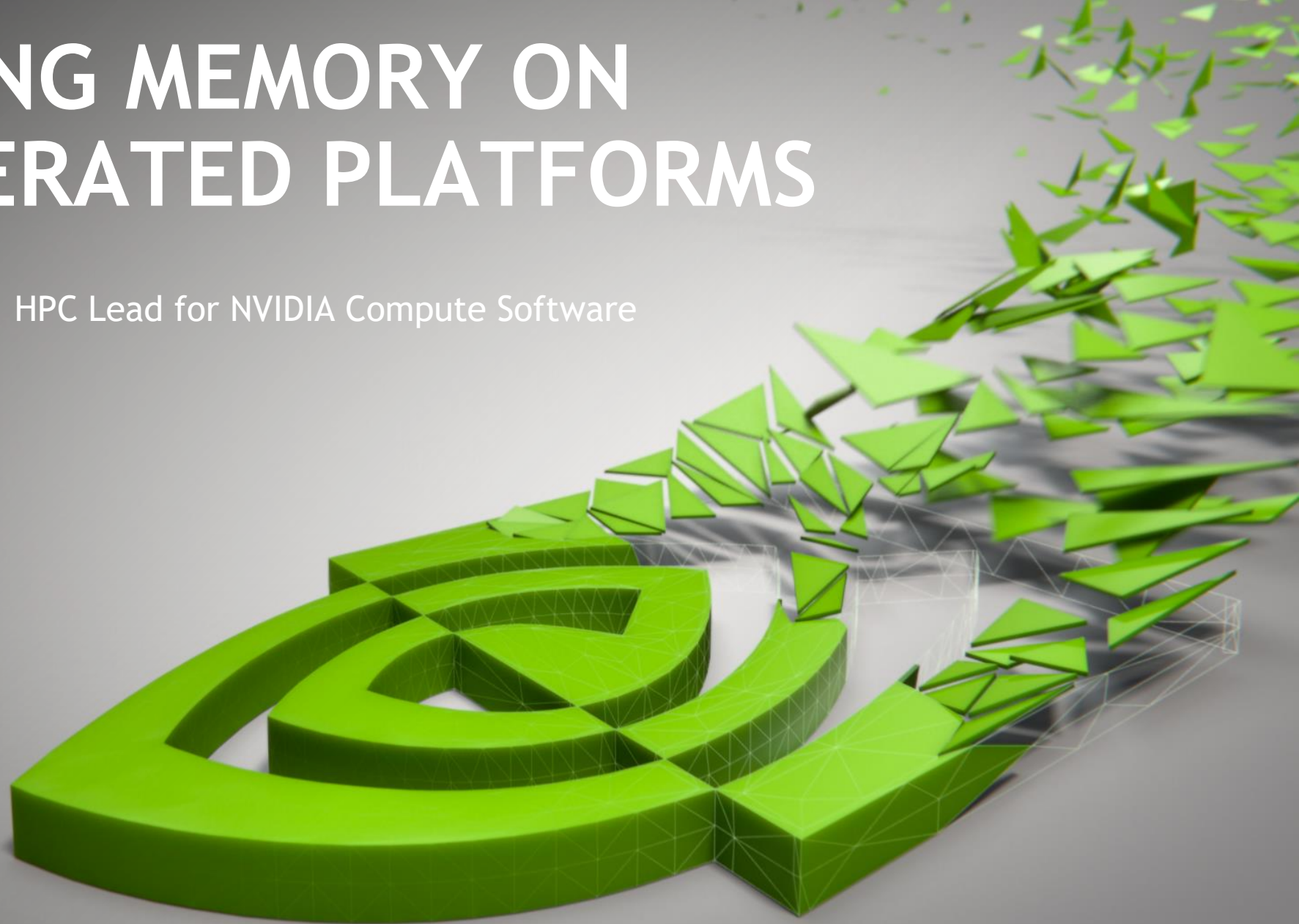


UNIFYING MEMORY ON ACCELERATED PLATFORMS

CJ Newburn

Principal Engineer, HPC Lead for NVIDIA Compute Software



OUTLINE

Motivation: Heterogeneous, highly-connected platforms are the path to scale

Evolution of device support: on a path toward a more ideal memory system

Under the hood: how paging really works

Exercising control: tuning for performance

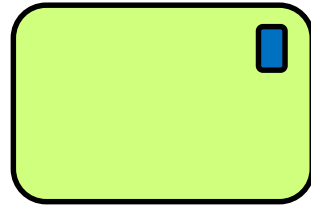
Heterogeneous memory manager

The next stage: Address translation service

Join the communal effort!

TOWARD A SCALED SYSTEM

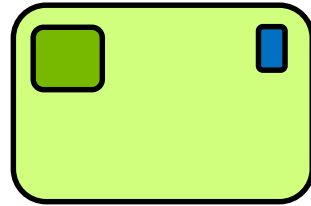
A basic CPU



TOWARD A SCALED SYSTEM

Lots of threads

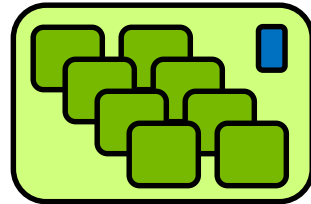
- Threads



TOWARD A SCALED SYSTEM

LOTS of threads

- Wider stronger nodes
 - Threads
 - Efficiency
 - Density

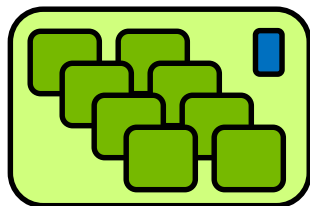


TOWARD A SCALED SYSTEM

LOTS of threads

- Wider stronger nodes

- Threads
- Efficiency
- Density



- Strong connections

- Push out IB “sharp edge” with a memory-bus link (NVLINK)
- Support many NICs
- Direct load/store

