King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 27: CUDA Unified Memory; More on Instruction Level Parallelism

Markus Hadwiger, KAUST

Reading Assignment #15++ (until Dec 14++)

Further suggested reading:

- Raihan et al., arXiv, Feb 2019, Modeling Deep Learning Accelerator Enabled GPUs
  - **https://arxiv.org/abs/1811.08309**
  - See also GPGPU-SIM: **http://www.gpgpu-sim.org/**
- CUTLASS 2.4 template library (last update Nov 2020)
  - **https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/**
  - **https://github.com/NVIDIA/cutlass**
- Register Cache: Caching for Warp-Centric CUDA Programs
  - **https://devblogs.nvidia.com/register-cache-warp-cuda/**
- cuSPARSE library description in the CUDA SDK
- CUSP library: **http://cusplibrary.github.io/**
- Incomplete-LU and Cholesky Preconditioned Iterative Methods Using CUSPARSE and CUBLAS, Maxim Naumov
  - **https://developer.download.nvidia.com/assets/cuda/files/psts_white_paper_final.pdf**

See **https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/**
and **http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf**

SINGLE POINTER

CPU vs GPU

CPU code

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

cpu_func1(data, N);

cpu_func2(data, N);


cpu_func3(data, N);

free(data);
```

GPU code w/ Unified Memory

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

cpu_func1(data, N);

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

cpu_func3(data, N);

free(data);       cudaFree()
```

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); cudaMallocManaged()

cpu_func1(data, N);

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

cpu_func3(data, N);

free(data);       cudaFree()
```

SINGLE POINTER

Full Control with Prefetching

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);
```

Unified Memory + Prefetching

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

cpu_func1(data, N);
cudaMemPrefetchAsync(data, N, GPU)
gpu_func2<<<...>>>(data, N);
cudaMemPrefetchAsync(data, N, CPU)
cudaDeviceSynchronize();
cpu_func3(data, N);

free(data);       cudaFree()
```

SINGLE POINTER

Deep Copy

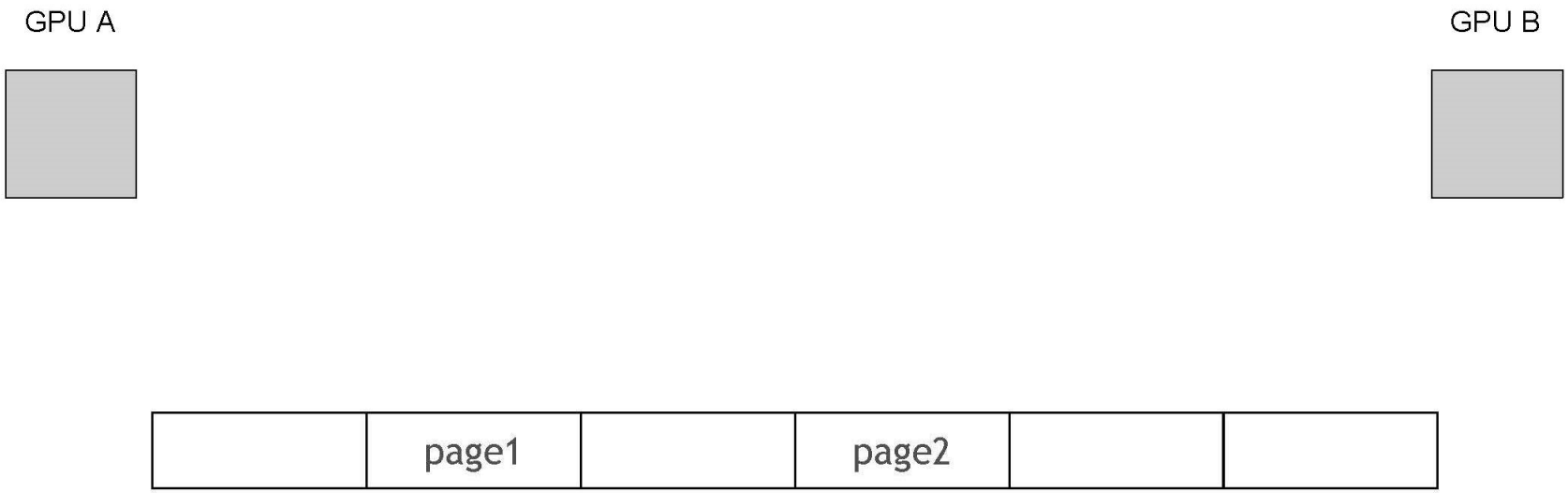
Explicit Memory Management

```
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

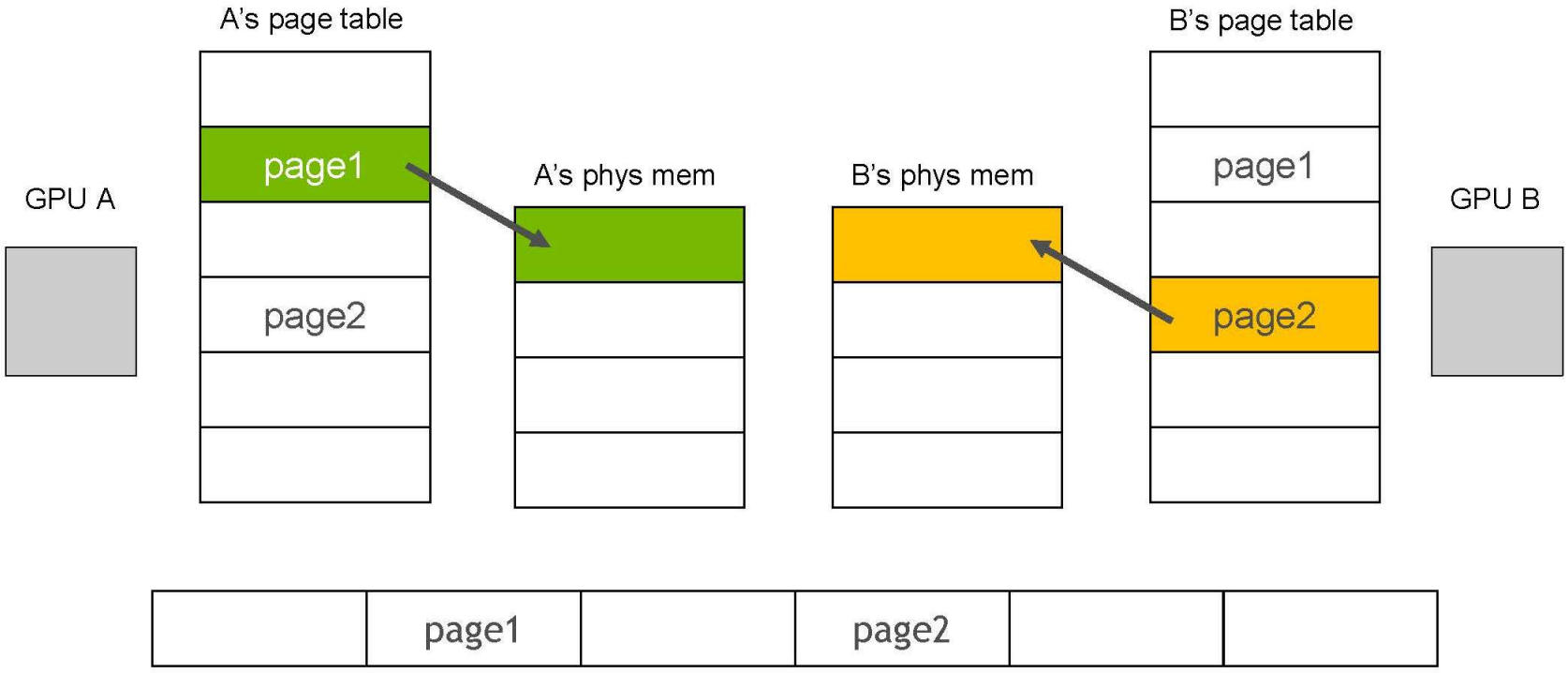
```
char **data;
// allocate and initialize data on the CPU

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

UNIFIED MEMORY BASICS



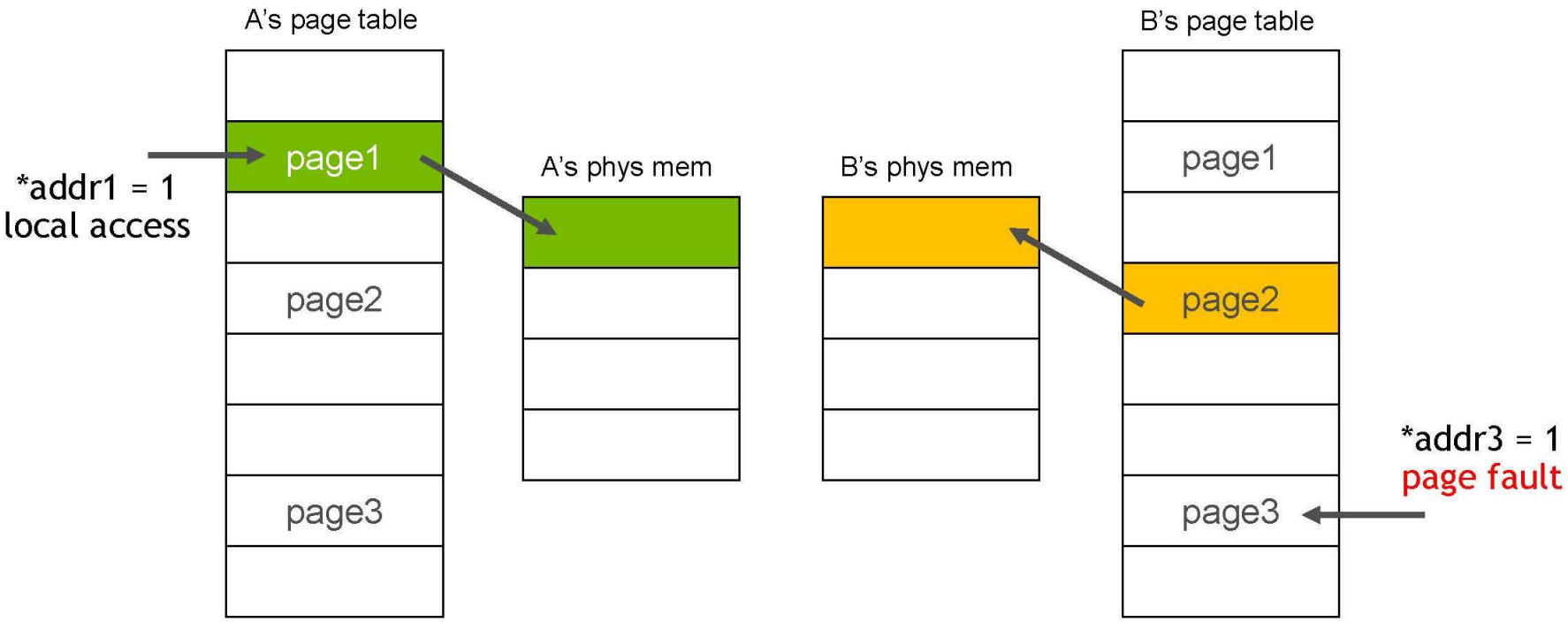
Single virtual memory shared between processors

UNIFIED MEMORY BASICS

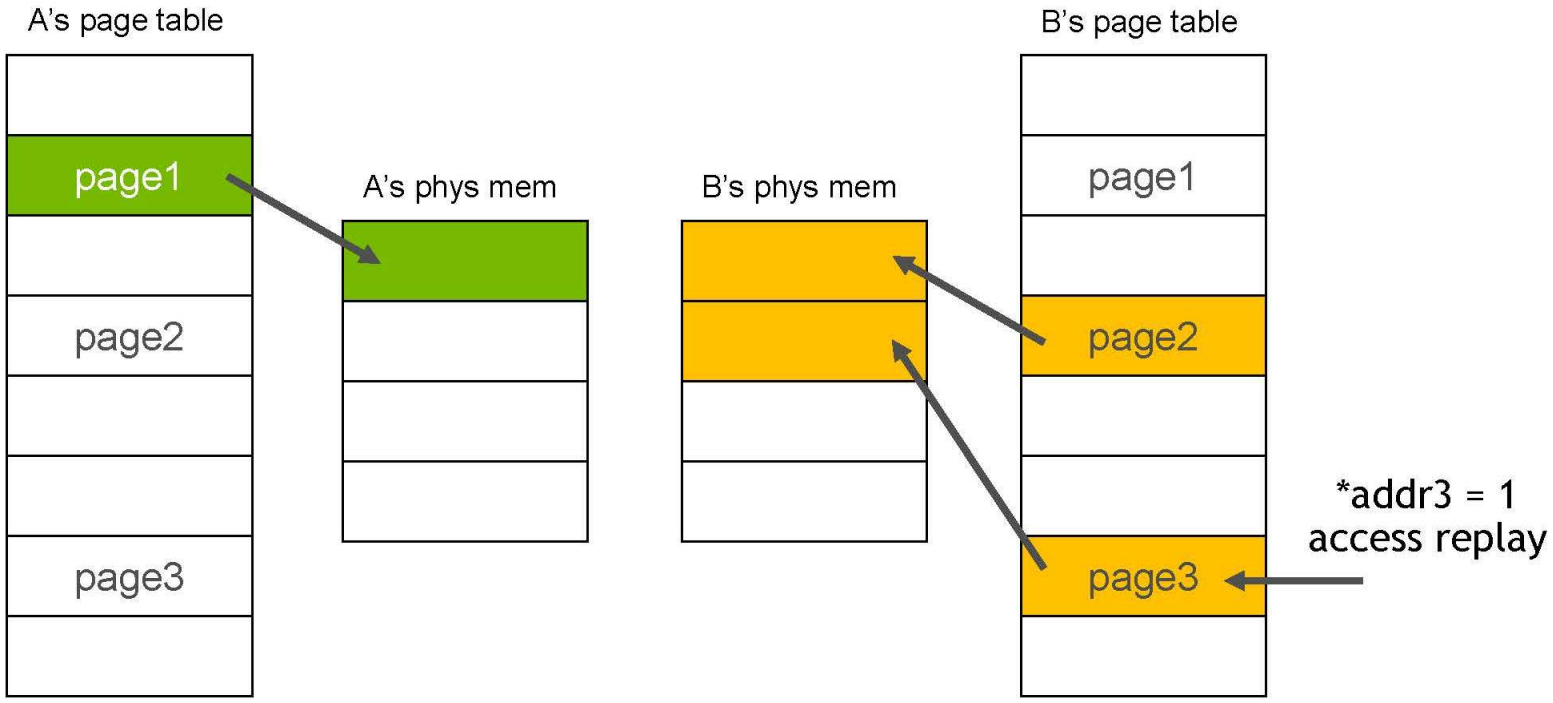


Single virtual memory shared between processors

UNIFIED MEMORY BASICS

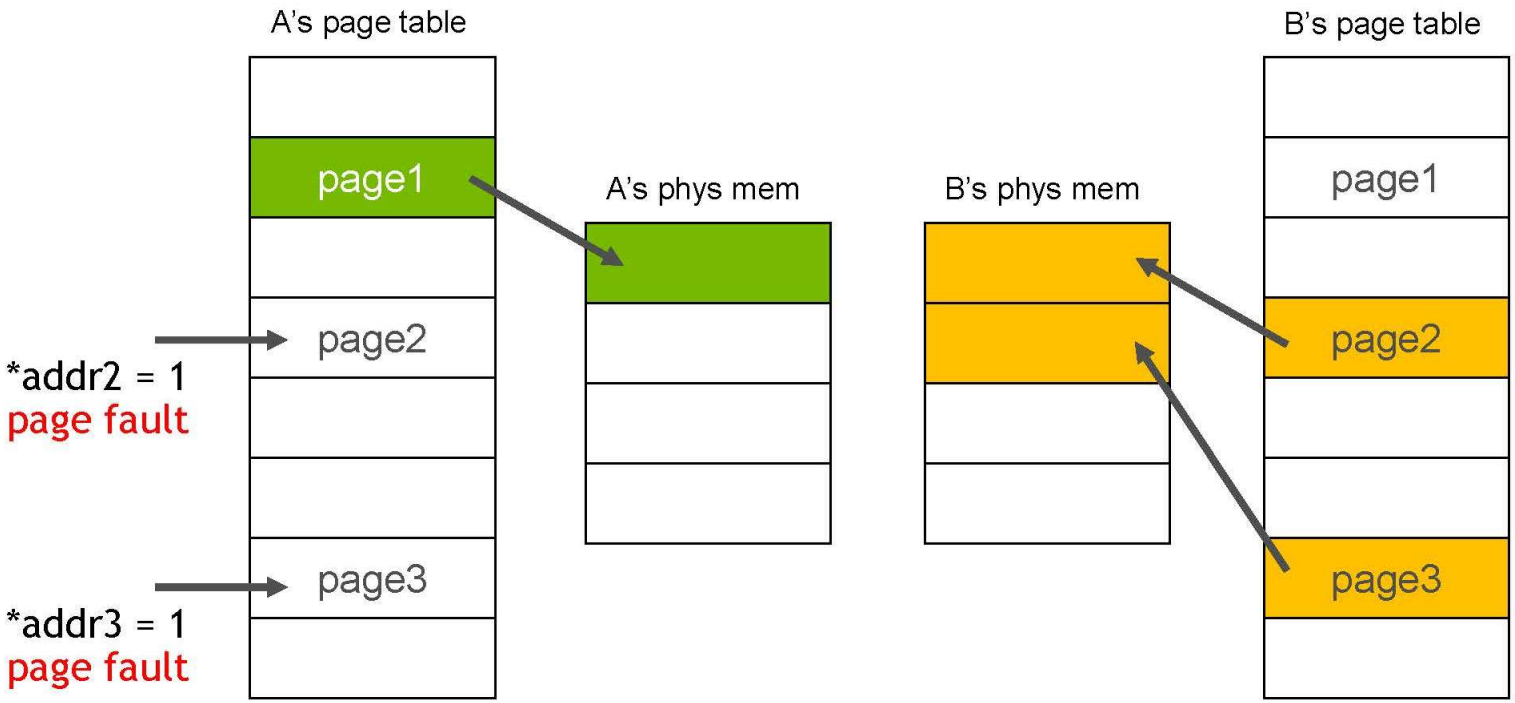


UNIFIED MEMORY BASICS

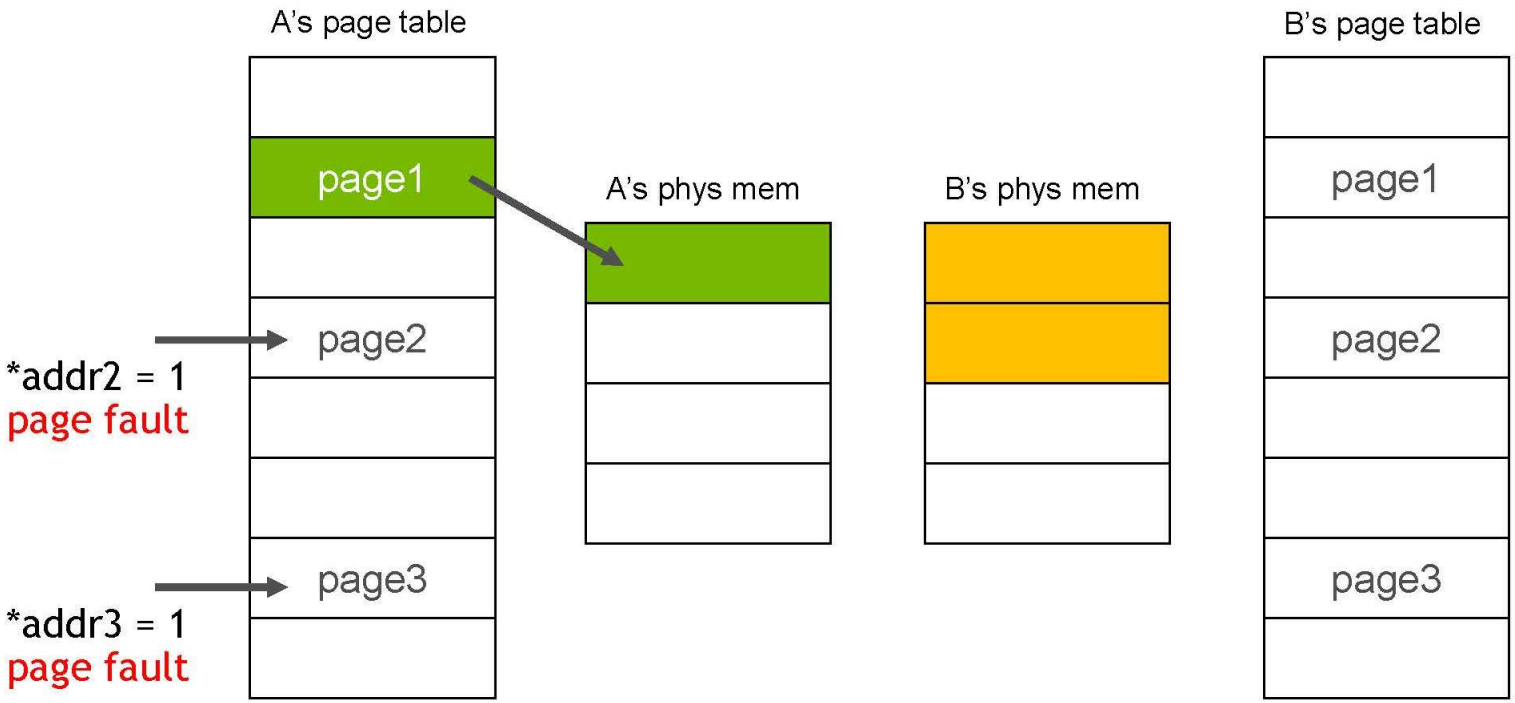


page3 populated and mapped into B's memory

UNIFIED MEMORY BASICS

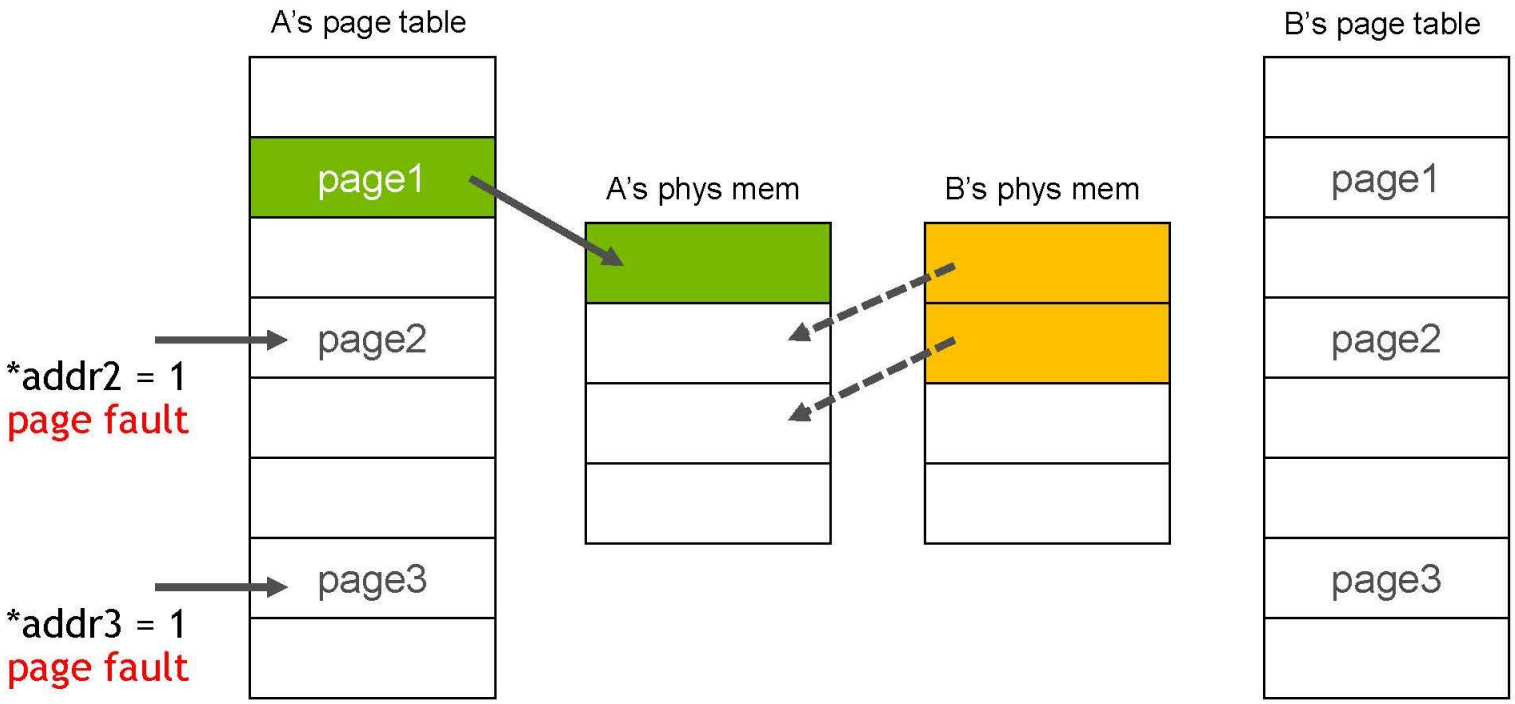


UNIFIED MEMORY BASICS



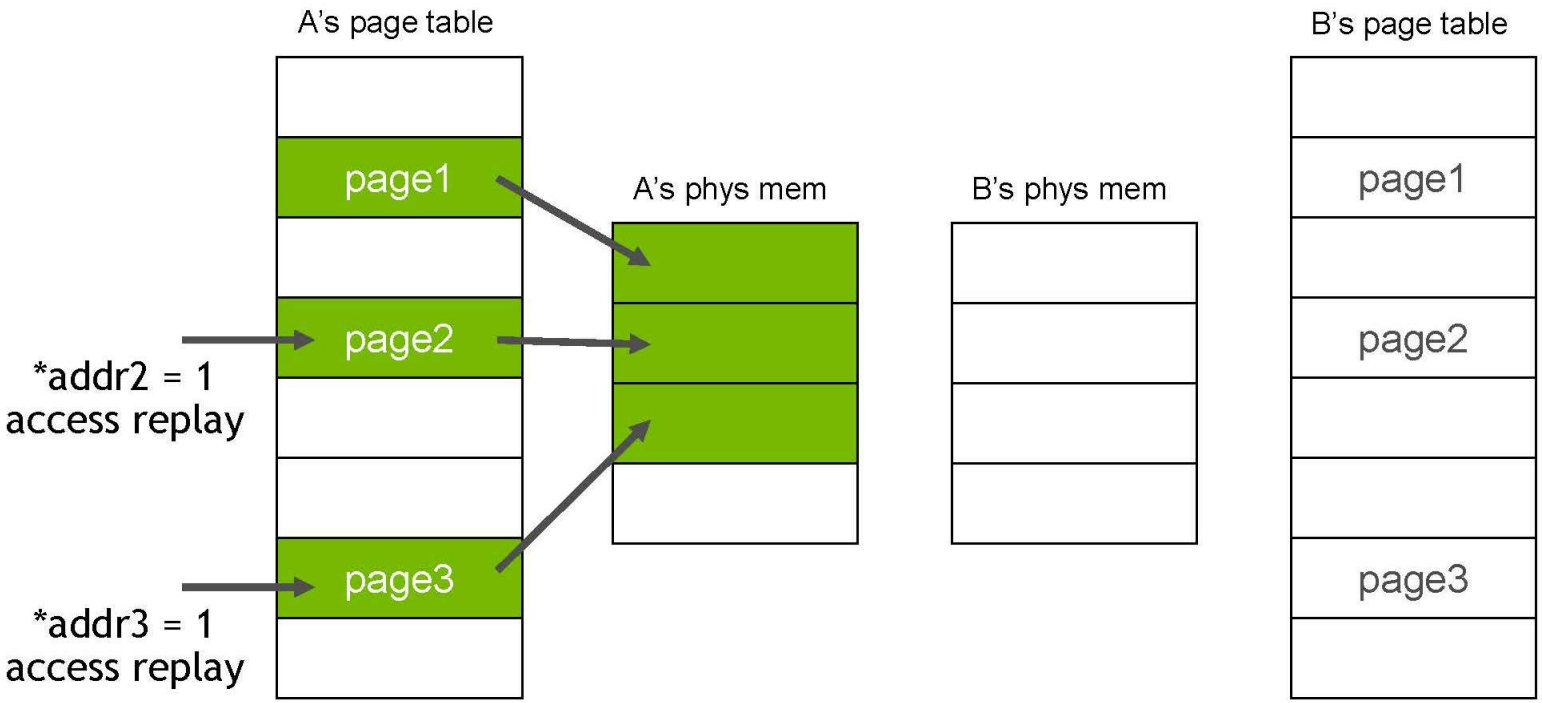
page2 and page3 unmapped from B's memory

UNIFIED MEMORY BASICS

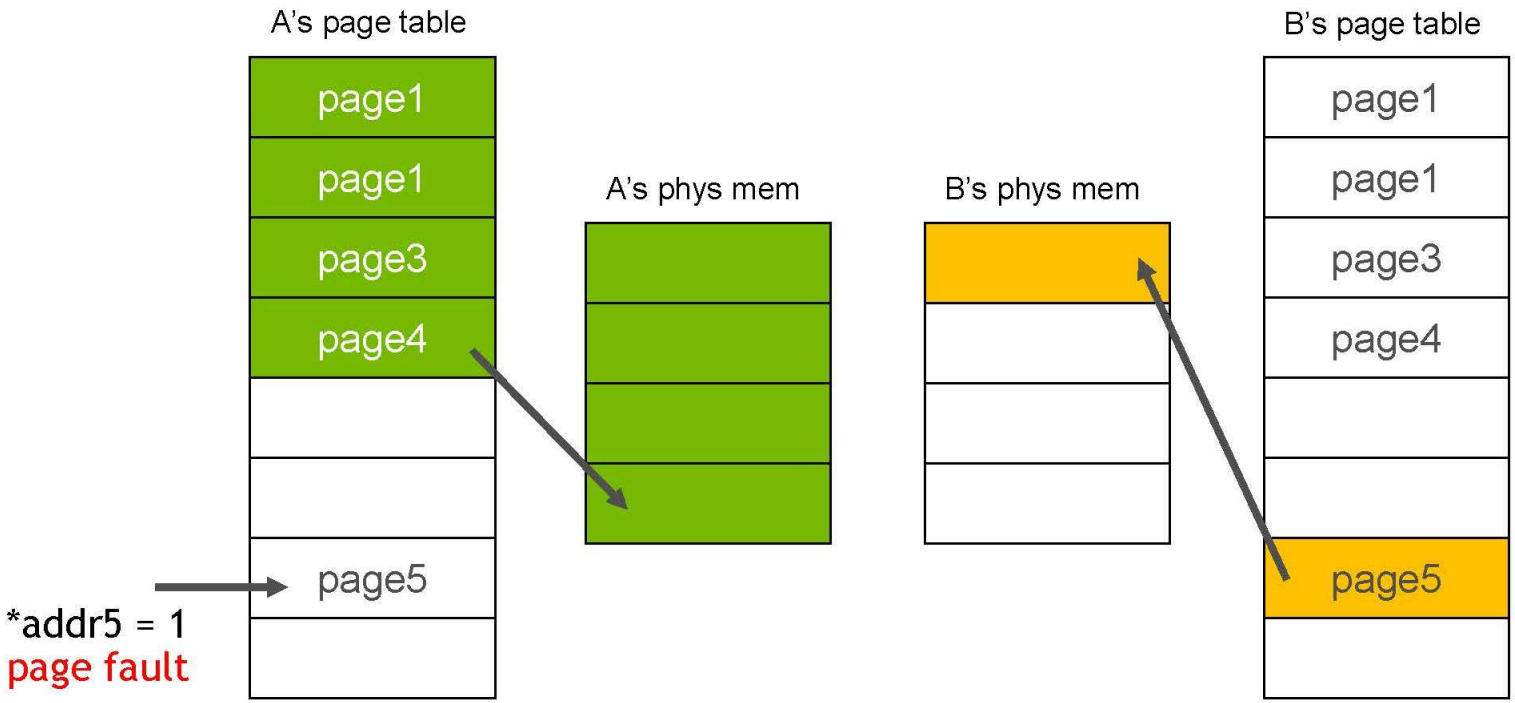


pages data migrated to A's physical memory

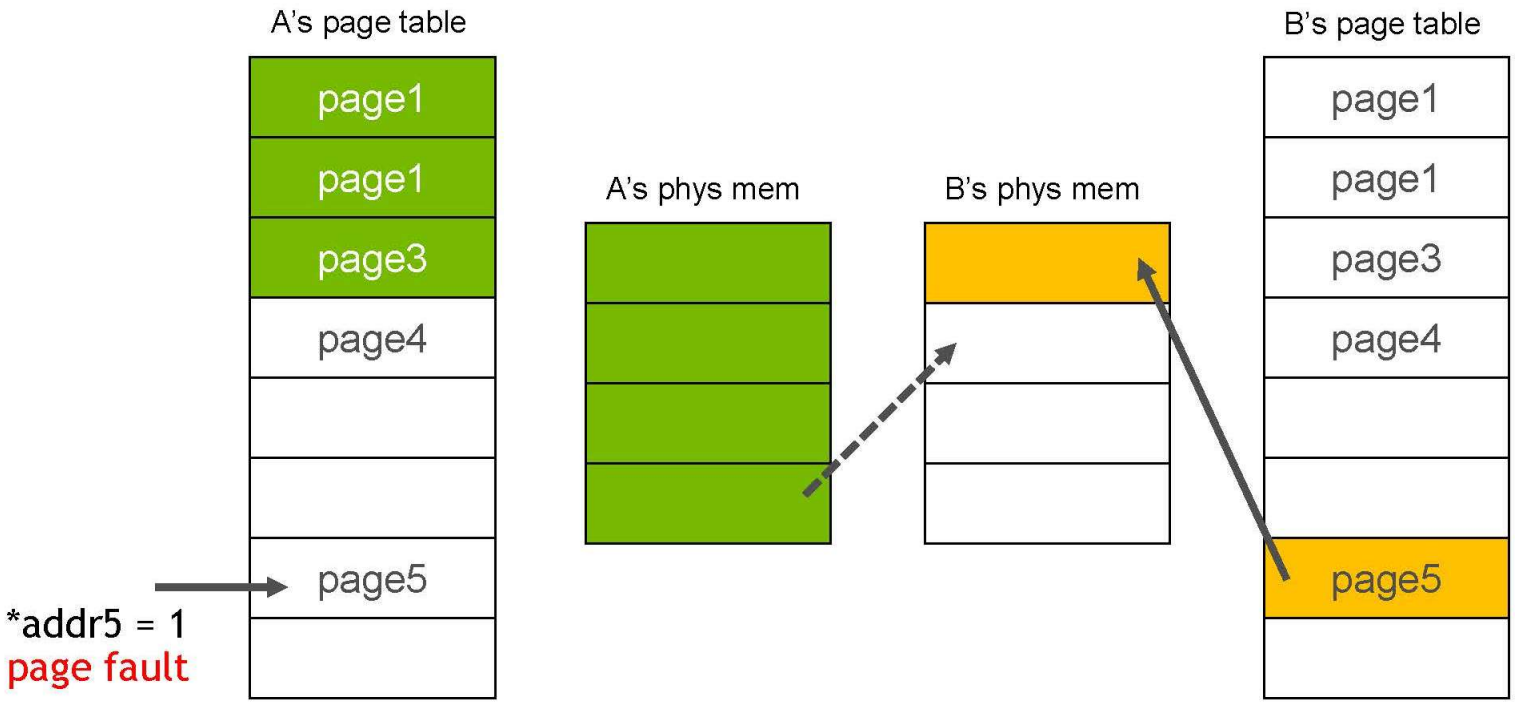
UNIFIED MEMORY BASICS



MEMORY OVERSUBSCRIPTION

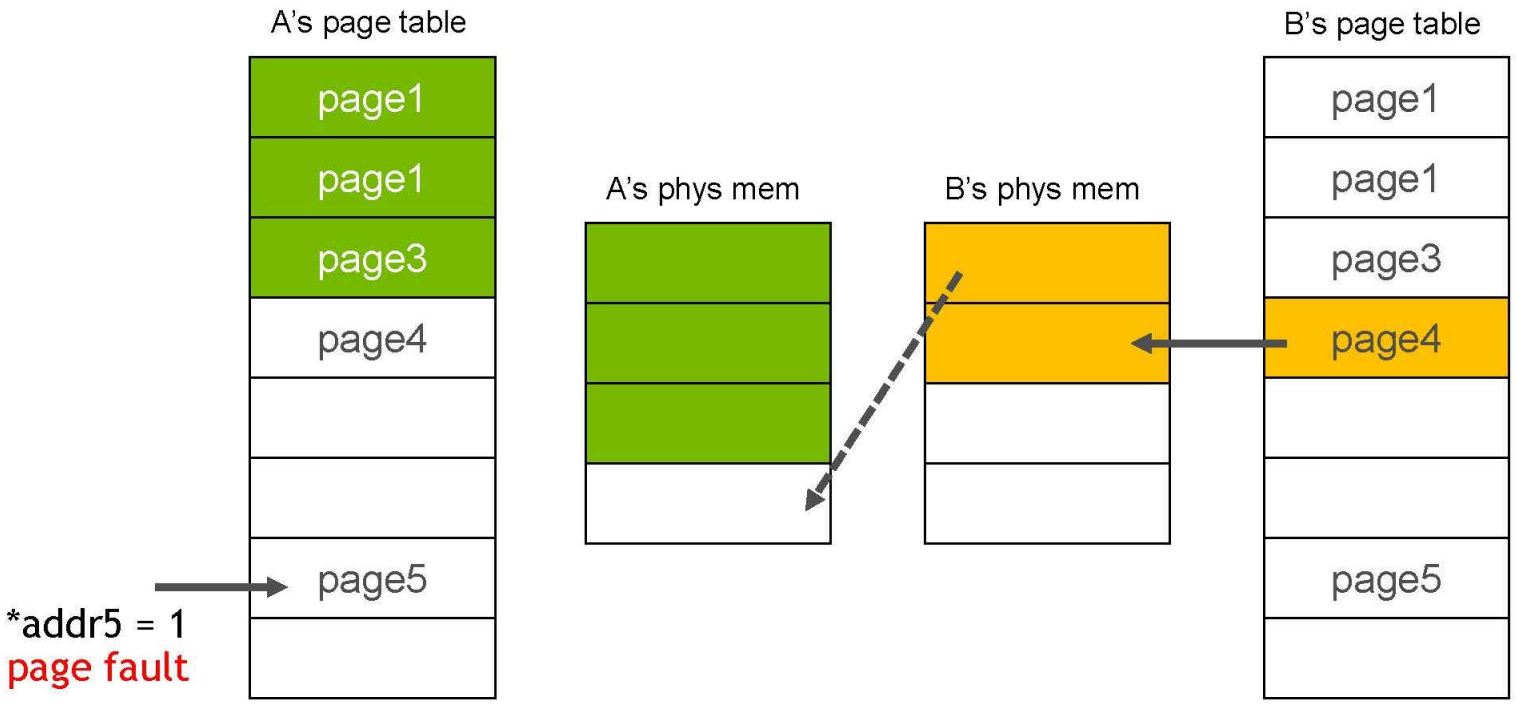


MEMORY OVERSUBSCRIPTION



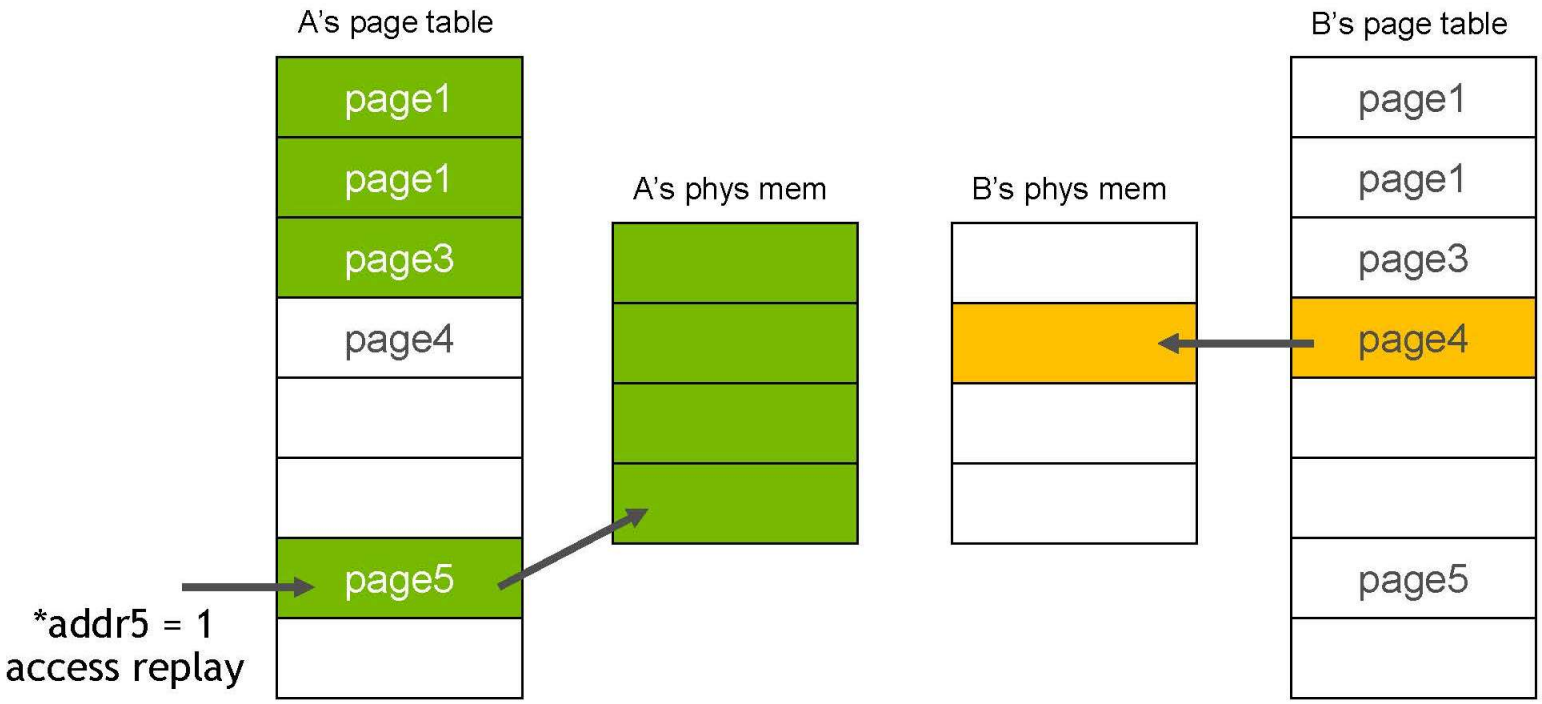
page4 unmapped from A's memory and migrated

MEMORY OVERSUBSCRIPTION



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

MEMORY OVERSUBSCRIPTION



SIMPLIFYING DL FRAMEWORK DESIGN

Eliminated **3,000 lines** of repetitive and error-prone code in Caffe

Developers can add new inherited Layer classes in a much simpler manner

The final call to a CPU function or a GPU kernel (caffe_gpu_gemm) still need to be explicit

Existing Design

