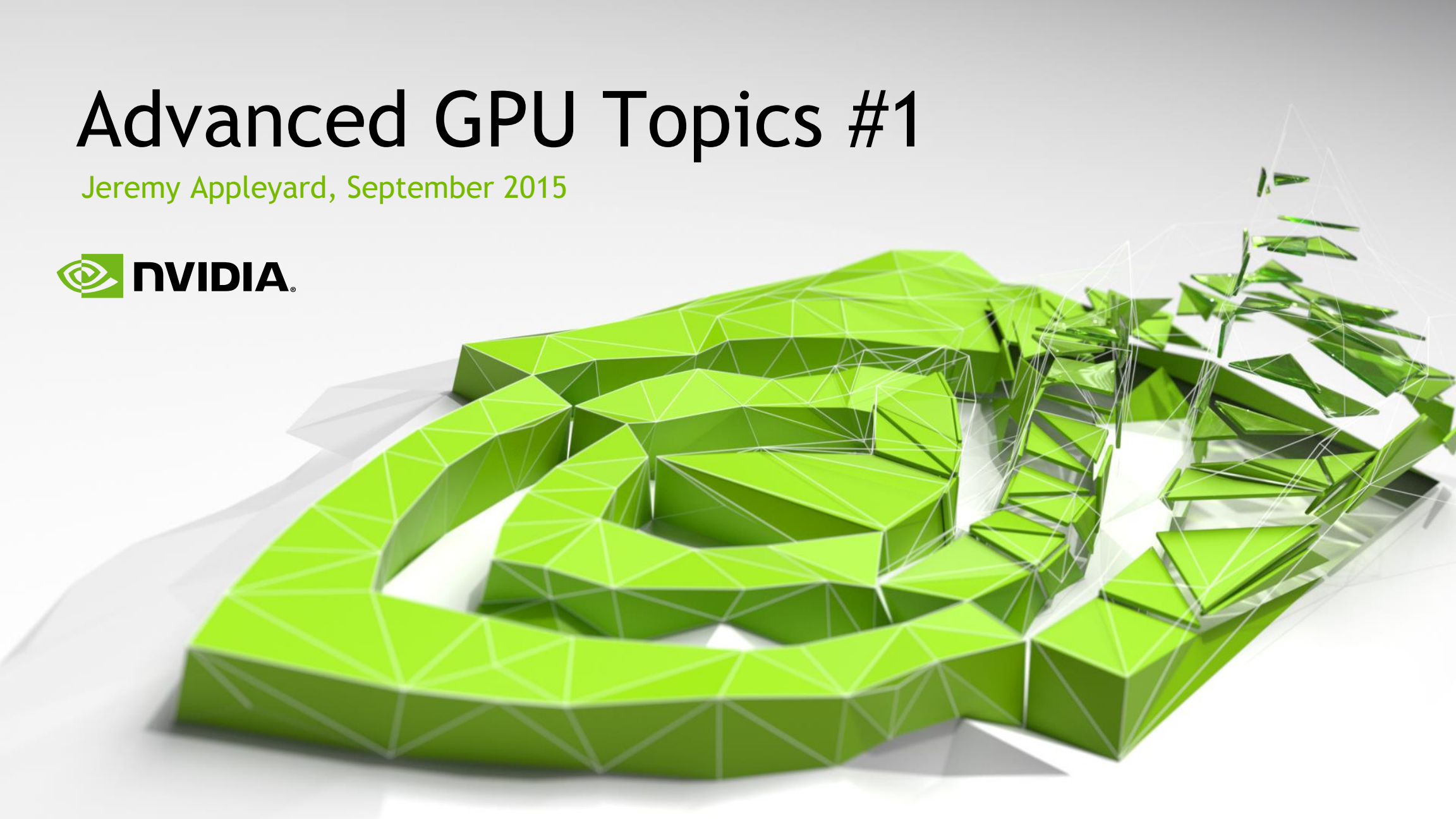
Advanced GPU Topics #1

Jeremy Appleyard, September 2015



In this talk

- Profiling GPU applications
- Optimizing Data Movement
- Using MPI with GPUs

Ask questions at any point!

Profiling GPU Applications

Profiling Tools

Many options!

From NVIDIA

- nvprof
- NVIDIA Visual profiler
  - Standalone (nvvp)
  - Integrated into Nsight Eclipse Edition (nsight)
- Nsight Visual Studio Edition

Third Party

- System
- VampirTrace
- PAPI CUDA component
- HPC Toolkit

This talk

- We will focus on nvprof and nvvp
- nvprof => NVIDIA profiler
  - Command line
- nvvp => NVIDIA Visual Profiler
  - GUI based

nvprof

Simple usage

- `nvprof ./<executable>`
- Example: vector addition from yesterday's talk

```
attributes(global) subroutine vecAdd_GPU(c, a, b, n)
   INTEGER, value :: n
   REAL, device, intent(in)  :: a(n), b(n)
   REAL, device, intent(out) :: c(n)

   INTEGER :: i

   i = (blockIdx%x - 1) * blockDim%x + threadIdx%x

   if (i <= n) then
      c(i) = a(i) + b(i)
   end if
end subroutine vecAdd_GPU
```

```
nvprof ./vecAdd
==34092== NVPROF is profiling process 34092, command: ./vecAdd
==34092== Profiling application: ./vecAdd
==34092== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 69.07%  3.8547ms         2  1.9274ms  1.9093ms  1.9454ms  [CUDA memcpy HtoD]
 29.04%  1.6205ms         1  1.6205ms  1.6205ms  1.6205ms  [CUDA memcpy DtoH]
  1.19%  66.178us         1  66.178us  66.178us  66.178us  kernel_vecadd_gpu_
  0.71%  39.650us         2  19.825us  19.617us  20.033us  __pgi_dev_cumemset_4

==34092== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 95.29%  321.93ms         3  107.31ms  215.28us  321.48ms  cudaMalloc
  2.70%  9.1355ms         3  3.0452ms  2.1550ms  3.5234ms  cudaMemcpy
  1.47%  4.9710ms       498  9.9810us     177ns  498.93us  cuDeviceGetAttribute
  0.22%  758.16us         3  252.72us  234.62us  284.11us  cudaFree
  0.15%  519.49us         6  86.581us  82.801us  89.206us  cuDeviceTotalMem
  0.13%  449.63us         6  74.938us  70.892us  81.049us  cuDeviceGetName
  0.02%  69.908us         3  23.302us  11.485us  43.331us  cudaLaunch
  0.00%  12.300us        10  1.2300us     179ns  8.0460us  cudaSetupArgument
  0.00%  3.4430us        12     286ns     193ns     716ns  cuDeviceGet
  0.00%  2.8280us         2  1.4140us     285ns  2.5430us  cuDeviceGetCount
  0.00%  2.5750us         3     858ns     448ns  1.6120us  cudaConfigureCall
```

```
nvprof ./vecAdd
==34092== NVPROF is profiling process 34092, command: ./vecAdd
==34092== Profiling application: ./vecAdd
==34092== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 69.07%  3.8547ms         2  1.9274ms  1.9093ms  1.9454ms  [CUDA memcpy HtoD]
 29.04%  1.6205ms         1  1.6205ms  1.6205ms  1.6205ms  [CUDA memcpy DtoH]
  1.19%  66.178us         1  66.178us  66.178us  66.178us  kernel_vecadd_gpu_
  0.71%  39.650us         2  19.825us  19.617us  20.033us  __pgi_dev_cumemset_4
```

- Top half of the profile is runtime measured from the GPU perspective
- What can we see?
  - Memcpy XtoX is memory copy to and from the GPU
  - The vector addition is only 1.19%!
  - This is valuable information!

```
nvprof ./vecAdd
==34092== NVPROF is profiling process 34092, command: ./vecAdd
==34092== Profiling application: ./vecAdd
==34092== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 69.07%  3.8547ms         2  1.9274ms  1.9093ms  1.9454ms  [CUDA memcpy HtoD]
 29.04%  1.6205ms         1  1.6205ms  1.6205ms  1.6205ms  [CUDA memcpy DtoH]
  1.19%  66.178us         1  66.178us  66.178us  66.178us  kernel_vecadd_gpu_
  0.71%  39.650us         2  19.825us  19.617us  20.033us  __pgi_dev_cumemset_4
```

- (PGI) OpenACC kernels will be named after the subroutine name and line number
- eg. `vecadd_17_gpu`
  - Subroutine vecadd
  - Line 17 in the file
  - Compiled for the gpu

- Bottom half of the profile is runtime measured from the CPU perspective
- What can we see?
  - First allocation is expensive (cuda initialisation)

```
==34092== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 95.29%  321.93ms         3  107.31ms  215.28us  321.48ms  cudaMalloc
  2.70%  9.1355ms         3  3.0452ms  2.1550ms  3.5234ms  cudaMemcpy
  1.47%  4.9710ms       498  9.9810us     177ns  498.93us  cuDeviceGetAttribute
  0.22%  758.16us         3  252.72us  234.62us  284.11us  cudaFree
  0.15%  519.49us         6  86.581us  82.801us  89.206us  cuDeviceTotalMem
  0.13%  449.63us         6  74.938us  70.892us  81.049us  cuDeviceGetName
  0.02%  69.908us         3  23.302us  11.485us  43.331us  cudaLaunch
  0.00%  12.300us        10  1.2300us     179ns  8.0460us  cudaSetupArgument
  0.00%  3.4430us        12     286ns     193ns     716ns  cuDeviceGet
  0.00%  2.8280us         2  1.4140us     285ns  2.5430us  cuDeviceGetCount
  0.00%  2.5750us         3     858ns     448ns  1.6120us  cudaConfigureCall
```

nvprof

More advanced options

- `nvprof -h`
- There are quite a few options!
- Some useful ones:
  - -o: creates an output file which can be imported into nvvp
  - -m and -e: collect **m**etrics or **e**vents
  - --analysis-metrics: collect all metrics for import into nvvp
  - --query-metrics and --query-events: query which metrics/events are available

nvprof

Events and Metrics

- Most are quite in-depth, however some useful ones for quick analysis
- In general, events are only for the expert. Rarely useful
- (A few) useful metrics:
  - dram_read_throughput: Main GPU memory read throughput
  - dram_write_throughput: Main GPU memory write throughput
  - flop_count_sp: Number of single precision floating point operations
  - flop_count_dp : Number of double precision floating point operations

nvprof

Metrics Example

```
nvprof --metrics dram_read_throughput,dram_write_throughput,flop_count_sp,flop_count_dp ./vecAdd
==32928== NVPROF is profiling process 32928, command: ./vecAdd
==32928== Warning: Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==32928== Profiling application: ./vecAdd
==32928== Profiling result:
==32928== Metric result:
Invocations                               Metric Name                         Metric Description         Min         Max         Avg
Device "Tesla K40m (0)"
       Kernel: kernel_vecadd_gpu_
      1                        dram_read_throughput              Device Memory Read Throughput  136.55GB/s  136.55GB/s  136.55GB/s
      1                       dram_write_throughput             Device Memory Write Throughput  70.062GB/s  70.062GB/s  70.062GB/s
      1                             flop_count_sp  Floating Point Operations(Single Precisi     1000000     1000000     1000000
      1                             flop_count_dp  Floating Point Operations(Double Precisi           0           0           0
```

nvprof

Metrics Example

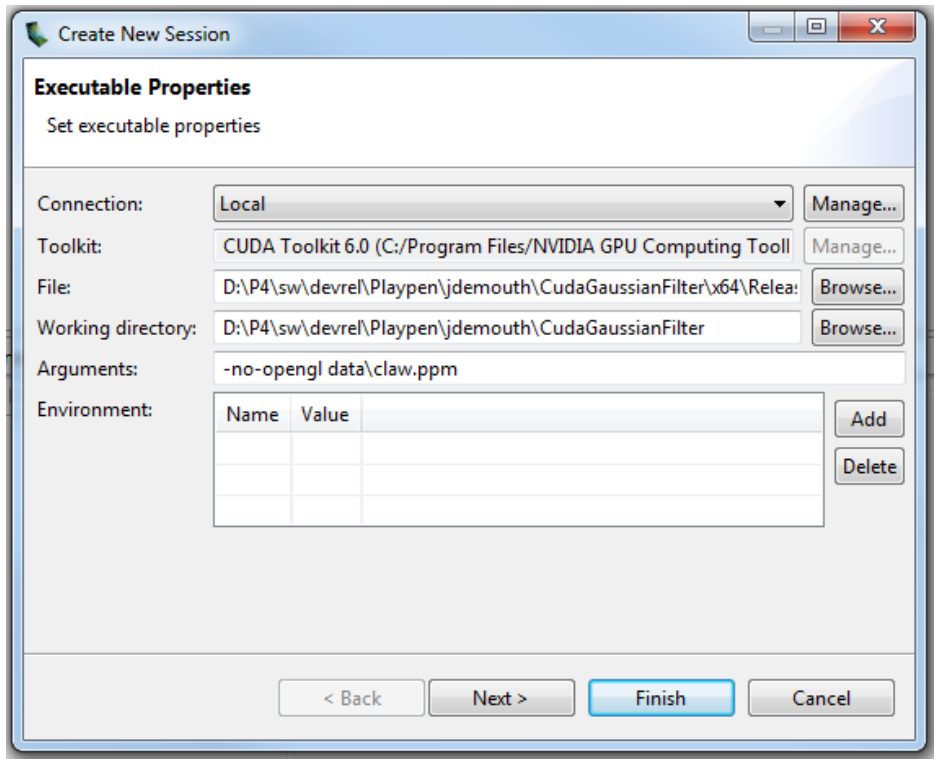
```
 dram_read_throughput  136.55GB/s
dram_write_throughput  70.062GB/s
        flop_count_sp  1000000
        flop_count_dp  0
```

