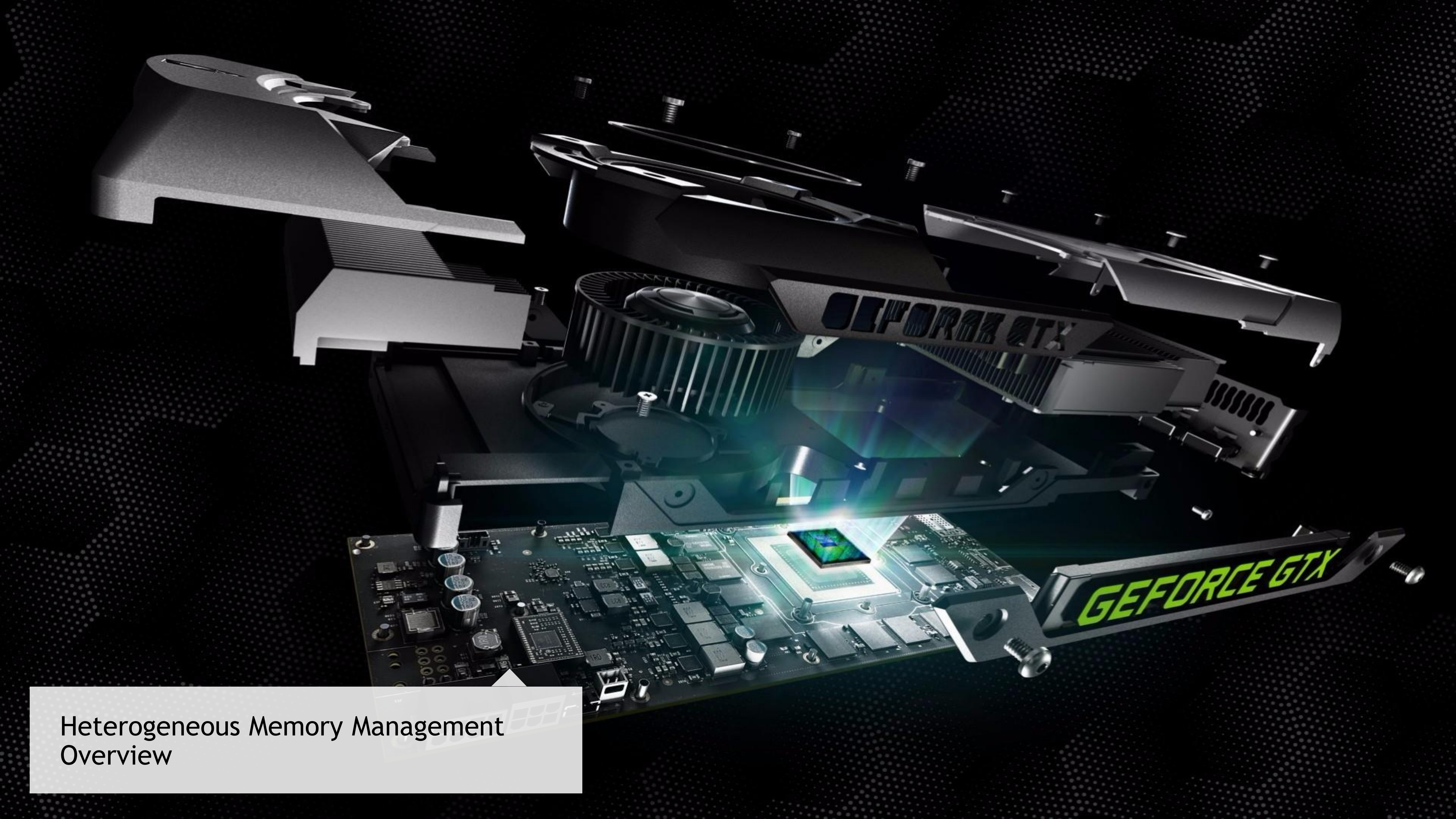
Using HMM to Blur the Lines between CPU and GPU Programming

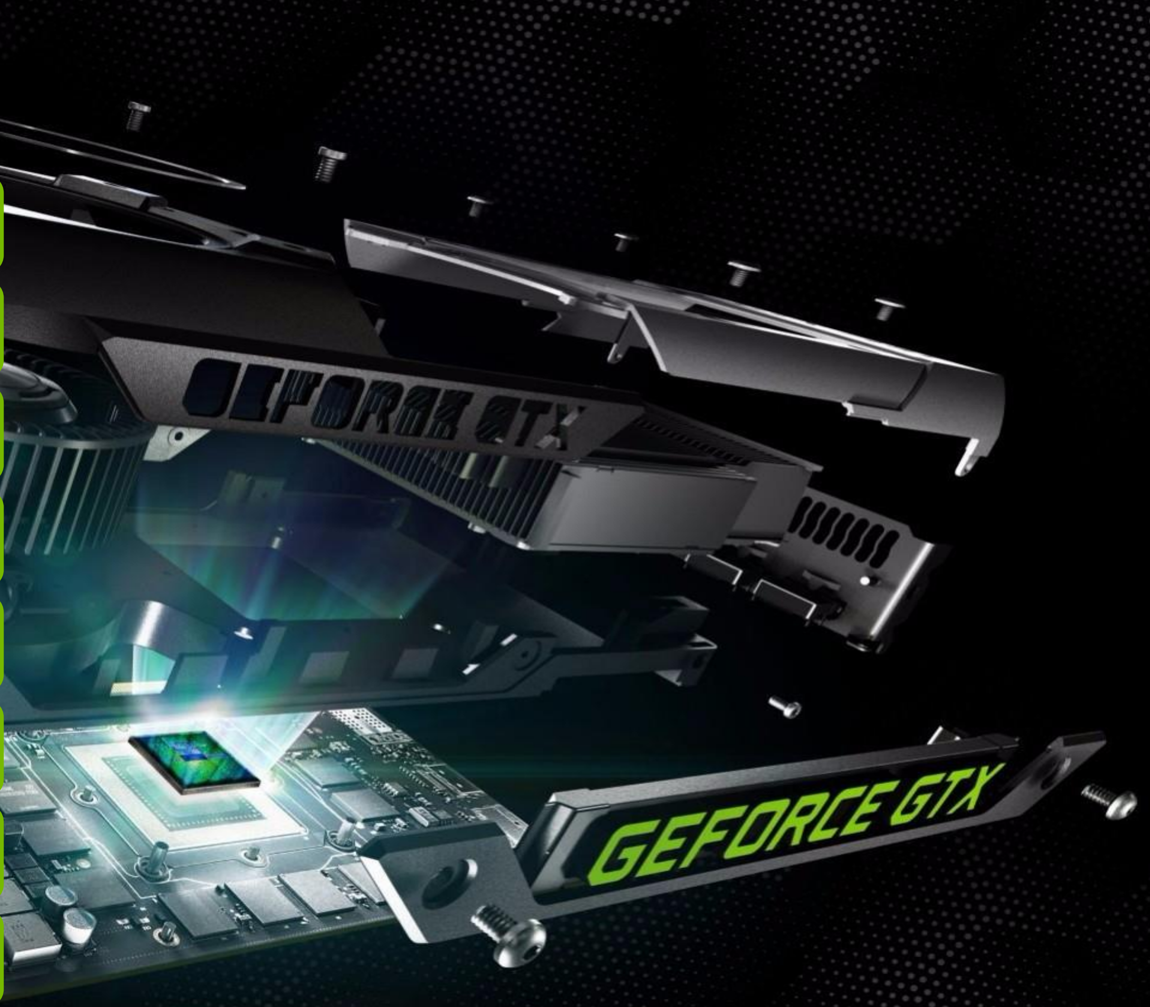
John Hubbard, May 10, 2017

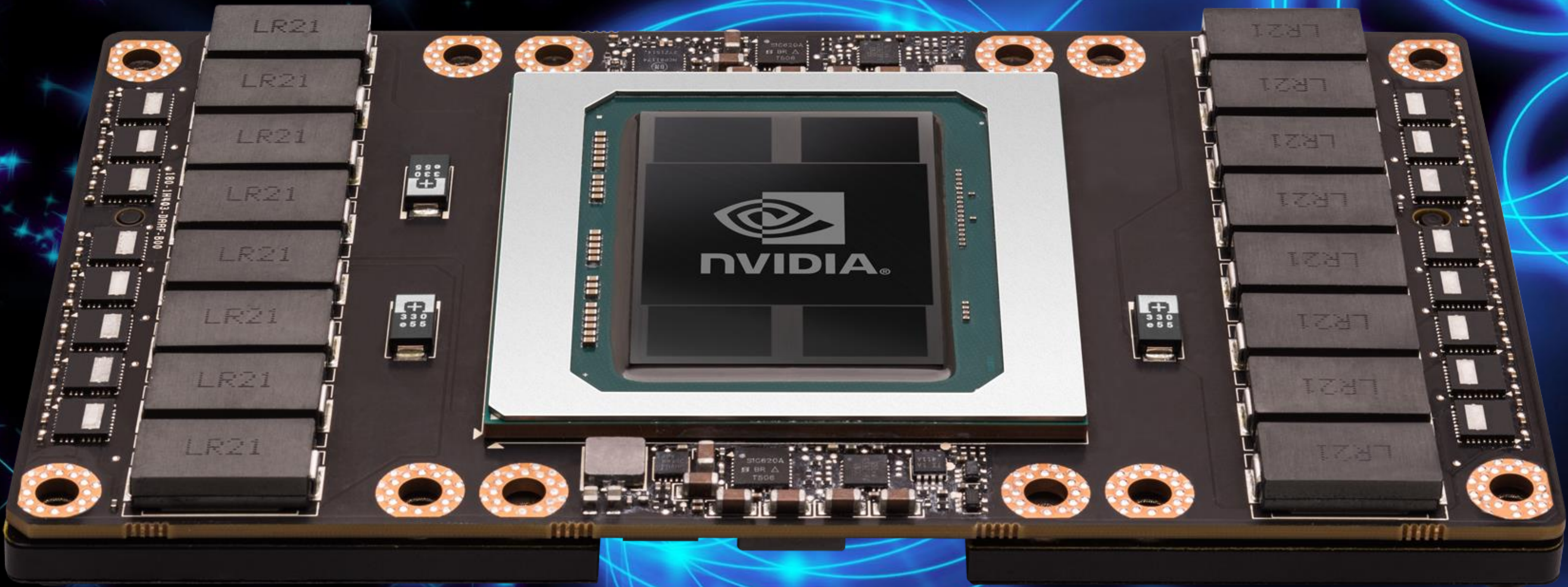


Heterogeneous Memory Management Overview

Agenda

- Overview
- HMM Benefits
- SW-HW stack: where does HMM fit in?
- Definitions
- How HMM works
- Profiling with HMM
- A little bit of history
- References
- Conclusion

