Programmazione di sistemi multicore

Michele Martinelli
Michele.martinelli@uniroma1.it

Avvisi

17 applicazioni (HPC e BC) + aiuto progetto

19 applicazioni (droni) + Esercizi Esame

Discussione progetti nella settimana dal 6 al 10 (ipotesi giovedì 9 lab Colossus)

Il secondo esonero sarà insieme al primo appello.

Modalità di esame: 30 domande miste tra risposta chiusa e aperta

date di esame per la prima sessione (esame completo):

- **14 Gennaio ore 16-18, Aula 2L - Via del Castro Laurenziano 7A**
- **4 Febbraio ore 16-18, Aula 2L - Via del Castro Laurenziano 7°**

- **Per chi vuole fare il progetto: chiedete il materiale entro giovedì mattina!**

Ambiente Multi-GPU

Le GPU non condividono la memoria globale

I dati sono scambiato sul PCIe

Ambiente Multi-GPU

La CPU può richiedere informazioni e scegliere una GPU con le seguenti primitive

– cudaGetDeviceCount (int* count)

– cudaSetDevice(int device)

– cudaGetDevice(int *current_device)

– cudaGetDeviceProperties(cudaDeviceProp *prop, int device)

– cudaChooseDevice(int* device,cudaDeviceProp* prop)

Ambiente Multi-GPU

- Scelta esplicita:

Si seleziona il device (cioè la GPU) invocando la funzione cudaSetDevice(devnum)

Deve essere chiamata prima della creazione del contesto (i.e., prima di qualsiasi altra primitiva CUDA)

- Scelta implicita:

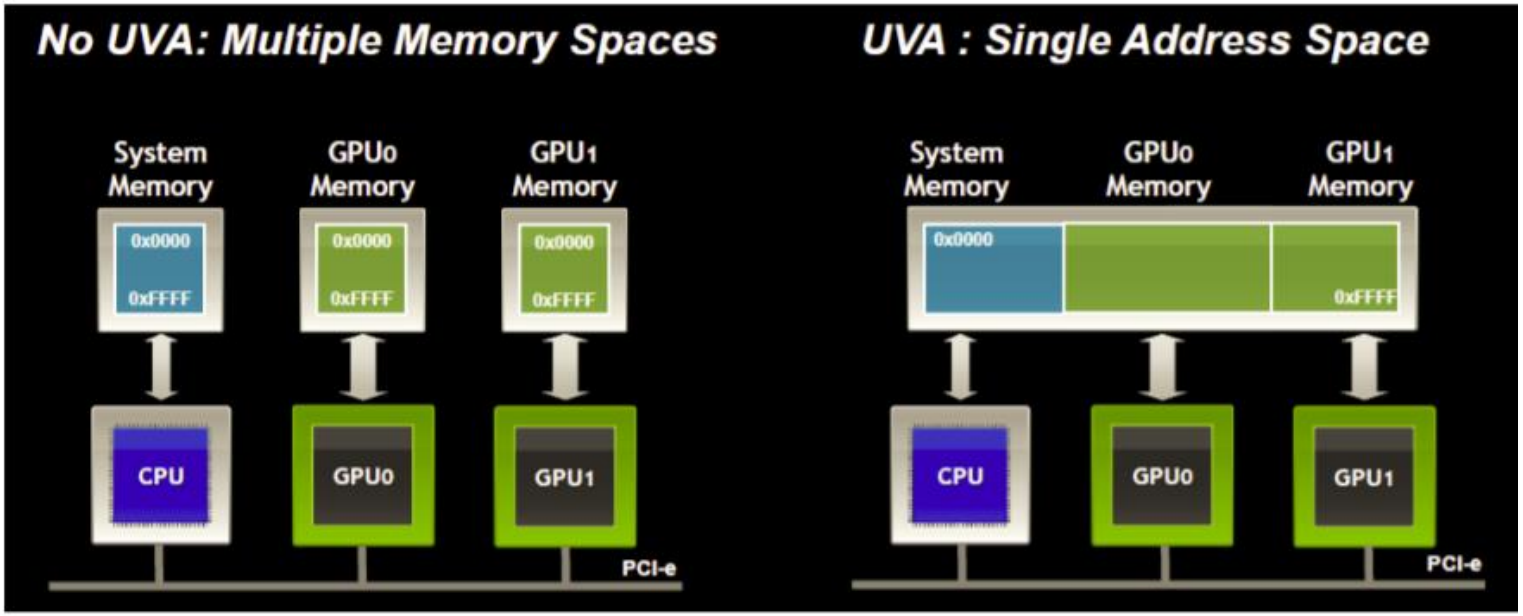
Se non viene chiamata cudaSetDevice, viene scelta la GPU 0

