Virtual memory in Accelerators

046278
Accelerators and Accelerated Systems
Mark Silberstein
• Virtual memory 101: replay from the OS
• Multiple address spaces
• Globally shared virtual memory
• Hardware page faults and page migration
• Accelerator-centric approach
Virtual Memory 101

• Why?
• VM: pages, page tables, page cache, swapping, page faults, page walks, MMU, TLB
• Mmap example
What would happen here?

```c
__global__ void foo(int* ptr){
    ptr++;
    *ptr=1;
}

__host__ void run_foo(){
    int h_ptr=malloc(4K);
    int g_ptr=cudaMalloc(4K);

    h_ptr++; *h_ptr=0;
    g_ptr++; *g_ptr=0;

    foo<<<>>>(h_ptr);
    foo<<<>>>(g_ptr);
}```
Multiple address spaces

__global__ void foo(int* ptr){
    ptr++;
    *ptr=1;
}

__host__ void run_foo(){
    int h_ptr=malloc(4K);
    int g_ptr=cudaMalloc(4K);

    h_ptr++; *h_ptr=0;
    g_ptr++; *g_ptr=0;

    foo<<<>>>(h_ptr);
    foo<<<>>>(g_ptr);

crash
Inside the accelerator memory

Acc MMU, Page table

Acc memory

Acc physical

Accelerator

Acc Virtual
Example: GPUs
(prior to GPU page faults)

• GPU address space managed by the CPU driver
  – cudaMemcpy/cudaMalloc operate in GPU address space

• Cannot be modified while a GPU kernel is running
  – In particular, cannot be modified from a running GPU kernel
GPU access to CPU memory

Accelerator

Acc MMU, Page table

Acc memory

Bus physical

CPU physical

CPU Memory

PCIe
GPU access to CPU memory

- Accelerator
- Acc MMU, Page table
- Acc memory
- Acc physical
- Acc Virtual
- Bus physical
- CPU Memory
- PCIe
- CPU physical
GPU access to CPU memory

What if this memory gets swapped out??
GPU access to CPU memory

- `cudaHostAlloc(MAPPED)`
  - allocates CPU memory to make it accessible to the GPU, and pins it in CPU virtual address space

- `cudaGetDevicePointer()`
  - maps CPU memory into GPU address space
  - retrieves the **GPU address** for CPU pointer allocated via `cudaHostAlloc`
  - How does `cudaGetDevicePointer` work??
  - Actually, not necessary with CUDA Unified Virtual Addressing
    - Enables passing CPU pointers allocated via `cudaHostAlloc` to GPUs
    - Enables passing pointers of different GPUs if peerAccess is enabled
CPU access to GPU memory

- CPU Memory
- CPU MMU, Page table
- PCIe
- Mapping of PCIe BAR
- CPU
- CPU MMU, Page table
- Acc memory
- Acc MMU, Page table
- Accelerator
- CPU access to GPU memory
GPUDirectRDMA

- NVIDIA support for exposing GPU memory for third-party devices (and CPU)
  - AMD GPUs support similar functionality
- Requires high-end cards (TESLA)
- Enables exposing GPU memory on the BAR, and retrieving its GPU memory address
- Example: copy file to GPU memory by passing a CPU ptr to mapped GPU memory to read()
- More on that later in the course
Unified Memory

- Problem: can we get rid of explicitly copying data to accelerator memory?

- Wouldn't it be great:

```c
__global__ void foo(int* ptr) {
    ptr++;
    *ptr=1;
}

__host__ void run_foo() {
    int* u_ptr=unified_malloc(4K);

    u_ptr++; *u_ptr=0;

