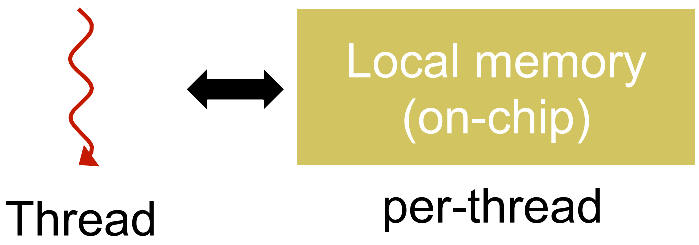
CUDA Memory Model

Hans Henrik Brandenborg Sørensen
DTU Computing Center
DTU Compute
<hhbs@dtu.dk>

Overview

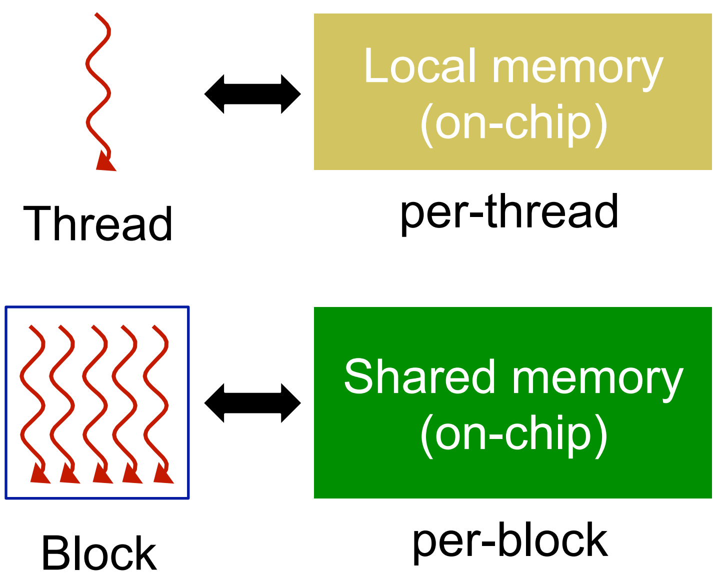
- Memory hierarchy
  - Four basic memory types
- Memory allocation
  - Declarations
  - Dynamic allocation
- Data transfer
  - Host to device
  - Device to host
- Multi-GPU