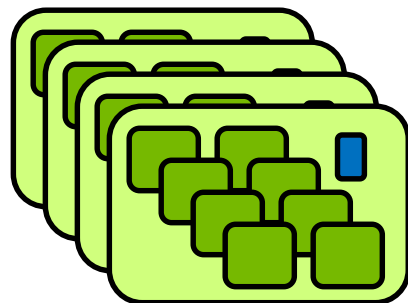
CHOOSING A SCALED PLATFORM

Better strong scaling, easier to manage

Wide nodes that may have high connectivity are a building block for efficiency

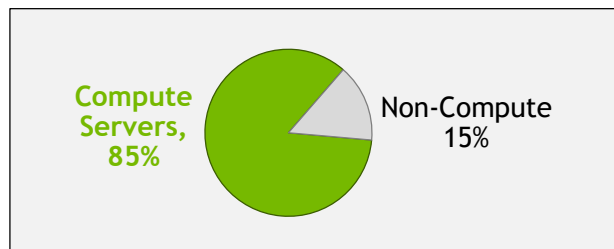
- Wider, stronger nodes

- Threads
- Efficiency
- Density

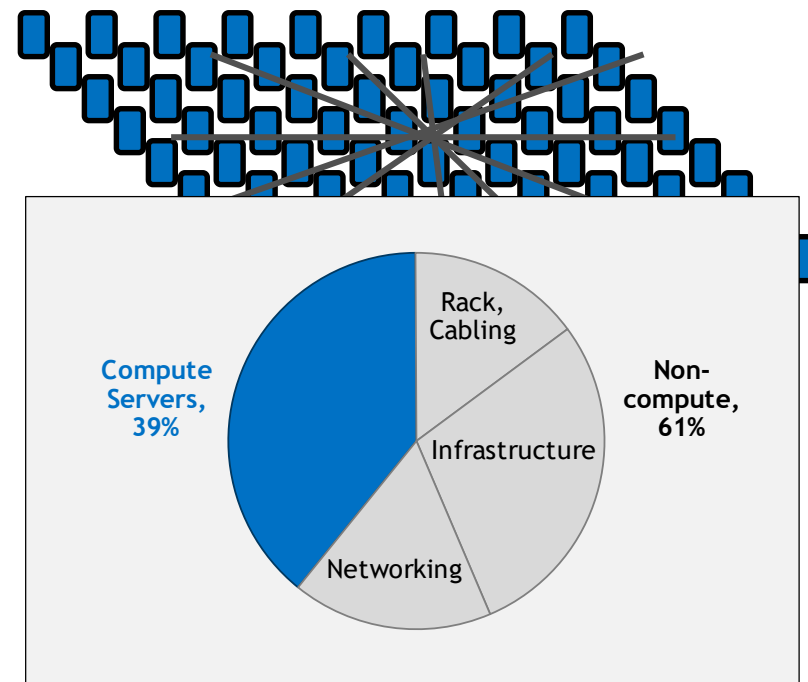


- Strong connections

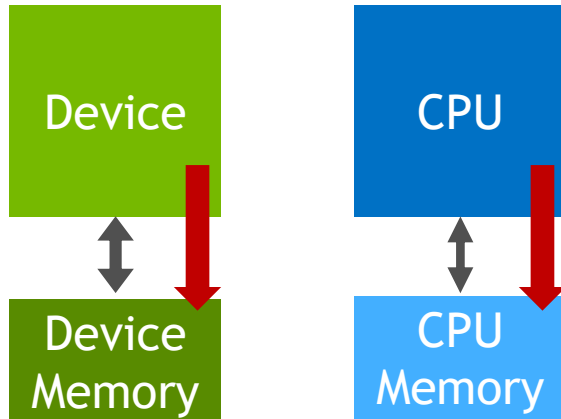
- Push out IB “sharp edge” with NVLINK
- Support many NICs
- Direct load/store



vs.



NON-UNIFIED MEMORY



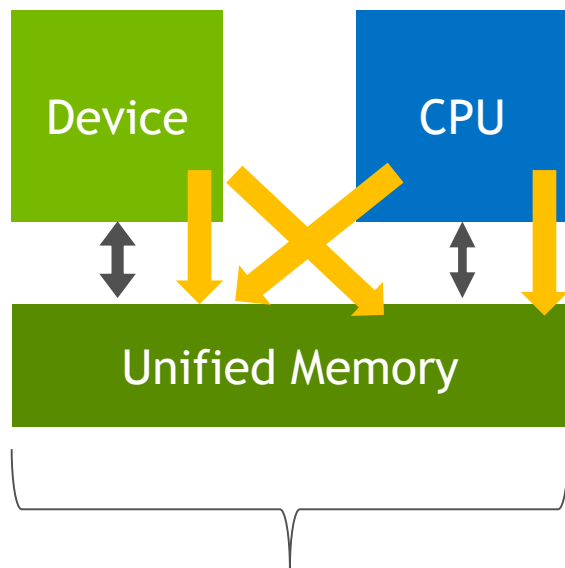
More to think
about, harder
to reason about

Two memories, not unified
Explicit access from only each side
Pinned on host and device

UNIFIED MEMORY

Dramatically Lower Developer Effort

CUDA 6+



Allocate Up To
GPU Memory Size

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

UNIFIED SYSTEM ALLOCATOR

Allocate unified memory using standard malloc, with HMM

CUDA 8 Code with System Allocator

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

    // Allocate memory using any standard allocator
    data = (char *) malloc(N * sizeof(char));

    fread(data, 1, N, fp);

    qsort<<<...>>>(data,N,1,compare);
    cudaDeviceSynchronize(); // avoids a race
    use_data(data);