- Problem was addition of 1,000,000 element single precision vectors
  - 1,000,000 single precision flop count is therefore expected!
  - ~50 GFLOP/s
  - About 1% peak FLOPs
- Measured dram throughput is ~206GB/s
  - Peak for this machine is 288 GB/s
  - 72% peak bandwidth

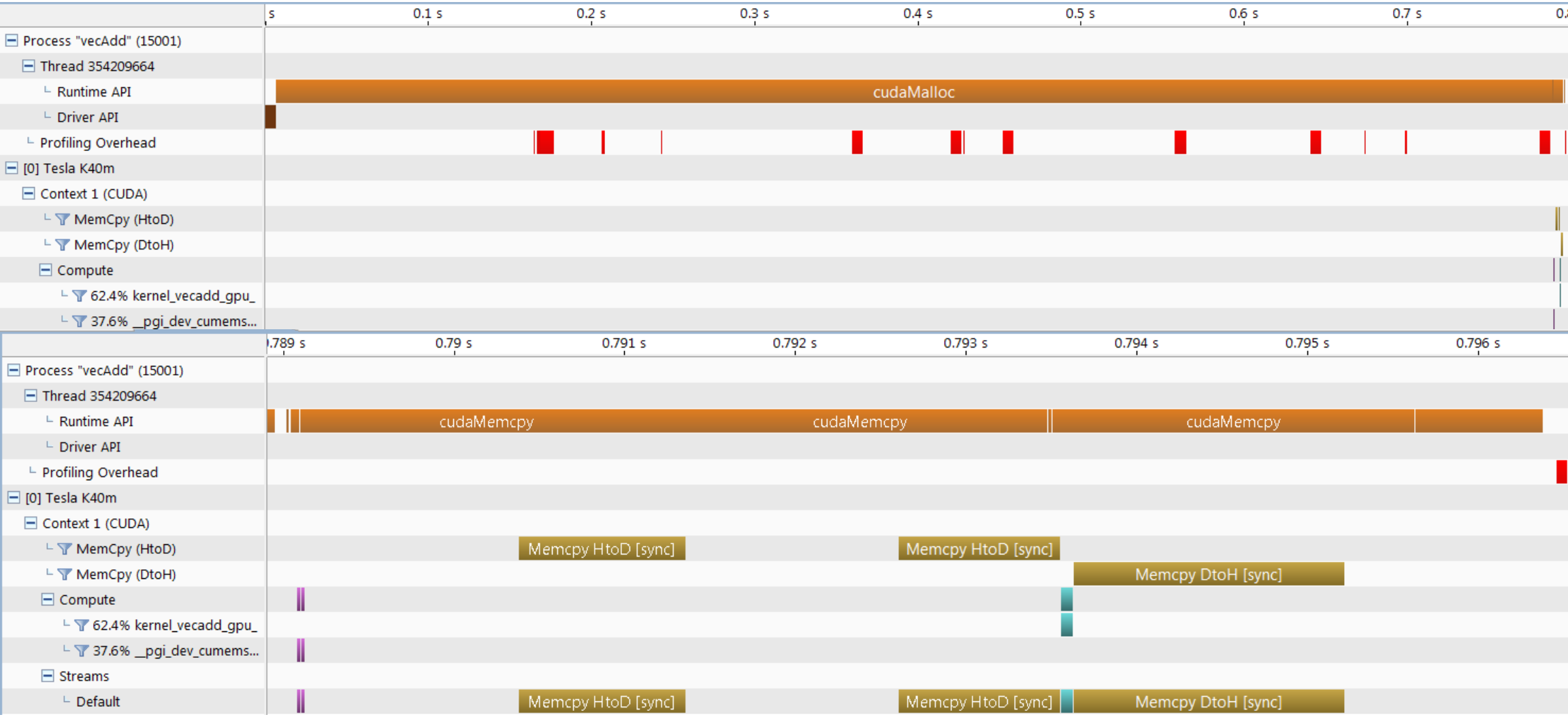
Visual Profiler

nvvp

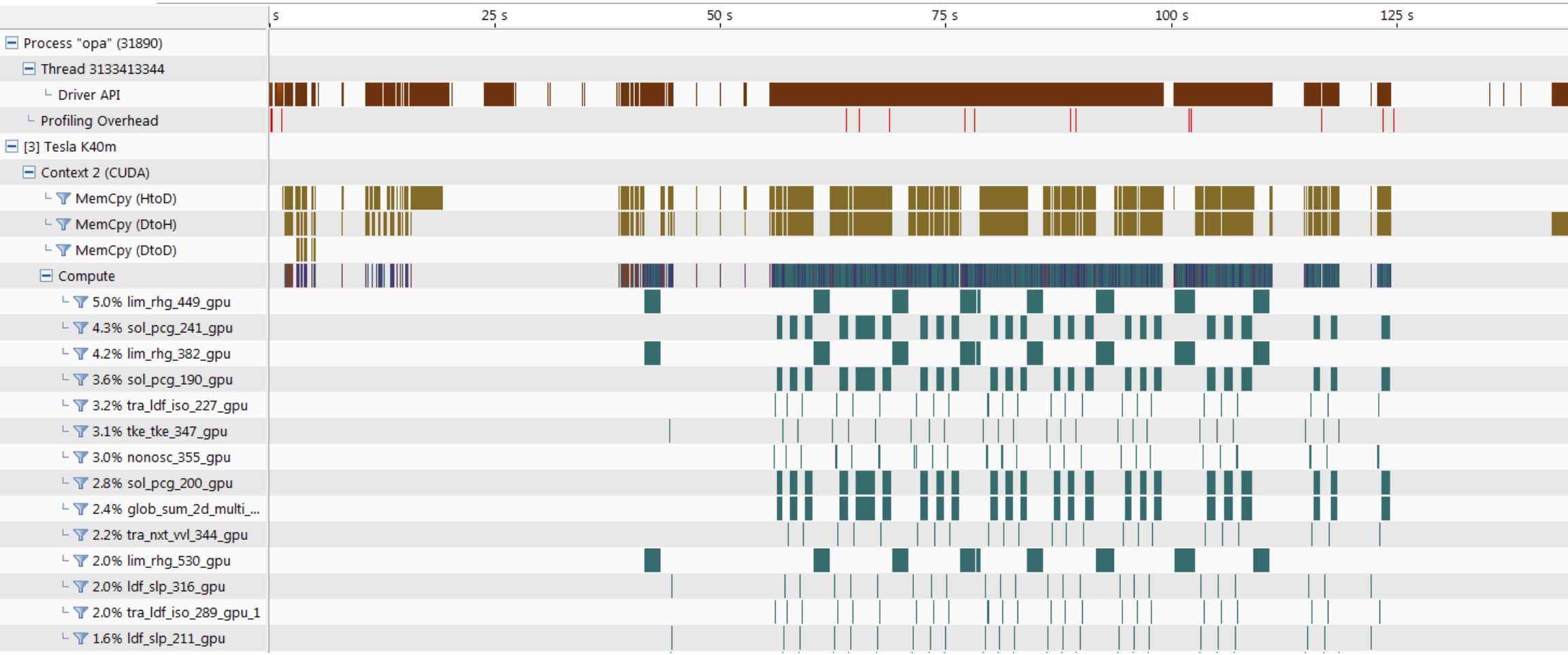
- Either import the output of `nvprof -o ...`
  - Need to select metrics in advance
- Or run application through GUI
  - Re-run on-the-fly to compute new metrics based on requirements



