

CS 623: Efficient Data Management in CUDA

Swarnendu Biswas

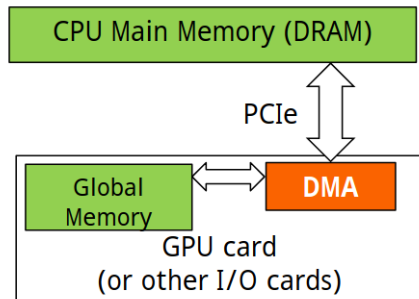
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2026-27-I



Data Transfer Between CPU and GPU

- Direct Memory Access (DMA) hardware transfers bytes between physical memory address spaces
 - ▶ Uses system interconnect, typically PCIe in today's systems
- `cudaMemcpy()` copies data as one or more DMA transfers
 - ▶ Source and destination pointers are virtual
- Virtual memory complicates data transfer
 - ▶ Pages in virtual address space are mapped into and out of the physical memory
 - ▶ The OS could accidentally page out the data that is being accessed by a DMA unit and page in another virtual page into the same physical location



Enabling DMA Transfer in CUDA with Pinned Memory

Definition

Pinned pages are virtual memory pages that are specially marked so that they cannot be paged out (also called Page-Locked Memory or Locked Pages)

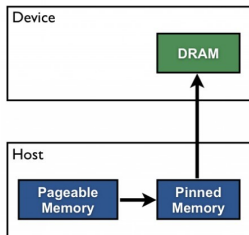
CPU memory that serves as the source or destination of a DMA transfer with `cudaMemcpy()` must be allocated as pinned memory

- The entire source and destination virtual address ranges are translated, and mapped to physical pages
- The CUDA driver checks if the entire memory page is pinned (i.e., unswappable)
- If the source or destination of `cudaMemcpy()` in the host is not pinned, it needs to be first copied to a pinned memory leading to extra overhead
 - ▶ `cudaMemcpy()` is faster if the host memory is allocated in pinned memory because no extra copy is needed

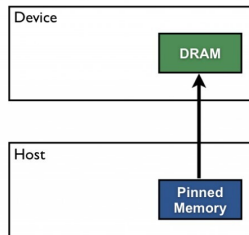
Pinned Memory with CUDA

- Allocate and free pinned memory with `cudaHostAlloc()` and `cudaFreeHost()` with the option `cudaHostAllocDefault`
- Pinned memory is a limited resource—over-subscription can have serious consequences

Pageable Data Transfer



Pinned Data Transfer



cudaMemCpy() with Pageable Memory



```
swarnendu@nilgiri:~/cuda-examples/src/vector$ nvprof ./vector-addition.out
==1941000== NVPROF is profiling process 1941000, command: ./vector-addition.out
Time taken (ms): h2d: 237.56 Kernel: 8.40618 d2h: 220.426
==1941000== Profiling application: ./vector-addition.out
==1941000== Profiling result:
          Type  Time(%)      Time   Calls    Avg      Min      Max  Name
GPU activities:  67.51%    474.62ms       2   237.31ms  237.16ms  237.46ms  [CUDA memcpy HtoD]
                31.30%    220.01ms       1   220.01ms  220.01ms  220.01ms  [CUDA memcpy DtoH]
                1.19%     8.3743ms       1    8.3743ms    8.3743ms    8.3743ms  vecAdd(float const *, float const *, float
*, unsigned int)
  API calls:     82.20%    695.45ms       3   231.82ms  220.42ms  237.64ms  cudaMemcpy
                16.13%    136.47ms       3    45.490ms  111.40us  136.24ms  cudaMalloc
                1.00%     8.4519ms       3    2.8173ms  2.8090us  8.3707ms  cudaEventSynchronize
                0.64%     5.3911ms       3    1.7970ms  758.27us  2.5189ms  cudaFree
                0.02%    142.30us      101    1.4080us   123ns   59.020us  cuDeviceGetAttribute
                0.00%    39.372us       6    6.5620us  1.7480us  18.458us  cudaEventRecord
                0.00%    32.013us       1    32.013us  32.013us  32.013us  cudaLaunchKernel
                0.00%    22.907us       1    22.907us  22.907us  22.907us  cuDeviceGetName
                0.00%    9.5030us       1    9.5030us  9.5030us  9.5030us  cuDeviceGetPCIBusId
                0.00%    9.2820us       2    4.6410us   865ns   8.4170us  cudaEventCreate
                0.00%    5.1920us       3    1.7300us   787ns   2.4430us  cudaEventElapsedTime
                0.00%    1.5260us       3      508ns   180ns   1.1610us  cuDeviceGetCount
                0.00%      601ns       1      601ns   601ns    601ns  cuDeviceTotalMem
                0.00%      598ns       2      299ns   125ns    473ns  cuDeviceGet
                0.00%      218ns       1      218ns   218ns    218ns  cuDeviceGetUuid

swarnendu@nilgiri:~/cuda-examples/src/vector$ |
```

