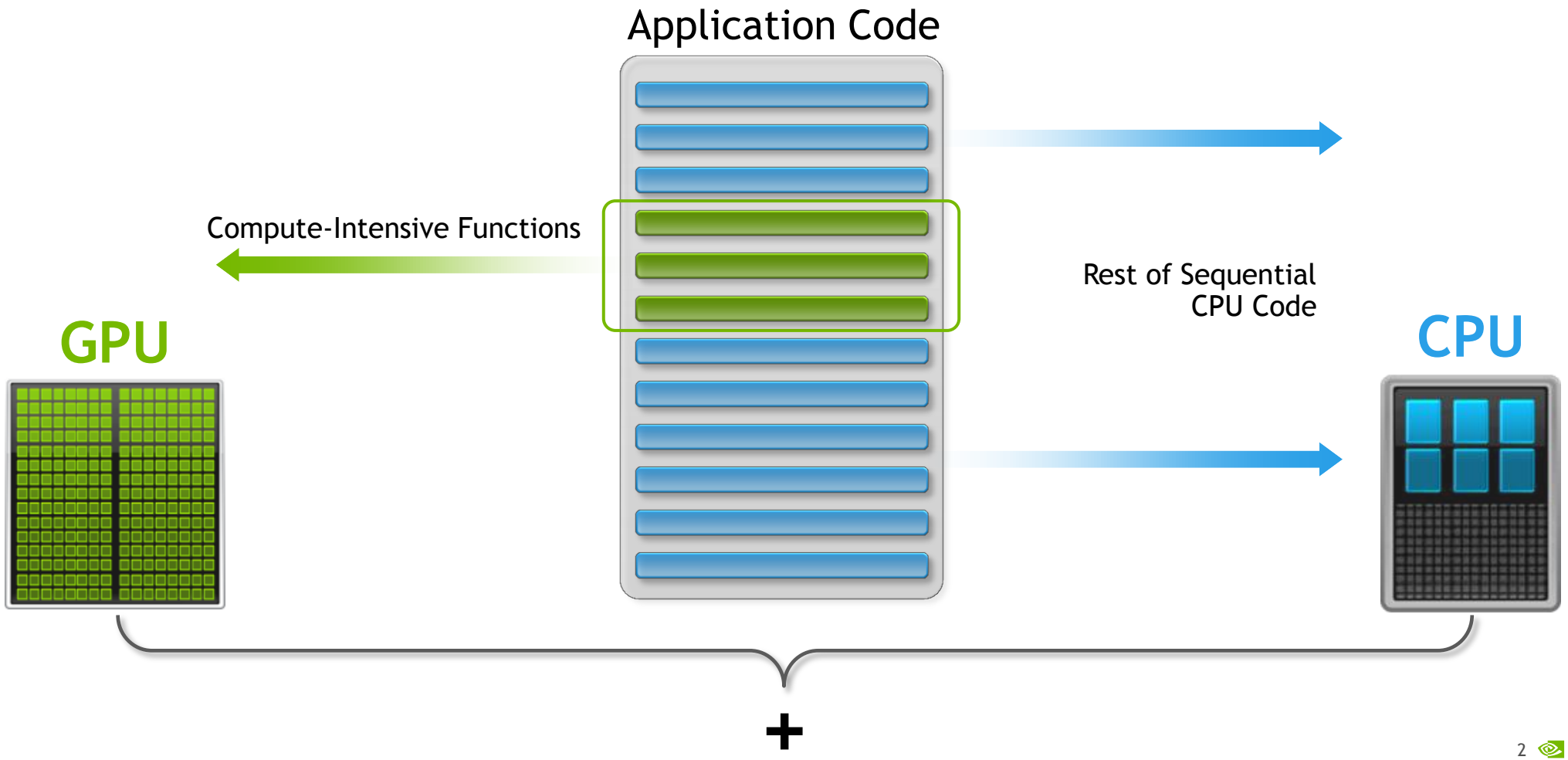
GPU ARCHITECTURES AND NEW PROGRAMMING MODEL FEATURES