    // Free the allocated memory
    free(data);
}
```

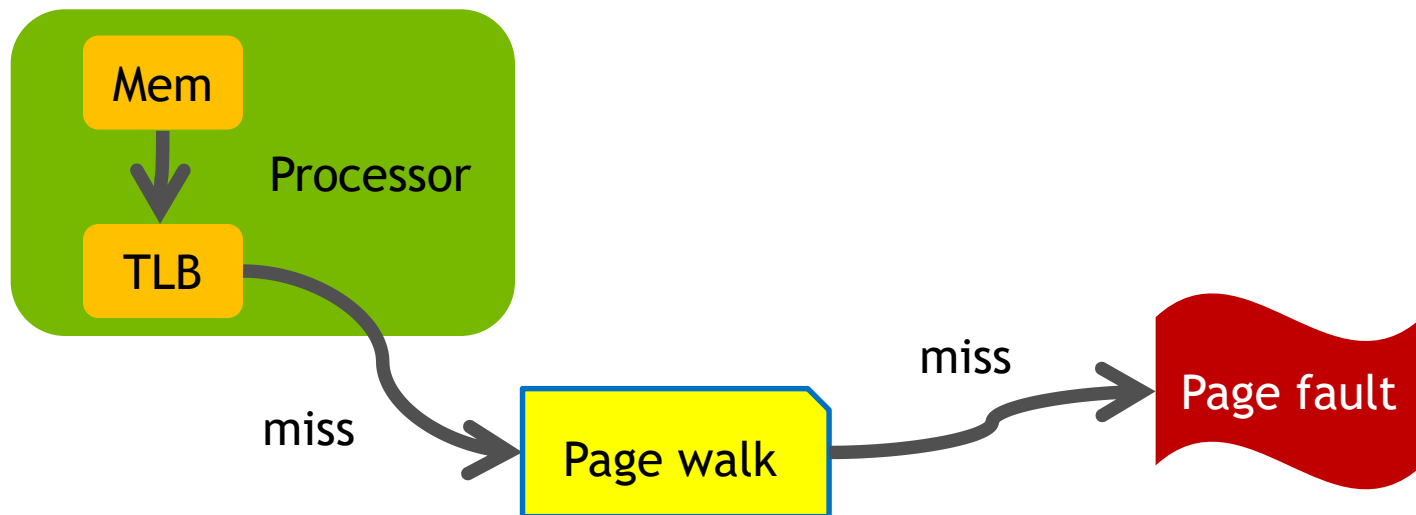
Removes CUDA specific allocator restrictions

Data movement is transparently handled

Requires operating system support

UNDER THE HOOD: ANATOMY OF A PAGE ACCESS

Intro to the basics

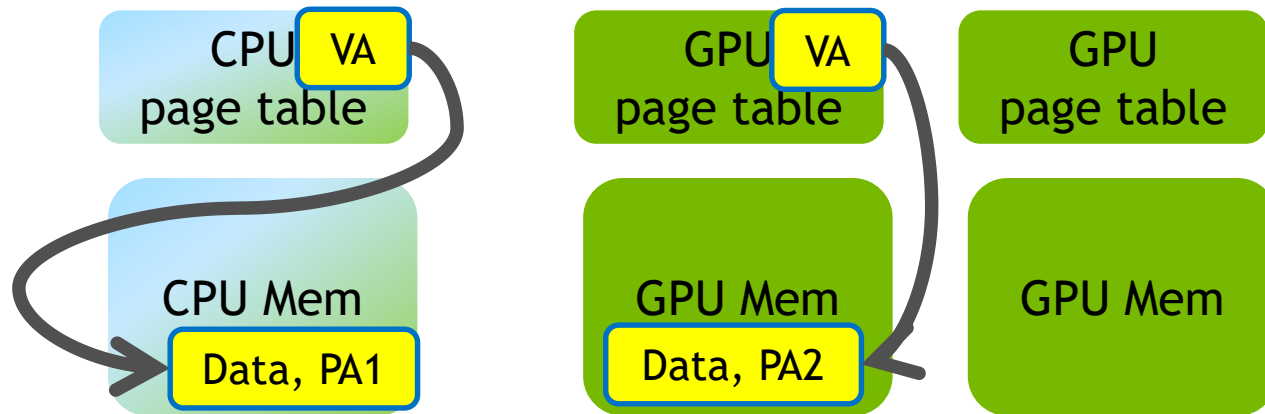


Page fault conditions

- Invalid: doesn't exist at all, or not valid where you are
- Not writable, upon an attempt to write or an atomic access

UNDER THE HOOD: PAGE TABLES

Distributed can be better



- Distinct: enables different Virtual Address (VA) → Physical Address (PA) mappings
- Local: enables faster access
- Special page table entries: invalid, special markings

UNDER THE HOOD: PAGE FAULTS

Getting SW involved

- Once you allocate, you have a virtual address
- With a virtual address, you can attempt access
- Upon a page fault, SW is involved
 - Deferred work: need not materialize the physical memory upon a virtual allocation
 - Smart and flexible work: apply heuristics for migration

UNDER THE HOOD: DRIVER

Tracking the details, exercising smarts

- User can specify policies and usage hints via APIs; driver tracks them
- Driver knows where pages are valid, whether they are writable
- Driver can also use a cost model

HETEROGENEOUS MEMORY MANAGER (HMM)

Filling a gap

Requirements

- Enable malloc'd and file-backed memory to be migrated from CPU to device
 - malloc/heap, global, statics
- Enable CPU memory to be DMA'd without being pinned
- Support RDMA
- Enable read duplication

HMM SUPPORT FOR MIGRATION: PTES

Steps to get what you don't have

Background

- CPU accesses TLB, TLB walks page table upon miss, fault upon invalid page table entry
- Page fault (PF) handler invoked, calls the driver for help if needed to get page

Implementation

- Invalid page table entry (PTE) indicates that CPU doesn't have the page
- Special encoding in PTE indicates that device has the page, so get driver's help
- Driver transfers data from device to CPU, clears special encoding, let PF finish

HMM SUPPORT FOR MIGRATION: NOTIFICATION

Sharing what you do have

Background

- Device may want to map to CPU's memory, for remote access
 - Needs to know CPU's mapping from virtual address (VA) to physical address (PA)