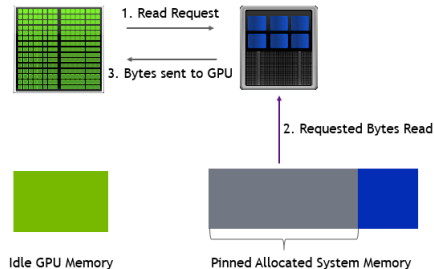
cudaMemCpy() with Pinned Memory



```
swarnendu@nilgiri:~/cuda-examples/src/vector$ nvprof ./vector-addition-hostalloc-default.out
==1941051== NVPROF is profiling process 1941051, command: ./vector-addition-hostalloc-default.out
Time taken (ms): h2d: 87.2645 Kernel: 8.52237 d2h: 81.3356
==1941051== Profiling application: ./vector-addition-hostalloc-default.out
==1941051== Profiling result:
          Type  Time(%)      Time   Calls    Avg      Min      Max  Name
GPU activities:  66.03%   174.50ms      2  87.249ms  87.246ms  87.252ms  [CUDA memcpy HtoD]
                30.77%   81.317ms      1  81.317ms  81.317ms  81.317ms  [CUDA memcpy DtoH]
                3.20%    8.4589ms      1  8.4589ms  8.4589ms  8.4589ms  vecAdd(float const *, float const *, float
*, unsigned int)
API calls:      68.93%   1.46052s      3  486.84ms  437.10ms  585.73ms  cudaHostAlloc
                18.30%   387.85ms      3  129.28ms  125.50ms  136.48ms  cudaFreeHost
                12.08%   255.92ms      3   85.306ms  81.333ms  87.328ms  cudaMemcpy
                 0.40%    8.4608ms      3   2.8203ms  3.1570us  8.4543ms  cudaEventSynchronize
                 0.25%    5.2468ms      3   1.7489ms  627.15us  2.5135ms  cudaFree
                 0.03%    555.08us      3   185.03us  106.57us  322.72us  cudaMalloc
                 0.01%   144.27us     101  1.4280us    166ns  57.635us  cuDeviceGetAttribute
                 0.00%    57.403us      1   57.403us  57.403us  57.403us  cudaLaunchKernel
                 0.00%    38.628us      6    6.4380us  1.7470us  26.804us  cudaEventRecord
                 0.00%    15.428us      2    7.7140us  1.1860us  14.242us  cudaEventCreate
                 0.00%    13.944us      1   13.944us  13.944us  13.944us  cuDeviceGetName
                 0.00%    10.629us      1   10.629us  10.629us  10.629us  cuDeviceGetPCIBusId
                 0.00%    4.1260us      3    1.3750us    769ns  2.4630us  cudaEventElapsedTime
                 0.00%    3.1800us      3    1.0600us    241ns  2.3700us  cuDeviceGetCount
                 0.00%      945ns      1      945ns    945ns    945ns  cuDeviceTotalMem
                 0.00%      812ns      2      406ns    156ns    656ns  cuDeviceGet
```

Zero-Copy Memory with CUDA

- Zero copy memory is pinned memory that is mapped into the device address space
 - ▶ Both host and device have fine-grained direct access
 - + Can leverage host memory when there is insufficient device memory
 - + Avoids explicit data transfers between host and device
 - ▶ Should be used for occasionally when the data is read-only or the GPU memory is really scarce



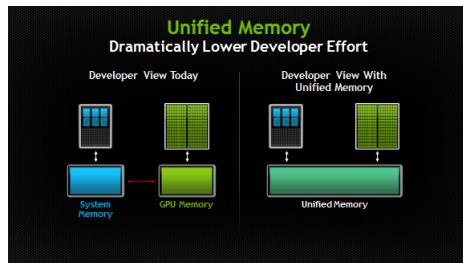
cudaMemCpy() with Zero-Copy Memory



```
swarnendu@nilgiri:~/cuda-examples/src/vector$ nvprof ./vector-addition-hostalloc-mapped.out
==1948543== NVPR0F is profiling process 1948543, command: ./vector-addition-hostalloc-mapped.out
Invocation i: 0 Time taken (ms): kernel: 378.814
Invocation i: 1 Time taken (ms): kernel: 373.741
Invocation i: 2 Time taken (ms): kernel: 373.747
Invocation i: 3 Time taken (ms): kernel: 373.714
Invocation i: 4 Time taken (ms): kernel: 373.742
Invocation i: 5 Time taken (ms): kernel: 373.717
Invocation i: 6 Time taken (ms): kernel: 376.482
Invocation i: 7 Time taken (ms): kernel: 373.599
Invocation i: 8 Time taken (ms): kernel: 376.099
Invocation i: 9 Time taken (ms): kernel: 376.226
==1948543== Profiling application: ./vector-addition-hostalloc-mapped.out
==1948543== Profiling result:
   Type  Time(%)   Time      Calls      Avg      Min      Max  Name
GPU activities: 100.00% 3.74542s      10  374.54ms  373.60ms  376.60ms  vecAdd(float const *, float const *, float
*, unsigned int)
  API calls:    57.77% 3.76791s      10  376.79ms  375.56ms  378.79ms  cudaEventSynchronize
                30.61% 1.99639s       3  665.46ms  583.01ms  742.25ms  cudaHostAlloc
                6.53% 425.82ms       3  141.94ms  141.12ms  143.34ms  cudaFreeHost
                5.07% 330.84ms       1  330.84ms  330.84ms  330.84ms  cudaSetDeviceFlags
                0.01% 447.18us      10  44.718us  27.583us  94.893us  cudaLaunchKernel
                0.00% 207.81us      20  10.390us  2.7920us  30.620us  cudaEventRecord
                0.00% 139.87us     101  1.3840us   140ns   55.603us  cuDeviceGetAttribute
                0.00% 59.182us      10  5.9180us  3.9300us  14.372us  cudaEventElapsedTime
                0.00% 41.062us       2  20.531us   940ns   40.122us  cudaEventCreate
```