CUDA memory model



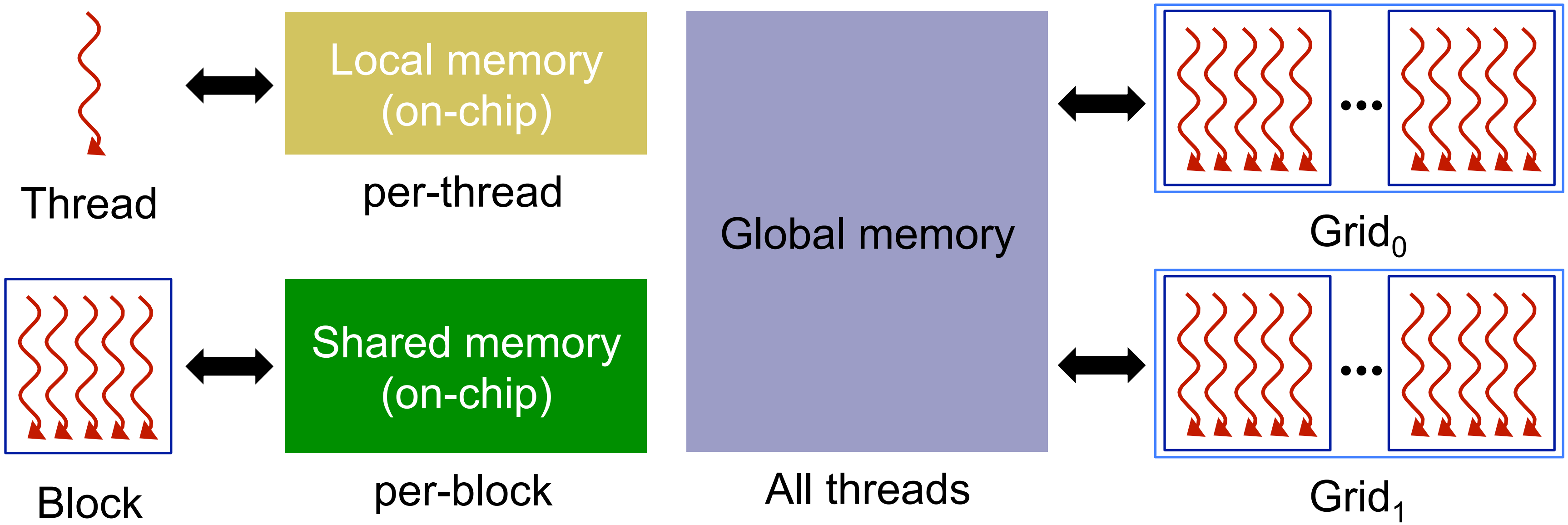
	Local
Size	1KB (regs)
Speed	N/A

CUDA memory model



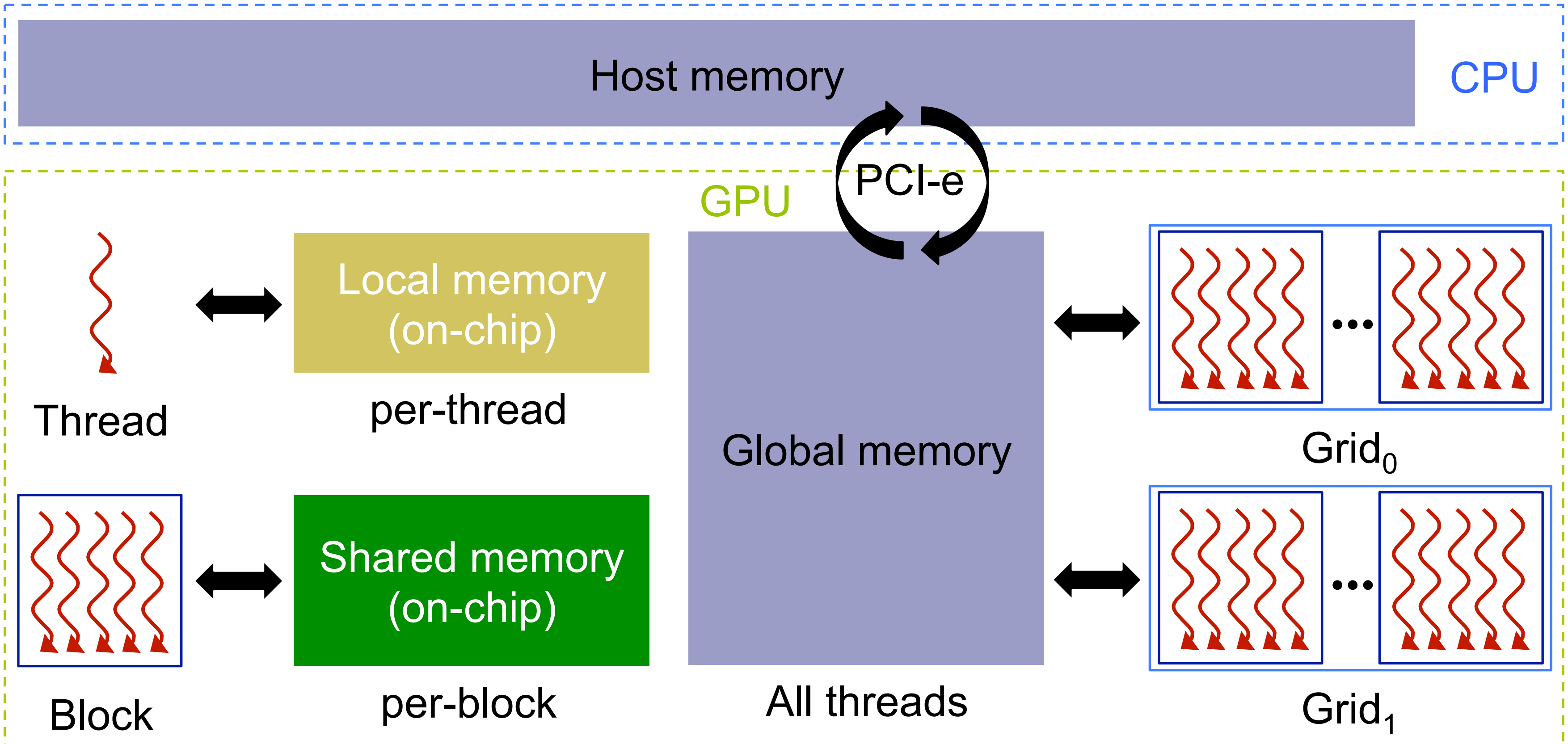
	Local	Shared
Size	1KB (regs)	16KB/48KB
Speed	N/A	1.4 TB/s aggr.

CUDA memory model



	Local	Shared	Global
Size	1KB (regs)	16KB/48KB	12 GB
Speed	N/A	1.4 TB/s aggr.	288 GB/s

CUDA memory model



	Local	Shared	Global	CPU “Host”
Size	1KB (regs)	16KB/48KB	12 GB	128 GB
Speed	N/A	1.4 TB/s aggr.	288 GB/s	< 8-16 GB/s

Local memory example

```
// Using different memory types in CUDA

__global__ void use_local_memory(double val)
{
   // Variable tid is in local memory and private to each thread
   int tid;

   // Built-in variables like threadIdx.x are in local memory
   tid = threadIdx.x + blockIdx.x * blockDim.x;

   // Parameter val is in local memory and private to each thread
   printf("tid=%i val=%lf\n", tid, val);
}
```

