FORTH
INSTITUTE OF COMPUTER SCIENCE

Unlocking the Power of GPUs: A Comprehensive Guide

Manos Pavlidakis
Institute of Computer Science, Foundation for Research and Technology - Hellas, Greece
manospavl@ics.forth.gr

Central Processing Unit (CPU)

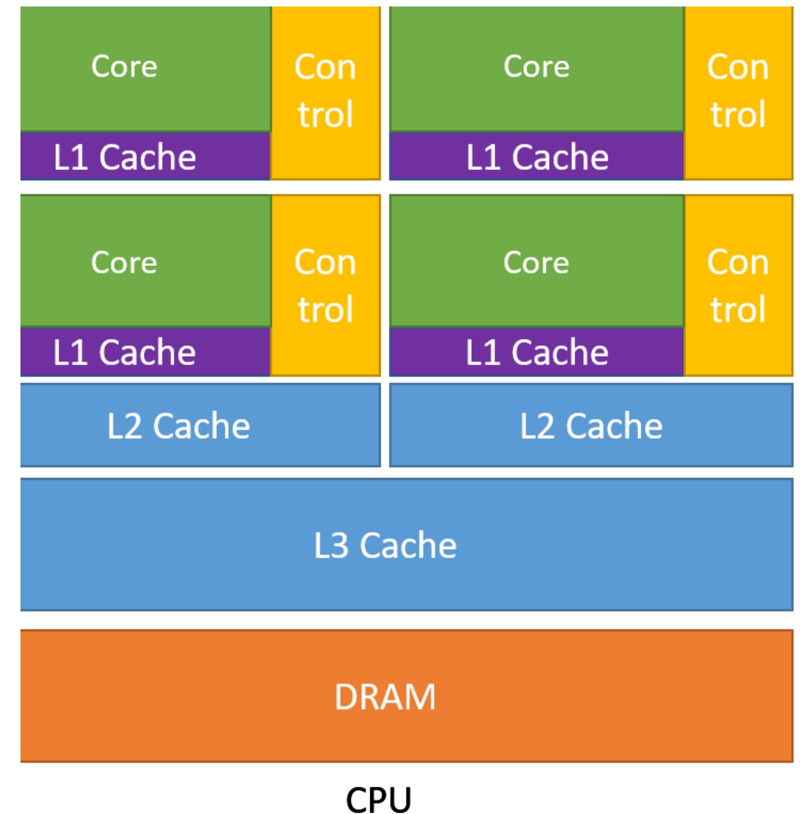
- Is a latency-reducing architecture
- Optimized for serial tasks

+ Strengths

- Very large main memory
- Very fast clock speeds
- Latency optimized via large caches
- Small number of threads can run very quickly

- Weaknesses

- Relatively low memory bandwidth
- Cache misses are very costly
- Low performance/watt

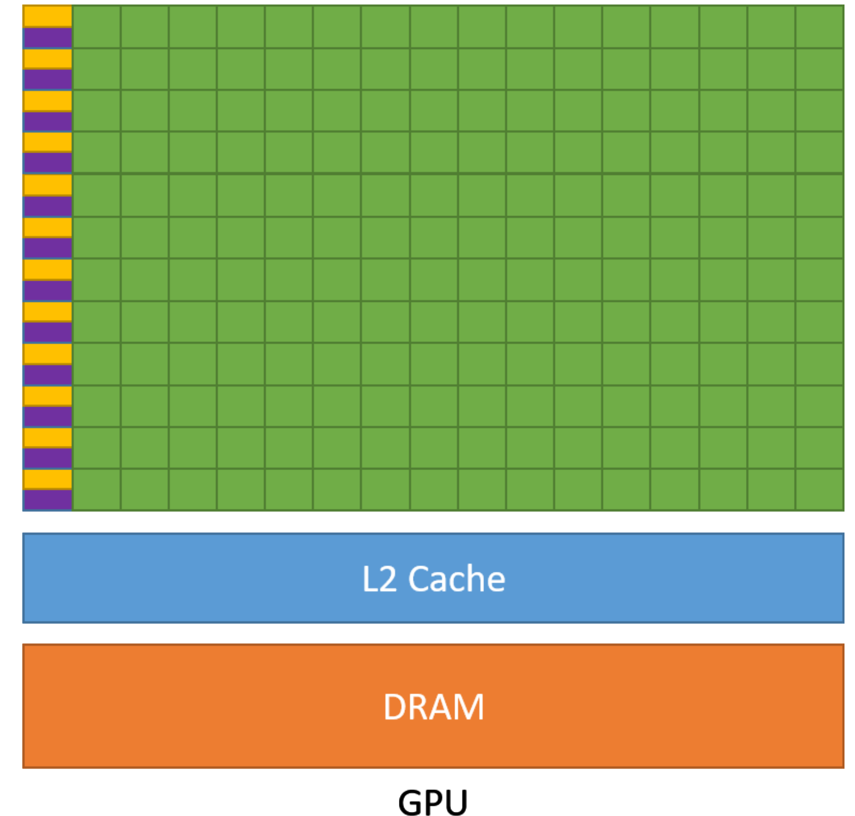


Graphic Processing Unit (GPU)

