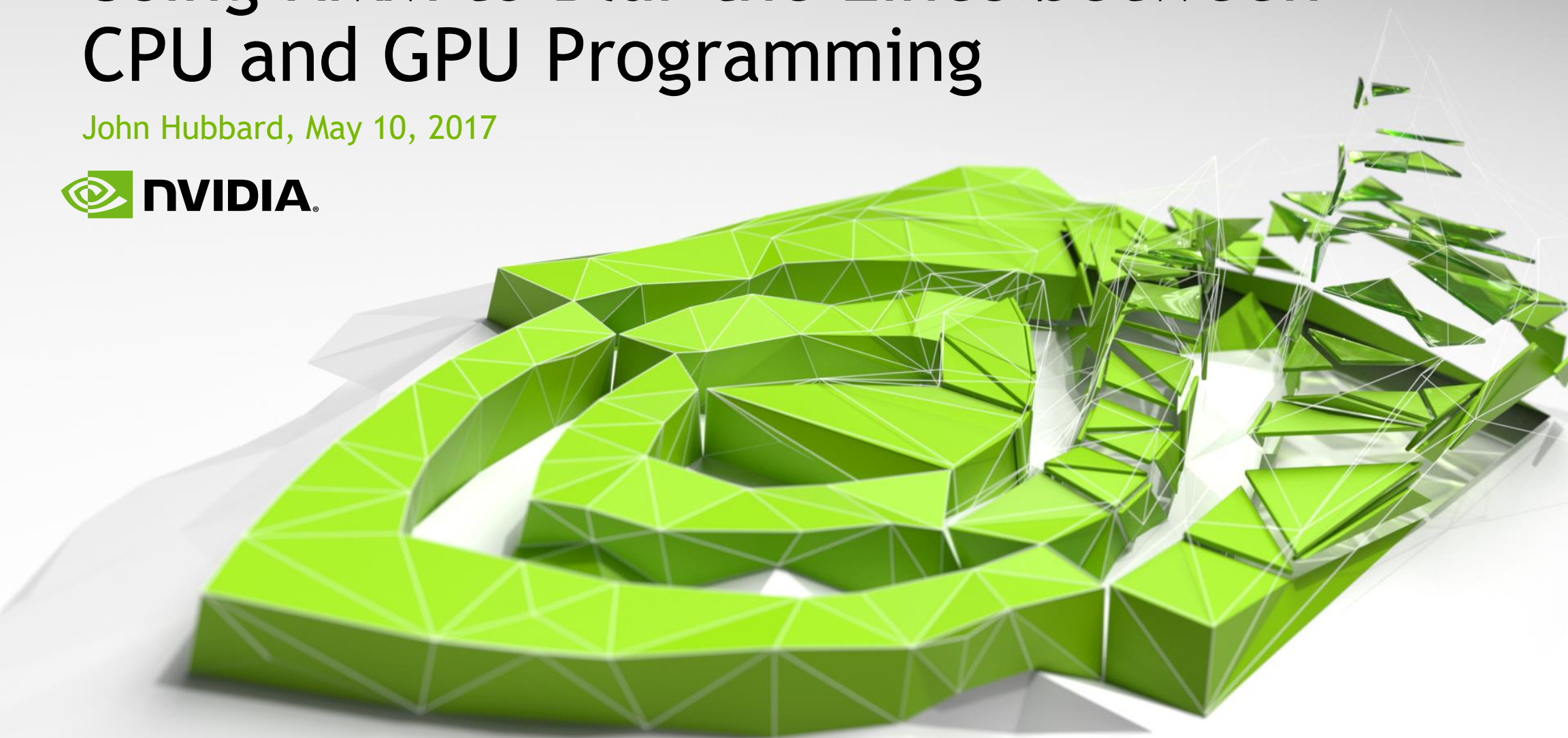
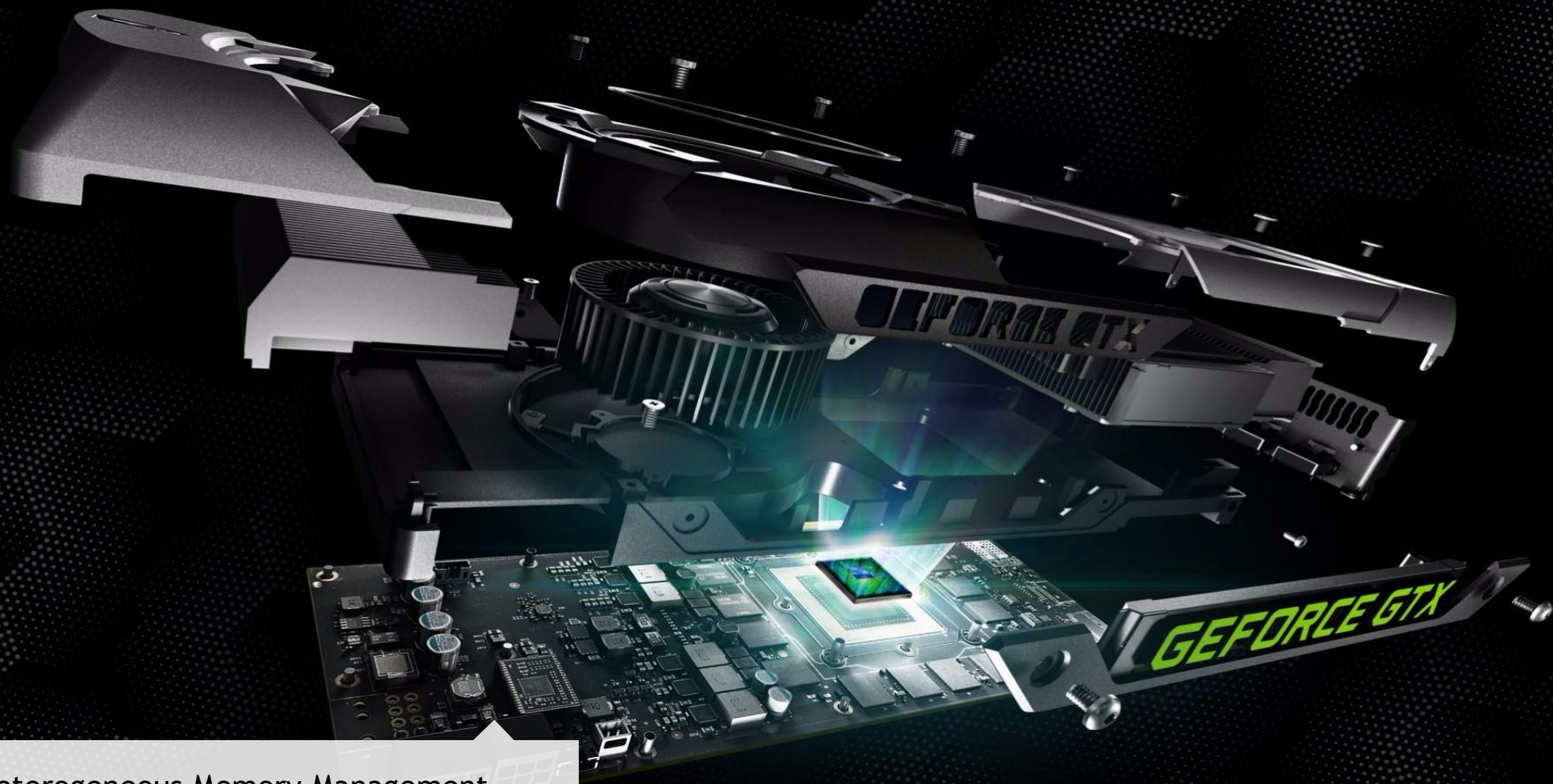


Using HMM to Blur the Lines between CPU and GPU Programming

John Hubbard, May 10, 2017





Heterogeneous Memory Management Overview

Agenda

Overview

HMM Benefits

SW-HW stack: where does HMM fit in?