    foo<<<>>>(u_ptr);
}
```
Unified memory: try #1

• Why not?

    __global__ void foo(int* ptr){
        ptr++;
        *ptr=1;
    }

    __host__ void run_foo(){
        int* u_ptr=unified_malloc(4K);

        u_ptr++; *u_ptr=0;

        foo<<<>>>(u_ptr);

    page fault!
    But we assume no page faults on GPUs!

    Is there a workaround??
Poor-man's unified memory
ADSM [I. Gelado 2010, CUDA 6]

- unified_alloc allocates one pointer, passed to both CPU and GPU
- The pointer is backed by two buffers
  - GPU buffer: GPUbbuf
  - CPU buffer: CPUbbuf
- CPU always accesses CPUbbuf
- GPU always accesses GPUbbuf
Poor-man's unified memory
ADSM [I. Gelado 2010, CUDA 6]

- unified_alloc allocates one pointer, passed to both CPU and GPU
- The pointer is backed by two buffers
  - GPU buffer: GPUbuf
  - CPU buffer: CPUbuf
- CPU always accesses CPUbuf
- GPU always accesses GPUbuf

Questions
How do they get synchronized?
How can it be the same ptr on CPU and GPU?
ADSM consistency try #1

• When the kernel is invoked, the CPUbuf is synchronized with the GPUbuf
• When the kernel stops, the GPUbuf is synchronized with the CPUbuf

Problems?
ADSM consistency: try #2

- Track dirty pages on CPU!
- Map CPU memory read-only
- When CPU process accesses
  - for write, mark page as dirty – this one has to be copied to GPU!
  - for read, copy from GPU if it was every copied there
- What about the GPU accesses?
Limitations

- Cannot over-subscribe GPU memory
- Cannot copy data to GPU on-demand
- Does not support concurrent CPU-GPU access
- Requires even read-only data to be copied to CPU

Solution??
Shared Virtual Address Space: GPU page faults

- Found in modern NVIDIA discrete GPUs (from Pascal) and AMD/Intel integrated GPUs
What does GPUs do when they PF

• They stall!
  - At least the thread that causes PF
  - Usually many more
  - They have to invalidate TLBs (TLB shootdown)

• May easily get to thousands of cycles!
Concurrent CPU-GPU access to shared pages

- Now GPU and CPU may have pointers to the same page. How to support consistency??
- Solution: Heterogeneous Memory Management
- Migrate a page to GPU upon GPU page fault
- Migrate a page to CPU upon CPU page fault

EXPENSIVE!!! Can we do better?
Prefetching and user hints

- cudaMemPrefetchAsync
- memAdvise
  - just like madvise for the page cache
  - readMostly: suggests full replication
  - preferredLocation: suggest where to keep data
    - enables access without migration
  - accessedBy: map permanently in a device
    - prevents migration
CPU-centric VM management

- Cons:
  - CPU in every page fault
  - CPU bottleneck when handling many page faults
  - CPU-GPU coordination for page cache management

Co-processor

I/O management and control by the CPU
VM management by an accelerator

- Why not?
  - Requires privileged execution to update page tables
  - Requires page fault to be caught by GPU
  - Requires exception handling

- Why yes?
Accelerator-centric VM management

- Layered on top of regular VM
- **Pros:**
  - no CPU involvement on page faults
  - Low page fault handling latency
  - Flexibility

**Diagram:**

- Page Table
  - Page Cache
- Address translation
- HW VM
  - Page Table
- Page fault handler
  - Peer-processor
  - Data and control path and OS I/O abstractions by the **accelerator**
- CPU
  - RAM
Accelerator-centric VM management

- Layered on top of regular VM
- Pros:
  - no CPU involvement on page faults
  - Low page fault handling latency
  - Flexibility

How can we build it on commodity hardware?
Do we really need new hardware?

- This part is based on the paper:
  "ActivePointers: the case for software address translation on GPUs" by S. Shahar, S. Bergman, M. Silberstein
Desired behavior

• CPU code
  ```c
  char* cpu_ptr=malloc(4K);
  ```

• GPU code
  ```c
  char* ptr=gmmmap(cpu_ptr,4K);
  int stride=blockSize.x;
  for(int i=0;i<size;i+=stride,ptr+=stride)
  {
    ptr[threadIdx.x]=25;
  }
  ```
Desired behavior

- **CPU code**
  ```c
  char* cpu_ptr = malloc(4K);
  ```

- **GPU code**
  ```c
  char* ptr = gmmap(cpu_ptr, 4K);
  int stride = blockSize.x;
  for (ptr; ptr < (cpu_ptr + stride); ptr += stride)
  {
    ptr[threadIdx.x] = 25;
  }
  ```

*One page fault on the first access must resolve to GPU page cache page*
Basic idea: lets implement page faults in software!

- Used only for \texttt{gmmmap}: \texttt{ActivePointer}: \texttt{Aptr}
- All pointer operations are overridden