- Is all about hiding latency
- Optimized for parallel tasks

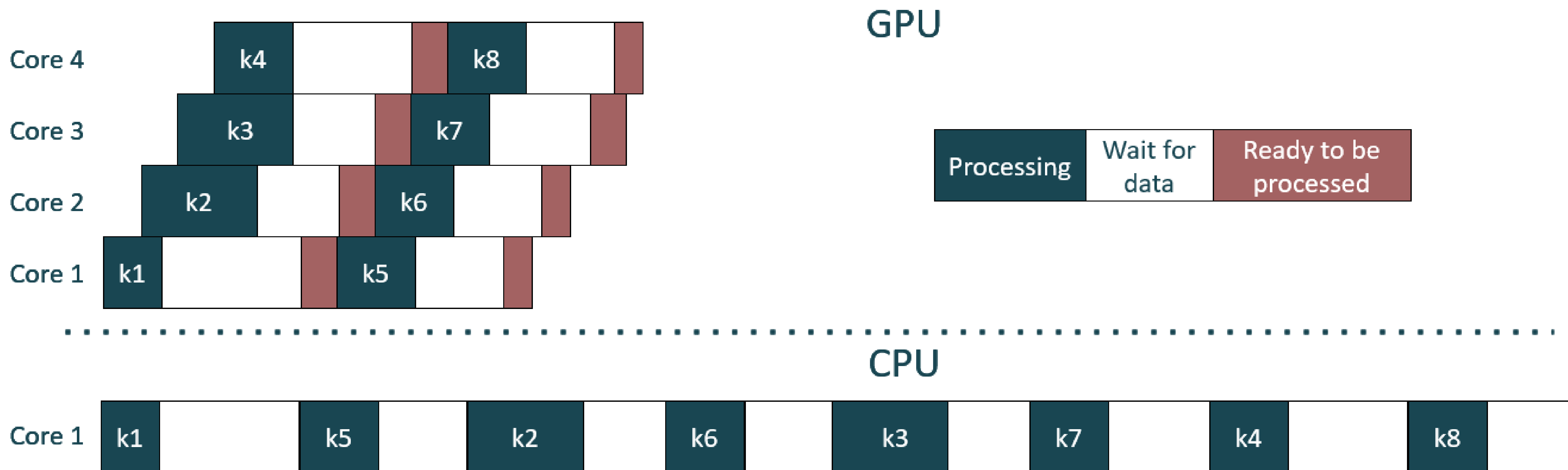
+ Strengths
    - High bandwidth main memory
    - Significantly more compute resources
    - Latency tolerant via parallelism
    - High throughput
    - High performance/watt

- Weaknesses
    - Relatively low memory capacity
    - Low single-thread performance



Low Latency vs High Throughput

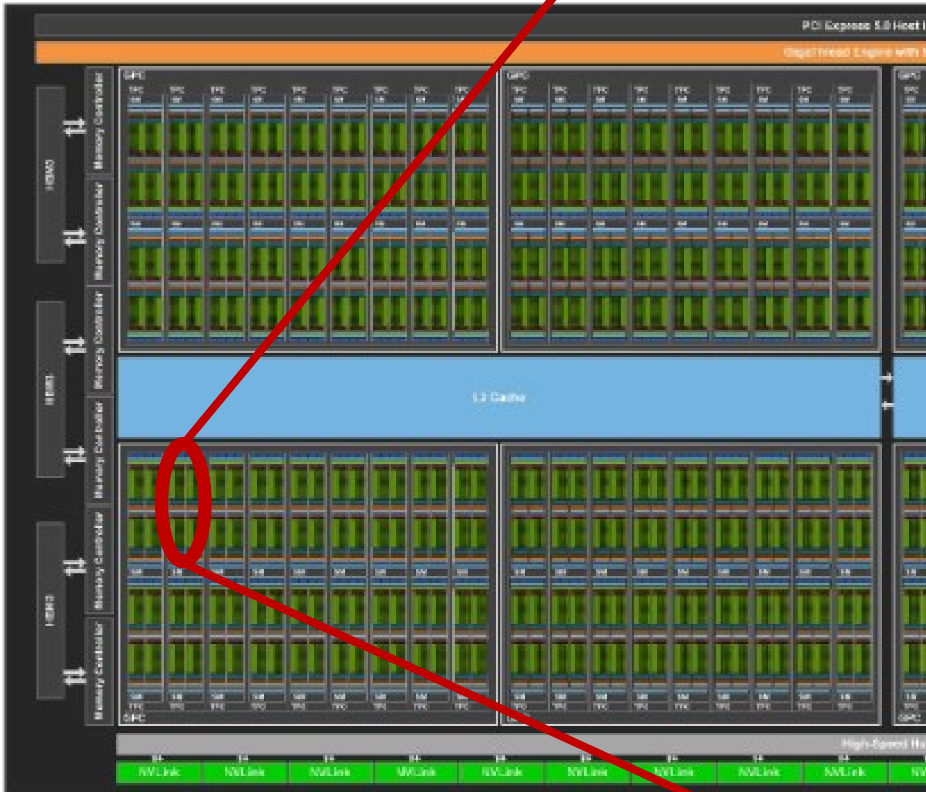
- CPU must minimize latency within each thread
- GPU hides latency with computation



GPU architecture

- NVIDIA H100 PCIe version specs
  - 114 SMs, 14592 FP32 CUDA Cores per GPU, 80 GB HBM2e, NVLin Gen 4, PCIe Gen 5
  - Peak FP64 = 25.6 TFLOPs, Peak FP64 Tensor Core = 51.2TFLOPs

GPU architecture



SM

L1 Instruction Cache

L0 Instruction Cache

Warp Scheduler (32 thread/clk)

Dispatch Unit (32 thread/clk)

Register File (16,384 x 32-bit)

