Review

- Last Class:
  - GPU architecture and GPU programming model
- Today's class:
  - GPU optimizations
- Announcement and reminder
  - HW4 dues tonight. Answers will be released by Friday

40

GPU Performance

41

Lesson 1:
Parallelism is Important

42

Thread Level Parallelism

- Superscalars and VLIWs are useful only if…
  - Program exhibits ILP (Instruction Level Parallelism) in the code
- GPUs are useful only if…
  - Program has **TLP (Thread Level Parallelism)** in the code
  - TLP can be expressed as the number of threads in the code
- How that TLP is laid out in the kernel is also important
  - How many **threads** are in a thread block
    - If less than threads in warp, some **SPs** may get unused
  - How many **thread blocks** are in the grid
    - If less than number of SMs, some **SM**s may get unused

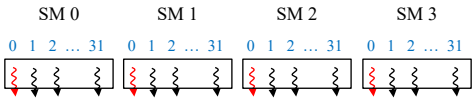
→ If not careful, your GPU may get **underutilized**

43

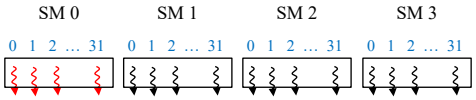
Example: Kernels with Bad Layout

- Suppose there are 4 SMs in GPU with 32 SPs in each SM.
  - Case 1, 2 below have enough TLP (1024 threads) but bad layout.
  - Utilized threads are marked in red. Rest are unused.

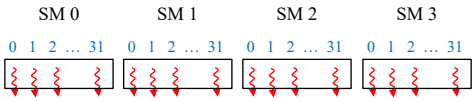
- Case 1: Not enough threads
  kernel<<<1024, 1>>>(…) ;



- Case 2: Not enough blocks
  kernel<<<1, 1024>>>(…) ;



- Balanced threads and blocks
  kernel<<<32, 32>>>(…) ;
  kernel<<<16, 64>>>(…) ;
  kernel<<<4, 256>>>(…) ;



44

Lesson 2:
Bandwidth is Important

45

Example: Computing $y(i) = A(i,j) * x(j)$

C program (on CPU)

```
void mv_cpu(float* y, float* A,
float* x, int n) {
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
      y[i] += A[i*n + j] * x[j];
}
```

```
void main ()
{
  …
  mv_cpu(y, A, x, n);
  …
}
```

CUDA program (on CPU+GPU)

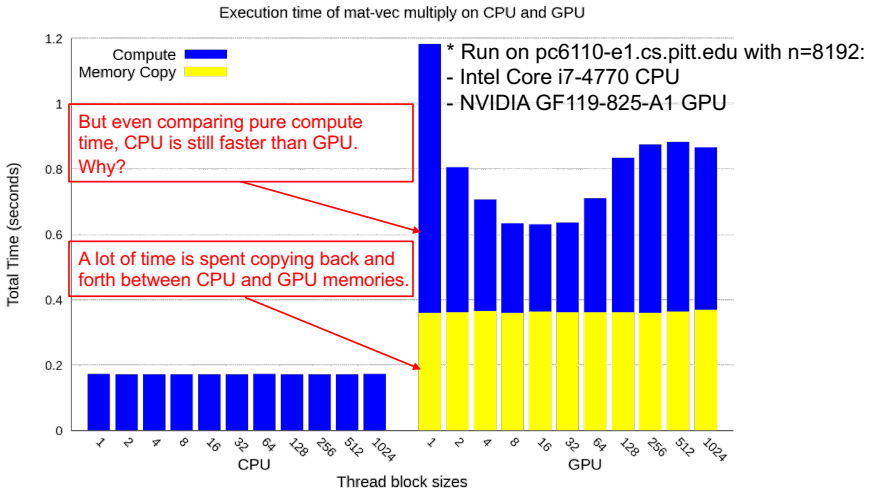
```
void mv_gpu(float* y, float* A, float* x, int n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) {
    for (int j = 0; j < n; j++)
      y[i] += A[i*n + j] * x[j];
  }
}
```

```
void main ()
{
  …
  int nblocks = (n + block_size - 1) / block_size;
  cudaMemcpy (…)
  mv_gpu <<<nblocks, block_size>>> (y, A, x, n);
  cudaMemcpy(…)
  …
}
```

46

Performance Results for $y(i) = A(i,j) * x(j)$

- Guess what? CPU is faster than GPU!