Nikolay Sakharnykh, 8/1/2016

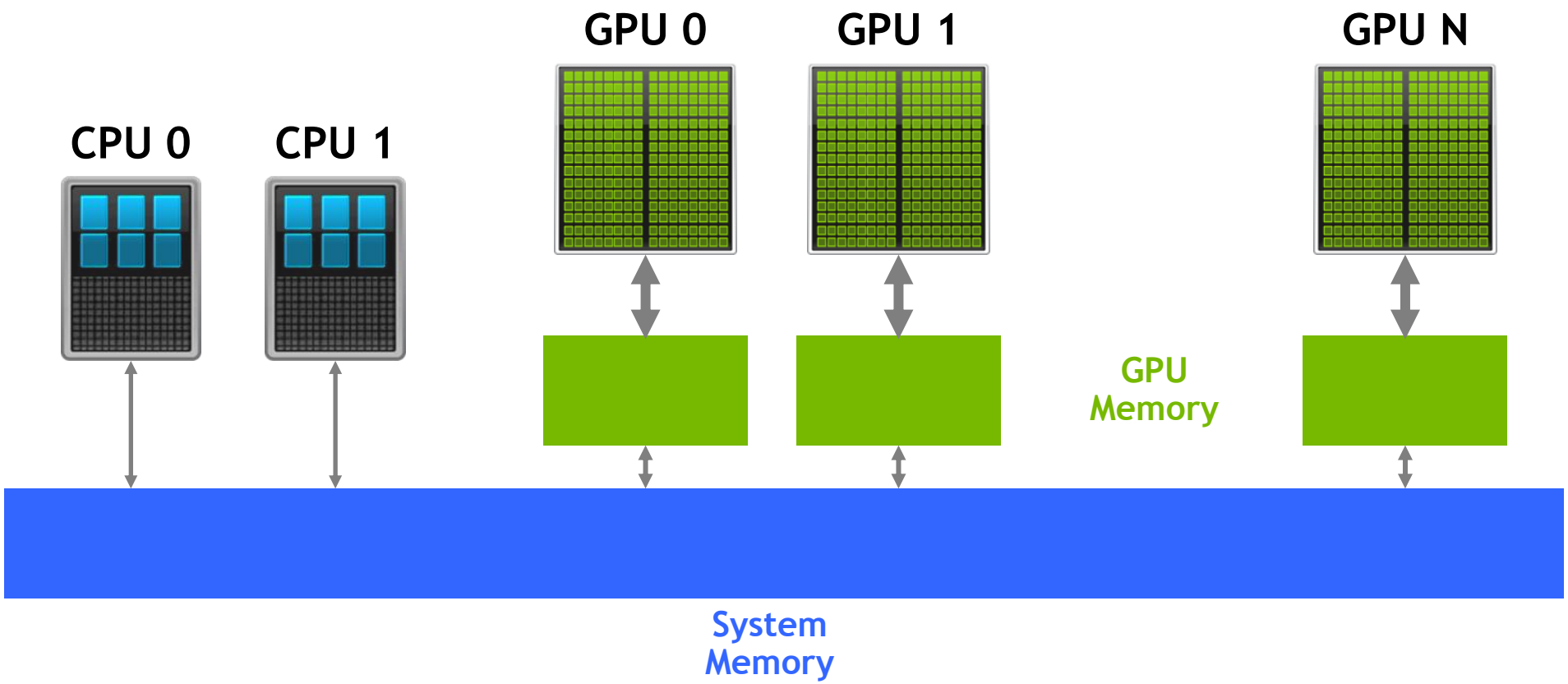
NVIDIA

HOW GPU ACCELERATION WORKS

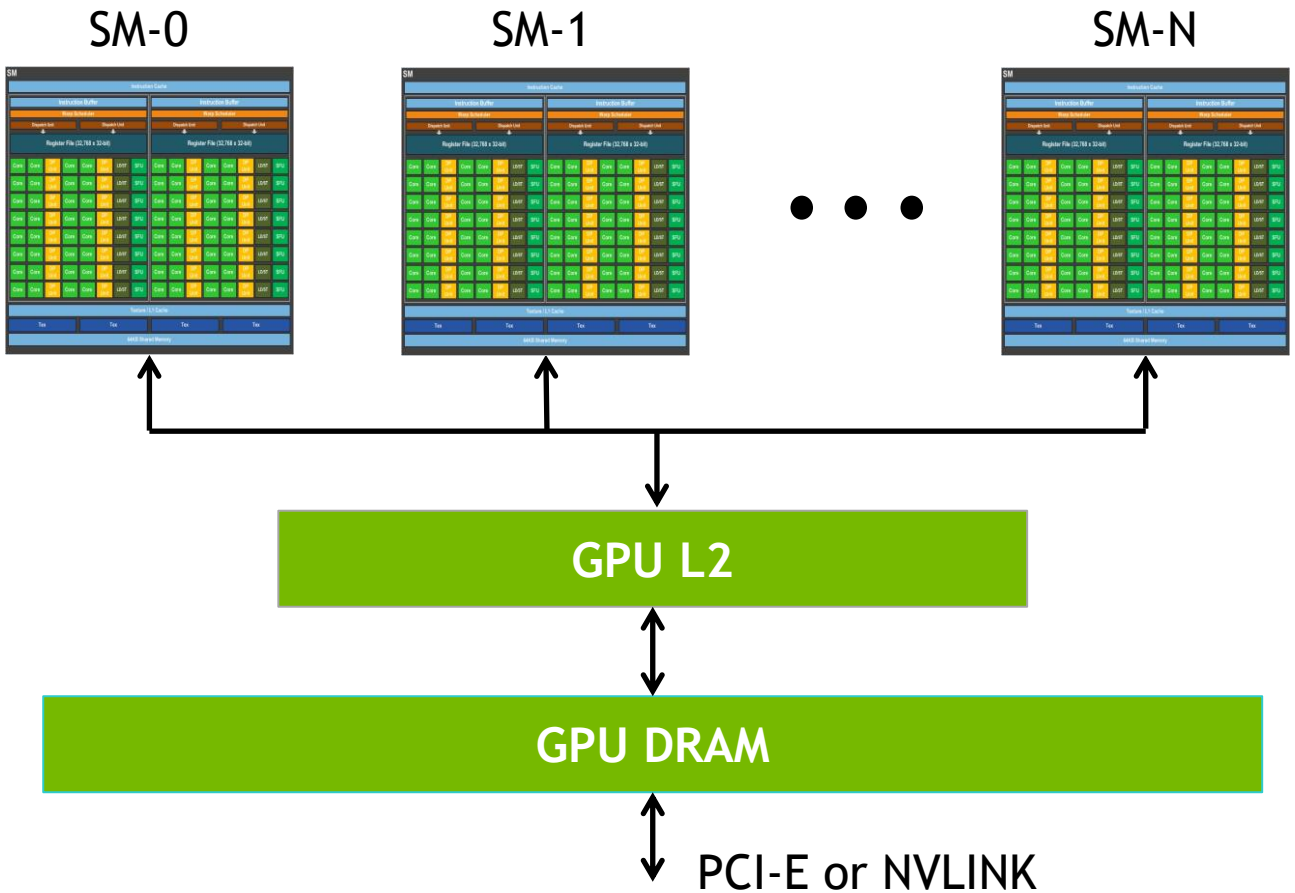


HETEROGENEOUS ARCHITECTURES

Memory hierarchy



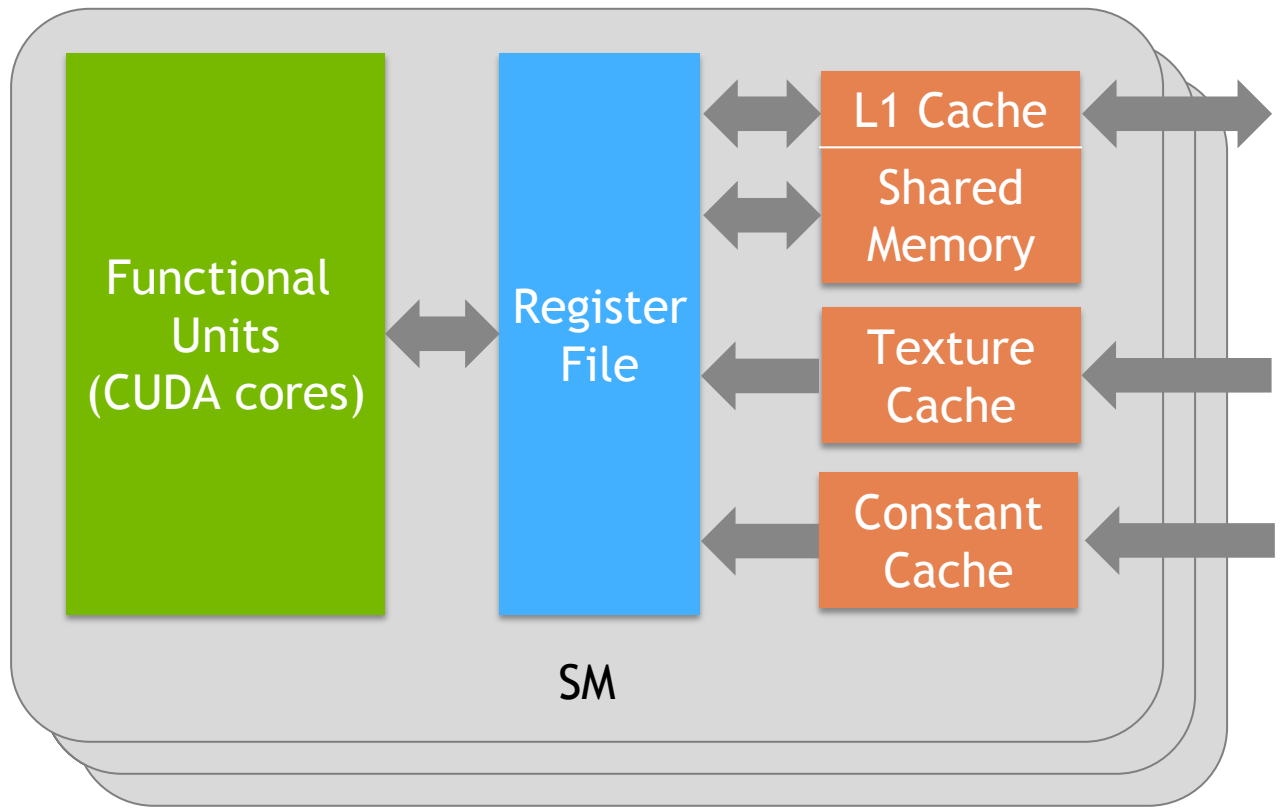
GPU ARCHITECTURE



GPU SM ARCHITECTURE

Kepler SM

	GK110
CUDA Cores	192
Register File	256 KB
Shared Memory	16-48 KB

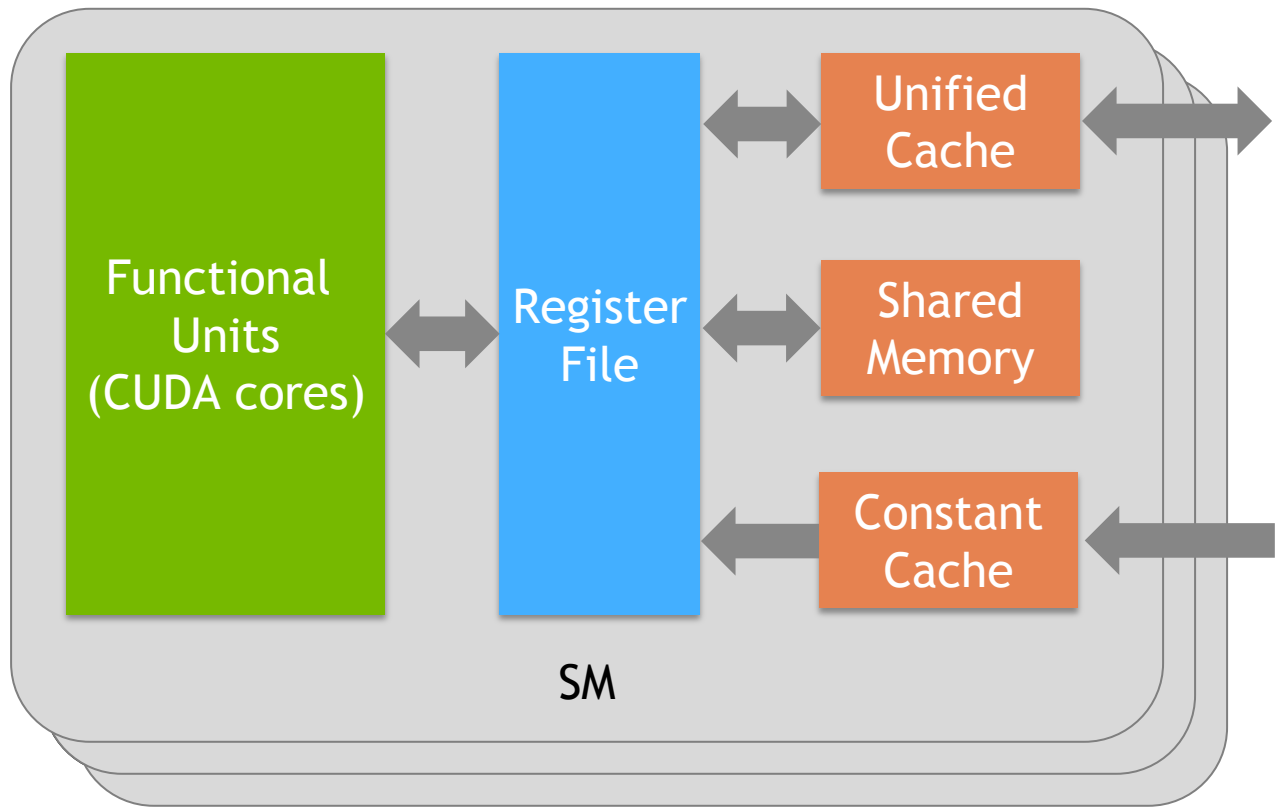


15 SMs on Tesla K40

GPU SM ARCHITECTURE

Maxwell SM

	GM200
CUDA Cores	128
Register File	256 KB
Shared Memory	96 KB

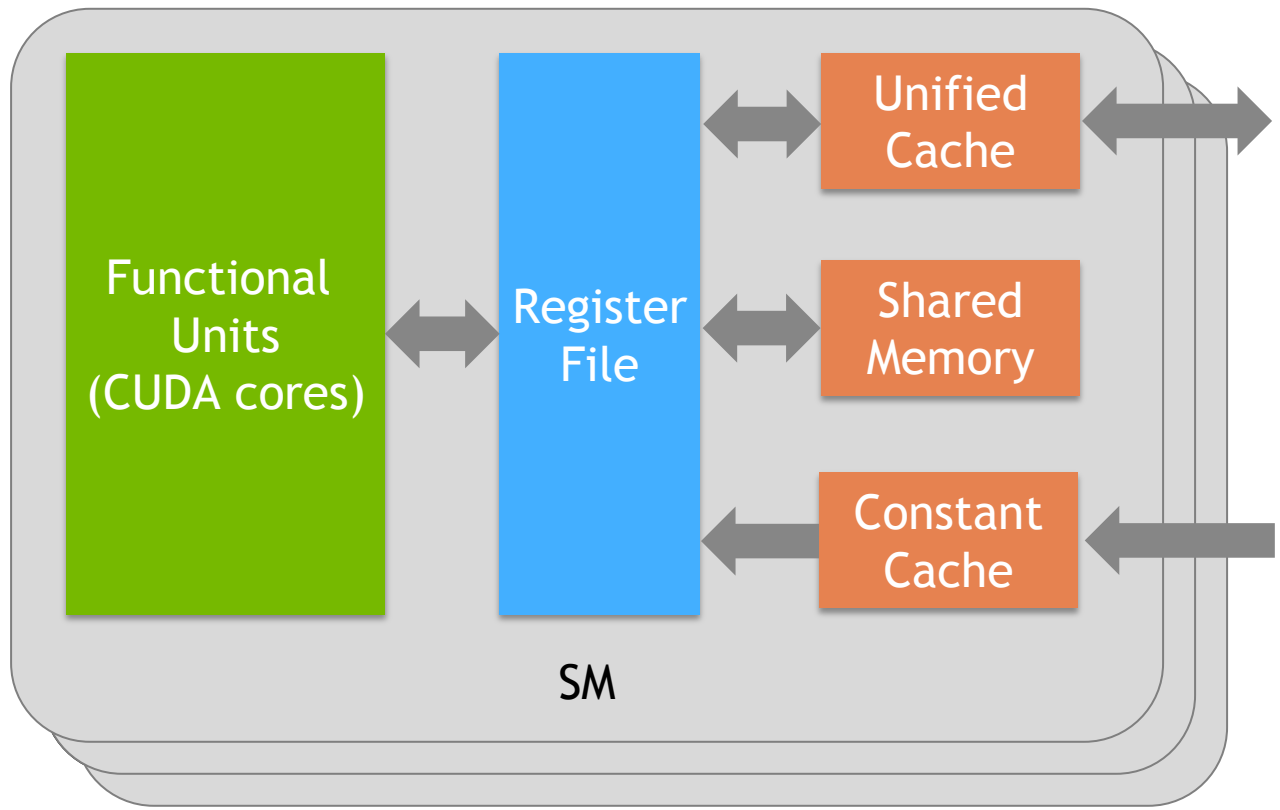


24 SMs on Tesla M40

GPU SM ARCHITECTURE

Pascal SM

	GP100
CUDA Cores	64
Register File	256 KB
