THE FUTURE OF UNIFIED MEMORY
Nikolay Sakharnykh, 4/5/2016
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. 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.

Linux keeps a mapping from virtual memory pages to physical memory pages called the "page table." A "page" is a 4KB chunk of memory.

<table>
<thead>
<tr>
<th>PID</th>
<th>virtual addr</th>
<th>physical addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1971</td>
<td>0x20000</td>
<td>0x192000</td>
</tr>
<tr>
<td>2310</td>
<td>0x20000</td>
<td>0x228000</td>
</tr>
<tr>
<td>2310</td>
<td>0x21000</td>
<td>0x9788000</td>
</tr>
</tbody>
</table>

When your program accesses a virtual address, it's accessing 0x21000.

I'll look that up in the page table and then access the right physical address.

Every program has its own virtual address space. 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, thanks I'll use that now!
**Page Faults**

Every Linux process has a page table.

- **Page Table**
  - Virtual Memory Address | Physical Memory Address
  - 0x1972000 | 0x142000
  - 0x19724000 | 0x1423000
  - 0x1520000 | Not in Memory
  - 0x1840000 | Not in Memory, 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.

"Not in memory" usually means the data is on disk!

Virtual memory

Having some virtual memory that is actually on disk is how swap and mmap work.

**How Swap Works**

1. Run out of RAM
2. Linux saves some RAM data to disk
3. Mark those pages as "not resident in memory" in the page table
4. When a program tries to access the memory there's a page fault!
5. Time to move some data back to 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
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?

https://pollev.com/abbasmazloum123
Unified Memory
HETEROGENEOUS ARCHITECTURES

Memory hierarchy

CPU

GPU 0

GPU 1

GPU N

System Memory

GPU Memory
UNIFIED MEMORY
Starting with Kepler and CUDA 6

Custom Data Management

System Memory

GPU Memory

Developer View With Unified Memory

Unified Memory
UNIFIED MEMORY
Single pointer for CPU and GPU

- CPU code

```c
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

```c
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);
}
```
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

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
UNIFIED MEMORY ON PRE-PASCAL

Kernel launch triggers bulk page migrations

- GPU memory: ~0.3 TB/s
- System memory: ~0.1 TB/s

CUDA & Page Faults

- cudaMallocManaged
- kernel launch
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

GPU memory
~0.7 TB/s

System memory
~0.1 TB/s

CUDAmallocManaged

page fault
page fault

interconnect
map VA to system memory

page fault
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
}
```

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

Unified Memory speed-up over zero-copy (NVLINK)
- **Baseline**: migrate on first touch
- **Optimized**: developer assists with hints for best placement in memory

On-demand migration

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

Application working set / GPU memory size
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

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here only 35% of memory is actually used!

Solution: use Unified Memory for allocations (on Pascal)
ON-DEMAND ALLOCATION

Dynamic queues
Memory is allocated on-demand so we don’t waste resources

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>page</td>
<td></td>
<td>page</td>
<td></td>
</tr>
</tbody>
</table>

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

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

    init_data(data, N);

    cudaMemcpyAsync(data, N, myGpuId, s);
    mykernel<<<..., s>>>(data, N, 1, compare);
    cudaMemcpyAsync(data, N, cudaCpuDeviceId, s);
    cudaMemcpy(data, N, 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

```c
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

```c
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-only copy will be
DIRECT MAPPING
Preferred location and direct access

cudaMemAdviseSet PreferredLocation

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**
```c
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**
```c
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

• Unified Memory allows the programmer to allocate a single pointer for the host and device memory.
• Allocate with cudaMemcpyManaged(&ptr, size);
• Copies are handled at the driver level through page faulting mechanism
• Unified memory enables efficient oversubscription of the GPU