Definitions

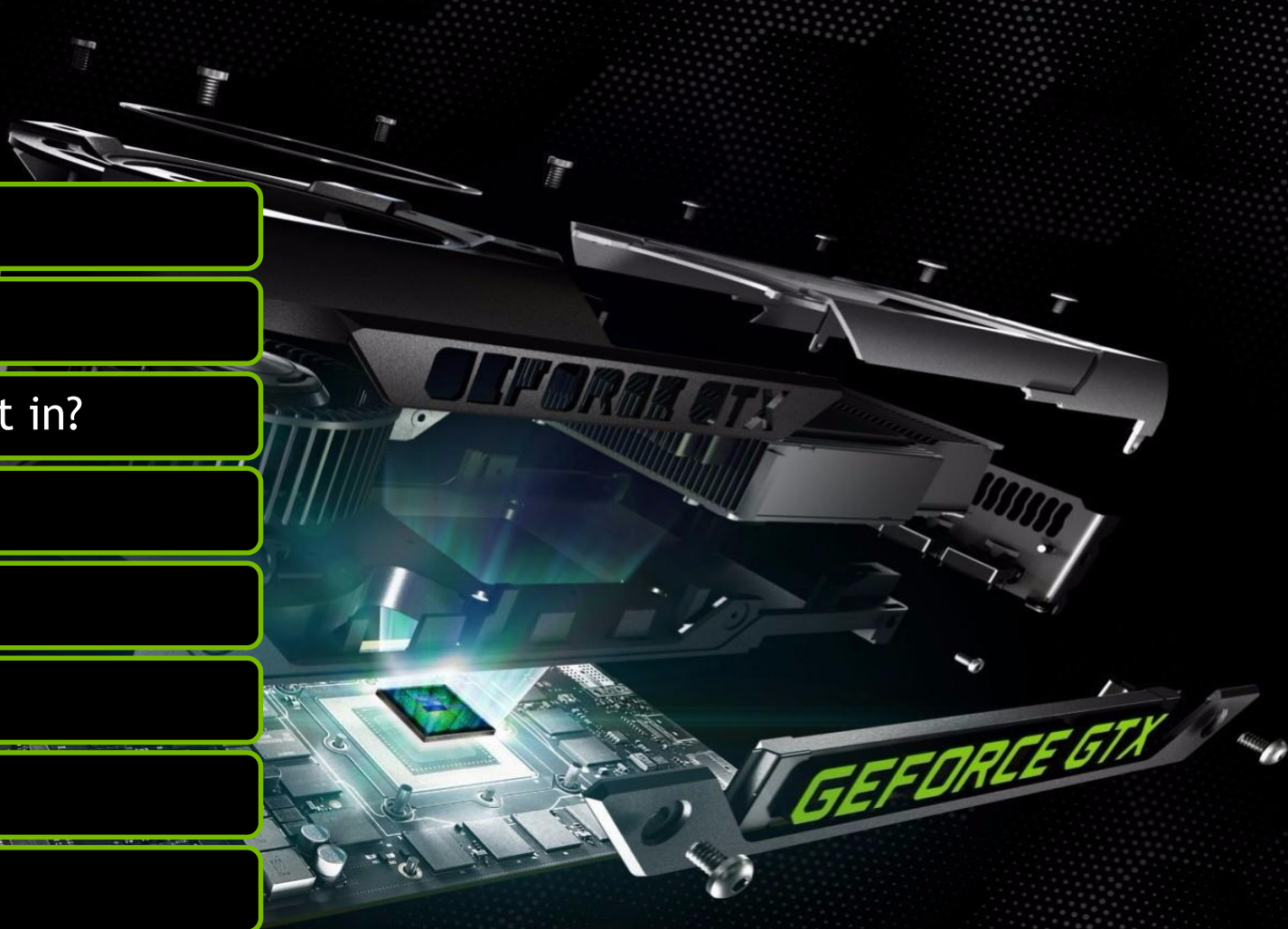
How HMM works

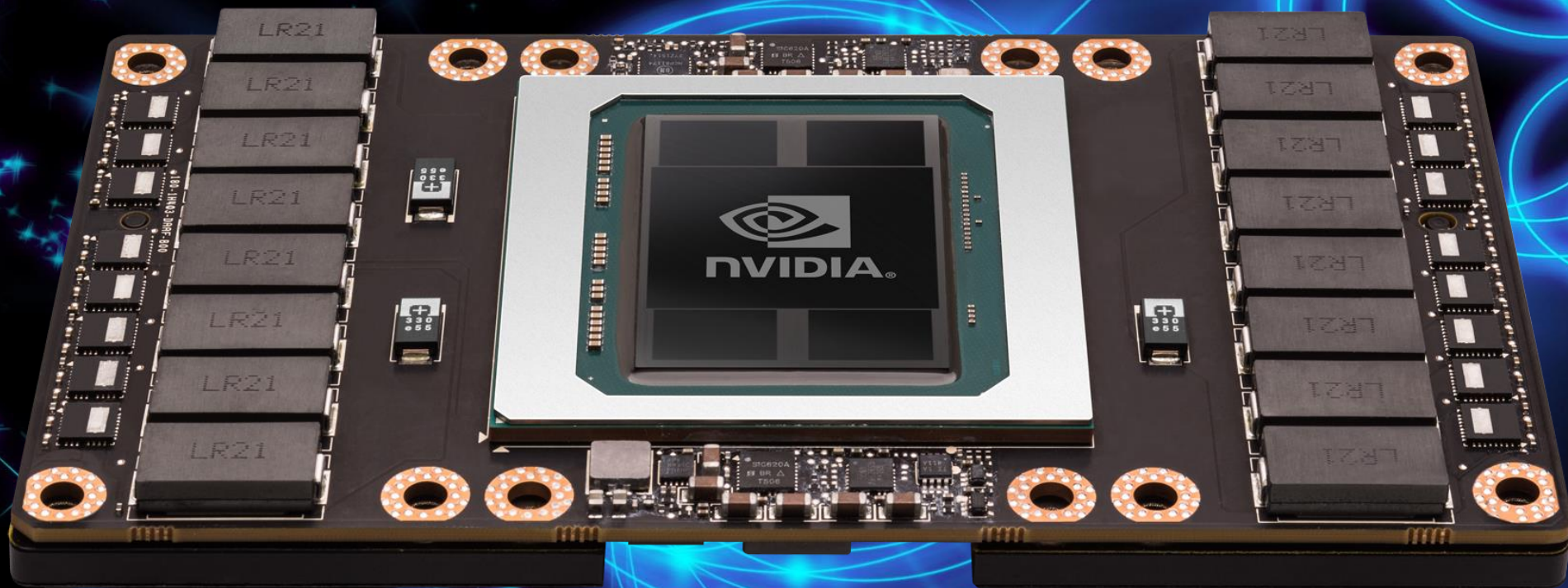
Profiling with HMM

A little bit of history

References

Conclusion

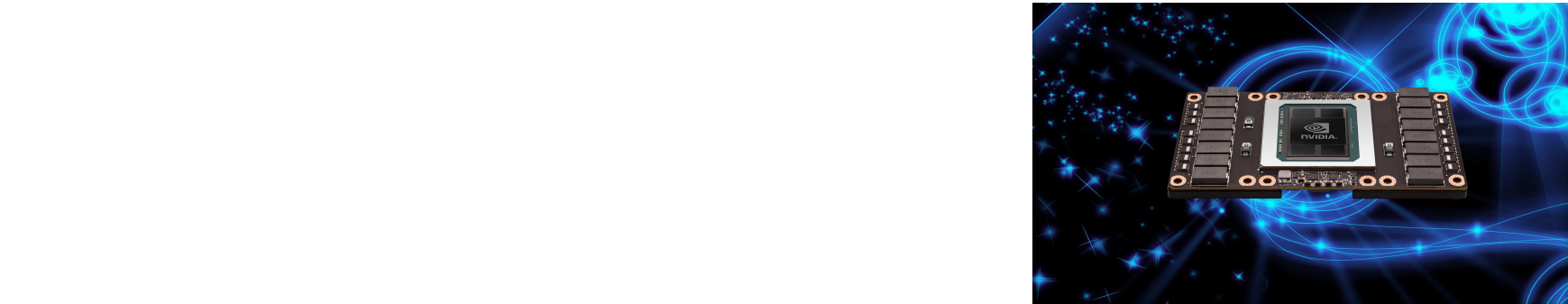




HMM Benefits

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;

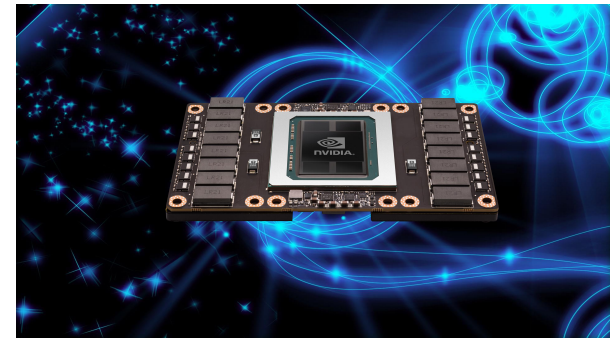
    compute_this<<<512,1000>>>>(pData);
    cudaDeviceSynchronize();