47

Performance Results for $y(i) = A(i,j) * x(j)$

- Was it because the GPU was wimpy and can't do enough **FLOPS**?
- NVIDIA GF119-825-A1 is a Fermi GPU Capability 2.1
  - Max FLOPS = **100.4 GFLOPS**
- What was the FLOPS achieved?
  - y[i] += A[i*n + j] * x[j] = 2 FP ops each iteration for n * n iterations
  - n = 8192, so FP ops = 8192 * 8192 * 2 = 134 M
  - Time = 0.27 seconds (shortest at 32 thread block size)
  - FLOPS = 134 M / 0.27 = **496 MFLOPS**
  - **Not even close to the limit!**

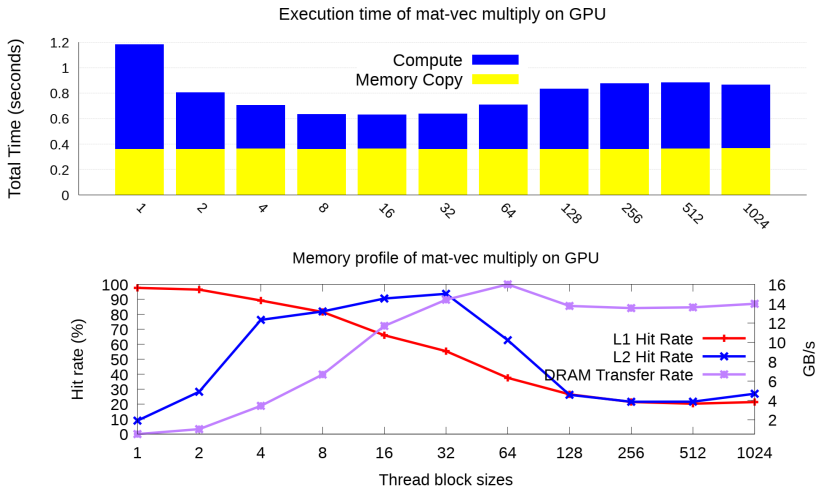
48

Performance Results for $y(i) = A(i,j) * x(j)$

- Could it be that the GPU didn't have enough **memory bandwidth**?
- NVIDIA GF119-825-A1 is a Fermi GPU Capability 2.1
  - Memory Type: DDR3
  - Memory Bandwidth: **14.00 GB/s**
- GPUs also have Performance Monitoring Units (PMUs)
  - NVIDIA Profiler (nvprof) provides an easy way to read them: https://docs.nvidia.com/cuda/profiler-users-guide/index.html
  - Let's use the PMU to profile the following:
    - DRAM Transfer Rate (GB/s)
    - L1 Hit Rate (%)
    - L2 Hit Rate (%)

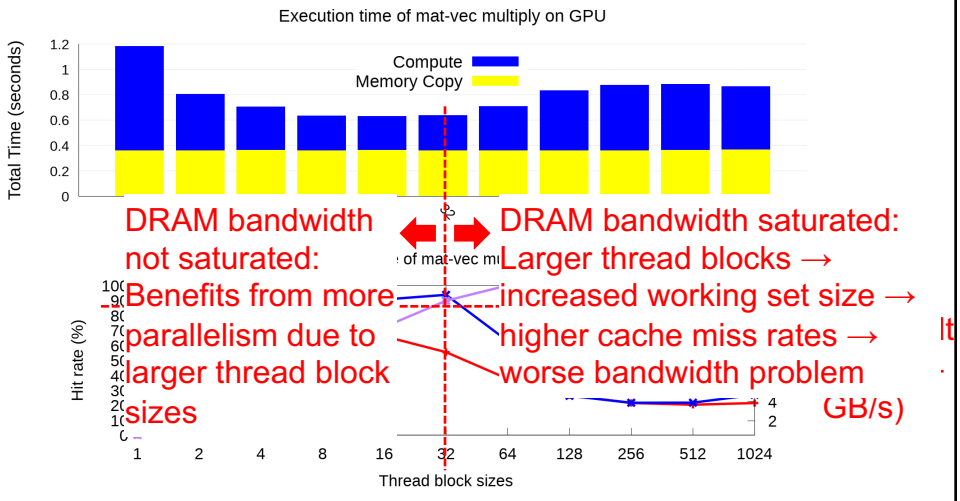
49

Memory Wall Hits Again



50

Memory Wall Hits Again



51

Is there a way we can reach max FLOPS?

- Let's take a look at the GPU design metrics again:
  - Max FLOPS = 100.4 GFLOPS
  - Memory Bandwidth: 14.00 GB/s
- To sustain max FLOPS, you need to do a lot of work per byte
  - 100.4 GFLOPS / 14.00 GB/s = 7.17 FP ops / byte
  - Or, about **28 FP ops / float** (4 bytes) fetched from memory
  - Otherwise, the memory bandwidth cannot sustain the FLOPS
- All GPUs have this problem with **memory bandwidth**:
  - It's easy to put in more SMs using transistors for Moore's Law
  - Your memory bandwidth is limited due to your DDR interface

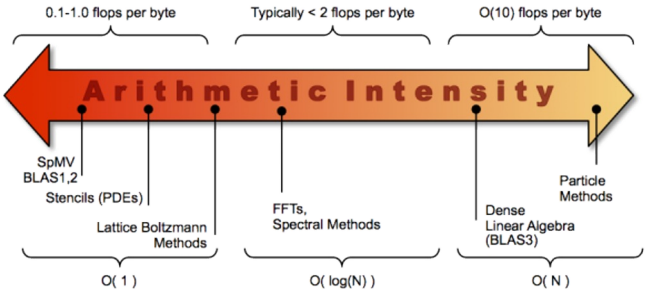
52

Arithmetic Intensity: A property of the program

- How many **FP ops / float** for our mat-vec multiplication?
  - y[i] += A[i*n + j] * x[j] each iteration with n * n iterations
  - FP ops = **2 * n * n** (one multiply and one add)
  - Float accesses = **n * n + 2n** (1 matrix and 2 vector accesses)
    - That's counting only cold misses but could be even more
  - So approx. **2 FP ops / float** (a far cry from 28 FP ops / float)
  - This metric is called **arithmetic intensity**
- Arithmetic intensity is a **property of the program** needed by GPUs
  - Just like TLP (thread-level-parallelism) is needed by GPUs
  - Matrix-vector multiplication has low intensity
    → Fundamentally not suited for fast GPU computation