INT32 FP32 FP32 FP64

TENSOR CORE 4th GENERATION

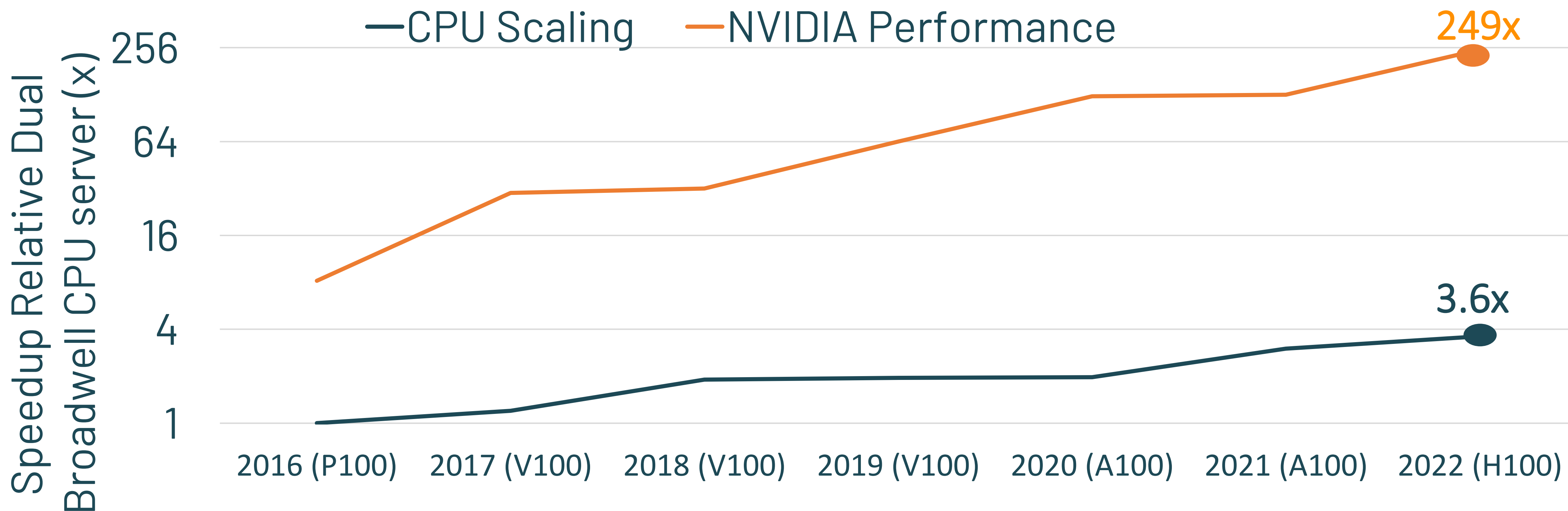
LD/ST SFU

Tensor Memory Accelerator

256 KB L1 Data Cache / Shared Memory

Tex Tex Tex Tex

CPUs vs GPUs



Geometric mean of application speedups relative to Dual Broadwell CPU ESTIMATES ONLY| benchmark applications: Amber [PME-Cellulose_NVE], Chroma [HMC], GROMACS [ADH Dodec], MILC [Apex Medium], NAMD [stmv_nve_cuda], PyTorch (BERT Large Fine Tuner], Quantum Espresso [AUSURF112-jR]; TensorFlow [ResNet-50], VASP 6 [Si Huge]; Random Forest make_blobs (160000 x 64 : 10)
https://resources.nvidia.com/en-gb-eurocc-developer-day/nvidia-eurocc-keyway

GPUs outperform CPUs in certain tasks

✓ Due to massive parallelism

CPU	GPU
Several Cores	Many Cores
Complex/Larger cores	Simpler/smaller cores
Low latency	High throughput
Good for serial processing	Good for parallel processing
Good for almost all operations	Perfect for some operations

Other accelerators

- Intel and AMD GPUs
- FPGA: Field-programmable gate array (Xilinx, Intel Altera)
- ASIC: Application-Specific Integrated Circuit
  - TPU: Tensor Processing Unit (Google)
- Accelerators fit perfectly to accelerate compute-intensive applications as:
  - Financial
  - Face detection
  - Autonomous driving
  - Language Translation
  - Genomics

How to select the optimal accelerator?

Application type	Processing speed	Processing/ Watt	Training	Inference
Speech processing	++	++	GPU, ASIC	CPU, ASIC
Face detection	++	++	GPU, FPGA	CPU, ASIC
Financial risk stratification	++	+	GPU, FPGA	CPU
Route planning	+	+	GPU	CPU
Dynamic pricing	++	+	GPU	CPU, ASIC
Autonomous driving	++	++	ASIC	GPU, ASIC, FPGA

https://www.mckinsey.com/industries/semiconductors/our-insights/artificial-intelligence-hardware-new-opportunities-for-semiconductor-companies#

Well known accelerator programming models

- CUDA → NVIDIA GPUs

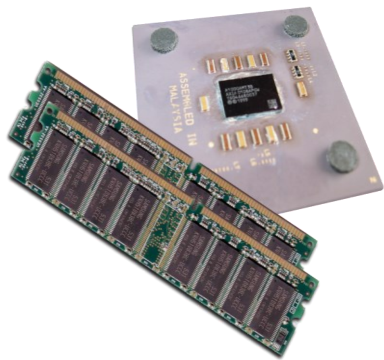
- HIP:
  - ROcM → AMD GPUs
  - CUDA → NVIDIA GPUs

- oneAPI/SYCL → Heterogeneous accelerators

A GPU application consist of host and device code

- Host: The CPU and its memory
  - Instruct the accelerator
- Device: The GPU and its memory
  - i.e. CUDA kernel