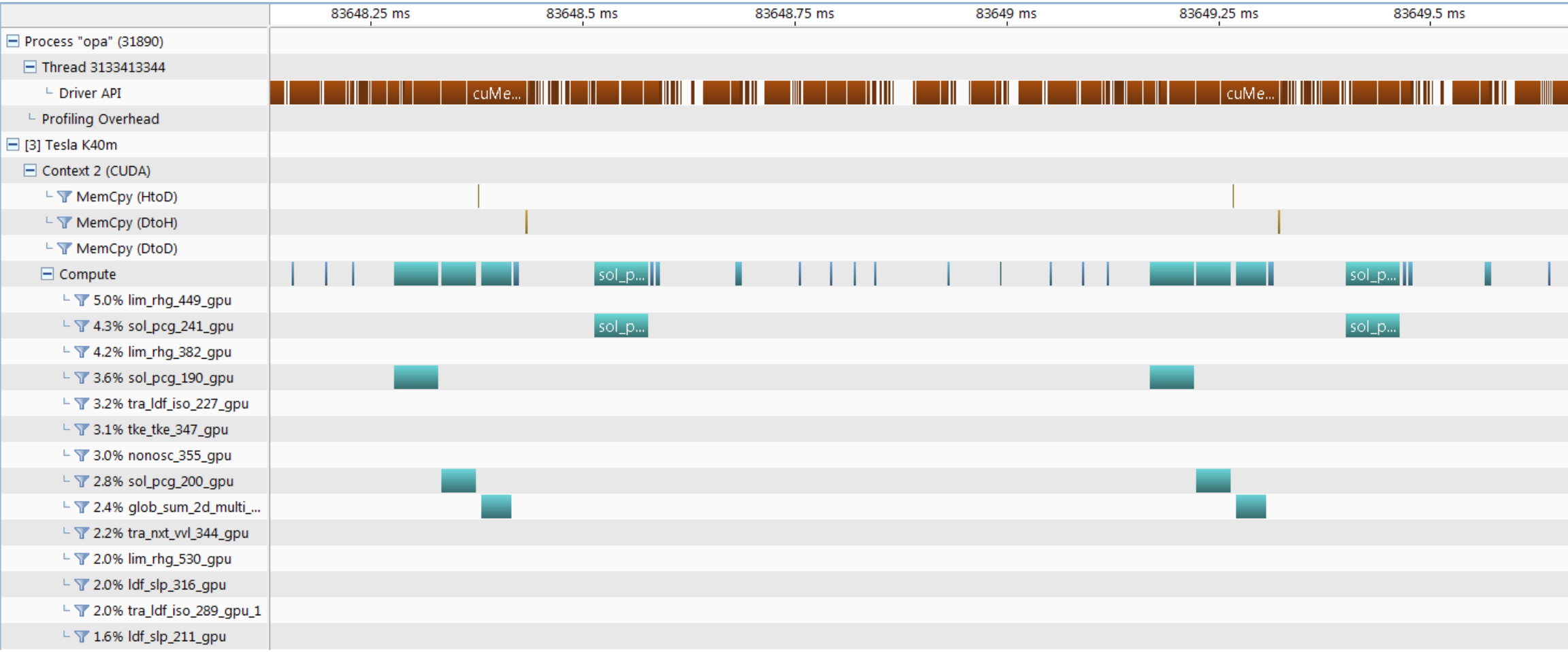
Vecadd - timeline



NEMO - timeline



NEMO - details

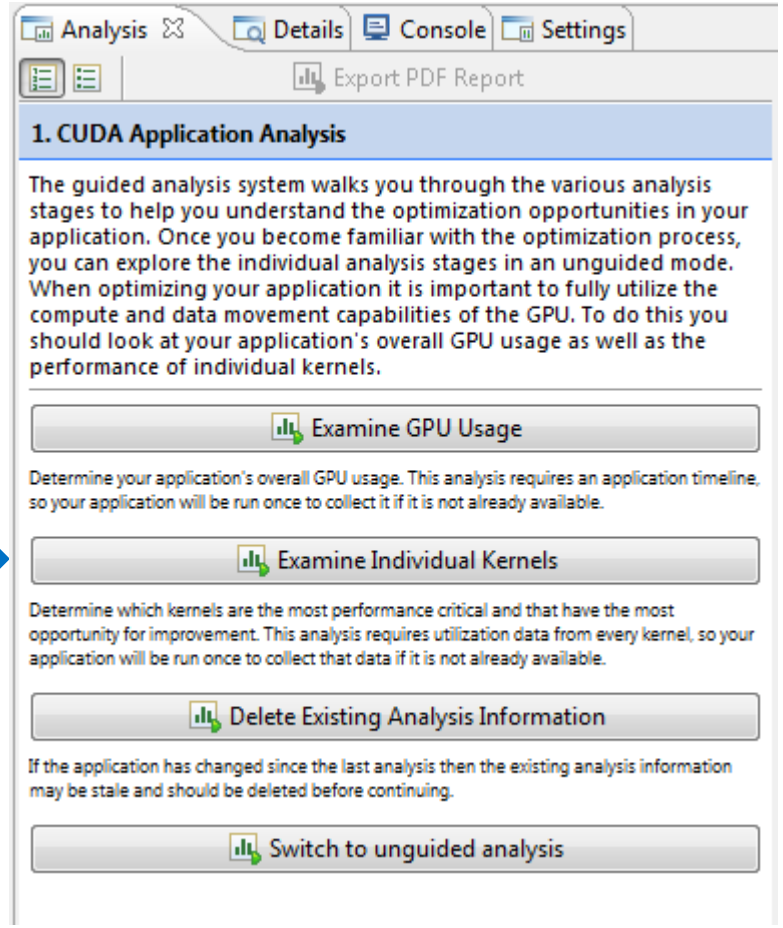


Back to vecadd

More details!

- If we collected our profiling information with –analysis-metrics, or if we're running the application through nvvp directly, more information is available!
- -analysis-metrics may take some time to run
  - Run separately after generating timeline info

This is what we want!



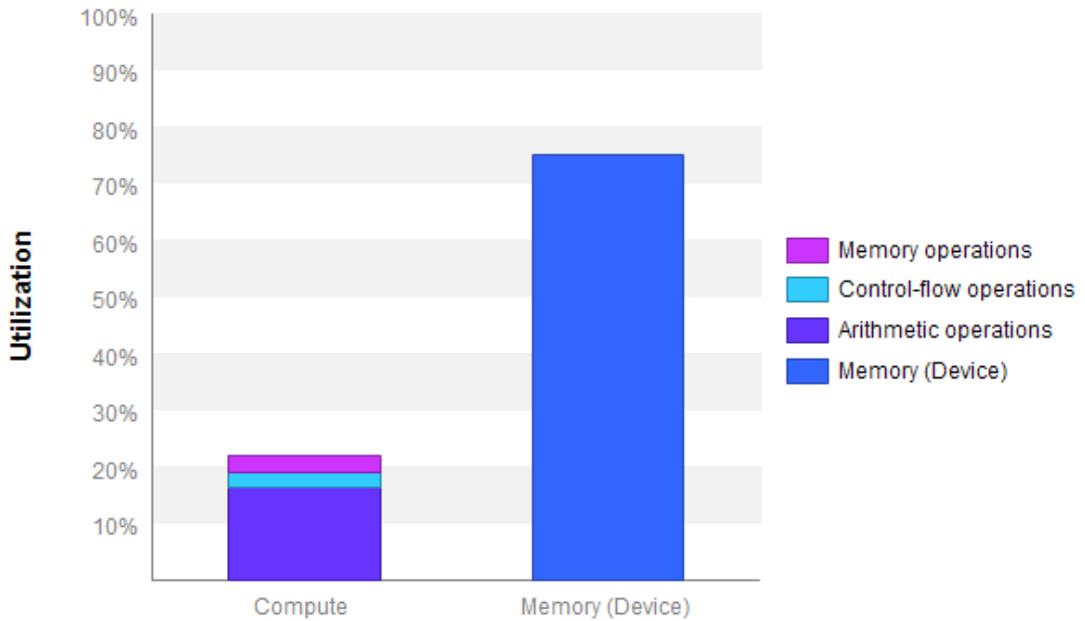
Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.

Rank	Description
100	[1 kernel instances] kernel_vecadd_gpu_
60	[2 kernel instances] __pgi_dev_cumemset_4

Kernel Performance Is Bound By Memory Bandwidth

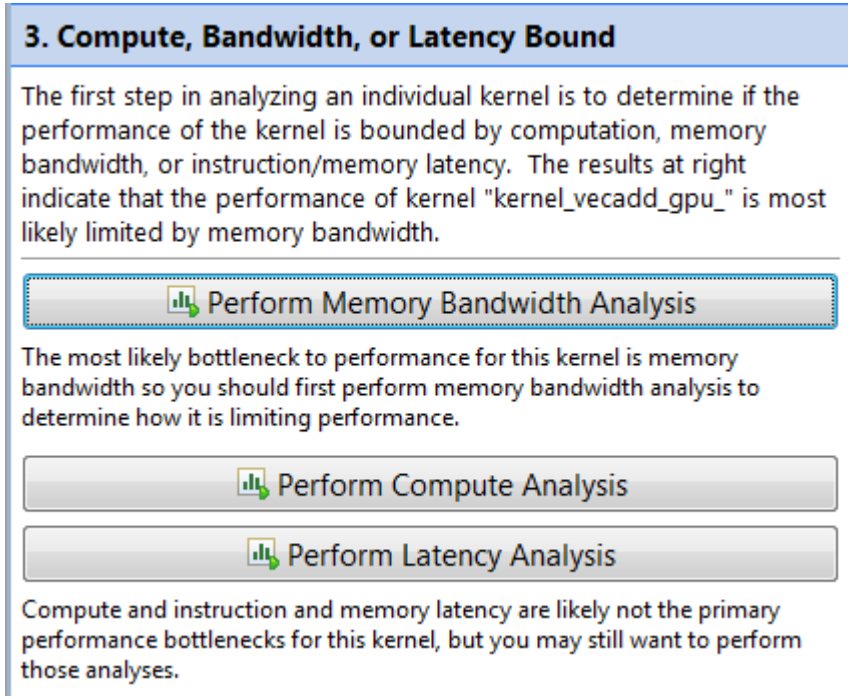
For device "Tesla K40m" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the Device memory.



- Matches our earlier calculations of ~72% peak bandwidth

Compute, Bandwidth or Latency Bound

- Other analysis options available
- The tool will guide you
  - Becomes more useful the more experience you get

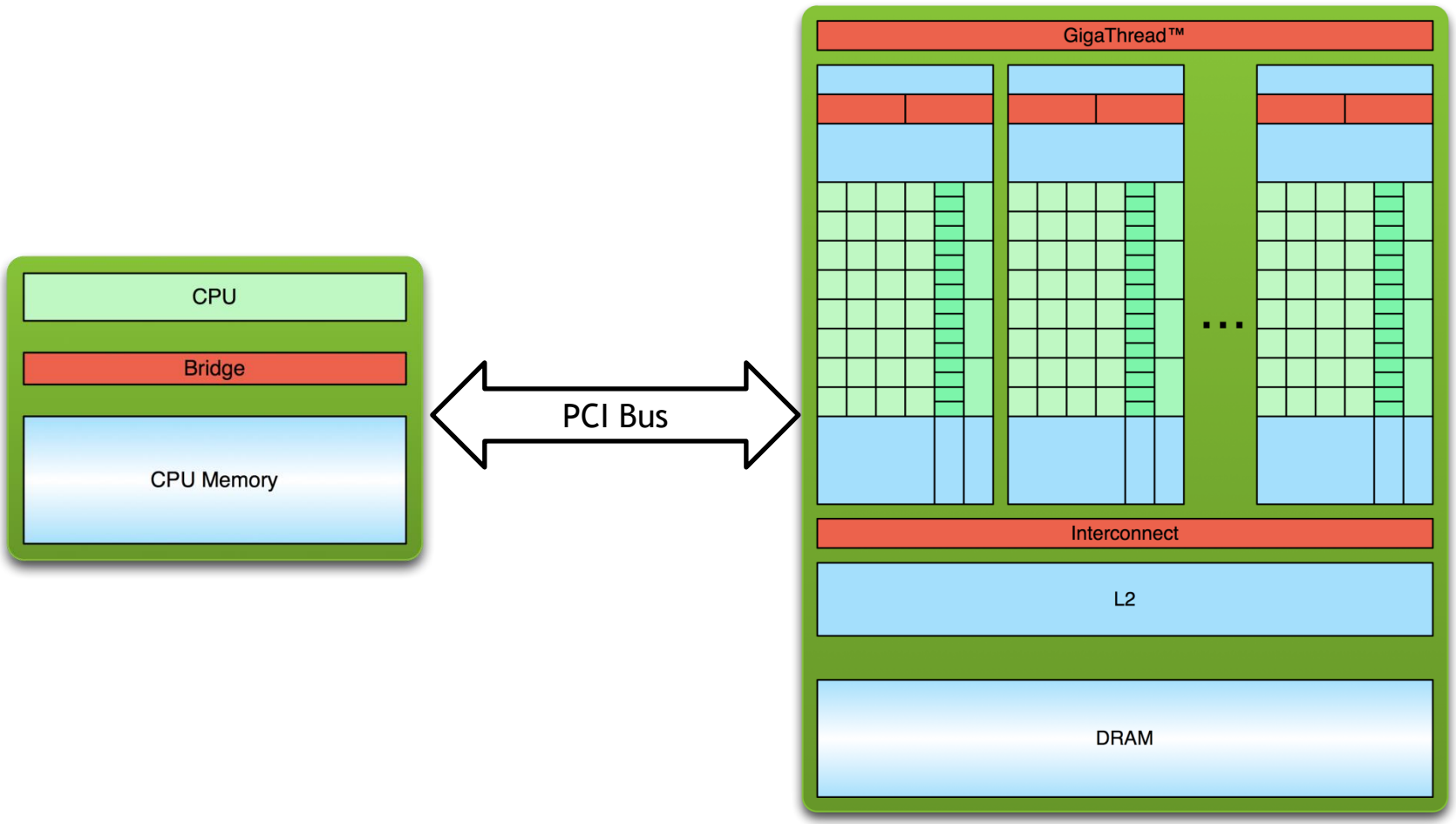


Profiling tips

- The nvprof output is a very good place to start
- The timeline is a good place to go next
- Only dig deep into a kernel if it's taking a significant amount of your time
- Where possible, try to match profiler output with theory
  - For example, if I know I'm moving 1GB, and my kernel takes 10ms, I expect the profiler to report 100GB/s.
  - Discrepancies are likely to mean your application isn't doing what you thought it was

Optimising Data Movement

Heterogeneous Computing



Data movement

What are the costs?

- Two costs:
  - Bandwidth
  - Latency
- Large copies will be bandwidth bound.
  - Peak PCI-E bandwidth is 16 GB/s
- Small copies will be high latency
  - Many small copies are expensive

Data movement

Several optimisation options

- Several ways to optimise data movement:
  1. Do less of it!
  2. Fuse small copies into larger ones
  3. Use pinned buffers
  4. Overlap data movement with execution

- Remember: profile before optimising!

Optimising Data Movement

1. Do less of it!

- Consider moving data movement outside of main loop
- For example, an iterative solver would not want data movement within the iteration loop
- Often, during the process of making an application run on a GPU these additional copies are unavoidable
    - If doing intermediate performance analysis, bear in mind these copies may disappear in the final application
- Consider if data can be passed as an argument (CUDA), or declared as private (OpenACC)

Optimising Data Movement

2. Fuse small copies into larger ones

- Small data transfers are much less efficient
- If the opportunity arises, merging copies can be beneficial
- Can be difficult to accomplish in some applications
    - Additional cost due to packing/unpacking may outweigh the savings

Optimising Data Movement

3. Use pinned buffers

- Page-locked (or pinned) allocations generally have higher bandwidth
- Can harm system performance if overused
- In CUDA:
  - cudaHostAlloc() – allocate page-locked memory
  - cudaHostRegister() – convert normal allocation to page-locked
  - Pinned attribute in CUDA Fortran
- In OpenACC:
  - (Hopefully) done automatically!