53

Arithmetic Intensity: A property of the program



* Courtesy of Lawrence Berkeley National Laboratory:
https://crd.lbl.gov/departments/computer-science/par/research/roofline/introduction/

- **BLAS**: Basic Linear Algebra Subprograms
  - BLAS 1: Vector operations only (e.g. saxpy) → Bad intensity
  - BLAS 2: **Ge**neral **M**atrix-**V**ector Multiplication (**GeMV**) → Bad intensity
  - BLAS 3: **Ge**neral **M**atrix **M**ultiplication (**GeMM**) → Good intensity

54

Matrix-Matrix Multiply: Good Arithmetic Intensity

- Matrix-multiplication:

```
for (int i=0; i<n; i++)
  for (int j=0; j<n; j++)
    for (int k=0; k<n; k++)
      C[i*n + j] += A[i*n + k] * B[k*n + j];
```

- What's the arithmetic intensity for this program?
  - FP ops = **2 * n * n * n** (one multiply and one add)
  - Float accesses = **3 * n * n** (3 matrix accesses)
    - If we only have cold misses and no capacity misses
  - Arithmetic intensity = 2 * n / 3 = **0.66 * n** = **O(n)**
  - Implication: The larger the matrix size, the better suited for GPUs!
    - Important result for deep learning and other apps

55

Example: Computing $C(i,j) = A(i,k) * B(k,j)$

C program (on CPU)

```
void mm_cpu(float* C, float* A, float* B, int n) {
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
      for (int k=0; k<n; k++)
        C[i*n + j] += A[i*n + k] * B[k*n + j];
}
```

```
void main ()
{
  mm_cpu(C, A, B, n);
}
```

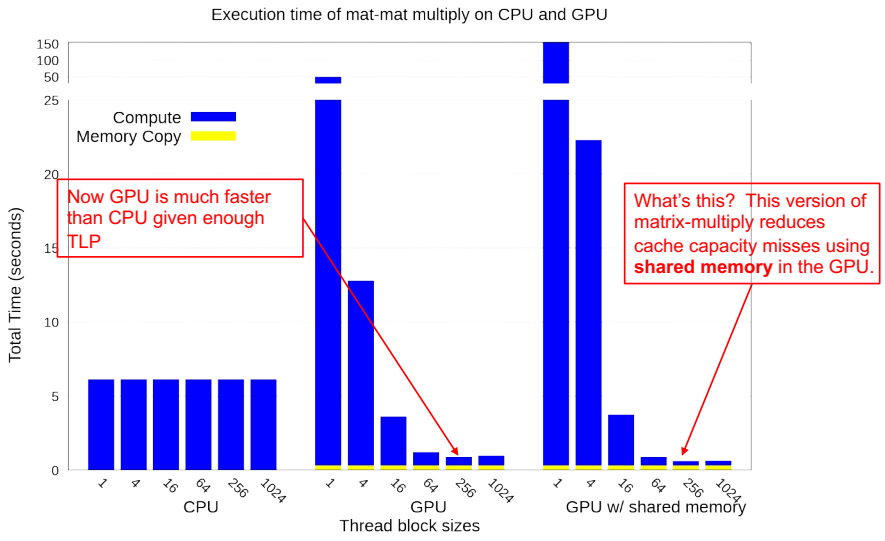
CUDA program (on CPU+GPU)

```
void mm_gpu(float* C, float* A, float* B, int n) {
  float Cvalue = 0;
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  for (int k = 0; k < n; ++k)
    Cvalue += A[i * n + k] * B[k * n + j];
  C[i * n + j] = Cvalue;
}
```

```
void main ()
{
  dim3 dimBlock(block_size, block_size);
  dim3 dimGrid(n / dimBlock.x, n / dimBlock.y);
  mm_gpu <<<dimGrid, dimBlock>>> (C, A, B, n);
}
```

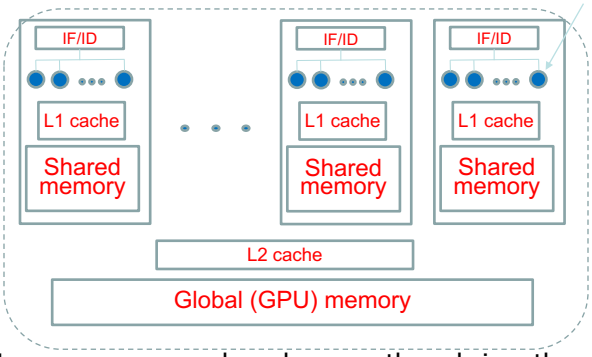
56

Performance Results for $C(i,j) = A(i,k) * B(k,j)$



57

So what is Shared Memory?