```
class ConvolutionLayer
{
public:
   void cpu_data()
   void cpu_diff()
   void gpu_data()
   void gpu_diff()

   void mutable_cpu_data()
   void mutable_cpu_diff()
   void mutable_gpu_data()
   void mutable_gpu_diff()

   void Forward_cpu()
   void Forward_gpu()
   void forward_cpu_gemm()
   void forward_gpu_gemm()
   void forward_cpu_bias()
   void forward_gpu_bias()

   void Backward_cpu()
   void Backward_gpu()
   void backward_cpu_gemm()
   void backward_gpu_gemm()
   void backward_cpu_bias()
   void backward_gpu_bias()
}
```

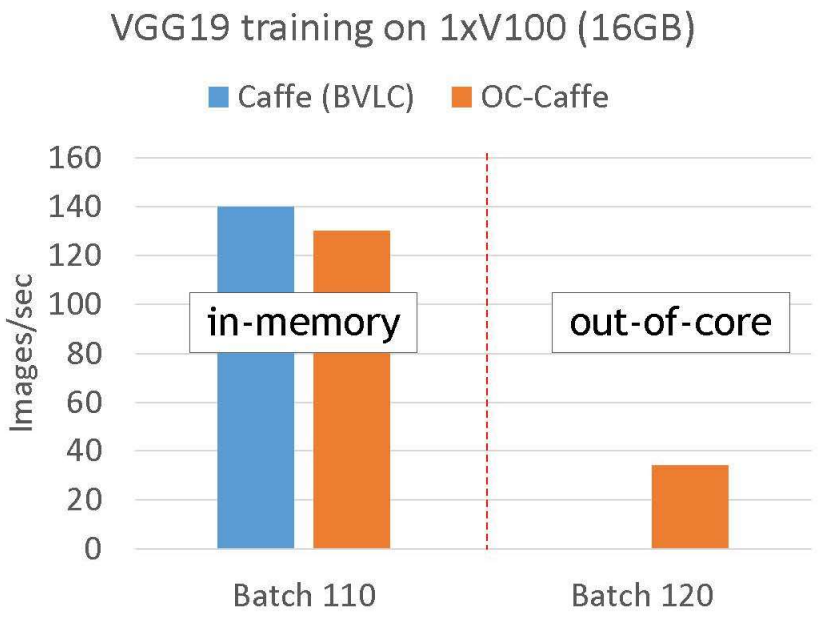
Unified Memory Design

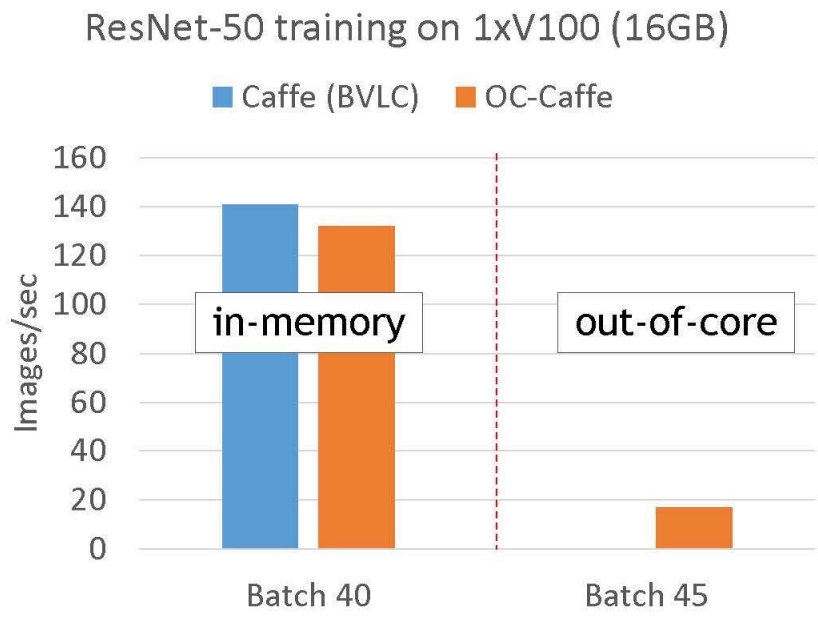
```
class ConvolutionLayer
{
public:
   void data()
   void diff()

   void mutable_data()
   void mutable_diff()

   void Forward()
   void forward_gemm()
   void forward_bias()

   void Backward()
   void backward_gemm()
   void backward_bias()
}
```

A. A. Awan, C-H Chu, H. Subramoni, X. Lu, D.K. Panda, "*OC-DNN: Designing Out-of-Core Deep Neural Network Training by Exploiting Unified Memory on Pascal and Volta GPUs*", <double-blind submission under review>

CAN THIS DESIGN OFFER GOOD PERF?

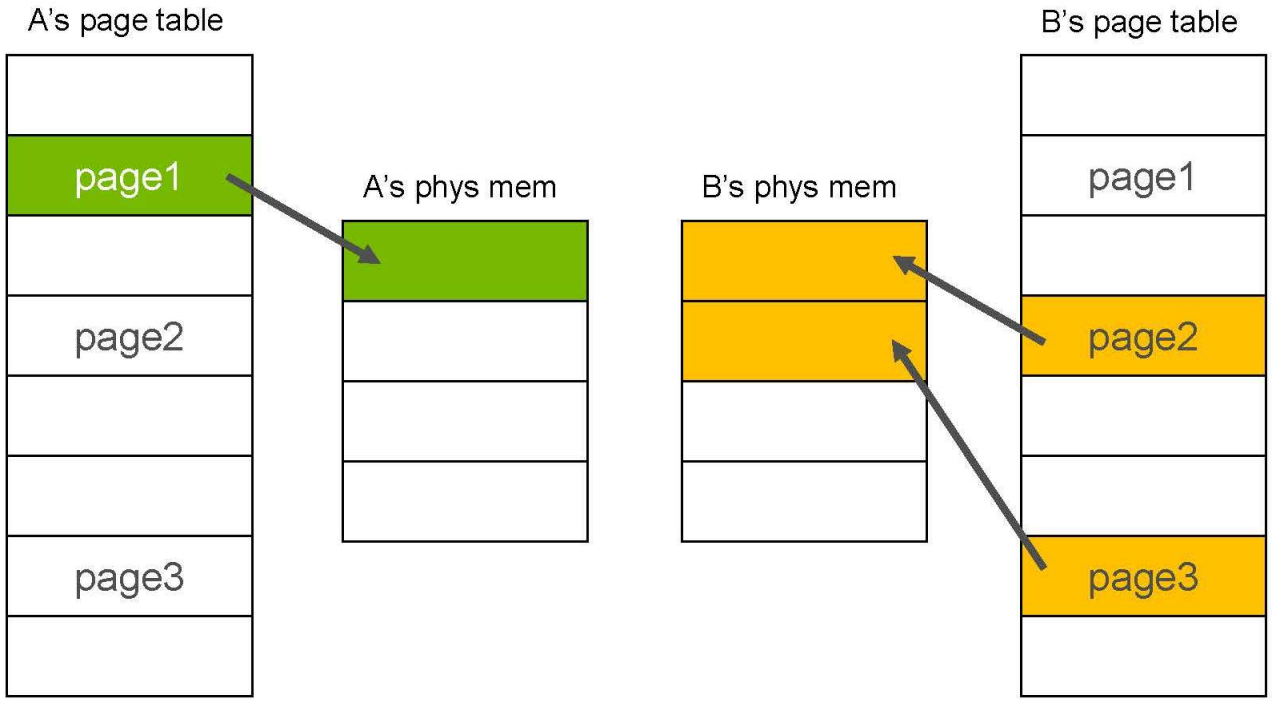
DL training with Unified Memory





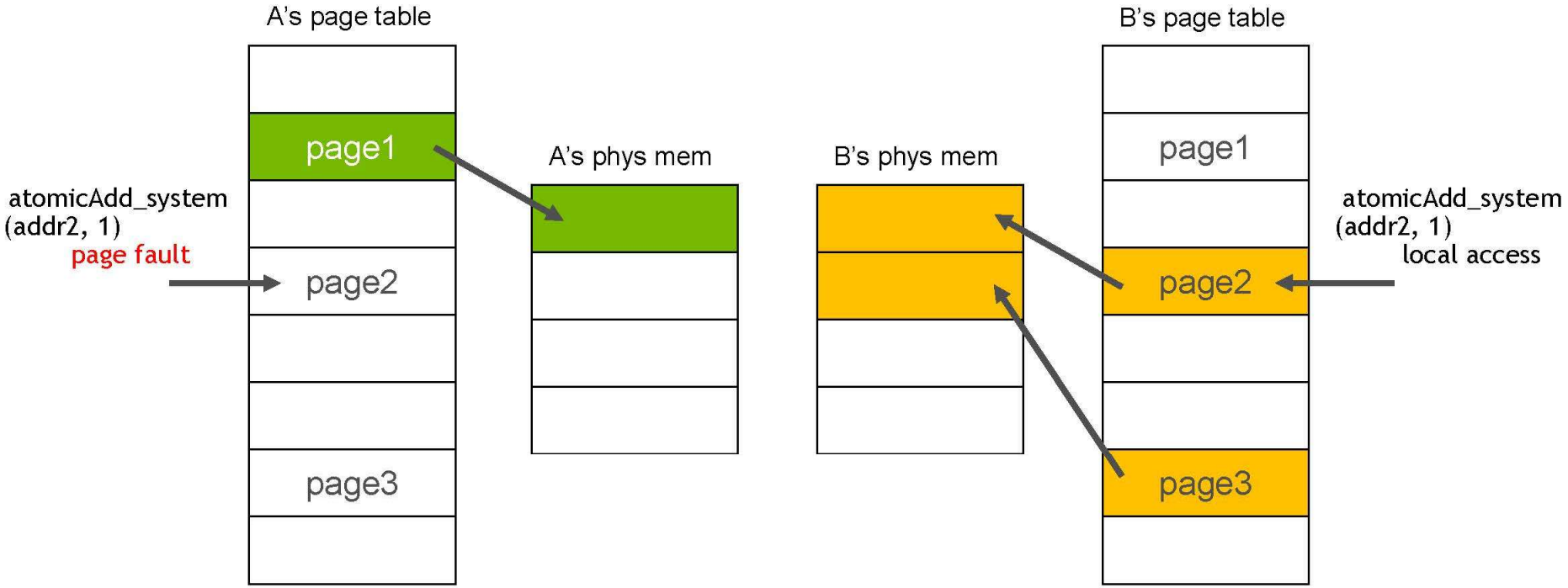
OC-Caffe will be released by the HiDL Team@OSU: hidl.cse.ohio-state.edu, mvapich.cse.ohio-state.edu

CONCURRENT ACCESS



CONCURRENT ACCESS

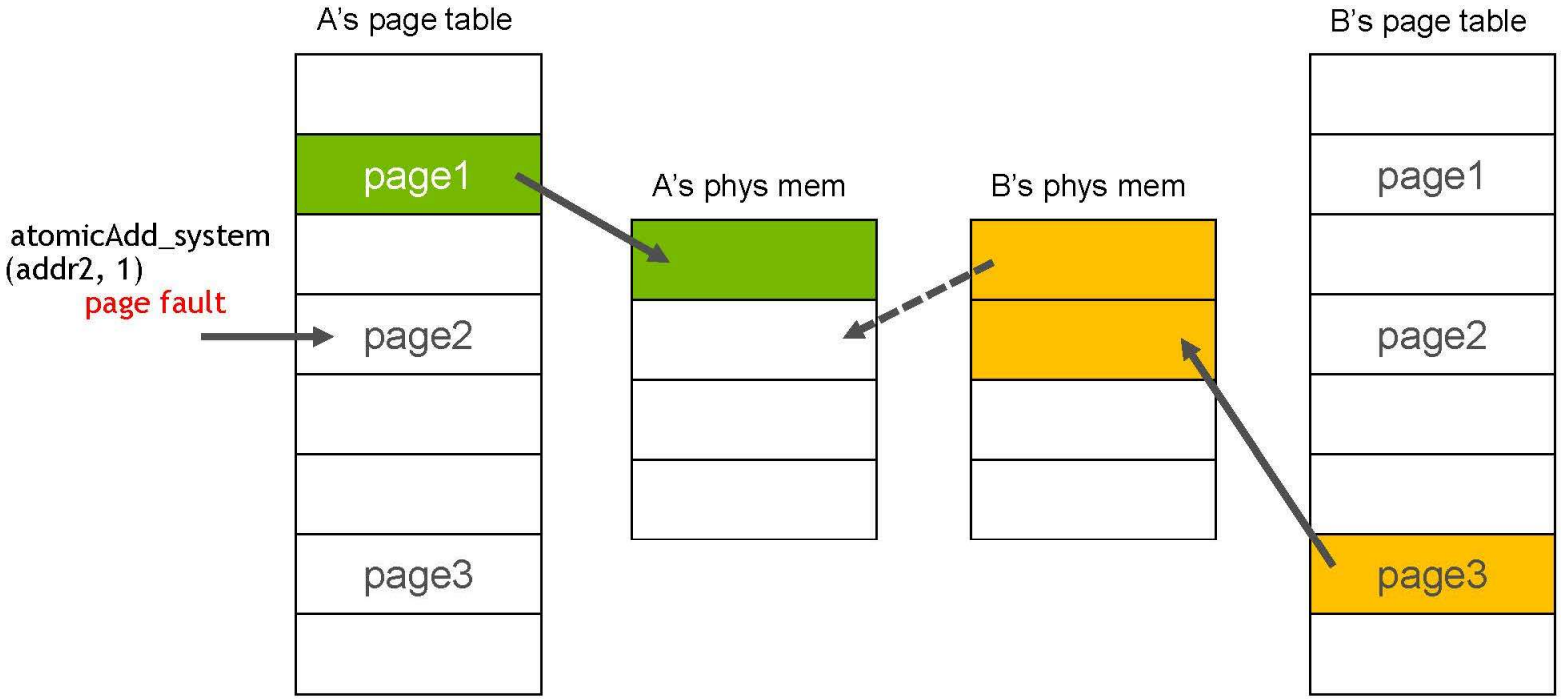
Exclusive Access*



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

CONCURRENT ACCESS

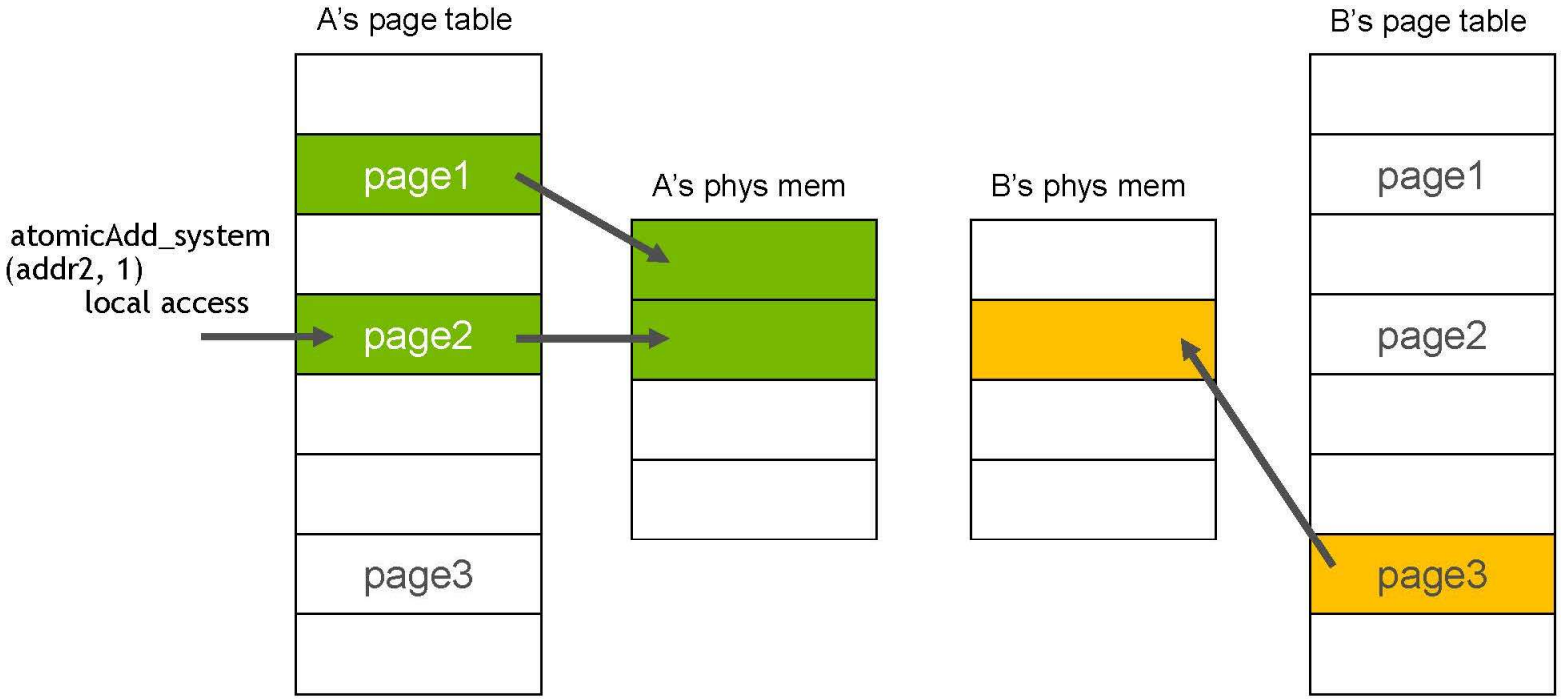
Exclusive Access



page2 unmapped in B's memory and migrated to A

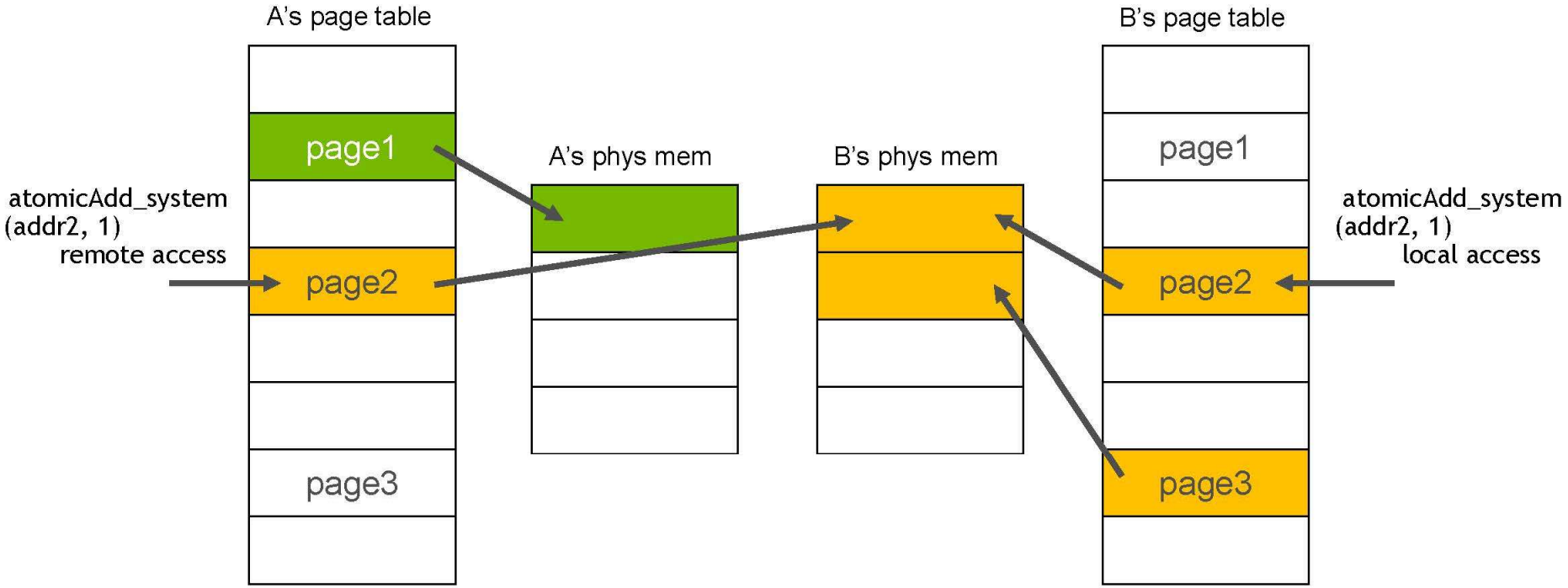
CONCURRENT ACCESS

Exclusive Access



CONCURRENT ACCESS

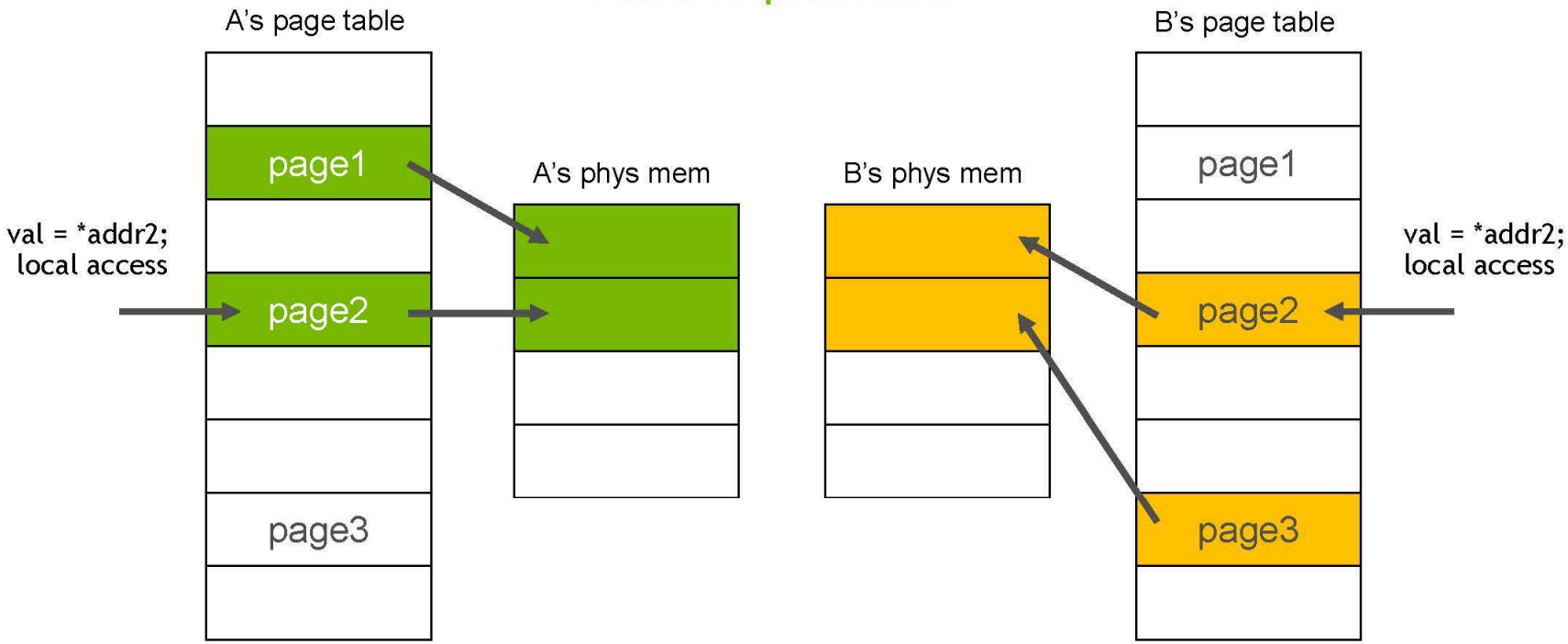
Atomics over NVLINK*



*both processors need to support atomic operations

CONCURRENT ACCESS

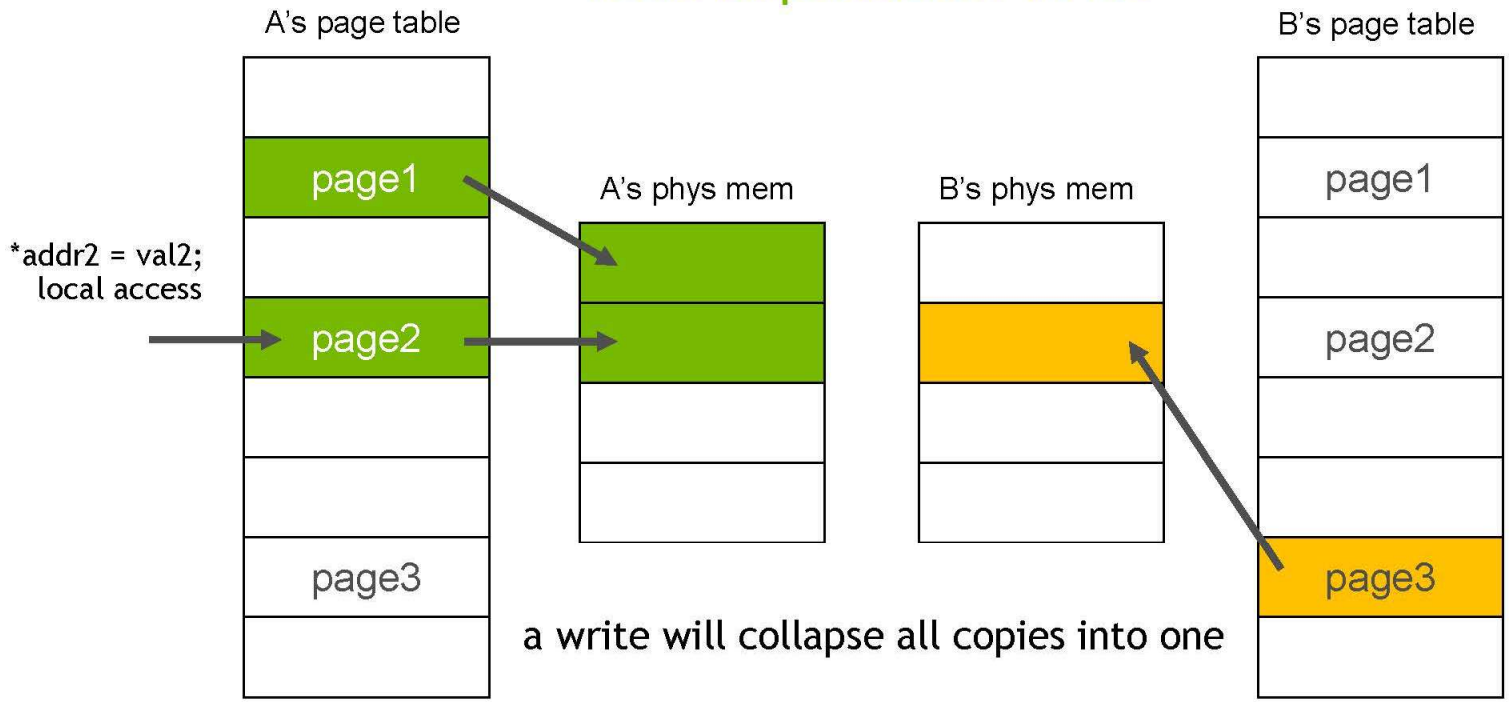
Read duplication*



*each processor must maintain its own page table

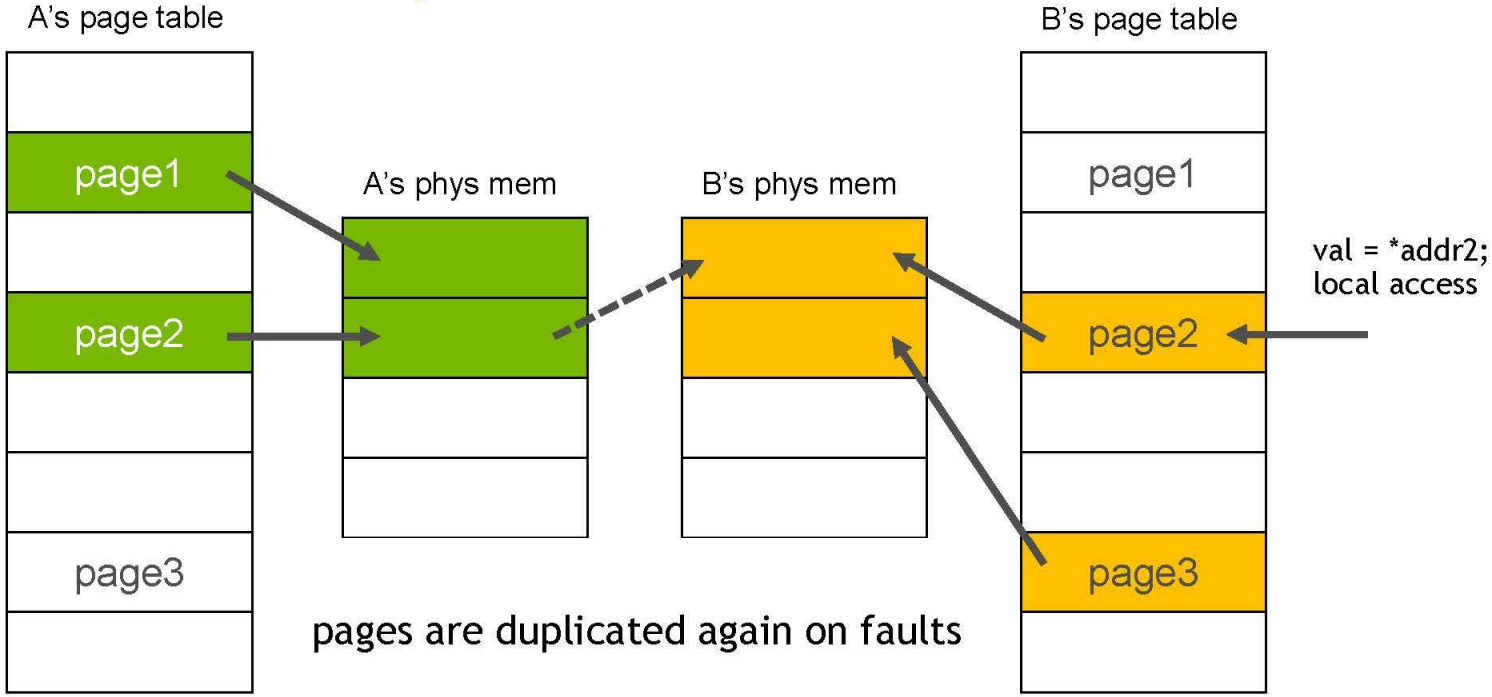
CONCURRENT ACCESS

Read duplication: write



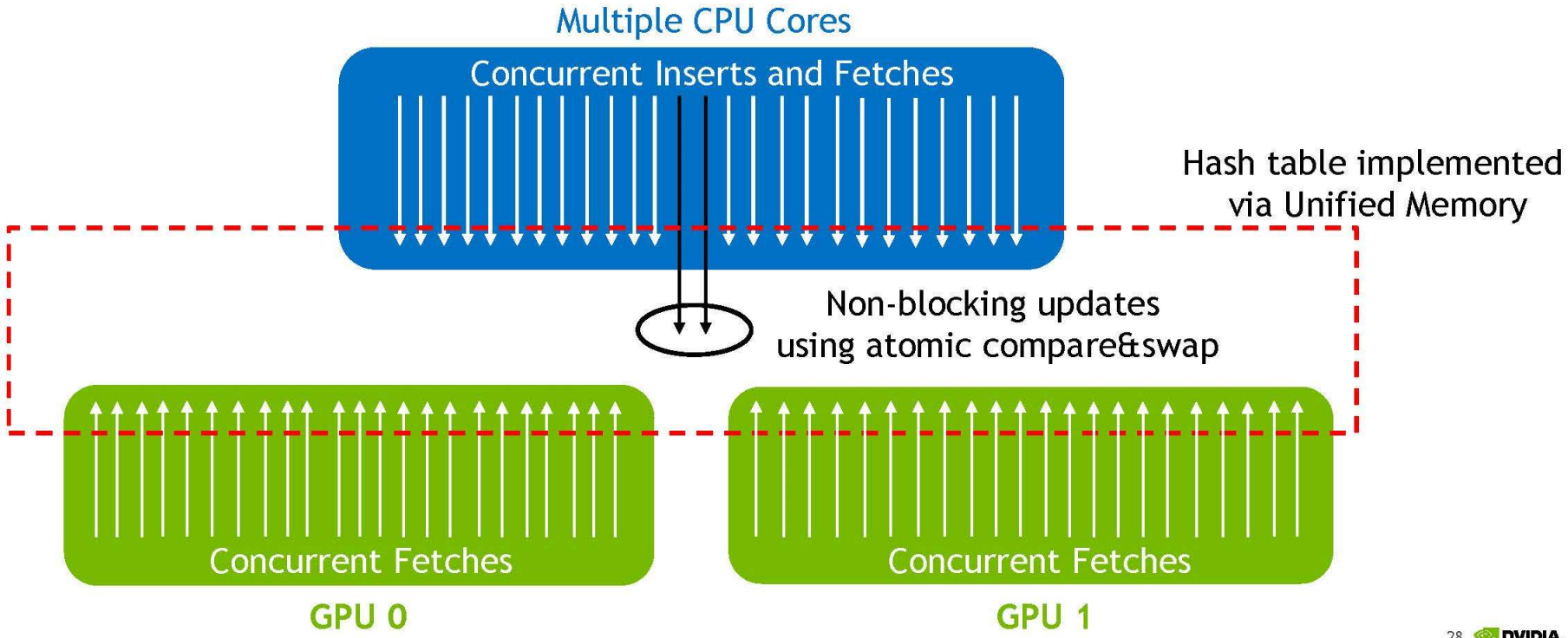
CONCURRENT ACCESS

Read duplication: read after write



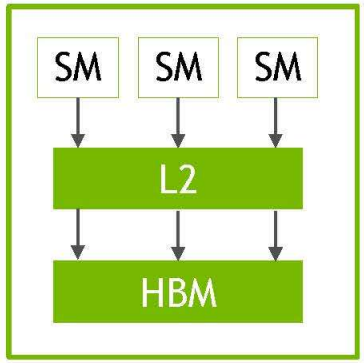
ANALYTICS USE CASE

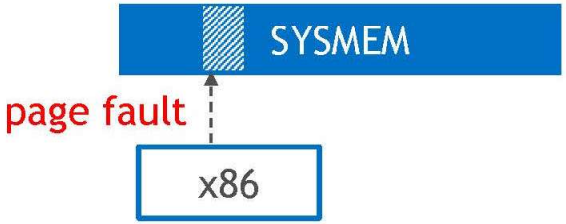
Design of a Concurrent Hybrid Hash Table



ANALYTICS USE CASE

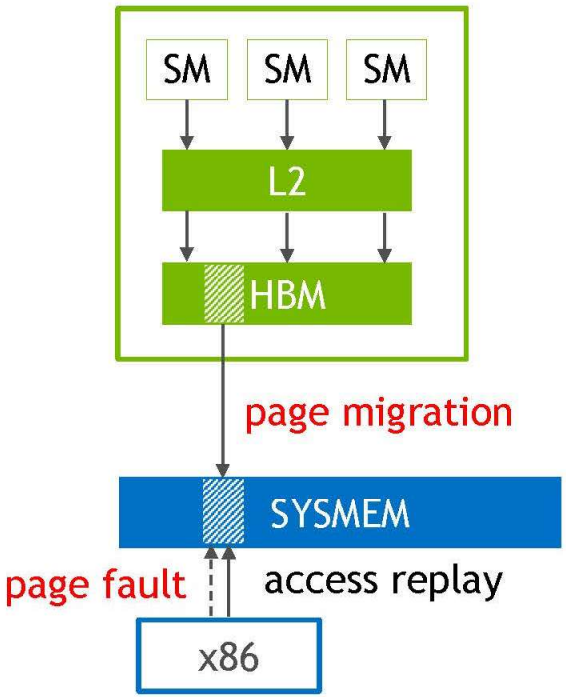
Concurrent Access To Hash Table





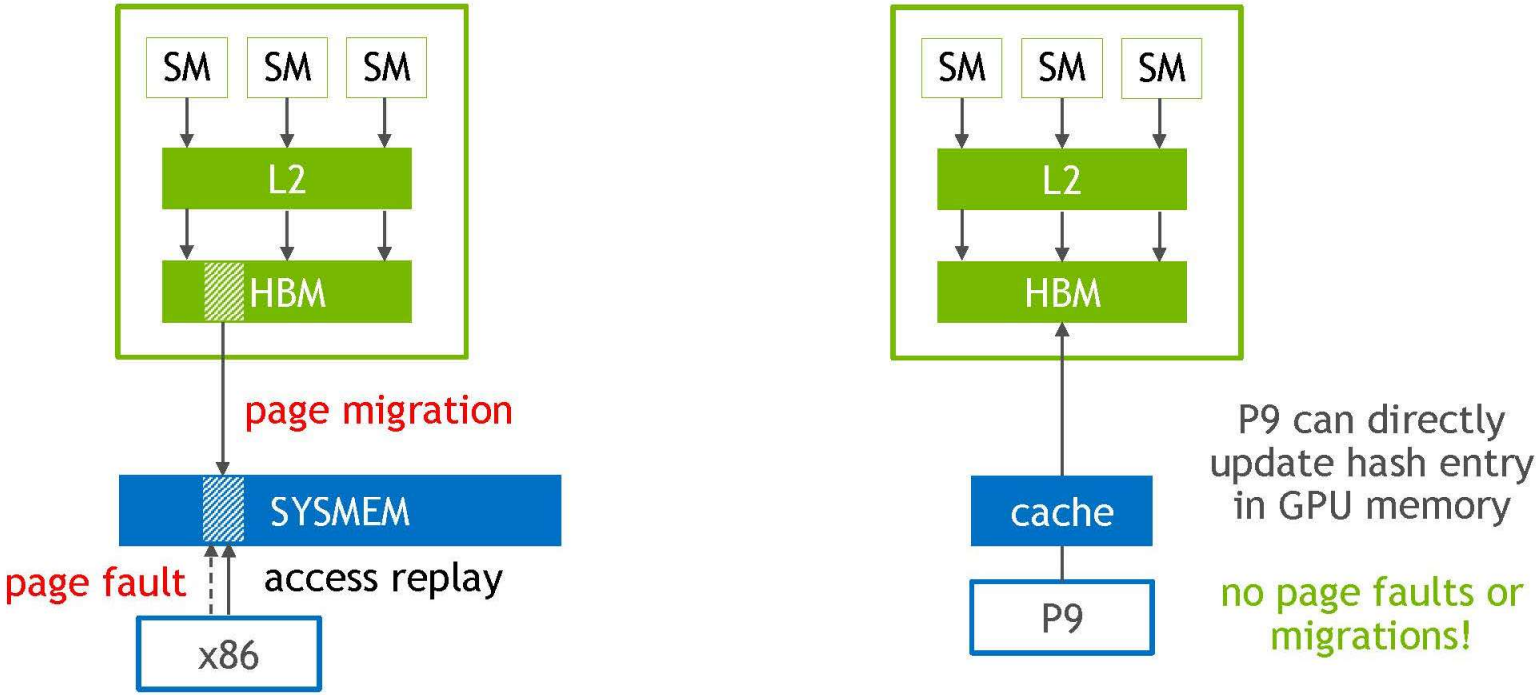
ANALYTICS USE CASE

Concurrent Access To Hash Table

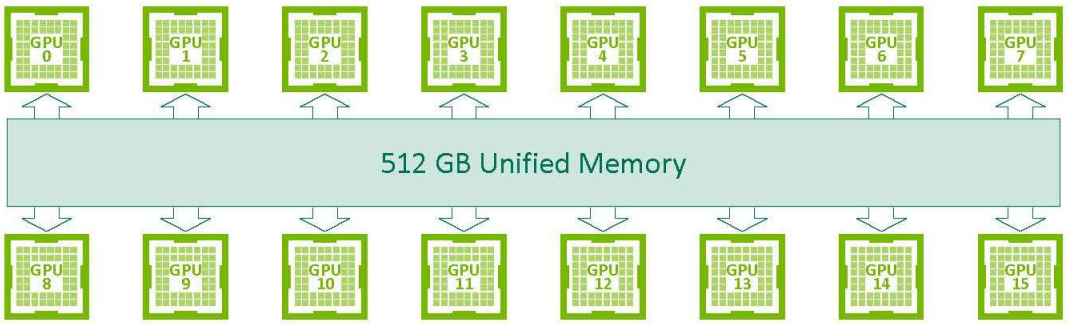


ANALYTICS USE CASE

Concurrent Access To Hash Table



UNIFIED MEMORY + DGX-2



UNIFIED MEMORY PROVIDES

Single memory view shared by all GPUs

Automatic migration of data between GPUs

User control of data locality

ENABLING MULTI-GPU

Single-GPU