Managed Memory

- Unified Virtual Memory (UVM) provides a single memory space accessible by all GPUs and CPUs in the system
- Use `cudaMallocManaged()` to allocate data in unified memory or use `__managed__` keyword in the global scope
 - ▶ Returns a pointer that can be accessed from both host and device code



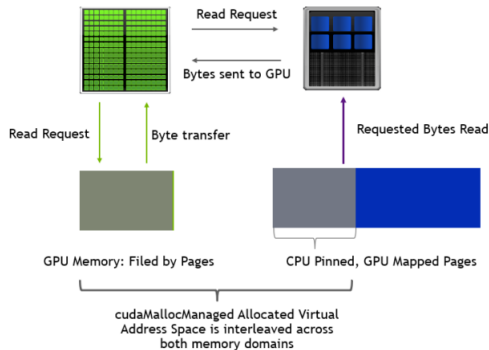
Unified Memory in CUDA 6

An Even Easier Introduction to CUDA

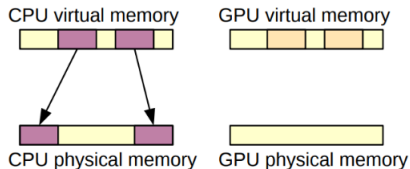
Unified Memory for CUDA Beginners

Managed Memory

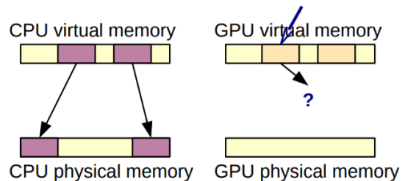
- UVM features have been evolving starting from CUDA 6+
 - ▶ 6.x: use a single pointer in both CPU functions and GPU kernels
 - ▶ 8.x: added 49-bit virtual addressing and on-demand page migration
- CUDA runtime **automatically migrates data** allocated in Unified Memory between host and device, different from UVA