Optimising Data Movement

4. Overlap data movement with execution

- With pinned memory PCI-E transfers can occur at the same time as kernel execution
- In CUDA:
  - cudaMemcpyAsync
- In OpenACC:
  - async clause on parallel or kernels directives
- This can greatly reduce the cost of memory copies – almost to none in the right case

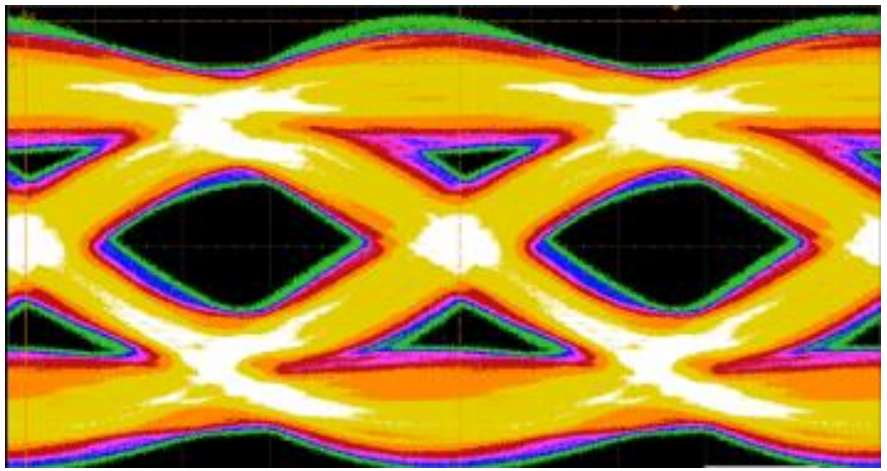
Optimising Data Movement

Summary

- There are good (and bad) ways of moving data between GPU and CPU
- For many applications this is not imporant
  - Data resides on the GPU for the application’s lifetime
- Next-gen GPU Pascal will bring hardware improvements - NVLINK

NVLINK

Faster communications between processors

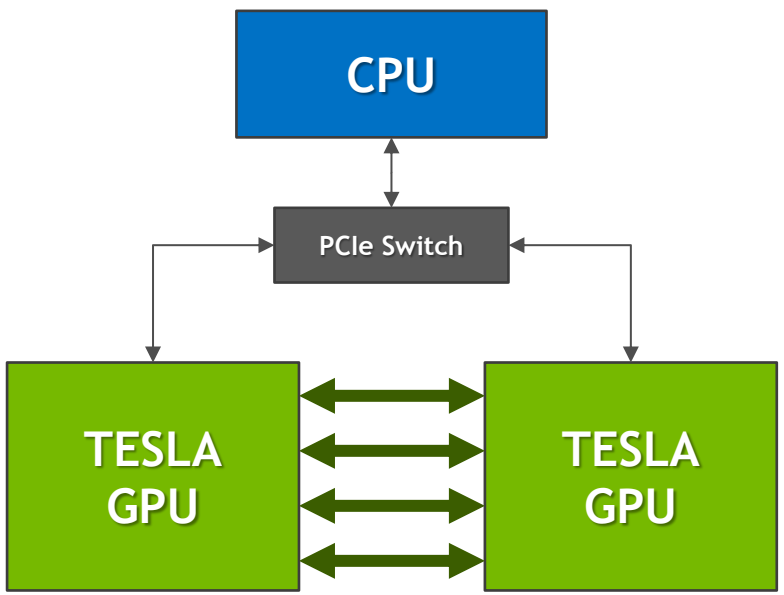


NVLINK

- High speed interconnect
  - Alongside/replacing PCI-E
- 80-200 GB/s
- Improved energy efficiency
- Improved flexibility

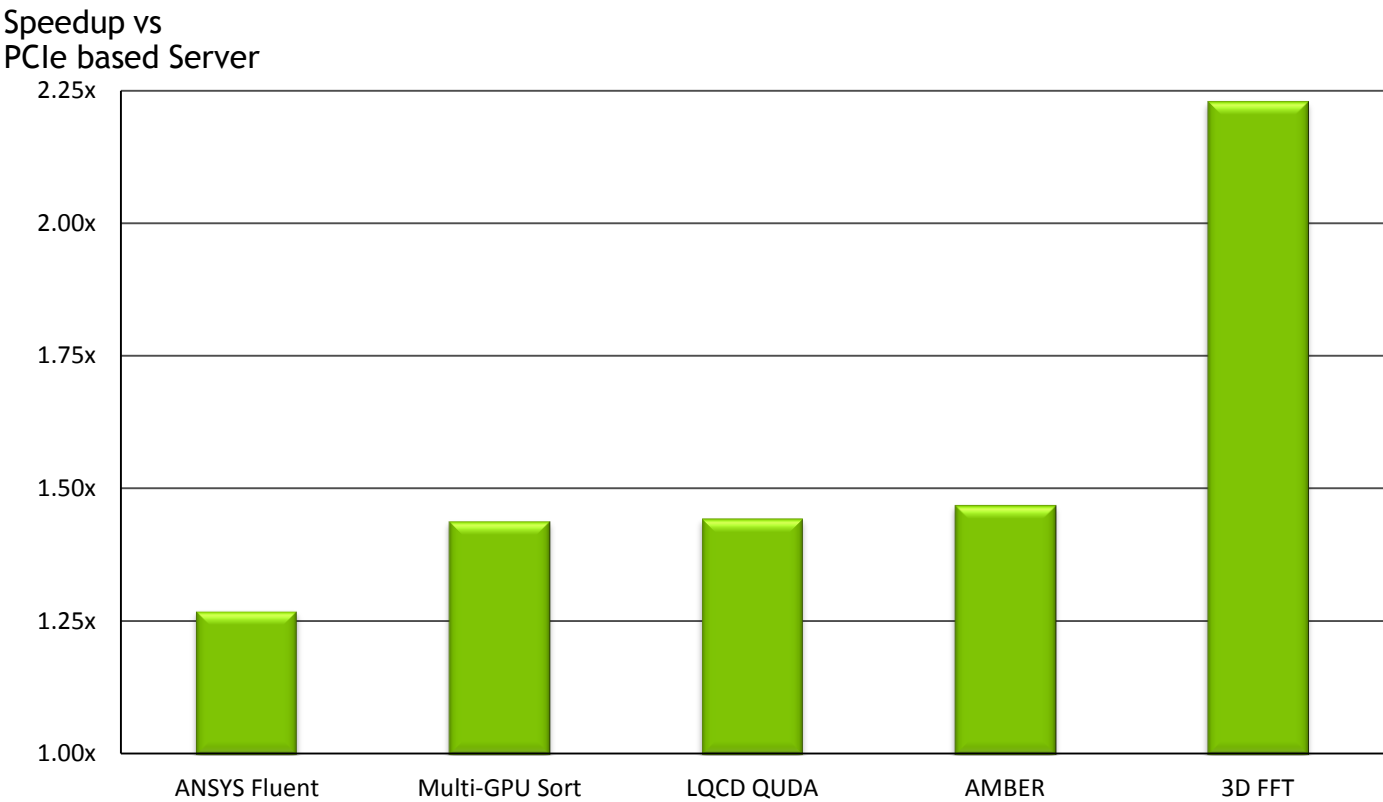
NVLink Unleashes Multi-GPU Performance

GPUs Interconnected with NVLink



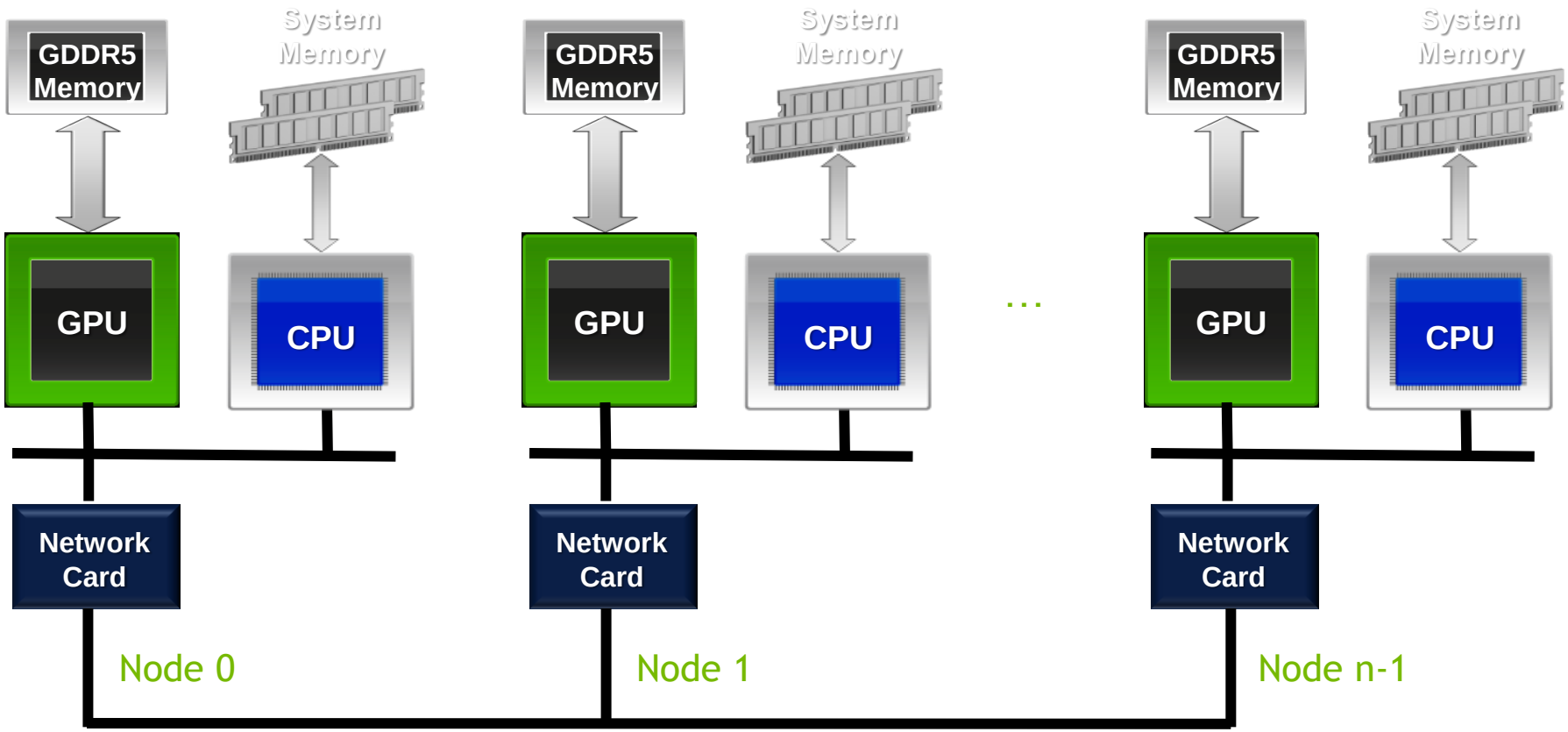
5x Faster than
PCIe Gen3 x16

Over 2x Application Performance Speedup
When Next-Gen GPUs Connect via NVLink Versus PCIe