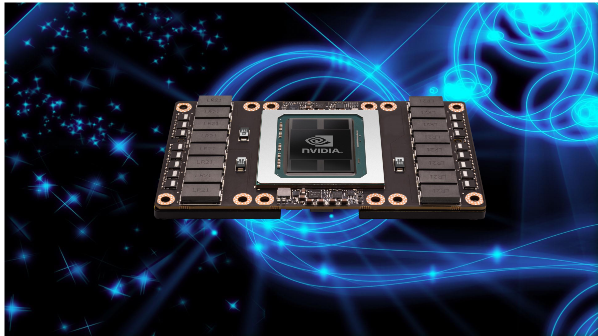




HMM Benefits

HMM Benefits

Simpler code



Standard Unified Memory (CUDA 8.0)

```
#include <stdio.h>
#define LEN sizeof(int)

__global__ void
compute_this(int *pDataFromCpu)
{
   atomicAdd(pDataFromCpu, 1);
}

int main(void)
{
   int *pData = NULL;
   cudaMallocManaged(&pData, LEN);
   *pData = 1;

   compute_this<<<512,1000>>>(pData);
   cudaDeviceSynchronize();

   printf("Results: %d\n", *pData);
   cudaFree(pData);
   return 0;
}
```

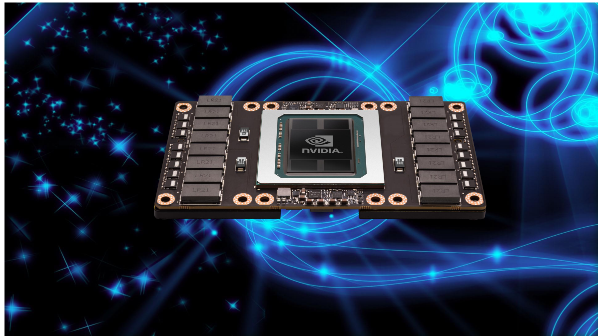
Unified Memory + HMM

```
#include <stdio.h>
#define LEN sizeof(int)

__global__ void
compute_this(int *pDataFromCpu)
{
   atomicAdd(pDataFromCpu, 1);
}

int main(void)
{
   int *pData = (int*)malloc(LEN);
   *pData = 1;

   compute_this<<<512,1000>>>(pData);
   cudaDeviceSynchronize();

   printf("Results: %d\n", *pData);
   free(pData);
   return 0;
}
```

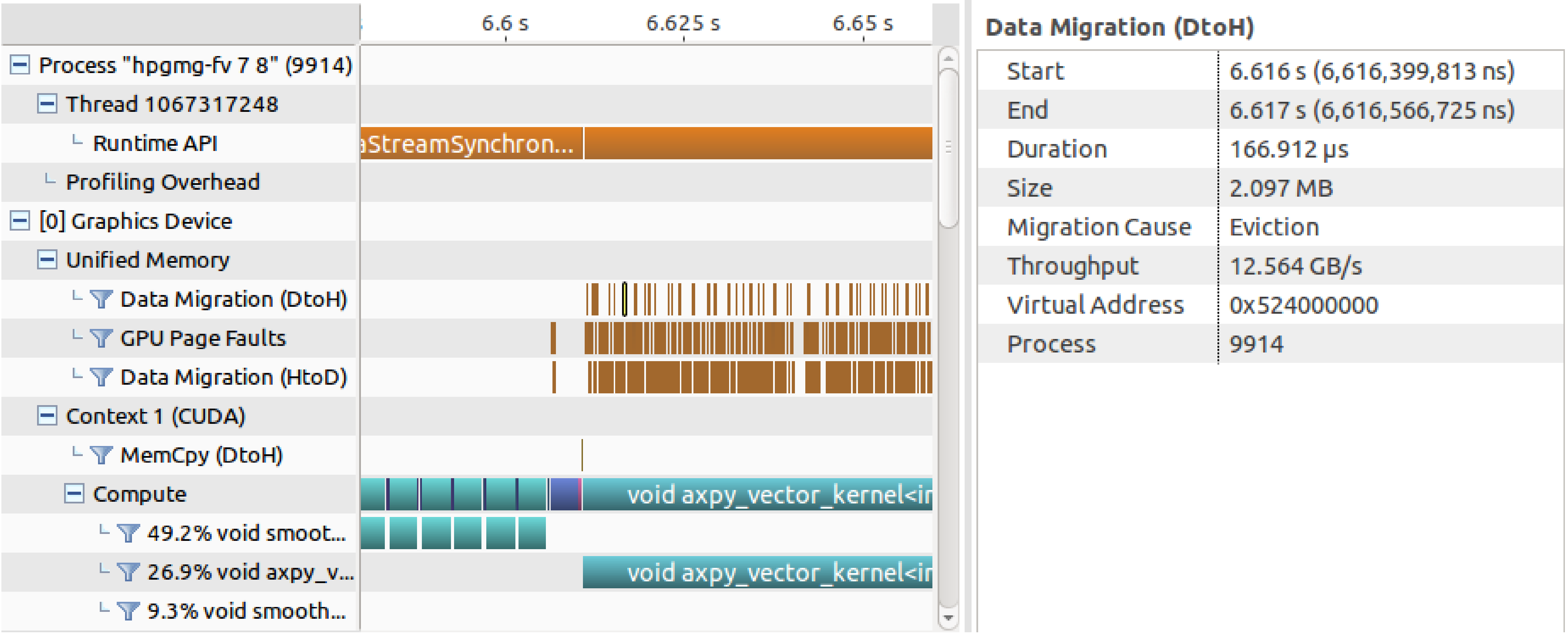
HMM Benefits

Simpler code

Code is still tunable



Profiling with Unified Memory: Visual Profiler



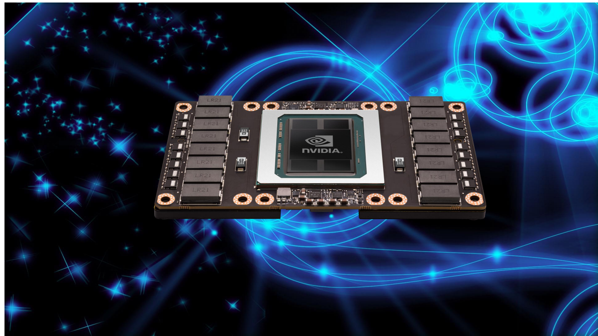
Source: https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal

HMM Benefits

Simpler code

Code is still tunable

Libraries can be used without changing them



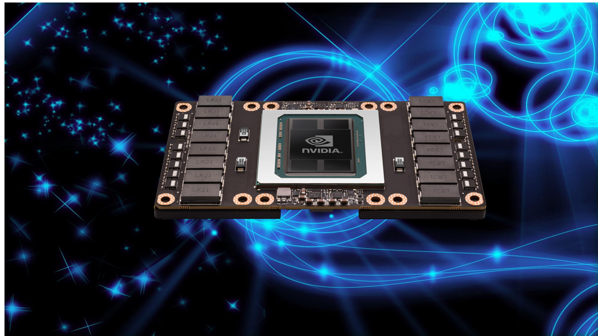
HMM Benefits

Simpler code

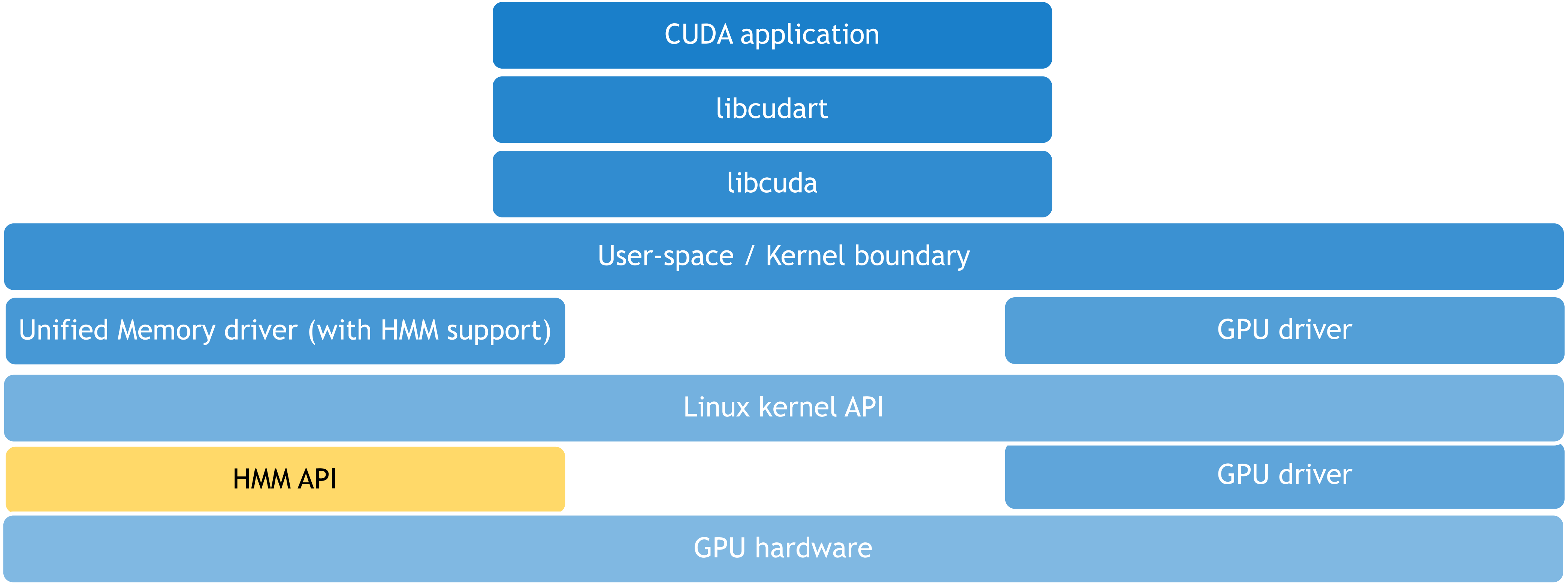
Code is still tunable

Libraries can be used without changing them

New programming languages are easily supported

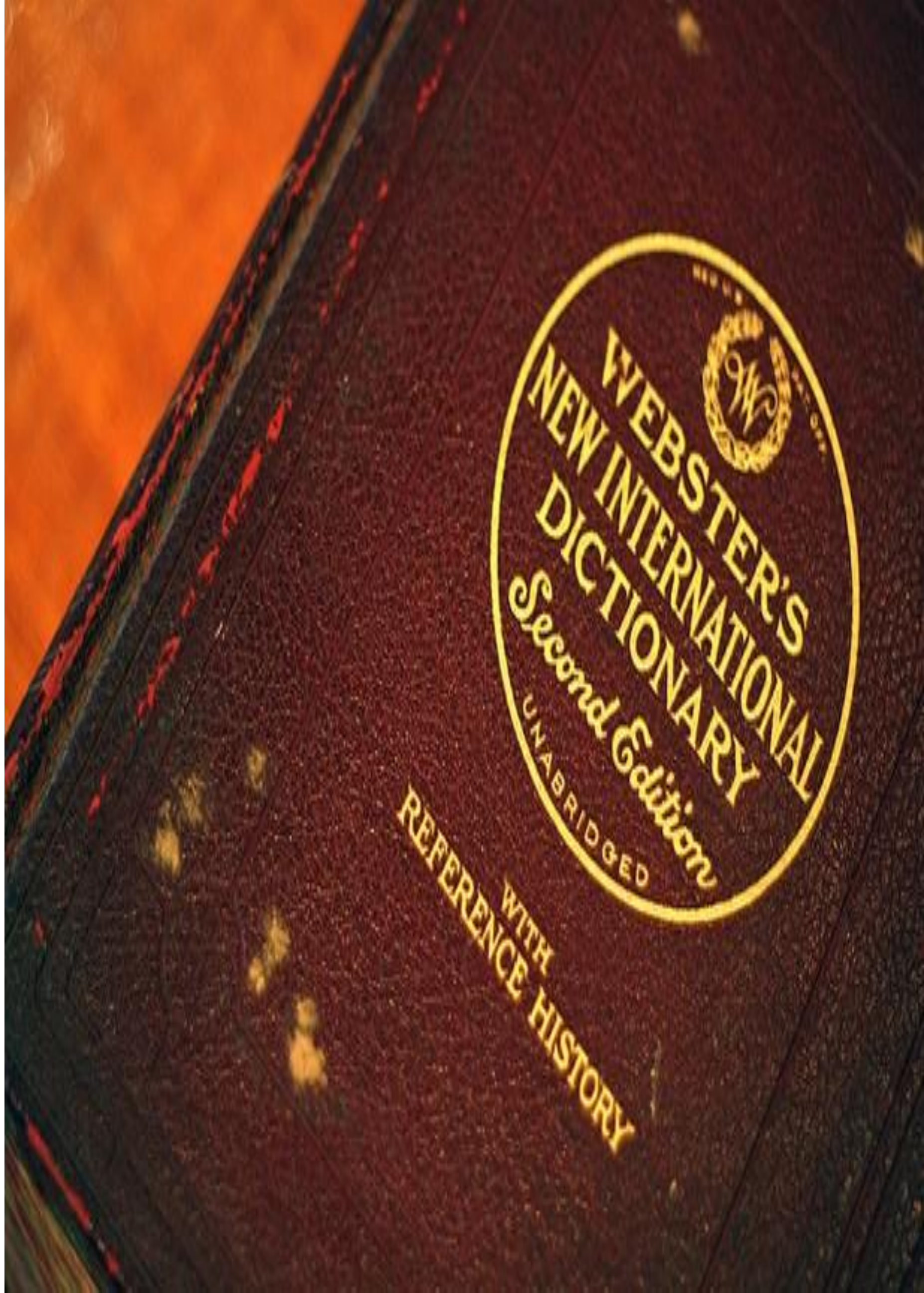


SW-HW stack: where does HMM fit in?



Definitions

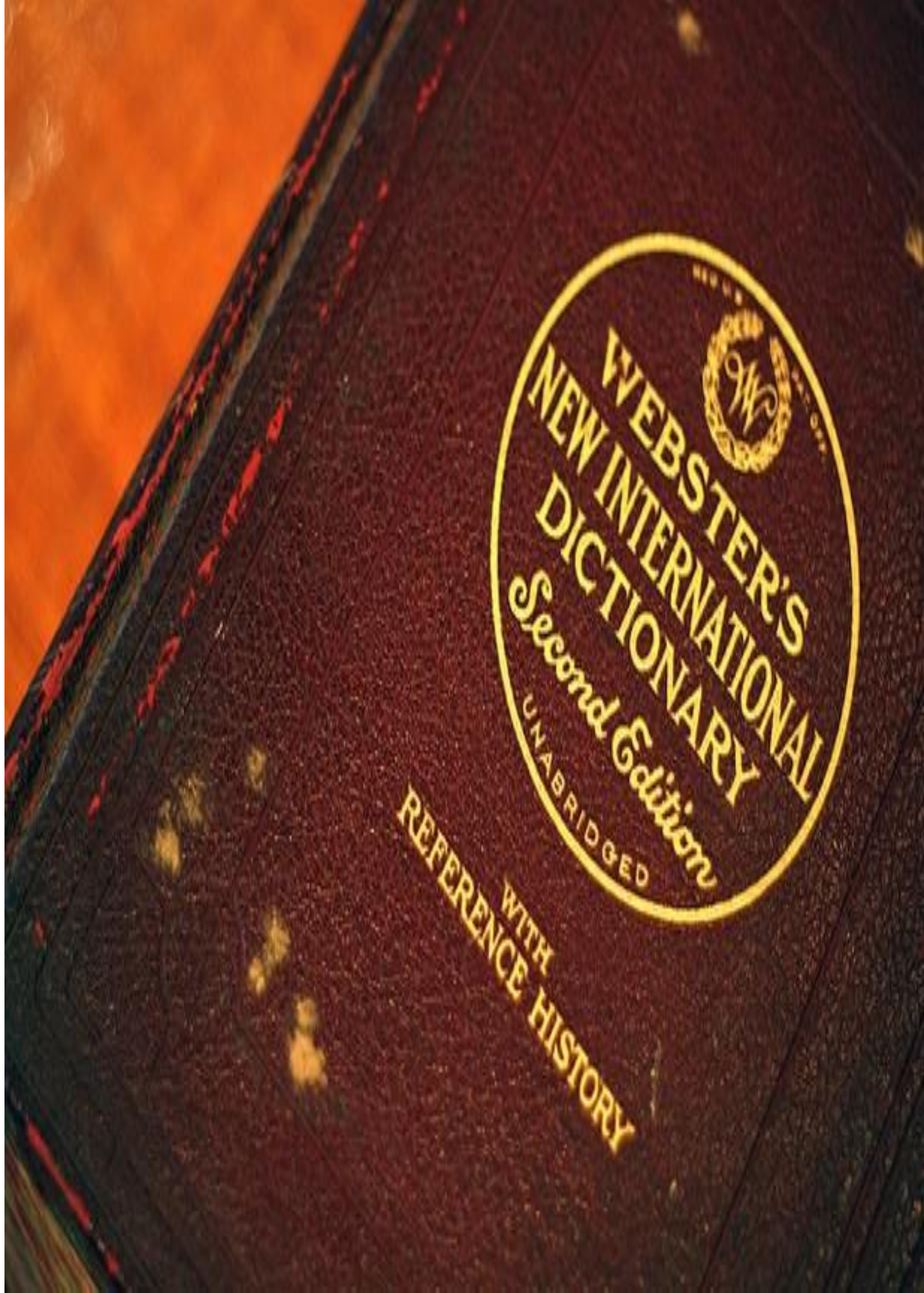
OS: Operating System



Definitions

OS: Operating System

Kernel: Linux operating system internals (not a CUDA kernel!)