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

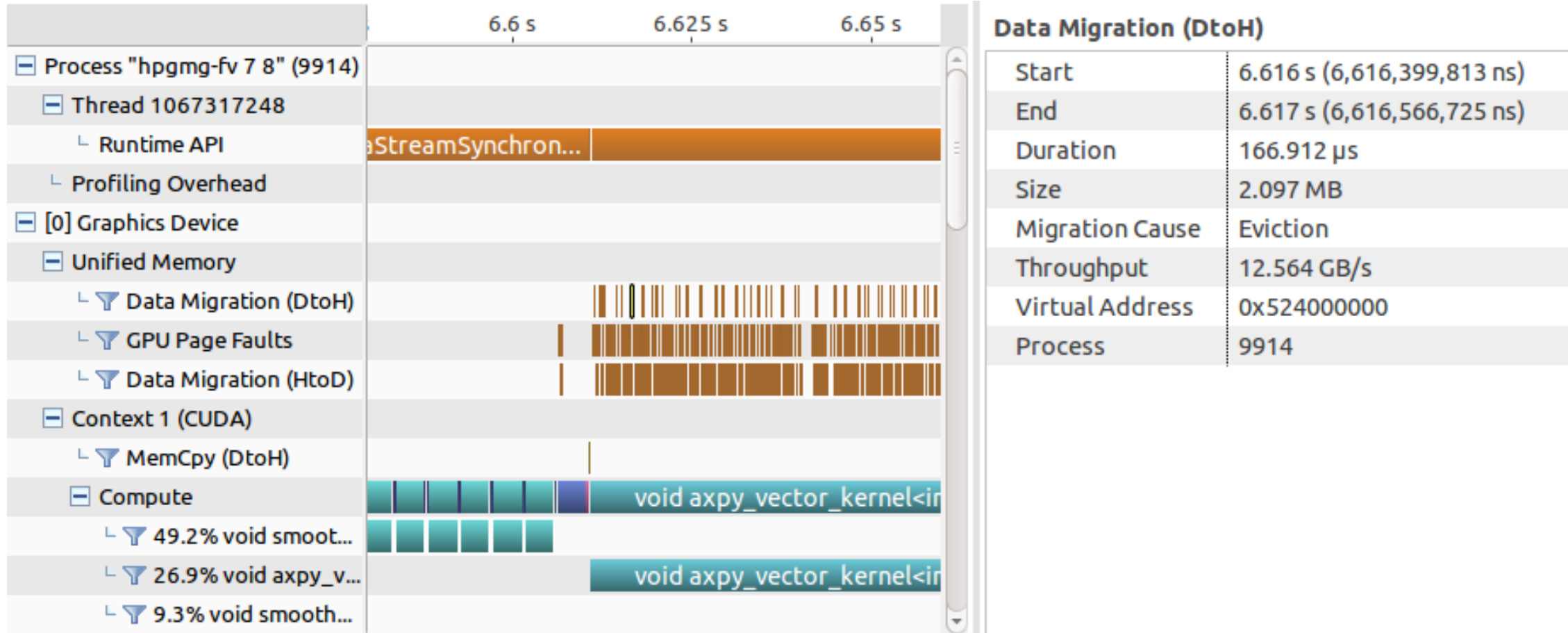
HMM Benefits

Simpler code

Code is still tunable



Profiling with Unified Memory: Visual Profiler



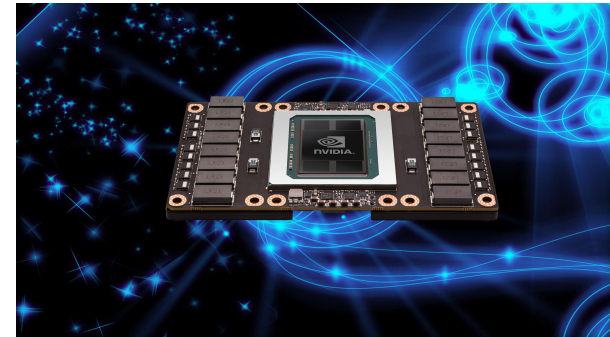
Source: <https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal>

HMM Benefits

Simpler code

Code is still tunable

Libraries can be used without changing them



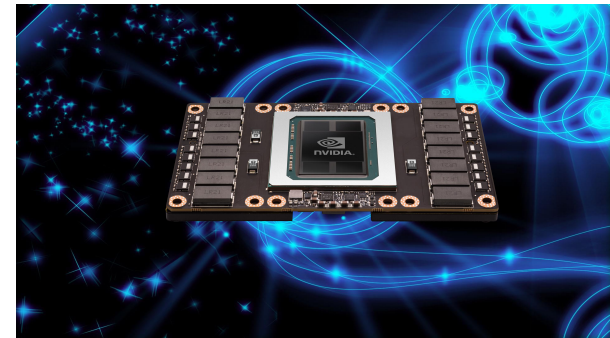
HMM Benefits

Simpler code

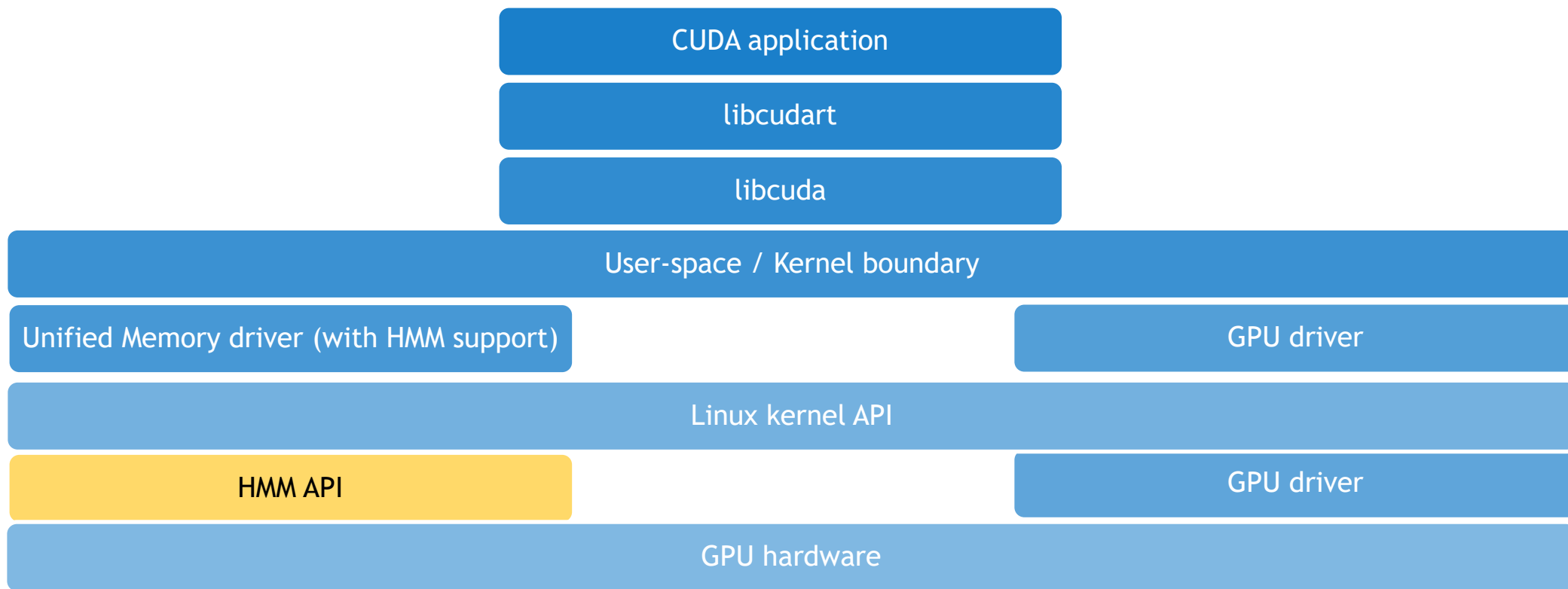
Code is still tunable

Libraries can be used without changing them

New programming languages are easily supported

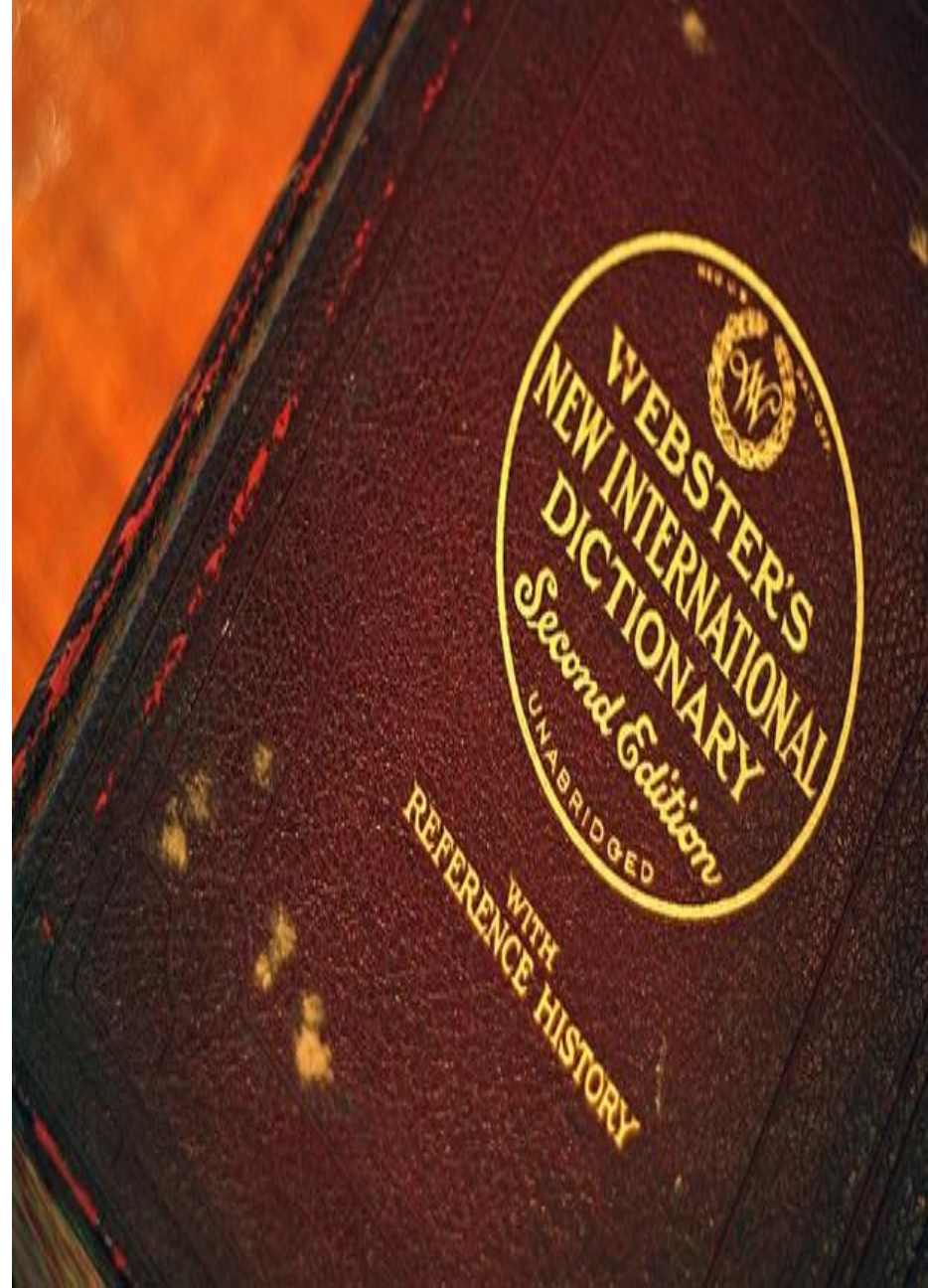


SW-HW stack: where does HMM fit in?