```
__global__ void kernel(int *data) {
  int idx = threadIdx.x + blockDim.x * blockIdx.x;

  doSomeStuff(idx, data, ...);
}

cudaMallocManaged(&data, N * sizeof(int));
// initialize data on the CPU

kernel<<<grid, block>>>(data);
```

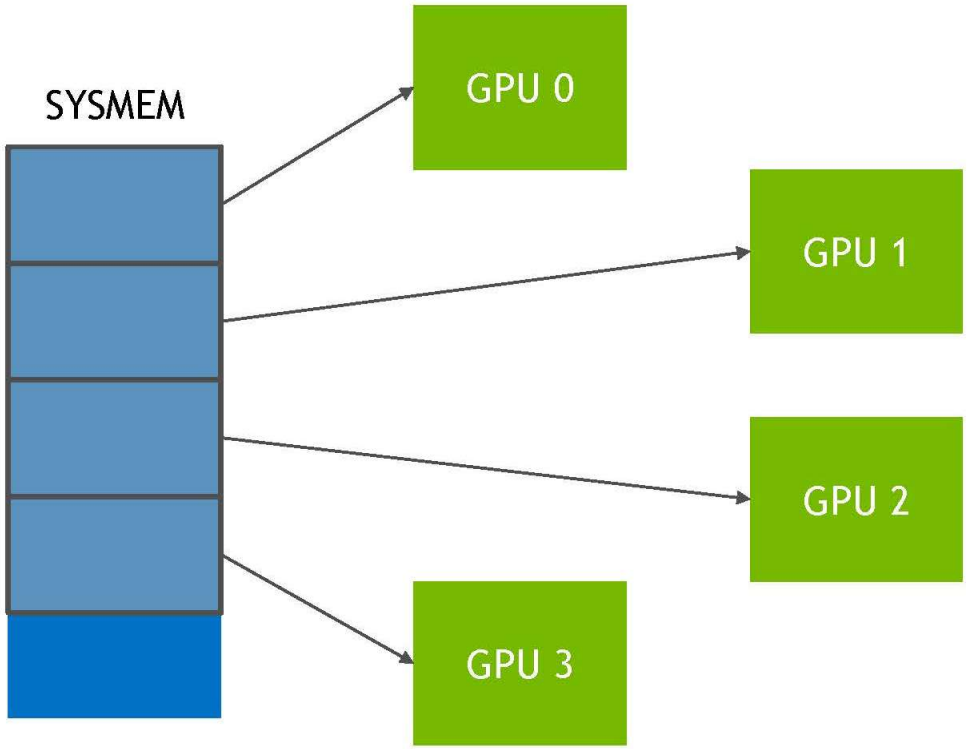
Multi-GPU

```
__global__ void kernel(int *data, int gpuId) {
  int idx = threadIdx.x + blockDim.x * (blockIdx.x
            + gpuId * gridDim.x);
  doSomeStuff(idx, data, ...);
}

cudaMallocManaged(&data, N * sizeof(int));
// initialize data on the CPU
for (int i = 0; i < numGPUs; i++) {
  cudaSetDevice(i);
  kernel<<<grid/numGPUs, block>>>(data, i);
}
```

update blockIdx.x

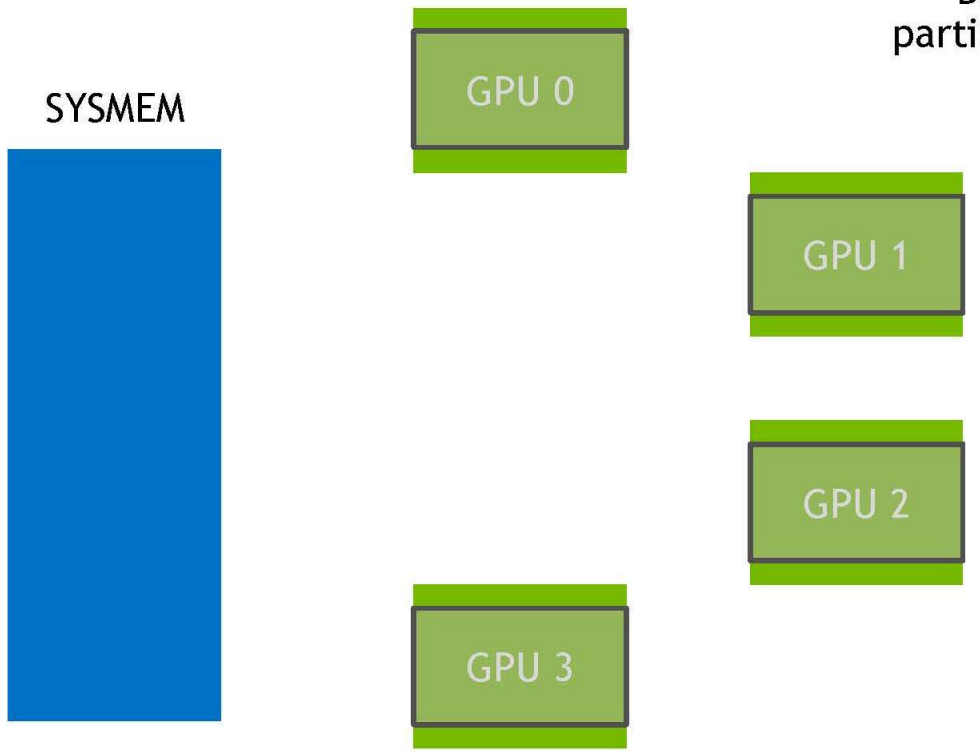
update launch config

MULTI-GPU WITH UNIFIED MEMORY



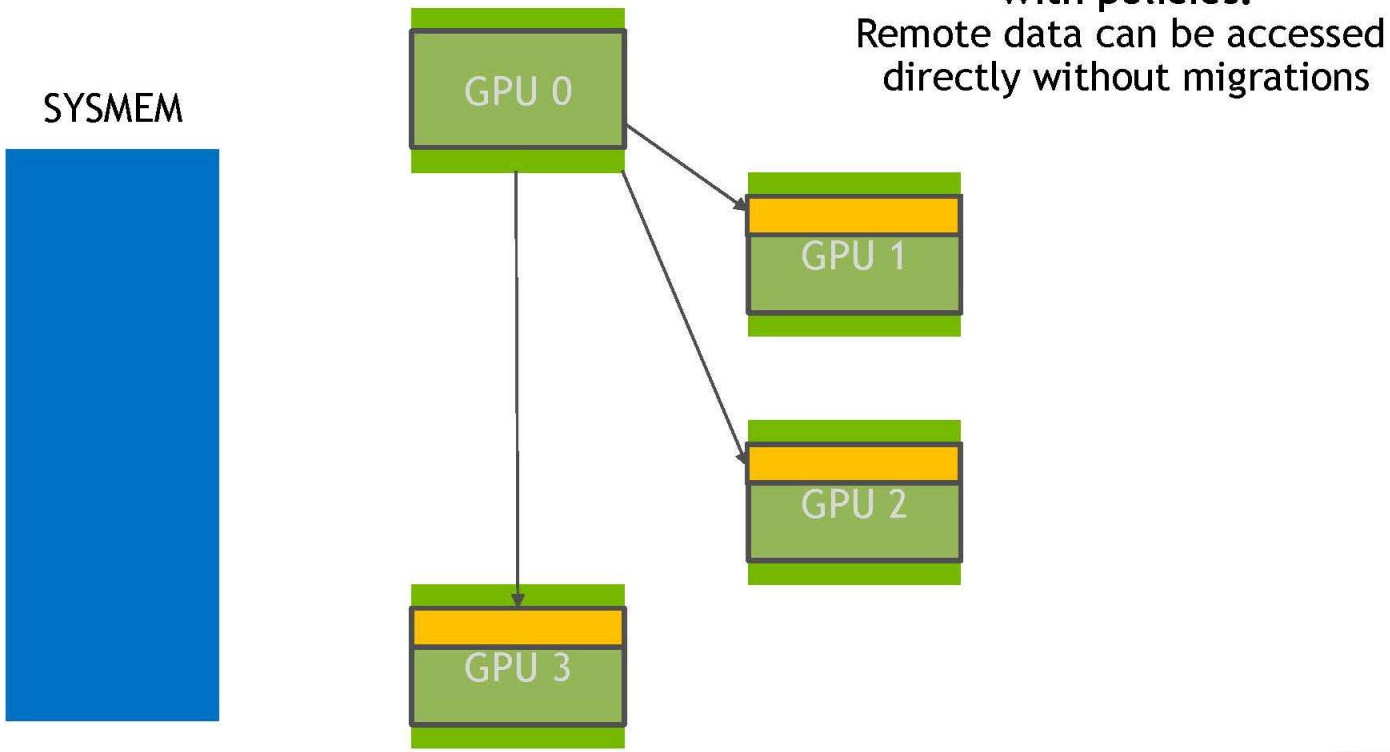
GPU kernels
initiate migrations

MULTI-GPU WITH UNIFIED MEMORY

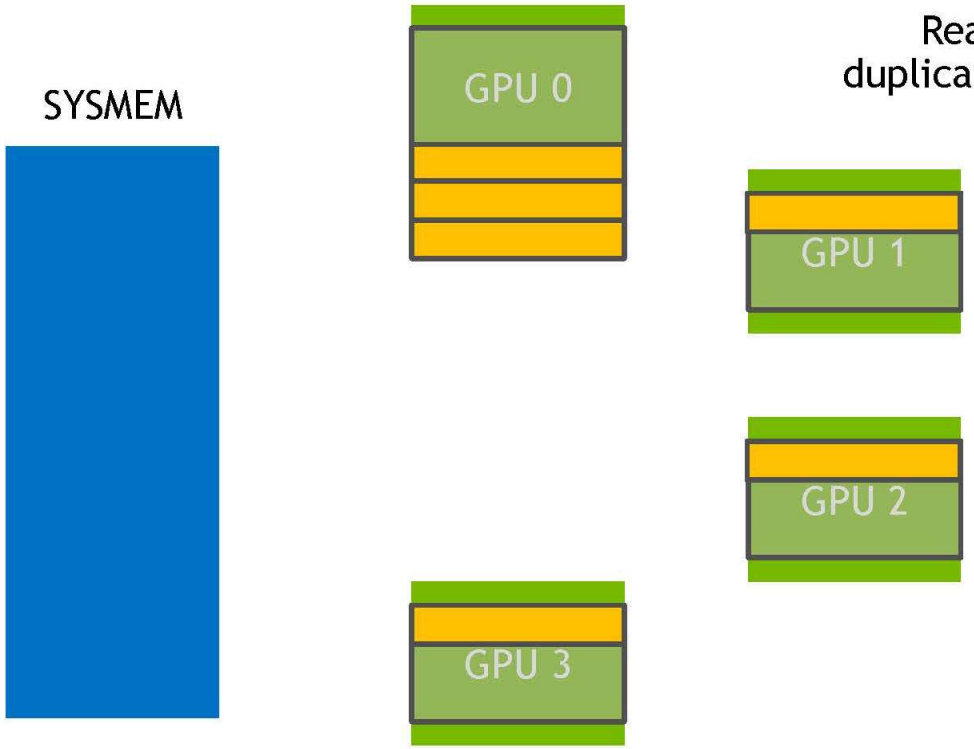


Data *automatically* partitioned between GPUs on first-touch

MULTI-GPU WITH UNIFIED MEMORY



MULTI-GPU WITH UNIFIED MEMORY

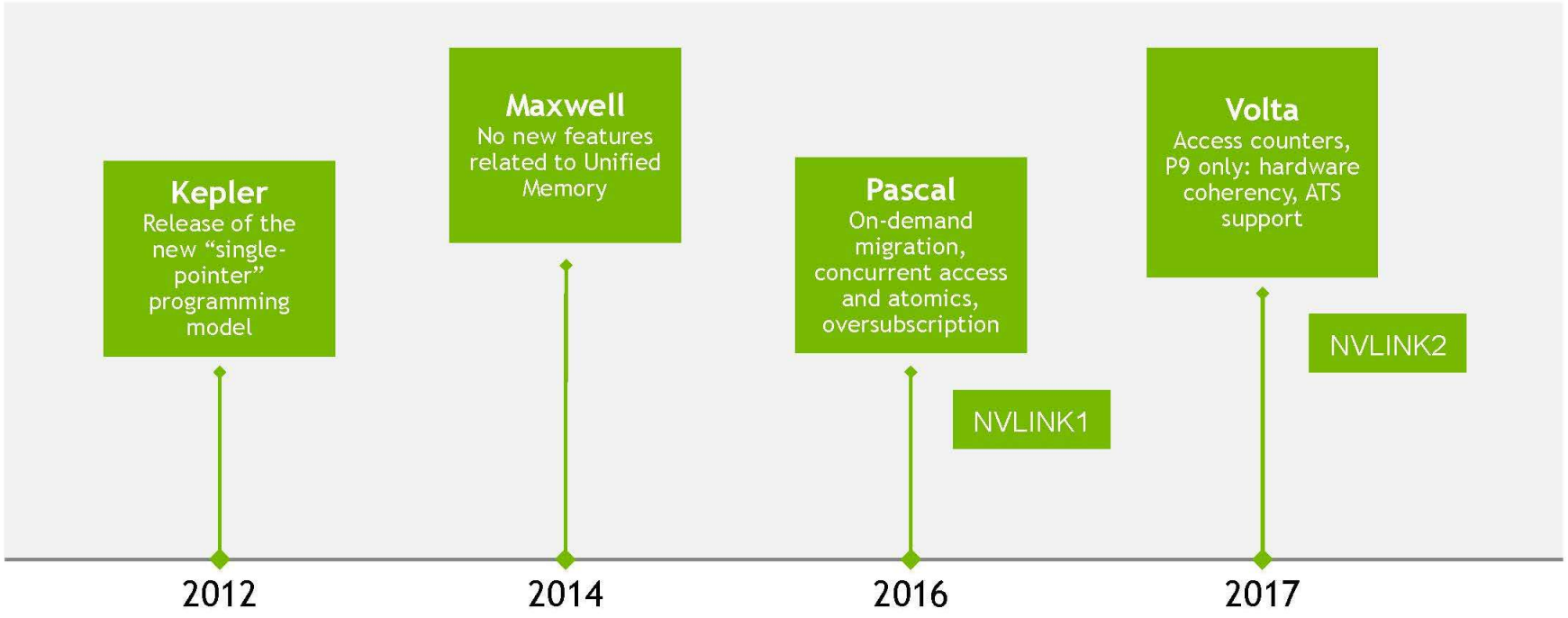


With policies:
Read-only data can be
duplicated and accessed locally

GPU ARCHITECTURE AND SOFTWARE EVOLUTION

UNIFIED MEMORY

Evolution of GPU architectures



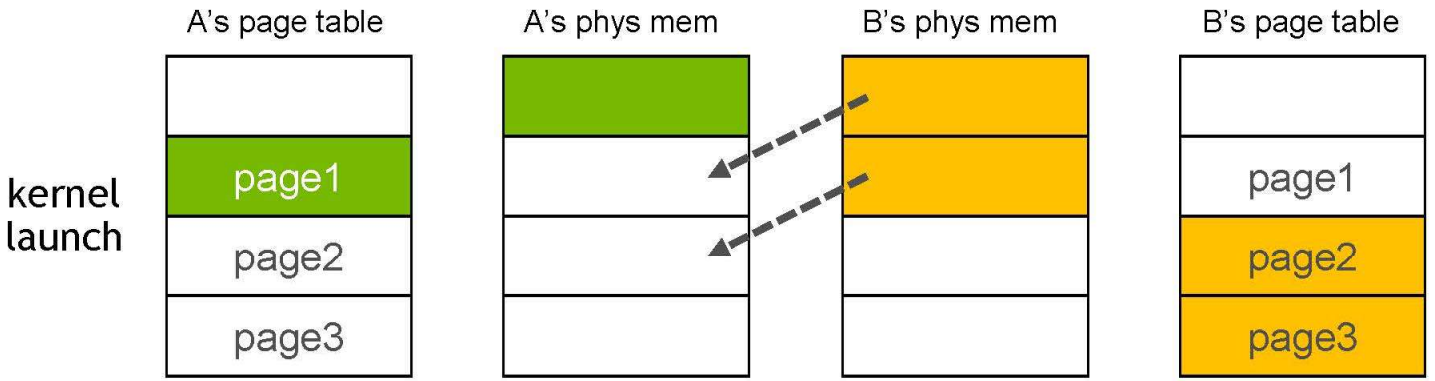
**Not all features are available on all platforms*

UNIFIED MEMORY: BEFORE PASCAL

Available since CUDA 6

No GPU page fault support: move **all dirty pages** on kernel launch

No concurrent access, no GPU memory oversubscription, no system-wide atomics

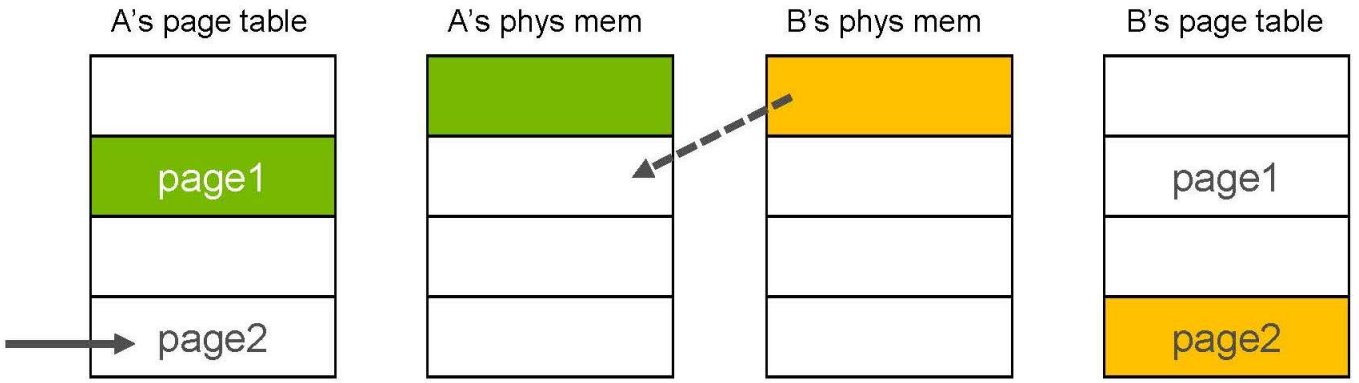


UNIFIED MEMORY: PASCAL AND VOLTA

Available since CUDA 8

GPU page fault support, concurrent access, extended VA space (48-bit)

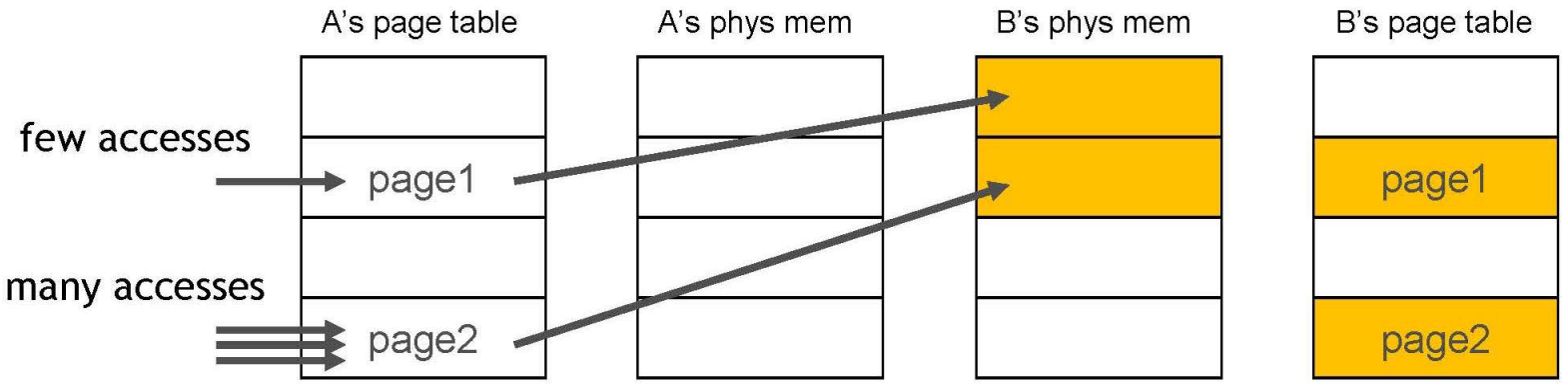
On-demand migration to accessing processor **on first touch**



UNIFIED MEMORY ON VOLTA+P9

New Feature: Access Counters

If memory is mapped to the GPU, migration can be triggered by access counters

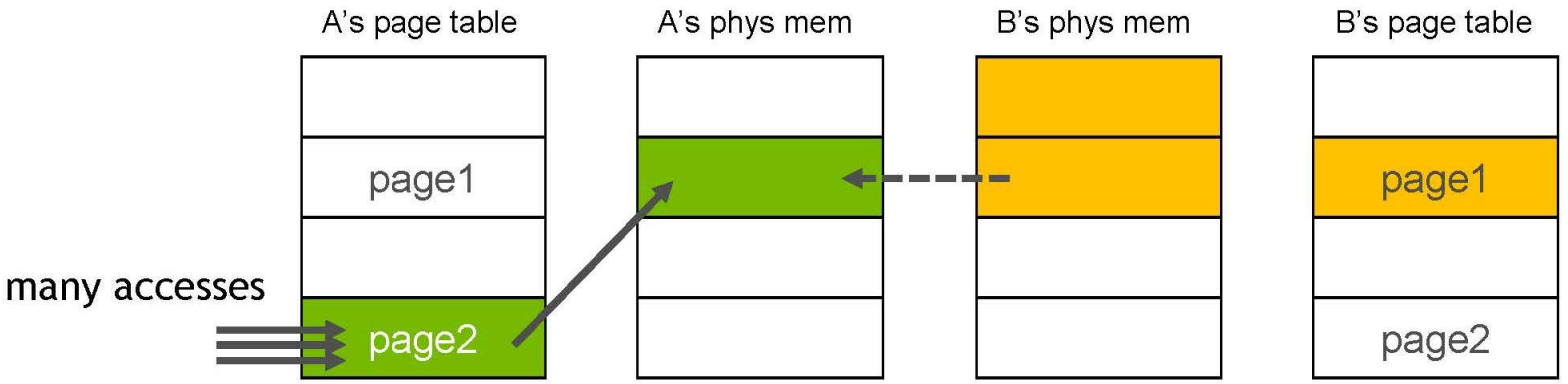


UNIFIED MEMORY ON VOLTA+P9

New Feature: Access Counters

With access counters **only hot pages** will be moved to the GPU

Migrations are *delayed* compared to the fault-based method



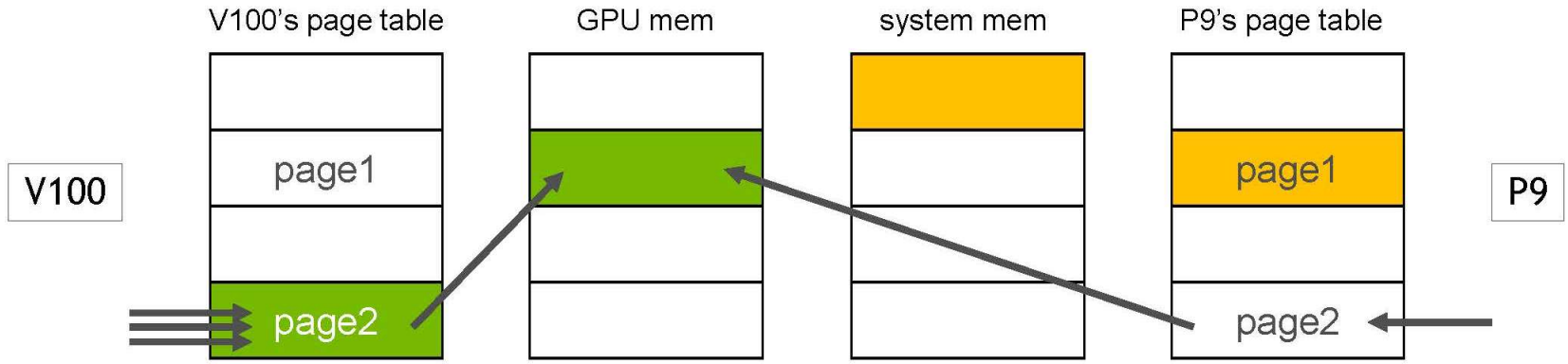
*When implemented this feature can be enabled with cudaMemAdvise policies

UNIFIED MEMORY ON VOLTA+P9

New Feature: Hardware Coherency with NVLINK2

CPU can directly access and *cache* GPU memory

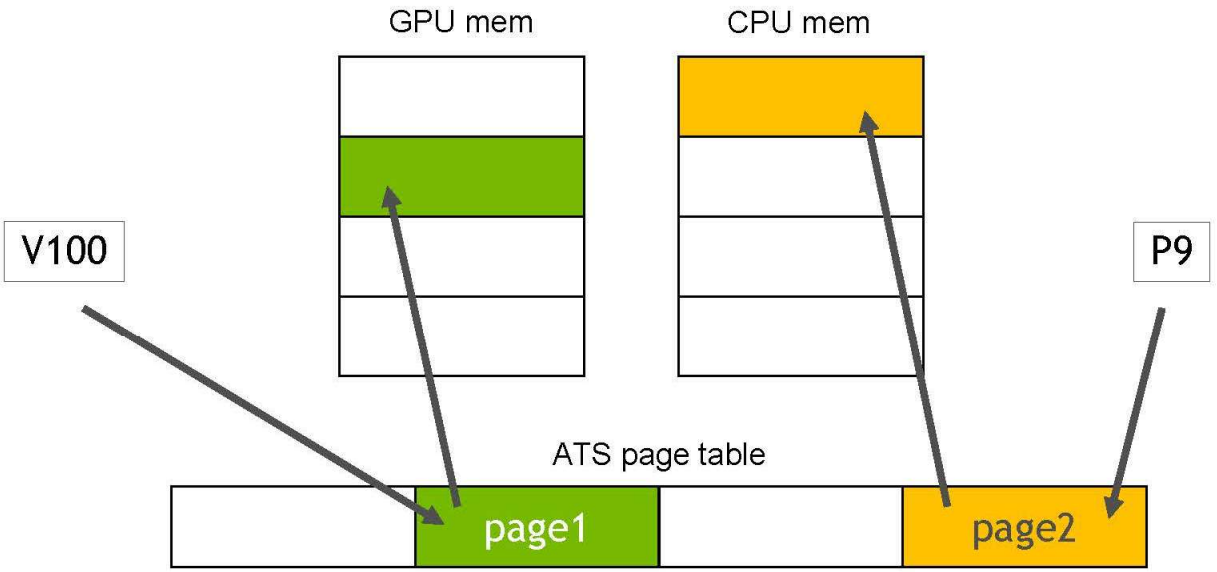
Native atomics support for all accessible memory



UNIFIED MEMORY ON VOLTA+P9

New Feature: ATS support

ATS: address translation service; CPU and GPU can share a *single* page table



UNIFIED MEMORY WITH SYSTEM ALLOCATOR

System allocator support allows GPU to access **all** system memory

malloc, stack, global, file system

P9: Address Translation Service (ATS)

Support enabled in CUDA 9.2

x86: Heterogeneous Memory Management (HMM)

Initial version of the patchset is integrated into 4.14 kernel

NVIDIA will be supporting upcoming versions of HMM

https://lkml.org/lkml/2017/6/23/443

WHAT YOU CAN DO WITH UNIFIED MEMORY

See it in action at the end of the talk!

Works everywhere today