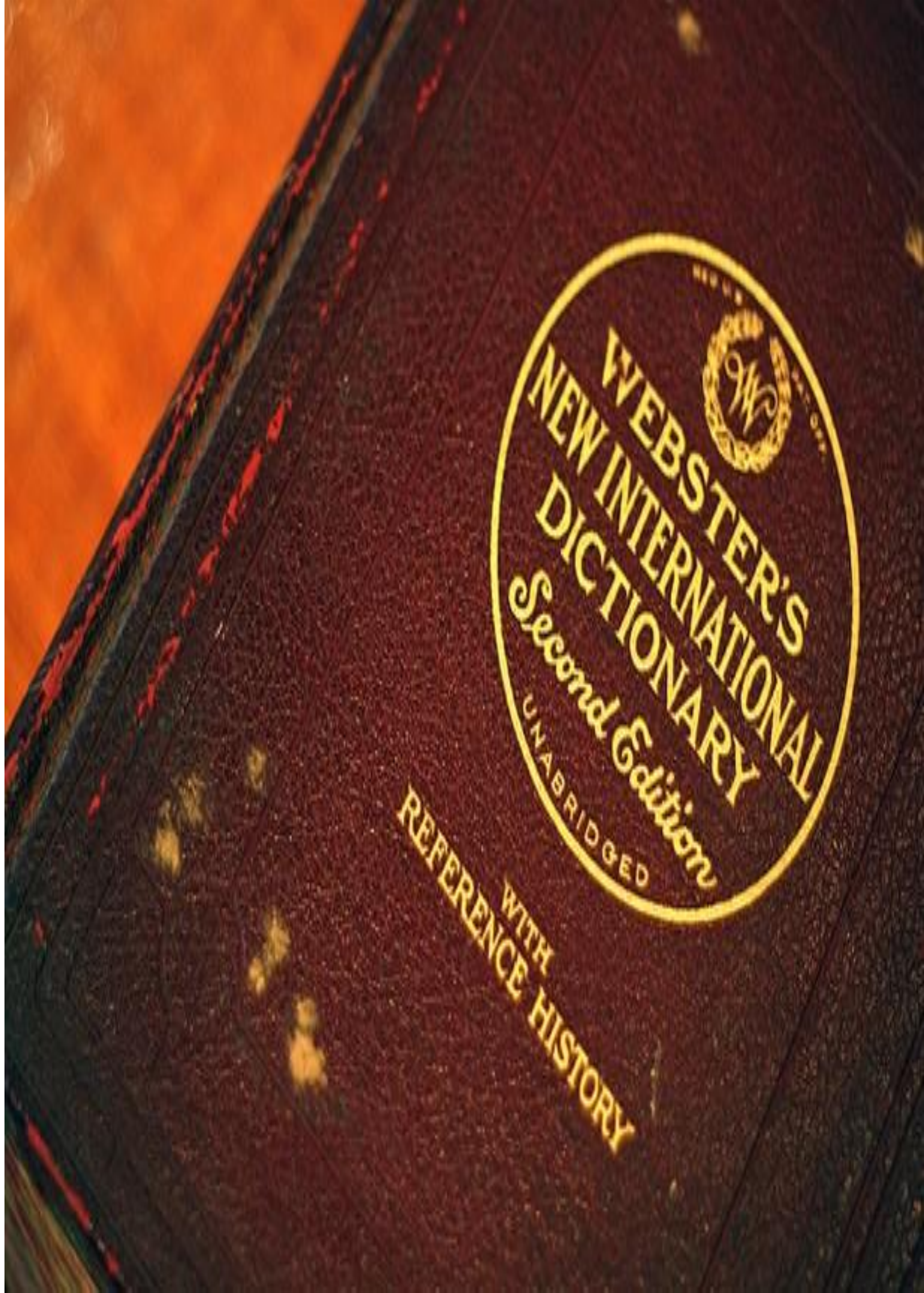


Definitions

OS: Operating System

Kernel: Linux operating system internals (not a CUDA kernel!)

Page: 4KB, 64KB, 2MB, etc.of physically contiguous memory. Smallest unit handled by the OS.



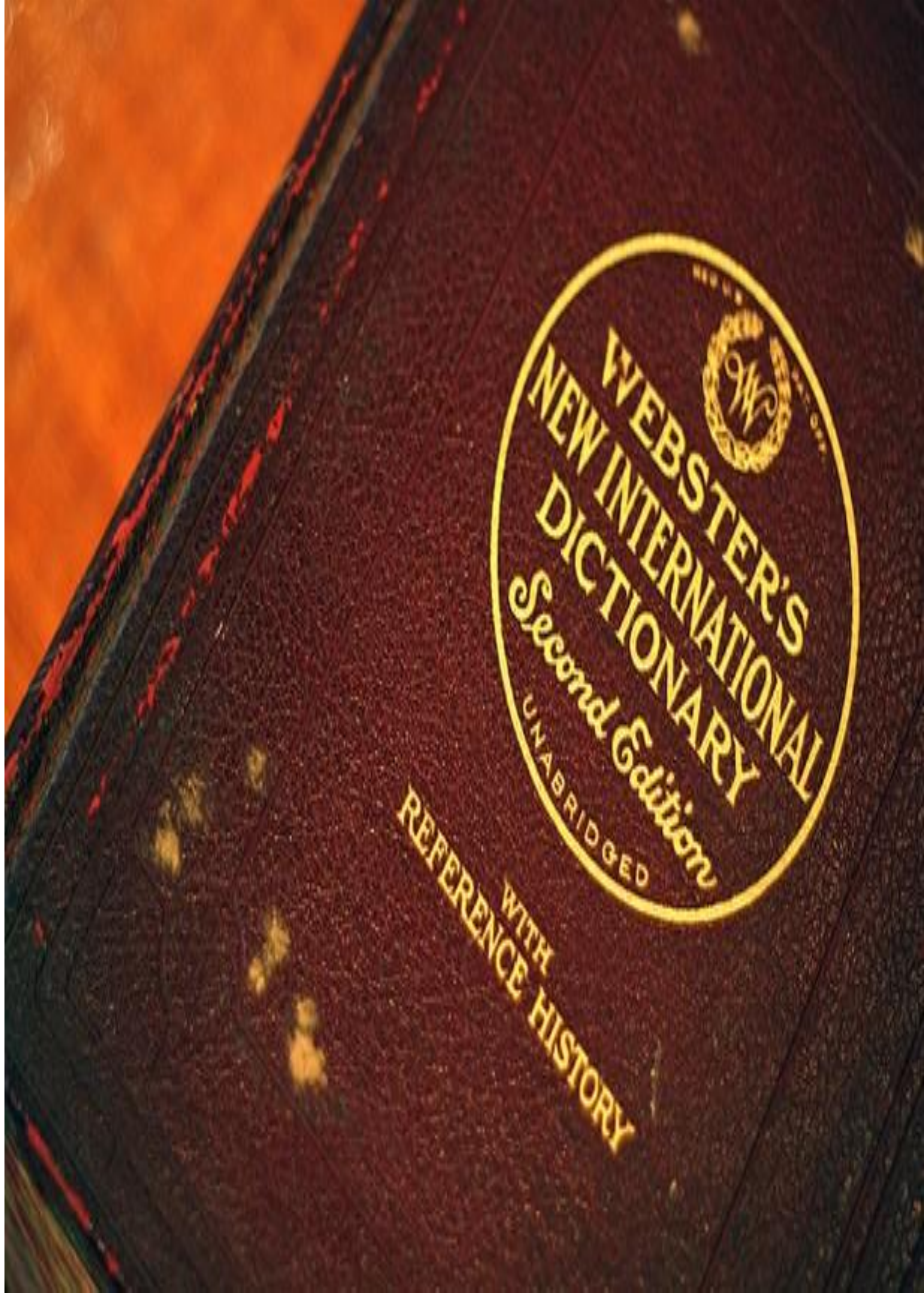
Definitions

OS: Operating System

Kernel: Linux operating system internals (not a CUDA kernel!)

Page: 4KB, 64KB, 2MB, etc.of physically contiguous memory. Smallest unit handled by the OS.

Page table: sparse tree containing virtual-to-physical address translations



Definitions

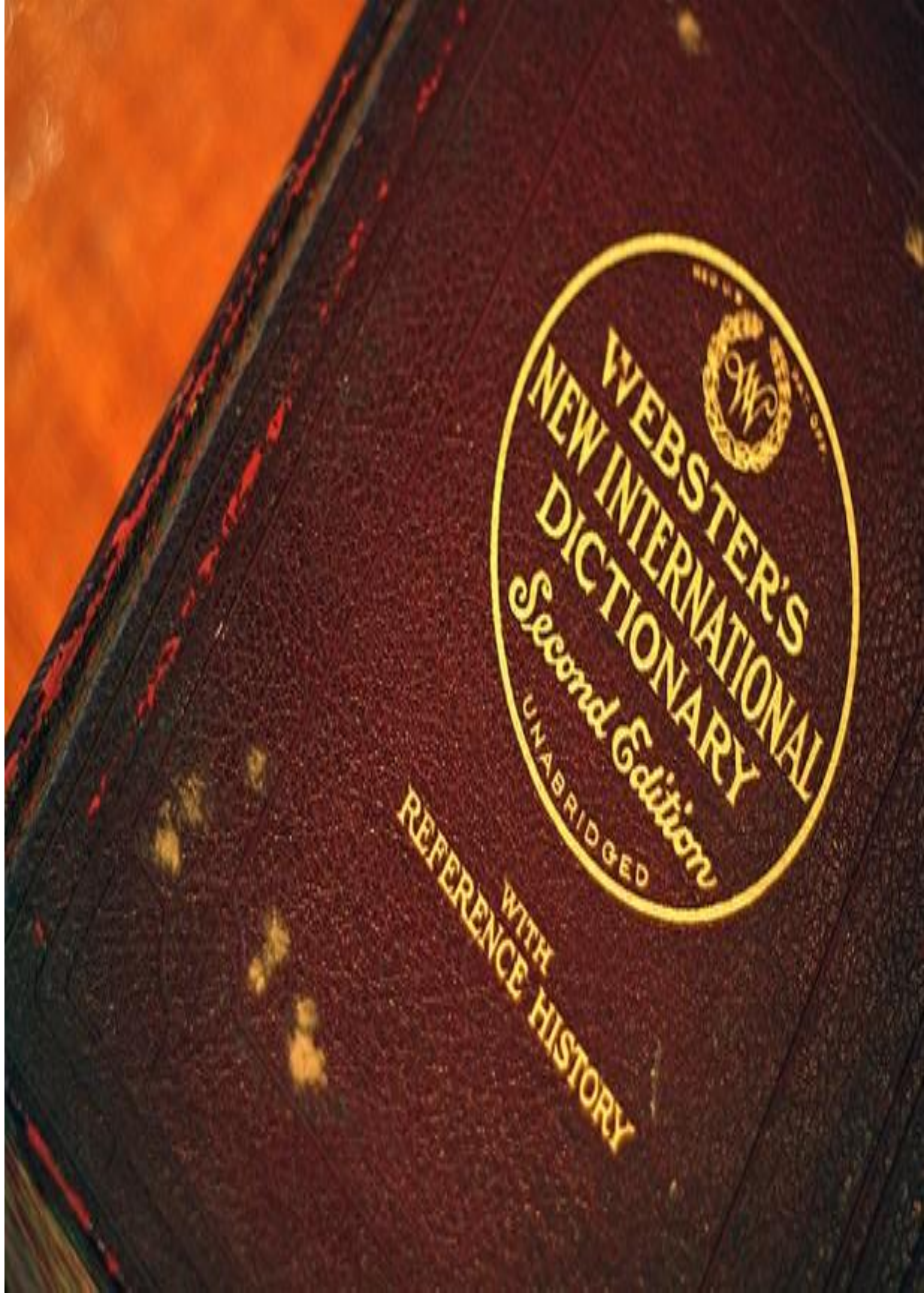
OS: Operating System

Kernel: Linux operating system internals (not a CUDA kernel!)

Page: 4KB, 64KB, 2MB, etc.of physically contiguous memory. Smallest unit handled by the OS.