Definitions

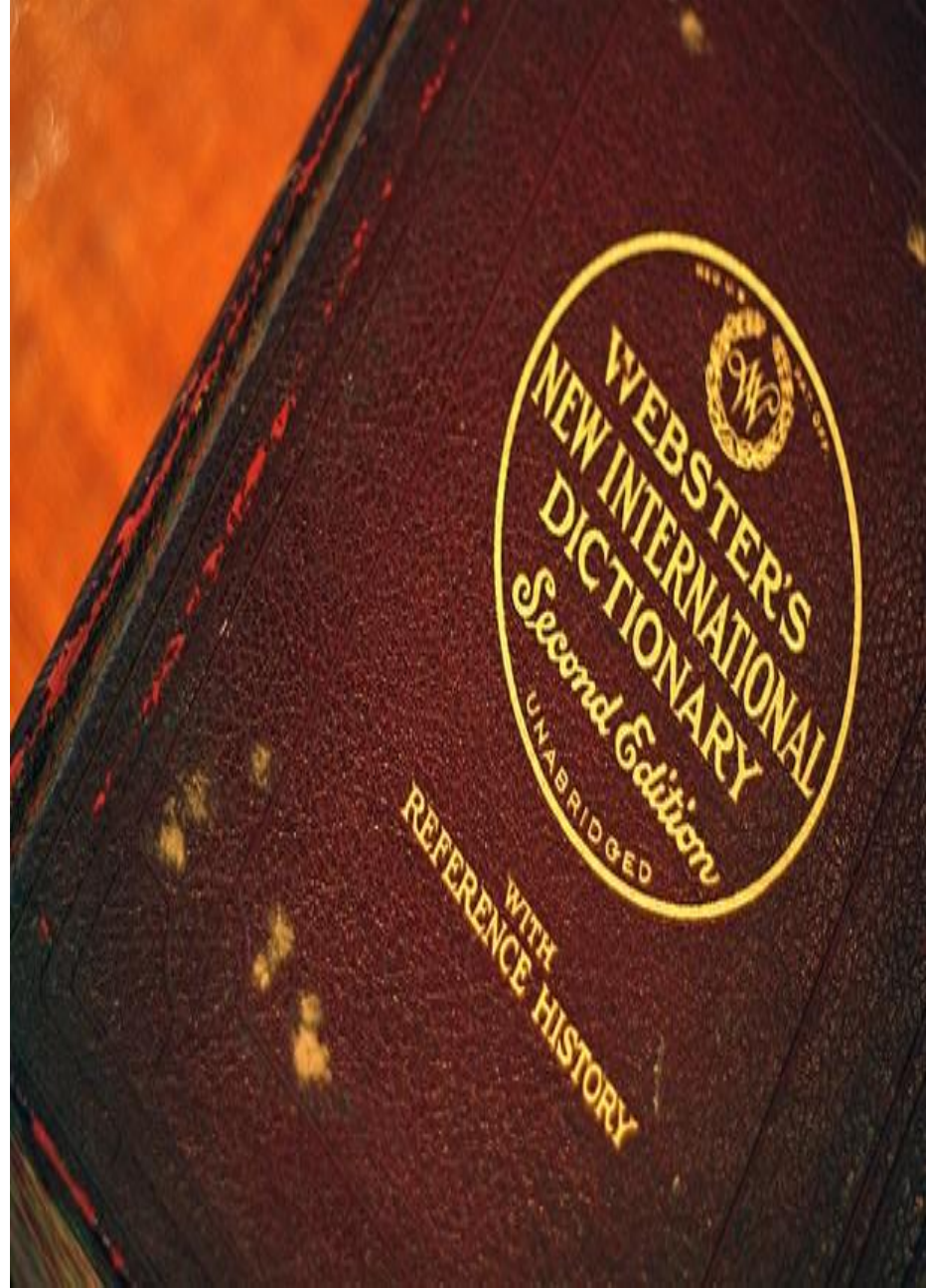
OS: Operating System



Definitions

OS: Operating System

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

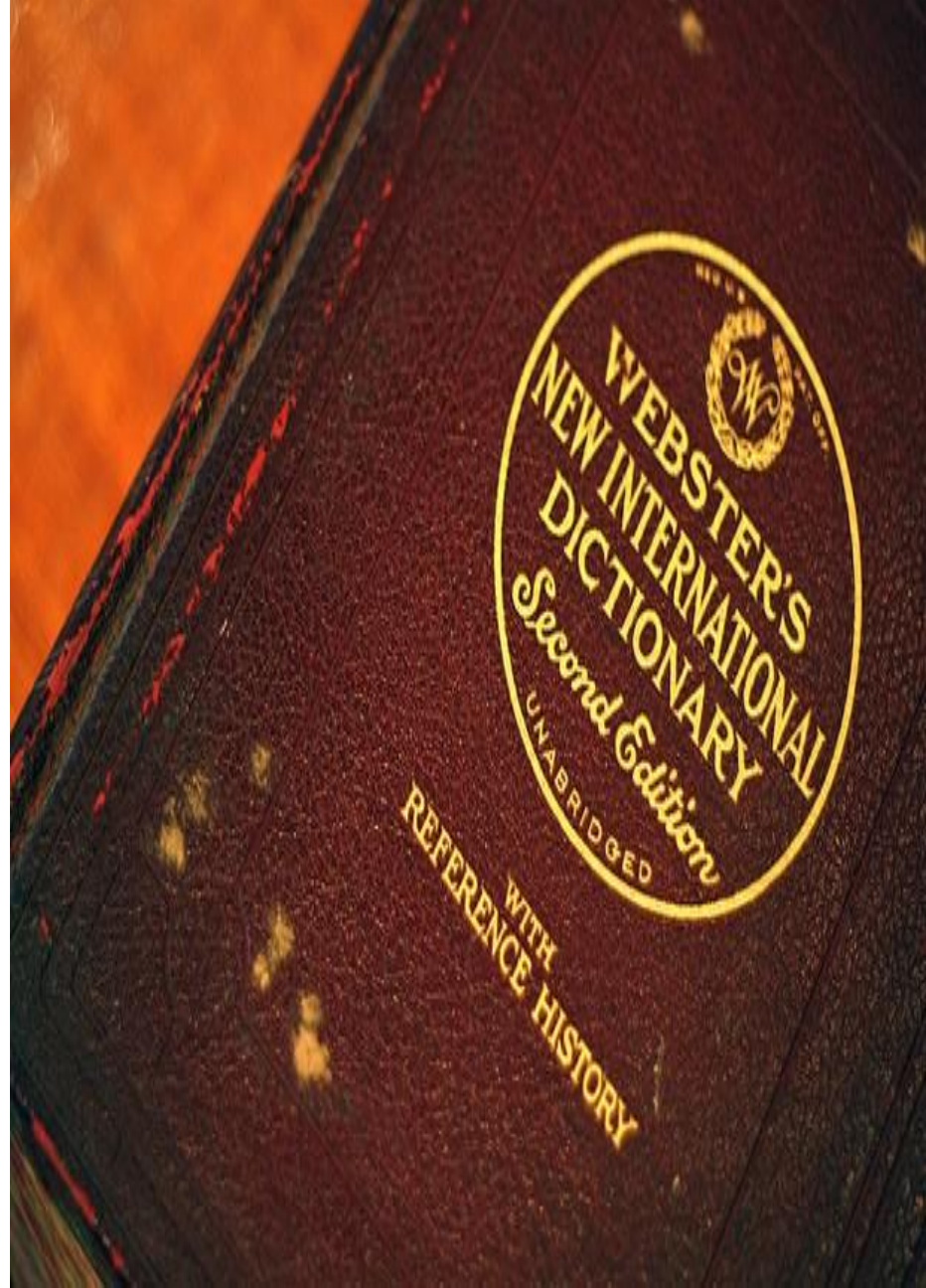


Definitions

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

