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

CPU

GPU 0

GPU 1

GPU N

System Memory

GPU Memory
UNIFIED MEMORY
Starting with Kepler and CUDA 6

Custom Data Management

Developer View With Unified Memory

System Memory
GPU Memory
Unified Memory
UNIFIED MEMORY IMPROVES DEV CYCLE

- Identify Parallel Regions
- Allocate and Move Data
- Optimize GPU Kernels
- Implement GPU Kernels
- Optimize Data Locality
- Implement GPU Kernels

Often time-consuming and bug-prone
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);
}
```
UNIFIED MEMORY ON PRE-PASCAL

Code example explained

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

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

GPU memory
~0.3 TB/s

System memory
~0.1 TB/s

CUDA Allocation
Managed
Kernel launch

PCI-E

Page fault
Page fault
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

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

GPU memory
~0.7 TB/s

System memory
~0.1 TB/s

CUDA malloc
Managed

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
**UM USE CASE: HPGMG**

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

[Diagram showing V-Cycle and F-Cycle with thresholds and CPU/GPU handling.]
HPGMRG AMR PROXY

Data locality and reuse of AMR levels

Legend:
- AMR level data
- N independent multi-grid solves

AMR Level

<table>
<thead>
<tr>
<th>Level</th>
<th>AMR Level Data</th>
<th>N independent multi-grid solves</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>t=0 start here</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OVERSUBSCRIPTION RESULTS

Application throughput (MDOF/s)

- x86
- K40
- P100 (x86 PCI-e)
- P100 (P8 NVLINK)

All 5 levels fit in GPU memory

P100 memory size (16GB)

Only 2 levels fit

Only 1 level fits

Application working set (GB)

- 1.4
- 4.7
- 8.6
- 28.9
- 58.6

x86 CPU: Intel E5-2630 v3, 2 sockets of 10 cores each with HT on (40 threads)
UNIFIED MEMORY: ATOMICS

Pre-Pascal: atomics from the GPU are atomic only for \textit{that} GPU
  
  GPU atomics to peer memory are \textbf{not} atomic for remote GPU
  
  GPU atomics to CPU memory are \textbf{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

```c
__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 μs 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

```c
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
__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();

Page fault within kernel are expensive.

<table>
<thead>
<tr>
<th></th>
<th>ms (GB/s)</th>
<th>Total time</th>
<th>Kernel time</th>
<th>Prefetch time</th>
</tr>
</thead>
<tbody>
<tr>
<td>No prefetch</td>
<td>476 (2.3)</td>
<td>476 (2.3)</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Prefetch</td>
<td>49 (22)</td>
<td>2 (536)</td>
<td>47 (11)</td>
<td></td>
</tr>
</tbody>
</table>

n=128M
HPGMG: DATA PREFETCHING

Prefetch next level while performing computations on current level

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

<table>
<thead>
<tr>
<th>compute</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>3</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefetch</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eviction</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RESULTS WITH USER HINTS

- **x86**
- **K40**
- **P100 (x86 PCI-e)**
- **P100 + hints (x86 PCI-e)**
- **P100 (P8 NVLINK)**
- **P100 + hints (P8 NVLINK)**

**Application working set (GB)**

- **1.4**
- **4.7**
- **8.6**
- **28.9**
- **58.6**

**Application throughput (MDOF/s)**

- **P100 memory size (16GB)**

- All 5 levels fit in GPU memory
- Only 2 levels fit
- Only 1 level fits

**x86 CPU:** Intel E5-2630 v3, 2 sockets of 10 cores each with HT on (40 threads)
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
data 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
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

Level N+1

Memory

Smoother
Residual
Restriction

GPU kernels

Smoother
Residual

CPU functions

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`

no page faults

~5% performance improvement on Tesla P100
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

![Bar chart showing performance comparison]

2xK40 with P2P support

- managed
- managed + cuda-aware
- zero-copy
- device + cuda-aware

MDOF/s
UNIFIED MEMORY AND DIRECTIVES
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)

gpushe.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/o UM

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

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

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

W/ UM

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 mpcg(4,me)
!$acc parallel loop
do i=m,me
   zelectron(1,m)=mpcg(1,m)
endo
!$acc end parallel
call foo(zelectron)
UNIFIED MEMORY SUMMARY

On-demand page migration

Performance overhead: migration, page fault

Optimization: prefetch, duplicate, mapping