Shared Memory	64 KB

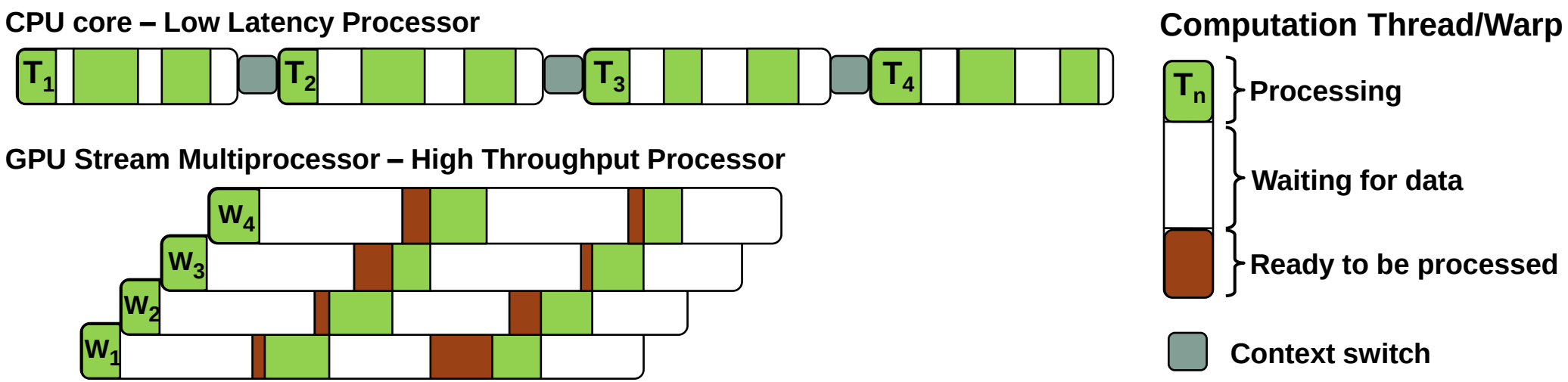


56 SMs on Tesla P100

LOW LATENCY OF HIGH THROUGHPUT?

CPU architecture must **minimize latency** within each thread

GPU architecture **hides latency** with computation from other threads (warps)



ACCELERATOR FUNDAMENTALS

Must expose enough parallelism to saturate the GPU

Accelerator threads are slower than CPU threads

Accelerators have orders of magnitude more threads

t0	t1	t2	t3
t4	t5	t6	t7
t8	t9	t10	t11
t12	t13	t14	t15

Fine-grained parallelism is good

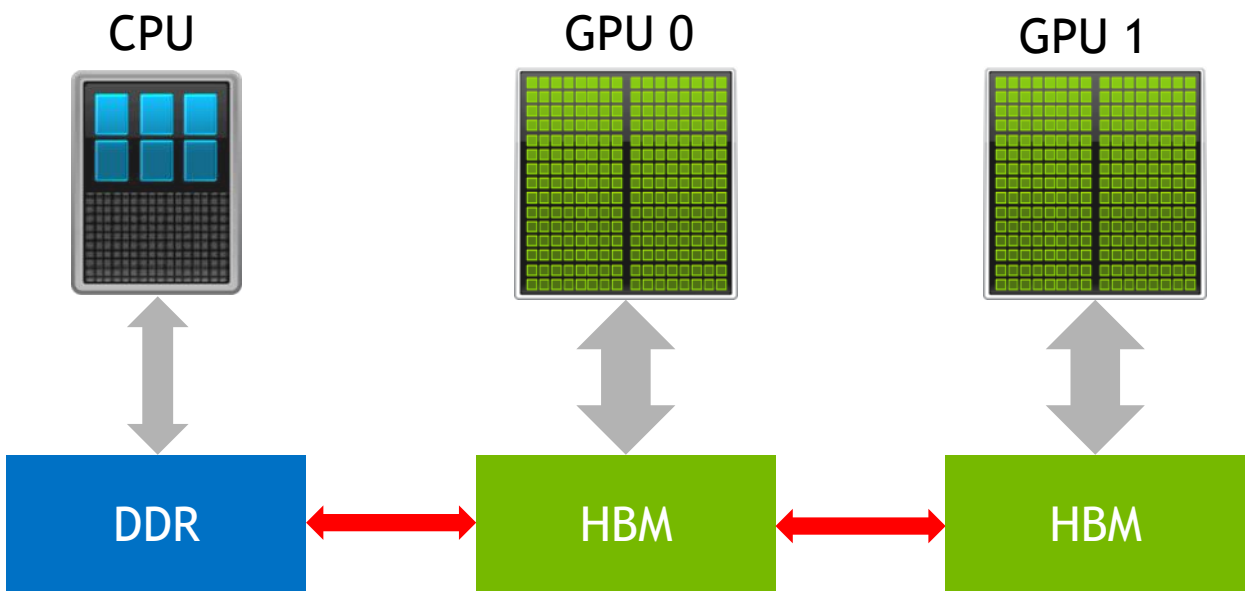
t0	t0	t0	t0
t1	t1	t1	t1
t2	t2	t2	t2
t3	t3	t3	t3

Coarse-grained parallelism is bad