```cpp
Aptr<
```
Basic idea: lets implement page faults in software!

- Used only for `gmmap`: `ActivePointer`: `Aptr`

- All pointer operations are overridden

```cpp
Aptr<char> ptr = gmmap(fd, offset, size);

int stride = blockSize.x;
for (int i = 0; i < size; i += stride, ptr += stride)
{
    ptr[threadIdx.x] = 25;
}

operator[](const offset) {
    int page = ptr.page + offset / PAGESIZE;
    if (!PageTbl[page].valid)
        HandlePageFault();
    return *((char*) (PageTbl[page].data + offset % PAGESIZE));
```

Basic idea: lets implement page faults in software!

- Used only for \texttt{gmmap}: ActivePointer: \texttt{Aptr}

- All pointer operations are overridden

```
Aptr<char> ptr=gmmap(fd,offset,size)
int stride=blockSize.x;
for(int i=0;i<size;i+=stride,ptr+=stride)
{
    ptr[threadIdx.x]=25;
}

operator[](const offset){
    int page=ptr.page+offset/PAGESIZE;
    if (!PageTbl[page].valid)
        HandlePageFault();
    return *((char*)(PageTbl[page].data +offset%PAGESIZE));
}
```

Expensive!
Software TLB - inefficient

- One TLB per core (Threadblock)
- TLB can reside in shared memory

Multiple shared memory accesses for each read
Contention on TLB updates
How to handle TLB invalidations?
ActivePointers
Main design principles

- Minimize page table lookups
  - translation is cached in hardware registers

- Pages locked in the buffer cache as long as they are in use
  - keep reference count for each page
### ActivePtr structure

<table>
<thead>
<tr>
<th>Valid</th>
<th>Page Cache Ptr</th>
<th>CPU RAM Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>NONE</td>
</tr>
</tbody>
</table>

64 bits
Example

cpu_ptr=4096

<table>
<thead>
<tr>
<th>ptr</th>
<th>Valid</th>
<th>Page Cache Ptr</th>
<th>CPU RAM addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
</tbody>
</table>

```c
ActivePtr ptr;
ptr = gmmap(cpu_ptr);
float x = *ptr;
ptr++;
float y = *ptr;
ptr+=4096;
```
cpu_ptr=4096

```c
ActivePtr ptr;
ptr = gmmmap(cpu_ptr);
float x = *ptr;
ptr++;
float y = *ptr;
ptr+=4096;
```

<table>
<thead>
<tr>
<th>ptr</th>
<th>Valid</th>
<th>Page Cache Ptr</th>
<th>CPU RAM addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>NONE</td>
<td>4096</td>
</tr>
</tbody>
</table>

```c

cpu_ptr=4096
```

```c
ActivePtr ptr;
ptr = gmmmap(cpu_ptr);
float x = *ptr;
ptr++;
float y = *ptr;
ptr+=4096;
```
Example

```c
ActivePtr ptr;
ptr = gmmap(cpu_ptr);
float x = *ptr;
ptr ++;
float y = *ptr;
ptr += 4096;
```
### Example

#### Page Table Entry

<table>
<thead>
<tr>
<th>RefCount</th>
<th>CPU addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4096</td>
</tr>
</tbody>
</table>

#### Page Cache Ptr

<table>
<thead>
<tr>
<th>Valid</th>
<th>Page Cache Ptr</th>
<th>CPU RAM addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>4096</td>
</tr>
</tbody>
</table>

```c
ActivePtr ptr;
pTR = gmemmap(cpu_ptr);
float x = *ptr;
pTR++;
float y = *ptr;
pTR+=4096;
```
Example

```c
ActivePtr ptr;
ptr = gmmmap(cpu_ptr);
float x = *ptr;
ptr++;
float y = *ptr;
ptr+=4096;
```

Page fault handler

```
Valid | Page Cache Ptr | CPU RAM addr
---|---------------|--------------
(ptr) 1 | 0xFFFFC0000 | 4096
```
Example

Page Table Entry

<table>
<thead>
<tr>
<th>RefCount</th>
<th>CPU addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4096</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Valid</th>
<th>Page Cache Ptr</th>
<th>CPU RAM addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0xFFFFC0004</td>
<td>4100</td>
</tr>
</tbody>
</table>