- **Shared Memory**: memory shared among threads in a thread block
  - Variables declared with **__shared__** modifier live in shared memory
  - Is same as L1 cache in terms of latency and bandwidth!
  - Storing frequently used data in shared memory can save on bandwidth

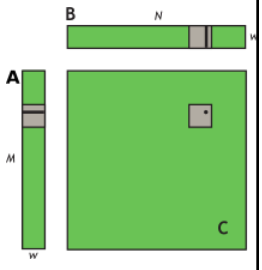
59

Loop Tiling with Shared Memory

- Store a "tile" within matrix in shared memory while operating on it
  - Can reduce accesses to DRAM memory

- Code in: *https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#shared-memory-in-matrix-multiplication-c-ab*

```
__shared__ float aTile[TILE_DIM][TILE_DIM],
                 bTile[TILE_DIM][TILE_DIM];
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
float sum = 0.0f;
aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
__syncthreads();
for (int i = 0; i < TILE_DIM; i++)
  sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
c[row*N+col] = sum;
```



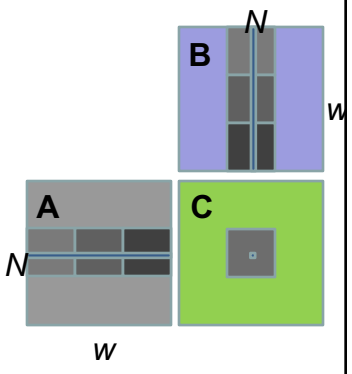
- Assumption: TILE_DIM = w. What if w > TILE_DIM?

60

Loop Tiling with Shared Memory

- TILE_DIM is limited by amount of shared memory. What if w > TILE_DIM?
  - Now must load A and B tiles **w / TILE_DIM** times per thread block
- Now code will look like:

```
__shared__ float aTile[TILE_DIM][TILE_DIM],
                 bTile[TILE_DIM][TILE_DIM];
float sum = 0.0f;
for (int t = 0; t < w / TILE_DIM; t++) {
  …
  aTile[threadIdx.y][threadIdx.x] = …;
  bTile[threadIdx.y][threadIdx.x] = …;
  __syncthreads();
  for (int i = 0; i < TILE_DIM; i++)
    sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
}
c[row*N+col] = sum;
```



61

Lesson 3:
Programmability vs. Performance

62

Explicit memory management

Explicit Memory Management

```
void *data, *d_data;
data = malloc(N);
cudaMalloc(&d_data, N);
cpu_func1(data, N);
cudaMemcpy(d_data, data, N, . . . )
gpu_func2<<<...>>>(d_data, N);
cudaMemcpy(data, d_data, N,
. . . )   cudaFree(d_data);
cpu_func3(data, N);

free(data);
```

Three problems:

- GPU memory capacity limitation
- Difficult to program
- Poor portability

63

SINGLE POINTER

Explicit vs Unified Memory

Explicit Memory Management

```
void *data, *d_data;
data = malloc(N);
cudaMalloc(&d_data, N);
cpu_func1(data, N);
cudaMemcpy(d_data, data, N, . . . )
gpu_func2<<<...>>>(d_data, N);
cudaMemcpy(data, d_data, N,
. . . )   cudaFree(d_data);
cpu_func3(data, N);

free(data);
```

GPU code w/ Unified Memory

```
void *data;
data = malloc(N);

cpu_func1(data, N);

gpu_func2<<<...>>>(data, N);
cudaDeviceSynchronize();

cpu_func3(data, N);

free(data);
```

64

SINGLE POINTER

Deep Copy

Explicit Memory Management

```
char **data;
// allocate and initialize data on the CPU

char **d_data;
char **h_data = (char**)malloc(N*sizeof(char*));
for (int i =0; i <N; i++) {
  cudaMalloc(&h_data[i], N);
  cudaMemcpy(h_data[i], data[i], N, ...);
}
cudaMalloc(&d_data, N*sizeof(char*));
cudaMemcpy(d_data, h_data, N*sizeof(char*), ...);

gpu_func<<<...>>>(d_data, N);
```

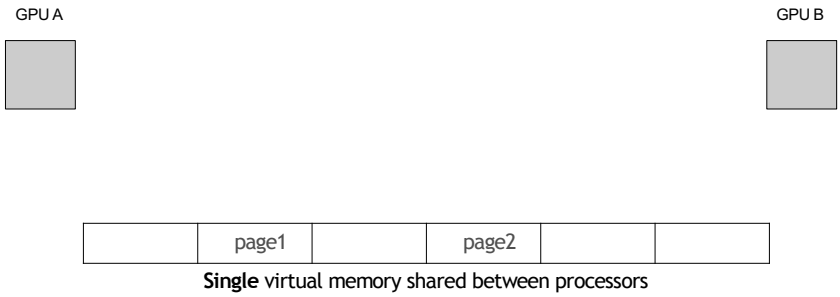
GPU code w/ Unified Memory

```
char **data;
// allocate and initialize data on the CPU

gpu_func<<<...>>>(data, N);
```