Gestione della GPU: esempio

```
cudaGetDevice (&cudadevice);
cudaGetDeviceProperties(&prop,cudadevice);
mname=prop.name;
muva=prop.unifiedAddressing;
mpc=prop.multiProcessorCount;
mtpb=prop.maxThreadsPerBlock;
shmsize=prop.sharedMemPerBlock;
printf(
“Device %d: number of multiprocessors:%d\n”
“max number of threads per block %d\n”
“shared memory per block %d\n”,cudadevice, mpc, mtbp, shmsize)
```

Indirizzamento virtuale unificato CPU – GPU

- La memoria lato host deve essere pinned
- Allocazione lato host con cudaHostAlloc()
- Allocazione lato device con cudaMalloc()
- Copia con cudaMemcpy()e parametro cudaMemcpyDefault
- E’ possibile accedere alla memoria host direttamente da device tramite cudaHostAlloc() e **cudaHostGetDevicePointer** (CUDA zero copy)



Paradigma RDMA

- È un meccanismo che permette ad un host su una rete di avere accesso diretto ai dati nella memoria di un altro host.
- Server principalmente ad eliminazione la necessità della copia intermedia dei dati tra il buffer di ricezione e l'applicazione finale nell'host ricevente.
- Con RDMA i dati in arrivo vengono riconosciuti come dati per una particolare applicazione e vengono quindi inviati in memoria.

- In questo caso è importante che la memoria sia lockata oltre che pinnata, altrimenti la scheda di rete potrebbe sovrascrivere aree di memoria utilizzate da altri processi

- RDMA è un'evoluzione del DMA!

Memoria pinned e locked

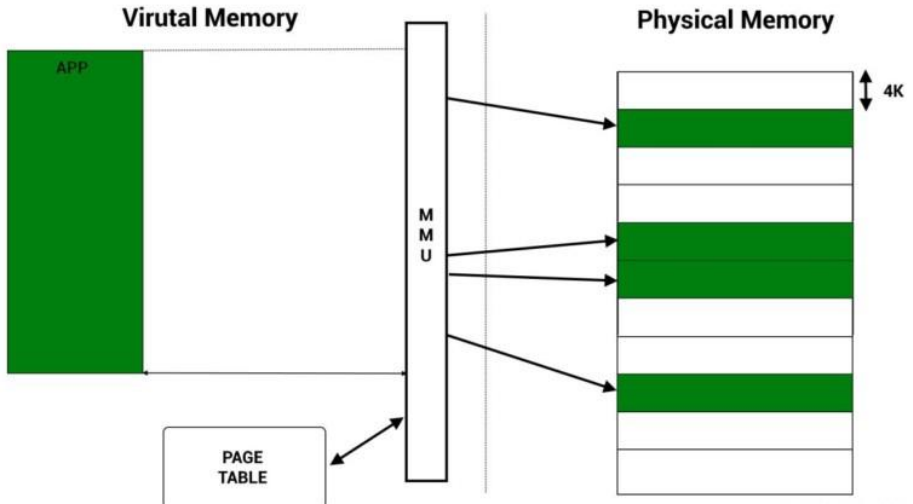
Il sistema normalmente usa un meccanismo di memoria virtuale

Un indirizzo virtuale di memoria corrisponde a uno fisico SOLO IN UN PRECISO MOMENTO