- But mapping could change at any time!
 - Swapped out, copy on write, managed by hypervisor
 - Device needs to know whether the mapping it has is still valid
- Device needs to be notified if the state of the page changes

HMM SUPPORT FOR MIGRATION: MAPPINGS

Sharing what you do have

Implementation

- Could get VA → PA mapping, but that mapping could change without notice
- New: notify that a mapping is about to change, wait for ack before changing

Implications

- DMA can start, and even if a mapping change is pending, it gets to finish first

OS SUPPORT FOR HMM

- First version of HMM (v25) is available in kernel 4.14
 - Released Nov 12, 2017 with HMM
 - <http://www.omgubuntu.co.uk/2017/11/linux-kernel-4-14-lts-features>
- Interest in HMM
 - RedHat
 - IBM
 - Mediatek
 - Mellanox
 - NVIDIA - listed as collaborator on patch

IN THE BACKLOG FOR HMM PATCHES

- Write protection
 - Device-driver-controlled: provides support for read-duplication and SysMem-based atomic operations between CPU and GPU
- Differentiate between unmap (e.g. free) and invalidate (e.g. migrate) page callbacks
 - Not having to assume unmap would avoid discarding all info about the mapping
- Support for migrating pages that are file-backed memory vs. anonymous memory
 - Example: mmap a file, then use it in UVM
 - File-backed memory requires support for additional fields in physical struct page, e.g. inode, struct file*.
- RMDA without pinning
 - HMM or ATS + page-fault-capable NIC + NIC and UVM driver support

ADDRESS TRANSLATION SERVICE (ATS)

A HW-centric solution

Key differences from SW-based HMM solution

- Single page table
- IO memory management unit (IOMMU) sends TLB invalidation to devices
 - Reference IBM POWER, AMD's IOMMU V2, Intel's VTD
- Need HW support for keeping devices coherent
- Needs device support - coherence; walk pages itself or ask for help

OS SUPPORT FOR ATS

Still in design phase

Device memory must be known to OS

- Device, BIOS, OS must collaborate
- Based on existing NUMA on-lining work

More work needed

- Significant modifications to current NUMA implementation required for performance
 - Example: when an alloc of a device fails, want to replace existing page vs. fail over