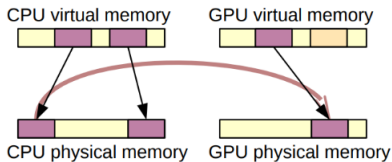
On-demand Paging



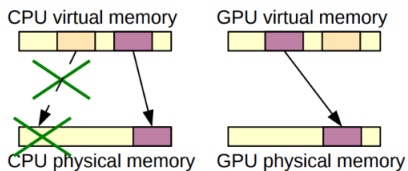
1 Data allocated in CPU memory



2 GPU touches unallocated page, triggers page fault



3 Page fault handler allocates page in GPU mem, copies contents



4 If GPU modifies page contents, invalidate CPU copy. Next CPU access will cause data to be copied back from GPU mem.

cudaMemcpy() with Managed Memory



```
swarnendu@nilgiri:~/cuda-examples/src/vector$ nvprof ./vector-addition-managed.out
==1948976== NVPROF is profiling process 1948976, command: ./vector-addition-managed.out
Invocation: 0 Time taken (ms): kernel: 1219.01
==1948976== Profiling application: ./vector-addition-managed.out
==1948976== Profiling result:
   Type  Time(%)   Time      Calls      Avg      Min      Max  Name
GPU activities: 100.00% 1.21679s      1 1.21679s 1.21679s 1.21679s vecAdd(float const *, float const *, float*, int)
  API calls:    73.02% 1.21899s      1 1.21899s 1.21899s 1.21899s cudaEventSynchronize
              17.84% 297.79ms      3 99.265ms 23.511us 297.70ms cudaMallocManaged
              9.12% 152.21ms      3 50.737ms 50.422ms 50.941ms cudaFree
              0.01% 142.95us     101 1.4150us      145ns 55.431us cuDeviceGetAttribute
              0.00% 75.898us      1 75.898us 75.898us 75.898us cudaLaunchKernel
              0.00% 59.680us      2 29.840us 5.5550us 54.125us cudaEventCreate
              0.00% 46.808us      2 23.404us 3.9890us 42.819us cudaEventRecord
              0.00% 30.918us      1 30.918us 30.918us 30.918us cuDeviceGetName
              0.00% 9.5370us      1 9.5370us 9.5370us 9.5370us cuDeviceTotalMem
              0.00% 9.3320us      1 9.3320us 9.3320us 9.3320us cuDeviceGetPCIBusId
              0.00% 7.1650us      1 7.1650us 7.1650us 7.1650us cudaEventElapsedTime
              0.00% 2.6680us      3 889ns      183ns 1.3890us cuDeviceGetCount
              0.00% 883ns      2 441ns      367ns 516ns cuDeviceGet
              0.00% 350ns      1 350ns      350ns 350ns cuDeviceGetUuid

==1948976== Unified Memory profiling result:
Device "Quadro RTX 5000 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
   36250  54.919KB  4.0000KB  0.9961MB  1.8986176B  224.7973ms  Host To Device
   18408  170.67KB  4.0000KB  0.9961MB  2.9960946B  261.0419ms  Device To Host
    4693      -      -      -      -      539.8935ms  Gpu page fault groups
Total CPU Page faults: 15349
```

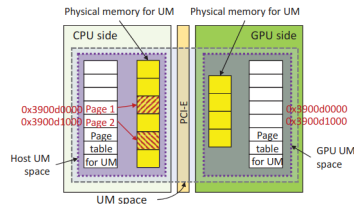
cudaMemcpy() with Managed Memory



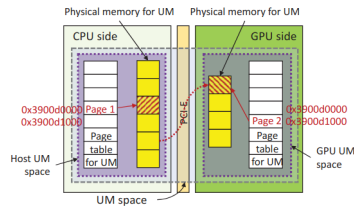
```
~/cuda-examples/src/vector$ nvprof --print-gpu-trace ./vector-addition-managed.out 2>&1 | tee output.txt
...
==1949017== Profiling result:
   Start   Duration      Grid Size    Block Size    Regs*    SSMem*    DSMem*          Device    Context
Stream      Unified Memory Virtual Address  Name
499.24ms      -          PC 0xe0f0cdc8  0x7fd774000000 [Unified Memory CPU page faults]
-            -          PC 0xe0f0cdc8  0x7fd774010000 [Unified Memory CPU page faults]
499.73ms      -          PC 0xe0f0cdc8  0x7fd774020000 [Unified Memory CPU page faults]
-            -          PC 0xe0f0cdc8  0x7fd774030000 [Unified Memory CPU page faults]
499.76ms      -          PC 0xe0f0cdc8  0x7fd774040000 [Unified Memory CPU page faults]
-            -          PC 0xe0f0cdc8  0x7fd774050000 [Unified Memory CPU page faults]
...
1.59851s      -          PC 0xe0f0cdf0  0x7fd76fe80000 [Unified Memory CPU page faults]
-            -          PC 0xe0f0cdf0  0x7fd76ff00000 [Unified Memory CPU page faults]
1.59862s      -          PC 0xe0f0cdf0  0x7fd76ff00000 [Unified Memory CPU page faults]
-            -          PC 0xe0f0cdf0  0x7fd76ff00000 [Unified Memory CPU page faults]
1.60122s  1.22684s      (1048576 1 1)  (256 1 1)      16      0B      0B  Quadro RTX 5000
1      7      -      -      vecAdd(float const *, float const *, float*, int) [117]
1.60123s  334.53us      -      -      -      -      -  Quadro RTX 5000
-            -          8  0x7fd730000000 [Unified Memory GPU page faults]
1.60142s  10.016us      -      -      -      -      -  Quadro RTX 5000
-            -      96.000000KB  0x7fd730000000 [Unified MemoryMemcpy HtoD]
1.60143s  4.6720us      -      -      -      -      -  Quadro RTX 5000
-            -      32.000000KB  0x7fd730018000 [Unified MemoryMemcpy HtoD]
...
```

Unified Virtual Memory (UVM)

- On pre-Pascal GPUs, `cudaMallocManaged()` allocated managed space on the device that was active at the time of the call
- Pascal onward, `cudaMallocManaged()` does not immediately allocate physical memory
 - ▶ Pages and page table entries are not created until the first access by the GPU or the CPU
- Hardware supports page faulting and migration
 - ▶ The GPU stalls the accessor threads when they access absent pages
 - ▶ The Page Migration Engine migrates pages to the device before resuming the threads



(a) After the host has accessed page 1 and page 2.



(b) After the GPU has accessed page 2.