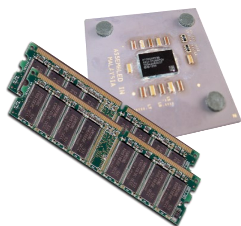
<u>Host</u>



<u>Device</u>

Host and Device code

Host code



Device code

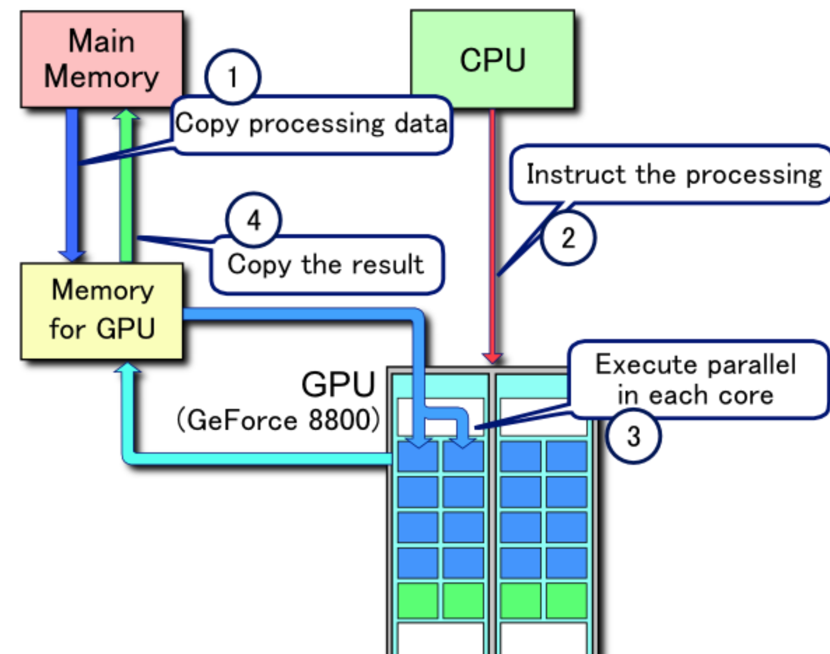
(CUDA kernel)

```
 1 #define THREADS_PER_BLOCK 256
 2 const int N=2048;
 3
 4 int main(void) {
 5     int *a, *b, *c; // host copies of a, b, c
 6     int *d_a, *d_b, *d_c; // device copies of a, b, c
 7     int size = N * sizeof(int);
 8
 9     // Alloc space for device
10     cudaMalloc((void **)&d_a, size);
11     cudaMalloc((void **)&d_b, size);
12     cudaMalloc((void **)&d_c, size);
13
14     // Alloc space on host
15     a = (int*)malloc(size); random_ints(a, N);
16     b = (int*)malloc(size); random_ints(b, N);
17     c = (int*)malloc(size);
18
19     // Copy inputs to device
20     cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
21     cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
22
23     // Launch add() kernel on GPU
24     add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_  b, d_c);
25
26     // Copy result back to host
27     cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
28
29     // Cleanup
30     free(a); free(b); free(c);
31     cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
32     return 0;
33 }
34 //Kernel
35 __global__ void add(int *a, int *b, int *c) {
36     int index = threadIdx.x + blockIdx.x * blockDim.x
37         c[index] = a[index] + b[index];
38 }
```

1 (lines 20–21) · 2 (line 24) · 4 (line 27) · 3 (line 36)

Executed by one **Host Thread**

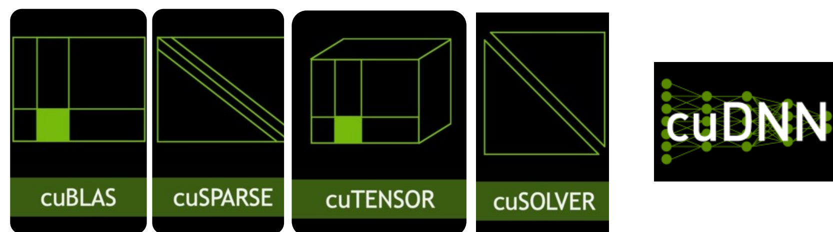
Executed by multiple **CUDA Threads**



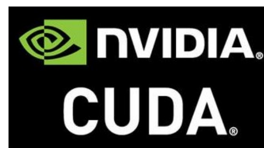
Choose the Abstraction layer that Works for You

Frameworks:

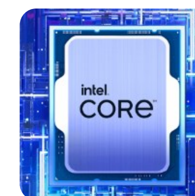
Accelerated libraries:



Parallel language:



Hardware:



Accelerated libraries by example

Frameworks:

Accelerated libraries:

Parallel language:

Hardware:

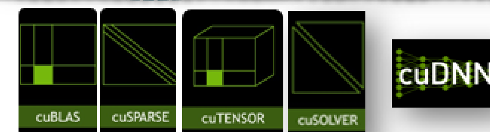
```
int main(
    // allocate host memory for all matrices
    float *h_A = (float *)malloc(mem_size_A);
    float *h_B = (float *)malloc(mem_size_B);
    float *h_C      = (float *) malloc(mem_size_C);

    // initialize host memory for input A and B
   ...

    // allocate device memory
    cudaMalloc((void **) &d_A, mem_size_A);
    cudaMalloc((void **) &d_B, mem_size_B);
    cudaMalloc((void **) &d_C, mem_size_C);

    // transfer data
    cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);

    // setup execution parameters
    dim3 threads(block_size, block_size);
    dim3 grid(matrix_size/threads.x, matrix_size/threads.y);

    // handle to the cuBLAS library context
    cublasHandle_t handle;
    cublasCreate(&handle);

    //Perform the mm
    cublasSgemm(handle, TransposeA, TransposeB, widthB, heightA, widhtA,
        &alpha, d_B, widthB, d_A, widthA, &beta, d_C, widthB));

    // copy result from device to host
    cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);

    // Destroy the handle
    cublasDestroy(handle);

    // clean up host and device memory
    free(...);
    cudaFree(d_A);cudaFree(d_B); cudaFree(d_C);
}
```

Frameworks by example

```
# Check if GPU is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Define two matrices and move them to the GPU
A = torch.tensor([[1, 2],
                  [3, 4]], device=device)

B = torch.tensor([[5, 6],
                  [7, 8]], device=device)

# Perform matrix multiplication
C = torch.matmul(A, B)

# Move the result back to CPU if needed
C_cpu = C.cpu()
```

Frameworks:

Accelerated libraries:



Parallel language:

Hardware:

JETSON XAVIER NX

MLPerf Data-center benchmarks

Intel Cooper Lake | Intel Ice Lake | Xilinx VCK5000 | Qualcom AI 100 | NVIDIA A100

Performance improvement (x)