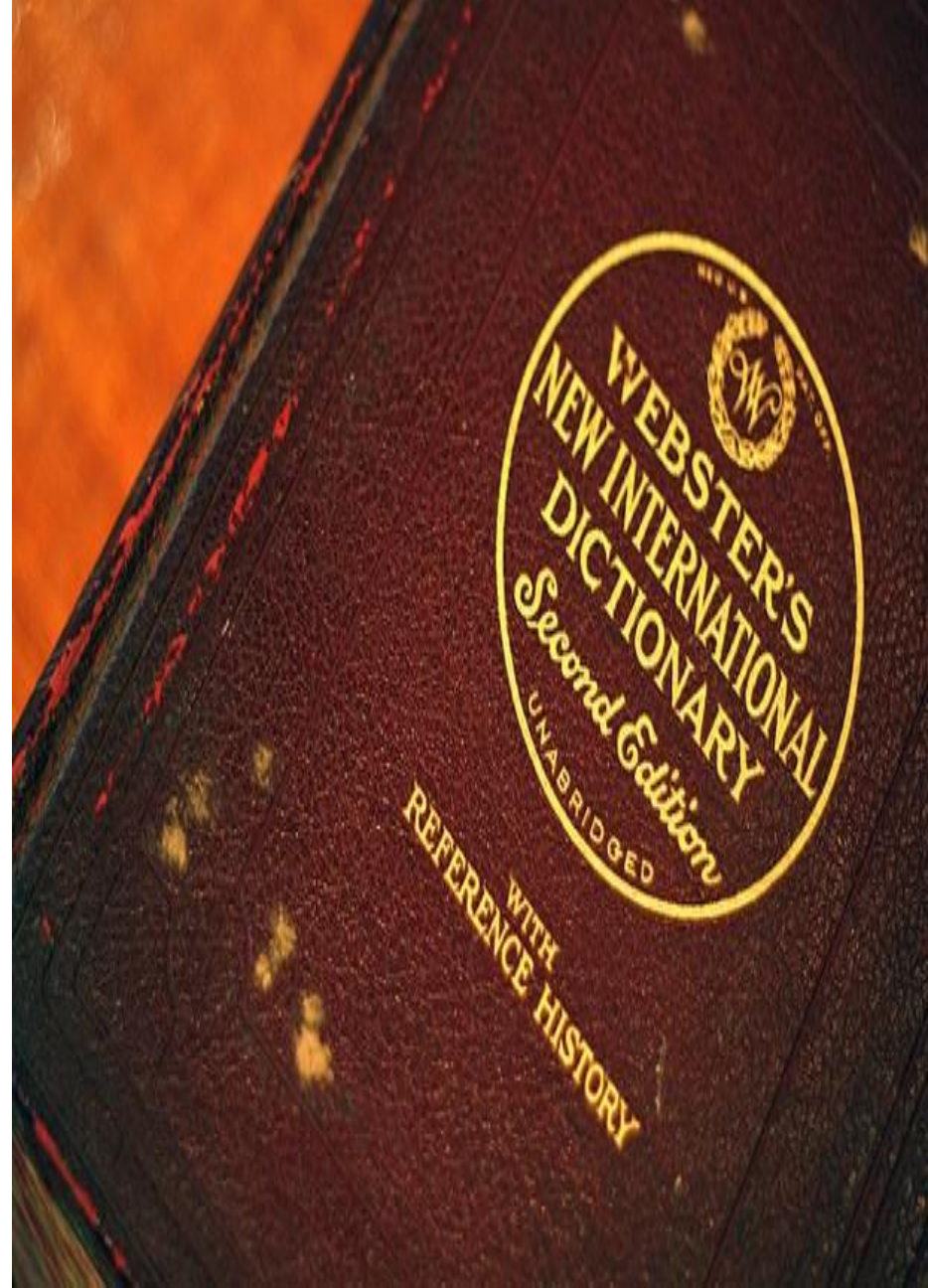
OS: Operating System

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

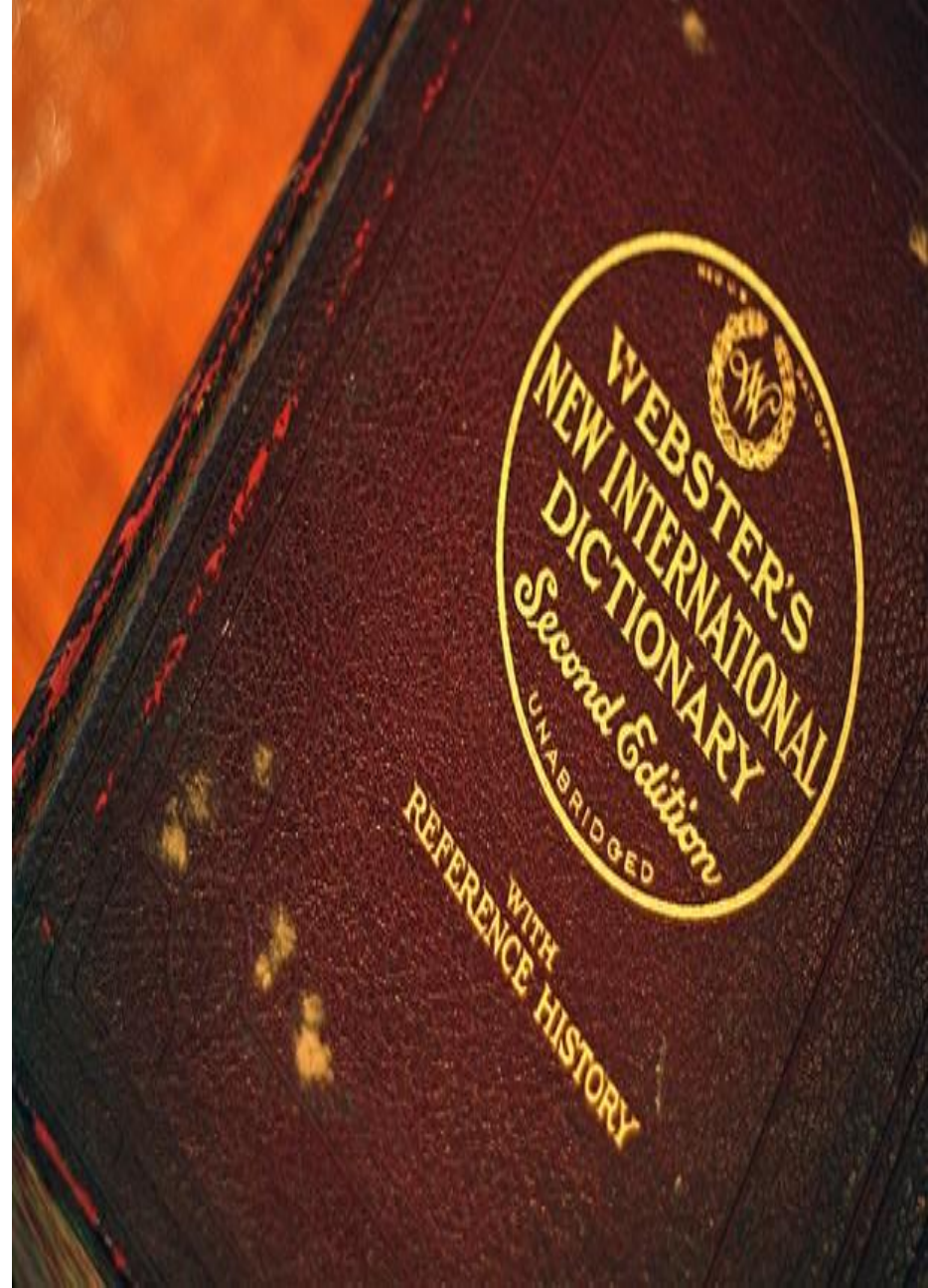
Kernel: Linux operating system internals (not a CUDA kernel!)

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!)