Pre-Pascal Behavior of cudaMallocManaged()

```
__global__ void add(int n, float* x, float* y) {  
    int index = blockIdx.x * blockDim.x + threadIdx.x;  
    int stride = blockDim.x * gridDim.x;  
    for (int i = index; i < n; i += stride)  
        y[i] = x[i] + y[i];  
}  
  
int main(void) {  
    int N = (1 << 20);  
    float *x, *y;  
    cudaMallocManaged(&x, N * sizeof(float));  
    cudaMallocManaged(&y, N * sizeof(float));  
    for (int i = 0; i < N; i++)  
        x[i] = 1.0f; y[i] = 2.0f;  
    int blockSize = 256;  
    int numBlocks = (N + blockSize - 1) / blockSize;  
    add<<<numBlocks, blockSize>>>(N, x, y);  
    cudaDeviceSynchronize();  
    ...  
}
```

x and y are allocated on GPU memory,
driver sets up page table entries

Page fault on CPU, x and y
are copied to CPU memory

Moves data back
to CPU memory

Lacked support for page faults, so all data
is copied to GPU before kernel launch

Pre-Pascal Behavior of cudaMallocManaged()

```
__global__ void add(int n, float* x, float* y) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}

int main(void) {
    int N = (1 << 20);
    float *x, *y;
    cudaMallocManaged(&x, N * sizeof(float));
    cudaMallocManaged(&y, N * sizeof(float));
    for (int i = 0; i < N; i++)
        x[i] = 1.0f;
    int blockSize = 1024;
    int numBlocks = (N + blockSize - 1) / blockSize;
    add<<<numBlocks, blockSize>>>(N, x, y);
    cudaDeviceSynchronize();
    ...
}
```

- No concurrent access
- No on-demand migration to GPU
- No oversubscription

Moves data back to CPU memory

Lacked support for page faults, so all data is copied to GPU before kernel launch

Behavior of `cudaMallocManaged()` for Pascal+

```
__global__ void add(int n, float* x, float* y) {  
    int index = blockIdx.x * blockDim.x + threadIdx.x;  
    int stride = blockDim.x * gridDim.x;  
    for (int i = index; i < n; i += stride)  
        y[i] = x[i] + y[i];  
}
```

Page migration cost is now part of the kernel execution time

```
int main(void) {  
    int N = (1 << 20);  
    float *x, *y;  
    cudaMallocManaged(&x, N * sizeof(float));  
    cudaMallocManaged(&y, N * sizeof(float));  
    for (int i = 0; i < N; i++)  
        x[i] = 1.0f; y[i] = 2.0f;  
    int blockSize = 256;  
    int numBlocks = (N + blockSize - 1) / blockSize;  
    add<<<numBlocks, blockSize>>>(N, x, y);  
    cudaDeviceSynchronize();  
    ...  
}
```

Physical memory for x and y are not immediately allocated, allocated only on first access or prefetch

Page fault on CPU, x and y are allocated on CPU memory

Supports page faults, so no data migration overhead before kernel launch

Profiling with nvprof on Turing GPU



```
swarnendu@nilgiri:~/cuda-examples/src/uvm$ nvprof ./add-grid.out
==2284609== NVPROF is profiling process 2284609, command: ./add-grid.out
==2284609== Profiling application: ./add-grid.out
==2284609== Profiling result:
   Type  Time(%)   Time      Calls      Avg      Min      Max  Name
GPU activities: 100.00%  3.2329ms    1  3.2329ms  3.2329ms  3.2329ms  add(float const *, float const *, float*, unsigned int)
  API calls:  97.02%  138.15ms    3  46.049ms  22.308us  138.02ms  cudaMallocManaged
              2.28%  3.2434ms    1  3.2434ms  3.2434ms  3.2434ms  cudaDeviceSynchronize
              0.53%  755.20us    3  251.73us  216.47us  315.78us  cudaFree
              0.09%  133.92us   101  1.3250us   133ns   52.233us  cuDeviceGetAttribute
              0.05%  71.813us    1  71.813us  71.813us  71.813us  cudaLaunchKernel
              0.02%  24.813us    1  24.813us  24.813us  24.813us  cuDeviceGetName
              0.01%  9.2100us    1  9.2100us  9.2100us  9.2100us  cuDeviceGetPCIBusId
              0.00%  2.4970us    3    832ns   192ns   2.0830us  cuDeviceGetCount
              0.00%   933ns     2    466ns   145ns    788ns  cuDeviceGet
              0.00%   584ns     1    584ns   584ns   584ns  cuDeviceTotalMem
              0.00%   282ns     1    282ns   282ns   282ns  cuDeviceGetUuid

==2284609== Unified Memory profiling result:
Device "Quadro RTX 5000 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    192   64.000KB  4.0000KB  964.00KB  12.00000MB  1.347288ms  Host To Device
    72   170.67KB  4.0000KB  0.9961MB  12.00000MB  1.024603ms  Device To Host
    21      -      -      -      -      3.202094ms  Gpu page fault groups
Total CPU Page faults: 72
swarnendu@nilgiri:~/cuda-examples/src/uvm$
```