65

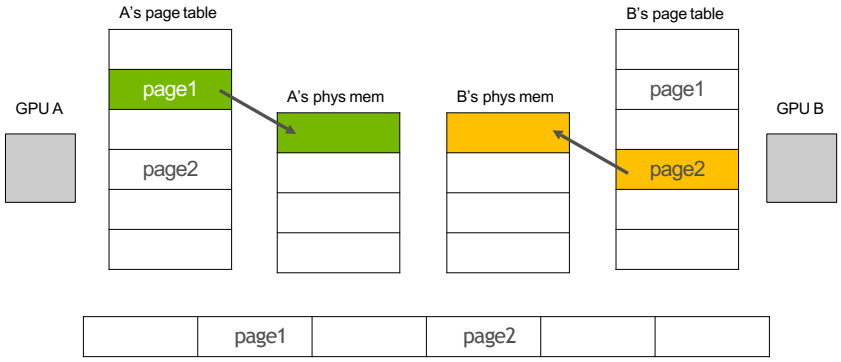
UNIFIED MEMORY BASICS



Single virtual memory shared between processors

66

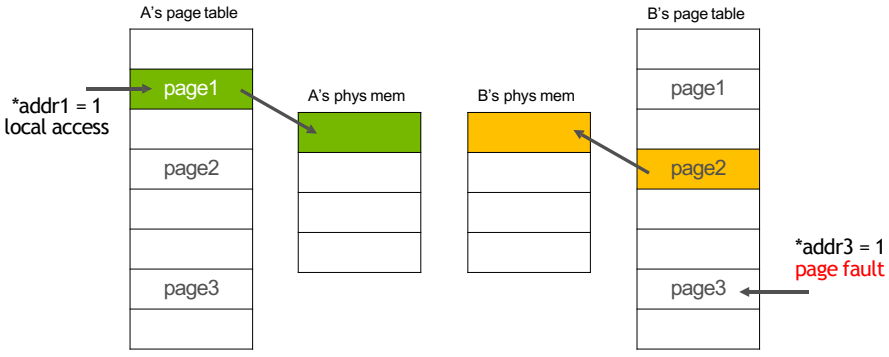
UNIFIED MEMORY BASICS



Single virtual memory shared between processors

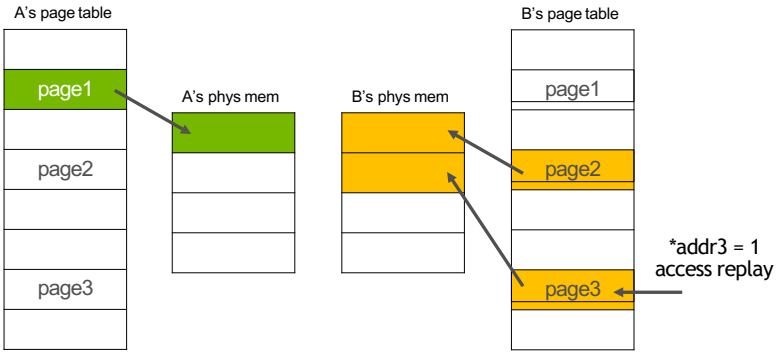
67

UNIFIED MEMORY BASICS



68

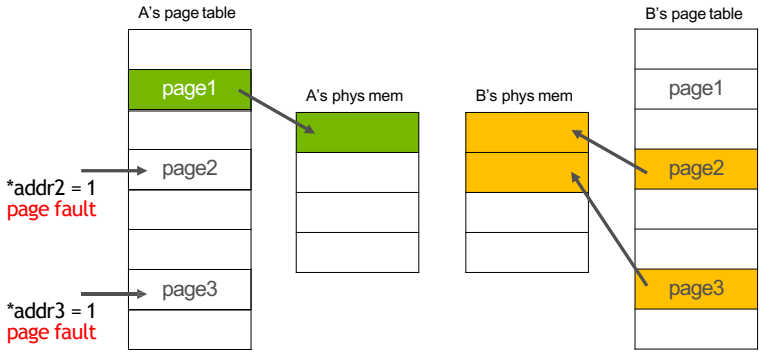
UNIFIED MEMORY BASICS



page3 populated and mapped into B's memory

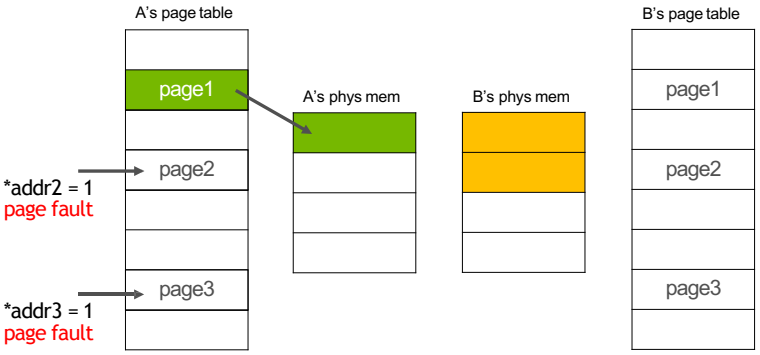
69