- Still want a way to apply smart heuristics

TAKE AWAYS

Try HMM today in kernel 4.14!

Function of Heterogeneous Memory Manager

- SW assist that helps maintain coherence between CPU and devices

Benefits of Heterogeneous Memory Manager

- Enables migration of CPU memory to a device
- Enables device to get notified when memory state (writability, validity) changes

HMM (v25) is available in kernel 4.14

Enable easiest data movement between CPU, GPU, network

Linux patch for HW-oriented Address Translation Service is WIP

Come join the communal effort!

What questions can we explore together?



*More on
exercising control*



EXERCISING CONTROL

Power to the people

Manual or managed

Migrate or remote access

Map proactively or reactively

Duplicate or not

MANUAL OR MANAGED

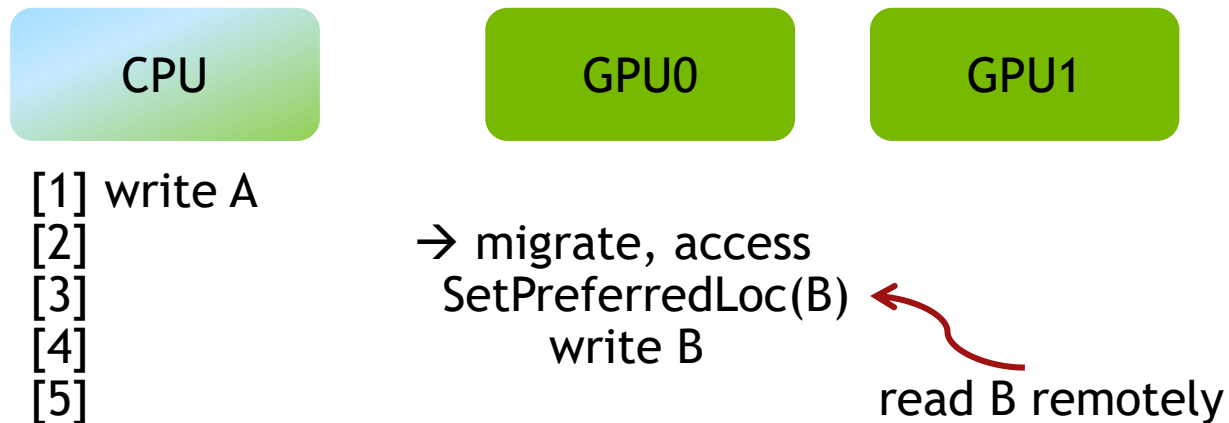
Manual

- `cudaMalloc(&address, size)`: place on the current device, access from device
- `cudaMallocHost(&address, size)`: place on host, access from host
- **Materialized and pinned** in either case: **cost paid immediately, not migratable**
- User knows what they are doing; give them **full control** with **minimal overhead**
- **Manually make GPU memory visible** from another GPU, `EnablePeerAccess`

Managed

- `cudaMallocManaged(&address, size)`: placed at first write, access from anywhere
 - Pre-Pascal: materialize and pin on CPU and GPU; can only access from one or the other
 - Pascal: **materialization is deferred, not pinned: cost paid on demand, migratable**
- Easy for user; **selective control** with `cudaMemAdvise`
- **No special steps to make memory visible to other GPUs**

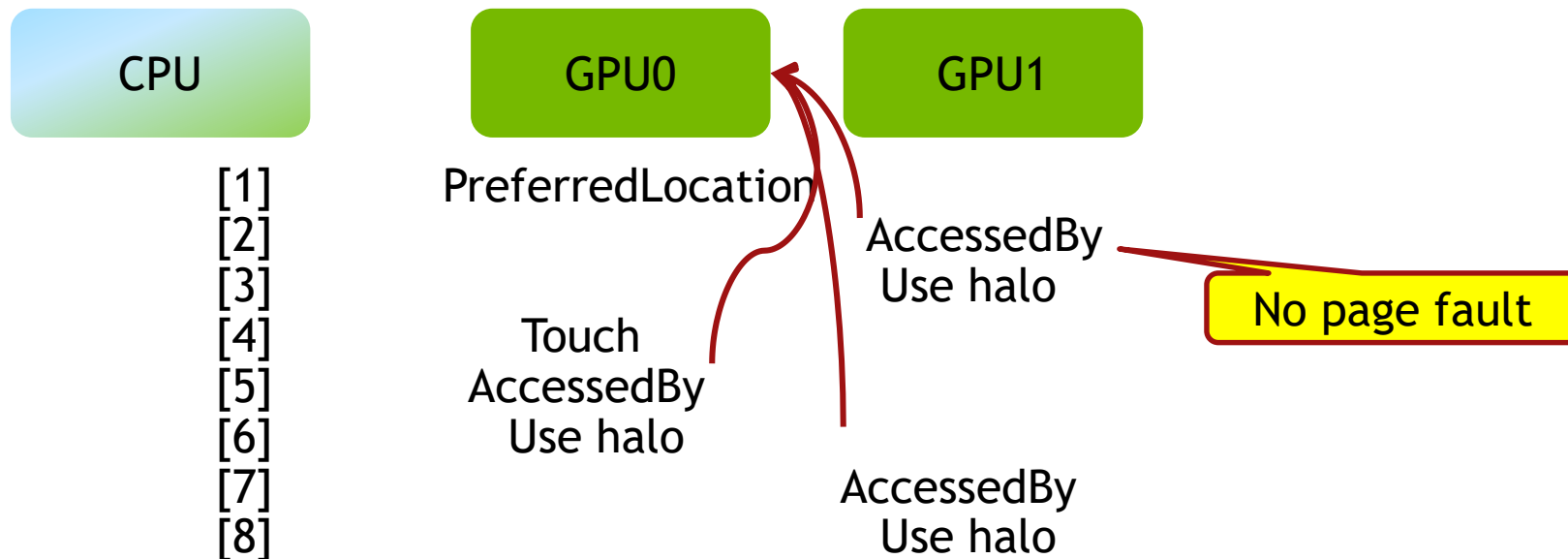
MIGRATE OR ACCESS REMOTELY



Default: migrate

Override: `cudaMemAdvise(address, size, SetPreferredLocation, deviceId)`

MAP PROACTIVELY OR REACTIVELY

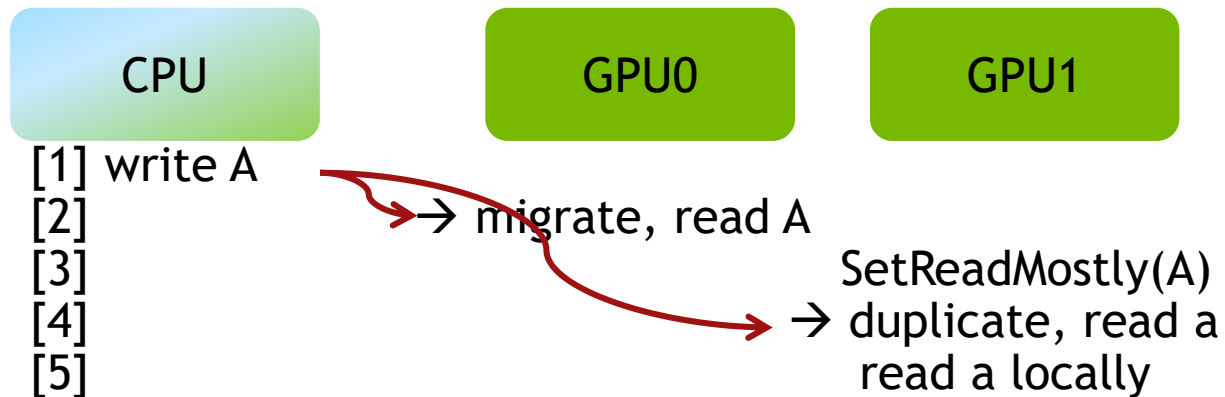


Map: create a page table entry at deviceId once page is materialized

Default: don't map, which could save overhead if not needed

Control: `cudaMemAdvise(address, size, SetAccessedBy, deviceId)`

DUPLICATE OR NOT



Default: migrate

Control: duplicate with `cudaMemAdvise(SetReadMostly)`

CUDA 8 UNIFIED MEMORY – EXAMPLE

Accessing data simultaneously by CPU and GPU codes