Page table: sparse tree containing virtual-to-physical address translations

Page table entry: a single (page's worth of) virtual-to-physical translation



Definitions

OS: Operating System

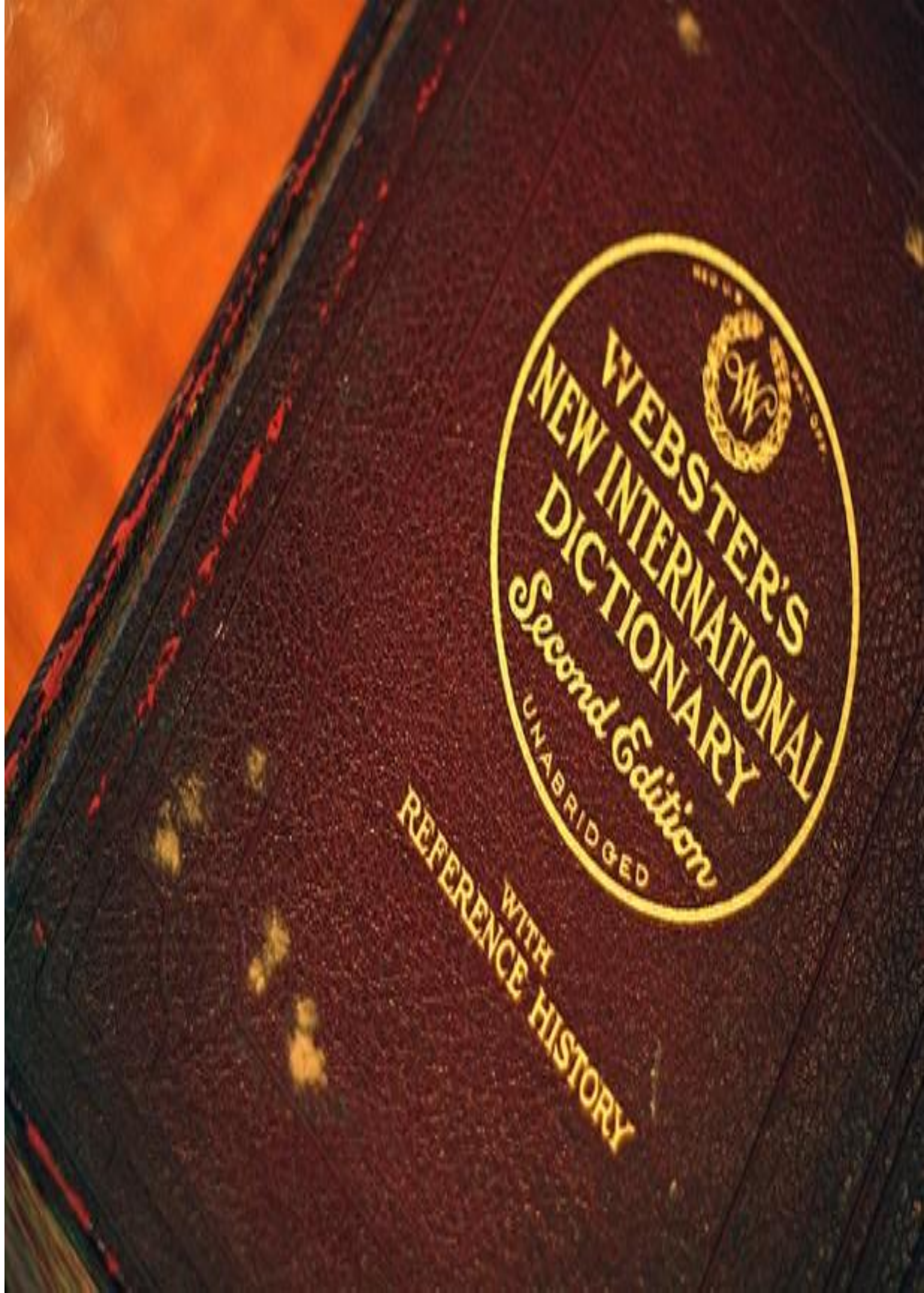
Kernel: Linux operating system internals (not a CUDA kernel!)

Page: 4KB, 64KB, 2MB, etc.of physically contiguous memory. Smallest unit handled by the OS.

Page table: sparse tree containing virtual-to-physical address translations

Page table entry: a single (page's worth of) virtual-to-physical translation

To map a (physical) page: create a page table entry for that page.



Definitions

OS: Operating System

Kernel: Linux operating system internals (not a CUDA kernel!)

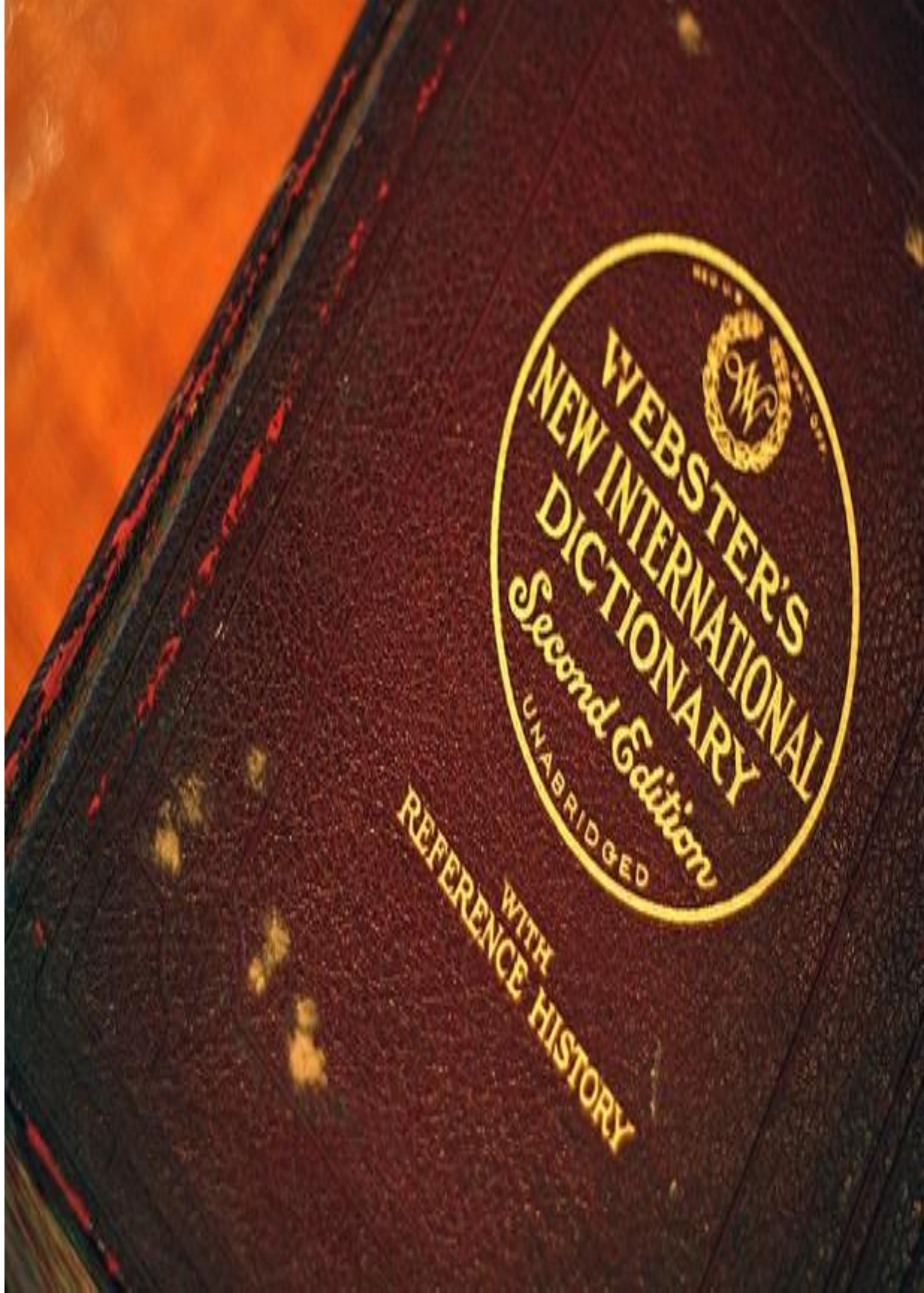
Page: 4KB, 64KB, 2MB, etc.of physically contiguous memory. Smallest unit handled by the OS.

Page table: sparse tree containing virtual-to-physical address translations

Page table entry: a single (page's worth of) virtual-to-physical translation

To map a (physical) page: create a page table entry for that page.

Unmap: remove a page table entry. Subsequent program accesses will cause page faults.



Definitions

OS: Operating System

Kernel: Linux operating system internals (not a CUDA kernel!)

Page: 4KB, 64KB, 2MB, etc.of physically contiguous memory. Smallest unit handled by the OS.