UNIFIED MEMORY BASICS



70

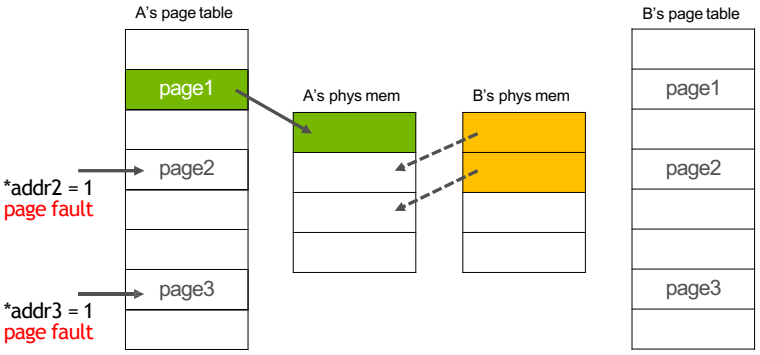
UNIFIED MEMORY BASICS



page2 and page3 unmapped from B's memory

71

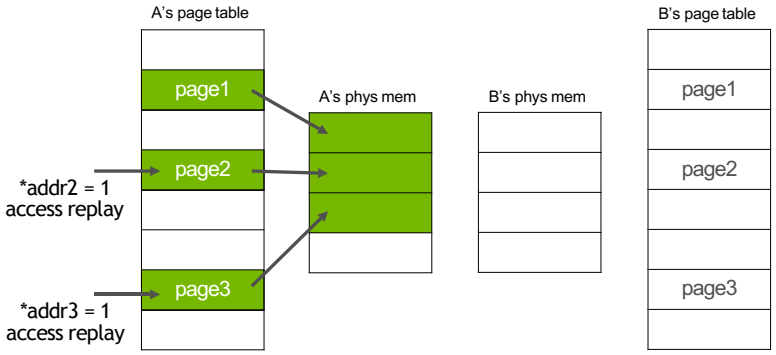
UNIFIED MEMORY BASICS



pages data migrated to A's physical memory

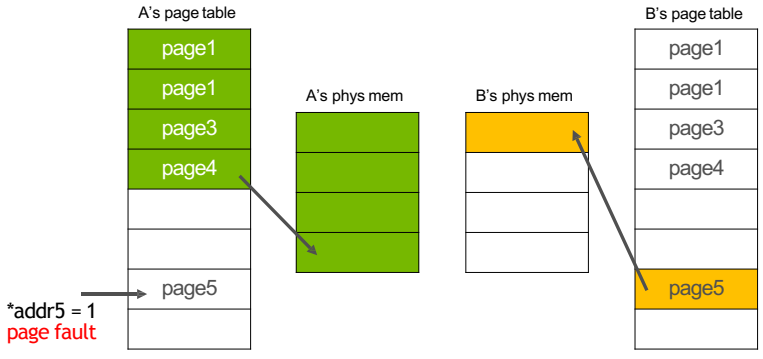
72

UNIFIED MEMORY BASICS



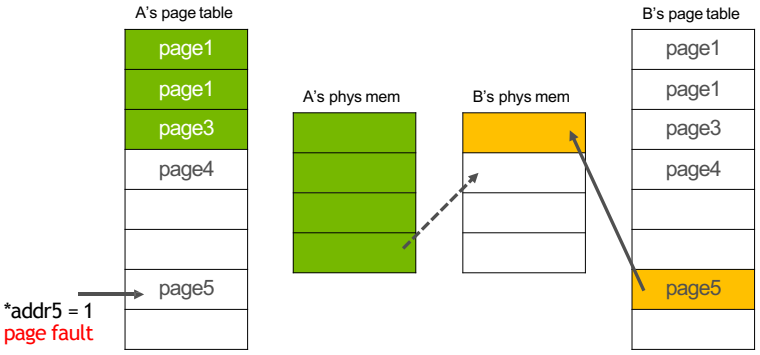
73

MEMORY OVERSUBSCRIPTION



74

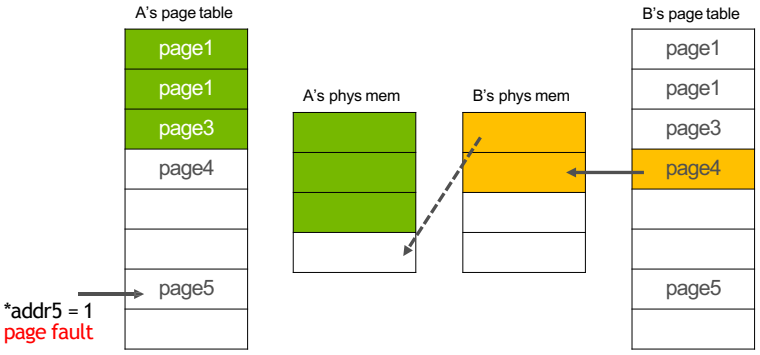
MEMORY OVERSUBSCRIPTION



page4 unmapped from A's memory and migrated