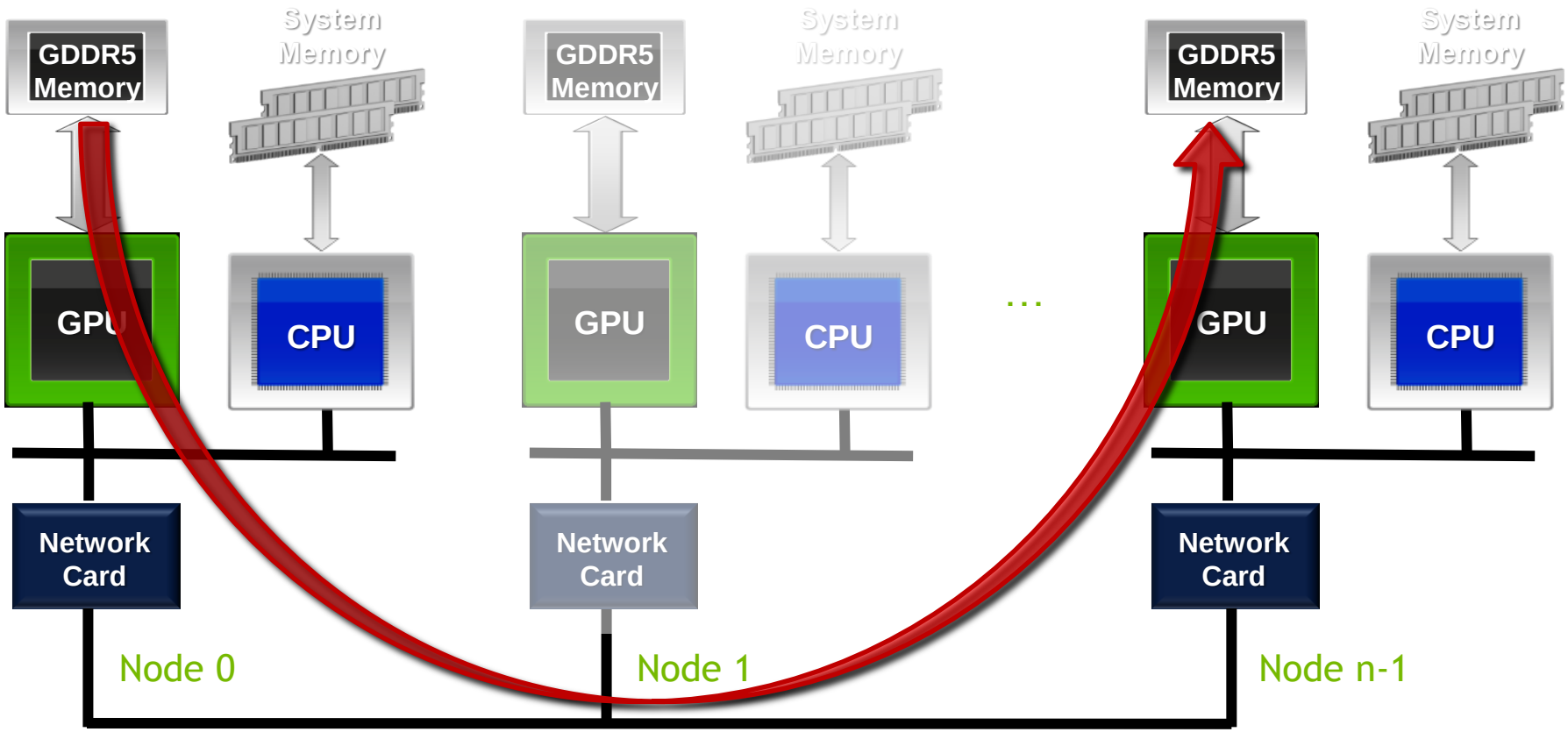


Using MPI with GPUs

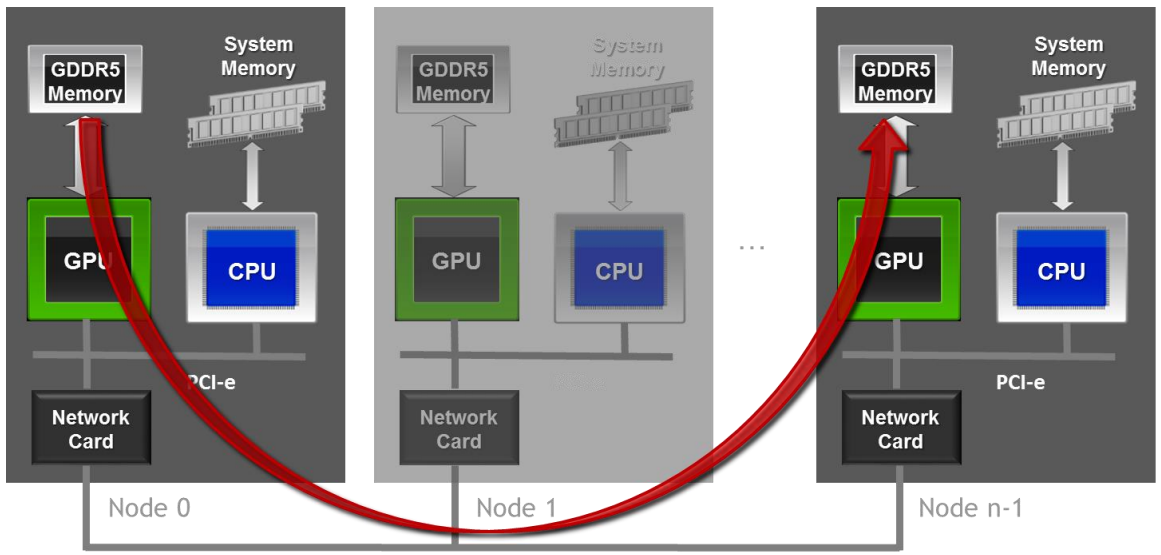
MPI+CUDA


GDDR5 Memory
System Memory
GPU
CPU
Network Card
Node 0
GDDR5 Memory
System Memory
GPU
CPU
Network Card
Node 1
...
GDDR5 Memory
System Memory
GPU
CPU
Network Card
Node n-1

MPI+CUDA


GDDR5 Memory
System Memory
GPU
CPU
Network Card
GDDR5 Memory
System Memory
GPU
CPU
Network Card
...
GDDR5 Memory
System Memory
GPU
CPU
Network Card
Node 0
Node 1
Node n-1

MPI+CUDA




```
//MPI rank 0
MPI_Send(s_buf_d, size, MPI_CHAR, n-1, tag, MPI_COMM_WORLD);

//MPI rank n-1
MPI_Recv(r_buf_d, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &stat);
```

Message Passing Interface
MPI

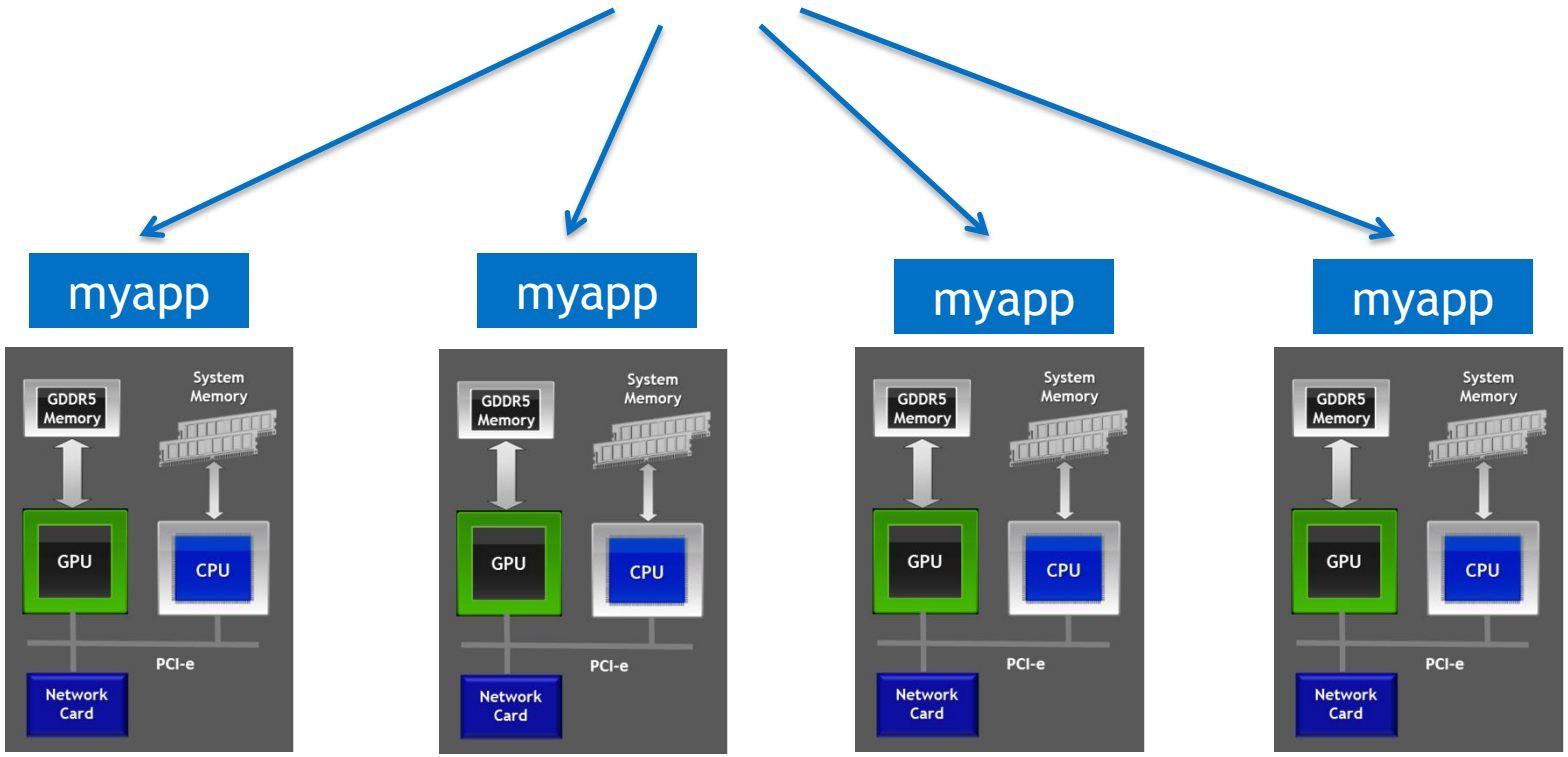
- Standard to exchange data between processes via messages
  - Defines API to exchanges messages
    - Pt. 2 Pt.: e.g. MPI_Send, MPI_Recv
    - Collectives, e.g. MPI_Reduce
- Multiple implementations (open source and commercial)
  - Binding for C/C++, Fortran, Python, …
  - E.g. MPICH, OpenMPI, MVAPICH, IBM Platform MPI, Cray MPT, …

MPI

Compiling and launching

```
$ mpicc -o myapp myapp.c

$ mpirun -np 4 ./myapp <args>
```



Launch MPI + CUDA/OpenACC programs

- Launch one process per GPU
  - MVAPICH: `MV2_USE_CUDA`

```
$ MV2_USE_CUDA=1 mpirun -np ${np} ./myapp <args>
```

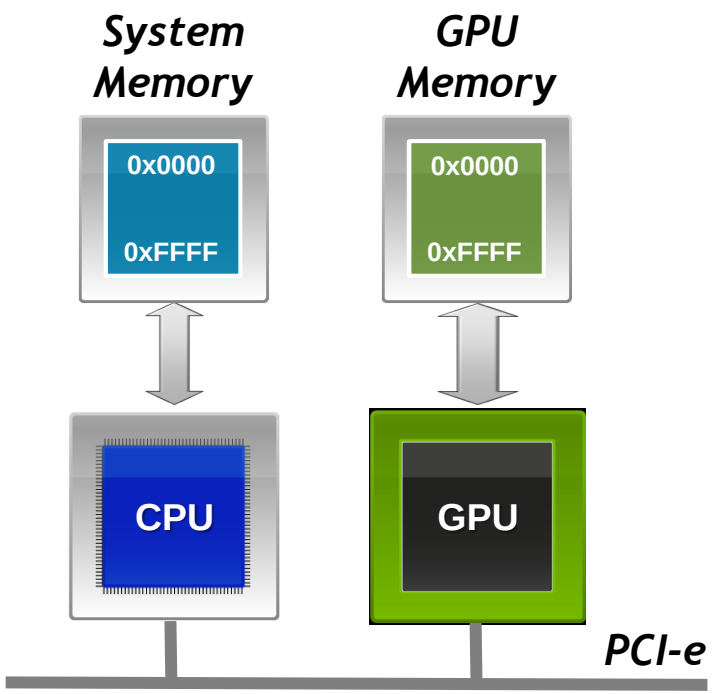
  - Open MPI: CUDA-aware features are enabled per default
  - Cray: `MPICH_RDMA_ENABLED_CUDA`
  - IBM Platform MPI: `PMPI_GPU_AWARE`