BEST PRACTICES

Optimize data locality for CPU and GPU

Minimize data transfers between CPU and GPU, and between peer GPUs



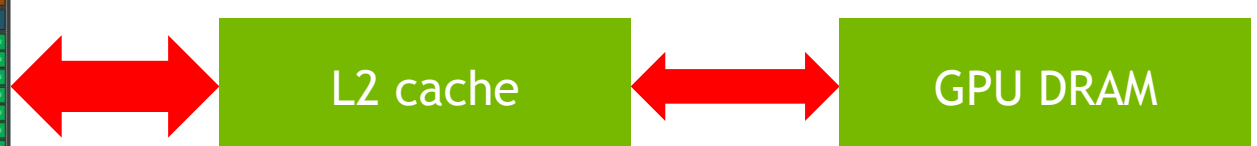
BEST PRACTICES

Optimize data locality for SM

Minimize redundant accesses to L2 and DRAM

- Store intermediate results in registers instead of DRAM
- Use shared memory for data frequently used within SM
- Use constant and read-only caches on SM

GPU SM

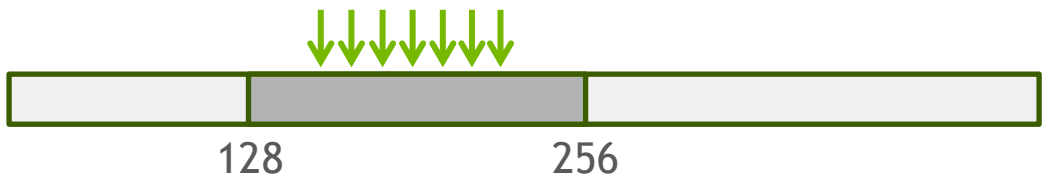


BEST PRACTICES

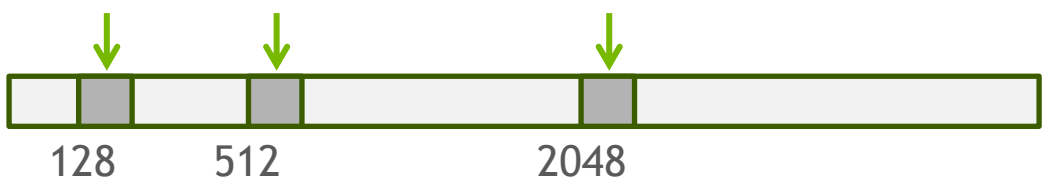
Coalesce memory requests

If addresses from a *warp* lie within the same cache line, that line is fetched once