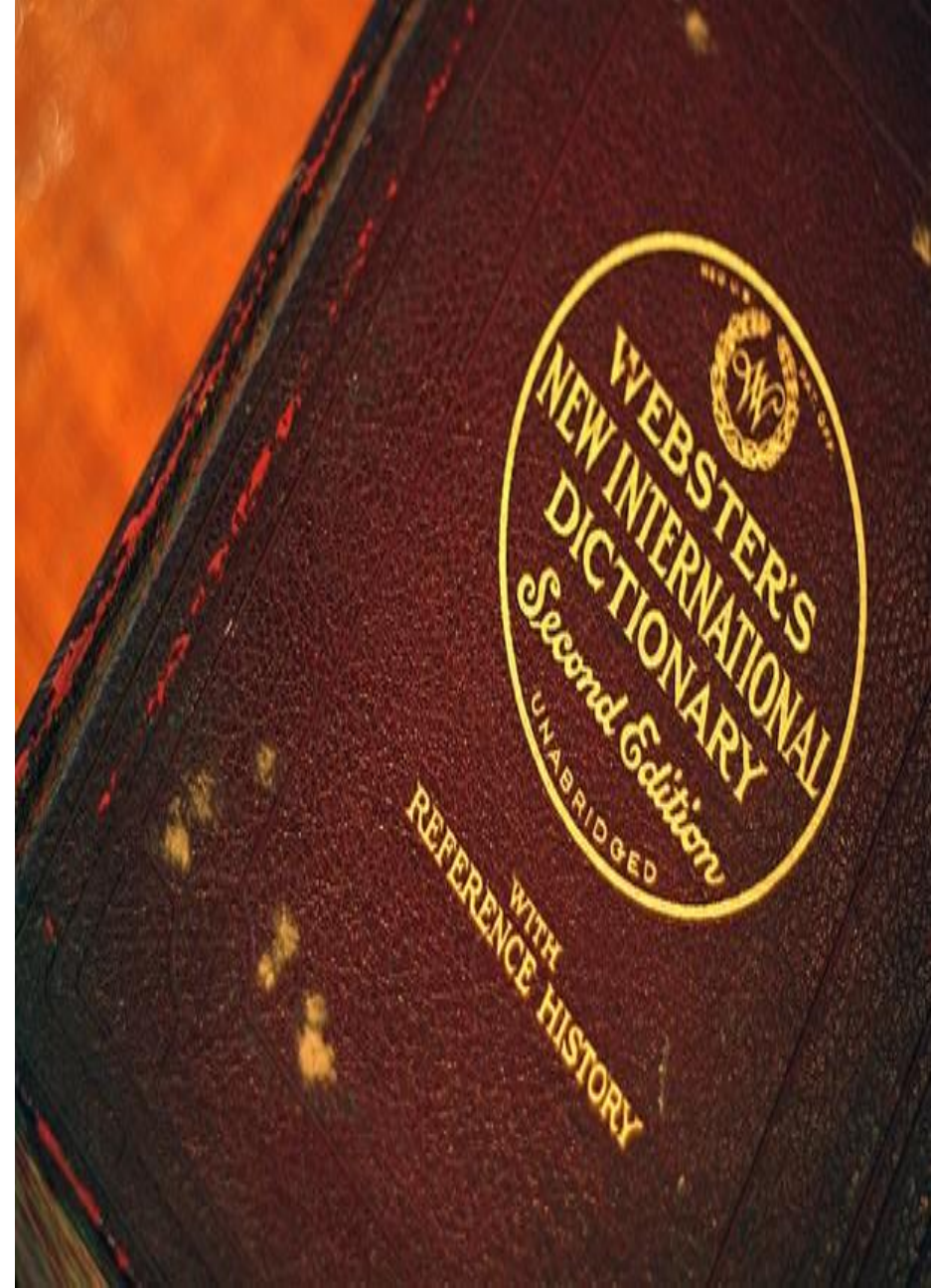
Page: 4KB, 64KB, 2MB, etc. of physically contiguous memory. Smallest unit handled by the OS.

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.

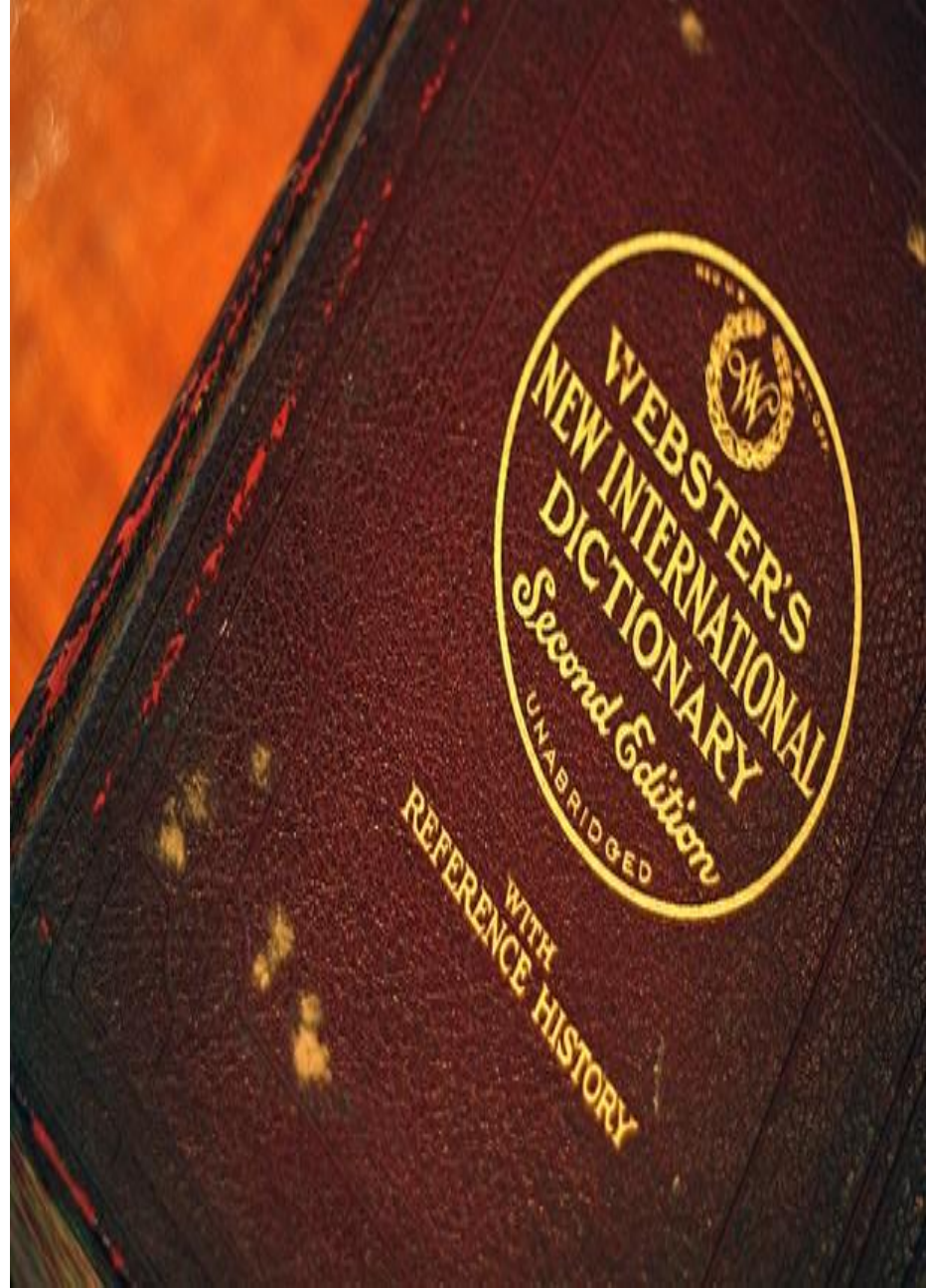
Page table: sparse tree containing virtual-to-physical address translations

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

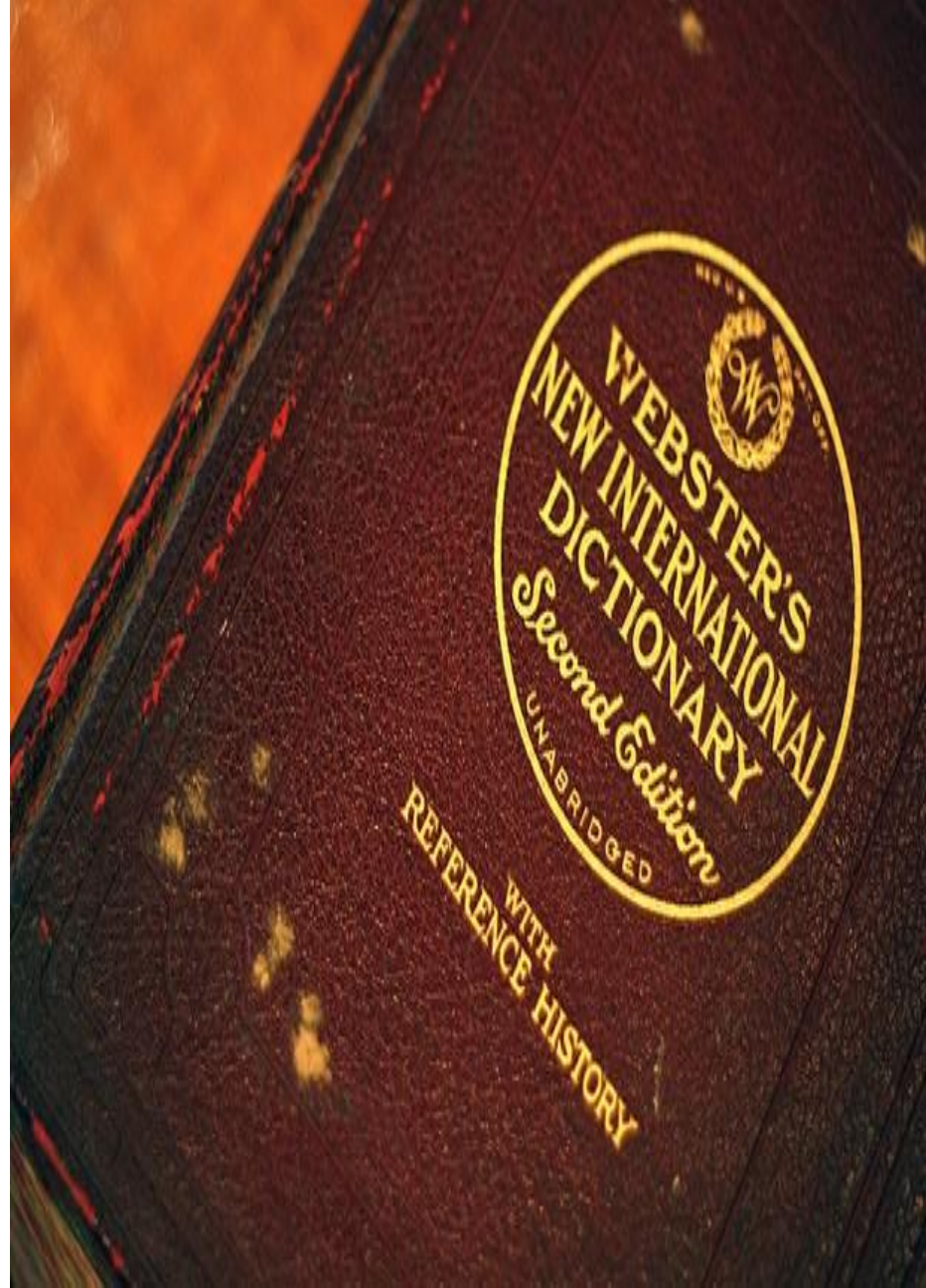
Page table entry: a single (page's worth of) virtual-to-physical translation