Page Migration: High-Level Mechanism

- Assume a Pascal+ GPU accesses a page not present in the GPU memory
- Address translation for the faulting page generates a fault message and locks the TLBs for the corresponding SM
 - ▶ On some architectures, two SMs share a TLB, so both are locked
 - ▶ Locking implies outstanding translations can proceed but new translations will be stalled until all faults are resolved
- GPU can generate many faults concurrently for the same page
- UVM driver processes the faults, removes duplicates, updates page table mappings, and transfers the data
- Fault handling adds significant overhead to streaming performance of UVM



(i) Initialize data on the device

```
__global__ void init(int n, float *x, float *y) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    int stride = blockDim.x * gridDim.x;  
    for (int i = index; i < n; i += stride) {  
        x[i] = 1.0f; y[i] = 2.0f;  
    }  
}
```

```
swarnendu@nilgiri:~/cuda-examples/src/uvvm$ nvprof ./add-grid-init.out  
==2284736== NVPROF is profiling process 2284736, command: ./add-grid-init.out  
==2284736== Profiling application: ./add-grid-init.out  
==2284736== Profiling result:  


|                 | Type | Time(%) | Time     | Calls | Avg      | Min      | Max      | Name                                                    |
|-----------------|------|---------|----------|-------|----------|----------|----------|---------------------------------------------------------|
| GPU activities: |      | 98.70%  | 2.8188ms | 1     | 2.8188ms | 2.8188ms | 2.8188ms | init(float*, float*, float*, unsigned int)              |
|                 |      | 1.30%   | 37.217us | 1     | 37.217us | 37.217us | 37.217us | add(float const *, float const *, float*, unsigned int) |
| API calls:      |      | 97.29%  | 137.84ms | 3     | 45.948ms | 14.291us | 137.74ms | cudaMallocManaged                                       |
|                 |      | 2.01%   | 2.8438ms | 1     | 2.8438ms | 2.8438ms | 2.8438ms | cudaDeviceSynchronize                                   |
|                 |      | 0.53%   | 749.47us | 3     | 249.82us | 214.63us | 316.77us | cudaFree                                                |
|                 |      |         |          |       |          |          |          |                                                         |

  
==2284736== Unified Memory profiling result:  
Device "Quadro RTX 5000 (0)"  


|  | Count | Avg Size | Min Size | Max Size | Total Size | Total Time | Name                  |
|--|-------|----------|----------|----------|------------|------------|-----------------------|
|  | 121   | 101.55KB | 4.0000KB | 0.9922MB | 12.00000MB | 1.259259ms | Host To Device        |
|  | 72    | 170.67KB | 4.0000KB | 0.9961MB | 12.00000MB | 1.025338ms | Device To Host        |
|  | 20    | -        | -        | -        | -          | 2.793011ms | Gpu page fault groups |

  
Total CPU Page faults: 72  
swarnendu@nilgiri:~/cuda-examples/src/uvvm$
```



- (ii) Use warm-up iterations or take the average of kernel multiple runs
- (iii) Prefetch data on the GPU with `cudaMemPrefetchAsync()`

```
int device = -1;
cudaGetDevice(&device);
cudaMemPrefetchAsync(X, N*sizeof(float), device, NULL);
cudaMemPrefetchAsync(Y, N*sizeof(float), device, NULL);
cudaMemPrefetchAsync(Z, N*sizeof(float), device, NULL);
```

```
swarnendu@nilgiri:~/cuda-examples/src/uvvm$ nvprof ./add-grid-prefetch.out
==2285986== NVPROF is profiling process 2285986, command: ./add-grid-prefetch.out
==2285986== Profiling application: ./add-grid-prefetch.out
==2285986== Profiling result:
   Type  Time(%)   Time    Calls   Avg      Min      Max   Name
GPU activities: 100.00% 29.855us    1 29.855us 29.855us 29.855us add(float const *, float const *, float*, unsigned int)
  API calls:   98.05% 137.99ms    3 45.996ms 35.369us 137.84ms cudaMallocManaged
              0.95%  1.3409ms    1  1.3409ms 1.3409ms 1.3409ms cudaDeviceSynchronize
              0.53%   743.23us    3  247.74us 214.38us 313.59us cudaFree
              0.30%  426.23us    3  142.08us  5.0270us 249.26us cudaMemPrefetchAsync

==2285986== Unified Memory profiling result:
Device "Quadro RTX 5000 (0)"
  Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    6    2.0000MB 2.0000MB 2.0000MB 12.00000MB  1.042556ms Host To Device
   72   170.67KB 4.0000KB 0.9961MB 12.00000MB  1.025625ms Device To Host
Total CPU Page faults: 72
swarnendu@nilgiri:~/cuda-examples/src/uvvm$
```



- Advise runtime on expected memory access behaviors with `cudaMemAdvise(ptr, count, hint, device)`
- Possible hints:
 - `cudaMemAdviseSetReadMostly` Specify read duplication
 - `cudaMemAdviseSetPreferredLocation` Suggest best location
 - `cudaMemAdviseSetAccessedBy` Suggest mapping
- Hints do not trigger data movement by themselves