```
__global__ void mykernel(char *data) {  
    data[1] = 'g';  
}  
  
void foo() {  
    char *data;  
    cudaMallocManaged(&data, 2);  
  
    mykernel<<<...>>>(data); // assume data not written  
    // no synchronize here  
    data[0] = 'c';  
    cudaFree(data);  
}
```

Both CPU code and CUDA kernel accessing 'data' simultaneously

Possible with CUDA 8 unified memory on Pascal

GPU MEMORY OVERSUBSCRIPTION

Many domains would benefit

Combustion

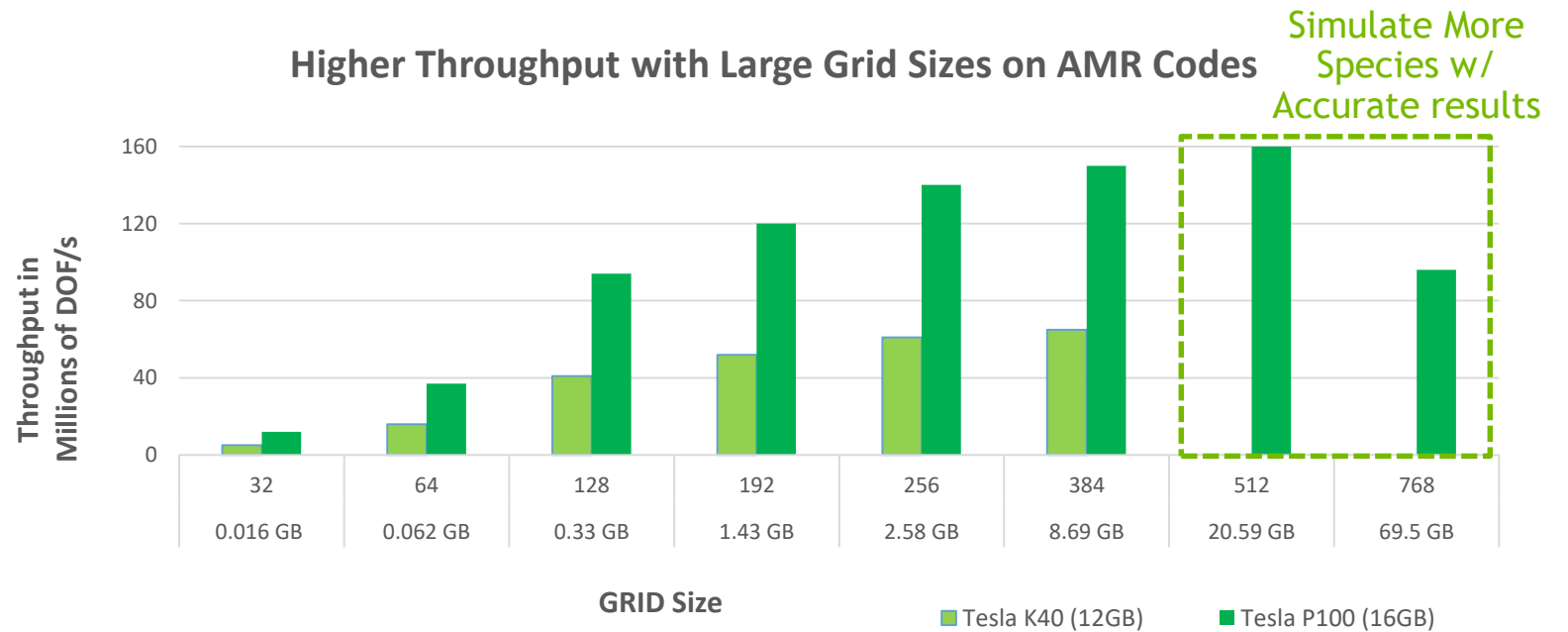
Many species & improved accuracy

Quantum chemistry

Larger systems

Ray-tracing

Larger scenes to render



ON-DEMAND ALLOCATION

Dynamic queues

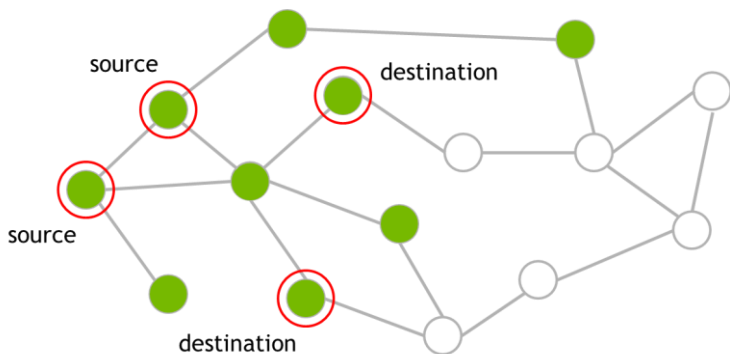
Problem: GPU populates queues with unknown size, need to overallocate



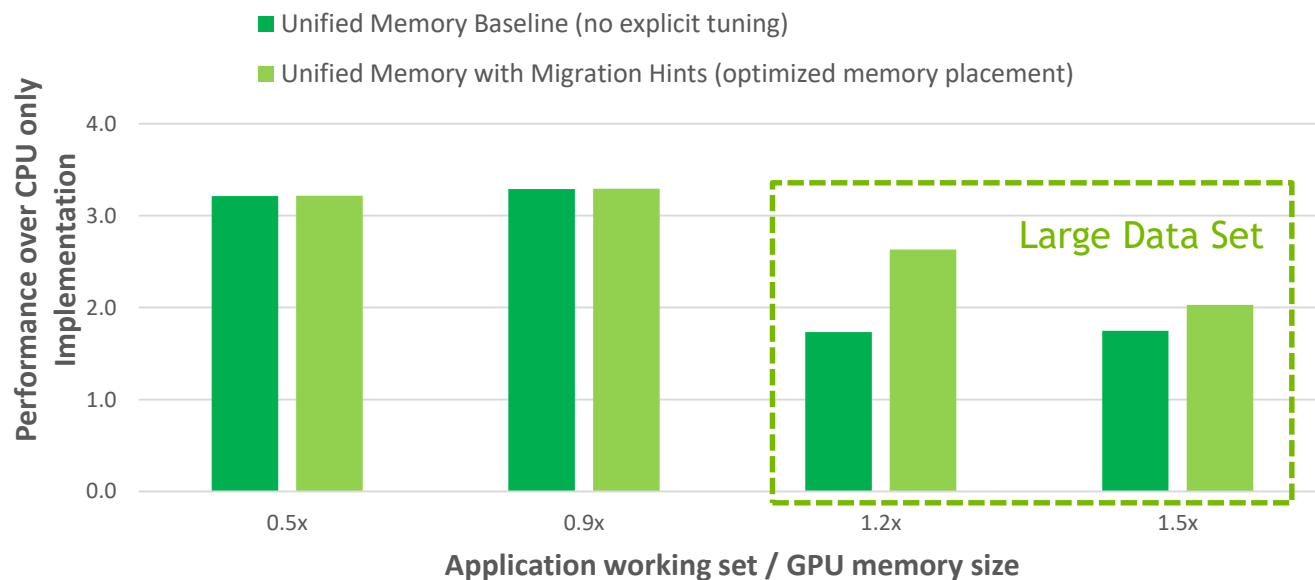
Solution: use Unified Memory for allocations (on Pascal)

ON-DEMAND PAGING

New use cases such as graph algorithms



Higher Performance with Unified Memory on Maximum Flow



PREFETCHING

Simple code example