Unified Virtual Addressing

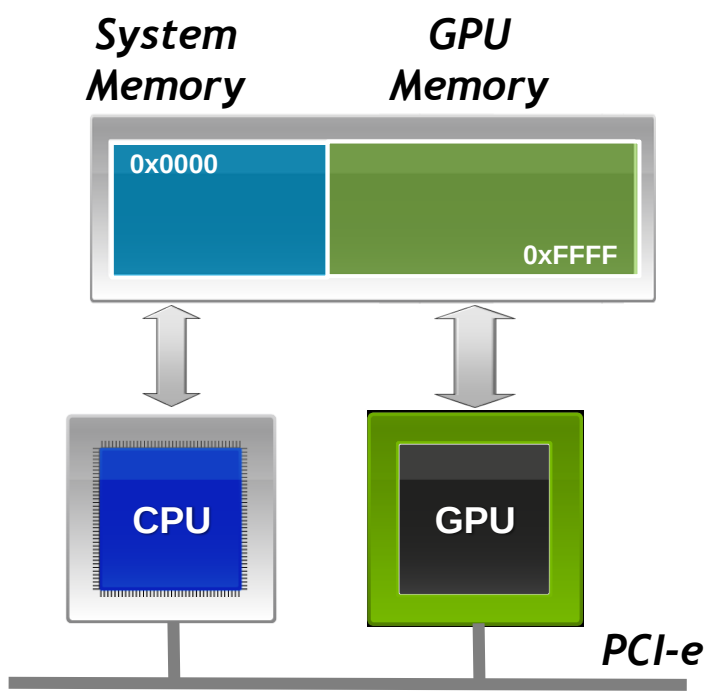
- One address space for all CPU and GPU memory
  - Determine physical memory location from a pointer value
  - Enable libraries to simplify their interfaces (e.g. MPI and cudaMemcpy)
- Support:
  - 64-bit applications on Linux
  - Windows using TCC mode

Unified Virtual Addressing

No UVA : Separate Address Spaces



UVA : Single Address Space



MPI + CUDA

With UVA CUDA-aware MPI

```
//MPI rank 0
call MPI_Send(s_buf_d,size,…)

//MPI rank n-1
call MPI_Recv(r_buf_d,size,…)
```

No UVA and regular MPI

```
//MPI rank 0
s_buf_h = s_buf_d
call MPI_Send(s_buf_h,size,…)

//MPI rank n-1
call MPI_Recv(r_buf_h,size,…)
r_buf_d = r_buf_h
```

MPI + OpenACC

With UVA CUDA-aware MPI

```
!$acc host_data use_device (s_buf, r_buf)
!MPI rank 0
call MPI_Send(s_buf,size,…)

!MPI rank n-1
call MPI_Recv(r_buf,size,…)
!$acc end host_data
```

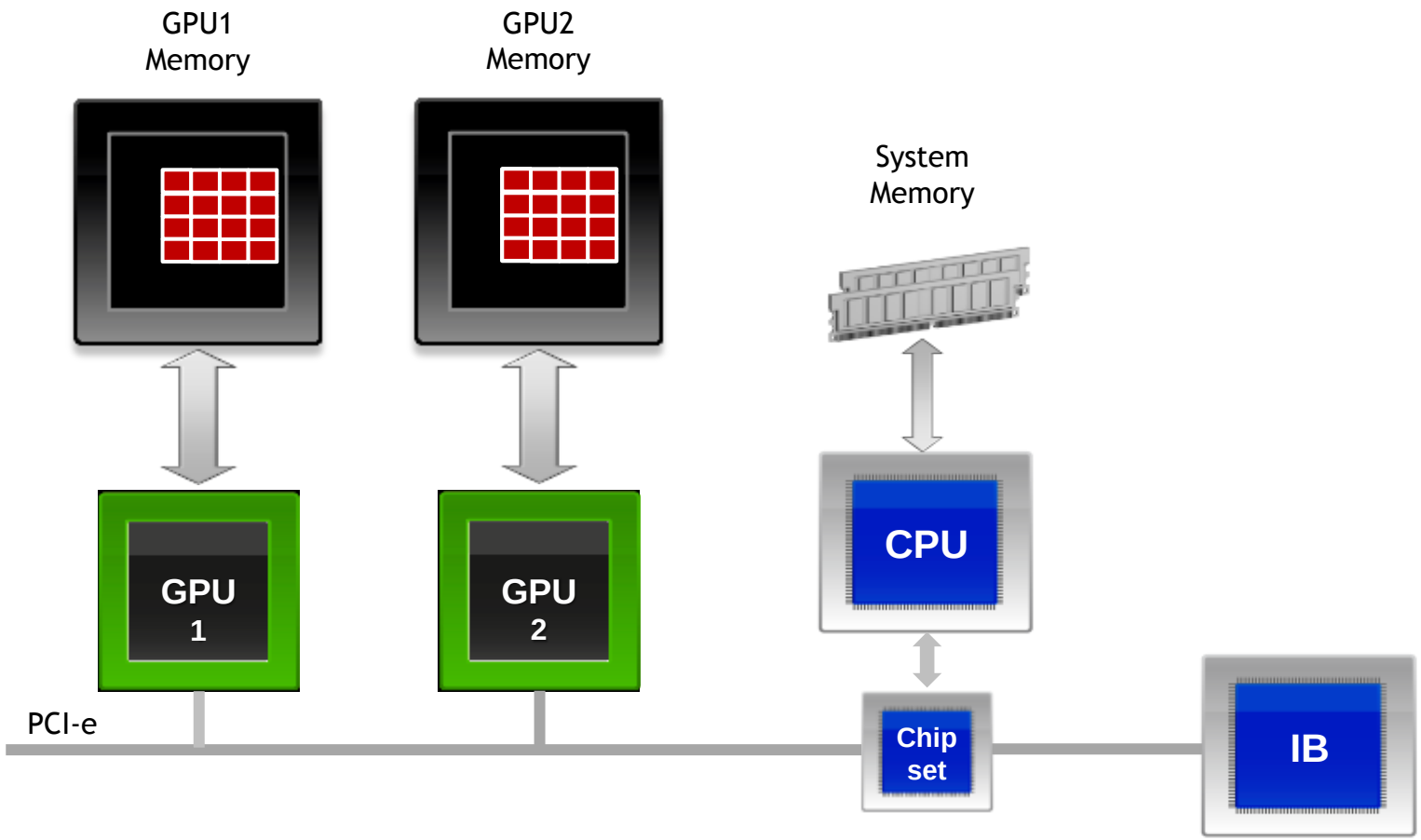
No UVA and regular MPI

```
!$acc update host(s_buf)
!MPI rank 0
call MPI_Send(s_buf,size,…)

!MPI rank n-1
call MPI_Recv(r_buf,size,…)
!$acc update device(r_buf)
```