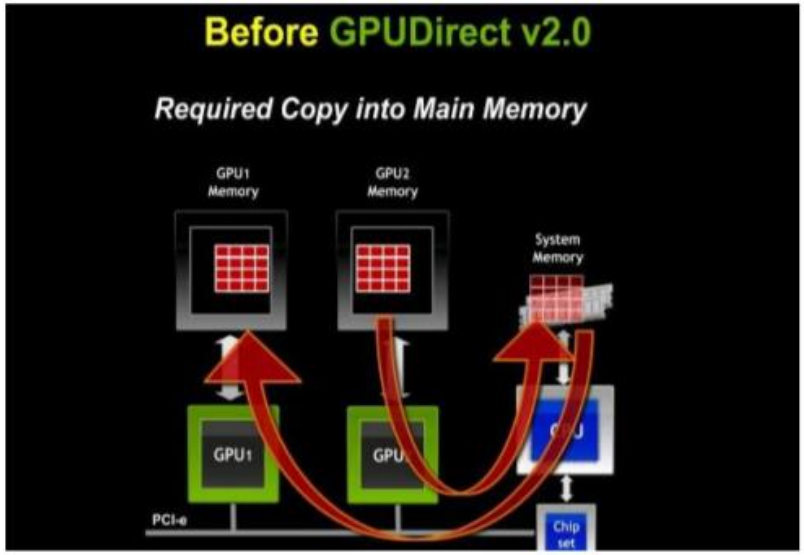
Quando la pagina di memoria non è più utilizzata, questa viene rimpiazzata

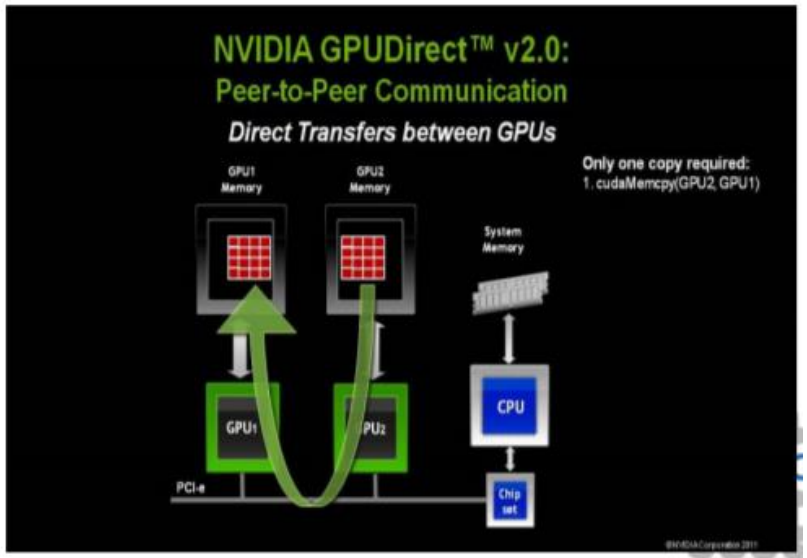
Che succede se a questo punto un device esterno (GPU ecc) va a scrivere in un certo indirizzo fisico? …non corrisponde più alla stessa pagina di memoria virtuale!

«Pinned» una pagina virtuale non può essere swappata (ma l'indirizzo fisico può cambiare)
«locked» rafforza il concetto: forza il sistema a non swappare MAI una pagina di memoria