Explicit Memory Hints in UVM

`cudaMemAdviseSetReadMostly`

- Data will usually be read-only
- UVM will make a “local” copy of the data for each processor that touches it
- A write will invalidate all copies except the one written

`cudaMemAdviseSetPreferredLocation`

- Suggests which compute device is the best location for the data
- If possible, data mappings will be provided when other processors touch it
- If mapping is not possible, data is migrated

`cudaMemAdviseSetAccessedBy`

- Does not cause movement or affect location of data
- Indicated compute device receives a mapping to the data
- If the data is migrated, mapping is updated

Driver-based Prefetching

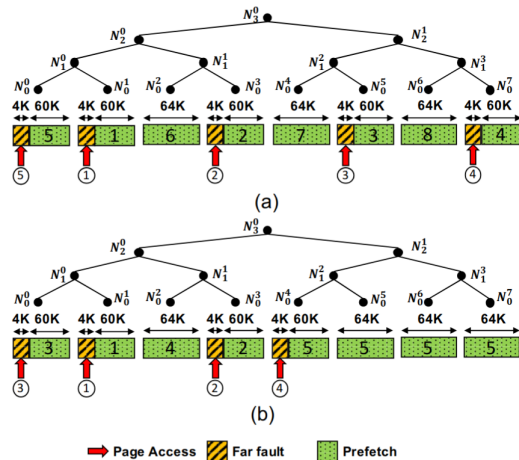
- Providing user hints can be complicated and error-prone
- CUDA driver can implement different prefetch policies: (i) random, (ii) sequential, or (iii) locality-aware
 - ▶ For example, “random” prefetches a random 4 KB page from the 2 MB large page boundary along with the 4 KB faulting page
- NVIDIA implements a locality-aware tree-based neighborhood prefetcher
 - ▶ Migrate multiples of 64 KB blocks contiguous in the virtual address space grouped in a single transfer
 - ▶ All pages being prefetched are local to the current faulty page and are within 2 MB large page boundary

D. Ganguly et al. Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. ISCA'19.

Mao Lin et al. Forest: Access-aware GPU UVM Management. ISCA'25.

Tree-based Neighborhood Prefetcher

- Allocation with `cudaMallocManaged()` is logically divided into 2 MB large pages
- The 2 MB pages are further divided into logical 64 KB basic blocks to create a full binary tree
- If the user-specified allocation request is not a multiple of 2 MB, the remainder allocation is rounded up to the next $2i * 64 \text{ KB}$
 - ▶ If the requests are for 4 MB and 192 KB, then two 2 MB trees and 1 256 KB trees are created



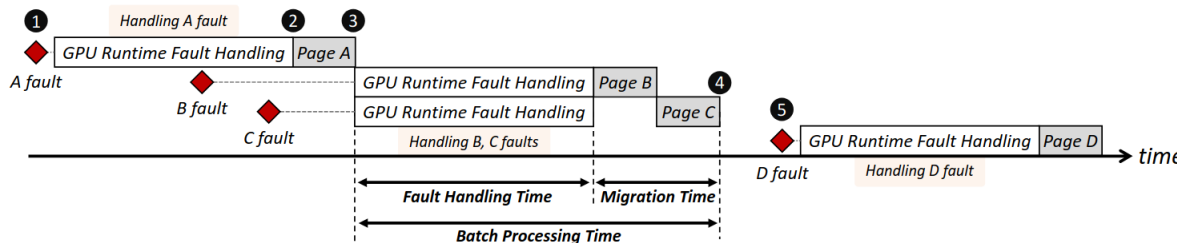
Note about UVM

- UVM provides a coherent view of a single virtual address space between CPUs and GPUs with automatic data migration via demand paging
 - + Allows GPUs to access a page that resides in the CPU memory as if it were in the GPU memory
 - + Allows applications to run without worrying about the device memory capacity
- UVM aims to improve programmer productivity
 - + Code is less verbose, makes it easy to work with nested data structures (think of C++ classes with dynamically allocated attributes)
- UVM kernels may have poorer performance
 - Negative is the substantial cost of address translation overhead and demand paging

```
struct dataElem {  
    int prop1;  
    int prop2;  
    char *name;  
}
```

Handling GPU Page Faults

- When a GPU tries to access a physical memory page that is not currently resident in device memory, a page fault is raised and GPU runtime migrates the requested page to the GPU memory
- Page fault handling is expensive because it requires long latency communications between the CPU and GPU over the PCIe bus
 - ▶ The GPU runtime processes a group of page faults together to amortize overhead