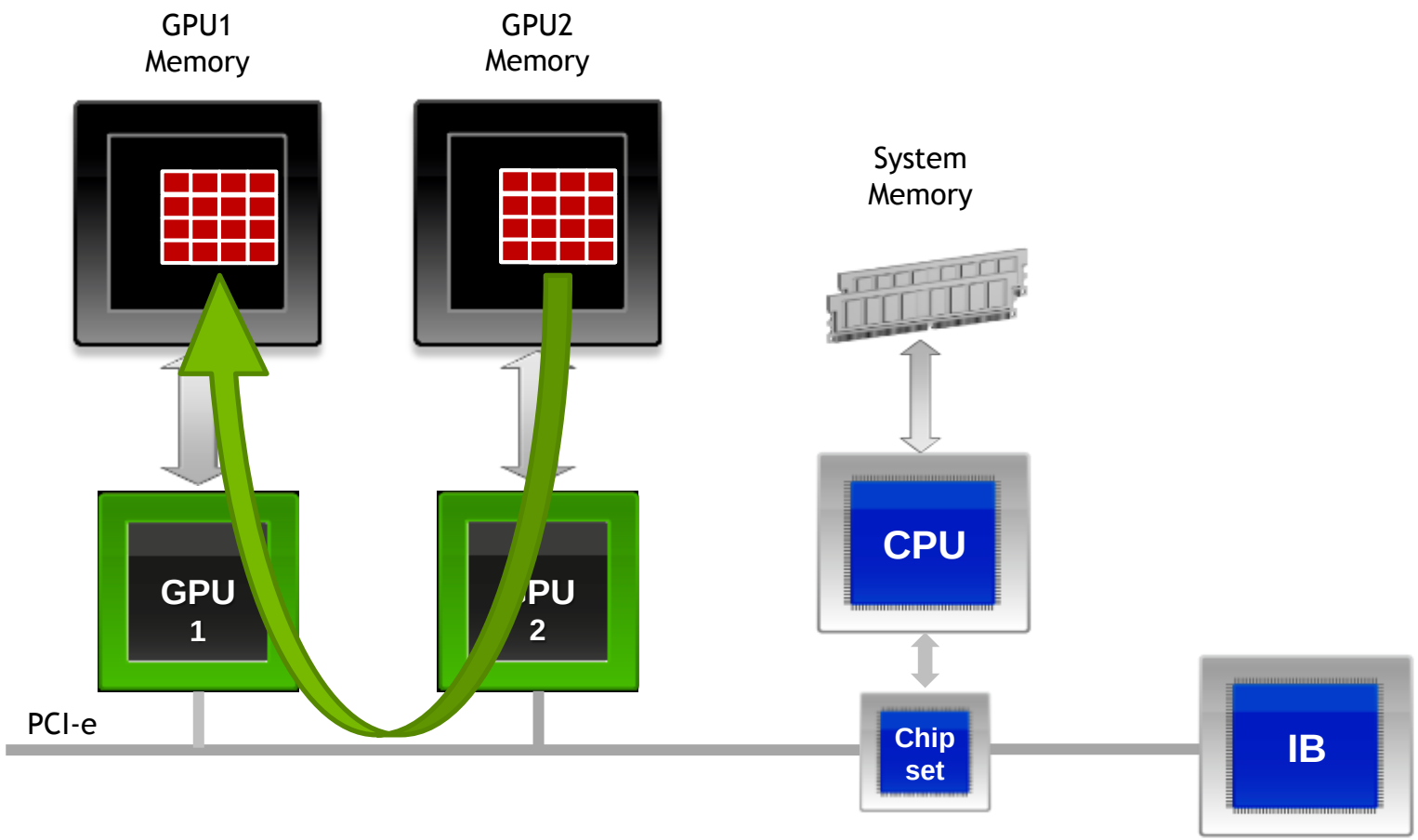
NVIDIA GPUDirect™

Peer to Peer tranfers



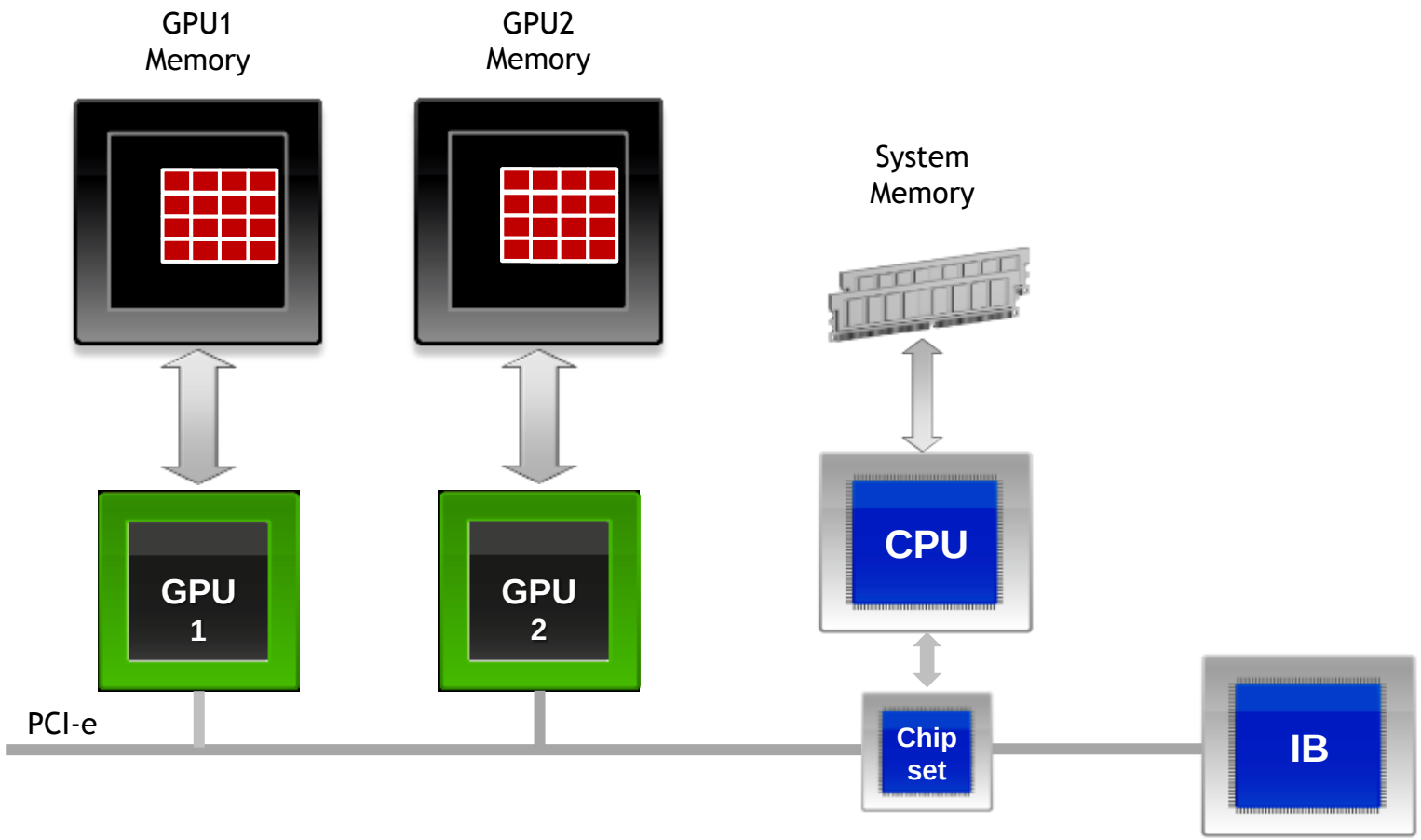
NVIDIA GPUDirect™

Peer to Peer tranfers



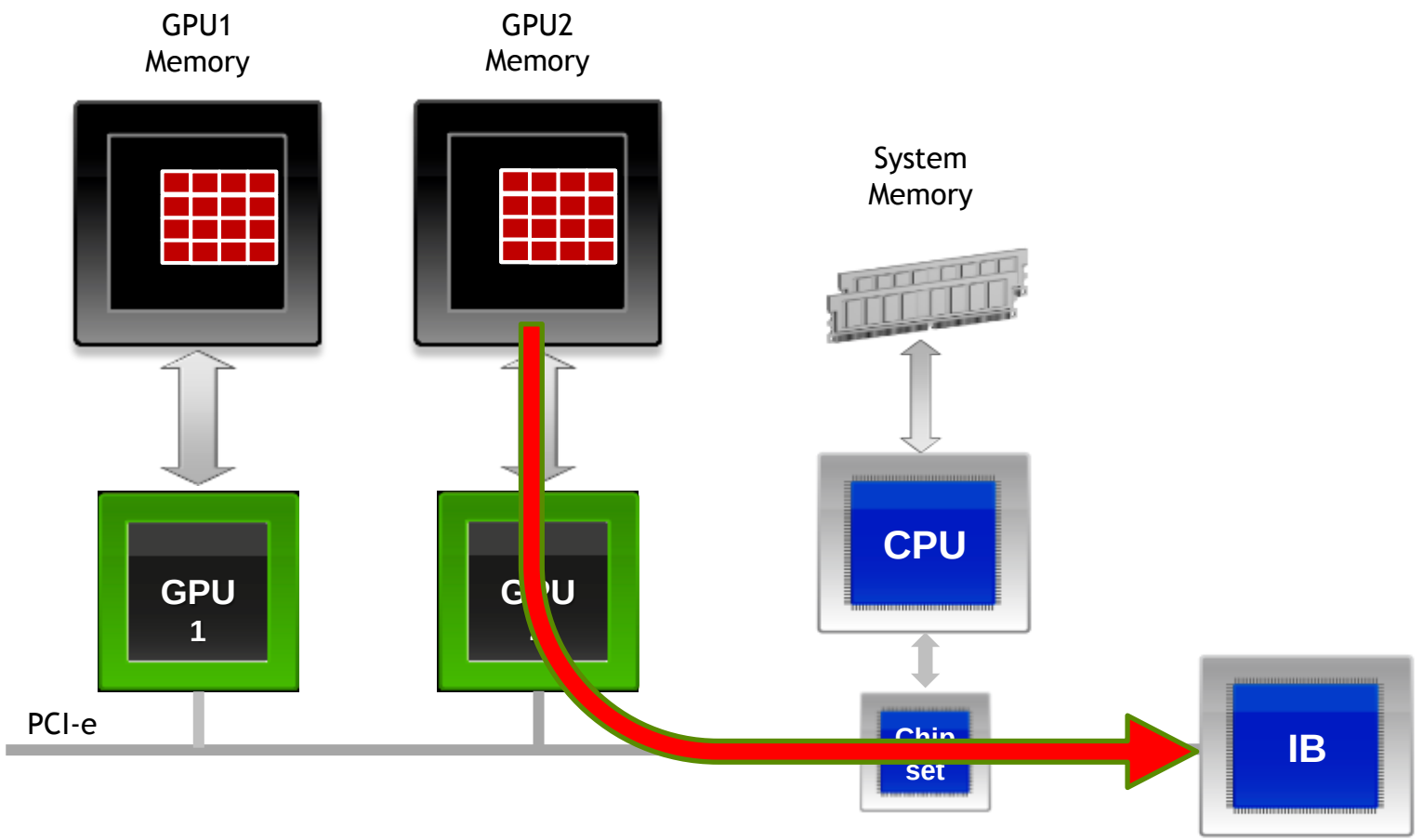
NVIDIA GPUDirect™

Support for RDMA

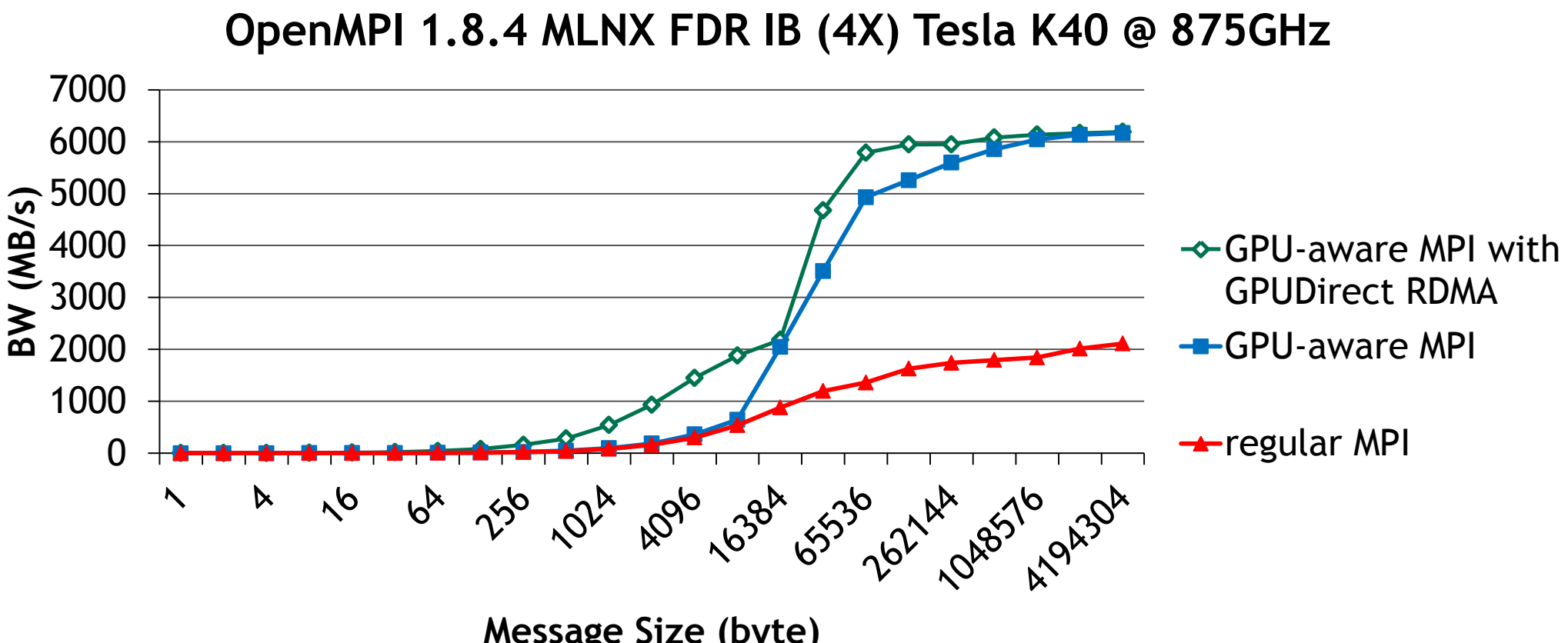


NVIDIA GPUDirect™

Support for RDMA



Performance Results two Nodes



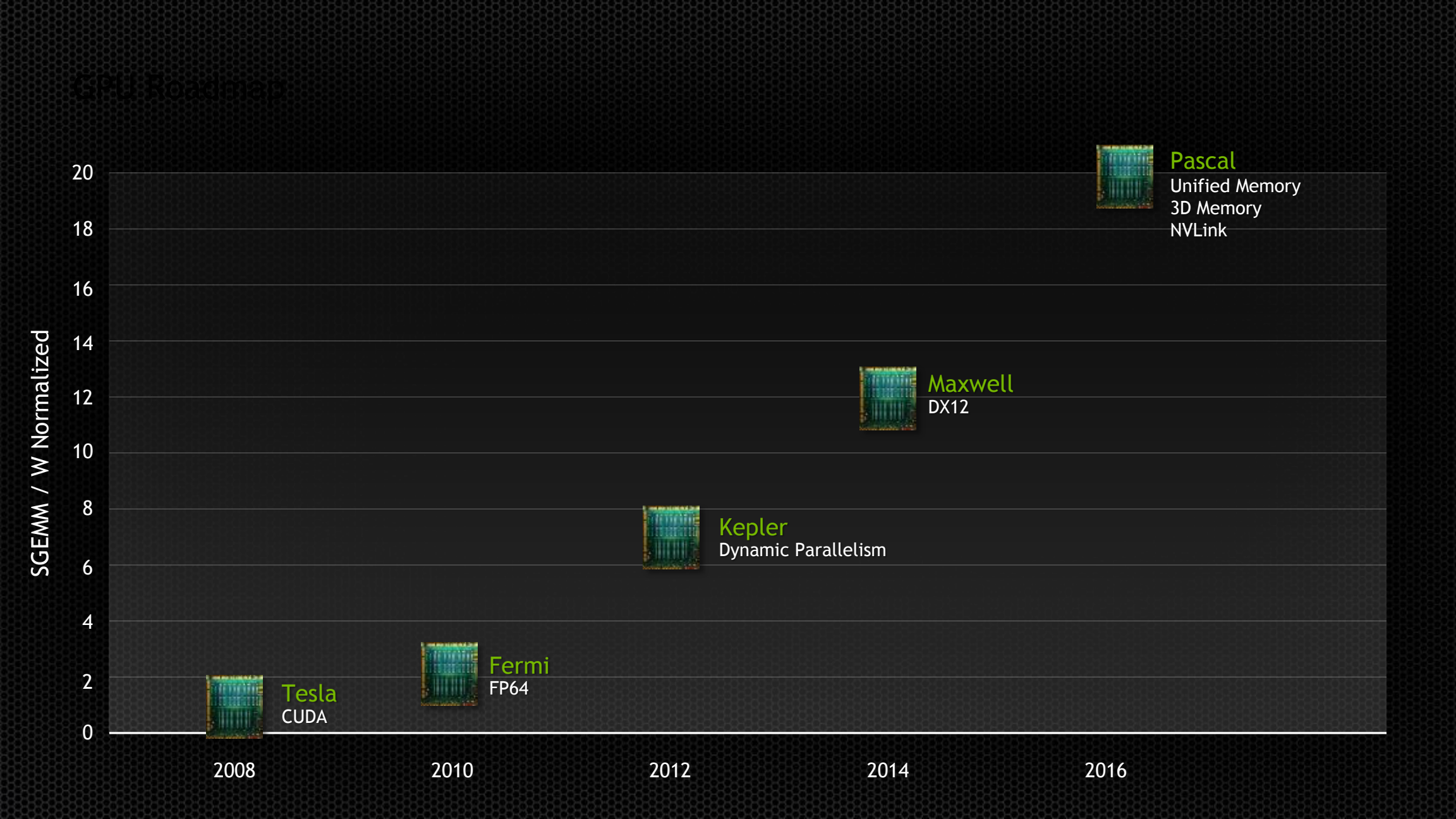
Latency (1 byte)	19.79 us	17.97 us	5.70 us

In this talk

- Profiling GPU applications
- Optimizing Data Movement
- Using MPI with GPUs

Any questions?