Comunicazioni Peer-to-Peer



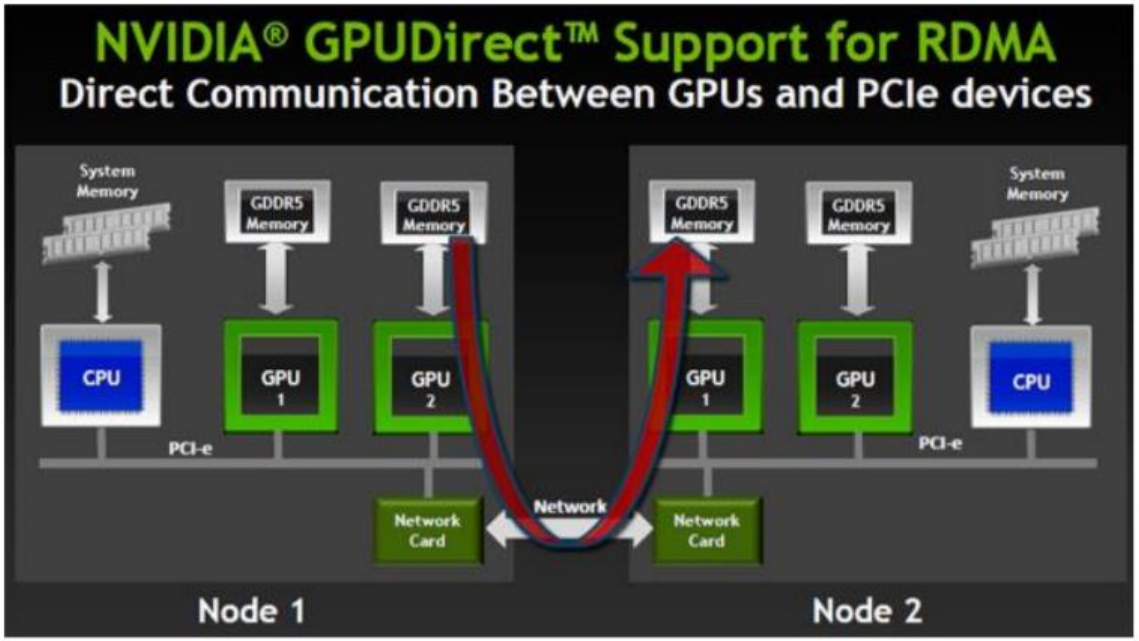


Comunicazioni GPUDirect RDMA

A partire da CUDA 5.0 La tecnologia GPUDirect permette la comunicazione diretta tra GPU e altri dispositivi PCI-E presenti nel sistema;

supporta l'accesso diretto alla memoria tra la GPU e le schede di rete.

La comunicazione tra le GPU avviene tramite chiamate MPI dove al buffer vengono passati gli indirizzi di memoria della GPU.



RDMA + MPI

Codice senza CUDA RDMA ed integrazione con MPI

- Mittente

```
cudaMemcpy(s_buf, s_device, size, cudaMemcpyDeviceToHost);
MPI_Send(s_buf, size, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
```

- Ricevente

```
MPI_Recv(r_buf, size, MPI_CHAR, 0, 1, MPI_COMM_WORLD, &req);
cudaMemcpy(r_device, r_buf, size, cudaMemcpyHostToDevice);
```

Codice con CUDA RDMA ed integrazione con MPI

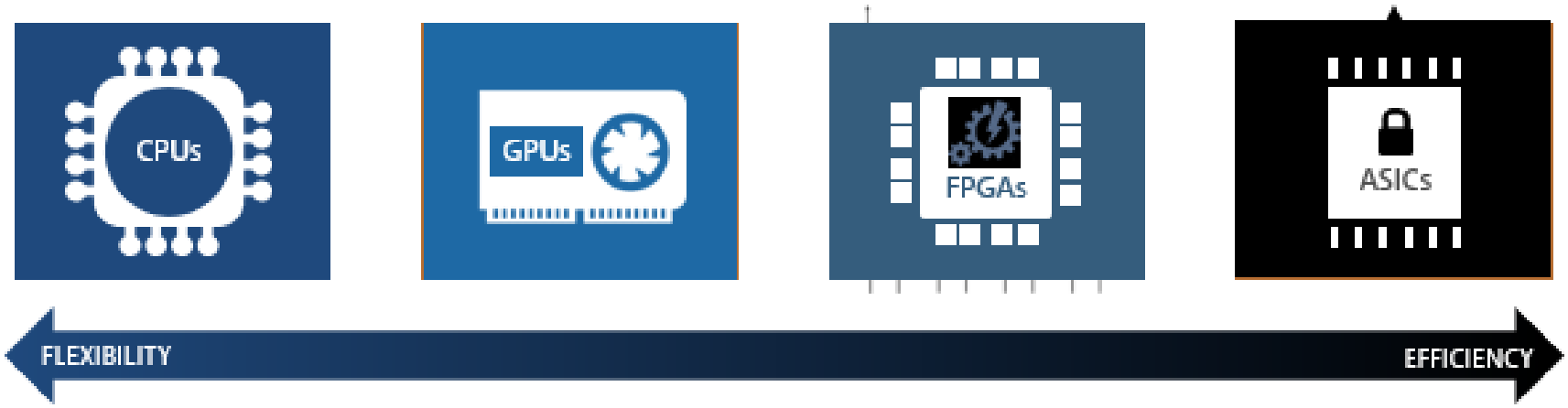
- Mittente

```
MPI_Send(s_device, size, ...);
```

