Unified Memory
Notes on GPU Data Transfers

Andreas Herten, Forschungszentrum Jülich, 24 April 2017  Handout Version
Overview

- Unified Memory enables easy access to GPU development
- But some tuning might be needed for best performance

Contents

Background on Unified Memory
  - History of GPU Memory
  - Unified Memory on Pascal
  - Unified Memory on Kepler

Practical Differences
- Revisiting
  - scale_vector_um Example
- Hints for Performance
- Task
Background on Unified Memory

History of GPU Memory
CPU and GPU Memory

*Location, location, location*

At the Beginning  CPU and GPU memory very distinct, own addresses
**CPU and GPU Memory**

*Location, location, location*

At the *Beginning* CPU and GPU memory very distinct, own addresses

**CUDA 4.0** Unified Virtual Addressing: pointer from same address pool, but data copy manual

**CUDA 6.0** Unified Memory*: Data copy by driver, but whole data at once

**CUDA 8.0** Unified Memory (truly): Data copy by driver, page faults on-demand initiate data migrations (Pascal)
void sortfile(FILE *fp, int N) {
    char *data;
    char *data_d;

    data = (char *)malloc(N);
    cudaMalloc(&data_d, N);

    fread(data, 1, N, fp);

    cudaMemcpy(data_d, data, N, cudaMemcpyHostToDevice);
    kernel<<<...>>>(data, N);

    cudaMemcpy(data, data_d, N, cudaMemcpyDeviceToHost);
    host_func(data);
    cudaFree(data_d); free(data);}
Implementation Details (on Pascal)

*Under the hood*

```c
cudaMallocManaged(&ptr, ...); // Empty! No pages anywhere yet (like malloc())

*ptr = 1; // CPU page fault: data allocates on CPU

kernel<<<...>>>(ptr); // GPU page fault: data migrates to GPU
```

- Pages populate on **first touch**
- Pages migrate on-demand
- GPU memory over-subscription possible
- Concurrent access from CPU and GPU to memory (page-level)
On-Demand Migration Flow at Pascal

- **GPU Memory**
  - $\approx 0.7 \, \text{TB/s}$

- **System Memory**
  - $\approx 0.1 \, \text{TB/s}$

Interconnect

Andreas Herten | Unified Memory | 24 April 2017
On-Demand Migration Flow at Pascal

- GPU Memory: \(\approx 0.7 \, \text{TB/s} \)
- System Memory: \(\approx 0.1 \, \text{TB/s} \)

Interconnect

\texttt{cudaMallocManaged}

Page fault
On-Demand Migration Flow at Pascal

GPU Memory
≈0.7 TB/s

cudaMallocManaged

Interconnect

System Memory
≈0.1 TB/s

Page fault
On-Demand Migration Flow at Pascal

GPU Memory
\[ \approx 0.7 \text{ TB/s} \]

System Memory
\[ \approx 0.1 \text{ TB/s} \]

Interconnect

cudaMallocManaged

Page fault
On-Demand Migration Flow at Pascal

GPU Memory
≈ 0.7 TB/s

System Memory
≈ 0.1 TB/s

 CUDAMallocManaged

Interconnect

Page fault

Andreas Herten | Unified Memory | 24 April 2017
On-Demand Migration Flow at Pascal

- **GPU Memory**
  - $\approx 0.7 \text{ TB/s}$

- **System Memory**
  - $\approx 0.1 \text{ TB/s}$

- Interconnect

- cudaMallocManaged

Andreas Herten | Unified Memory | 24 April 2017
On-Demand Migration Flow at Pascal

- GPU Memory: $\approx 0.7 \text{ TB/s}$
- System Memory: $\approx 0.1 \text{ TB/s}$

Interconnect
On-Demand Migration Flow at Pascal

GPU Memory
≈0.7 TB/s

System Memory
≈0.1 TB/s

Page fault
cudaMalloc Managed
Interconnect
On-Demand Migration Flow at Pascal

GPU Memory
≈0.7 TB/s

System Memory
≈0.1 TB/s

Page fault

Interconnect
On-Demand Migration Flow at Pascal

GPU Memory
≈0.7 TB/s

System Memory
≈0.1 TB/s

Interconnect
On-Demand Migration Flow at Pascal

GPU Memory
≈0.7 TB/s

CUDA MallocManaged

System Memory
≈0.1 TB/s

Interconnect

Andreas Herten | Unified Memory | 24 April 2017
On-Demand Migration Flow at Pascal

GPU Memory
≈ 0.7 TB/s

System Memory
≈ 0.1 TB/s

CUDA MallocManaged
Interconnect

Page fault
On-Demand Migration Flow at Pascal

GPU Memory
≈0.7 TB/s

System Memory
≈0.1 TB/s

Interconnect

Page fault

cudaMallocManaged
On-Demand Migration Flow at Pascal

GPU Memory
≈0.7 TB/s

System Memory
≈0.1 TB/s

Interconnect

Andreas Herten | Unified Memory | 24 April 2017
On-Demand Migration Flow at Pascal