```c
ActivePtr ptr;
ptr = gmmap(cpu_ptr);
float x = *ptr;
ptr++;
float y = *ptr;
ptr+=4096;
```
Example

Page Table Entry

<table>
<thead>
<tr>
<th>RefCount</th>
<th>CPU addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4096</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Valid</th>
<th>Page Cache Ptr</th>
<th>CPU RAM addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0xFFF00004</td>
<td>4100</td>
</tr>
</tbody>
</table>

```c
ActivePtr ptr;
ptr = gmmap(cpu_ptr);
float x = *ptr;
ptr++;
float y = *ptr;
ptr+=4096;
}
```

Fault-free Lookup-free access
Example

Unlink page

Page Table Entry

<table>
<thead>
<tr>
<th>RefCount</th>
<th>CPU addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4096</td>
</tr>
</tbody>
</table>

0xFFFFC0000

Valid          | Page Cache Ptr          | CPU RAM addr |
----------------|--------------------------|--------------|
0              | NONE                     | 8196         |

```c

ActivePtr ptr;
ptr = gmmap(cpu_ptr);
float x = *ptr;
ptr++;
float y = *ptr;
ptr+=4096;
```

Crossing page boundary
Example

Unlink page

<table>
<thead>
<tr>
<th>Valid</th>
<th>Page Cache Ptr</th>
<th>CPU RAM addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>4100</td>
</tr>
</tbody>
</table>

```c
ActivePtr ptr;
ptr = gmmmap(cpu_ptr);
float x = *ptr;
ptr++;
float y = *ptr;
// ptr+=4096;
```

ptr becomes inaccessible
ActivePointers: state machine

uninitialized

Assignment

Out-of-page arithmetics

unlinked

Page fault

linked
Active pointer: operators

- Works mostly as a regular pointer
- Standard arithmetic operations (+, -, ++, etc)
- Dereferencing using star (*).

[] is not supported – cannot modify the base pointer
Challenge:
Thread level address translation

• Reminder: warps = 32 threads in lockstep
• Warp threads may access different pointers
  – Faults for different pages
  – No faults

Divergence!
Thread-level address translation

- **Challenge:** avoiding warp-level deadlocks

```c
while( trylock(lock) );
update page table entry
unlock( lock );
```
Idea:
Translation aggregation mechanism

- Quickly identify fault-free accesses (fast path)
- Handle faults in order
- Aggregate faults to the same page
- Accesses the page cache using a single non-divergent control flow
Aggregation mechanism

- Each thread tests for page fault and votes
- if (all fault free) - early abort
  - use warp-level voting instructions
- else: handle requests to the same page together
  - Choose page to handle
  - Count number of requesting threads
  - Update reference count and broadcast result
Evaluation

- Commodity NVIDIA K80 GPU
- CUDA
- GPUfs
Latency overheads

- A single GPU thread performing memory copy using ActivePointers

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Read</th>
<th>Read+Inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw access</td>
<td>225</td>
<td>257</td>
</tr>
<tr>
<td>ActivePointers</td>
<td>271 (+20%)</td>
<td>423 (+65%)</td>
</tr>
</tbody>
</table>
Throughput overheads

- Same experiment with **full GPU occupancy**

<table>
<thead>
<tr>
<th>Transfer bandwidth</th>
<th>4-byte</th>
<th>8-byte</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>99.65 GB/s</td>
<td>148.7 GB/s</td>
</tr>
<tr>
<td></td>
<td>(65.4%)</td>
<td>(97.6%)</td>
</tr>
</tbody>
</table>

Free-computation bubble
Latency hiding – the key to performance

- Different compute/memory ratio
Image collage

• End-to-end evaluation on K80 GPU

Pointer access to 40GB DB file in CPU memory

• No measurable overhead
• 2.6x over 12 CPU cores with 256-bit AVX
• 3.5x over CPU + GPU
What we learned

- GPU VM similar to CPU VM
- Extra work required to handle memory divide
- Hardware page faults help
- Software VM management allows more flexible VM policies for accelerators