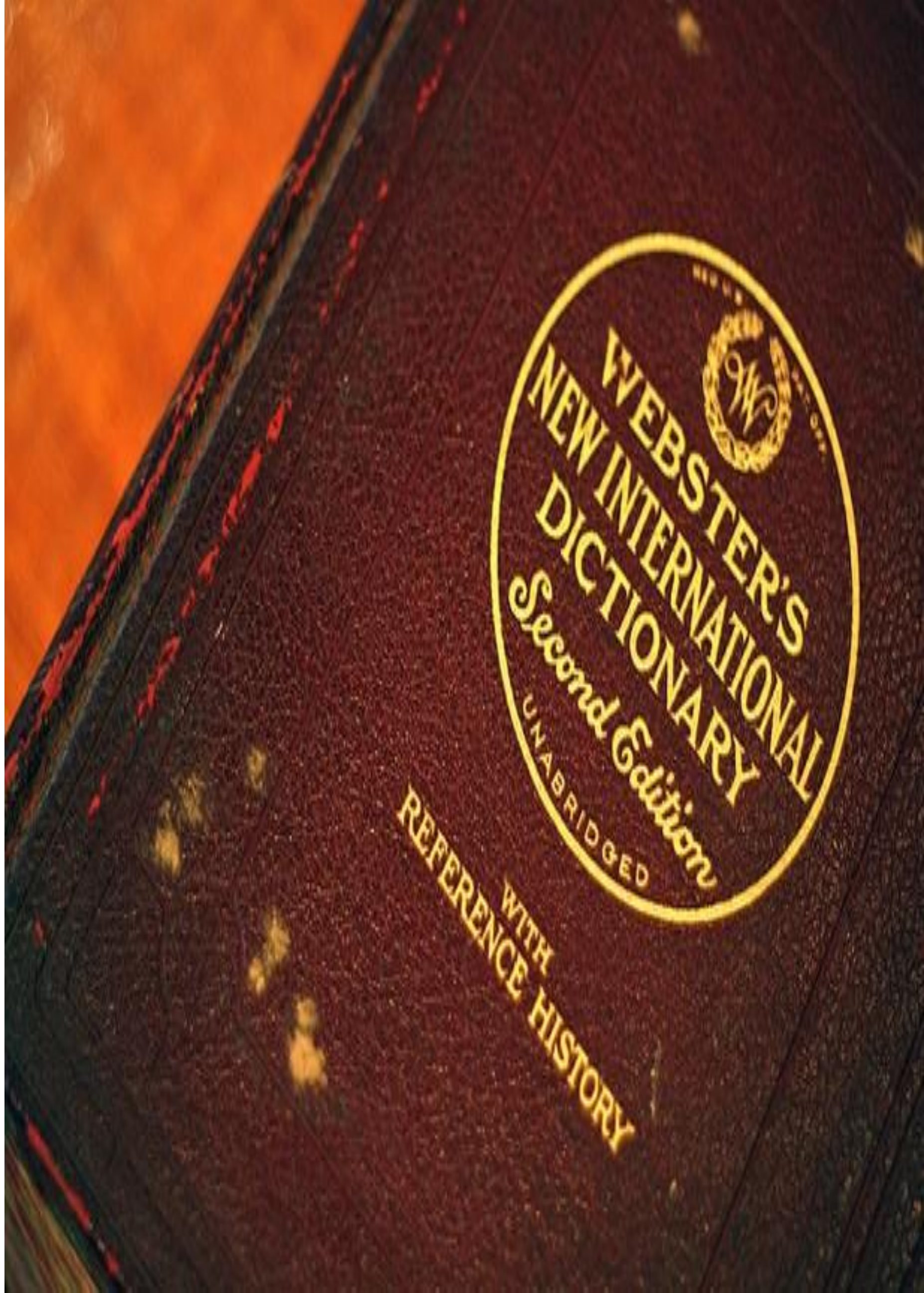
Page table: sparse tree containing virtual-to-physical address translations

Page table entry: a single (page's worth of) virtual-to-physical translation

To map a (physical) page: create a page table entry for that page.

Unmap: remove a page table entry. Subsequent program accesses will cause page faults.

Page fault: a CPU (or GPU) exception caused by a missing page table entry for a virtual address.



Definitions

OS: Operating System

Kernel: Linux operating system internals (not a CUDA kernel!)

Page: 4KB, 64KB, 2MB, etc.of physically contiguous memory. Smallest unit handled by the OS.

Page table: sparse tree containing virtual-to-physical address translations

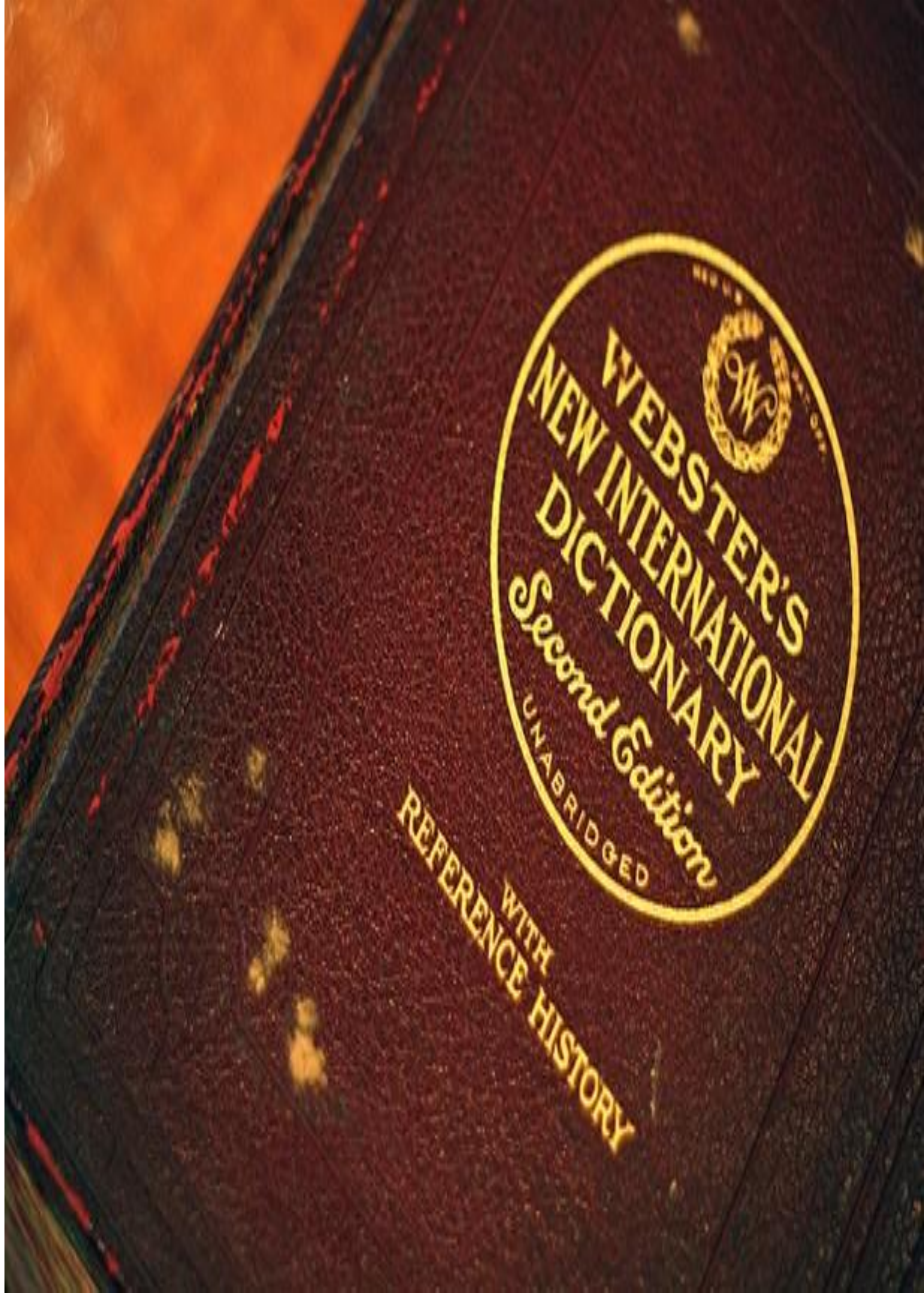
Page table entry: a single (page's worth of) virtual-to-physical translation

To map a (physical) page: create a page table entry for that page.

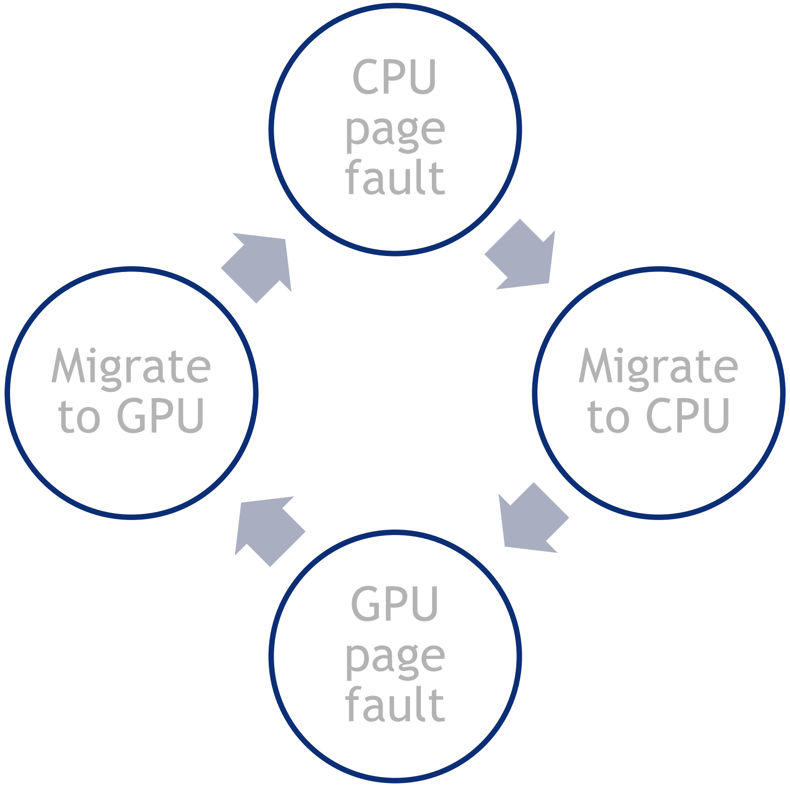
Unmap: remove a page table entry. Subsequent program accesses will cause page faults.