- Ricevente

```
MPI_Recv(r_device, size, ...);
```

Sistemi ibridi: GPU + CPU + ACC

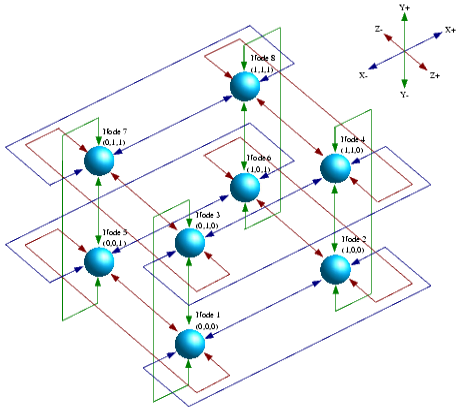


Esempi di applicazione del calcolo multicore

High Performance Computing (HPC)

Scaling to a certain number of computing nodes requires the usage of remotely attached accelerators AND local ones: transit from local to remote memory through the network becomes critical.

- Example: Apenet



Realtime stream processing (HEP)

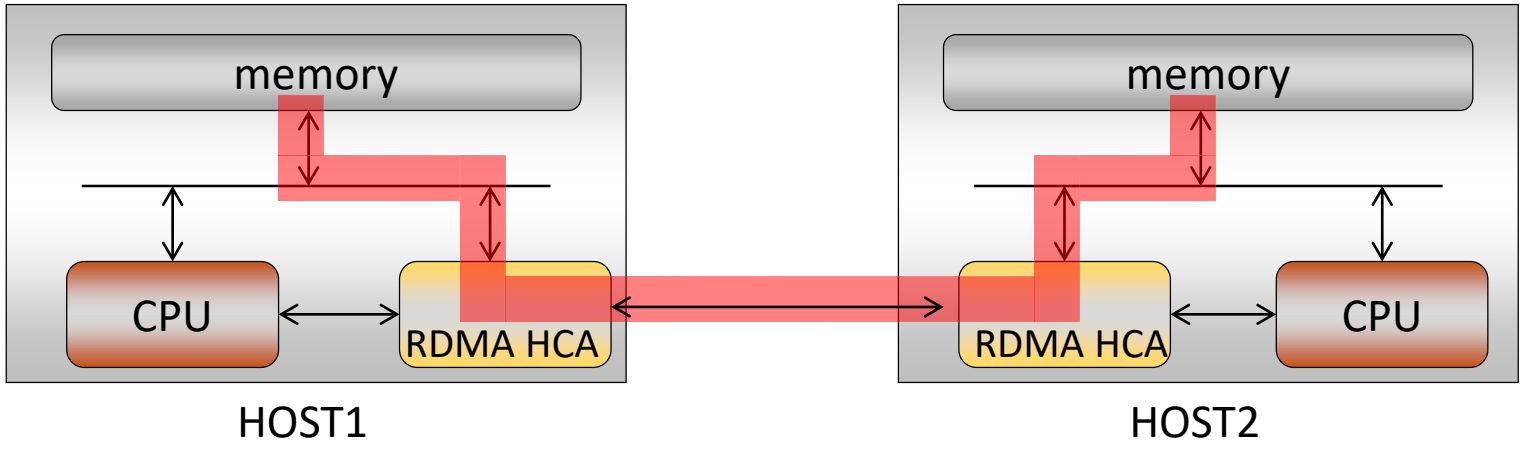
Complex environment in terms of rate, bandwidth and latency, a lot of data comes from the experiment, we need to separate the interesting data from the (huge) background (uninteresting events, noise,…)

- Example: Nanet

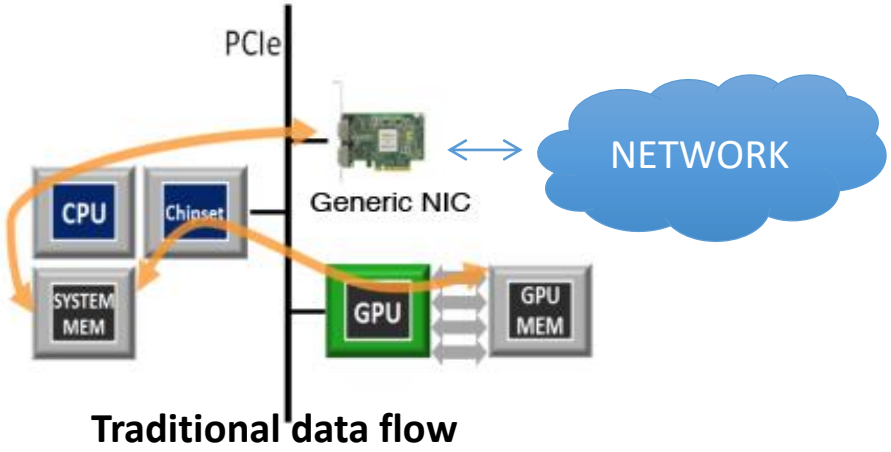
APEnet+ project

APEnet+ is a custom board aimed for low latency / high bandwidth interconnect network