4.5
4
3.5
3
2.5
2
1.5
1
0.5
0

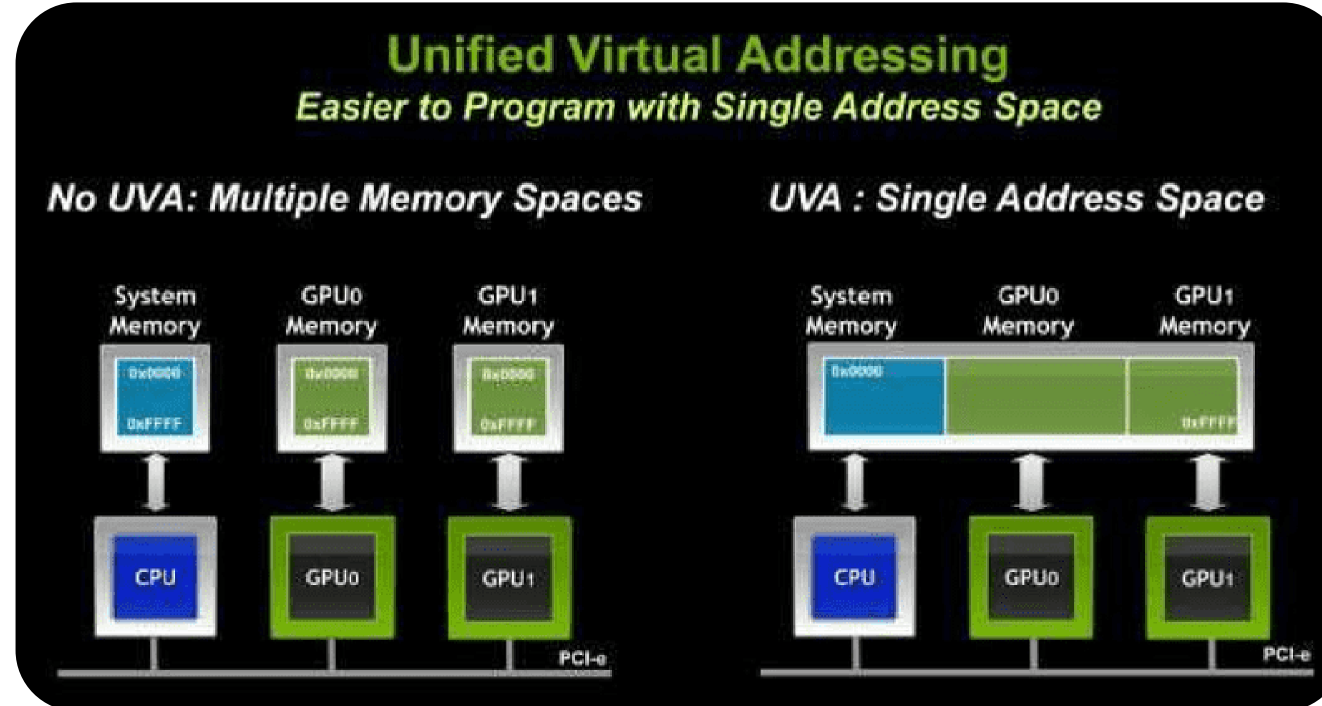
NA
NA NA
314x
NA NA NA
NA NA NA
20x
NA NA NA

Image Classification (ResNet-50)
Object Detection (SSD-Large)
Medical Imaging (3D U-Net)
Recommendation (DLRM)
Speech Recognition(RNN-T)
NLP (BERT)

https://inaccel.com/cpu-gpu-or-fpga-performance-evaluation-of-cloud-computing-platforms-for-machine-learning-training/

Interesting technologies: Unified Memory

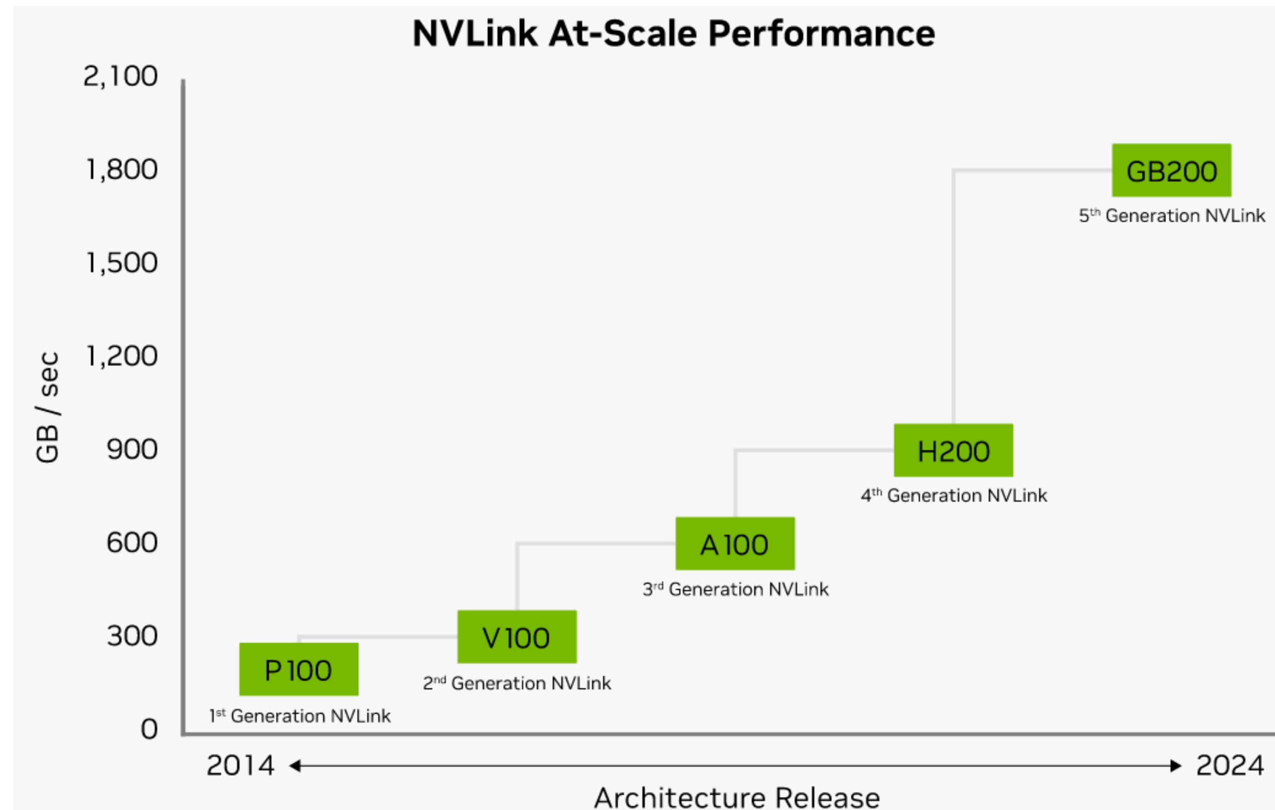
- One common address space between GPUs and CPUs