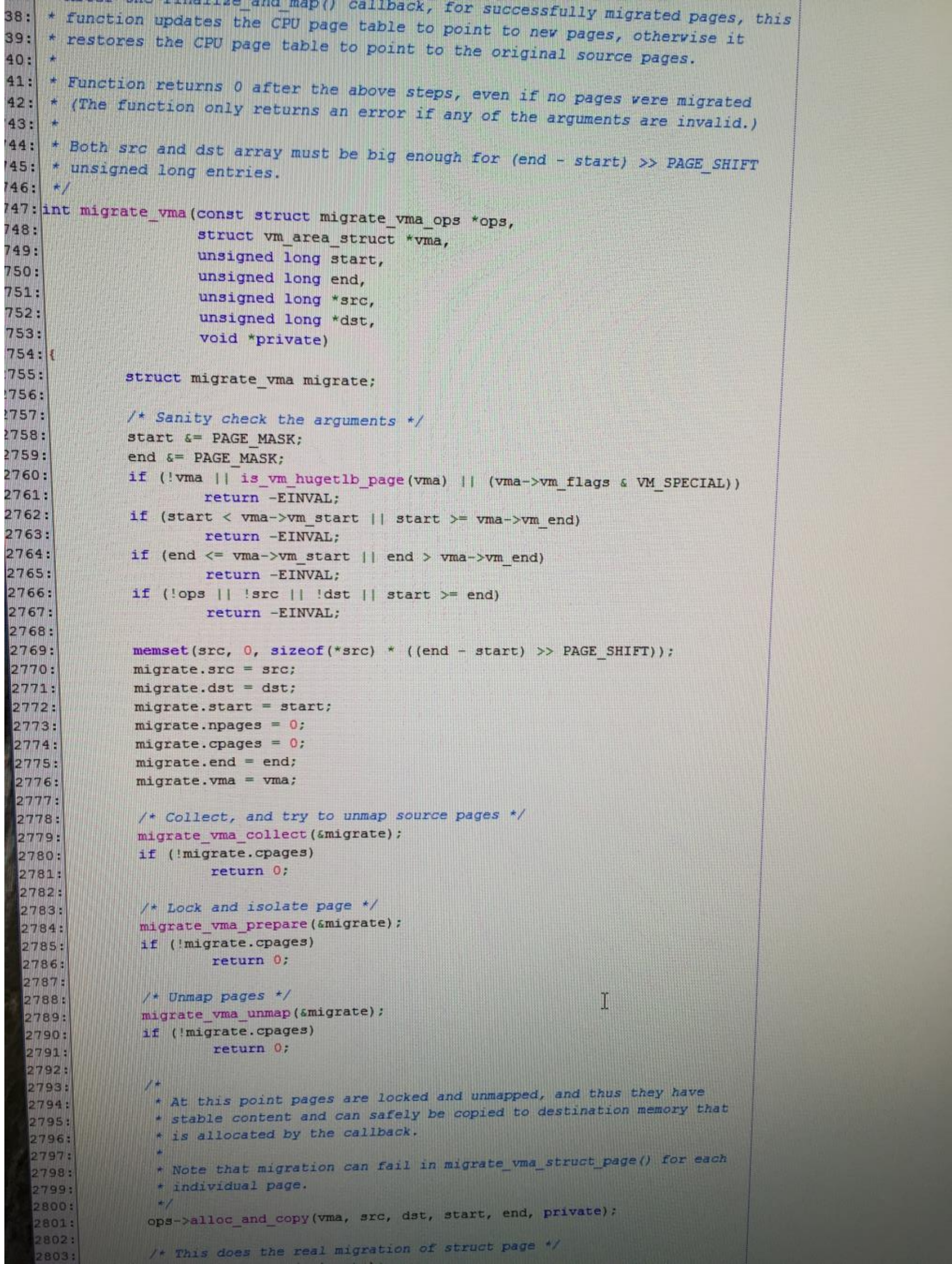
Page fault: a CPU (or GPU) exception caused by a missing page table entry for a virtual address.

Page migration: unmap a page from CPU, copy to GPU, map on GPU (or the reverse). Also GPU-to-GPU.



How HMM works - 1





```c
 * map() callback, for successfully migrated pages, this
 * function updates the CPU page table to point to new pages, otherwise it
 * restores the CPU page table to point to the original source pages.
 *
 * Function returns 0 after the above steps, even if no pages were migrated
 * (The function only returns an error if any of the arguments are invalid.)
 *
 * Both src and dst array must be big enough for (end - start) >> PAGE_SHIFT
 * unsigned long entries.
 */
int migrate_vma(const struct migrate_vma_ops *ops,
                struct vm_area_struct *vma,
                unsigned long start,
                unsigned long end,
                unsigned long *src,
                unsigned long *dst,
                void *private)
{
        struct migrate_vma migrate;

        /* Sanity check the arguments */
        start &= PAGE_MASK;
        end &= PAGE_MASK;
        if (!vma || is_vm_hugetlb_page(vma) || (vma->vm_flags & VM_SPECIAL))
                return -EINVAL;
        if (start < vma->vm_start || start >= vma->vm_end)
                return -EINVAL;
        if (end <= vma->vm_start || end > vma->vm_end)
                return -EINVAL;
        if (!ops || !src || !dst || start >= end)
                return -EINVAL;

        memset(src, 0, sizeof(*src) * ((end - start) >> PAGE_SHIFT));
        migrate.src = src;
        migrate.dst = dst;
        migrate.start = start;
        migrate.npages = 0;
        migrate.cpages = 0;
        migrate.end = end;
        migrate.vma = vma;

        /* Collect, and try to unmap source pages */
        migrate_vma_collect(&migrate);
        if (!migrate.cpages)
                return 0;

        /* Lock and isolate page */
        migrate_vma_prepare(&migrate);
        if (!migrate.cpages)
                return 0;

        /* Unmap pages */
        migrate_vma_unmap(&migrate);
        if (!migrate.cpages)
                return 0;

        /*
         * At this point pages are locked and unmapped, and thus they have
         * stable content and can safely be copied to destination memory that
         * is allocated by the callback.
         *
         * Note that migration can fail in migrate_vma_struct_page() for each
         * individual page.
         */
        ops->alloc_and_copy(vma, src, dst, start, end, private);

        /* This does the real migration of struct page */
```

How HMM works - 2

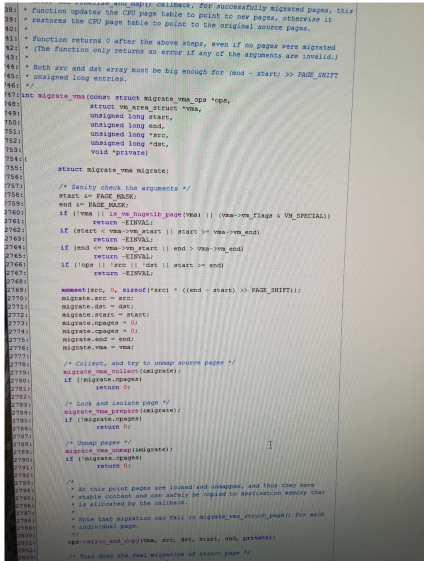
CPU page fault occurs

HMM receives page fault, calls UM driver

UM copies page data to GPU, unmaps from GPU

HMM maps page to CPU

OS kernel resumes CPU code



How HMM works - 3

GPU page fault occurs

UM driver receives page fault

UM driver fails to find page in its records

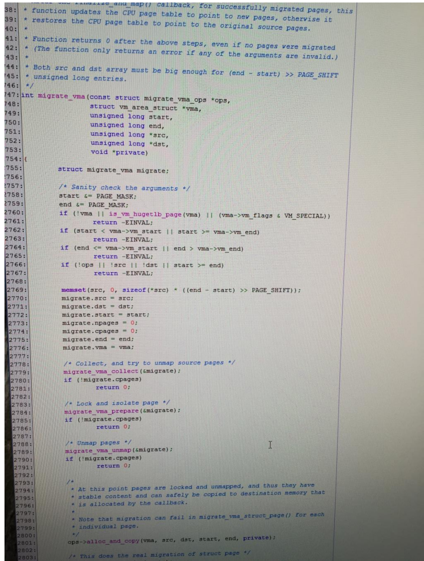
UM asks HMM about the page, HMM has a malloc record of the page

UM tells HMM that page will be migrated from CPU to GPU

HMM unmaps page from CPU

UM copies page data to GPU

UM causes GPU to resume execution (“replays” the page fault)



Profiling with Unified Memory + HMM

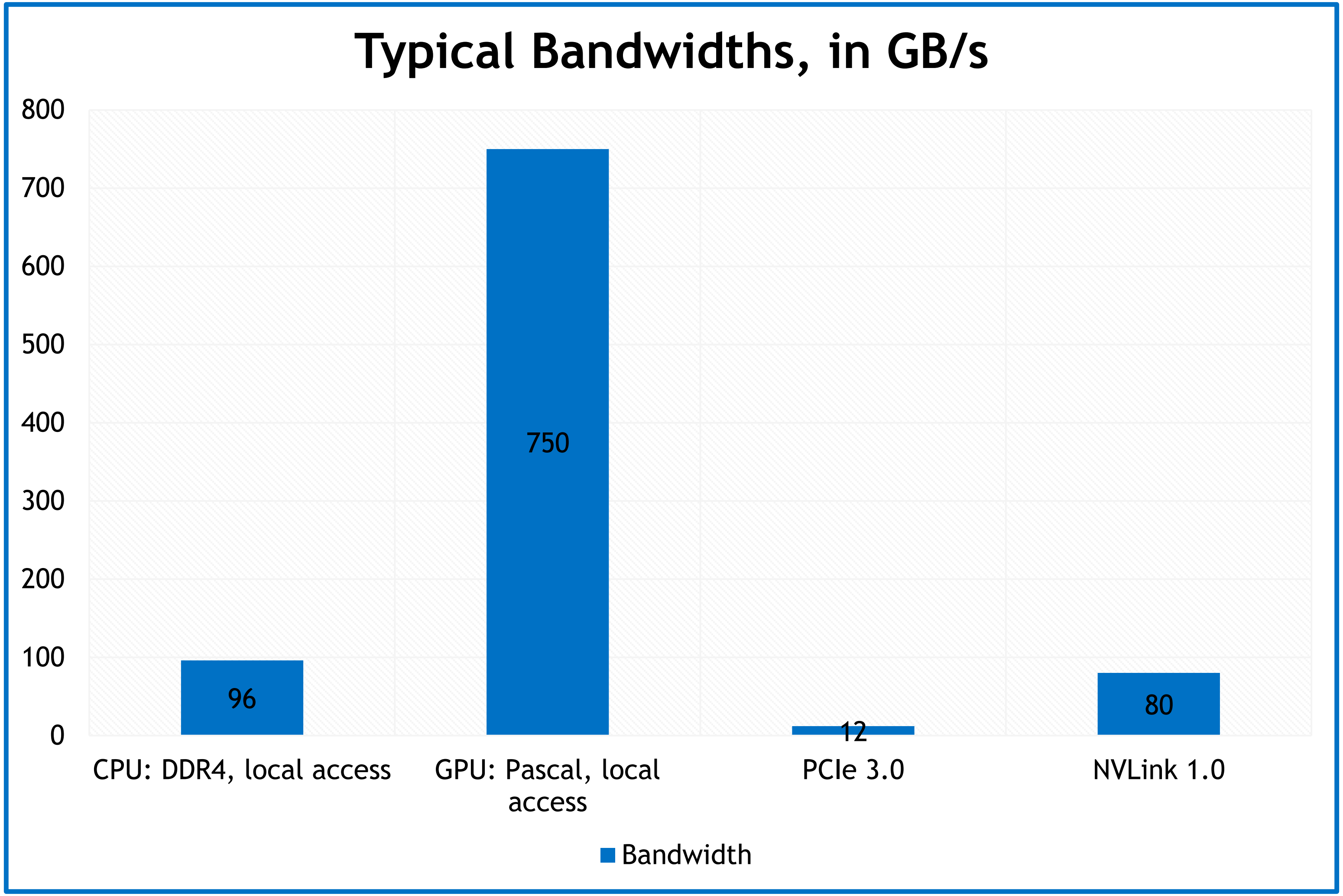
This is the code that we are profiling, in the next slide:

Unified Memory + HMM

```
#include <stdio.h>
#define LEN sizeof(int)

__global__ void
compute_this(int *pDataFromCpu)
{
   atomicAdd(pDataFromCpu, 1);
}

int main(void)
{
   int *pData = (int*)malloc(LEN);
   *pData = 1;

   compute_this<<<512,1000>>>(pData);
   cudaDeviceSynchronize();

   printf("Results: %d\n", *pData);
   free(pData);
   return 0;
}
```

Profiling with Unified Memory + HMM: nvprof

```
$ /usr/local/cuda/bin/nvprof --unified-memory-profiling per-process-device ./hmm_app
==19835== NVPROF is profiling process 19835, command: ./hmm_app
Results: 512001
==19835== Profiling application: ./hmm_app
==19835== Profiling result:
Time(%) Time Calls Avg Min Max Name
100.00% 1.2904ms 1 1.2904ms 1.2904ms 1.2904ms compute_this(int*)


==19835== Unified Memory profiling result:
Device "GeForce GTX 1050 Ti (0)"
  Count Avg Size Min Size Max Size Total Size Total Time Name
  2 32.000KB 4.0000KB 60.000KB 64.00000KB 42.62400us Host To Device
  2 32.000KB 4.0000KB 60.000KB 64.00000KB 37.98400us Device To Host
  1 - - - - 1.179410ms GPU Page fault groups
Total CPU Page faults: 2


==19835== API calls:
Time(%) Time Calls Avg Min Max Name
 98.88% 388.41ms 1 388.41ms 388.41ms 388.41ms cudaMallocManaged
  0.39% 1.5479ms 190 8.1470us 768ns 408.58us cuDeviceGetAttribute
  0.33% 1.3125ms 1 1.3125ms 1.3125ms 1.3125ms cudaDeviceSynchronize
  0.19% 739.71us 2 369.86us 363.81us 375.90us cuDeviceTotalMem
  0.13% 524.45us 1 524.45us 524.45us 524.45us cudaFree
  0.04% 137.87us 1 137.87us 137.87us 137.87us cudaLaunch
  0.03% 126.84us 2 63.417us 58.109us 68.726us cuDeviceGetName
  0.00% 11.524us 1 11.524us 11.524us 11.524us cudaConfigureCall
  0.00% 6.4950us 1 6.4950us 6.4950us 6.4950us cudaSetupArgument
  0.00% 6.2160us 6 1.0360us 768ns 1.2570us cuDeviceGet
  0.00% 4.5400us 3 1.5130us 838ns 2.6540us cuDeviceGetCount
```


Typical Bandwidths, in GB/s
800
700
600
500
400
300
200
100
0
96
750
12
80
CPU: DDR4, local access
GPU: Pascal, local access
PCIe 3.0
NVLink 1.0
Bandwidth

Tuning still works

cudaMemPrefetchAsync: this is the new cudaMemcpy

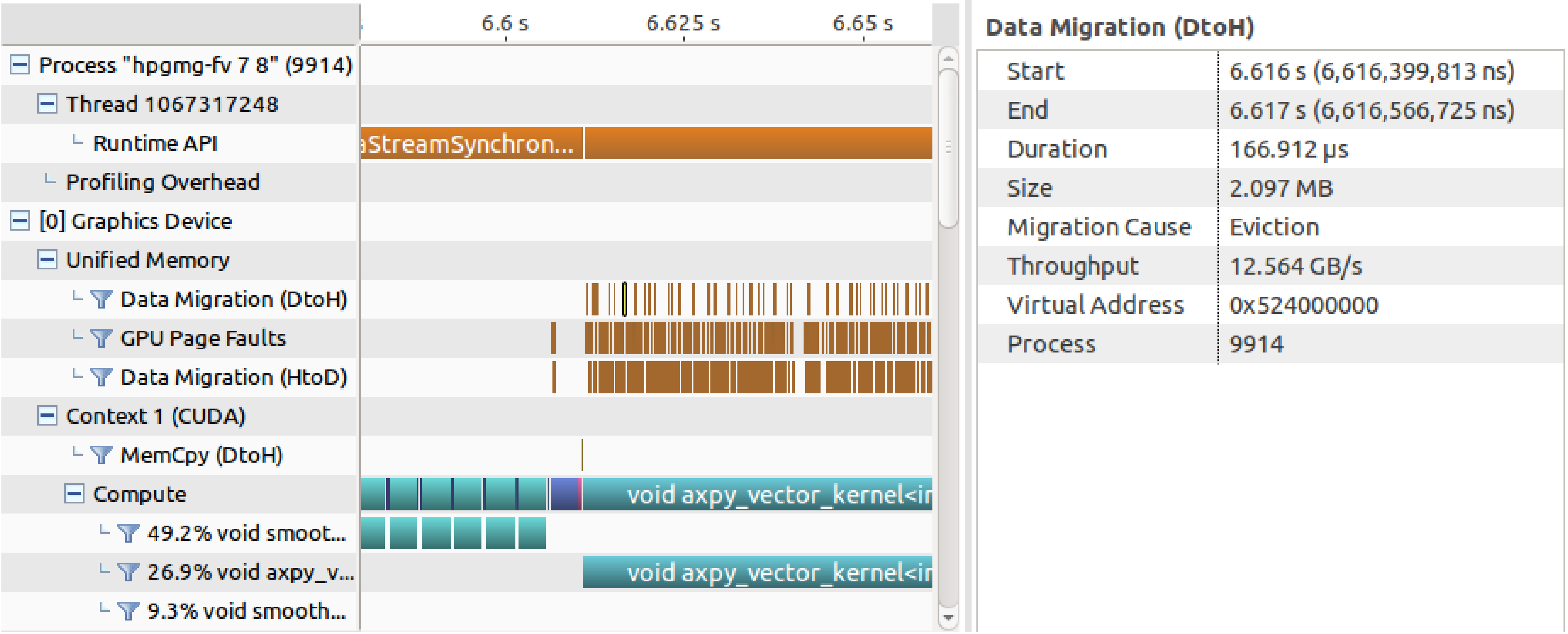
cudaMemAdvise

cudaMemAdviseSetReadMostly

cudaMemAdviseSetPreferredLocation

cudaMemAdviseSetAccessedBy

Profiling with Unified Memory: Visual Profiler



Source: https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal

HMM History

HMM History

Prehistoric: Pascal replayable page faulting hardware is envisioned and spec'd out

2012: discussions with Red Hat, Jerome Glisse begin

April, 2014: CUDA 6.0: First ever release of Unified Memory, CPU page faults but no GPU page faults. Works surprisingly well…

May, 2014: HMM v1 posted to linux-mm and linux-kernel

November, 2014: HMM patchset review: Linus Torvalds: "NONE OF WHAT YOU SAY MAKES ANY SENSE"

Mid-2016: Pascal GPUs become available (a Linux kernel prerequisite)

March, 2017: linux-mm summit: HMM a major topic of discussion

May, 2017: HMM v21 posted (3 year anniversary)

References

https://devblogs.nvidia.com/parallelforall/inside-pascal/
https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal/
http://docs.nvidia.com/cuda/cuda-c-programming-guide
http://www.spinics.net/lists/linux-mm/msg126148.html (HMM v21 patchset)

Conclusion

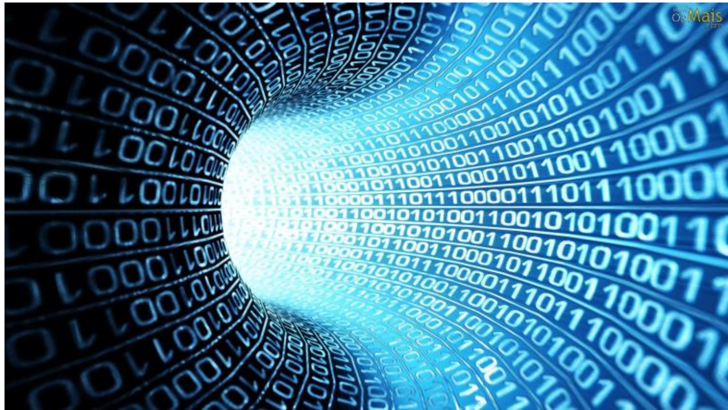
Conclusion: what you've learned

HMM is a Linux kernel patch + support in NVIDIA's driver

HMM memory acts just like UM

HMM uses page faults just like UM

Profiling and tuning still work the same as UM



Conclusion: what to do next

Write a small HMM-ready program

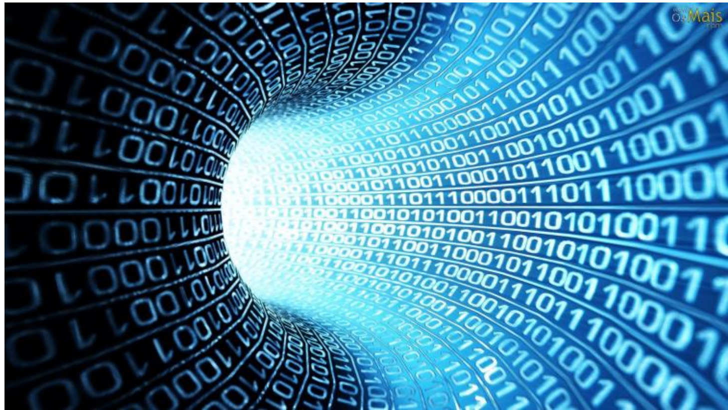
Run nvprof and look at page faults

Run nvvp and look at page faults

Port a CUDA program to HMM

Talk to me about HMM at the GTC party

Questions and Answers



NVIDIA