```
int *data;
cudaMallocManaged(&data, sizeof(int) * n);
kernel<<<grid, block>>>(data);
```

Works on Power9 + ATS in CUDA 9.2
Will work in the future on x86 + HMM

```
int *data = (int*)malloc(sizeof(int) * n);
kernel<<<grid, block>>>(data);
```

```
int data[1024];
kernel<<<grid, block>>>(data);
```

```
int *data = (int*)alloca(sizeof(int) * n);
kernel<<<grid, block>>>(data);
```

```
extern int *data;
kernel<<<grid, block>>>(data);
```

UNIFIED MEMORY LANGUAGES

CUDA C/C++: **cudaMallocManaged**

CUDA Fortran: **managed** attribute (per allocation)

Python: **pycuda.driver.managed_empty** (allocate numpy.ndarray)

OpenACC: **-ta=managed** compiler option (all dynamic allocations)

UNIFIED MEMORY + OPENACC

Effortless way to run you code on GPUs

Literally adding a single line will get your code running on the GPU

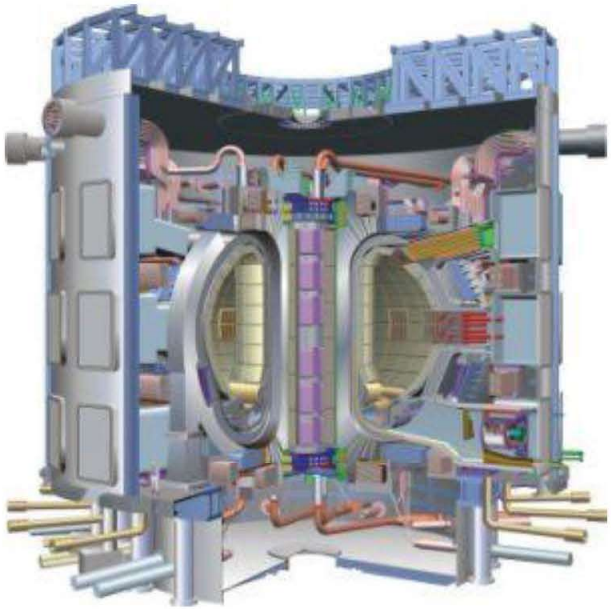
```
#pragma acc kernels
{
  for (i = 0; i < n; ++i) {
    c[i] = a[i] + b[i];
    ...
  }
}

...
```

Initiate parallel execution

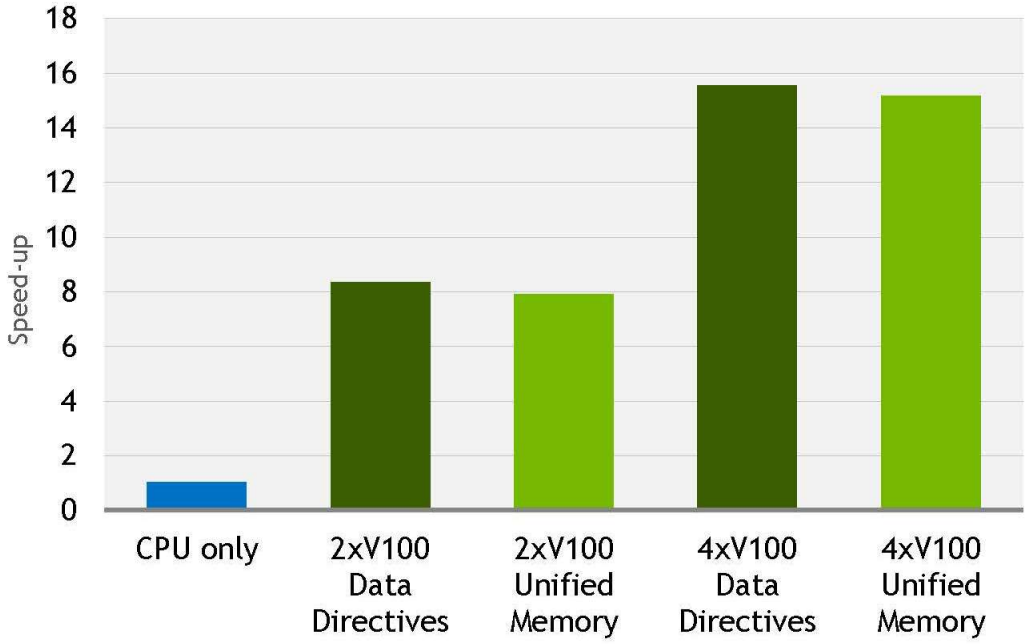
Easy to optimize later: add loop and data directives

GYROKINETIC TOROIDAL CODE

Particle-In-Cell production code



http://phoenix.ps.uci.edu/gtc_group



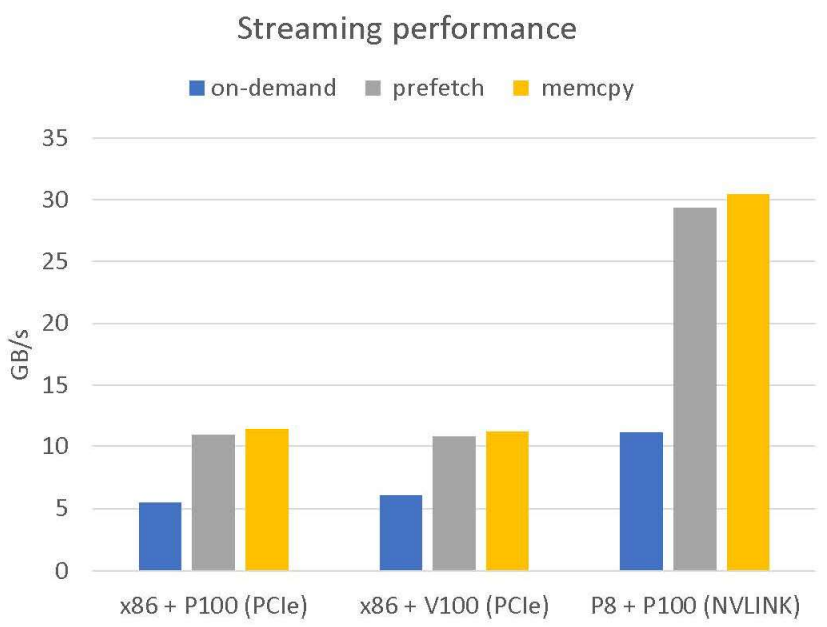
CPU: Haswell E5-2698 v3 @ 2.30GHz, dual socket 16-core; MPI: MVAPICH-GDR

PERFORMANCE DEEP DIVE

STREAMING BENCHMARK

How fast is on-demand migration?

```
__global__ void kernel(int *host, int *device) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  device[i] = host[i];
}

// allocate and initialize memory
cudaMallocManaged(&host, size);
cudaMalloc(&device, size);
memset(host, 0, size);

// benchmark CPU->GPU migration
if (prefetch)
  cudaMemPrefetchAsync(host, size, gpuId);
kernel<<<grid, block>>>(host, device);
```



UNDERSTANDING PROFILER OUTPUT

```
==14487== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  23.270ms         1  23.270ms  23.270ms  23.270ms  void kernel(int*, int*)
      API calls:   79.56%  23.272ms         1  23.272ms  23.272ms  23.272ms  cudaDeviceSynchronize
                   20.42%  5.9732ms         1  5.9732ms  5.9732ms  5.9732ms  cudaLaunch
                    0.01%  2.0490us         1  2.0490us  2.0490us  2.0490us  cudaConfigureCall
                    0.01%  1.8360us         4     459ns     138ns     833ns  cudaSetupArgument

==14487== Unified Memory profiling result:
Device "Tesla V100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    3012  21.758KB  4.0000KB  952.00KB  64.00000MB  13.49043ms  Host To Device
      81         -         -         -           -  23.23181ms  Gpu page fault groups
```

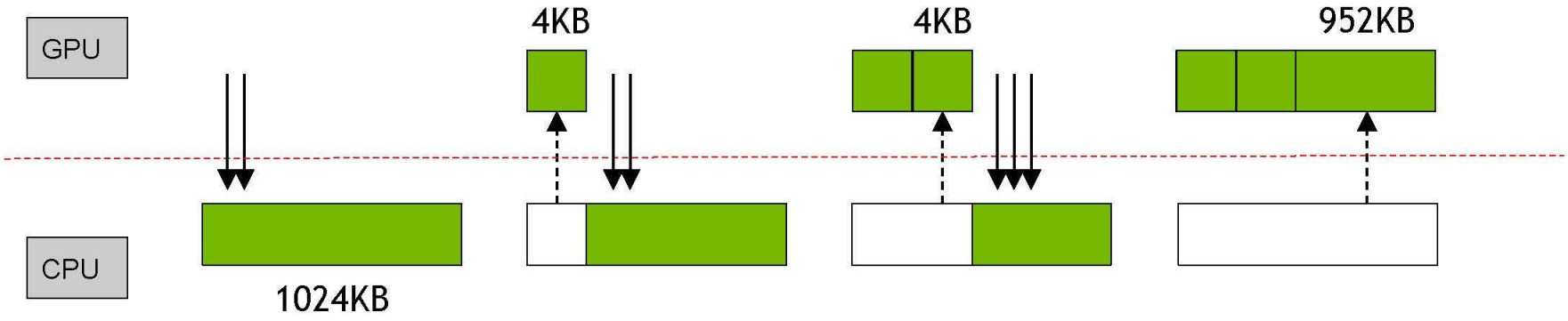
HEURISTIC PREFETCHING

Do Not Confuse with API-prefetching

GPU architecture supports different page sizes

Contiguous pages up to a large page size are promoted to the larger size

Driver prefetches whole regions if pages are accessed *densely*



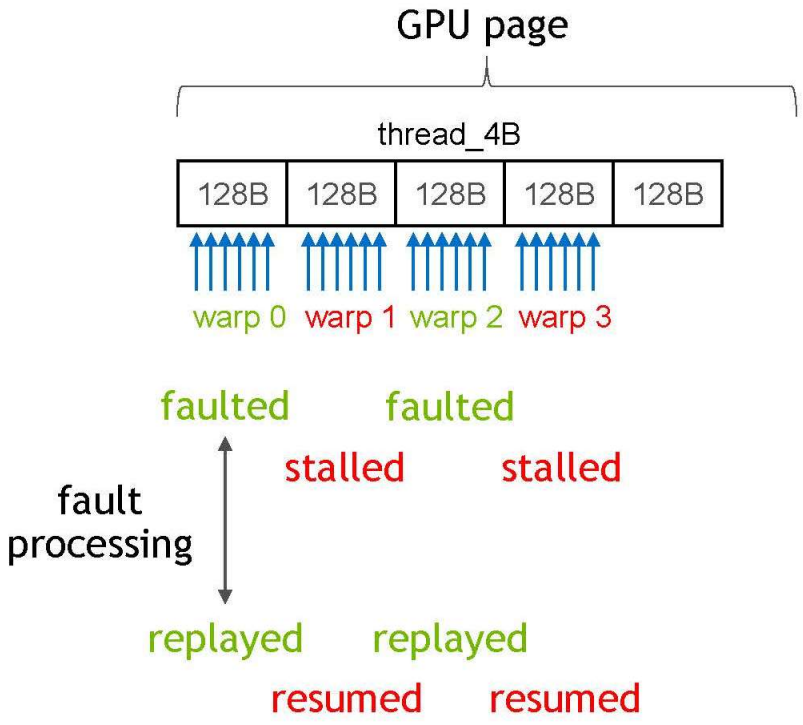
WHAT IS PAGE FAULT GROUPS?

```
==14487== Unified Memory profiling result:
Device "Tesla V100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    3012  21.758KB  4.0000KB  952.00KB  64.00000MB  13.49043ms  Host To Device
      81         -         -         -           -  23.23181ms  Gpu page fault groups
```

nvprof --print-gpu-trace ...

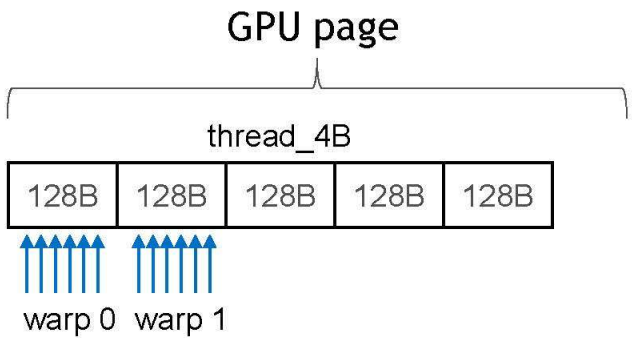
```
Unified Memory  Virtual Address  Name
             8     0x3900010000  [Unified Memory GPU page faults]
             9     0x3900040000  [Unified Memory GPU page faults]
             5     0x3900108000  [Unified Memory GPU page faults]
             5     0x3900200000  [Unified Memory GPU page faults]
```

PAGE FAULTS HANDLING

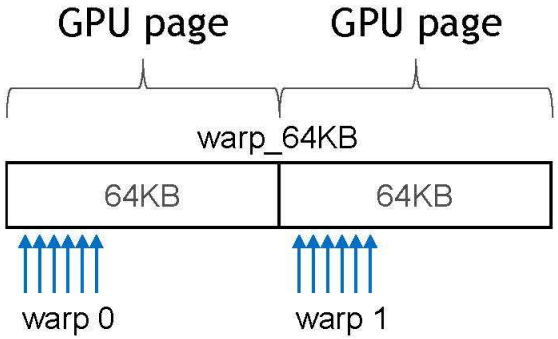


OPTIMIZING ON-DEMAND MIGRATION

Increase fault concurrency to reduce page fault stalls



multiple faults per page
warps are stalled on fault processing



fewer warps are stalled
"spread-out" pattern improves prefetching

OPTIMIZING ON-DEMAND MIGRATION

Thread/4B

```
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    3012  21.758KB  4.0000KB  952.00KB  64.00000MB  13.49043ms  Host To Device
      81         -         -         -           -  23.23181ms  Gpu page fault groups

Unified Memory  Virtual Address  Name
             8     0x3900010000  [Unified Memory GPU page faults]
             9     0x3900040000  [Unified Memory GPU page faults]
             5     0x3900108000  [Unified Memory GPU page faults]
```

Warp/64KB

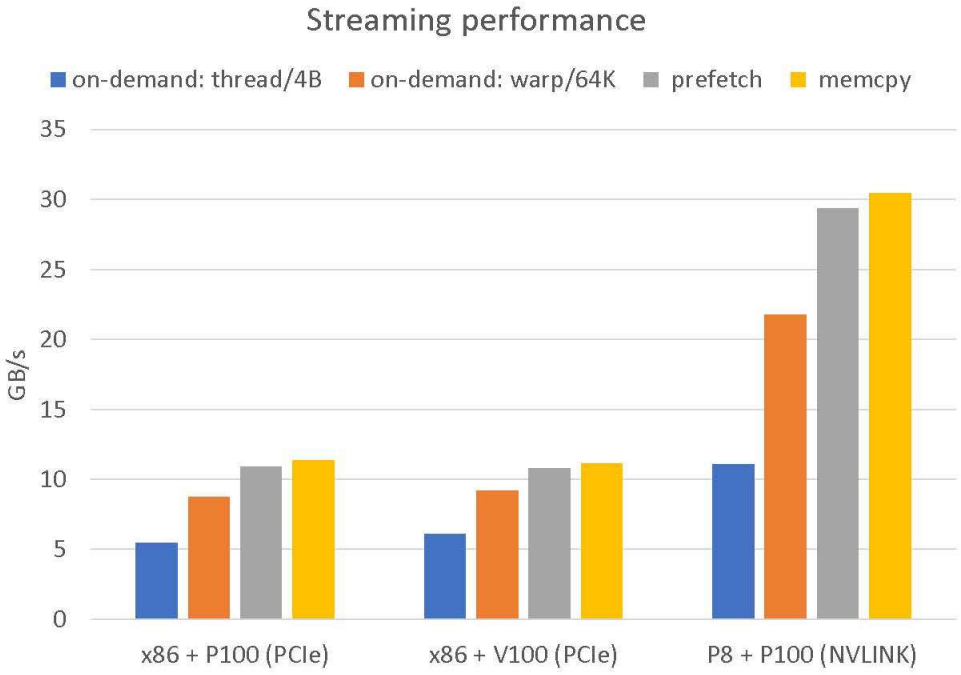
```
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
     957  68.481KB  4.0000KB  576.00KB  64.00000MB  8.242080ms  Host To Device
       6         -         -         -           -  9.769984ms  Gpu page fault groups