https://nichijou.co/cudaRandom-UVA/

Interesting technologies: NVLink/NVSwitch

- Faster Scale-Up Interconnects

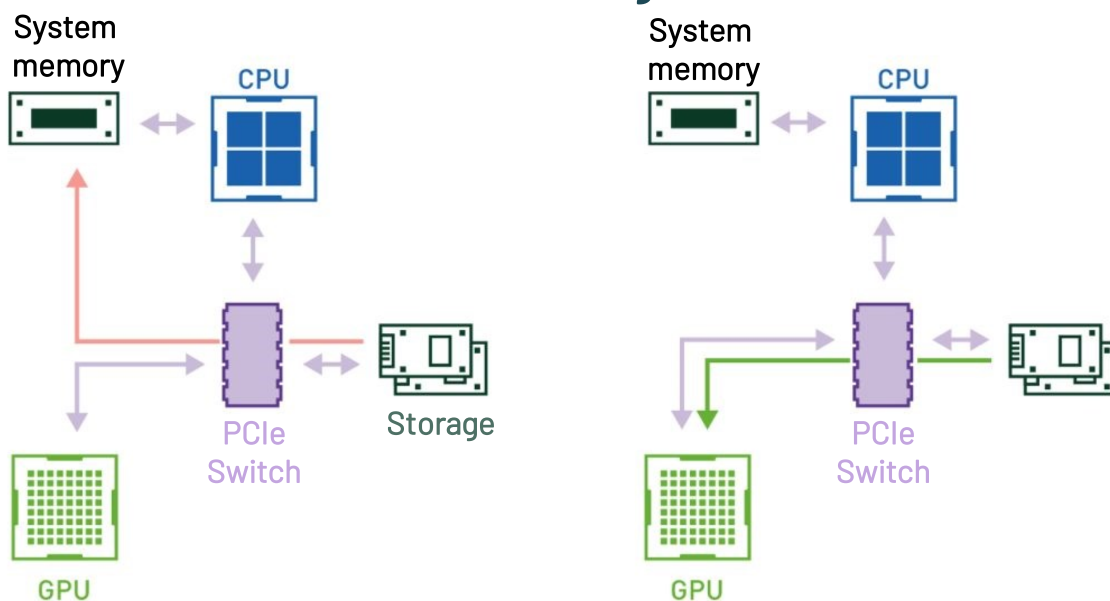


https://www.nvidia.com/en-us/data-center/nvlink/

Interesting GPU technologies: GPUDirect

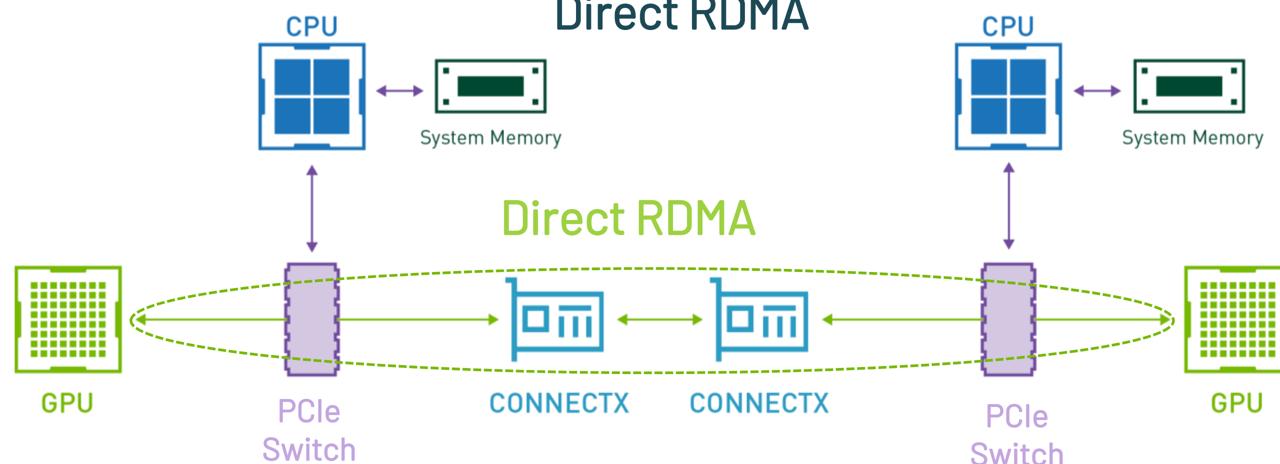
- Provides high-bandwidth and low-latency communications