```
void foo(cudaStream_t s) {
    char *data;
    cudaMallocManaged(&data, N);

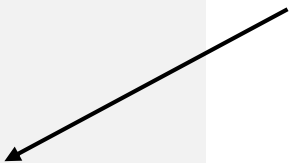
    init_data(data, N);

    cudaMemPrefetchAsync(data, N, myGpuId, s);
    mykernel<<<..., s>>>(data, N, 1, compare);
    cudaMemPrefetchAsync(data, N, cudaCpuDeviceId, s);
    cudaStreamSynchronize(s);


    use_data(data, N);

    cudaFree(data);
}
```

GPU faults are expensive
prefetch to avoid excess faults



CPU faults are less expensive
may still be worth avoiding



READ DUPLICATION: MEMADVISE

CPU and GPU, until there's a write

cudaMemAdviseSetReadMostly

Use when data is *mostly read* and occasionally written to

```
init_data(data, N);
```

```
cudaMemAdvise(data, N, cudaMemAdviseSetReadMostly, myGpuId);
```

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

← Read-only copy will be
created on GPU page fault

```
use_data(data, N);
```

← CPU reads will not page fault

READ DUPLICATION: PREFETCH

CPU and GPU, until there's a write

Prefetching creates read-duplicated copy of data and avoids page faults

Note: writes are allowed but will generate page fault and remapping

```
init_data(data, N);
```

```
cudaMemAdvise(data, N, cudaMemAdviseSetReadMostly, myGpuId);
```

```
cudaMemPrefetchAsync(data, N, myGpuId, cudaStreamLegacy);
```

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