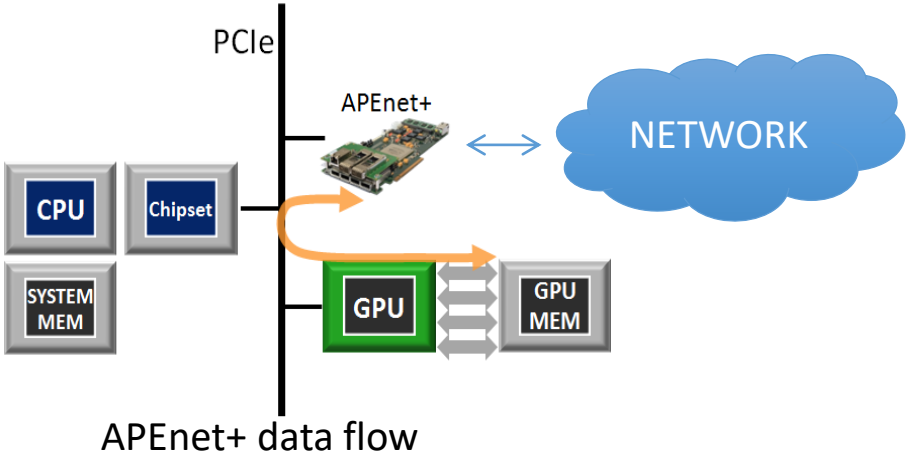
- RDMA communication paradigm (real zero-copy, CPU offloading):



- Nvidia GPUDirect capable:



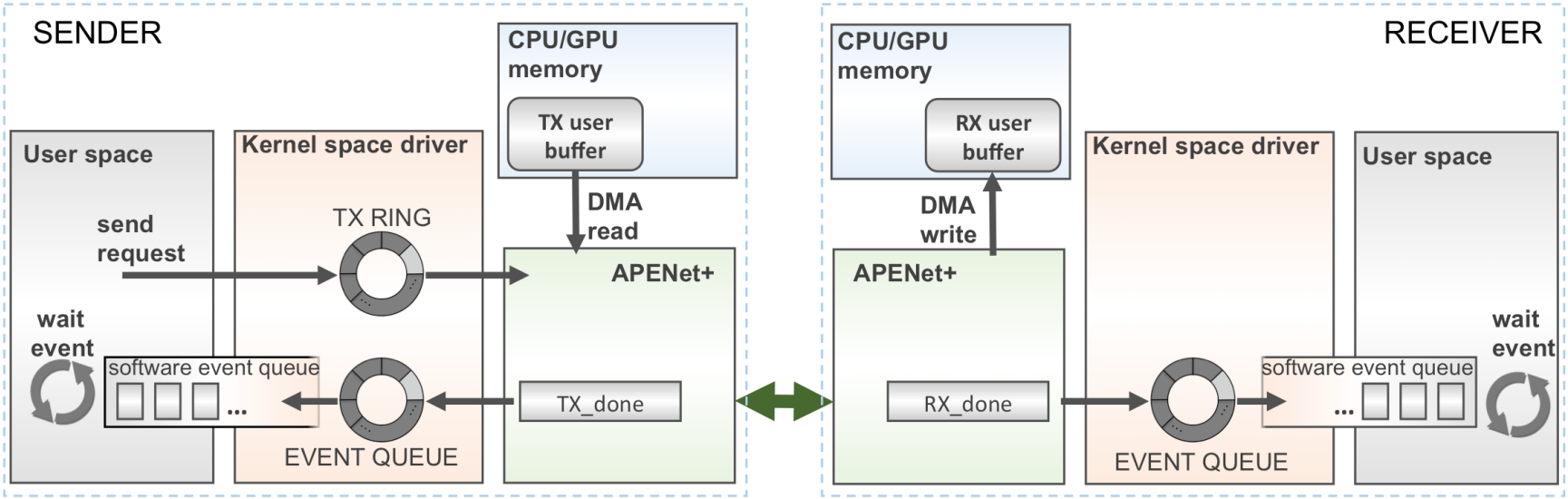
Traditional data flow



APEnet+ data flow

APEnet+ V5 SW Architecture - RDMA

RDMA communication



Buffers allocation (pin and lock memory)