Best case: addresses lie in a single cache line (128B), **4**x32B transactions

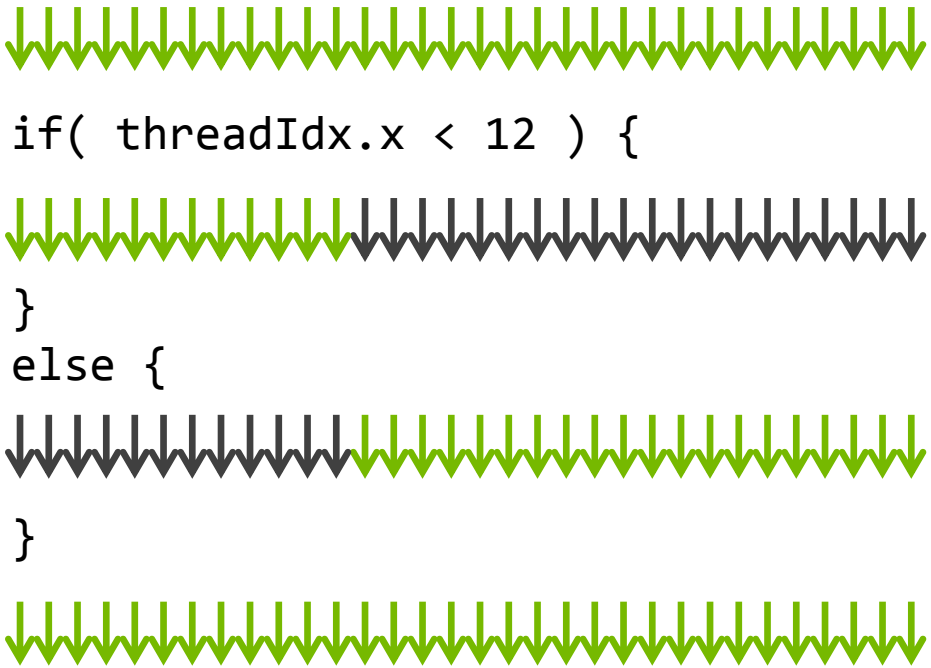


Worst case: fully scattered access, 32 allocated cache lines, **32**x32B transactions



BEST PRACTICES

Avoid warp divergence



Instructions are issued per warp

Different execution paths within a warp are serialized

Different warps can execute different code with no impact on performance

Avoid branching on thread index

BEST PRACTICES

Other common optimizations

Minimize thread block synchronization

Expose instruction-level parallelism

Use 64-bit and 128-bit vector loads

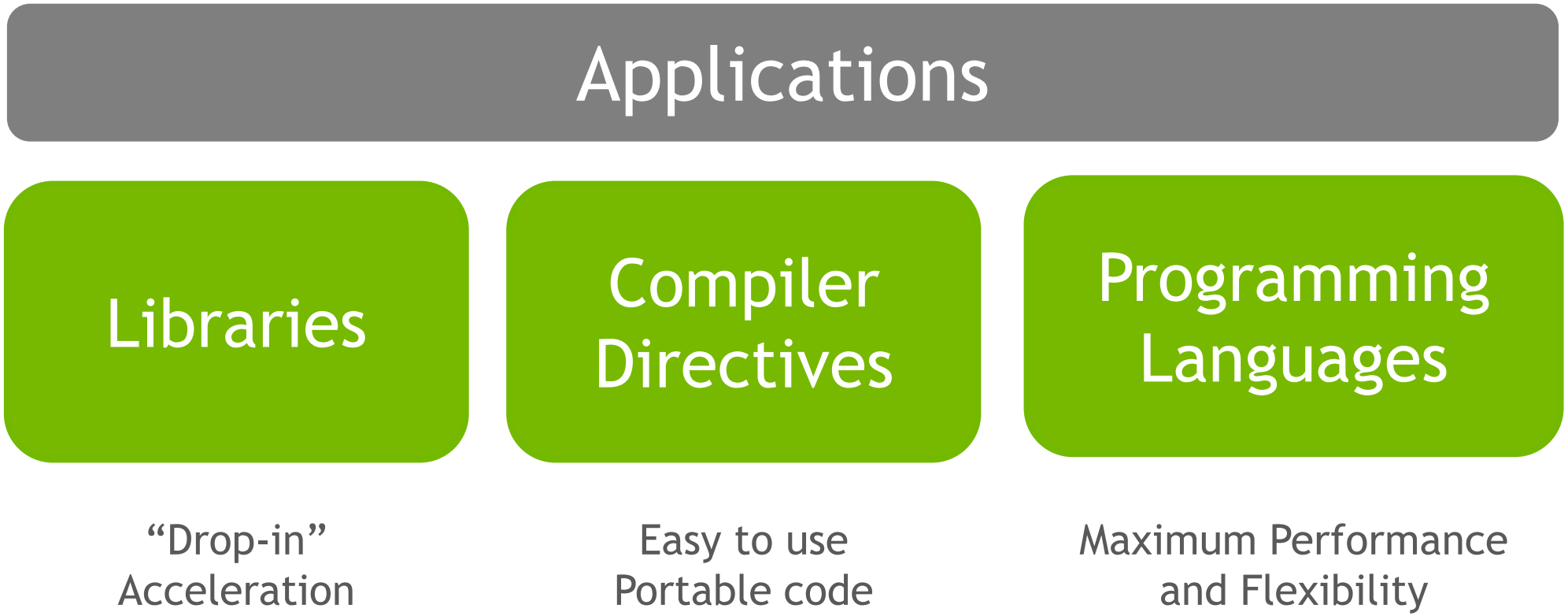
Control occupancy with compiler hints

Tile computation for better cache reuse

Use mixed or reduced precision

PROGRAMMING GPUS

3 WAYS TO PROGRAM GPUS

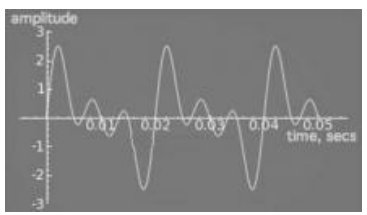


OpenACC hands-on session today 7:30pm - 9:30pm

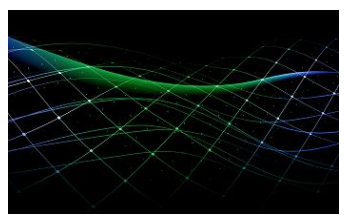
NVIDIA DEVELOPER LIBRARIES

cuBLAS
cuBLAS-XT
NVBLAS

cuFFT
cuFFT-XT

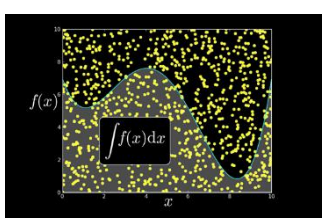


cuSPARSE
cuSOLVER
AMGX



cuDNN

cuRAND

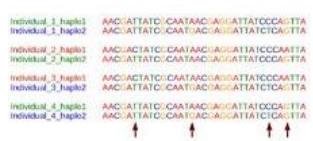


THRUST

NPP

NVENC

NVBIO



https://developer.nvidia.com/gpu-accelerated-libraries

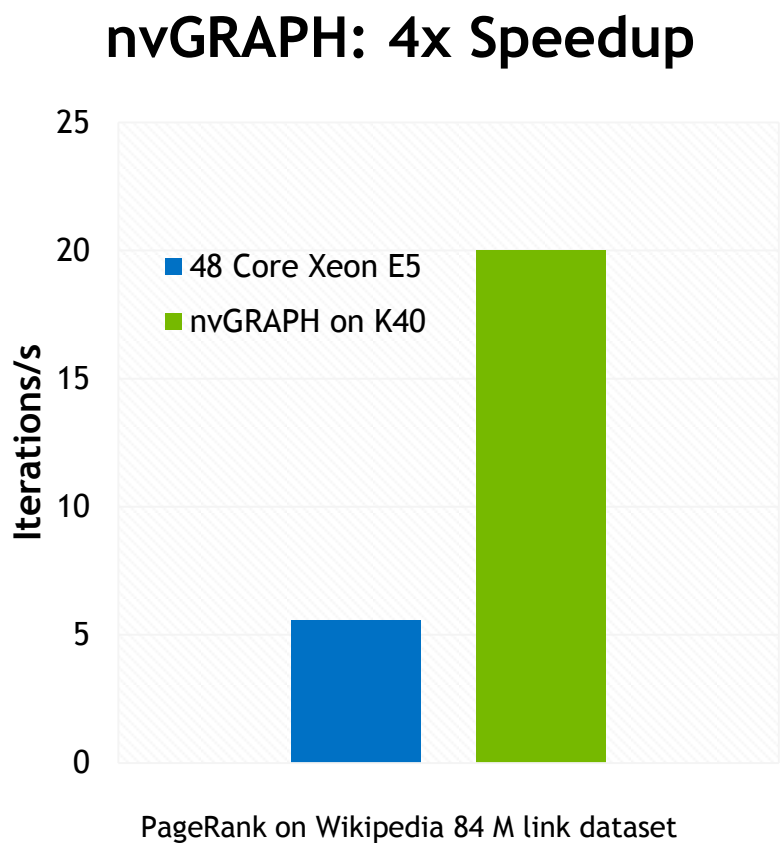
NVGRAPH

Accelerated Graph Analytics

Process graphs with up to 2.5 Billion edges on a single GPU (24GB M40)