Direct storage



- Avoid extra copies in host memory
- Eliminates the use of CPU

Direct RDMA

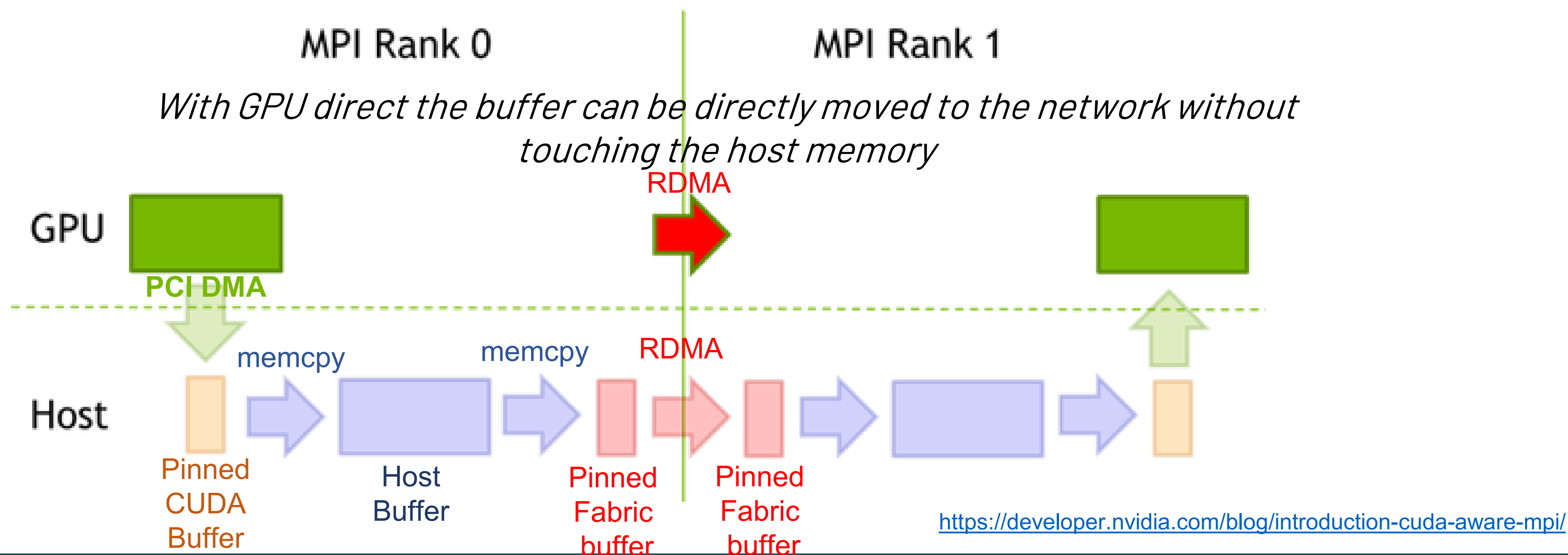


- Provides direct communication between remote GPUs
- Eliminates the use of CPUs and the required buffer copies of data via the system memory
- Results in 10X better performance

https://developer.nvidia.com/gpudirect

Interesting GPU technologies: CUDA aware MPI

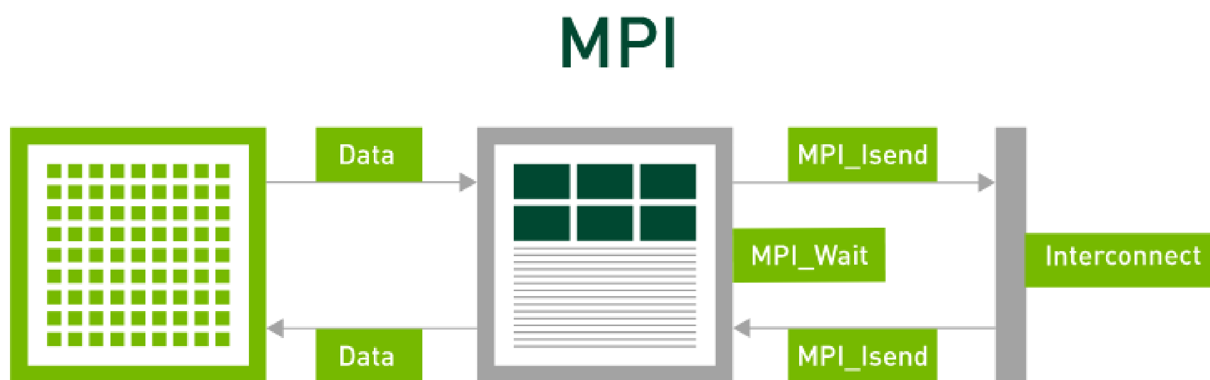
- It uses the traditional MPI properties
- BUT: Takes advantage of NVIDIA technologies
  - Such as GPUDirect and Unified Memory

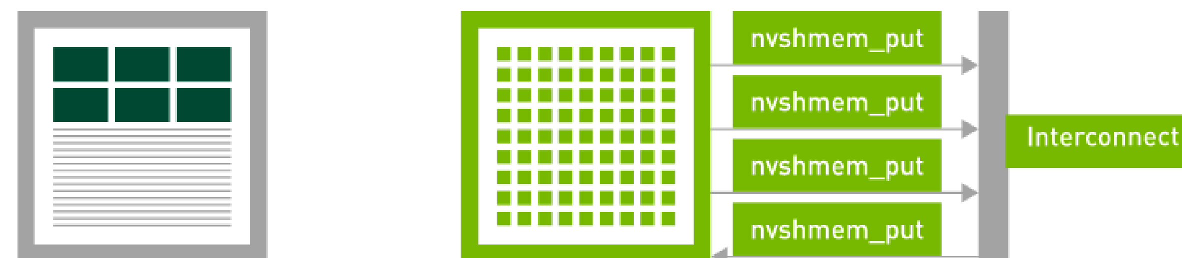


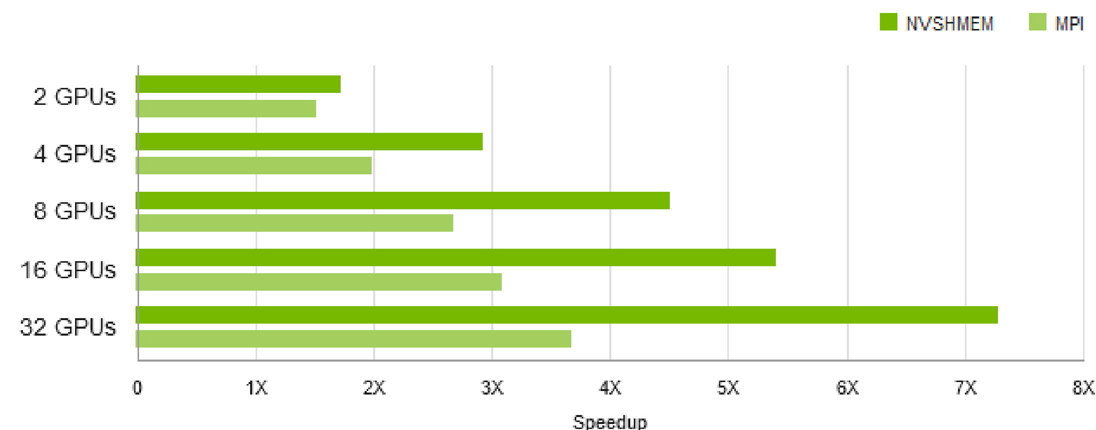
https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/