On tx side: buffer with data to transfer

On rx side: buffer where data must be stored

Send request -> descriptor with TX physical address and RX virtual address

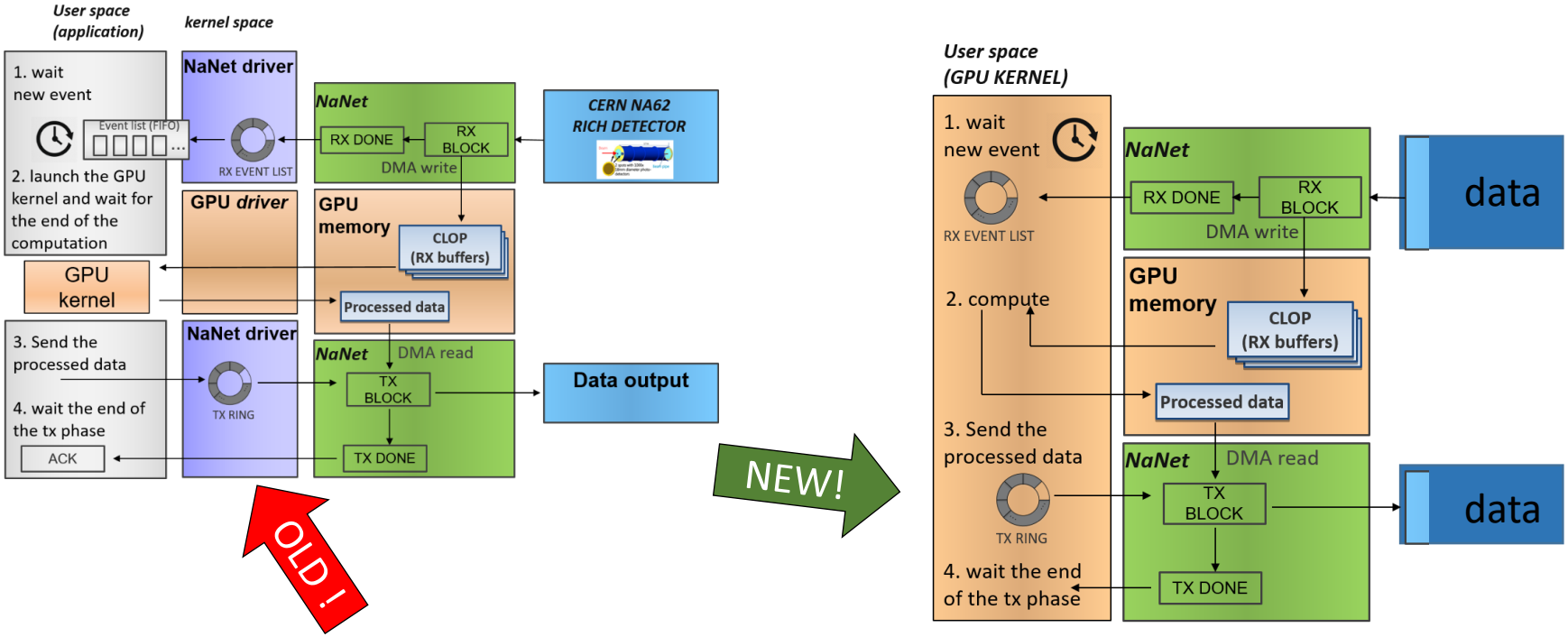
Wait event:

On tx side: wait for the TX_DONE event

On rx side: wait for RX_DONE event

NaNet Design

The idea: a persistent CUDA kernel directly drive the NIC



Eliminate the latency due to the user ⇔ kernel space switch by accessing the board directly from a persistent CUDA kernel

Save the overhead of launching the CUDA kernel every time a new bunch of events arrives since the kernel is already running on the GPU