GPU Memory
≈0.7 TB/s

System Memory
≈0.1 TB/s

Interconnect

Page fault

Andreas Herten | Unified Memory | 24 April 2017
On-Demand Migration Flow at Pascal

GPU Memory
≈0.7 TB/s

Interconnect

cudaMallocManaged

System Memory
≈0.1 TB/s

Page fault

Andreas Herten | Unified Memory | 24 April 2017
On-Demand Migration Flow at Pascal

GPU Memory
≈0.7 TB/s

System Memory
≈0.1 TB/s

Interconnect
On-Demand Migration Flow at Pascal

GPU Memory
\[ \approx 0.7 \text{ TB/s} \]

System Memory
\[ \approx 0.1 \text{ TB/s} \]

Interconnect

cudaMallocManaged

Page fault

Andreas Herten | Unified Memory | 24 April 2017
On-Demand Migration Flow at Pascal

- **GPU Memory**: \( \approx 0.7 \text{ TB/s} \)
- **System Memory**: \( \approx 0.1 \text{ TB/s} \)

**Interconnect**

**Map memory to system memory**

Andreas Herten | Unified Memory | 24 April 2017
On-Demand Migration Flow at Pascal

**GPU Memory**
≈0.7 TB/s

**System Memory**
≈0.1 TB/s

Only needed page is copied (≥4 kB)!
Migration on Kepler

GPU Memory
≈0.3 TB/s

System Memory
≈0.1 TB/s

PCI-Express
Migration on Kepler

GPU Memory
≈0.3 TB/s

cudaMallocManaged

PCI-Express

System Memory
≈0.1 TB/s
Migration on Kepler

GPU Memory
\[ \approx 0.3 \text{ TB/s} \]

System Memory
\[ \approx 0.1 \text{ TB/s} \]
Migration on Kepler

GPU Memory
≈ 0.3 TB/s

System Memory
≈ 0.1 TB/s

Page fault

CUDA
Managed
Page
fault

PCI-Express

Andreas Herten | Unified Memory | 24 April 2017

# 8
Migration on Kepler

GPU Memory
\[ \approx 0.3 \text{ TB/s} \]

PCI-Express

System Memory
\[ \approx 0.1 \text{ TB/s} \]
Migration on Kepler

GPU Memory
\[ \approx 0.3 \text{ TB/s} \]

System Memory
\[ \approx 0.1 \text{ TB/s} \]
Migration on Kepler

GPU Memory
≈ 0.3 TB/s

System Memory
≈ 0.1 TB/s

PCI-Express

Page fault

Andreas Herten | Unified Memory | 24 April 2017
Migration on Kepler

GPU Memory
≈0.3 TB/s

System Memory
≈0.1 TB/s

Andreas Herten | Unified Memory | 24 April 2017
Migration on Kepler

GPU Memory
\[ \approx 0.3 \text{ TB/s} \]

System Memory
\[ \approx 0.1 \text{ TB/s} \]
Migration on Kepler

GPU Memory
≈0.3 TB/s

System Memory
≈0.1 TB/s

PCI-Express
Migration on Kepler

GPU Memory
≈0.3 TB/s

System Memory
≈0.1 TB/s

Kernel launch
Page fault not supported

PCI-Express

Andreas Herten | Unified Memory | 24 April 2017
Migration on Kepler

GPU Memory
≈0.3 TB/s

System Memory
≈0.1 TB/s

PCI-Express
Migration on Kepler

- GPU Memory: \( \approx 0.3 \text{ TB/s} \)
- System Memory: \( \approx 0.1 \text{ TB/s} \)

PCI-Express
Implementation before Pascal

Kepler (JURECA), Maxwell, …

- Pages populate on GPU with `cudaMallocManaged()`
- Might migrate to CPU if touched there first
- Pages migrate in bulk to GPU on kernel launch
- No over-subscription possible
Practical Differences

Revisiting scale_vector_um Example
Comparing UM on Pascal & Kepler

Different scales

Who profiled scale_vector_um on JURON, who on JURECA?
→ What are run times for kernel?

==109924== Profiling result:
Time(%)  Time   Calls  Avg  Min  Max  Name
100.00%  519.36us  1  519.36us  519.36us  519.36us  scale(float, float*, float*, int)

==12922== Profiling result:
Time(%)  Time   Calls  Avg  Min  Max  Name
100.00%  13.216us  1  13.216us  13.216us  13.216us  scale(float, float*, float*, int)

Why?!
Shouldn’t P100 be about 3\times faster than K80?
Comparing UM on Pascal & Kepler

Different scales

Who profiled `scale_vector_um` on JURON, who on JURECA?

→ What are run times for kernel?
Comparing UM on Pascal & Kepler

What happens?