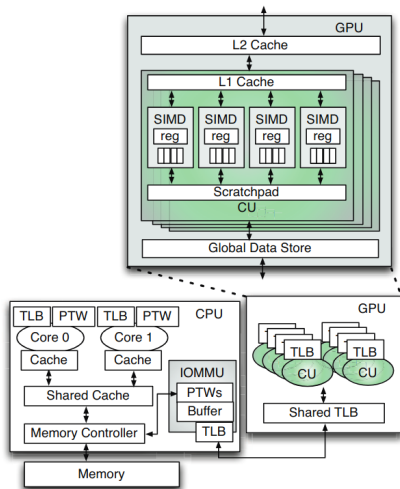


Memory Access Divergence

- An SIMD memory instruction cannot complete until data for all threads are available
 - ▶ Problematic for irregular applications with little scope for coalescing
 - ▶ Execution of one instruction requires multiple cache accesses when accesses fall on distinct cache lines and multiple virtual-to-physical address translations when accesses fall on distinct pages
- Negative impact from divergence can impact address translation more than cache access
 - ▶ Irregular memory accesses can lead from 1 to warp size (32/64) address translation requests, most will miss in the TLB
 - ▶ A page table walk on a TLB miss can take up to four memory accesses, for a total of 128–256 memory accesses per instruction
 - ▶ GPUs employ physical caches which makes address translation the bottleneck

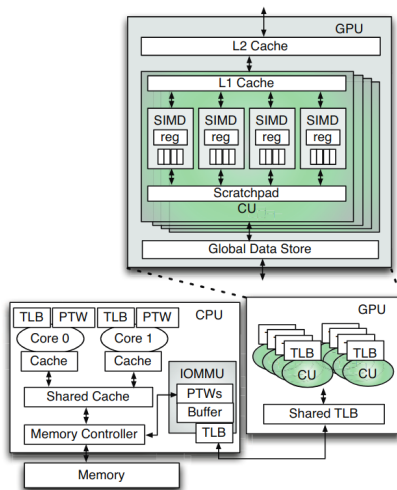
Address Translation Request

- IOMMU is a hardware component on the CPU that services address translation requests for accesses to the DRAM by any accelerator (e.g., GPU)
 - ▶ IOMMU supports multiple page table walkers (e.g., 8–16) to concurrently service multiple page table walk requests (TLB misses)
 - ▶ IOMMU employs small page walk caches for the first three levels of the page tables
- GPU multiprocessors (compute units) share a private L1 TLB across SIMD units
- The GPU's L1 TLBs are backed by a larger L2 TLB that is shared across all the CUs in the GPU



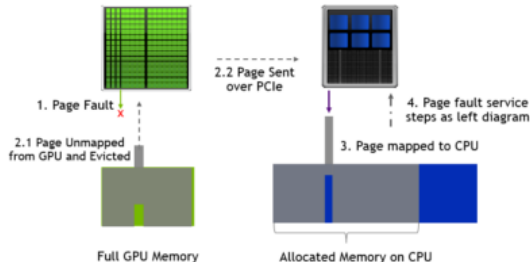
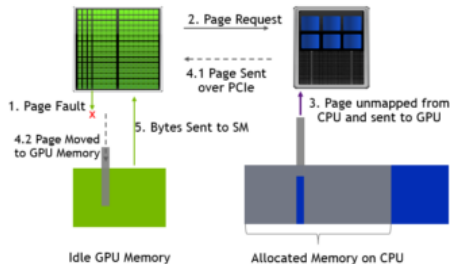
Address Translation Request

- (i) An address translation request is generated by a SIMD load/store instruction
- (ii) A hardware coalescer merges multiple requests to the same page generated by the same SIMD instruction
- (iii) The coalesced translation request looks up the GPU's L1 TLB and then the GPU's shared L2
- (iv) On a miss in the GPU's L2 TLB, the request is sent to the IOMMU
- (v) At the IOMMU, the request looks up the IOMMU's TLBs
- (vi) On a miss, the request queues up as a page walk request in the IOMMU buffer
- (vii) When an IOMMU's page table walker becomes free, it selects a pending request from the IOMMU buffer in some order
- (viii) The page table walker first performs a PWC lookup and then completes the walk of the page table, generating one to four memory accesses
- (ix) On finishing a walk, the desired translation is returned to the TLBs and ultimately to the GPU SIMD unit that requested it



Memory Oversubscription

- UVM support allows GPUs to oversubscribe memory
- With oversubscription, a memory page is first evicted from GPU memory to system memory, followed by transfer of requested memory from CPU to GPU



Prefetching with Memory Oversubscription

- Aggressive prefetching under memory constraint can be counter-productive, may cause displacement of heavily-accessed pages
- CUDA drivers implement LRU 4 KB page replacement policy
- Penalty of faults are greater under oversubscription
 - ▶ Threads need to be stalled for writing back pages along with the latency to migrate new pages
- An option is to disable prefetching on oversubscription

References

-  NVIDIA Corporation. CUDA C++ Programming Guide, Release 12.9.2.
-  NVIDIA Corporation. CUDA C++ Best Practices Guide, Release 12.9.