75

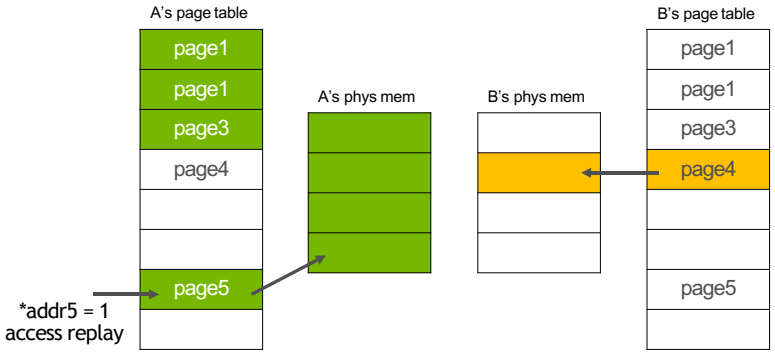
MEMORY OVERSUBSCRIPTION



page4 mapped in B's memory, page5 unmapped and migrated to A

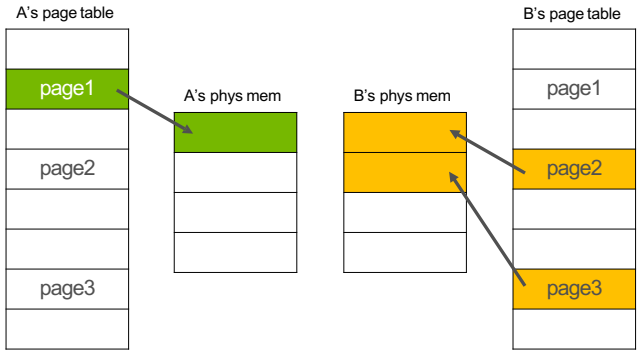
76

MEMORY OVERSUBSCRIPTION



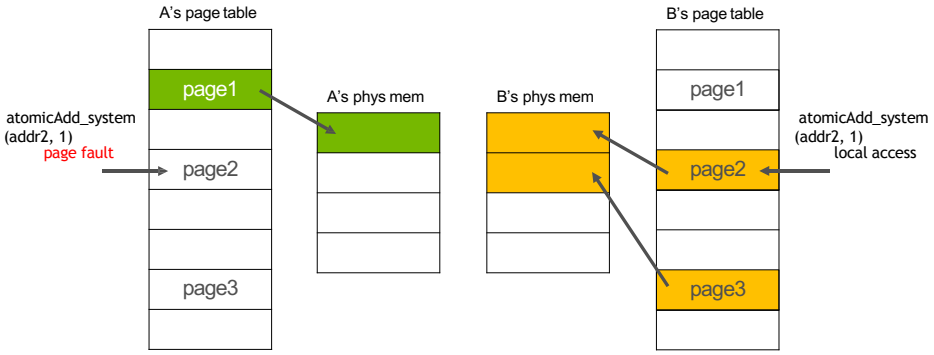
77

CONCURRENT ACCESS



CONCURRENT ACCESS

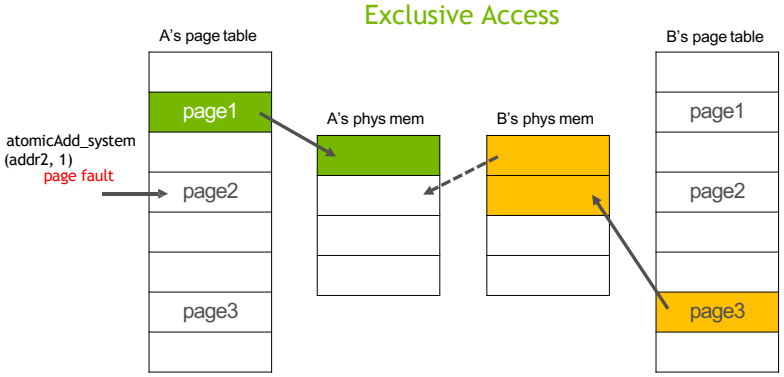
Exclusive Access*



*this is a possible implementation and to guarantee this behavior you need to use cudaMemAdvise policies