Unified Memory  Virtual Address  Name
             1     0x39000d0000  [Unified Memory GPU page faults]
             1     0x39000c0000  [Unified Memory GPU page faults]
             1     0x3900080000  [Unified Memory GPU page faults]
```

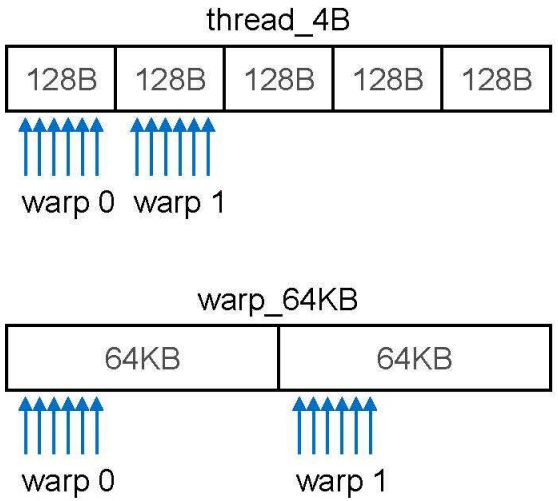
more efficient prefetching

fewer stalls

STREAMING BENCHMARK

How fast is on-demand migration?

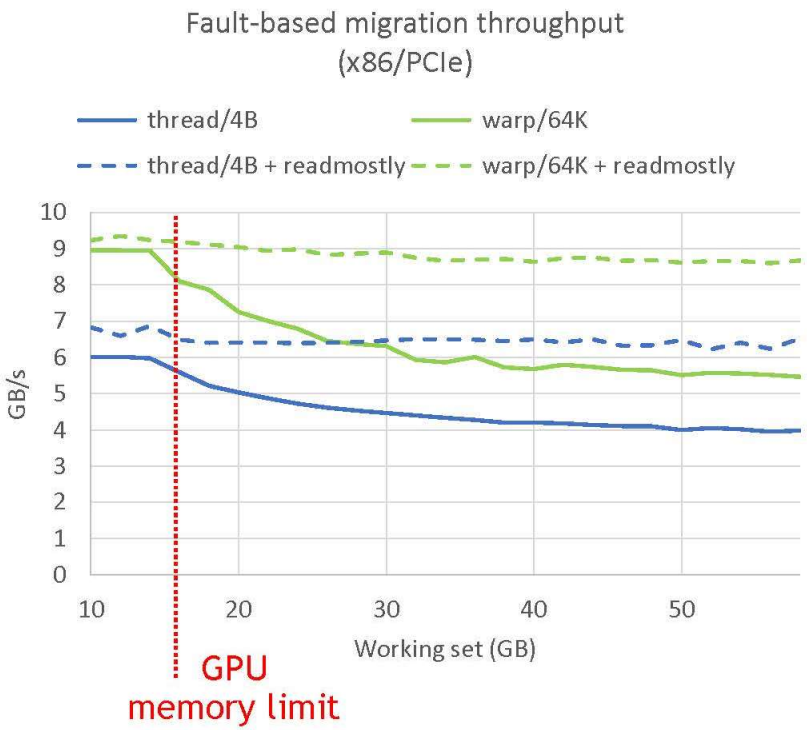


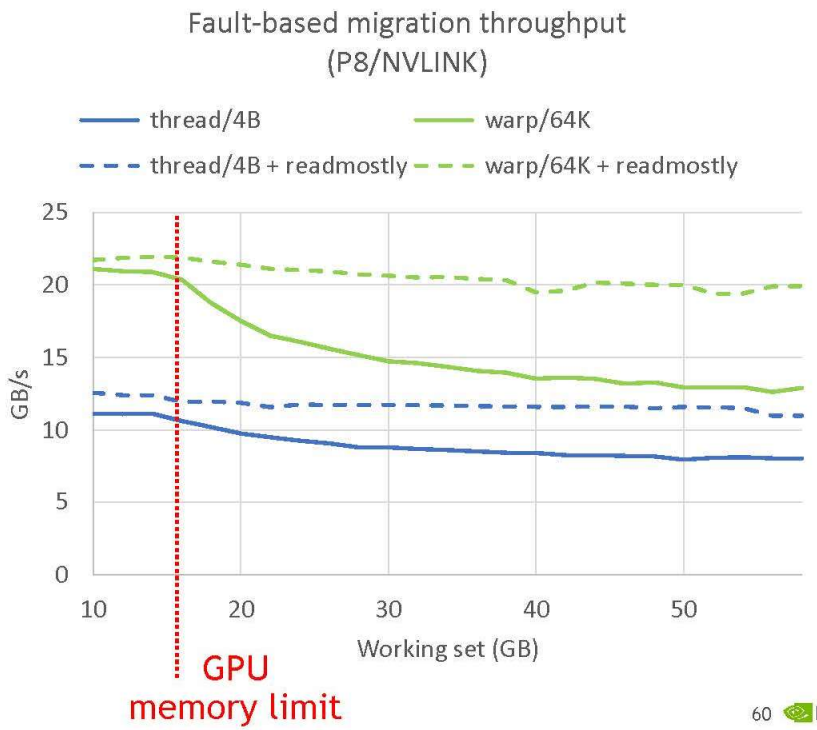


Also check the Parallel Forall blog: https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/

GPU MEMORY OVERSUBSCRIPTION

Let's see how perf changes as we increase the working set





EVICTION ALGORITHM

What Pages Are Moving Out of the GPU

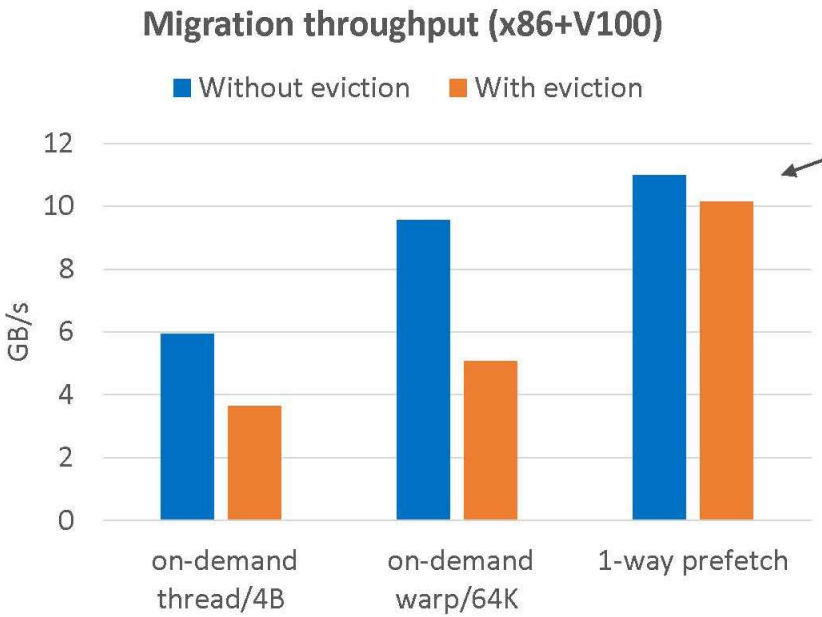
Driver keeps a list of physical chunks of GPU memory

Chunks from the front of the list are evicted first (LRU)

A chunk is considered “in use” when it is fully-populated or migrated

Prefetching and policies may impact eviction heuristic in the future

PREFETCHING AND EVICTIONS

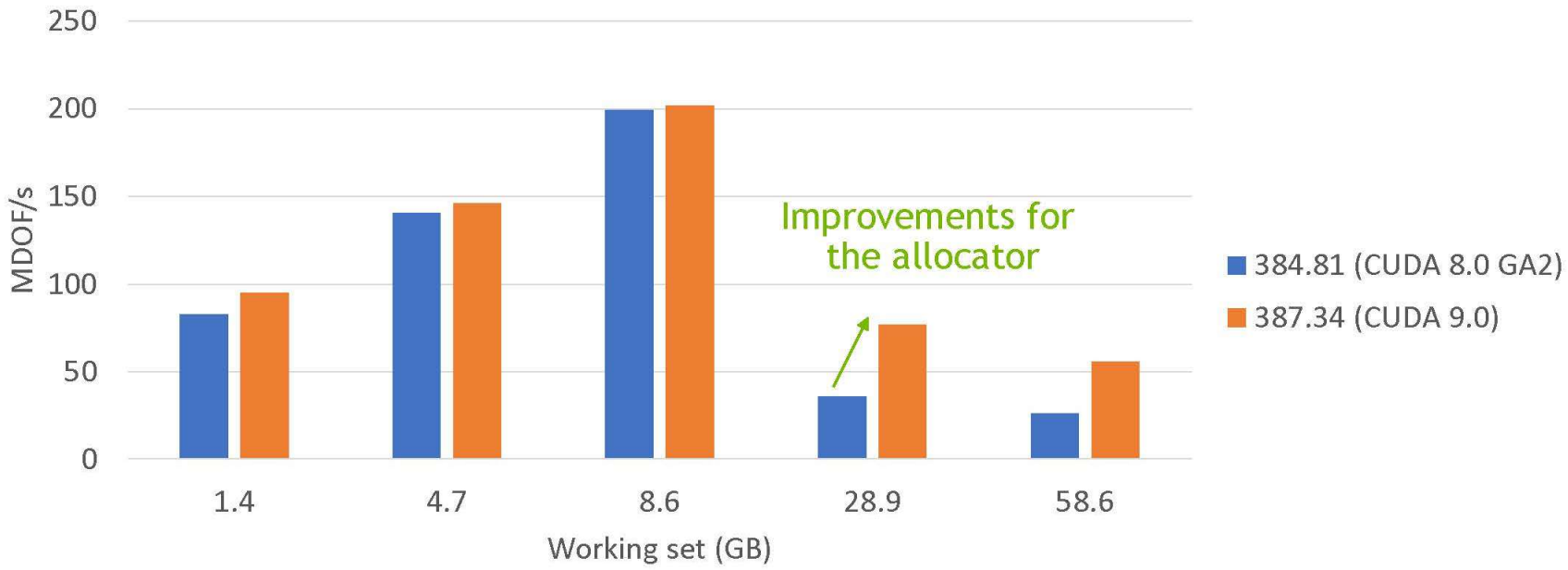


Prefetch can be overlapped with evictions
without using CUDA streams!
(enabled in CUDA 9.1)

cudaMemcpy solution requires
scheduling DtoH and HtoD copies
into two separate streams

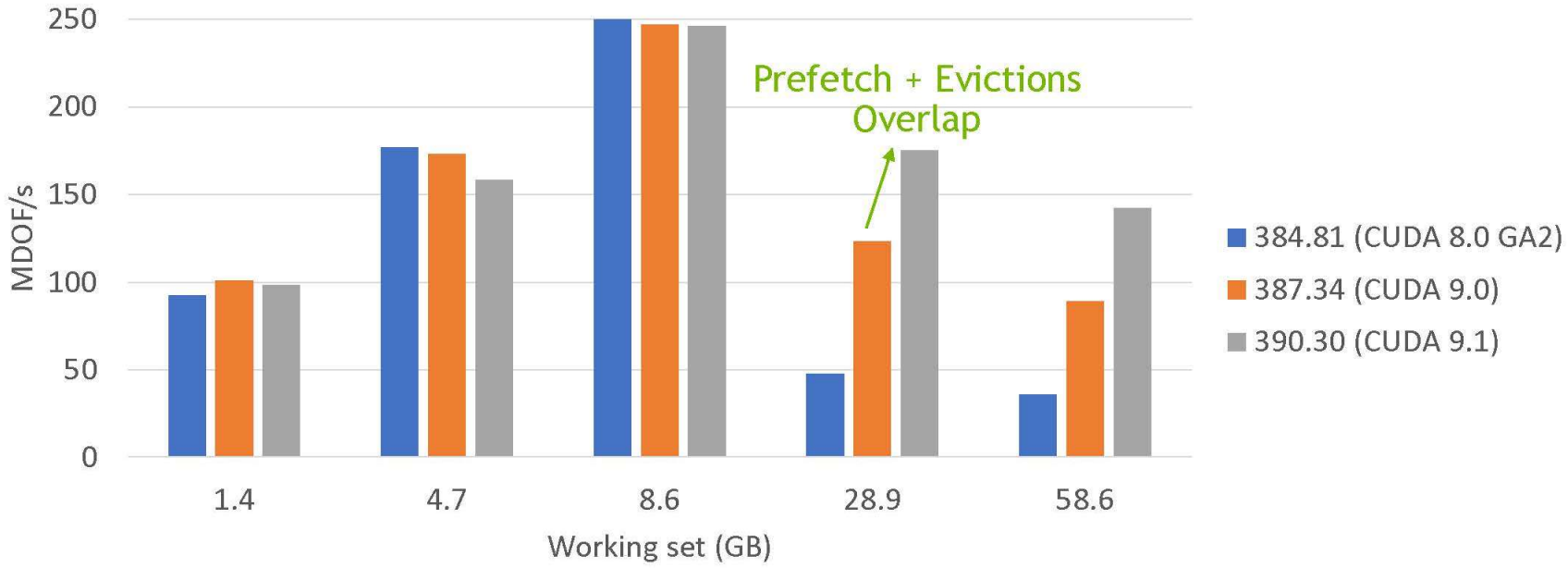
DRIVER ENHANCEMENTS

Tesla V100 + x86: HPGMG-AMR default (no hints)



DRIVER ENHANCEMENTS

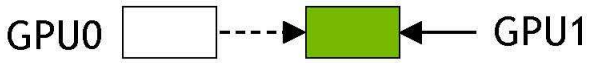
Tesla V100 + x86: HPGMG-AMR optimized (prefetches)



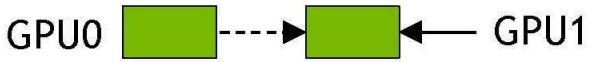
LOCALITY AND ACCESS CONTROL

cudaMemAdvise

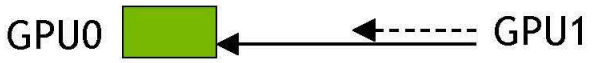
Default: data *migrates* on first touch



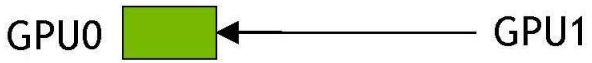
ReadMostly: data *duplicated* on first touch



PreferredLocation: *resist* migrating away from the preferred location



AccessedBy: establish *direct mapping* and avoid faults



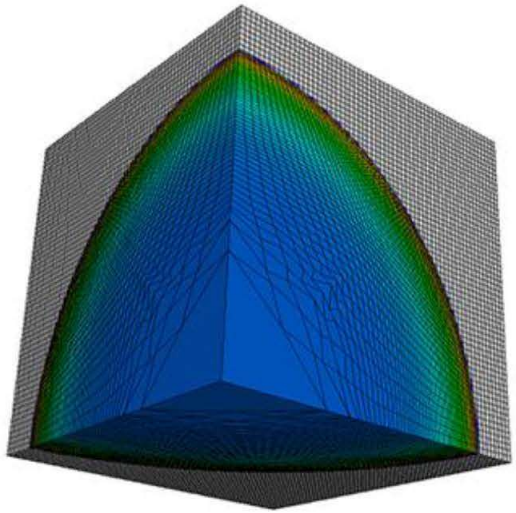
LULESH

CORAL OpenACC app

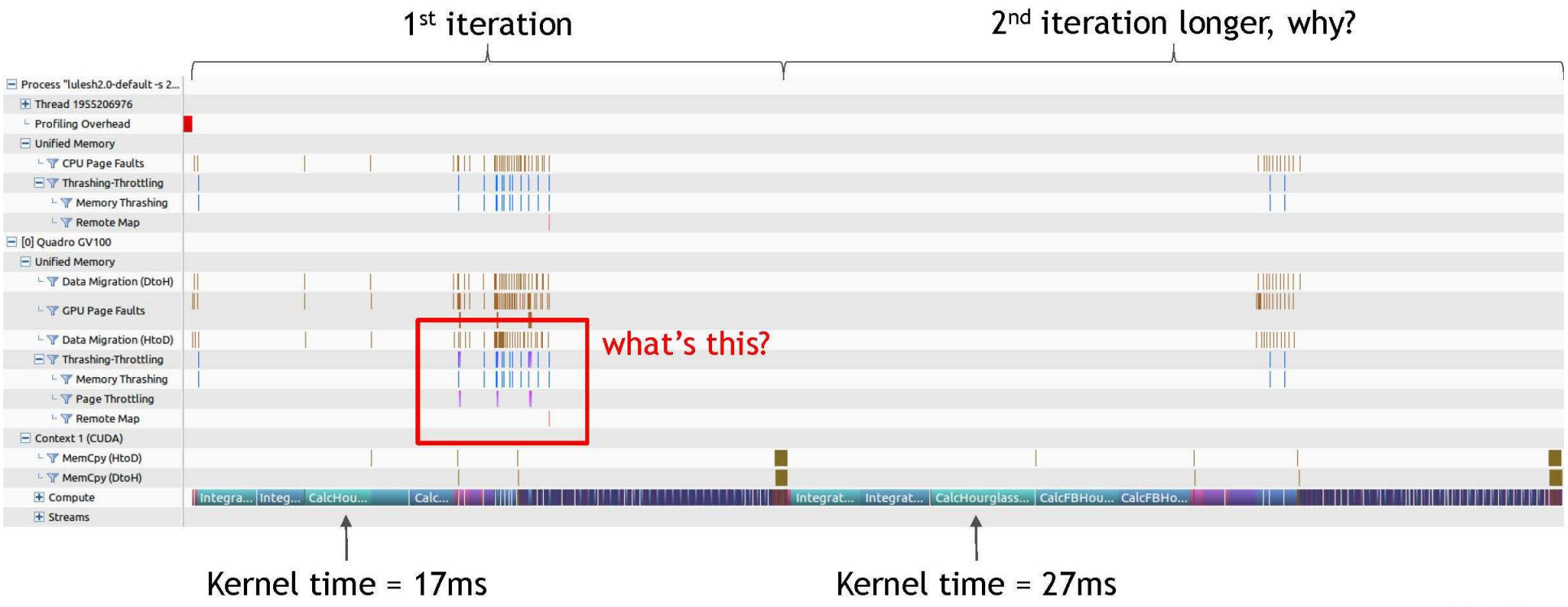
```
while not converged:
  LagrangeNodal
    CalcForceForNodes
    CalcAccelerationForNodes
    BoundaryConditionsForNodes
    CalcVelocityForNodes
    CalcPositionForNodes
  LagrangeElements
    CalcLagrangeElements
    CalcQForElems
    MaterialPropertiesForElems
  CalcTimeConstraintsForElems
    CalcCourantConstraintForElems
    CalcHydroConstraintForElems
```

few heavy GPU kernels

many small GPU kernels
many small CPU functions

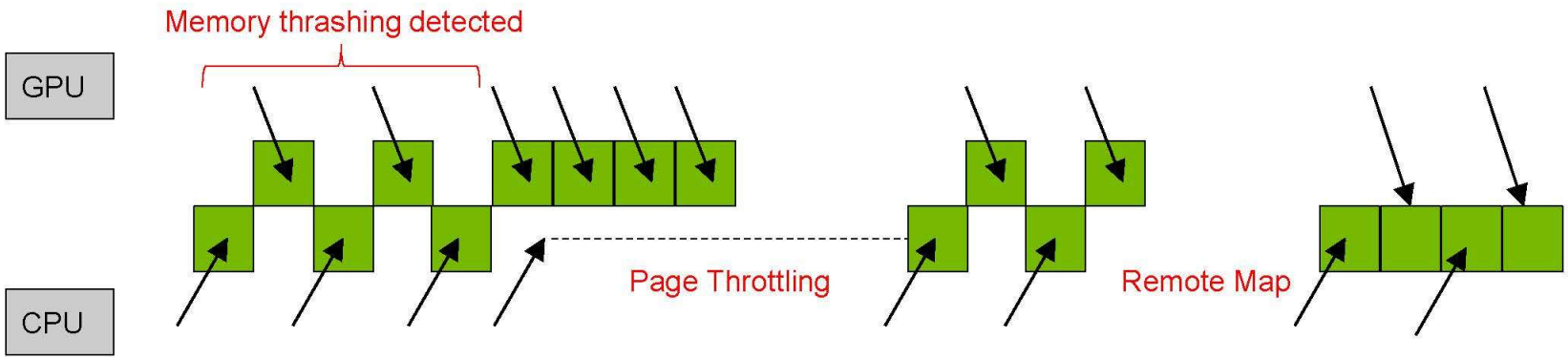


USING VISUAL PROFILER



THRASHING MITIGATION

Processors frequently read or write to the same page

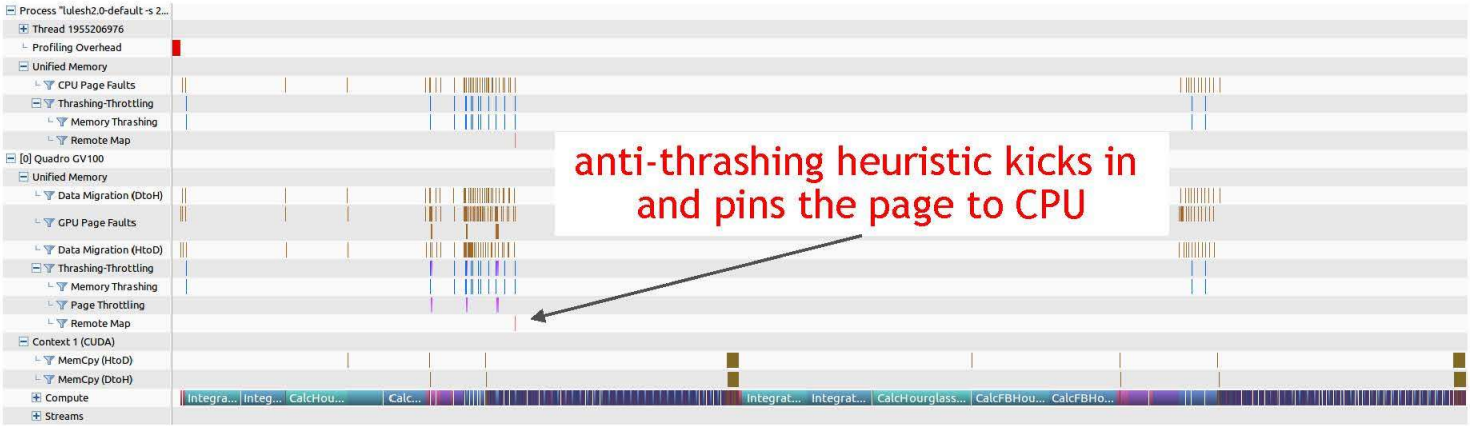


Before CUDA 9.2: when memory is pinned we lose any insight into access pattern

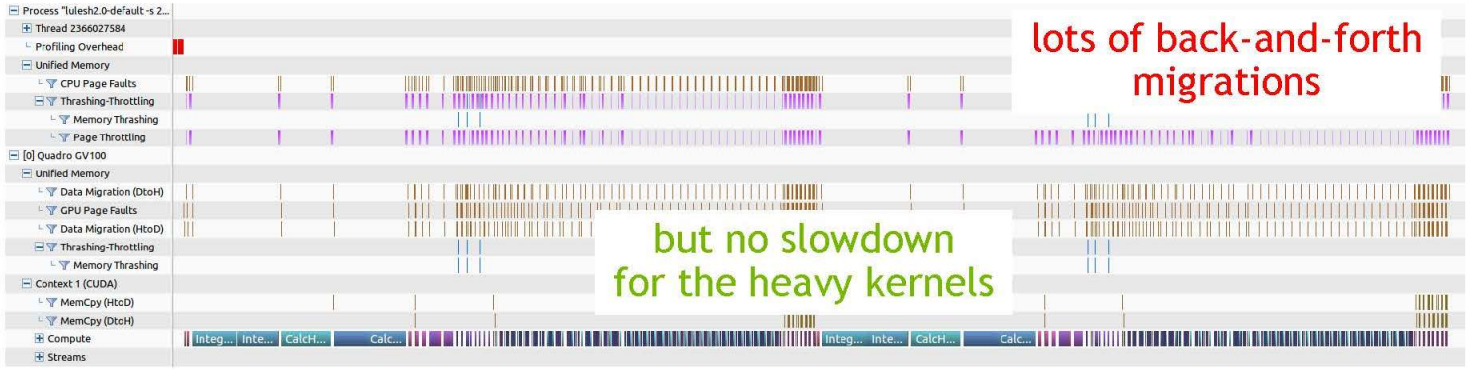
In the future we may use access counters on Volta to find a better location

LULESH: PREFERRED LOCATION = GPU

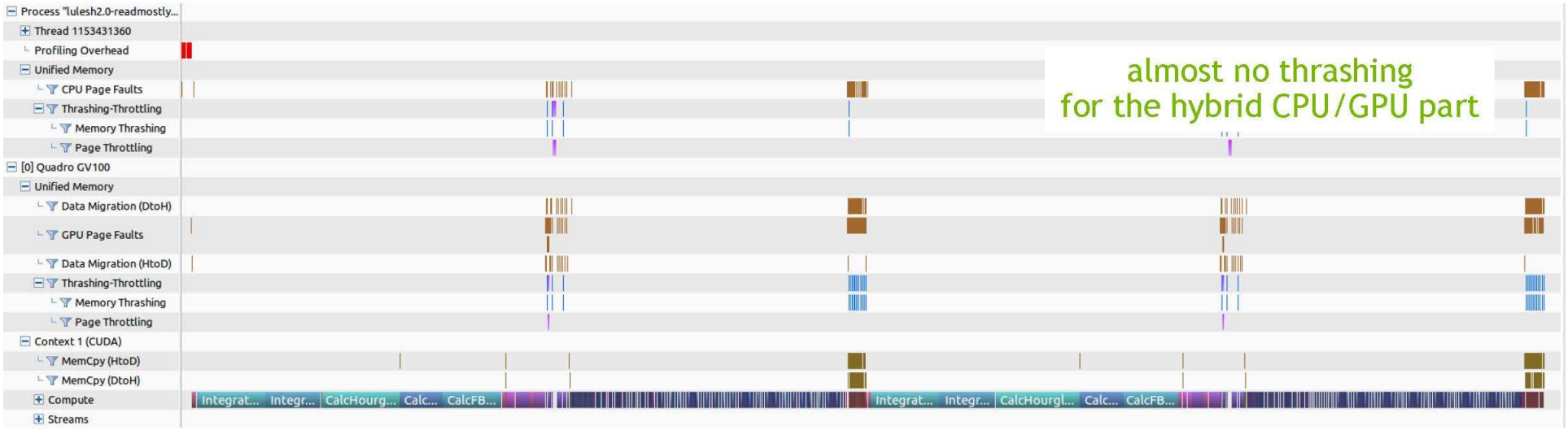
No policies



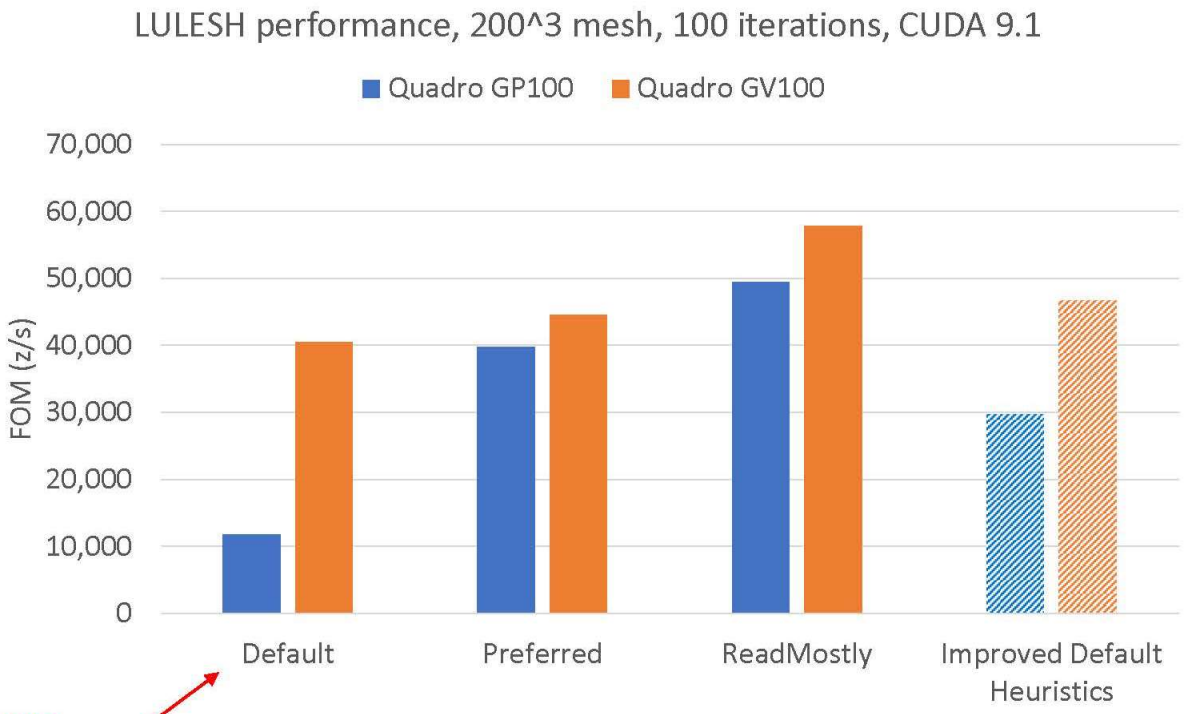
Preferred Location



LULESH: READ MOSTLY



LULESH: PERF IMPROVEMENTS



GP100 has 500x more sysmem writes than GV100

Will be enabled in the future drivers

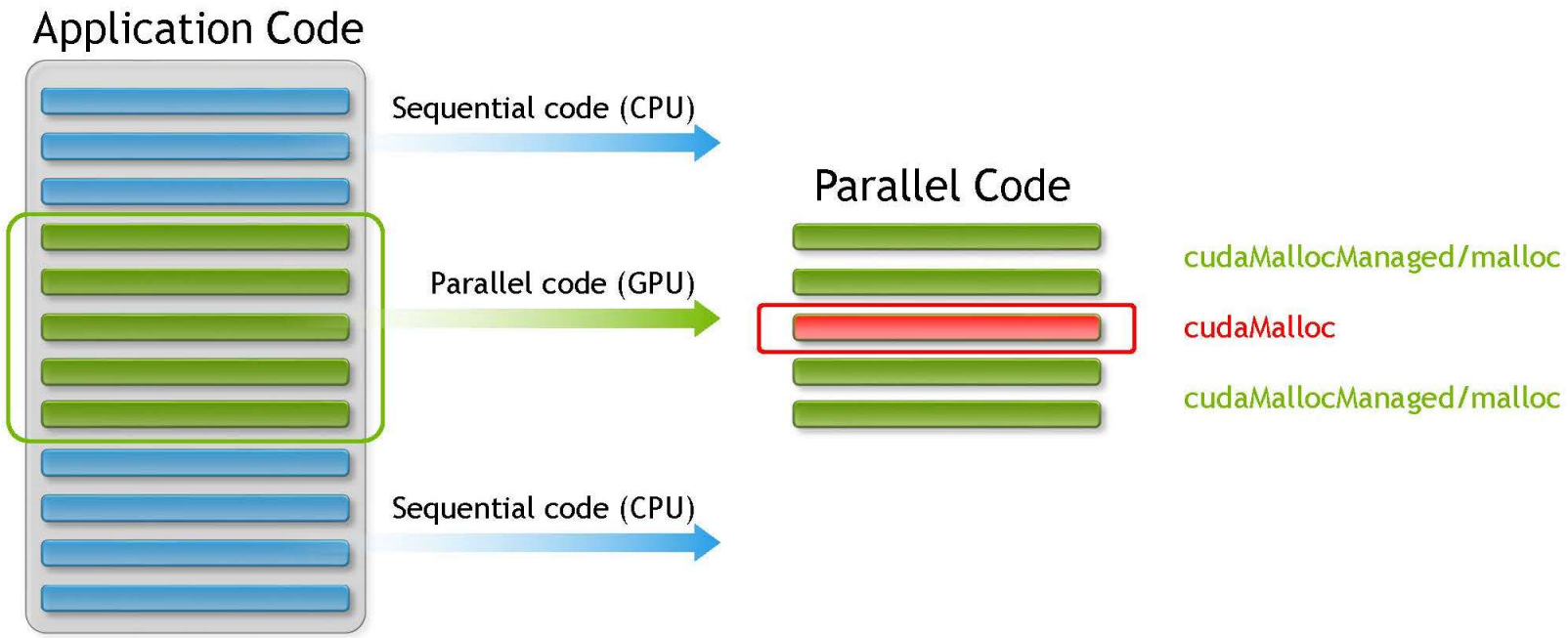
WHEN TO USE UNIFIED MEMORY

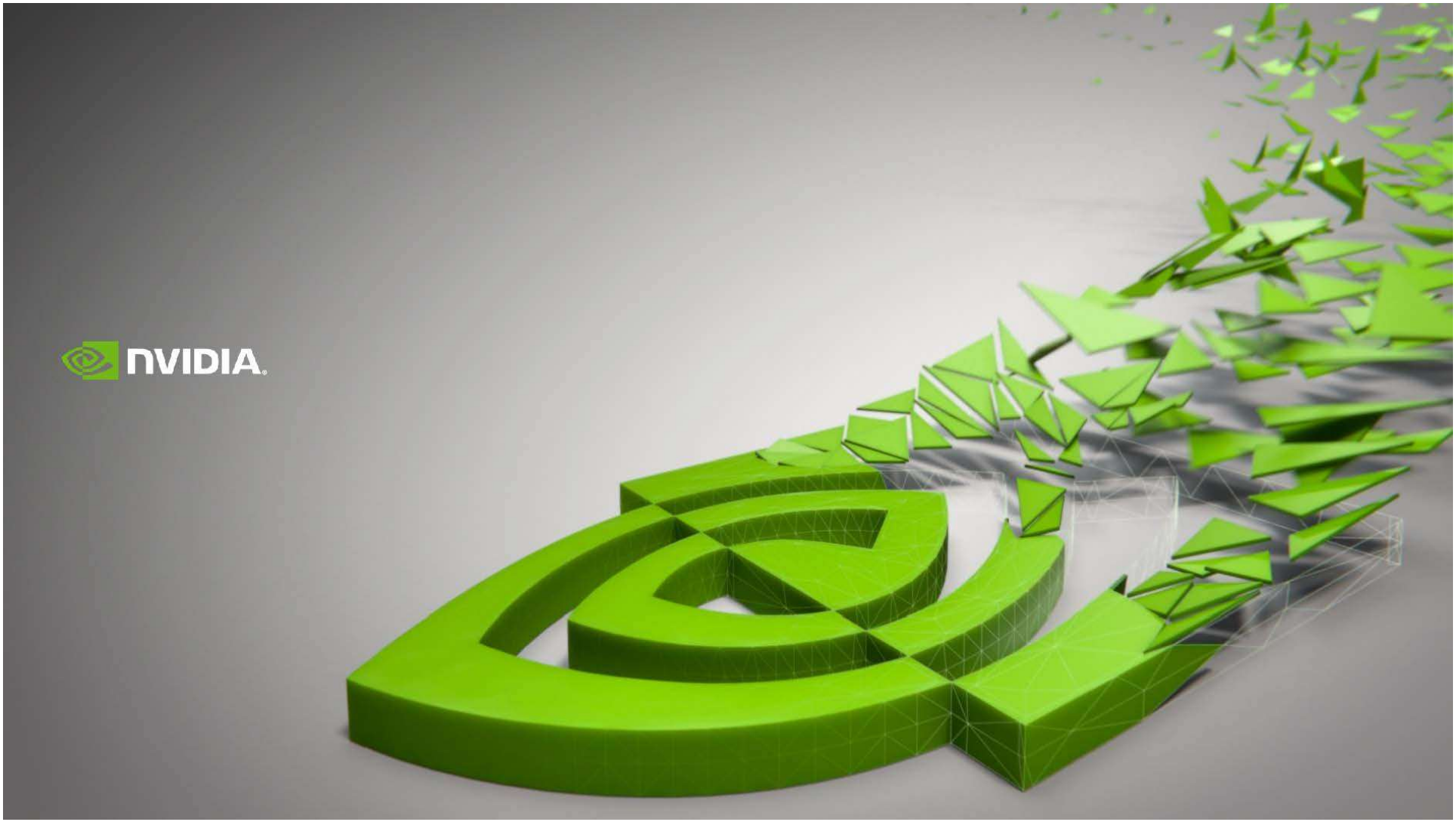
	cudaMalloc	cudaMallocManaged

Pinned allocation	`cudaMalloc`	`cudaMallocManaged`
		`PreferredLocation(GPU)`
		`SetAccessedBy(peer GPUs)`
		`cudaMemPrefetchAsync(GPU)`

cudaMemcpy: ptrA -> ptrB	Staging for non-pinned allocations or between non-P2P GPUs	Staging or a copy kernel required in all cases
Memory migration	Not possible	`cudaMemPrefetchAsync`
Debugging	Difficult	Easy
Oversubscription	No	Yes
IPC support	Yes	No

WHEN TO USE UNIFIED MEMORY




NVIDIA.

UNIFIED MEMORY PLATFORMS

	KEPLER	PASCAL	VOLTA
Linux + x86	No GPU fault support No concurrent access	On-demand migration	On-demand migration
Linux + Power		On-demand migration 80GB/s CPU-GPU BW*	On-demand migration 150GB/s CPU-GPU BW** Access counters HW coherency ATS support
Windows	No GPU fault support No concurrent access		
MacOS	No GPU fault support No concurrent access		
Tegra	Cached on CPU and iGPU No concurrent access		

**IBM Minsky: 4xP100 + 2xP8, 2xNVLINK1 links between P100 and P8, bi-directional aggregate BW*
***IBM Newell: 4xV100 + 2xP9, 3xNVLINK2 links between V100 and P9, bi-directional aggregate BW*

READ DUPLICATION

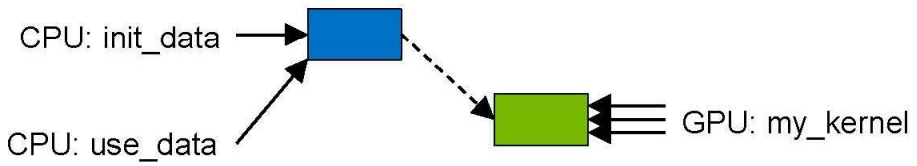
Usage example

```
char *data;
cudaMallocManaged(&data, N);

init_data(data, N);

cudaMemAdvise(data, N, ..SetReadMostly, myGpuId);
cudaMemPrefetchAsync(data, N, myGpuId, s);
mykernel<<<..., s>>>(data, N);


use_data(data, N);

cudaFree(data);
```

The prefetch creates a copy instead of moving data

Both processors can read data **simultaneously** without faults

Writes will collapse all copies into one, subsequent reads will fault and duplicate



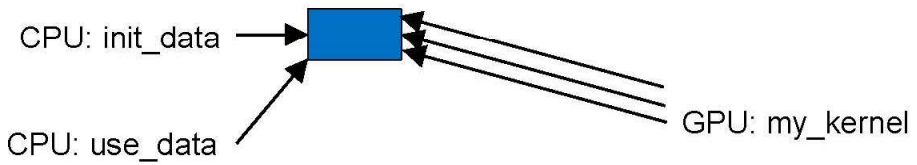
PREFERRED LOCATION

Resisting migrations

```
char *data;
cudaMallocManaged(&data, N);

init_data(data, N);

cudaMemAdvise(data, N, ..PreferredLocation, cudaCpuDeviceId);

mykernel<<<..., s>>>(data, N);


use_data(data, N);

cudaFree(data);
```

The kernel will *page fault* and generate direct mapping to data on the CPU

The driver will "resist" migrating data away from the preferred location



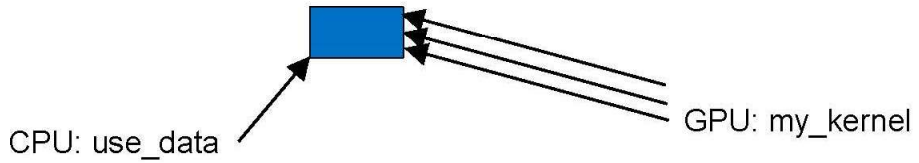
PREFERRED LOCATION

Page population on first-touch

```
char *data;
cudaMallocManaged(&data, N);


cudaMemAdvise(data, N, ..PreferredLocation, cudaCpuDeviceId);

mykernel<<<..., s>>>(data, N);


use_data(data, N);

cudaFree(data);
```

The kernel will *page fault*, populate pages on the CPU and generate direct mapping to data on the CPU

Pages are populated on the preferred location if the faulting processor can access it



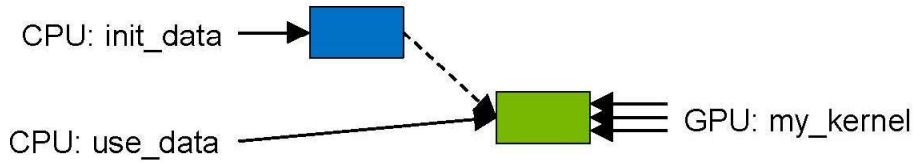
PREFERRED LOCATION ON P9+V100

CPU can directly access GPU memory

```
char *data;
cudaMallocManaged(&data, N);

init_data(data, N);

cudaMemAdvise(data, N, ..PreferredLocation, gpuId);

mykernel<<<..., s>>>(data, N);


use_data(data, N);

cudaFree(data);
```

The kernel will *page fault* and migrate data to the GPU

CPU will fault and access data directly instead of migrating



on non P9+V100 systems the driver will migrate back to the CPU

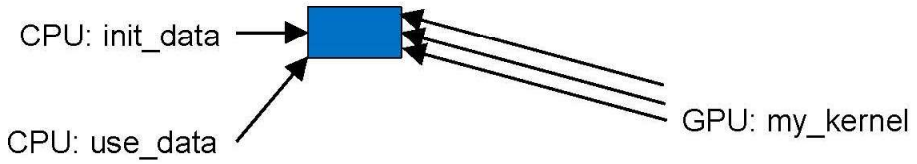
ACCESSED BY

Usage example

```
char *data;
cudaMallocManaged(&data, N);

init_data(data, N);

cudaMemAdvise(data, N, ..SetAccessedBy, myGpuId);

mykernel<<<..., s>>>(data, N);


use_data(data, N);

cudaFree(data);
```

GPU will establish direct mapping of data in CPU memory, **no page faults** will be generated

Memory can move freely to other processors and mapping will carry over



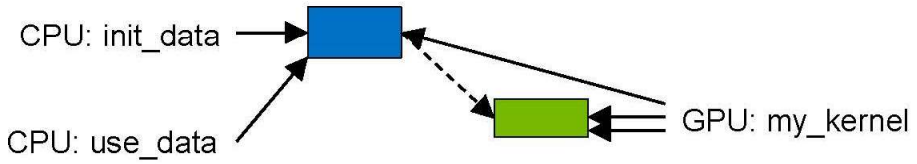
ACCESSED BY

Using access counters on Volta

```
char *data;
cudaMallocManaged(&data, N);

init_data(data, N);

cudaMemAdvise(data, N, ..SetAccessedBy, myGpuId);

mykernel<<<..., s>>>(data, N);


use_data(data, N);

cudaFree(data);
```

GPU will establish direct mapping of data in CPU memory, **no page faults** will be generated

Access counters may eventually trigger migration of **frequently accessed pages** to the GPU



MANAGED VS MALLOC ON VOLTA+P9

First touch allocation policy

```
ptr = cudaMallocManaged(size);

doStuffOnGpu<<<...>>>(ptr, size);
```

GPU page faults

Unified Memory driver allocates on GPU

GPU accesses **GPU memory**

```
ptr = malloc(size);

doStuffOnGpu<<<...>>>(ptr, size);
```

GPU uses ATS, faults

OS allocates on CPU (by default)

GPU uses ATS to access **CPU memory**

**You may alter this behavior by using cudaMemAdvise policies*

MANAGED VS MALLOC ON P9

cudaMallocManaged: same behavior as x86

```
ptr = cudaMallocManaged(size);

fillData(ptr, size);

doStuffOnGpu<<<...>>>(ptr, size);

cudaDeviceSynchronize();

doStuffOnCpu(ptr, size);
```

GPU page faults
ptr migrated to GPU

CPU page faults
ptr migrated to CPU

MANAGED VS MALLOC ON P9

malloc: no on-demand migrations*

```
ptr = malloc(size);

fillData(ptr, size);

doStuffOnGpu<<<...>>>(ptr, size);

cudaDeviceSynchronize();

doStuffOnCpu(ptr, size);
```

GPU uses ATS to access CPU memory (no on-demand migration except cudaMemPrefetchAsync*)

CPU accesses CPU memory

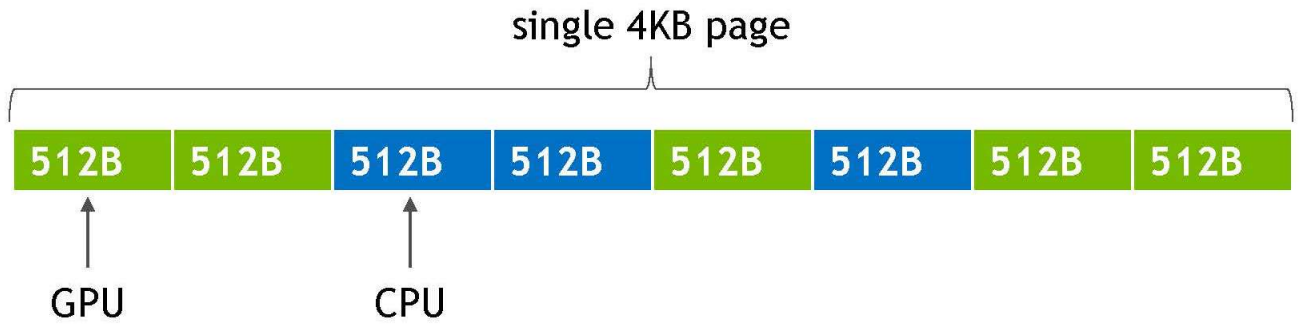
*In the future Volta access counters will be used to migrate malloc memory

HYPRE-INSPIRED USE CASE

Algebraic Multi-Grid library: https://github.com/LLNL/hypre

Lots of small allocations: multiple variables may end up on the same page

If used by different processors this will result in **false-sharing**



FALSE-SHARING

Issues with false-sharing:

- Spurious migrations, thrashing mitigation does not solve it
- Performance hints are applied on page boundaries, due to suballocation data may inherit the wrong policies

How to mitigate this:

- Use separate allocators or memory pools for CPU and GPU

Better Performance at Lower Occupancy

Vasily Volkov
UC Berkeley

September 22, 2010

Prologue

It is common to recommend:

- running more threads per multiprocessor
- running more threads per thread block

Motivation: this is the only way to hide latencies

- But…

Faster codes run at lower occupancy:

Multiplication of two large matrices, single precision (SGEMM):

	CUBLAS 1.1	CUBLAS 2.0	
Threads per block	512	64	8x smaller thread blocks
Occupancy (G80)	67%	33%	2x lower occupancy
Performance (G80)	128 Gflop/s	204 Gflop/s	1.6x higher performance

Batch of 1024-point complex-to-complex FFTs, single precision:

	CUFFT 2.2	CUFFT 2.3	
Threads per block	256	64	4x smaller thread blocks
Occupancy (G80)	33%	17%	2x lower occupancy
Performance (G80)	45 Gflop/s	93 Gflop/s	2x higher performance

Maximizing occupancy, you may lose performance

Two common fallacies:

- multithreading is the only way to hide latency on GPU
- shared memory is as fast as registers

This talk

I. Hide arithmetic latency using fewer threads
II. Hide memory latency using fewer threads
III. Run faster by using fewer threads
IV. Case study: matrix multiply
V. Case study: FFT

Part I:

Hide arithmetic latency using fewer threads

Arithmetic latency

Latency: time required to perform an operation

- ≈20 cycles for arithmetic; 400+ cycles for memory
- Can't start a *dependent* operation for this time
- Can hide it by overlapping with other operations