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

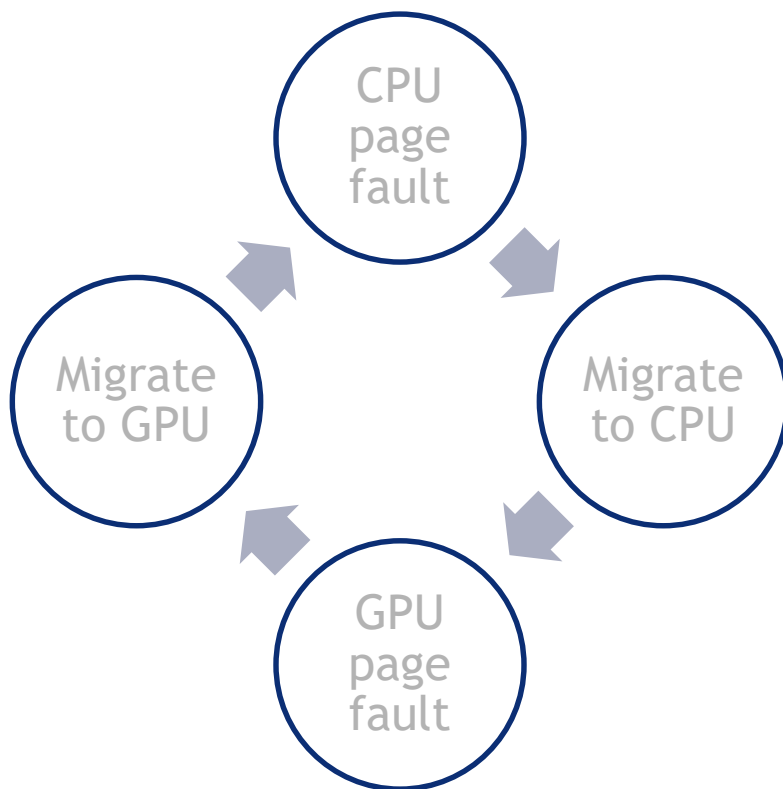
Unmap: remove a page table entry. Subsequent program accesses will cause page faults.

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



```
38: /* ... initialize and map() callback, for successfully migrated pages, this
39: * function updates the CPU page table to point to new pages, otherwise it
40: * restores the CPU page table to point to the original source pages.
41: * Function returns 0 after the above steps, even if no pages were migrated
42: * (The function only returns an error if any of the arguments are invalid.)
43: *
44: * Both src and dst array must be big enough for (end - start) >> PAGE_SHIFT
45: * unsigned long entries.
46: */
47: int migrate_vma(const struct migrate_vma_ops *ops,
48:                 struct vm_area_struct *vma,
49:                 unsigned long start,
50:                 unsigned long end,
51:                 unsigned long *src,
52:                 unsigned long *dst,
53:                 void *private)
54: {
55:     struct migrate_vma migrate;
56:
57:     /* Sanity check the arguments */
58:     start &= PAGE_MASK;
59:     end &= PAGE_MASK;
60:     if (!vma || !is_vm_hugetlb_page(vma) || (vma->vm_flags & VM_SPECIAL))
61:         return -EINVAL;
62:     if (start < vma->vm_start || start >= vma->vm_end)
63:         return -EINVAL;
64:     if (end <= vma->vm_start || end > vma->vm_end)
65:         return -EINVAL;
66:     if (!ops || !src || !dst || start >= end)
67:         return -EINVAL;
68:
69:     memset(src, 0, sizeof(*src) * ((end - start) >> PAGE_SHIFT));
70:     migrate.src = src;
71:     migrate.dst = dst;
72:     migrate.start = start;
73:     migrate.npages = 0;
74:     migrate.cpages = 0;
75:     migrate.end = end;
76:     migrate.vma = vma;
77:
78:     /* Collect, and try to unmap source pages */
79:     migrate_vma_collect(&migrate);
80:     if (!migrate.cpages)
81:         return 0;
82:
83:     /* Lock and isolate page */
84:     migrate_vma_prepare(&migrate);
85:     if (!migrate.cpages)
86:         return 0;
87:
88:     /* Unmap pages */
89:     migrate_vma_unmap(&migrate);
90:     if (!migrate.cpages)
91:         return 0;
92:
93:     /*
94:     * At this point pages are locked and unmapped, and thus they have
95:     * stable content and can safely be copied to destination memory that
96:     * is allocated by the callback.
97:     *
98:     * Note that migration can fail in migrate_vma_struct_page() for each
99:     * individual page.
100:    */
1001:    ops->alloc_and_copy(vma, src, dst, start, end, private);
1002:
1003:    /* 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

```
881 * Function unmap_vma_mmap_callback, for successfully migrated pages, this
882 * restores the CPU page table to point to new pages, otherwise it
883 *
884 * Function returns 0 after the above steps, even if no pages were migrated
885 * (The function only returns an error if any of the arguments are invalid.)
886 *
887 * Both src and dst array must be big enough for (end - start) >> PAGE_SHIFT
888 *
889 * unsigned long entries.
890 */
891 static int migrate_vma(struct mm_struct *mm, struct vm_area_struct *vma,
892                       unsigned long start,
893                       unsigned long end,
894                       unsigned long *src,
895                       unsigned long *dst,
896                       void *private)
897 {
898     struct migrate_vma migrate;
899     /* Sanity check the arguments */
900     start = PAGE_MASK;
901     end = PAGE_MASK;
902     if (!vma || !is_vm_hugetlb_page(vma) || (vma->vm_flags & VM_SPECIAL))
903         return -EINVAL;
904     if (start < vma->vm_start || start >= vma->vm_end)
905         return -EINVAL;
906     if (end <= vma->vm_start || end > vma->vm_end)
907         return -EINVAL;
908     if ((ops || !src || !dst) || start >= end)
909         return -EINVAL;
910     memset(&migrate, 0, sizeof(migrate));
911     migrate.src = src;
912     migrate.dst = dst;
913     migrate.start = start;
914     migrate.end = end;
915     migrate.ops = ops;
916     migrate.private = private;
917     migrate.vma = vma;
918     /* Collect, and try to unmap source pages */
919     migrate_vma_collect(&migrate);
920     if (!migrate.ops)
921         return 0;
922     /* Lock and isolate page */
923     migrate_vma_prepare(&migrate);
924     if (!migrate.ops)
925         return 0;
926     /* Unmap page */
927     migrate_vma_unmap(&migrate);
928     if (!migrate.ops)
929         return 0;
930     /*
931      * At this point pages are locked and unmapped, and then they have
932      * enough content and can safely be copied to destination memory that
933      * is allocated by the caller.
934      *
935      * Note that migration can fail in migrate_vma_collect() for each
936      * individual page.
937      *
938      * new folio and copy(vma, src, dst, start, end, private);
939      *
940      * This does the final migration of source page */
941     return 0;
942 }
```

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)

```
881 * Function updates the CPU page table to point to new pages, otherwise it
882 * restores the CPU page table to point to the original source pages.
883 *
884 * Function returns 0 after the above steps, even if no pages were migrated
885 * (The function only returns an error if any of the arguments are invalid.)
886 *
887 * Both src and dst array must be big enough for (end - start) >> PAGE_SHIFT
888 *
889 * Unmigrated long entries.
890 */
891
892 static int migrate_vma(struct vm_area_struct *vma,
893                      struct vm_area_struct *src,
894                      unsigned long start,
895                      unsigned long end,
896                      unsigned long *src,
897                      unsigned long *dst,
898                      void *private)
899 {
900     struct vm_area_struct *migrate;
901
902     /* Sanity check the arguments */
903     if (start >= PAGE_SIZE)
904         return -EINVAL;
905     if (vma != &vm_area_struct)
906         return -EINVAL;
907     if (start < vma->vm_start || start >= vma->vm_end)
908         return -EINVAL;
909     if (end <= vma->vm_start || end >= vma->vm_end)
910         return -EINVAL;
911     if ((src != NULL || dst != NULL) && start >= end)
912         return -EINVAL;
913
914     memset(src, 0, sizeof(*src) * (end - start) >> PAGE_SHIFT);
915     migrate->src = src;
916     migrate->dst = dst;
917     migrate->start = start;
918     migrate->pages = 0;
919     migrate->pages = 0;
920     migrate->end = end;
921     migrate->vm = vma;
922
923     /* Collect, and try to unmap source pages */
924     migrate_vma_collect(migrate);
925     if (!migrate->pages)
926         return 0;
927
928     /* Lock and isolate page */
929     migrate_vma_lock_page(migrate);
930     if (!migrate->pages)
931         return 0;
932
933     /* Unmap page */
934     migrate_vma_unmap(migrate);
935     if (!migrate->pages)
936         return 0;
937
938     /* At this point pages are locked and unmapped, and thus they have
939     * private content and can safely be copied to destination memory that
940     * is allocated by the caller.
941     *
942     * Note that migration can fail in migrate_vma_collect() for each
943     * individual page.
944     *
945     * Use malloc_and_copy(vma, src, dst, start, end, private)
946     * to do the real migration of source page */
947     return 0;
948 }
```

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
```