CONCURRENT ACCESS

Exclusive Access

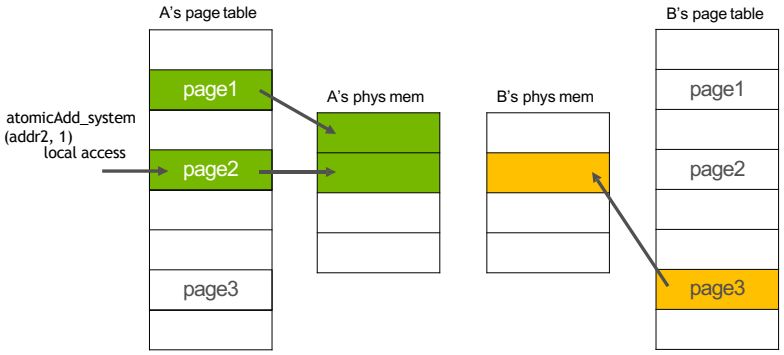


page2 unmapped in B's memory and migrated to A

80

CONCURRENT ACCESS

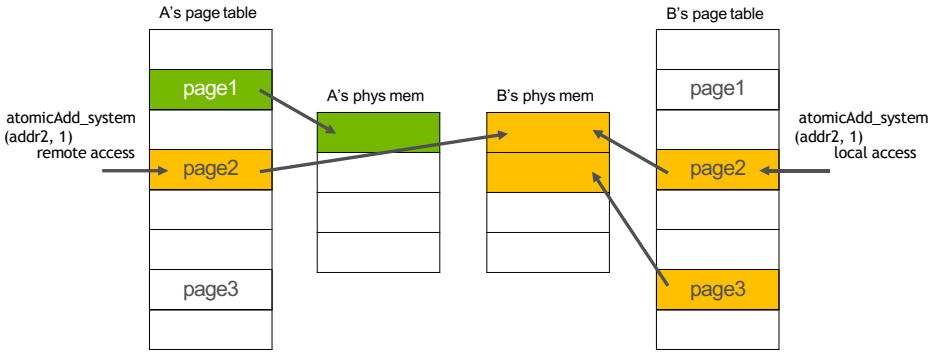
Exclusive Access



81

CONCURRENT ACCESS

Atomics over NVLINK*

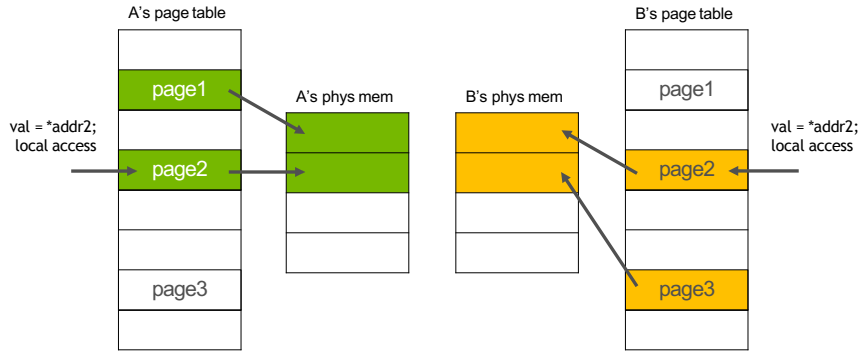


*both processors need to support atomic operations

82

CONCURRENT ACCESS

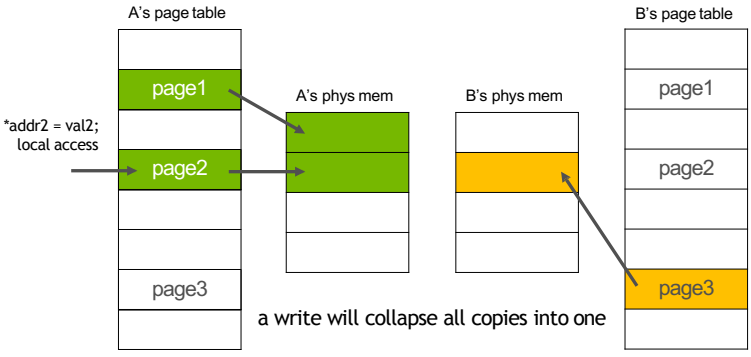
Read duplication*



*each processor must maintain its own page table

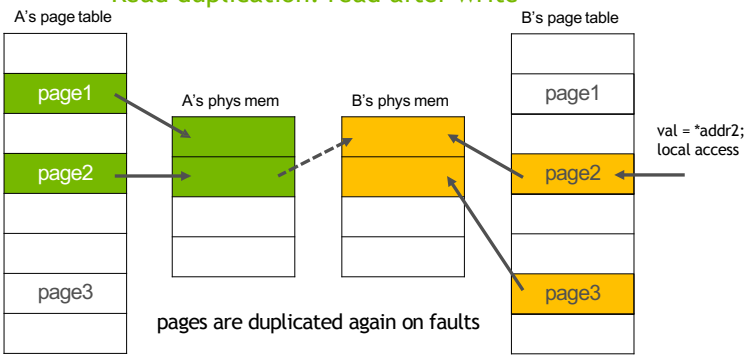
CONCURRENT ACCESS

Read duplication: write



CONCURRENT ACCESS

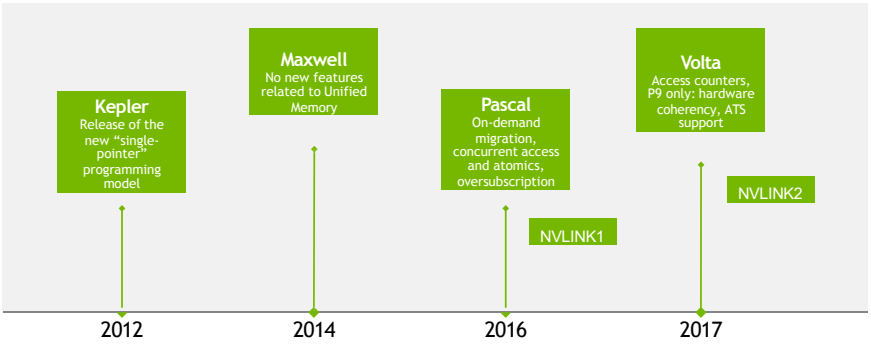
Read duplication: read after write



85

UNIFIED MEMORY

Evolution of GPU architectures



*Not all features are available on all platforms