**JURON**  Kernel is launched, data is needed by kernel, data migrates host→device
⇒ Run time of kernel **incorporates** time for data transfers

**JURECA**  Data will be needed by kernel – so data migrates host→device **before** kernel launch
⇒ Run time of **kernel** without any transfers

- Implementation on Pascal is the more convenient one
- Total run time of whole program does not principally change
  
  *Except it gets shorter because of faster architecture*
- But data transfers sometimes sorted to kernel launch
⇒ **What can we do about this?**
Performance Hints for UM

General hints

- **Keep data local**
  Prevent migrations at all if data is processed by close processor

- **Minimize thrashing**
  Constant migrations hurt performance

- **Minimize page fault overhead**
  Fault handling costs $O(10 \mu s)$, stalls execution
Performance Hints for UM

New API routines

New API calls to augment data location knowledge of runtime

- `cudaMemPrefetchAsync(data, length, device, stream)`
  Prefetches data to device (on stream) asynchronously

- `cudaMemAdvise(data, length, advice, device)`
  Advise about usage of given data, advice:
  - `cudaMemAdviseSetReadMostly`: Data is mostly read and occasionally written to
  - `cudaMemAdviseSetPreferredLocation`: Set preferred location to avoid migrations; first access will establish mapping
  - `cudaMemAdviseSetAccessedBy`: Data is accessed by this device; will pre-map data to avoid page fault

- Use `cudaCpuDeviceId` for device CPU, or use `cudaGetDevice()` as usual to retrieve current GPU device id (default: 0)
void sortfile(FILE *fp, int N) {
    char *data;
    // ...
    cudaMallocManaged(&data, N);

    fread(data, 1, N, fp);

    cudaMemAdvise(data, N, cudaMemAdviseSetReadMostly, device);
    cudaMemPrefetchAsync(data, N, device);
    kernel<<<...>>>(data, N);
    cudaDeviceSynchronize();

    host_func(data);
    cudaFree(data); }

Read-only copy of data is created on GPU during prefetch → CPU and GPU reads will not fault

Prefetch data to avoid expensive GPU page faults
Tuning scale_vector_um

Express data movement

- Location of code: Unified_Memory/exercises/tasks/scale/
- Look at Instructions.rst for instructions
  1. Show runtime that data should be migrated to GPU before kernel call
  2. Build with make (CUDA needs to be loaded!)
  3. Run with make run
     
        Or bsub -I -R "rusage[ngpus_shared=1]" ./scale_vector_um
  4. Generate profile to study your progress – see make profile
- See also CUDA C programming guide for details on data usage

Finished early? There’s one more task in the appendix!
Conclusions
What we’ve learned

- **Unified Memory** is implemented differently on Pascal (JURON) and Kepler (JURECA)
- With CUDA 8.0, there are new API calls to express **data locality**

Thank you for your attention!

a.herten@fz-juelich.de
Appendix
Jacobi Task
Glossary
Jacobi Task
One more time…

- Location of code: Unified_Memory/exercises/tasks/jacobi/
- See Jiri Kraus’ slides on Unified Memory from last year at Unified_Memory/exercises/slides/jkraus-unified_memory-2016.pdf
- Short instructions
  - Avoid data migrations in while loop of Jacobi solver: apply boundary conditions with provided GPU kernel; try to avoid remaining migrations
  - Build with make (CUDA needs to be loaded!)
  - Run with make run
  - Look at profile – see make profile
API  A programmatic interface to software by well-defined functions. Short for application programming interface. 53

ATI  Canada-based GPUs manufacturing company; bought by AMD in 2006. 53

CUDA  Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 4, 5, 49, 50, 53

GCC  The GNU Compiler Collection, the collection of open source compilers, among other for C and Fortran. 53
Glossary II

**LLVM**  An open source compiler infrastructure, providing, among others, Clang for C. 53

**NVIDIA**  US technology company creating GPUs. 53

**NVLink**  NVIDIA’s communication protocol connecting CPU ↔ GPU and GPU ↔ GPU with 80 GB/s. PCI-Express: 16 GB/s. 53

**OpenACC**  Directive-based programming, primarily for many-core machines. 53

**OpenCL**  The *Open Computing Language*. Framework for writing code for heterogeneous architectures (*CPU, GPU, DSP, FPGA*). The alternative to *CUDA*. 53
Glossary III

**OpenGL**  The *Open Graphics Library*, an **API** for rendering graphics across different hardware architectures. 53

**OpenMP**  Directive-based programming, primarily for multi-threaded machines. 53

**P100**  A large **GPU** with the Pascal architecture from **NVIDIA**. It employs **NVLink** as its interconnect and has fast **HBM2** memory. 53

**SAXPY**  Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 53

**Tesla**  The **GPU** product line for general purpose computing of **NVIDIA**. 53
Thrust  A parallel algorithms library for (among others) GPUs. See [https://thrust.github.io/](https://thrust.github.io/).