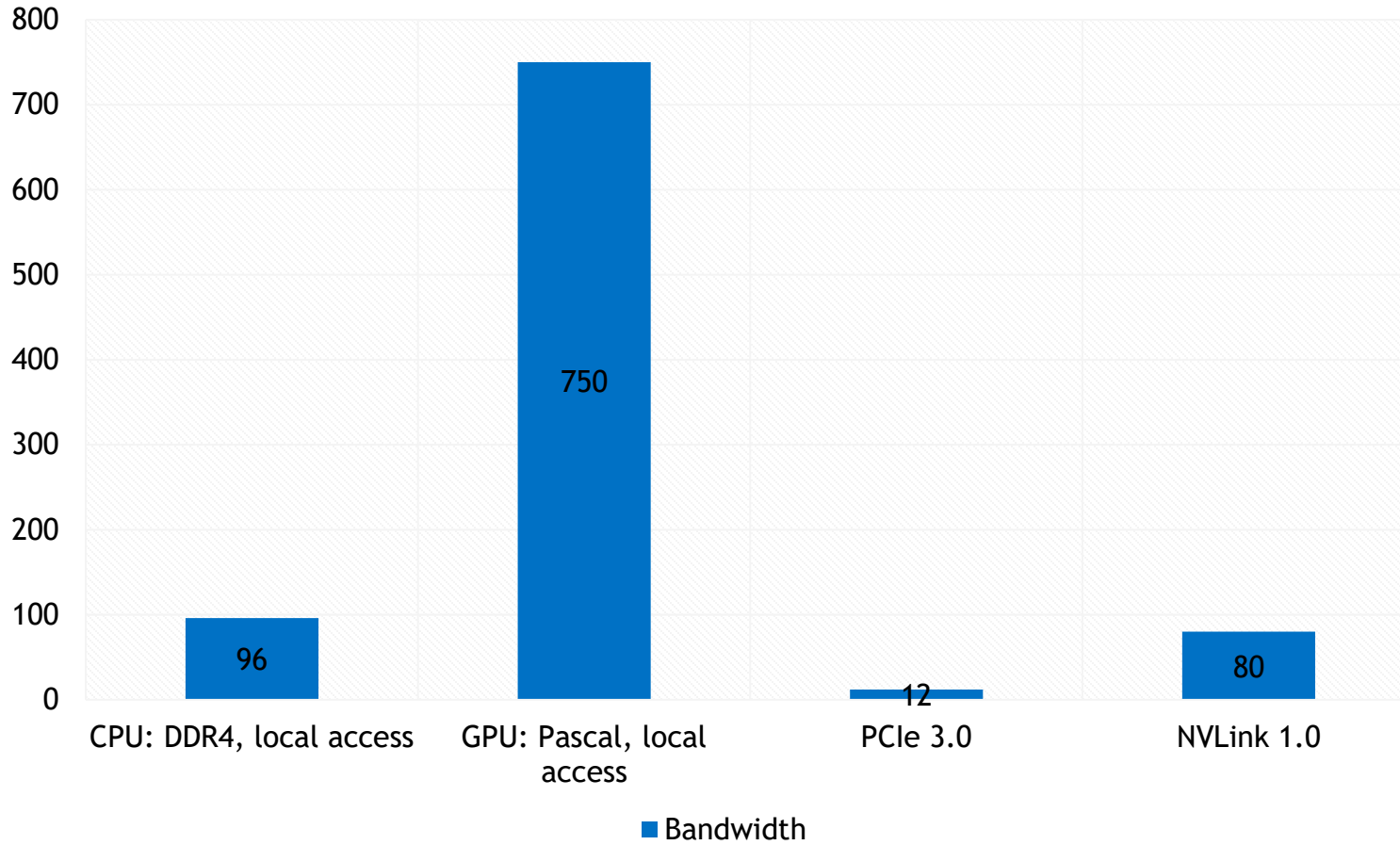
```
0.00% 11.524us 1 11.524us 11.524us 11.524us cudaConfigureCall
```

```
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



Tuning still works

`cudaMemPrefetchAsync`: this is the new `cudaMemcpy`

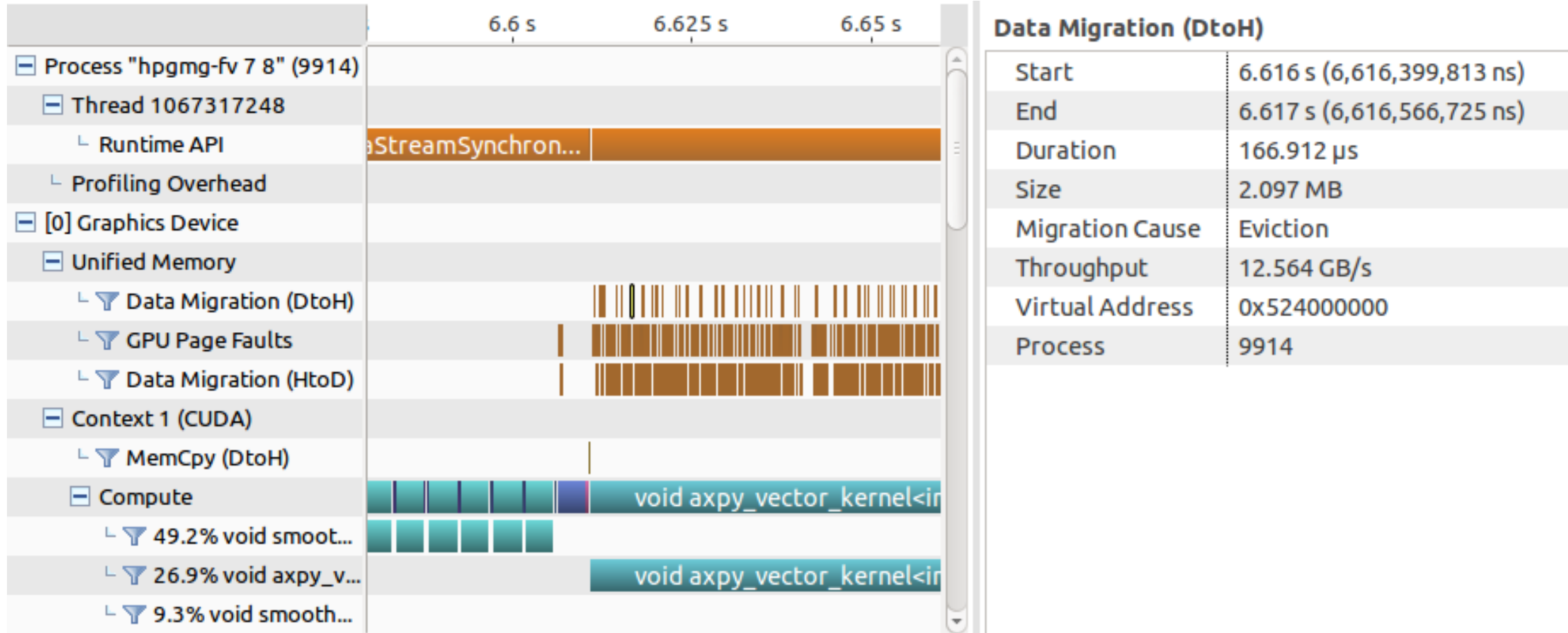
`cudaMemAdvise`

`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

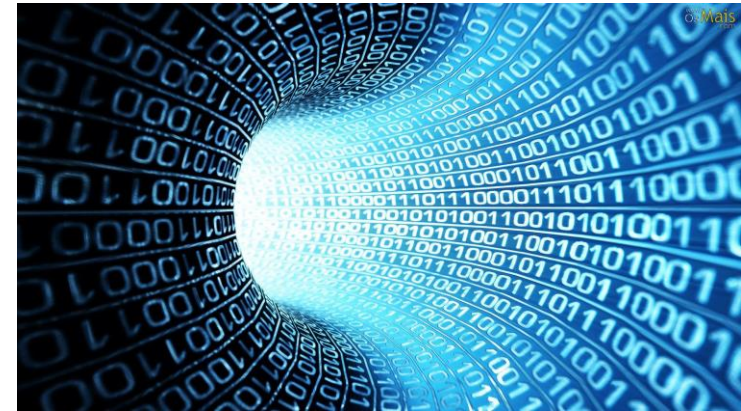
Conclusion: what you've learned

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