Accelerate a wide range of applications:

PageRank	Single Source Shortest Path	Single Source Widest Path
Search	Robotic Path Planning	IP Routing
Recommendation Engines	Power Network Planning	Chip Design / EDA
Social Ad Placement	Logistics & Supply Chain Planning	Traffic sensitive routing

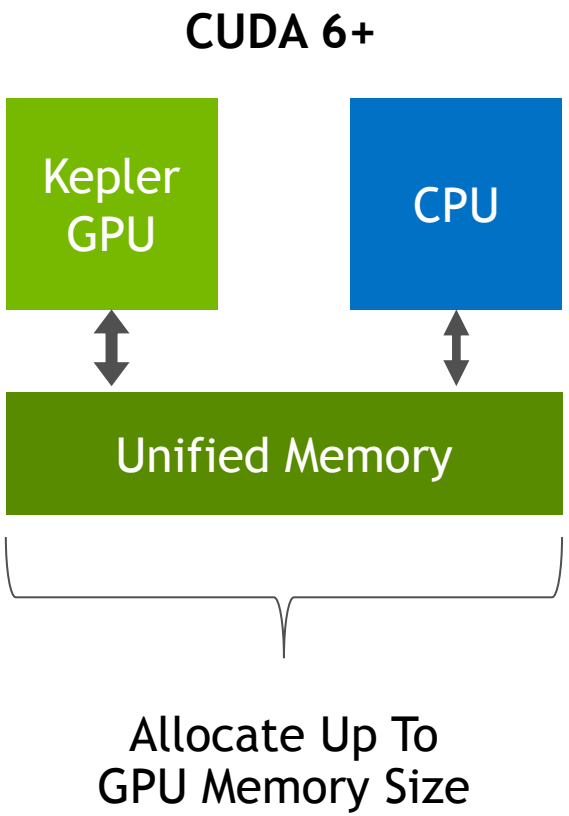


https://developer.nvidia.com/nvgraph

UNIFIED MEMORY

UNIFIED MEMORY

Dramatically Lower Developer Effort



Simpler Programming & Memory Model

- Single allocation, single pointer, accessible anywhere
- Eliminate need for *explicit copy*
- Greatly simplifies code porting

Performance Through Data Locality

- Migrate data to accessing processor
- Guarantee global coherence
- Still allows explicit hand tuning

SIMPLIFIED MEMORY MANAGEMENT CODE

Single pointer for CPU and GPU

CPU code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

GPU code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

UNIFIED MEMORY ON PRE-PASCAL

Code example explained

```
cudaMallocManaged(&ptr, ...);
*ptr = 1;
qsort<<<...>>>(ptr);
```

← Pages are populated in GPU memory

← **CPU page fault**: data migrates to CPU

← **Kernel launch**: data migrates to GPU

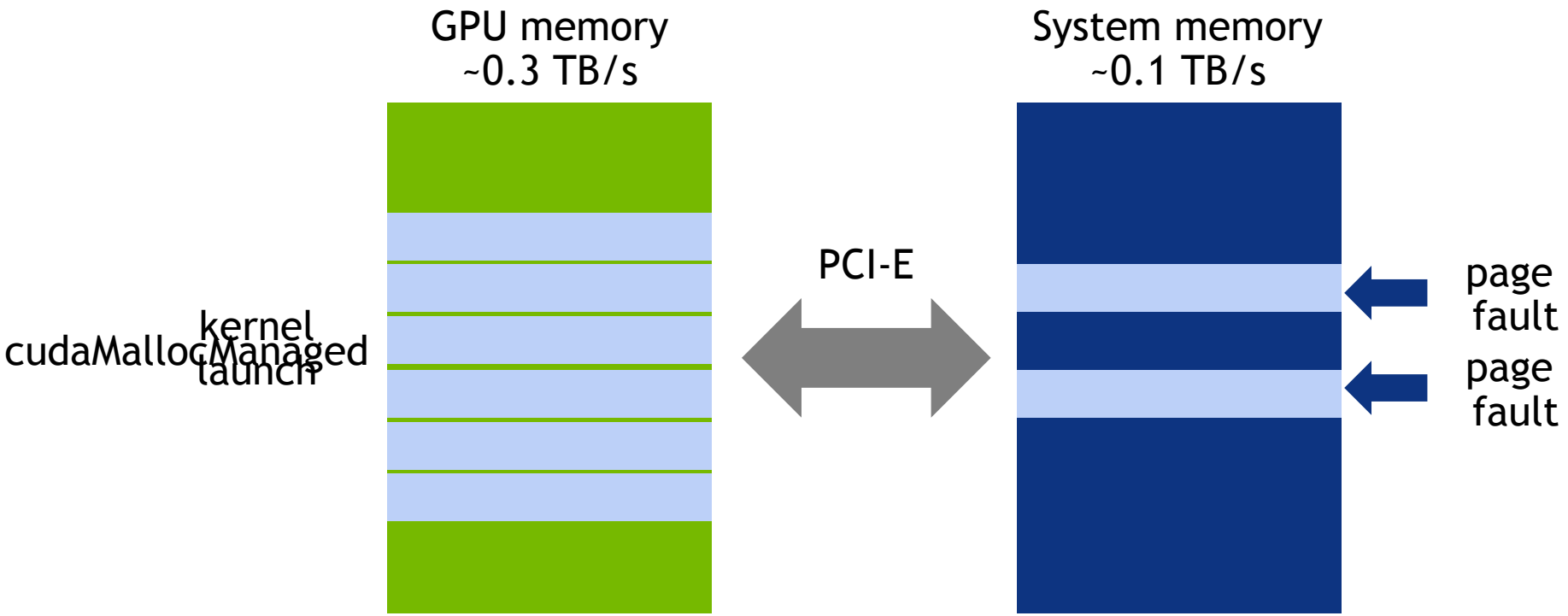
GPU always has address translation during the kernel execution

