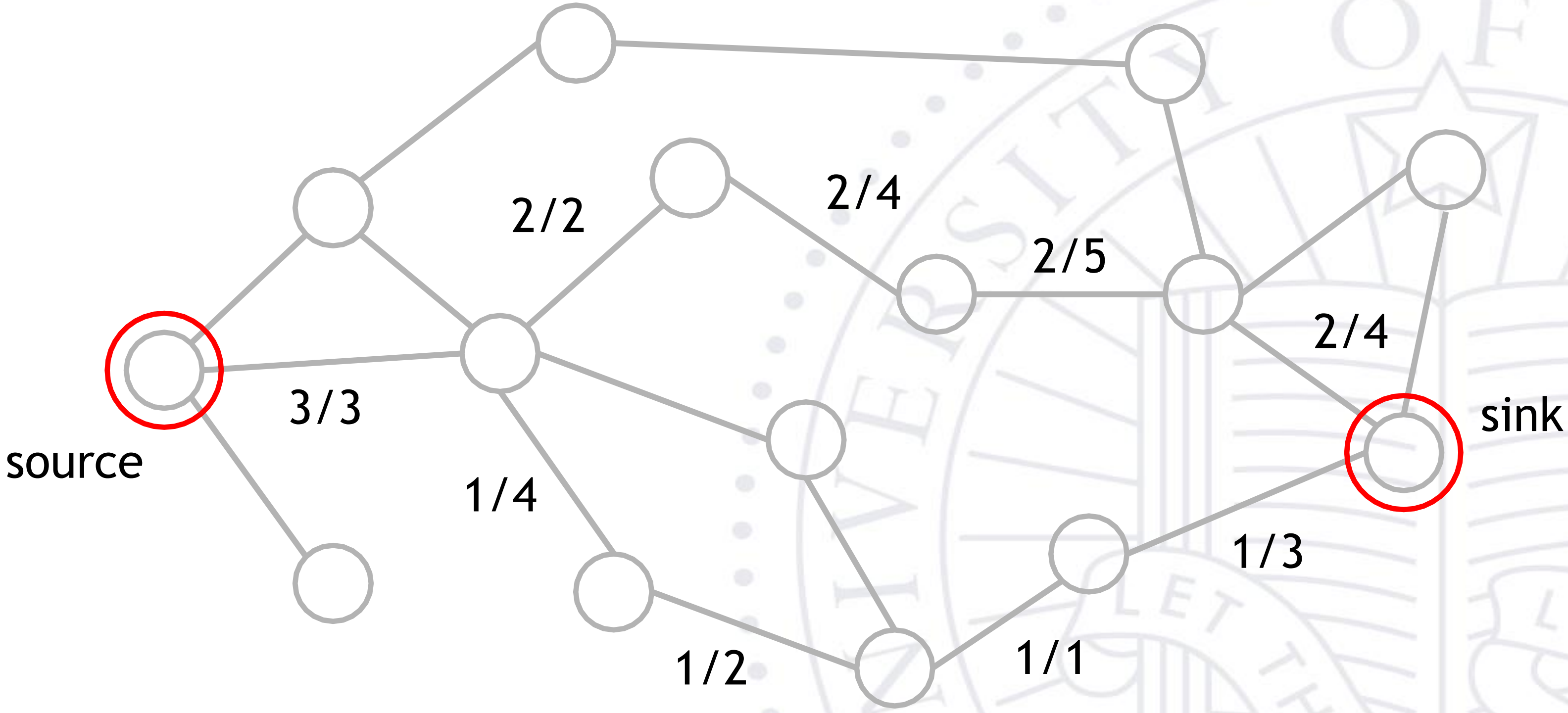
Is unified memory different than pinned memory? Why or why not?

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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:

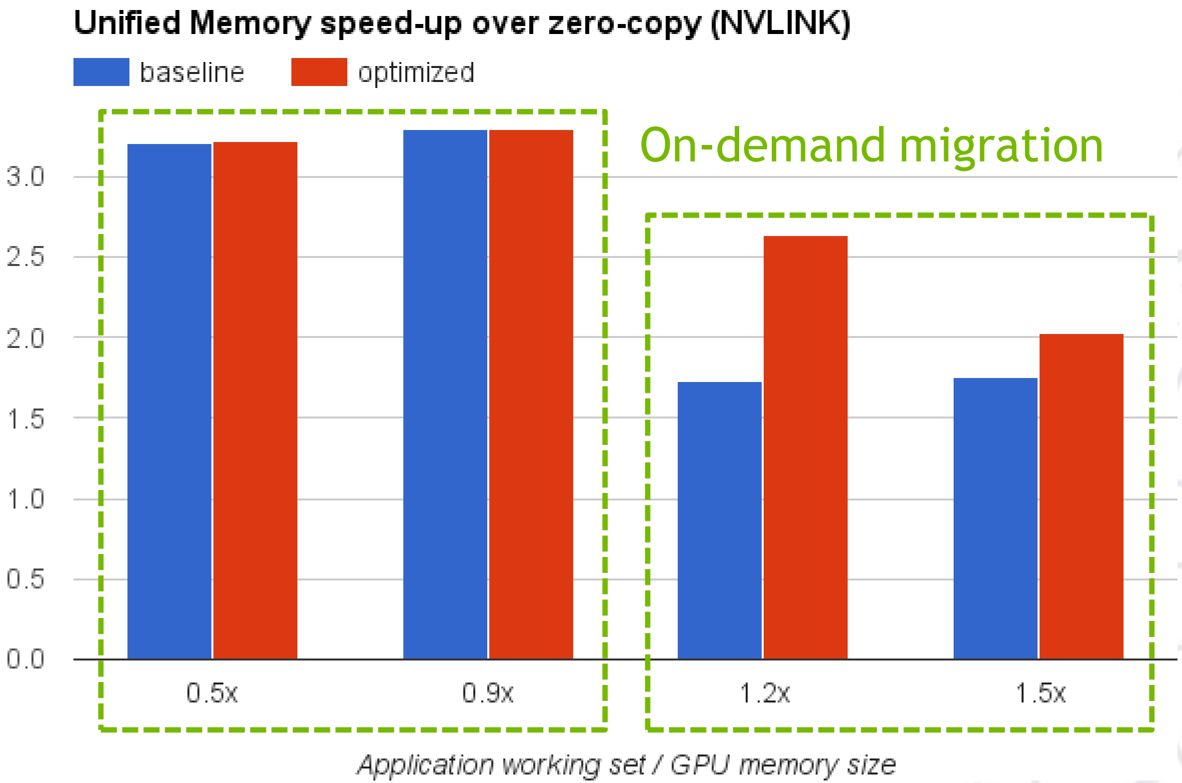
Both CPU and GPU bring only necessary vertices/edges on-demand

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 memory oversubscription

GPU OVERSUBSCRIPTION

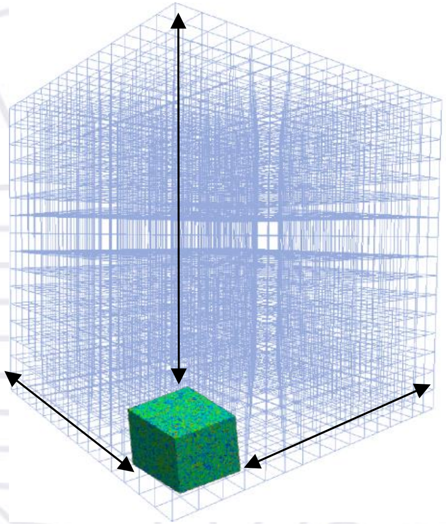
Now possible with Pascal

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!

ON-DEMAND ALLOCATION

Dynamic queues

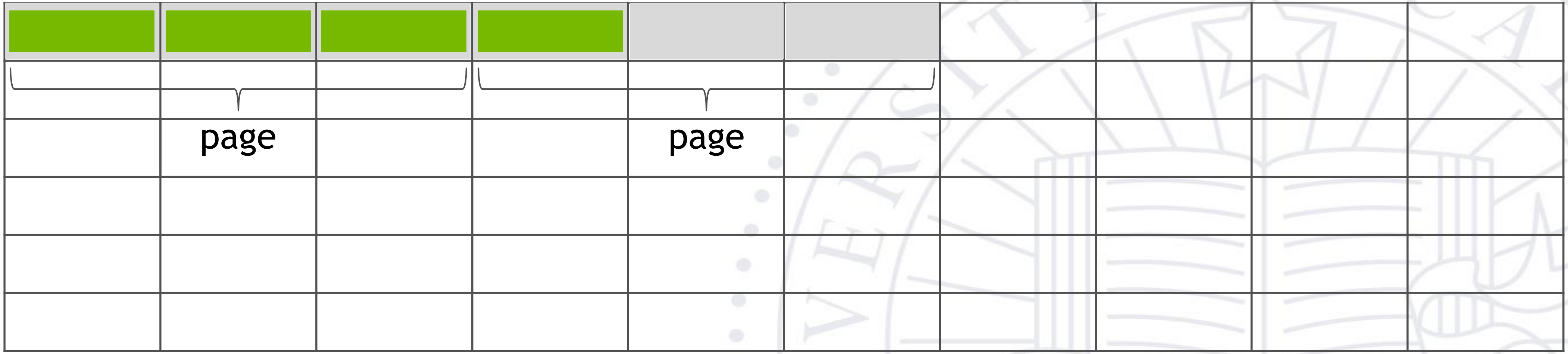
Problem: GPU populates queues with unknown size, need to overallocate

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

← CPU reads will not page fault (use_data(data, N);)

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);
```

CPU and GPU reads will not fault

- Read-only copy will be created during prefetch

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

When poll is active, respond at **PollEv.com/marcuschow119**

In Unified Memory, When would explicit copying would provide a benefit to your program? When would you not do that?