```
x = a + b;// takes ≈20 cycles to execute
y = a + c;// independent, can start anytime
(stall)
z = x + d;// dependent, must wait for completion
```

Arithmetic throughput

Latency is often confused with throughput

- E.g. “arithmetic is 100x faster than memory – costs 4 cycles per warp (G80), whence memory operation costs 400 cycles”
  - One is rate, another is time

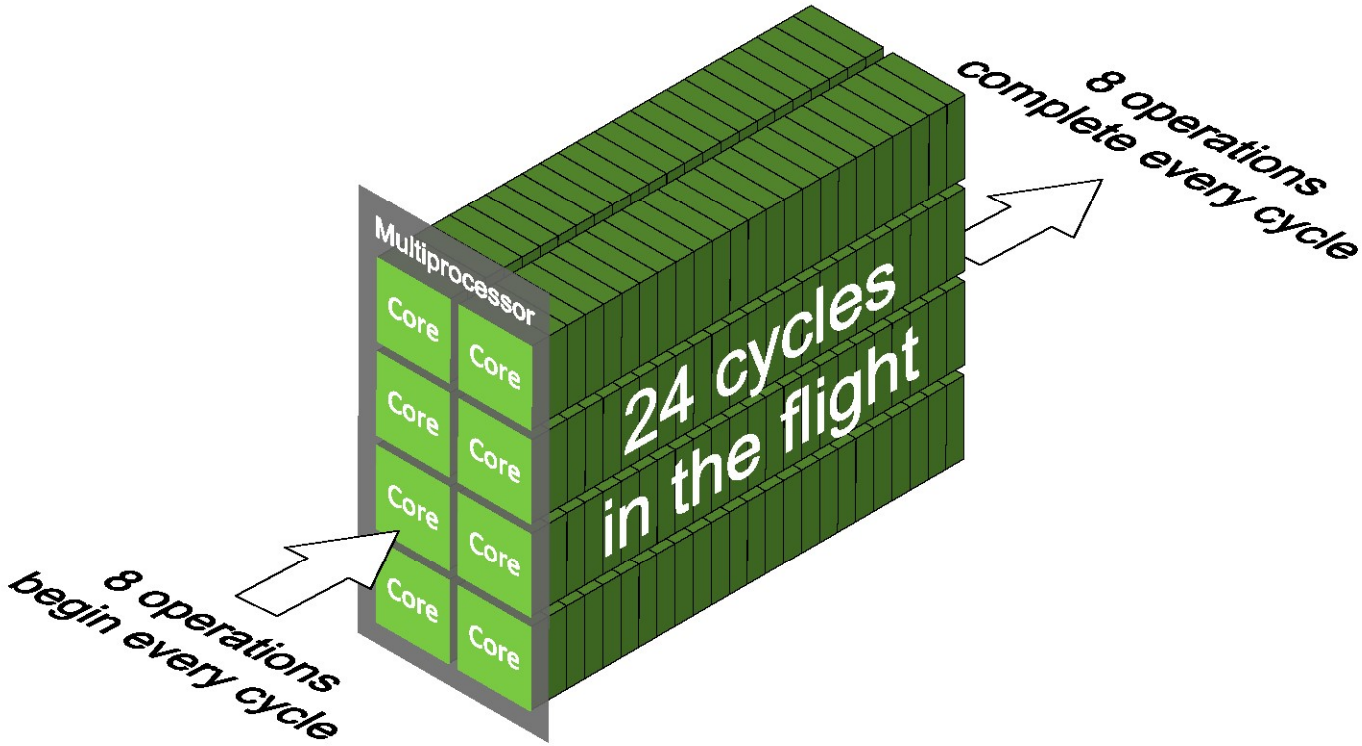
Throughput: how many operations complete per cycle

- Arithmetic: 1.3 Tflop/s = 480 ops/cycle (op=multiply-add)
- Memory: 177 GB/s ≈ 32 ops/cycle (op=32-bit load)

Hide latency = do other operations when waiting for latency

- Will run faster
- But not faster than the peak
- How to get the peak?

Use Little's law



Needed parallelism = **Latency** x **Throughput**

Arithmetic parallelism in numbers

GPU model	Latency (cycles)	Throughput (cores/SM)	Parallelism (operations/SM)
G80-GT200	≈24	8	≈192
GF100	≈18	32	≈576
GF104	≈18	48	≈864

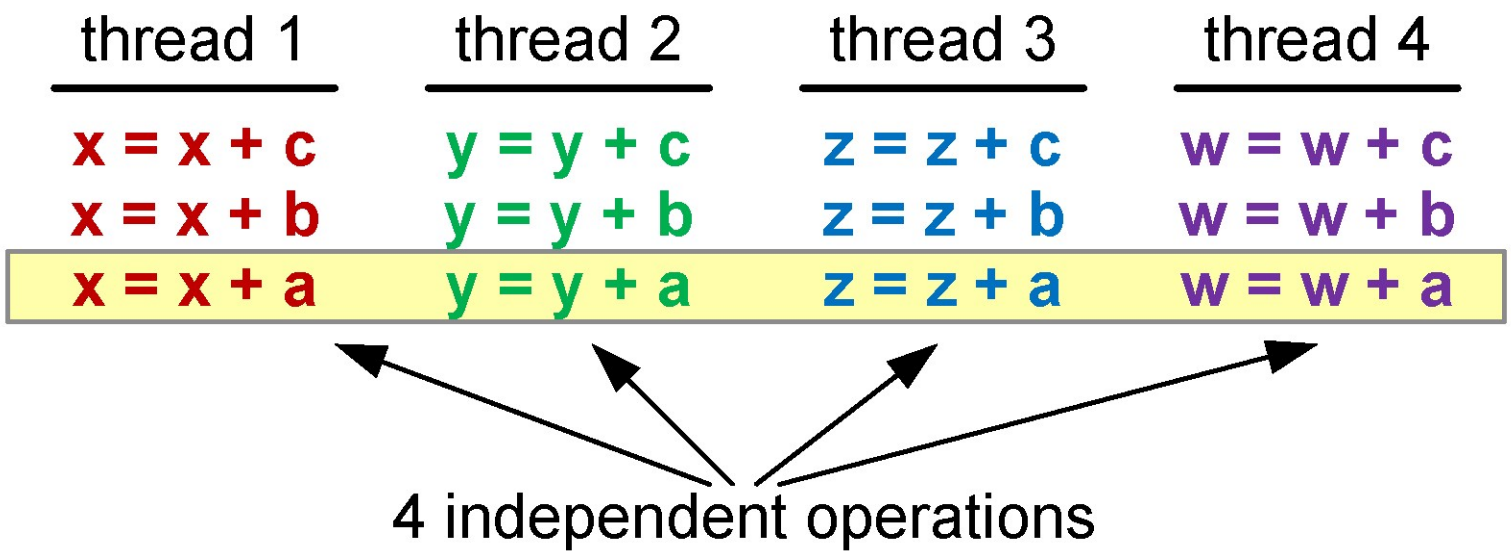
(latency varies between different types of ops)

Can't get 100% throughput with less parallelism

– Not enough operations in the flight = idle cycles

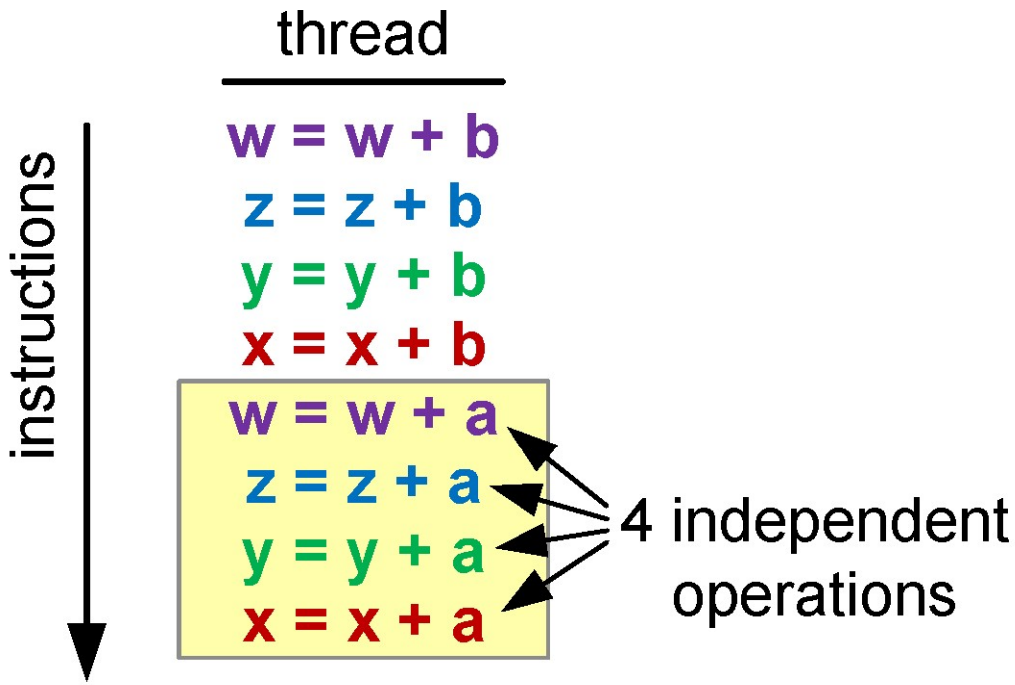
Thread-level parallelism (TLP)

It is usually recommended to use threads to supply the needed parallelism, e.g. 192 threads per SM on G80:



Instruction-level parallelism (ILP)

But you can also use parallelism among instructions in a single thread:



You can use both ILP and TLP on GPU

This applies to all CUDA-capable GPUs. E.g. on G80:

- Get ≈100% peak with 25% occupancy if no ILP
- Or with 8% occupancy, if 3 operations from each thread can be concurrently processed

On GF104 you *must* use ILP to get >66% of peak!

- 48 cores/SM, one instruction is broadcast across 16 cores
- So, must issue 3 instructions per cycle
- But have only 2 warp schedulers
- Instead, it can issue 2 instructions per warp in the same cycle

Let's check it experimentally

Do many arithmetic instructions with no ILP:

```
#pragma unroll UNROLL
for( int i = 0; i < N_ITERATIONS; i++ )
{
  a = a * b + c;
}
```

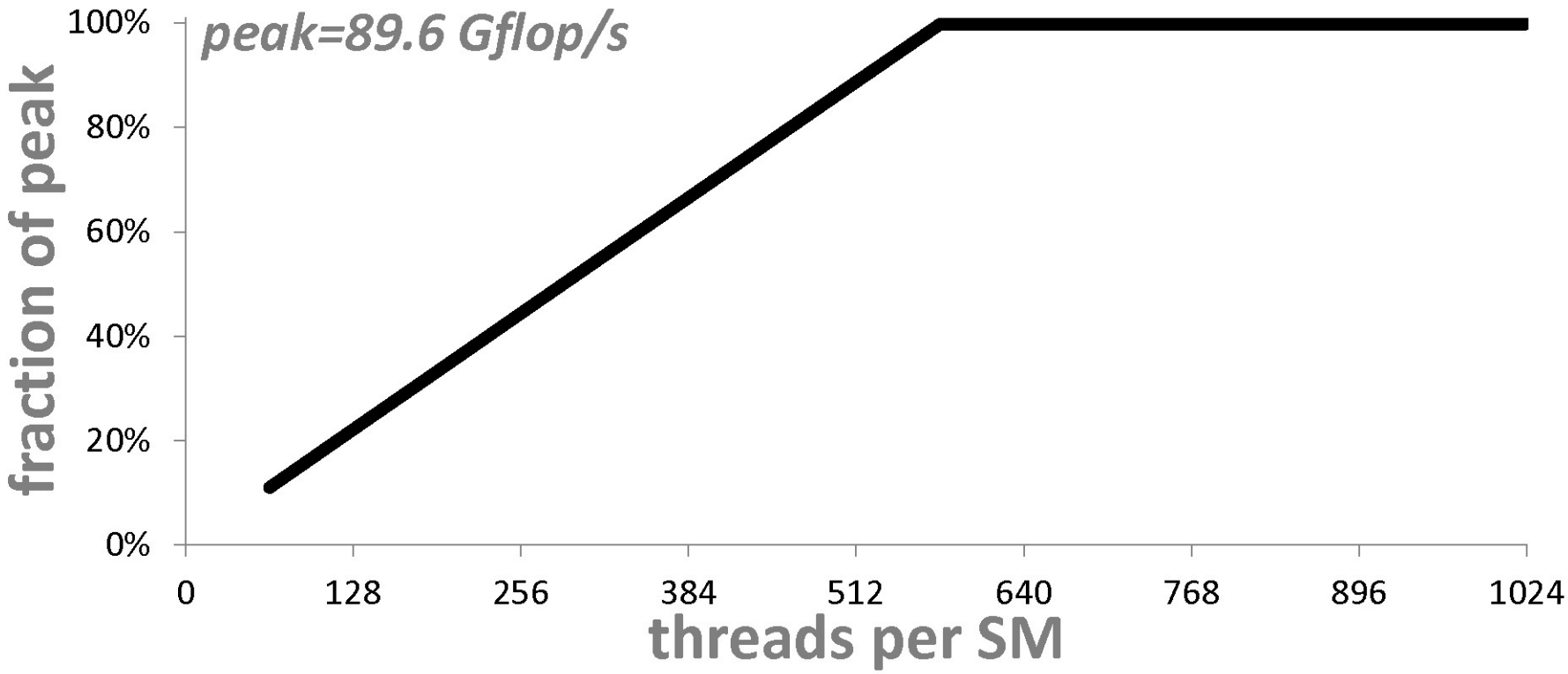
Choose large **`N_ITERATIONS`** and suitable **`UNROLL`**

Ensure **`a`**, **`b`** and **`c`** are in registers and **`a`** is used later

Run 1 block (use 1 SM), vary block size

– See what fraction of peak (1.3TFLOPS/15) we get

Experimental result (GTX480)



No ILP: need 576 threads to get 100% utilization

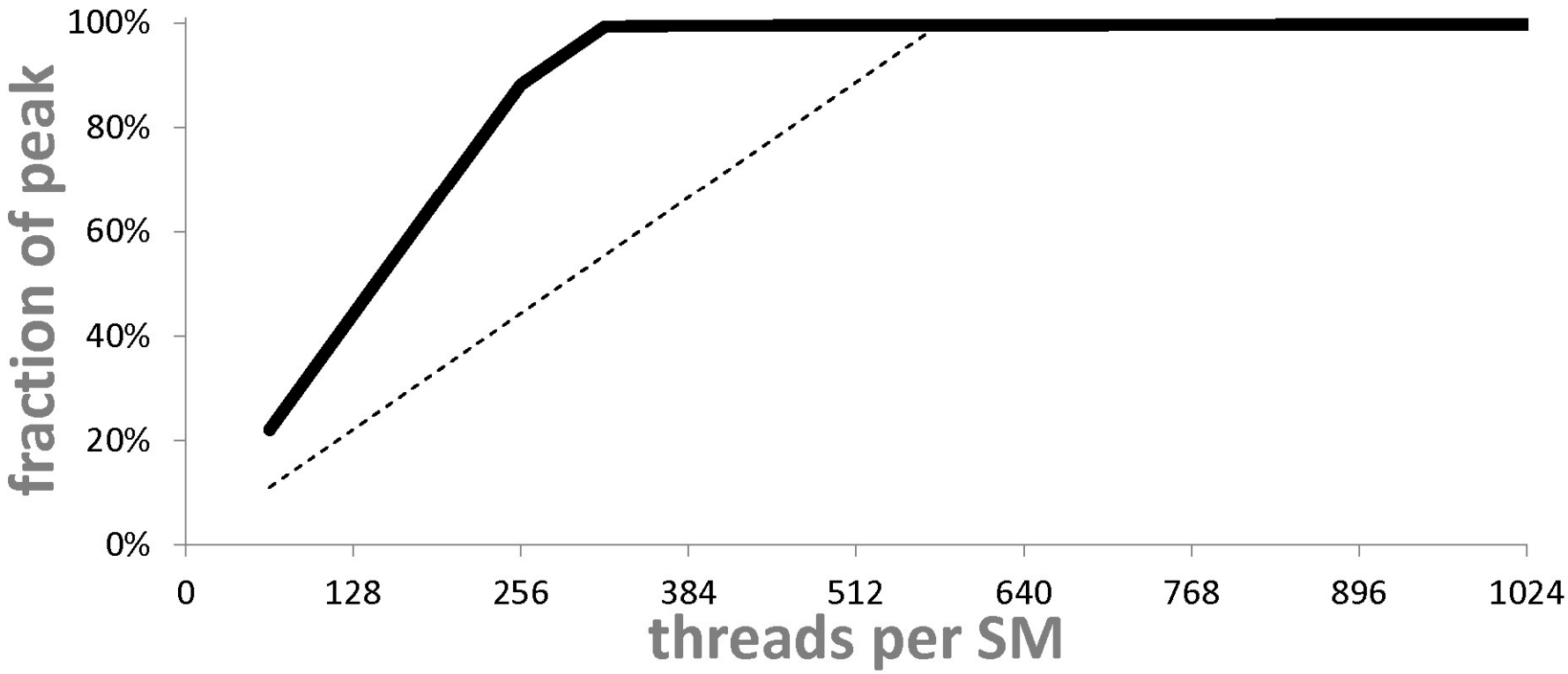
Introduce instruction-level parallelism

Try ILP=2: two independent instruction per thread

```
#pragma unroll UNROLL
for( int i = 0; i < N_ITERATIONS; i++ )
{
  a = a * b + c;
  d = d * b + c;
}
```

If multithreading is the only way to hide latency on GPU, we've got to get the same performance

GPUs can hide latency using ILP



ILP=2: need 320 threads to get 100% utilization

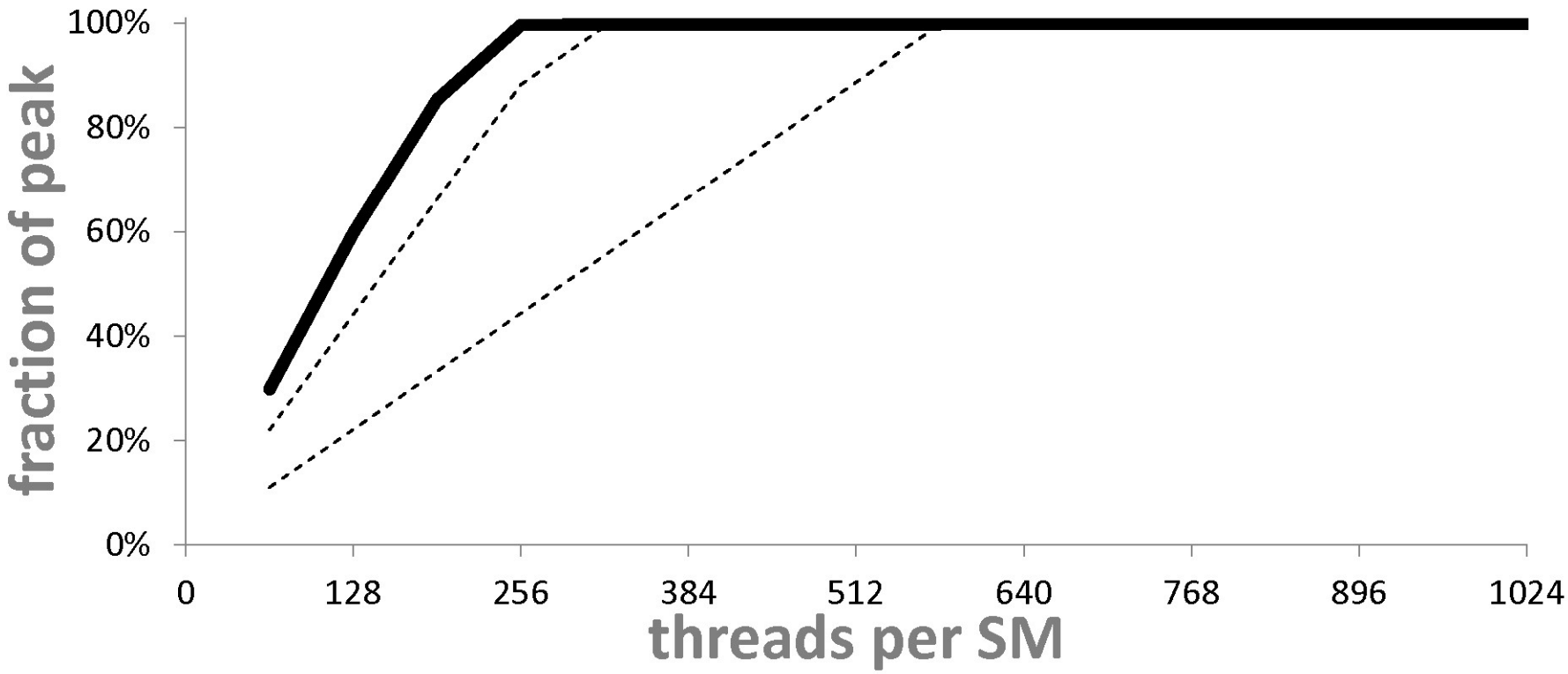
Add more instruction-level parallelism

ILP=3: triples of independent instructions

```
#pragma unroll UNROLL
for( int i = 0; i < N_ITERATIONS; i++ )
{
  a = a * b + c;
  d = d * b + c;
  e = e * b + c;
}
```

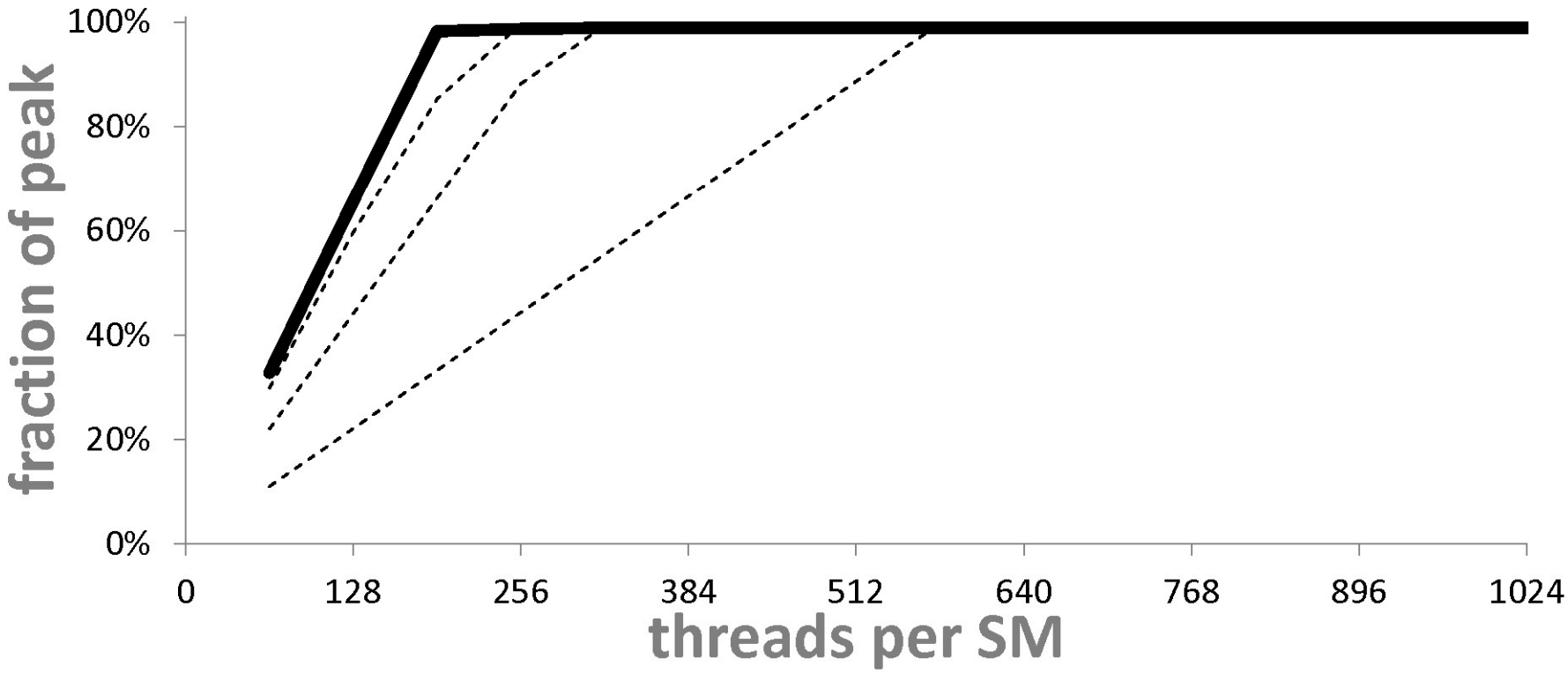
How far can we push it?

Have more ILP – need fewer threads



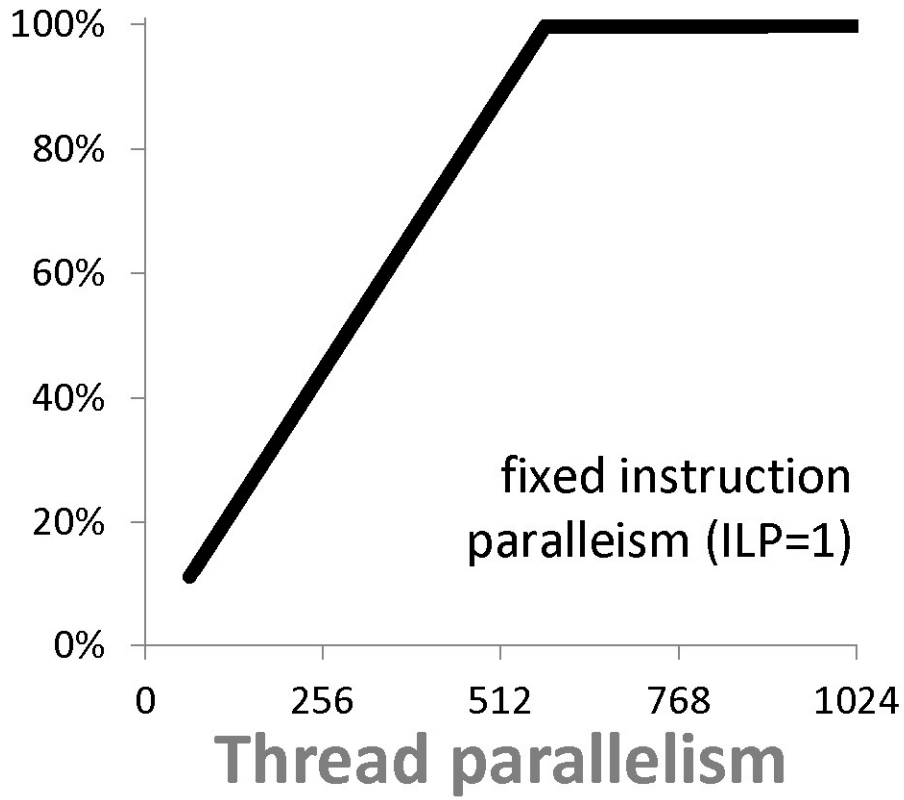
ILP=3: need 256 threads to get 100% utilization

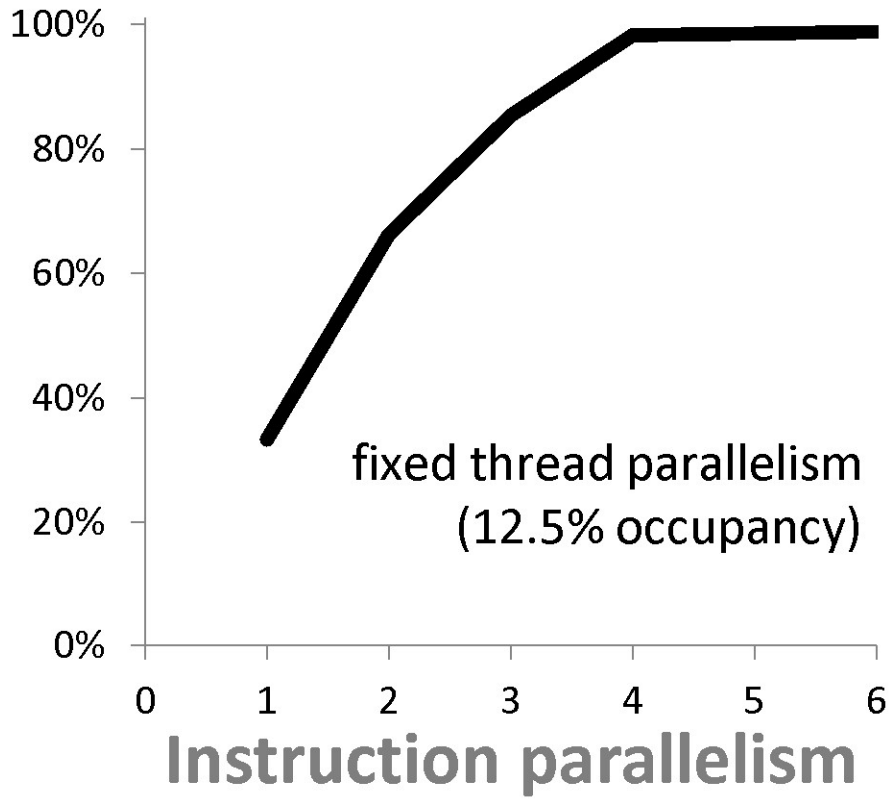
Unfortunately, doesn't scale past ILP=4



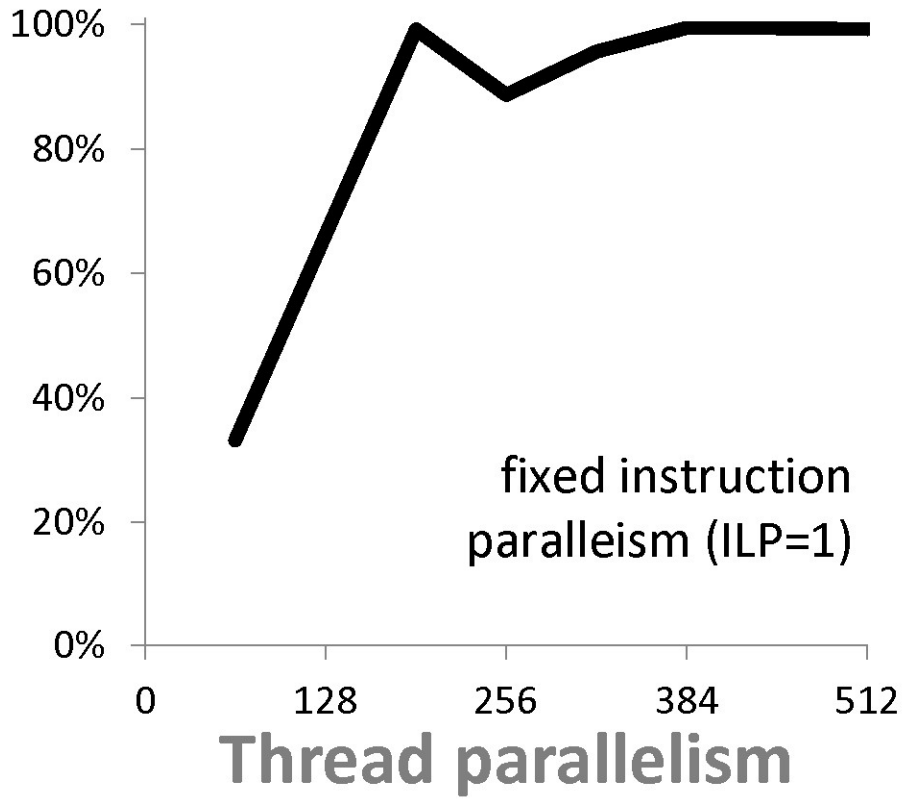
ILP=4: need 192 threads to get 100% utilization

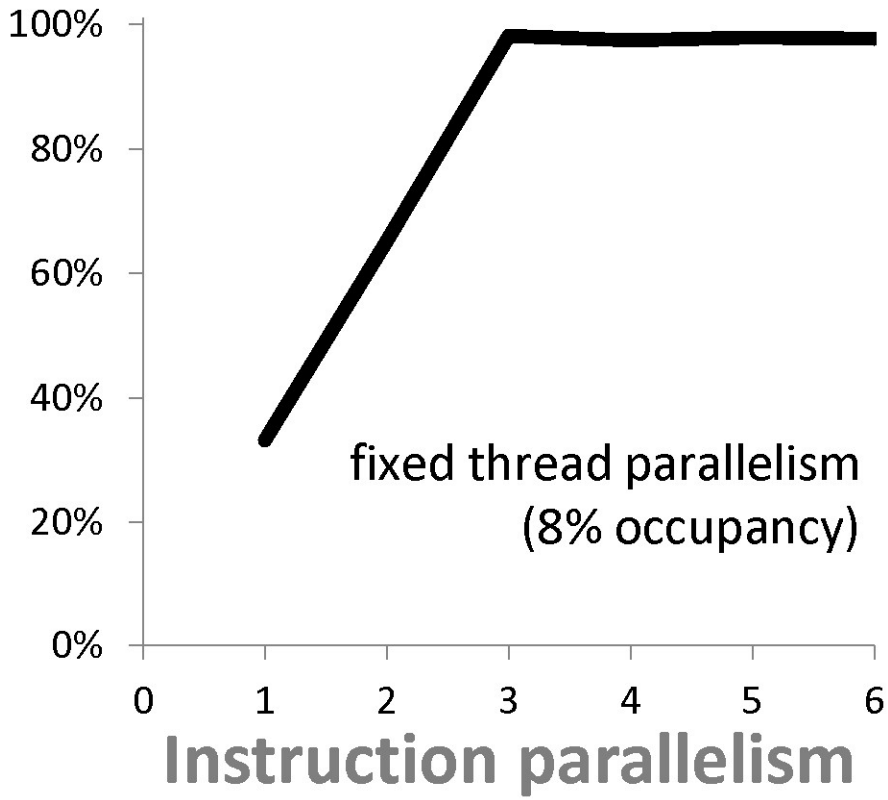
Summary: can hide latency either way





Applies to other GPUs too, e.g. to G80:





Fallacy:

Increasing occupancy is the only way to improve latency hiding

– *No, increasing ILP is another way.*

Fallacy:

Occupancy is a metric of utilization

– *No, it's only one of the contributing factors.*

Fallacy:

"To hide arithmetic latency completely, multiprocessors should be running at least 192 threads on devices of compute capability 1.x (…) or, on devices of compute capability 2.0, as many as 384 threads" (**CUDA Best Practices Guide**)

- *No, it is doable with 64 threads per SM on G80-GT200 and with 192 threads on GF100.*

Part II:

Hide memory latency using fewer threads

Hiding memory latency

Apply same formula but for memory operations:

Needed parallelism = **Latency** x **Throughput**

	Latency	Throughput	Parallelism
Arithmetic	≈18 cycles	32 ops/SM/cycle	576 ops/SM
Memory	< 800 cycles (?)	< 177 GB/s	< 100 KB

So, hide memory latency = keep 100 KB in the flight

– Less if kernel is compute bound (needs fewer GB/s)

How many threads is 100 KB?

Again, there are multiple ways to hide latency

- Use multithreading to get 100KB in the flight
- Use instruction parallelism (more fetches per thread)
- Use bit-level parallelism (use 64/128-bit fetches)

Do more work per thread – need fewer threads

- Fetch 4B/thread – need 25 000 threads
- Fetch 100 B/thread – need 1 000 threads

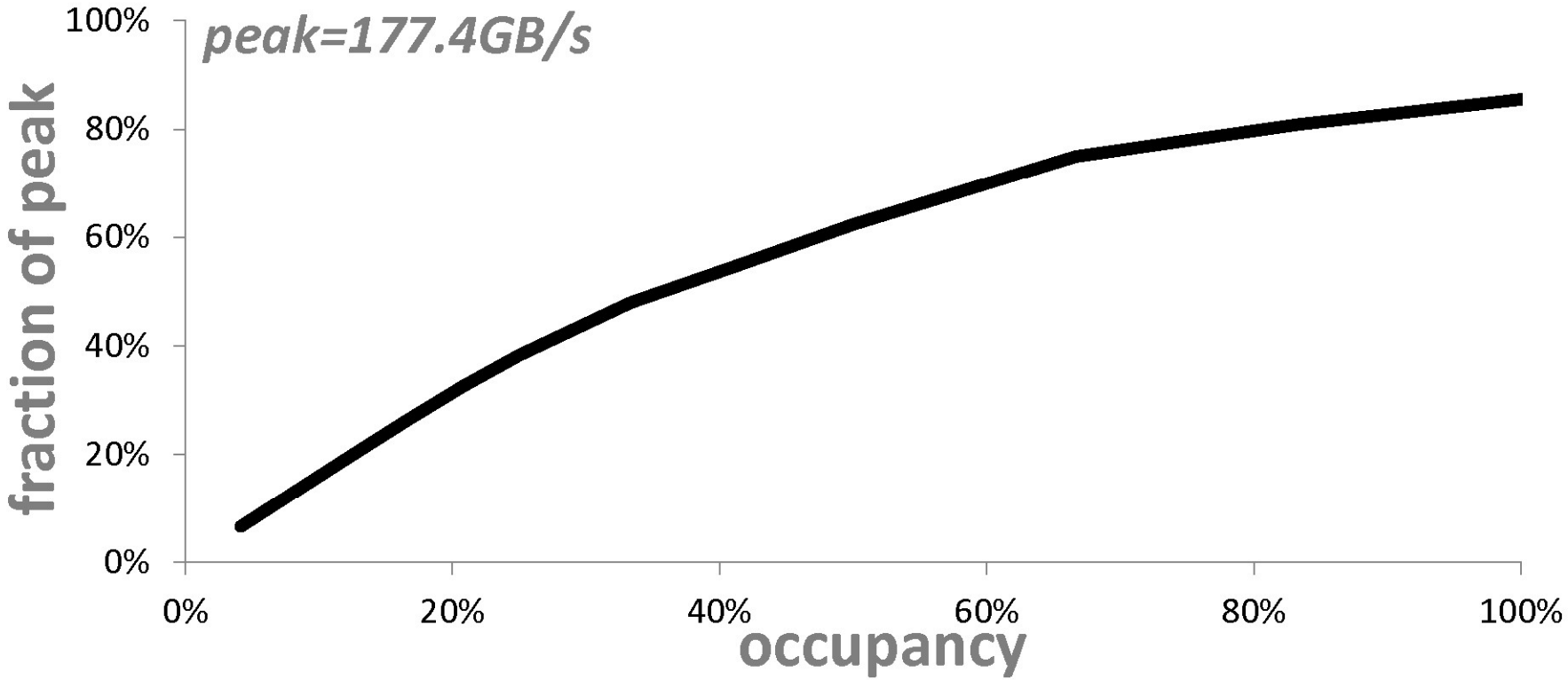
Empirical validation

Copy one float per thread:

```
__global__ void memcpy( float *dst, float *src )
{
  int block = blockIdx.x + blockIdx.y * gridDim.x;
  int index = threadIdx.x + block * blockDim.x;

  float a0 = src[index];
  dst[index] = a0;
}
```

Run many blocks, allocate shared memory dynamically to control occupancy

Copying 1 float per thread (GTX480)



Must maximize occupancy to hide latency?

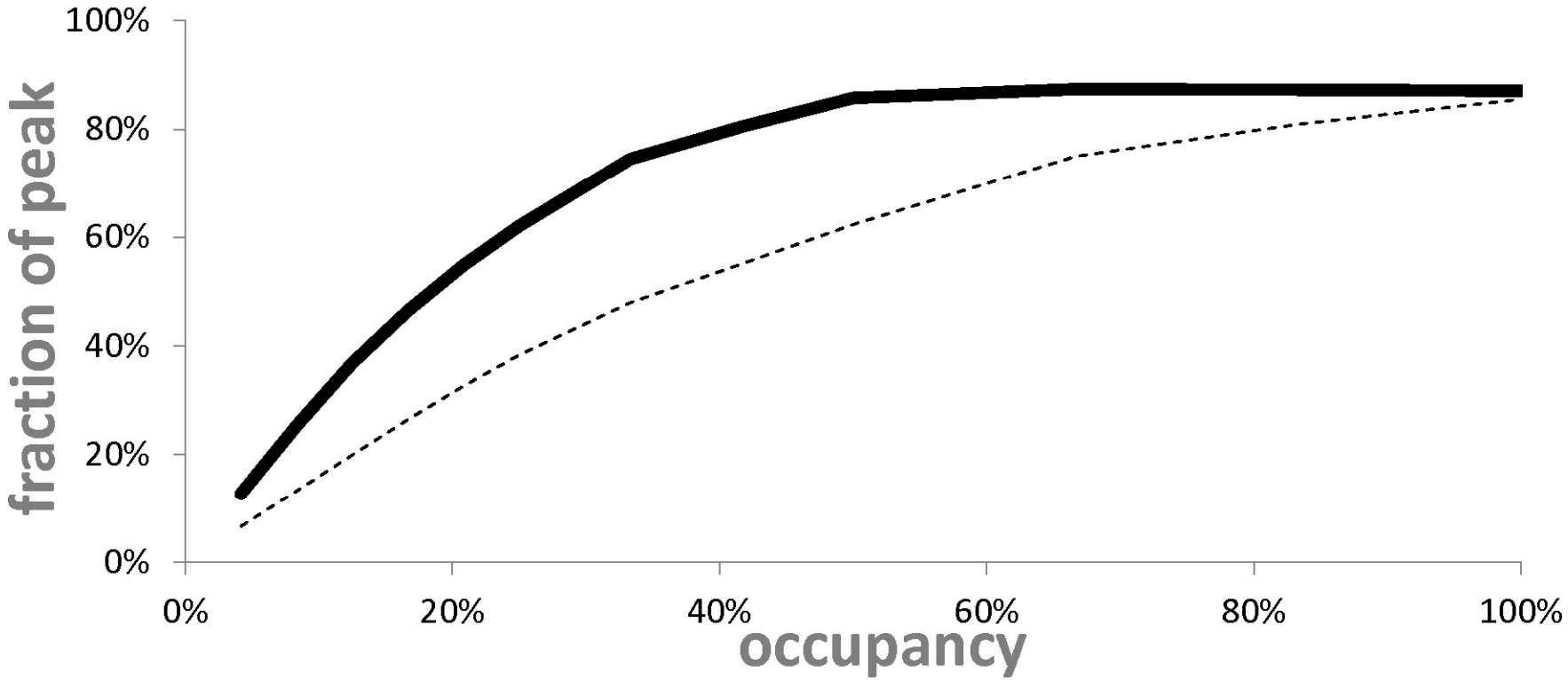
Do more parallel work per thread

```
__global__ void memcpy( float *dst, float *src )
{
  int iblock= blockIdx.x + blockIdx.y * gridDim.x;
  int index = threadIdx.x + 2 * iblock * blockDim.x;

  float a0 = src[index];
  //no latency stall
  float a1 = src[index+blockDim.x];
  //stall
  dst[index] = a0;
  dst[index+blockDim.x] = a1;
}
```

Note, **threads don't stall on memory access**

- Only on data dependency

Copying 2 float values per thread



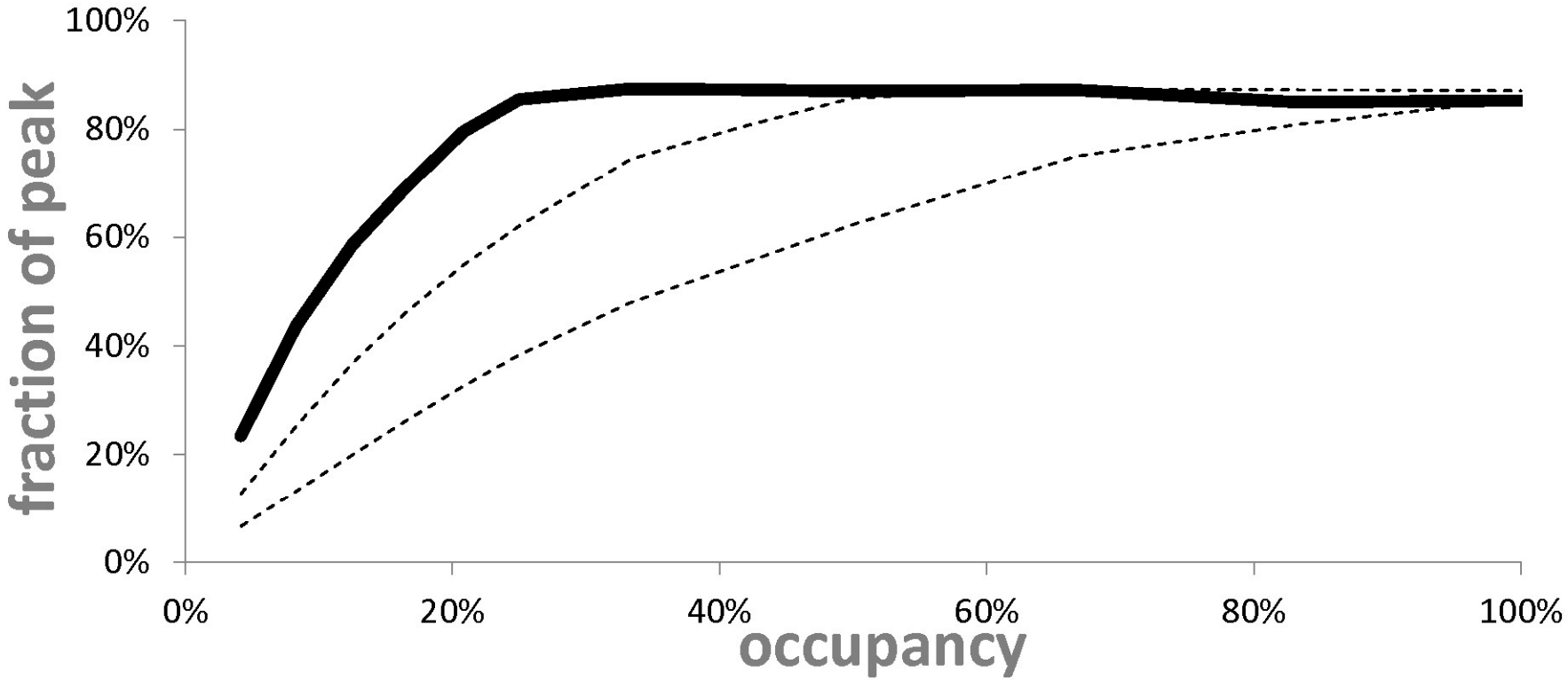
Can get away with lower occupancy now

Do more parallel work per thread

```
__global__ void memcpy( float *dst, float *src )
{
  int iblock = blockIdx.x + blockIdx.y * gridDim.x;
  int index  = threadIdx.x + 4 * iblock * blockDim.x;

  float a[4];//allocated in registers
  for(int i=0;i<4;i++) a[i]=src[index+i*blockDim.x];
  for(int i=0;i<4;i++) dst[index+i*blockDim.x]=a[i];
}
```

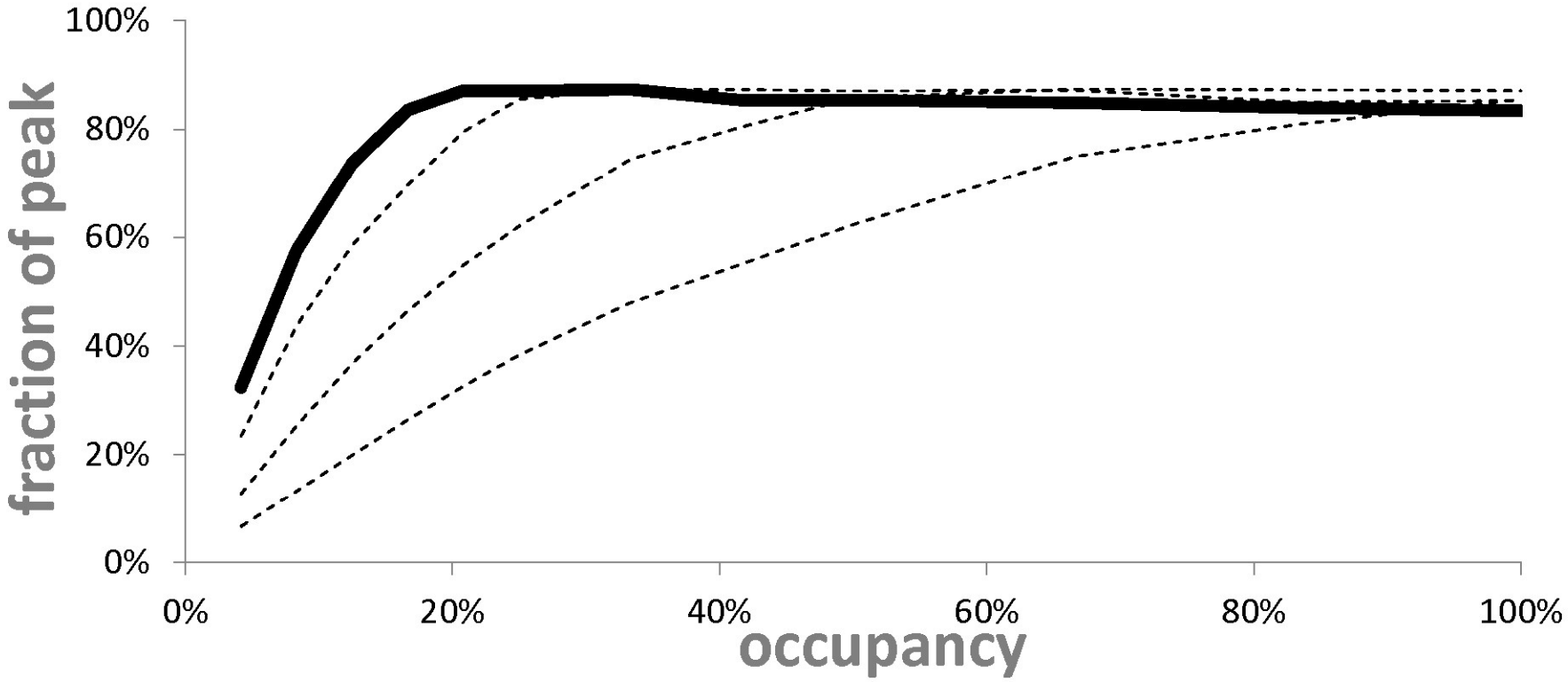
Note, **local arrays are allocated in registers if possible**

Copying 4 float values per thread

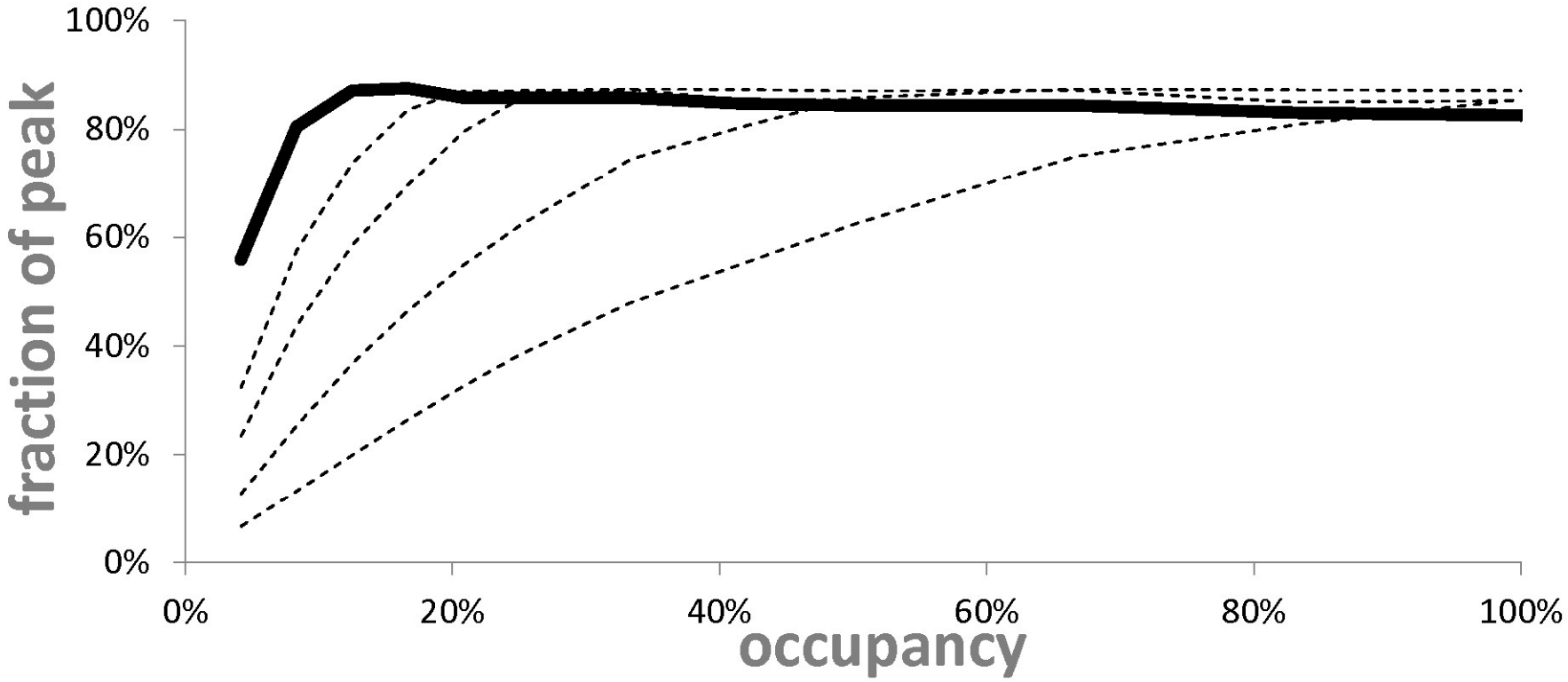


Mere 25% occupancy is sufficient. How far we can go?

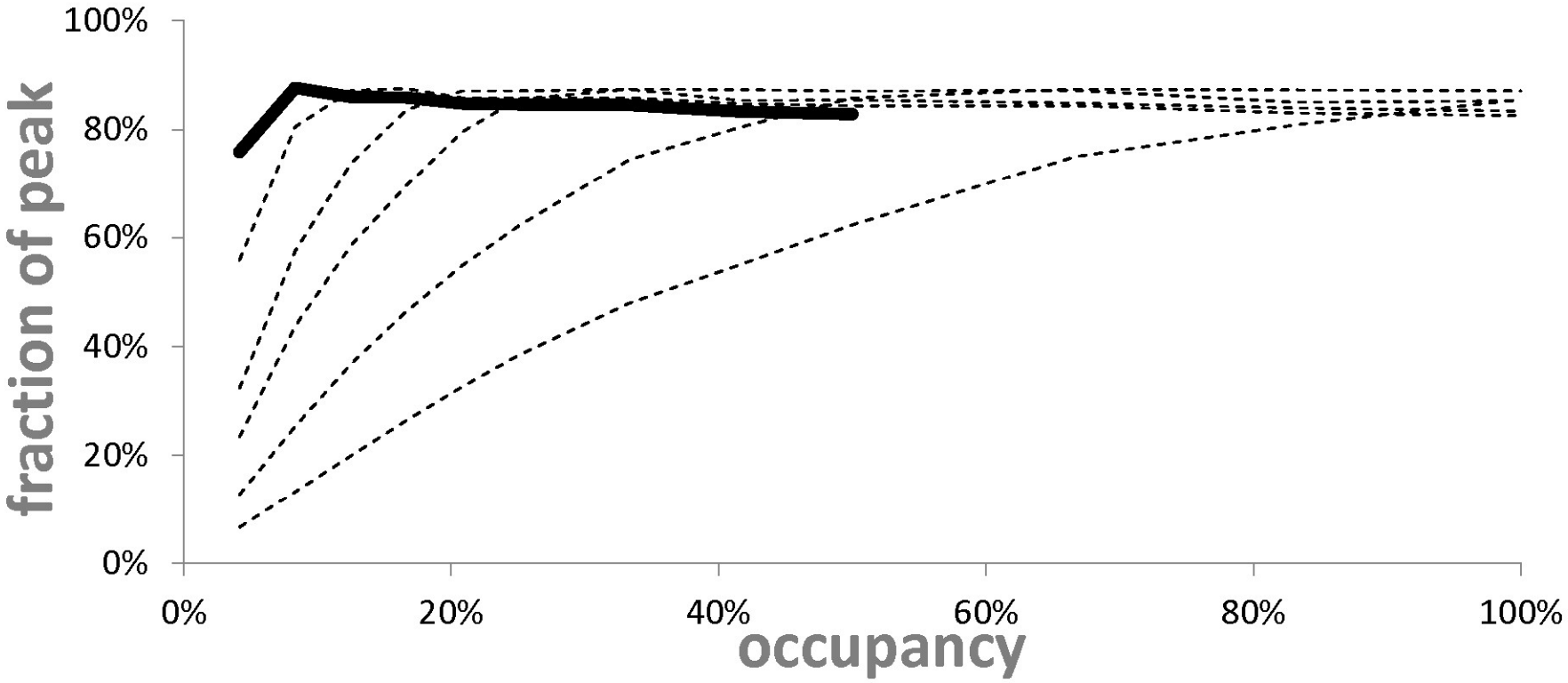
Copying 8 float values per thread



Copying 8 float2 values per thread

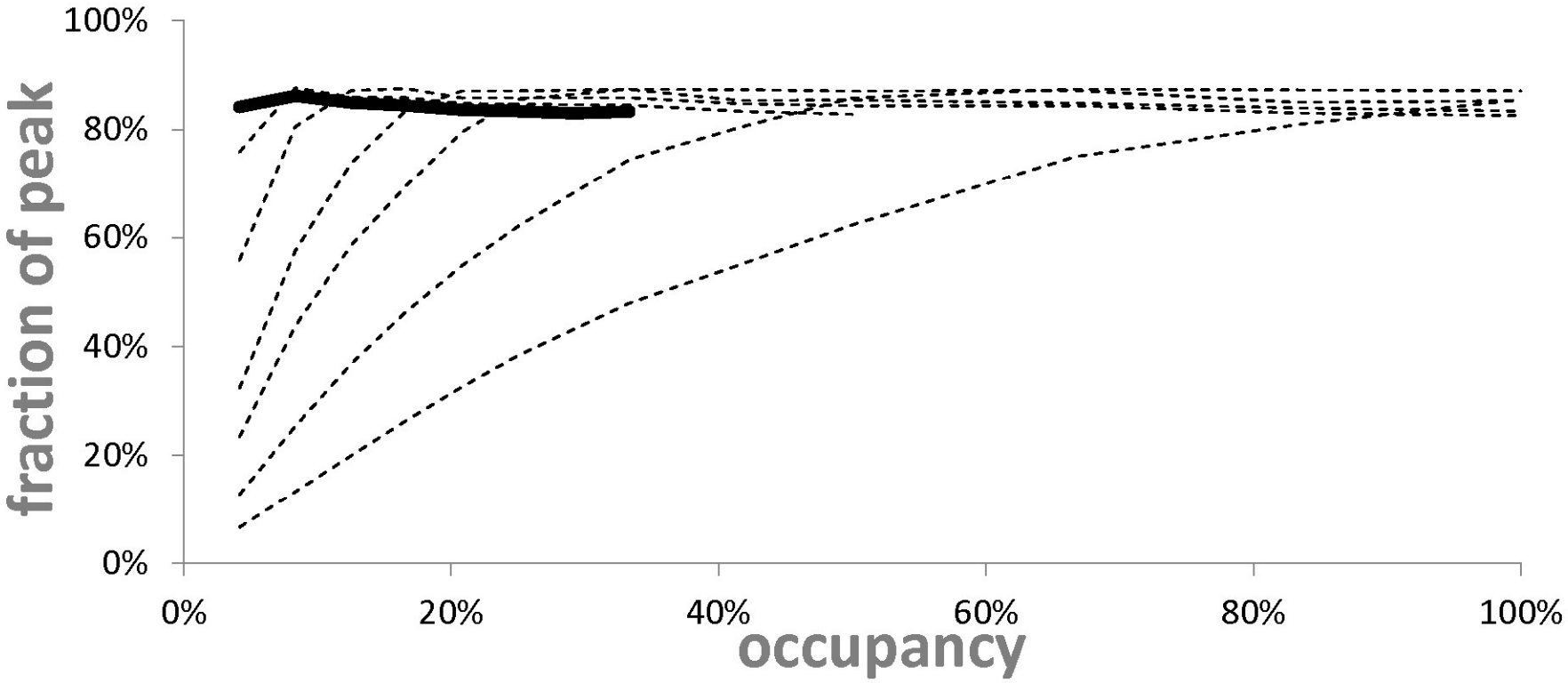


Copying 8 float4 values per thread



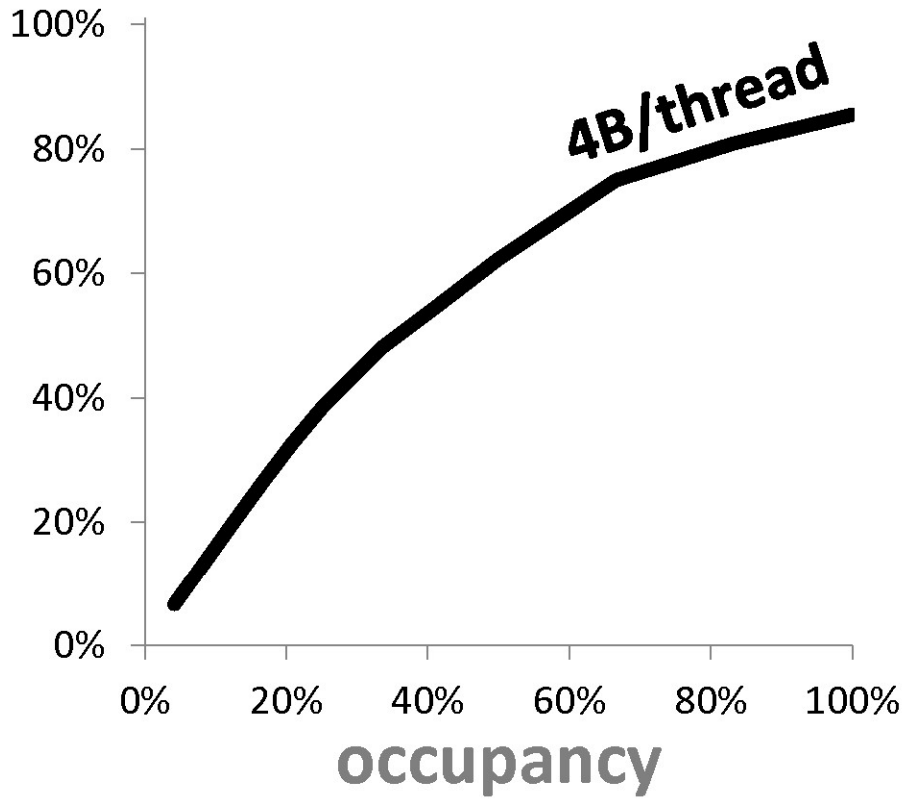
87% of pin bandwidth at only 8% occupancy!

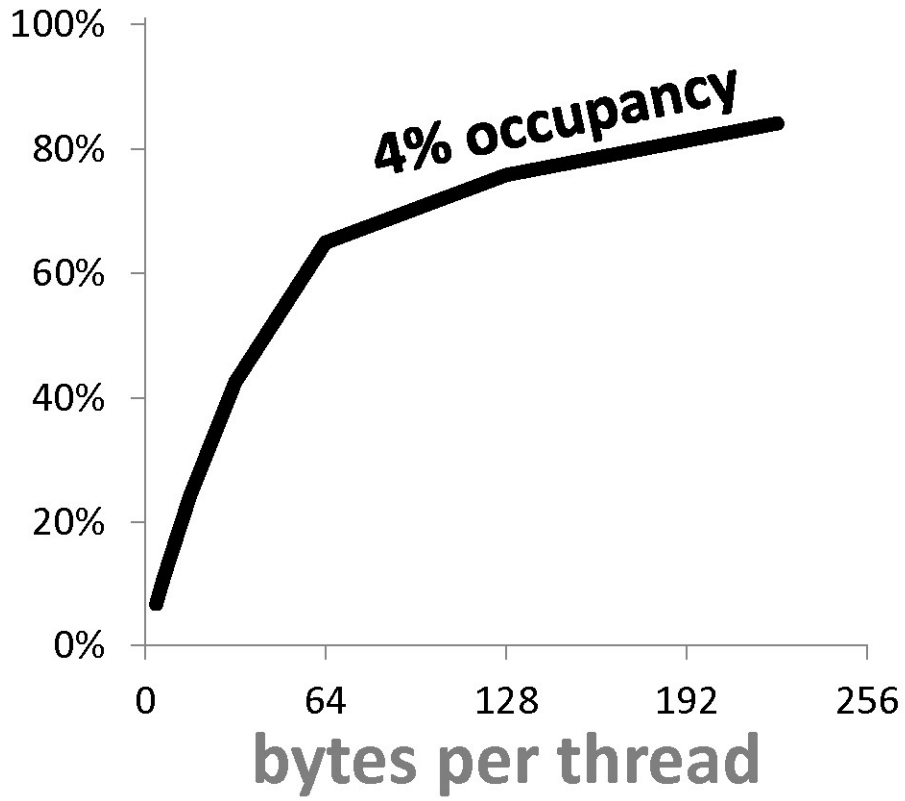
Copying 14 float4 values per thread



84% of peak at 4% occupancy

Two ways to hide memory latency





Fallacy:

"Low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation" (**CUDA Best Practices Guide**)

- *We've just seen 84% of the peak at mere 4% occupancy. Note that this is above 71% that cudaMemcpy achieves at best.*

Fallacy:

"In general, more warps are required if the ratio of the number of instructions with no off-chip memory operands (…) to the number of instructions with off-chip memory operands is low." (**CUDA Programming Guide**)

- *No, we've seen 87% of memory peak with only 4 warps per SM in a memory intensive kernel.*

Part III:

Run faster by using fewer threads

Fewer threads = more registers per thread

Registers per thread:

GF100: **20** at 100% occupancy, **63** at 33% occupancy — **3x**

GT200: **16** at 100% occupancy, ≈**128** at 12.5% occupancy — **8x**

Is using more registers per thread better?

Only registers are fast enough to get the peak

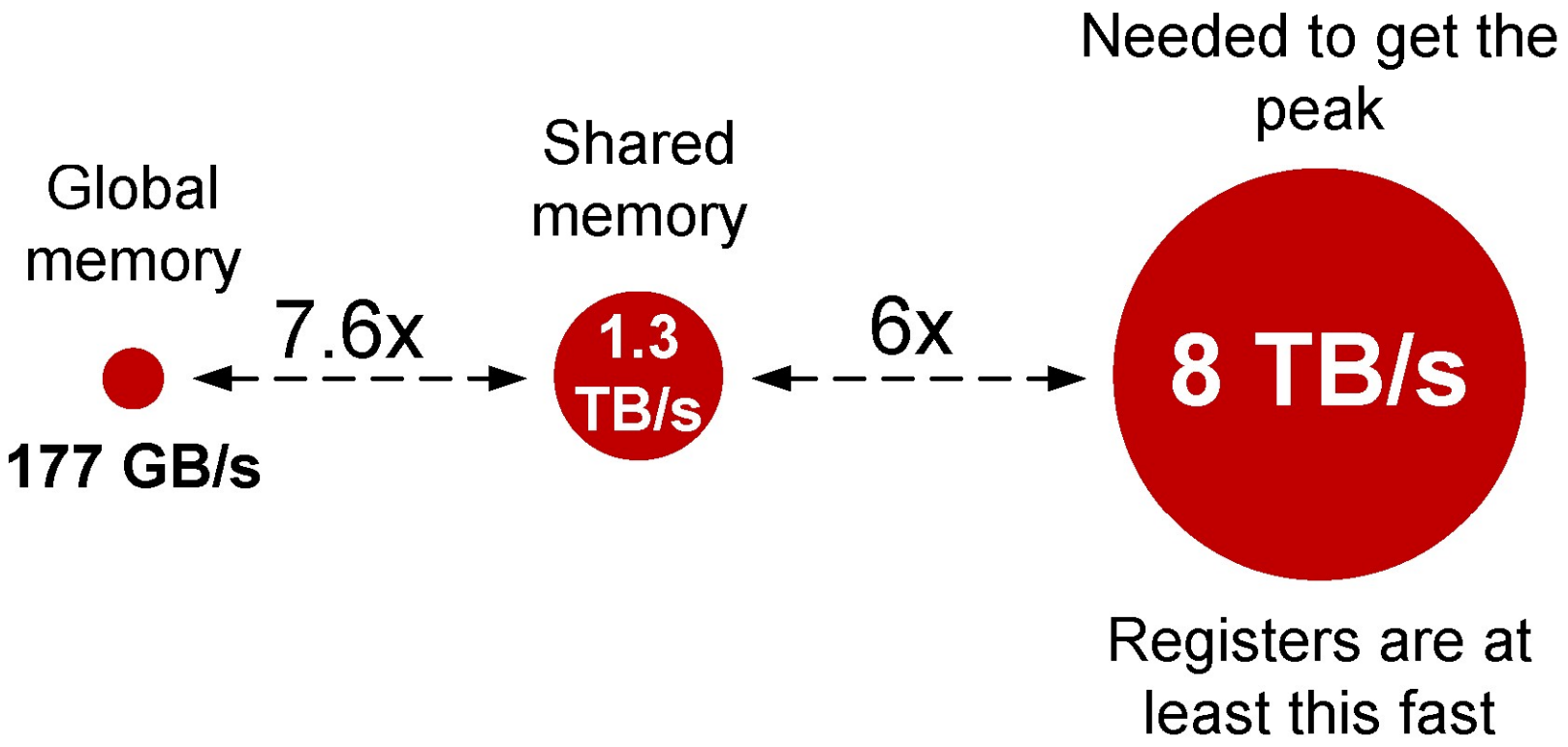
Consider a*b+c: 2 flops, 12 bytes in, 4 bytes out

This is **8.1 TB/s** for 1.3 Tflop/s!

Registers can accommodate it. Can shared memory?

– 4B*32banks*15 SMs*half 1.4GHz = 1.3TB/s only

Bandwidth needed vs bandwidth available



Fallacy:

"In fact, for all threads of a warp, accessing the shared memory is as fast as accessing a register as long as there are no bank conflicts between the threads.." (**CUDA Programming Guide**)

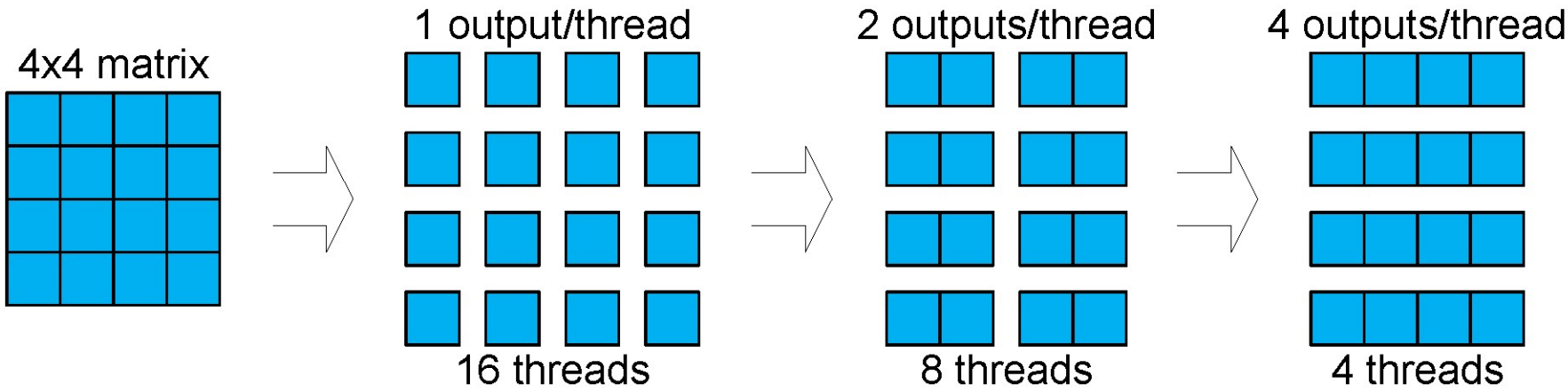
– *No, shared memory bandwidth is ≥6x lower than register bandwidth on Fermi. (≥3x before Fermi.)*

Running fast may require low occupancy

- ***Must* use registers to run close to the peak**
- The larger the bandwidth gap, the more data must come from registers
- This may require many registers = **low occupancy**

This often can be accomplished by *computing multiple outputs per thread*

Compute multiple outputs per thread



More data is local to a thread in registers

– *may* need fewer shared memory accesses

Fewer threads, but more parallel work in thread

– So, low occupancy should not be a problem

From Tesla to Fermi: regression?

The gap between shared memory and arithmetic throughput has increased:

- G80-GT200: **16** banks vs **8** thread processors (**2:1**)
- GF100: **32** banks vs **32** thread processors (**1:1**)
- GF104: **32** banks vs **48** thread processors (**2:3**)

Using fast register memory could help. But instead, register use is restricted:

- G80-GT200: up to ≈128 registers per thread
- Fermi: up to ≈64 registers per thread

Part IV:

Case study: matrix multiply

Baseline: matrix multiply in CUDA SDK

- I'll show very specific steps for SDK 3.1, GTX480
- Original code shows 137 Gflop/s
- First few changes:
  - Use larger matrices, e.g. 1024x1024 (matrixMul.cu)
    - "uiWA = uiHA = uiWB = uiHB = uiWC = uiHC = 1024 ; "
    - Get 240 Gflop/s
  - Remove "–maxrregcount 32" (or increase to 63)
    - Not important now, but will matter later
  - Increase BLOCK_SIZE to 32 (matrixMul.h)
    - Must add #pragma unroll (see next slide); 242 Gflop/s

Matrix multiply example in SDK