Pages allocated **before** they are used – cannot oversubscribe GPU

Pages migrate to GPU only on kernel launch – cannot migrate on-demand

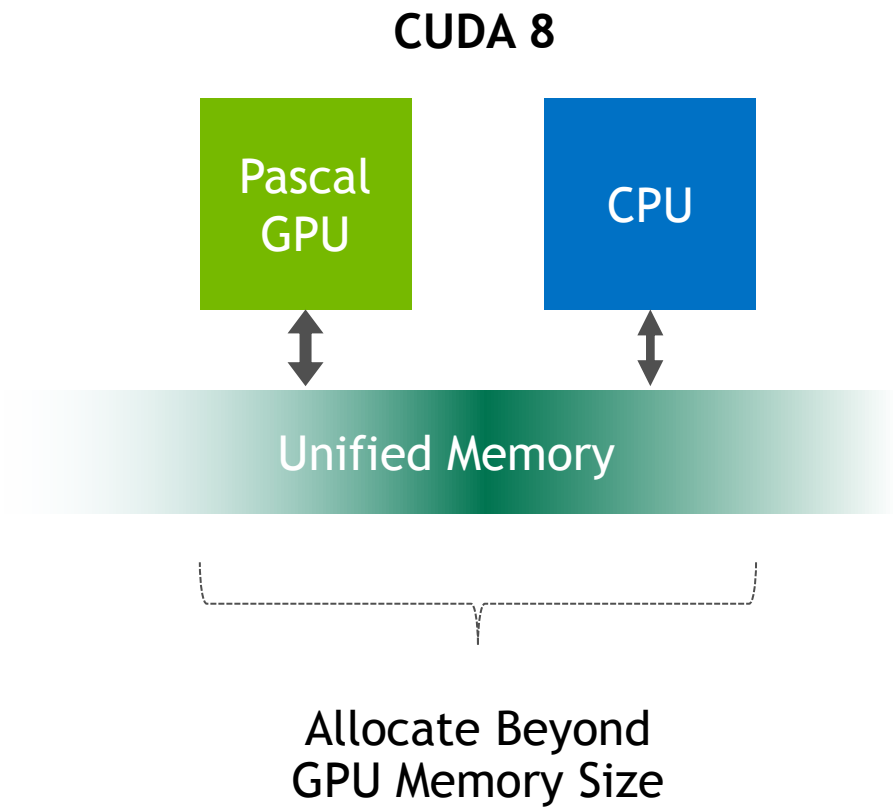
UNIFIED MEMORY ON PRE-PASCAL

Kernel launch triggers bulk page migrations



CUDA 8: UNIFIED MEMORY

Large datasets, simple programming, high performance



Enable Large Data Models	Oversubscribe GPU memory Allocate up to system memory size
Simpler Data Access	CPU/GPU Data coherence Unified memory atomic operations
Tune Unified Memory Performance	Usage hints via cudaMemAdvise API Explicit prefetching API

UNIFIED MEMORY ON PASCAL

Now supports GPU page faults

```
cudaMallocManaged(&ptr, ...);
*ptr = 1;
qsort<<<...>>>(ptr);
```

← Empty, no pages anywhere (similar to malloc)

← **CPU page fault**: data allocates on CPU

← **GPU page fault**: data migrates to GPU

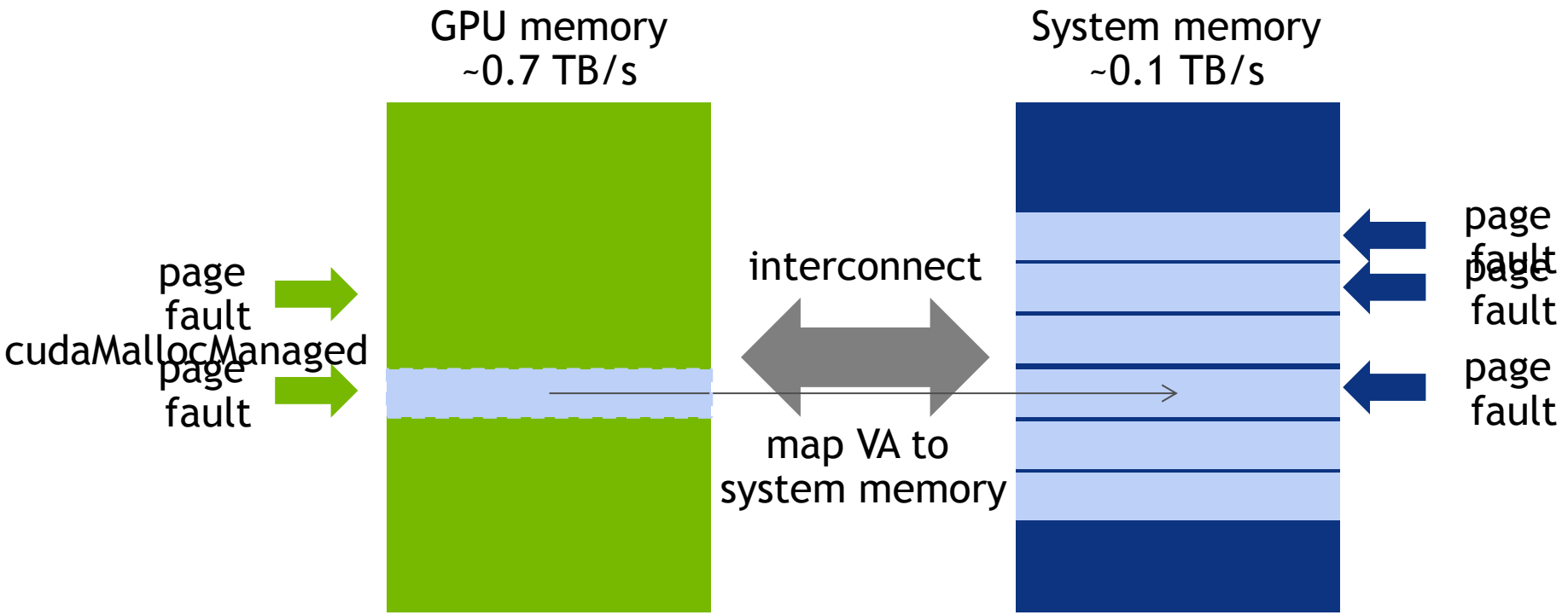
If GPU does not have a VA translation, it issues an interrupt to CPU