Bonus slides - Pascal!

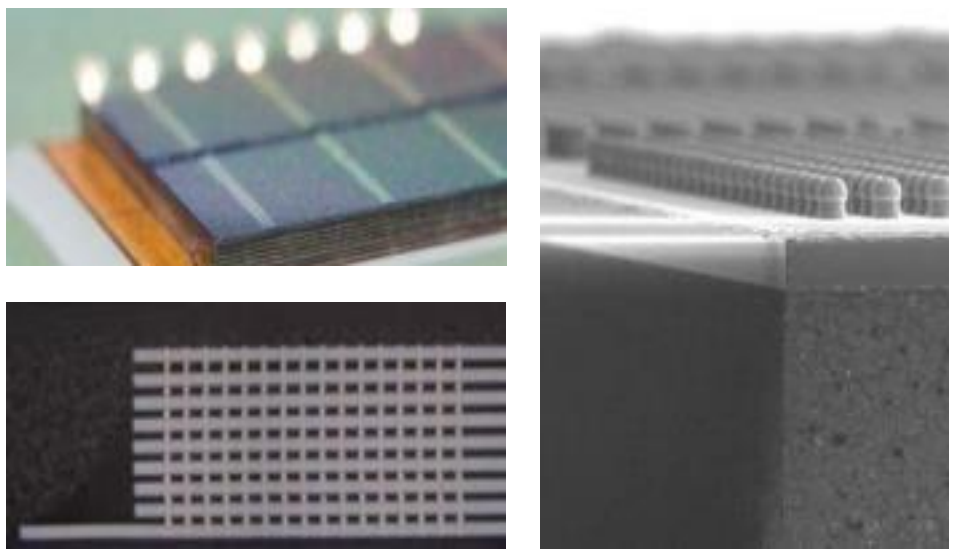

GPU Roadmap
SGEMM / W Normalized
20
18
16
14
12
10
8
6
4
2
0
2008
2010
2012
2014
2016
Tesla
CUDA
Fermi
FP64
Kepler
Dynamic Parallelism
Maxwell
DX12
Pascal
Unified Memory
3D Memory
NVLink

Pascal

- Faster and larger global memory
- Faster communication between processors
- More powerful unified memory
- Mixed precision computing

Stacked Memory

High performance global memory

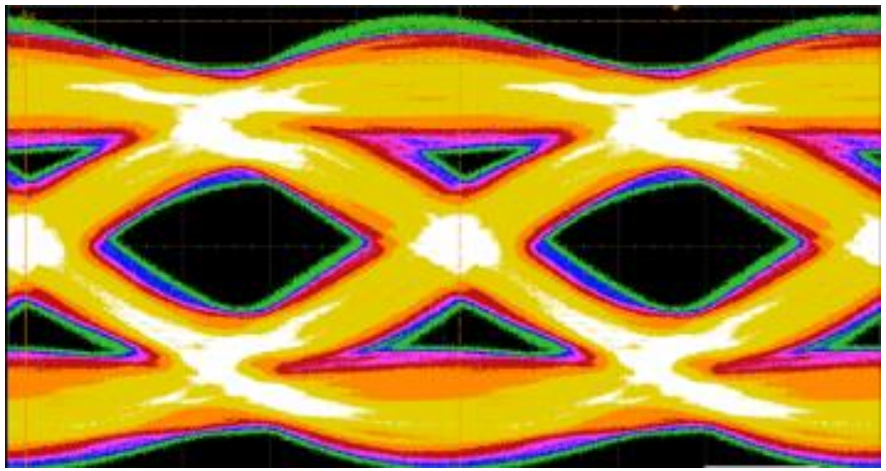


3D Stacked Memory

- 4x Higher Bandwidth (~1 TB/s)
- 3x Larger Capacity (up to 32GB)
- 4x More Energy Efficient per bit

NVLINK

Faster communications between processors

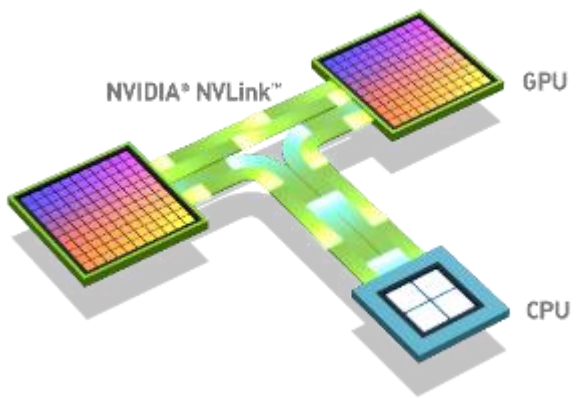


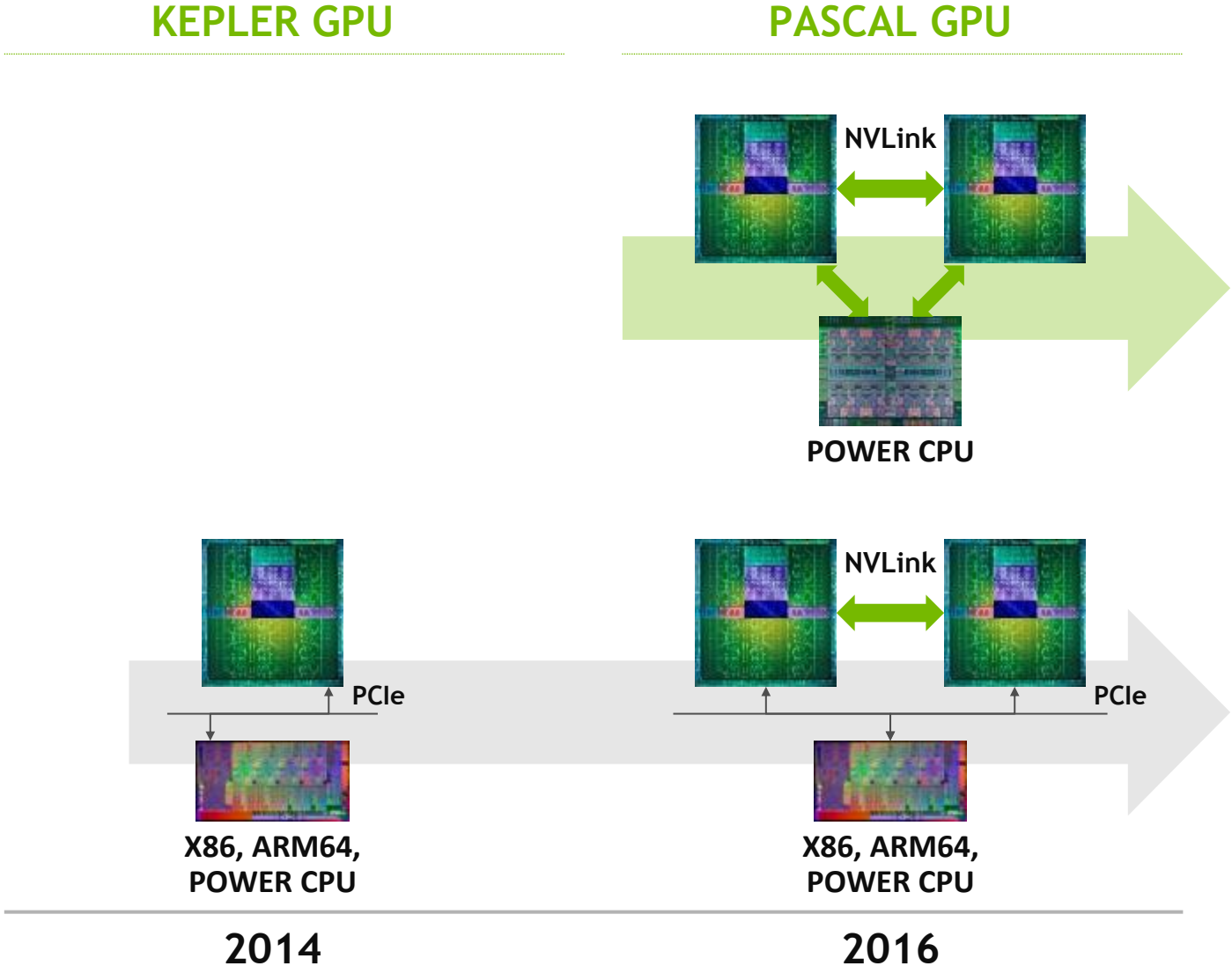
NVLINK

- High speed interconnect
  - Alongside/replacing PCI-E
- 80-200 GB/s
- Improved energy efficiency
- Improved flexibility

NVLink

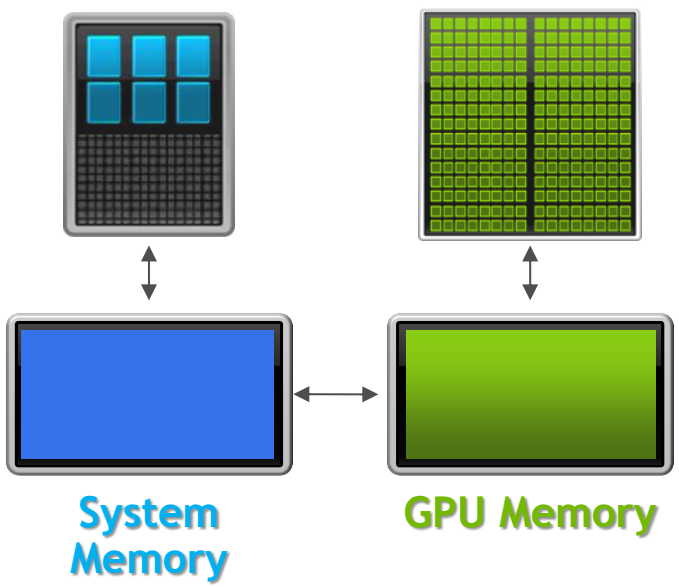
High-Speed GPU Interconnect



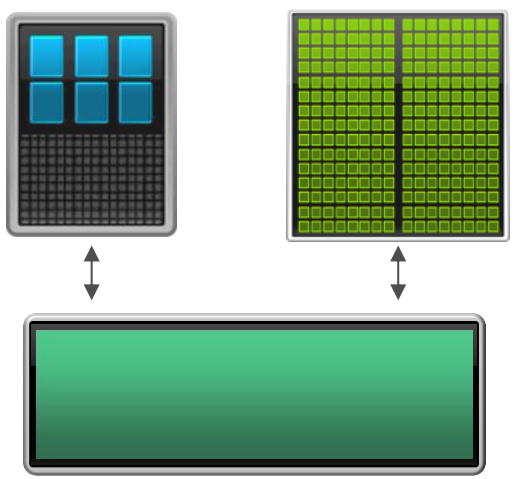


Unified Memory: Simpler & Faster with NVLink

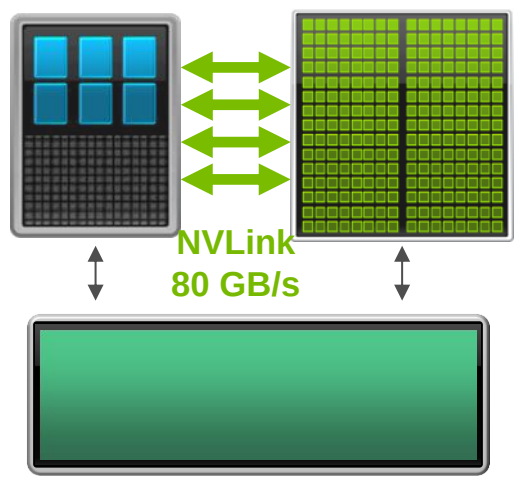
Traditional Developer View



Developer View With Unified Memory



Developer View With Pascal & NVLink



Share Data Structures at
CPU Memory Speeds, not PCIe speeds
Oversubscribe GPU Memory

Mixed precision computing

IEEE 16-bit float support

- Halving precision results in:
  - Half the memory footprint
  - Half the bandwidth required
  - Double the computational throughput
- Obviously comes at an accuracy penalty
  - Application dependent
  - 16/32/64 bit all supported
- Supported in software now