```
use_data(data, N);
```

Read-only copy will be
created during prefetch

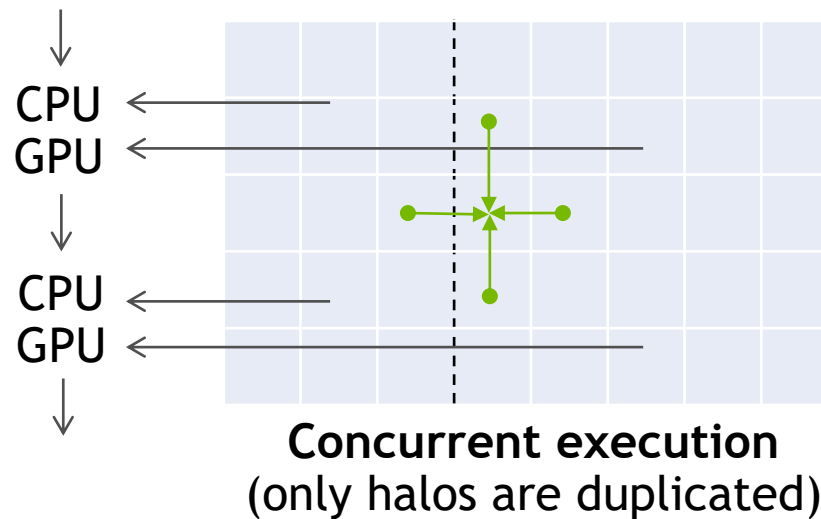
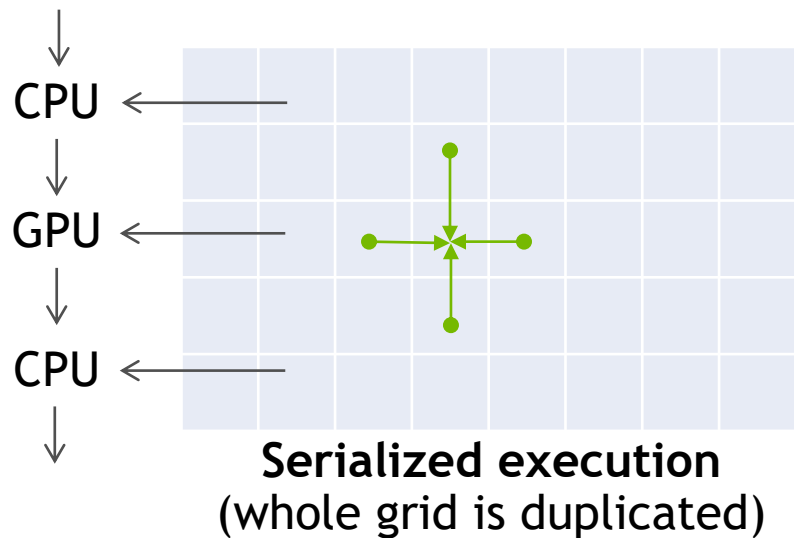
CPU and GPU reads
will not fault

READ DUPLICATION

Use cases

Useful during initial stages of porting - lots of CPU code using the same structures

Other examples: mesh connectivity, matrix coefficients, control state



DIRECT MAPPING

Preferred location and direct access

cudaMemAdviseSetPreferredLocation

Set preferred location to avoid migrations

First access will page fault and establish mapping

cudaMemAdviseSetAccessedBy

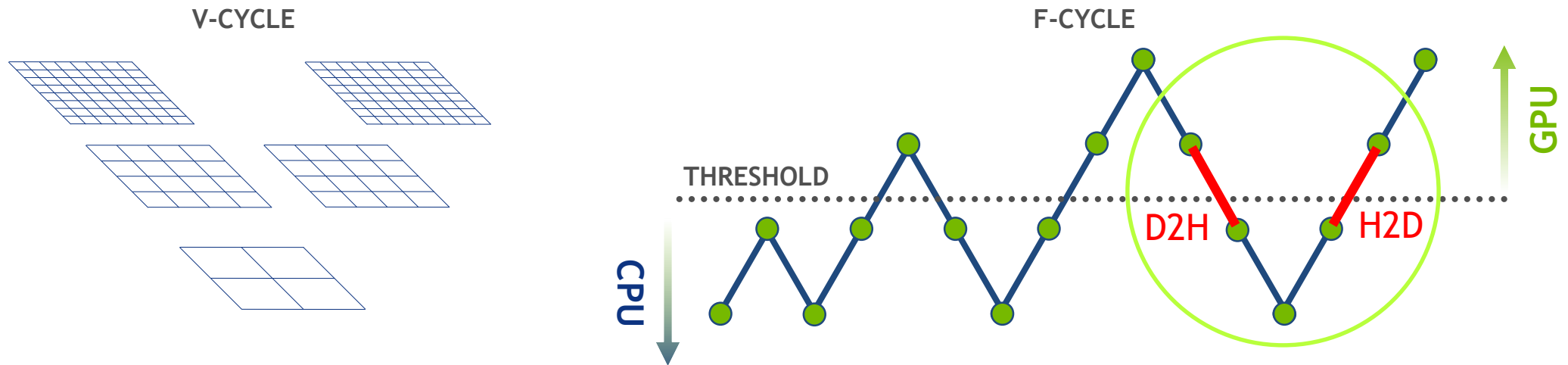
Pre-map data to avoid page faults wherever possible

First access avoid page fault when possible

Actual data location can be anywhere

DIRECT MAPPING

Use case: HPGMG



Hybrid implementation with Unified Memory: **fine** grids on **GPU**, **coarse** grids on **CPU**

Implicit CPU \leftrightarrow GPU communication during restriction and interpolation phases

DIRECT MAPPING

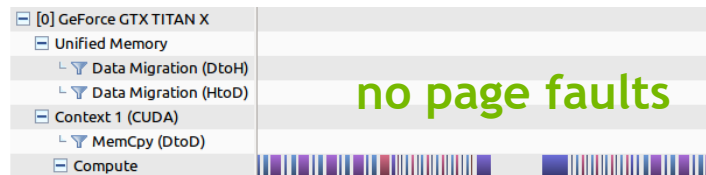
Use case: HPGMG

Problem: excessive faults and migrations at CPU-GPU crossover point



Solution: pin coarse levels to CPU and map them to GPU page tables

Pre-Pascal: allocate data with `cudaMallocHost` or `malloc + cudaHostRegister`



DIRECT MAPPING

Use case: HPGMG

CUDA 8 solution with performance hints (works with cudaMallocManaged)

```
// set preferred location to CPU to avoid migrations
cudaMemAdvise(ptr, size, cudaMemAdviseSetPreferredLocation, cudaCpuDeviceId);

// keep this region mapped to my GPU to avoid page faults
cudaMemAdvise(ptr, size, cudaMemAdviseSetAccessedBy, myGpuId);

// prefetch data to CPU and establish GPU mapping
cudaMemPrefetchAsync(ptr, size, cudaCpuDeviceId, cudaStreamLegacy);
```

20-30% estimated performance improvement on Tesla P100

LANGUAGE ECOSYSTEM OVERVIEW

For NVIDIA GPUs

Parallelism Models

OpenACC

CUDA-Fortran

CUDA C++

OpenCL

OpenMP 4

Compute Toolchains

GCC*

PGI Compiler

NV Compilers

Clang*

NVVM

NVVM

LLVM*
[NVPTX]

PTX**

[ptxas]

GPU microcode

libcuda.so
nvidia.ko



*denotes OSS
** open specification

OSS TOOLCHAIN POSSIBILITIES

For NVIDIA GPUs

Parallelism Models

OpenACC

CUDA C++

OpenCL

OpenMP 4

Vulkan

Compute Toolchains