Unified Memory driver could decide to map or migrate depending on heuristics

Pages populated and data migrated **on first touch**

UNIFIED MEMORY ON PASCAL

True on-demand page migrations



UNIFIED SYSTEM ALLOCATOR

Any memory will be available for GPU*

CPU code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

GPU code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  free(data);
}
```

*on supported operating systems

SUMMIT

SUMMIT

2017 OLCF Leadership System

Vendor: IBM (Prime) / NVIDIA™ / Mellanox Technologies®

Approximately 3400 nodes, each with:

- IBM POWER9 CPUs + NVIDIA Volta GPUs
- CPUs and GPUs connected with high speed NVLink
- Large coherent memory: over 512 GB (HBM + DDR4)
- Over 40 TF peak performance

Dual-rail Mellanox® EDR-IB full, non-blocking fat-tree interconnect

SUMMIT

How does Summit compare to Titan

Feature	Summit	Titan
Application Performance	5-10x Titan	Baseline
Number of Nodes	~3,400	18,688
Node performance	> 40 TF	1.4 TF
Memory per Node	>512 GB (HBM + DDR4)	38GB (GDDR5+DDR3)
NVRAM per Node	800 GB	0
Node Interconnect	NVLink (5-12x PCIe 3)	PCIe 2
System Interconnect (node injection bandwidth)	Dual Rail EDR-IB (23 GB/s)	Gemini (6.4 GB/s)
Interconnect Topology	Non-blocking Fat Tree	3D Torus
Processors	IBM POWER9 NVIDIA Volta™	AMD Opteron™ NVIDIA Kepler™
File System	120 PB, 1 TB/s, GPFS™	32 PB, 1 TB/s, Lustre®
Peak power consumption	10 MW	9 MW

SUMMIT

Titan & Summit Application Differences

Fewer but much more powerful nodes

1/6th the number of nodes, but 25x more powerful

Must exploit more node-level parallelism

Multiple CPUs and GPU to keep busy

Likely requires OpenMP or OpenACC programming model

Very large memory

Summit has ~15x more memory per node than Titan

Interconnect is only ~3x the bandwidth of Titan

Need to exploit data locality within nodes to minimize message passing traffic

RESOURCES

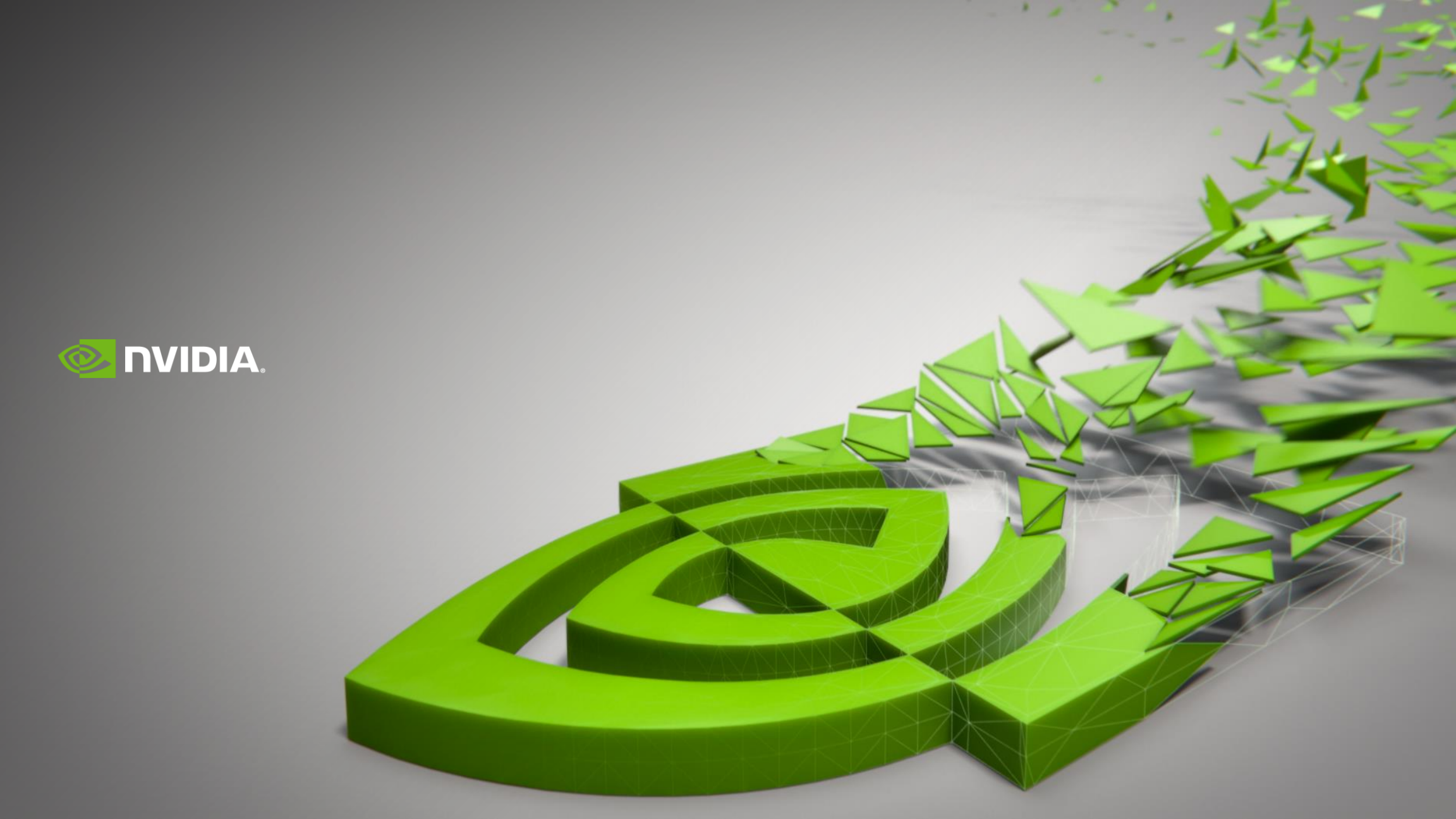
Learn more about GPUs

CUDA resource center: http://docs.nvidia.com/cuda

GTC on-demand: http://on-demand-gtc.gputechconf.com

Parallel Forall blog: http://devblogs.nvidia.com/parallelforall

Self-paced labs: http://nvidia.qwiklab.com


NVIDIA

COOPERATIVE GROUPS

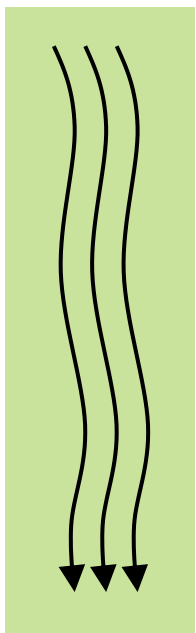
COOPERATIVE GROUPS

A Programming Model for Coordinating Groups of Threads

Support clean composition across software boundaries (e.g. Libraries)

Optimize for hardware fast-path using safe, flexible synchronization

A programming model that can scale from Kepler to future platforms



COOPERATIVE GROUPS SUMMARY

Flexible, Explicit Synchronization

Thread groups are explicit objects in the program

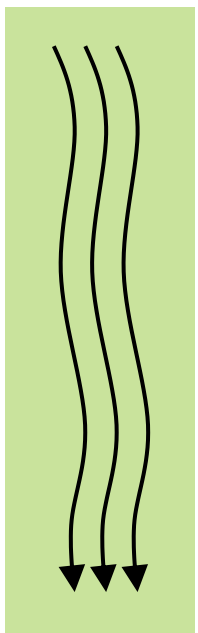
```
thread_group group = this_thread_block();
```

Collectives, such as barriers, operate on thread groups

```
sync(group);
```

New groups are constructed by partitioning existing groups

```
thread_group tiled_partition(thread_group base, int size);
```



MOTIVATING EXAMPLE

Optimizing for Warp Size

```
__device__
int warp_reduce(int val) {
  extern __shared__ int smem[];
  const int tid = threadIdx.x;

  #pragma unroll
  for (int i = warpSize/2; i > 0; i /= 2) {
    smem[tid] = val;       __syncthreads();
    val += smem[tid ^ i]; __syncthreads();
  }
  return val;
}
```

__syncthreads() is too expensive when sharing is only within warps

MOTIVATING EXAMPLE

Implicit Warp-Synchronous Programming is Tempting…

```
__device__
int warp_reduce(int val) {
  extern __shared__ int smem[];
  const int tid = threadIdx.x;

  #pragma unroll
  for (int i = warpSize/2; i > 0; i /= 2) {
    smem[tid] = val;
    val += smem[tid ^ i];
  }
  return val;
}
```

Barriers separating steps removed.
UNSAFE!

MOTIVATING EXAMPLE

Safe, Explicit Programming for Performance

Approximately equal performance to unsafe warp programming

```
__device__
int warp_reduce(int val) {
  extern __shared__ int smem[];
  const int tid = threadIdx.x;

  #pragma unroll
  for (int i = warpSize/2; i > 0; i /= 2) {
    smem[tid] = val;       sync(this_warp());
    val += smem[tid ^ i];  sync(this_warp());
  }
  return val;
}
```

Safe *and* Fast!

PASCAL: MULTI-BLOCK COOPERATIVE GROUPS

Provide a new launch mechanism for multi-block groups

Cooperative Groups collective operations like `sync(group)` work across all threads in the group

Save bandwidth and latency compared to multi-kernel approach required on Kepler GPUs