Interesting GPU technologies: NVSHMEM

- Is a parallel programming interface based on OpenSHMEM
- Creates a global address space for data residing on multiple GPUs
- Can be accessed via
  - Fine-grained GPU-initiated operations
  - CPU-initiated operations
  - Operations on CUDA® streams





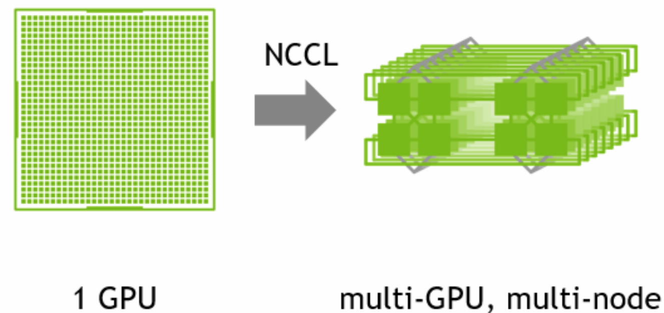


https://developer.nvidia.com/nvshmem

https://developer.nvidia.com/nccl
https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/mpi.html

Interesting GPU technologies: NCCL

- Implements multi-GPU and multi-node communication primitives
  - MPI can be used for CPU-to-CPU communication and NCCL for GPU-to-GPU communication
- Provides routines such as all-gather, all-reduce, broadcast, and p2p send/recv
- All are optimized to achieve high bandwidth and low latency
- Using PCIe for a single node and NVLink high-speed interconnects across nodes



Performance

NCCL conveniently removes the need for developers to optimize their applications for specific machines. NCCL provides fast collectives over multiple GPUs both within and across nodes.

Ease of Programming

NCCL uses a simple C API, which can be easily accessed from a variety of programming languages.NCCL closely follows the popular collectives API defined by MPI (Message Passing Interface).

Compatibility

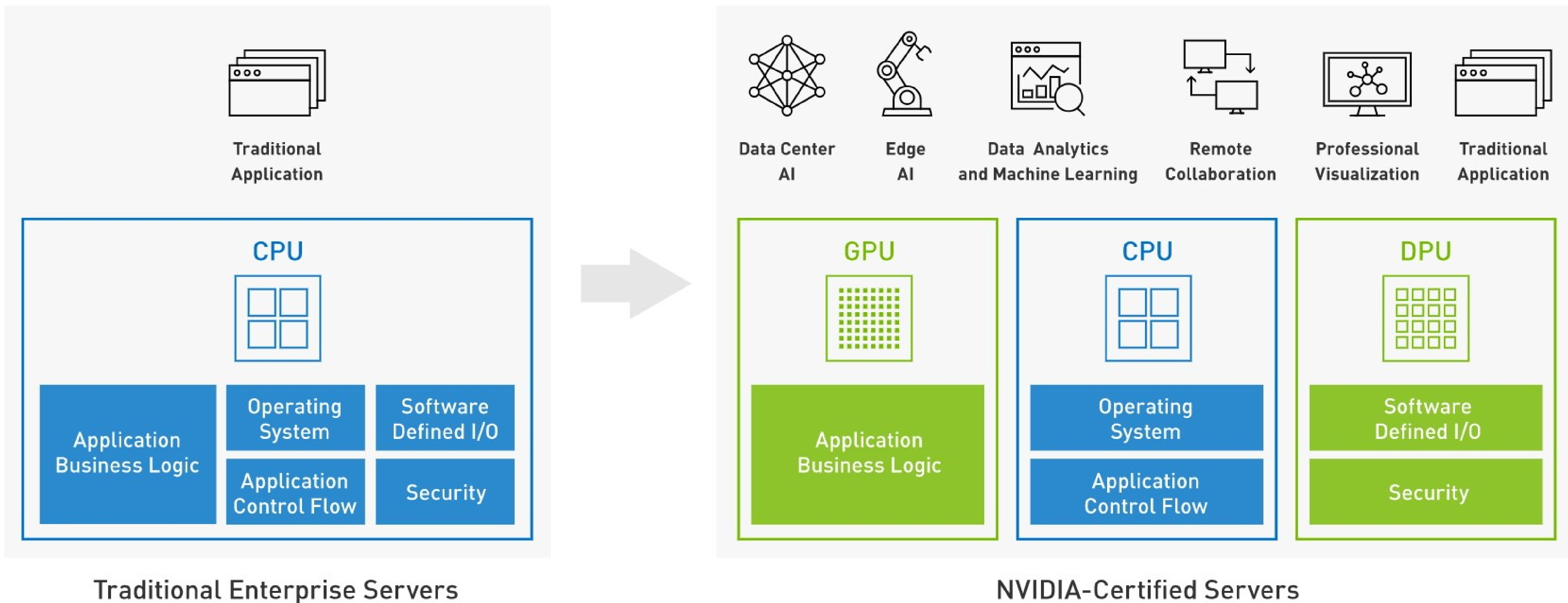
NCCL is compatible with virtually any multi-GPU parallelization model, such as: single-threaded, multi-threaded (using one thread per GPU) and multi-process (MPI combined with multi-threaded operation on GPUs).

https://developer.nvidia.com/nccl
https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/mpi.html

Similar technologies for other accelerators

- Unified Memory
  - oneAPI/SYCL supports USM for Intel accelerators
  - ROCm supports also USM for AMD GPUs
- Fast interconnects
  - Intel has XeLink and AMD Infinity Fabric
- Accelerator aware MPI
  - Intel has Intel MPI for GPU clusters and AMD uses an extension of OpenMPI and Cray MPICH
- SHMEM
  - Intel has Intel SHMEM and and AMD has ROC_SHMEM
- Collective Communication Library
  - Intel has oneCCL and AMD offers RCCL

From CPU only Servers to Accelerated Computing



https://blogs.nvidia.com/blog/what-is-accelerated-computing/

Thank you

Questions?

Manos Pavlidakis
manospavl@ics.forth.gr