```
float Csub = 0;
for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    __syncthreads();

#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += AS(ty, k) * BS(k, tx);
    __syncthreads();
}
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```

Baseline performance

- One output per thread so far
- 242 Gflop/s
  - 2 flops per 2 shared memory accesses = 4 B/flop
  - So, bound by shared memory bandwidth to 336 Gflop/s
  - We'll approach 500 Gflop/s in a few slides
- 21 register per thread (sm_20)
- 67% occupancy
- But only 1 block fits per SM
  - Can't overlap global memory access with arithmetic

Two outputs per thread (I)

In the new code we use 2x smaller thread blocks

– But same number of blocks

matrixMul.cu:

```
// setup execution parameters
dim3 threads(BLOCK_SIZE, BLOCK_SIZE/2); //32x16
dim3 grid(uiWC / BLOCK_SIZE, uiHC / BLOCK_SIZE);
```

2x fewer threads, but 2x more work per thread:

Two outputs per thread (II)

```
float Csub[2] = {0,0};//array is allocated in registers
for (int a = aBegin, b = bBegin; a <= aEnd;
                     a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    AS(ty+16, tx) = A[a + wA * (ty+16) + tx];
    BS(ty+16, tx) = B[b + wB * (ty+16) + tx];
    __syncthreads();
```

Define 2 outputs and do 2x more loads

Two outputs per thread (III)

```
#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub[0] += AS(ty, k) * BS(k, tx);
        Csub[1] += AS(ty+16, k) * BS(k, tx);
    }
    __syncthreads();
}
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+16) + tx] = Csub[1];
```

Do 2x more flops and stores

Two outputs per thread: performance

- Now 341 Gflop/s — **1.4x speedup**
  - Already above 336 Gflop/s bound
- 28 registers
  - 2x more work with only 1.3x more registers
- Now 2 threads blocks fit per SM
  - Because fewer threads per block, 1536 max per SM
  - Now can overlap memory access with arithmetic
  - This is one reason for the speedup
- Same 67% occupancy

Shared memory traffic is now lower

- Data fetched from shared memory is now **reused**:

```
for (int k = 0; k < BLOCK_SIZE; ++k)
{
    Csub[0] += AS(ty, k) * BS(k, tx);
    Csub[1] += AS(ty+16, k) * BS(k, tx);
}
```

- Now 3B/flop in shared memory accesses
- New bound: 448 Gflop/s
  - We'll surpass this too

Four outputs per thread (I)

Apply same idea again

Shrink thread blocks by another factor of 2:

```
// setup execution parameters
dim3 threads(BLOCK_SIZE, BLOCK_SIZE/4); //32x8
dim3 grid(uiWC / BLOCK_SIZE, uiHC / BLOCK_SIZE);
```

Four outputs per thread (II)

```
float Csub[4] = {0,0,0,0};//array is in registers
for (int a = aBegin, b = bBegin; a <= aEnd;
                    a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    AS(ty+8, tx) = A[a + wA * (ty+8) + tx];
    BS(ty+8, tx) = B[b + wB * (ty+8) + tx];
    AS(ty+16, tx) = A[a + wA * (ty+16) + tx];
    BS(ty+16, tx) = B[b + wB * (ty+16) + tx];
    AS(ty+24, tx) = A[a + wA * (ty+24) + tx];
    BS(ty+24, tx) = B[b + wB * (ty+24) + tx];
    __syncthreads();
```

Four outputs per thread (III)

```
#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub[0] += AS(ty, k) * BS(k, tx);
        Csub[1] += AS(ty+8, k) * BS(k, tx);
        Csub[2] += AS(ty+16, k) * BS(k, tx);
        Csub[3] += AS(ty+24, k) * BS(k, tx);
    }
    __syncthreads();
}
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+8) + tx] = Csub[1];
C[c + wB * (ty+16) + tx] = Csub[2];
C[c + wB * (ty+24) + tx] = Csub[3];
```

Four outputs per thread: performance

- Now 427 Gflop/s — 1.76x speedup vs. baseline!
  - Because access shared memory even less
- 41 registers
  - Only ≈2x more registers
  - So, ≈2x **fewer** registers per thread block
- 50% occupancy — 1.33x lower
  - **Better performance at lower occupancy**
- 3 thread blocks per SM
  - Because fewer registers per thread block

Eight outputs per thread: performance

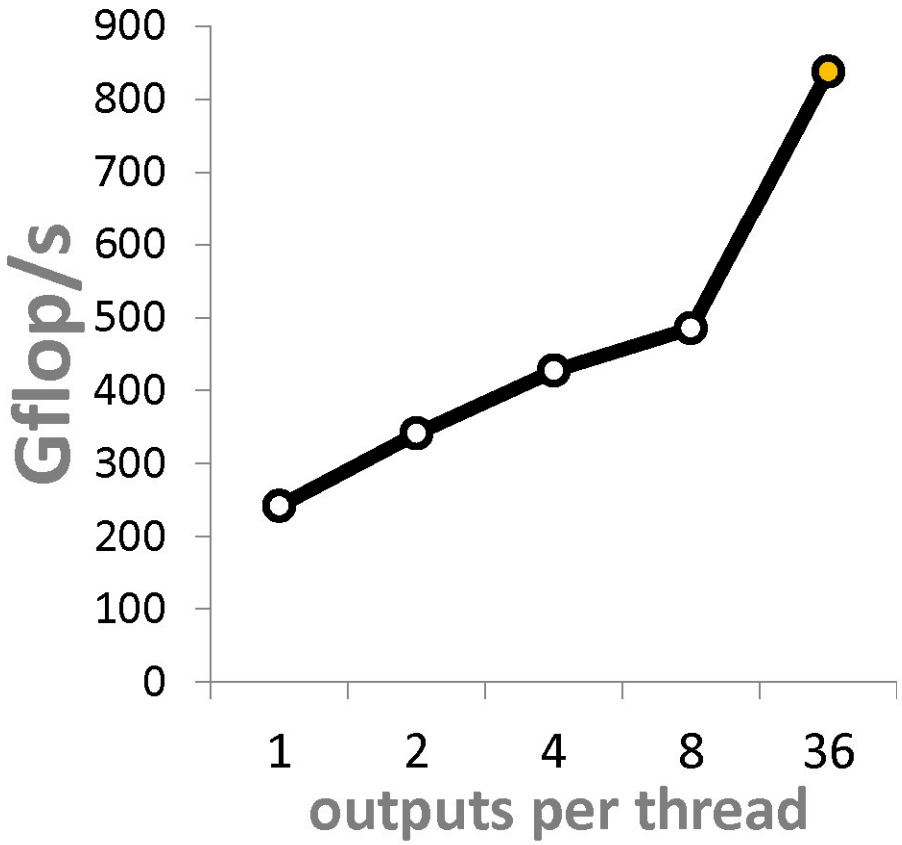
- Now 485 Gflop/s — 2x speedup vs. baseline!
  - Only 2.25 B/flop — 1.8x lower
- 63 registers — 3x more
  - But do 8x more work!
- 33% occupancy — 2x lower
  - **Better performance at lower occupancy**
- 4 thread blocks per SM

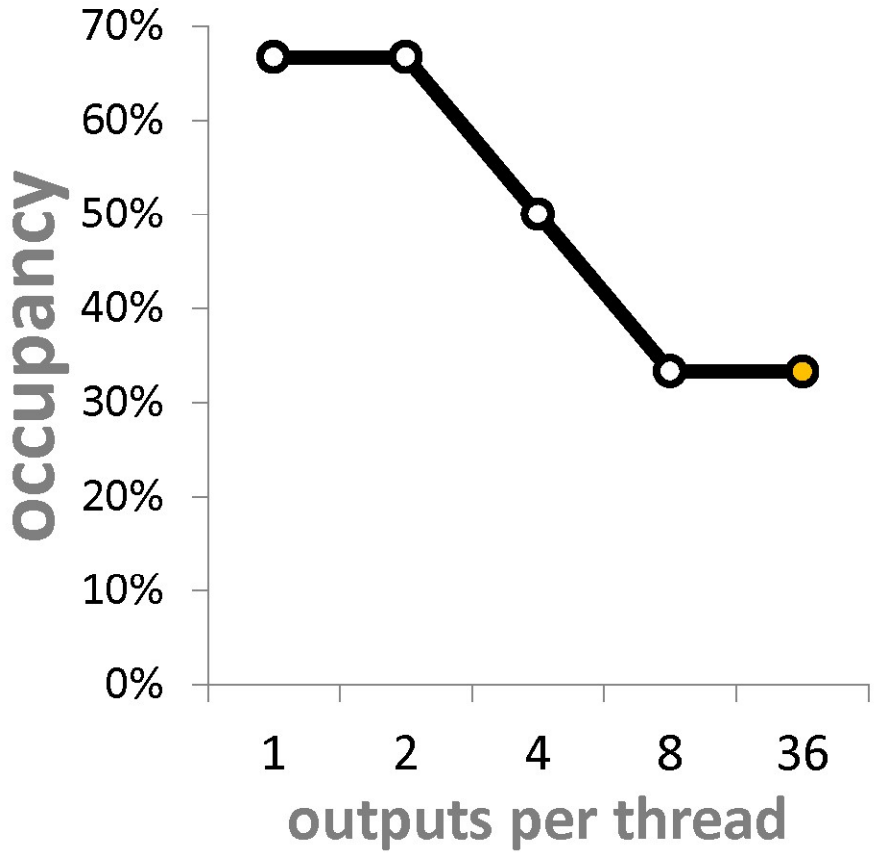
How much faster we can get?

MAGMA BLAS — up to 838 Gflop/s

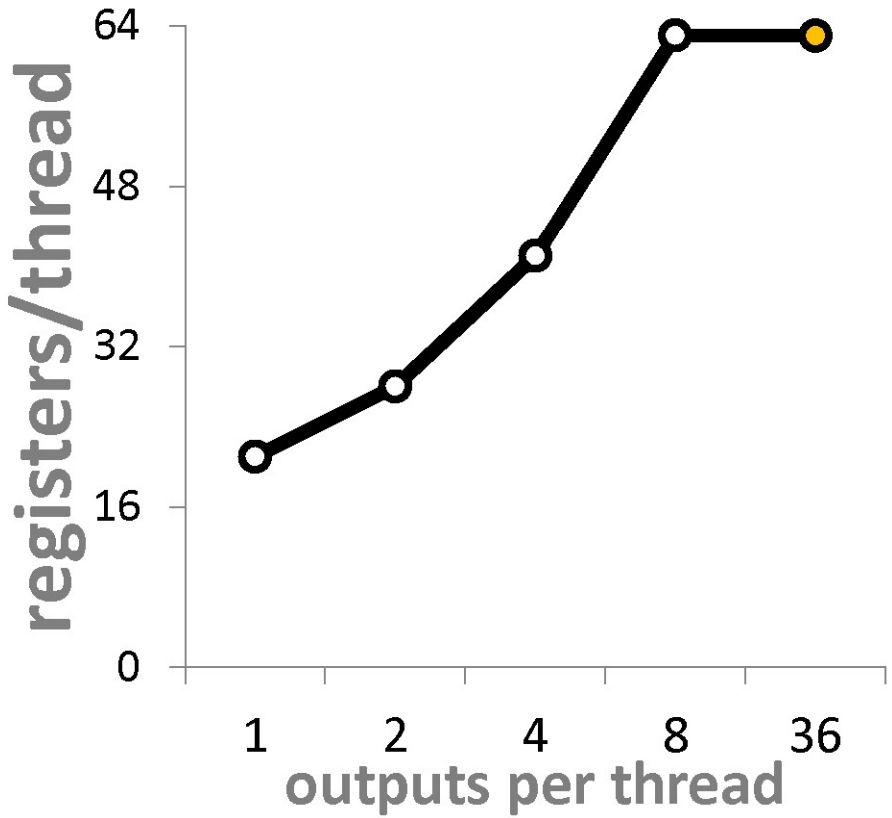
- 36 outputs per thread
- 0.67 B/flop only — 6x lower
- 33% occupancy
- 2 thread blocks per SM

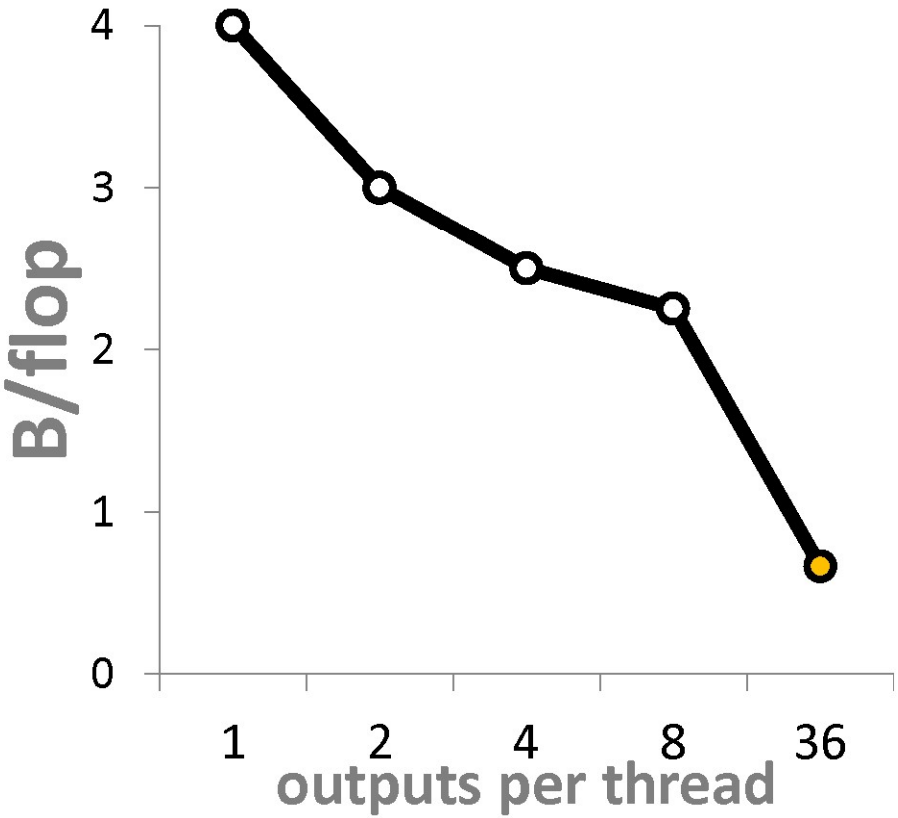
GFLOPS go up, occupancy goes down





Register use goes up, smem traffic down

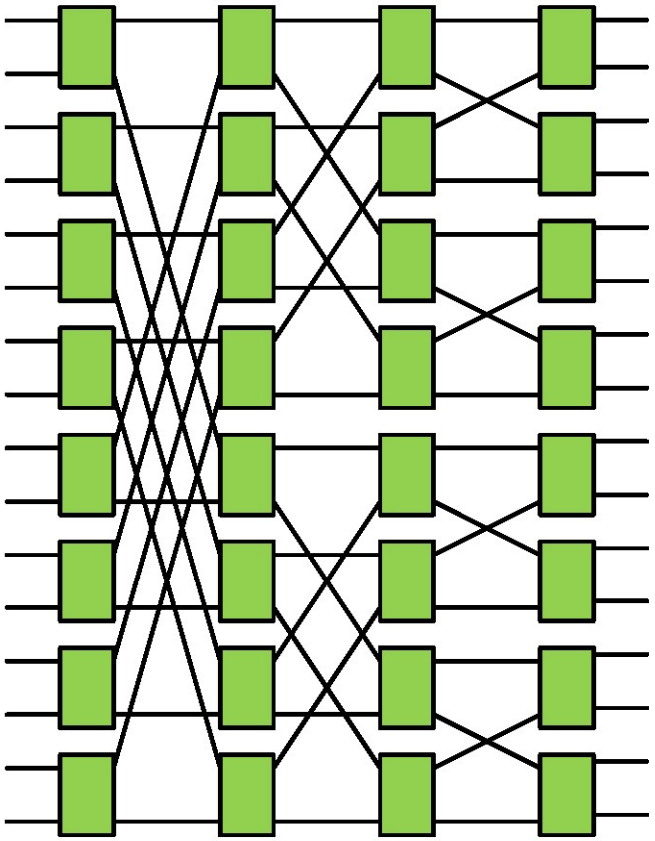




Part V:

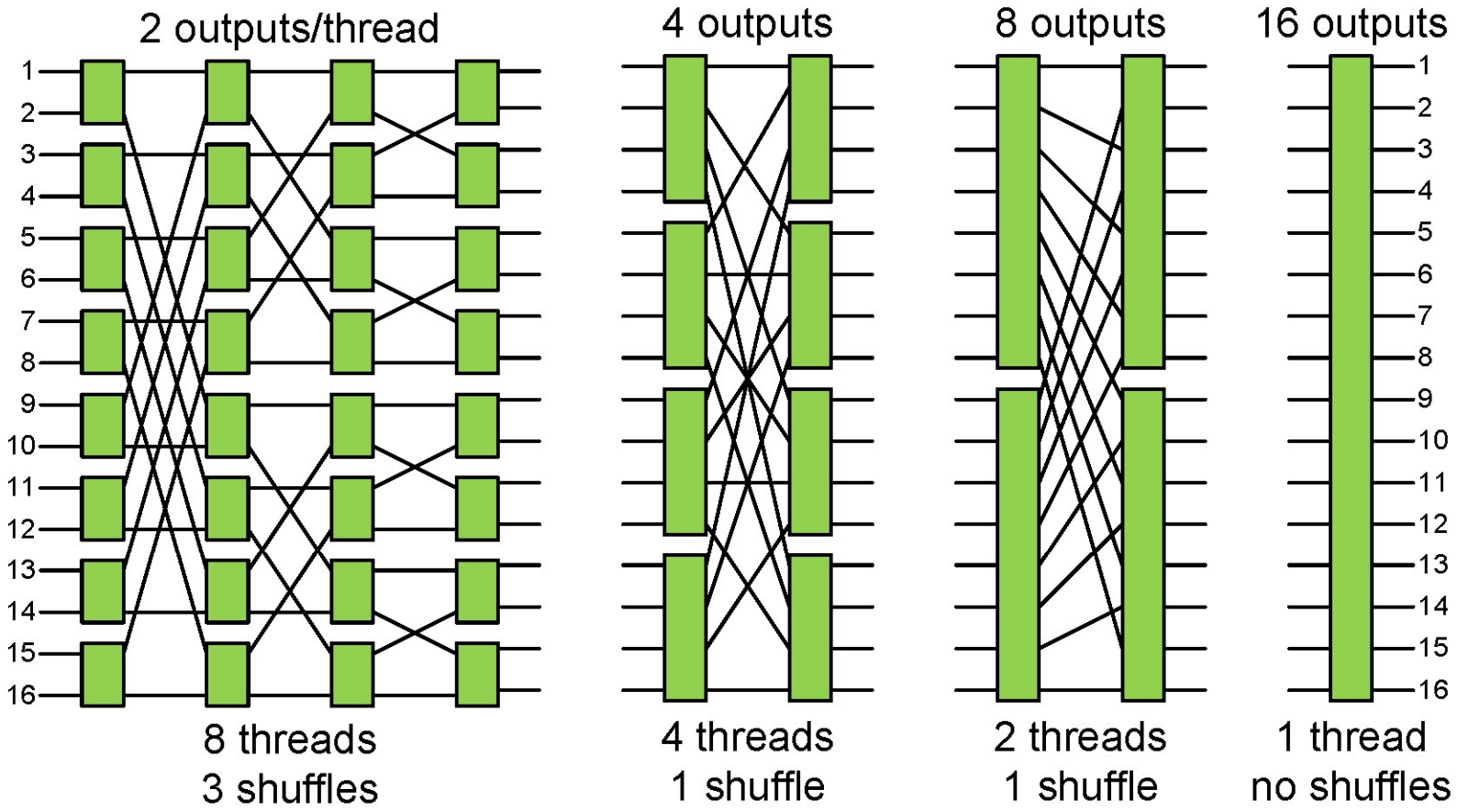
Case Study: FFT

Mapping Cooley-Tukey to GPU



- Cooley-Tukey splits large FFT into smaller FFTs
- Assume FFT fits into thread block
- Small FFT are done in registers
- Shuffles are done using shared memory

Fewer threads – lower shared memory traffic



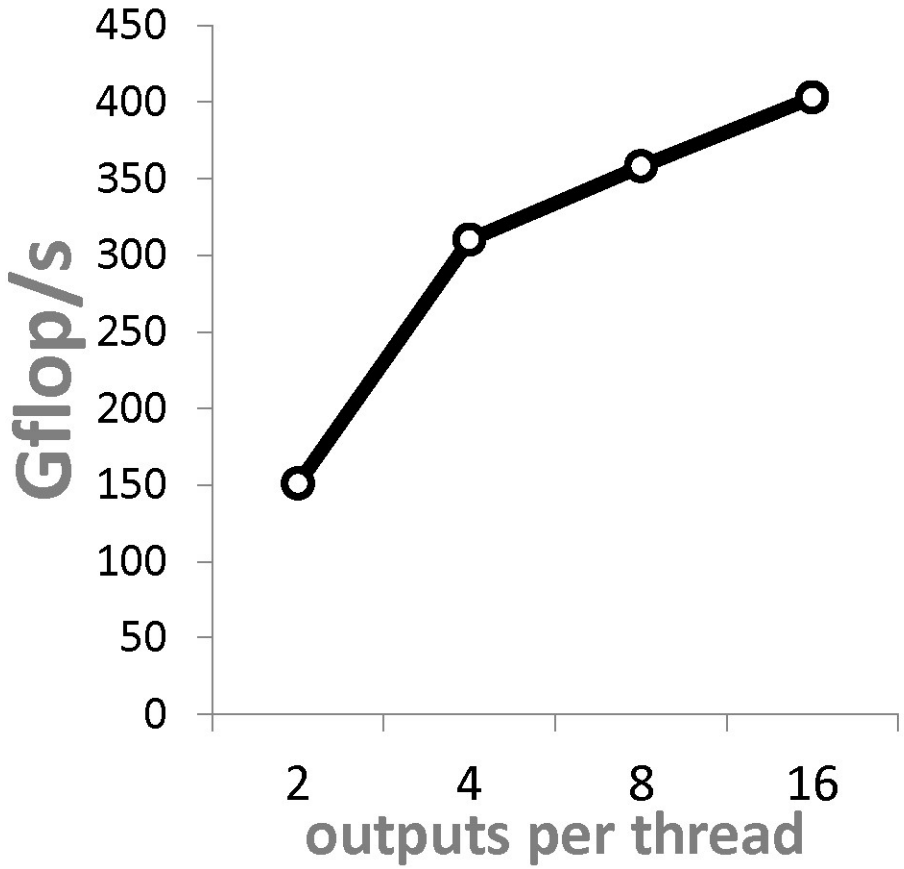
Two outputs per thread

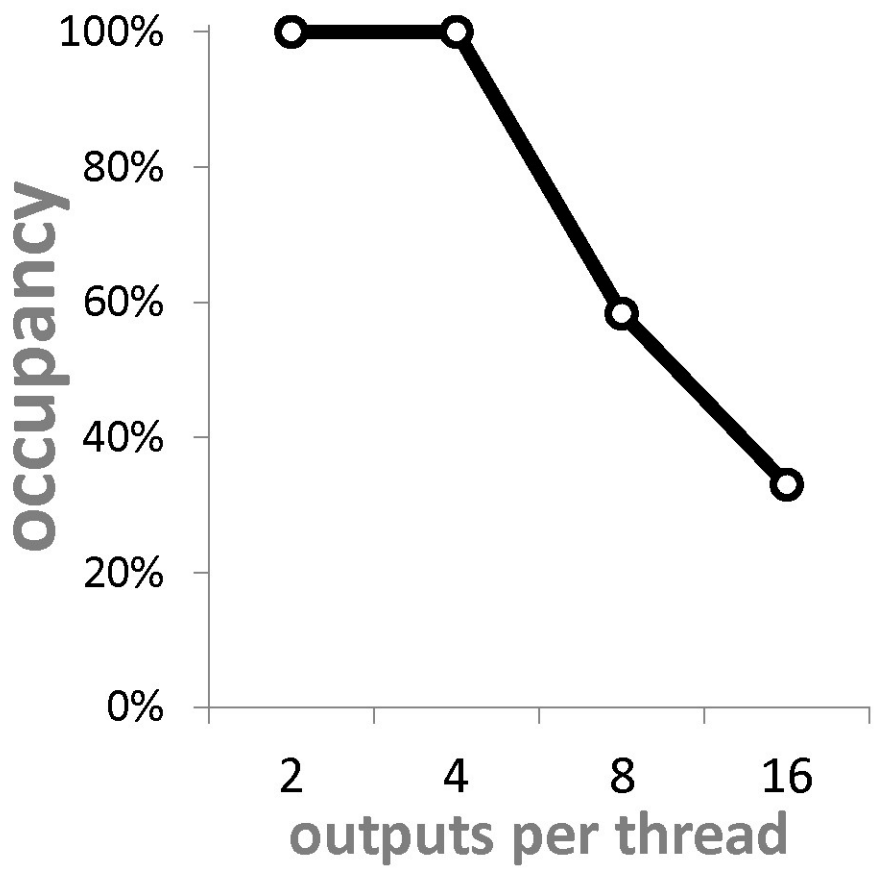
```
__global__ void FFT1024( float2 *dst, float2 *src ){
  float2 a[2]; int tid = threadIdx.x;
  __shared__ float smem[1024];
  load<2>( a, src+tid+1024*blockIdx.x, 512 );
  FFT2( a );
#pragma unroll
  for( int i = 0; i < 9; i++ ) {
     int k = 1<<i;
     twiddle<2>( a, tid/k, 1024/k );
     transpose<2>( a, &smem[tid+(tid&~(k-1))], k, &smem[tid], 512 );
     FFT2( a );
  }
  store<2>( a, dst+tid+1024*blockIdx.x, 512 );
}
```

Sixteen outputs per thread

```
__global__ void FFT1024( float2 *dst, float2 *src ){
   float2 a[16]; int tid = threadIdx.x;
   __shared__ float smem[1024];
   load<16>( a, src+tid+1024*blockIdx.x, 64 );
   FFT4( a, 4, 4, 1 );// four FFT4
   twiddle<4>( a, threadIdx.x, 1024, 4 );
   transpose<4>( a, &smem[tid*4], 1, &smem[tid], 64, 4 );
#pragma unroll
   for( int i = 2; i < 10-4; i += 4 ) {
     int k = 1<<i;
     FFT16( a );
     twiddle<16>( a, threadIdx.x/k, 1024/k );
     transpose<16>( a, &smem[tid+15*(tid&~(k-1))], k, &smem[tid], 64 );
   }
   FFT16( a );
   store<16>( a, dst+tid+1024*blockIdx.x, 64 );
}
```

GFLOPS go up, occupancy goes down

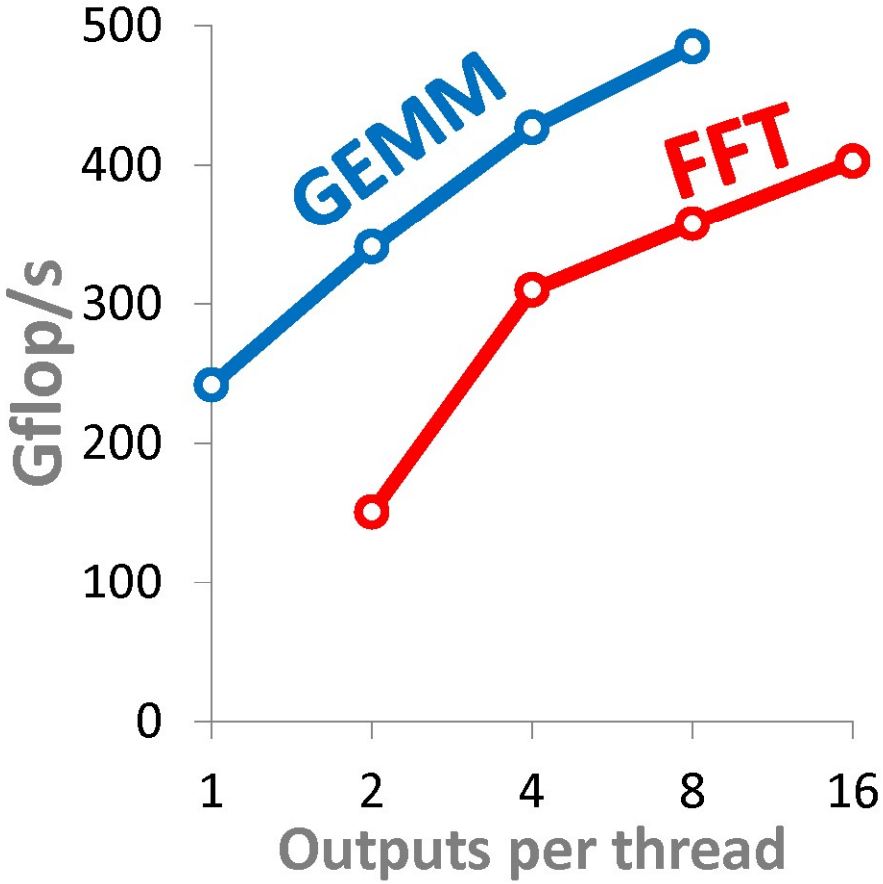


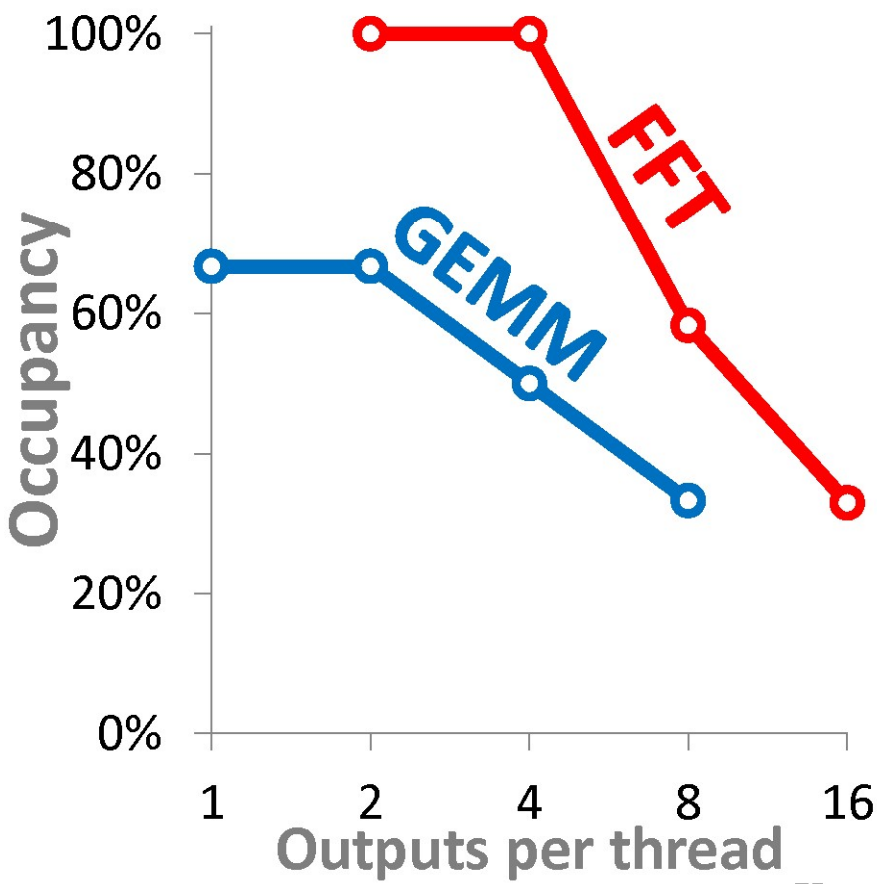


Summary

- Do more parallel work per thread to hide latency with fewer threads
- Use more registers per thread to access slower shared memory less
- Both may be accomplished by computing multiple outputs per thread

Compute more outputs per thread





Thank you.

- NVIDIA, Nikolay Sakharnykh
- Vasily Volkov