Local memory example

```
// Using different memory types in CUDA

__global__ void use_local_memory(double val)
{
   // Variable tid is in local memory and private to each thread
   int tid;

   // Built-in variables like threadIdx.x are in local memory
   tid = threadIdx.x + blockIdx.x * blockDim.x;

   // Parameter val is in local memory and private to each thread
   printf("tid=%i val=%lf\n", tid, val);
}
```

```
#define N 30
int main() {
   // Launch kernel using 6 threads per block
   use_local_memory<<<N/6, 6>>>(2.0);
   cudaDeviceSynchronize();
}
```

Local memory limitations

- Hardware limits

Query	Compute Capability		
	1.x (Tesla)	2.x (Fermi)	3.x (Kepler) 6.x (Pascal)
Max 32-bit registers per thread	128	63	255
Max 32-bit registers per block	8192	32768	65536

Local memory limitations

- Hardware limits

Query	Compute Capability		
	1.x (Tesla)	2.x (Fermi)	3.x (Kepler) 6.x (Pascal)
Max 32-bit registers per thread	128	63	255
Max 32-bit registers per block	8192	32768	65536

- If all registers are used we spill into global memory
  - Variables are quickly cached in L1 and still fast to use

Global memory allocation

- `cudaMalloc()`
  - Dynamically allocate global memory on device
  - Requires two parameters
    - Address of a pointer of type void*
    - Number of bytes to allocate

Global memory allocation

- `cudaMalloc()`
  - Dynamically allocate global memory on device
  - Requires two parameters
    - Address of a pointer of type void*
    - Number of bytes to allocate
- `cudaFree()`
  - Frees global memory

Global memory allocation

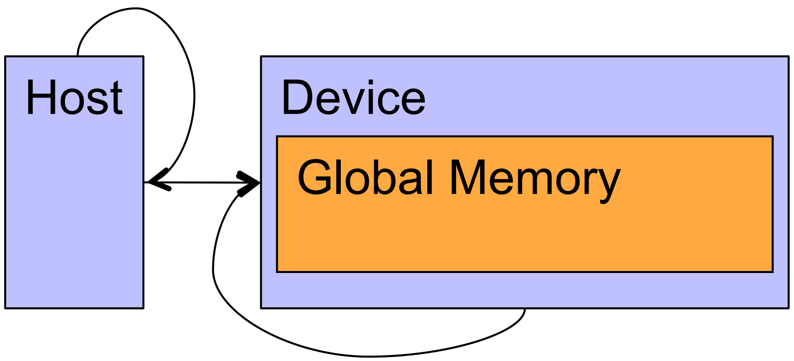
- `cudaMalloc()`
  - Dynamically allocate global memory on device
  - Requires two parameters
    - Address of a pointer of type void*
    - Number of bytes to allocate
- `cudaFree()`
  - Frees global memory

”`d_`” on the variable name is useful to indicate that this points to device memory (not required syntax)

```
// Allocate mem for an array of N doubles
double *d_a;
int size = N * sizeof(double);
cudaMalloc((void**)&d_a, size);
...
cudaFree(d_a);
```

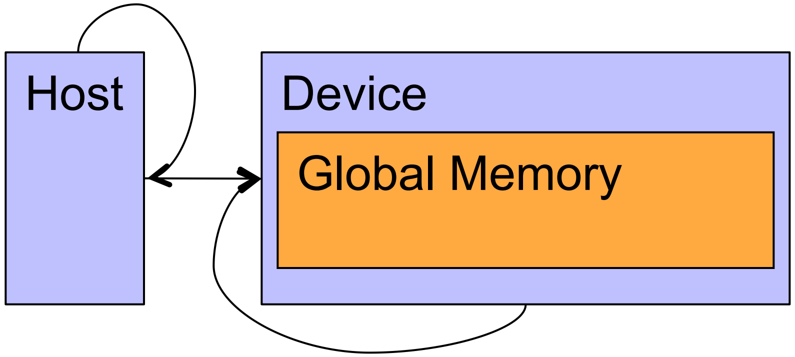
Data transfer

- `cudaMemcpy()`
  - Memory transfer
    - Host to host (completeness)
    - Host to device
    - Device to host
    - Device to device (copy data)



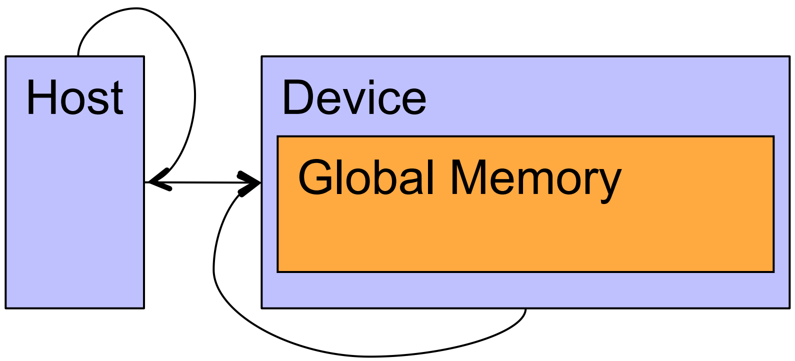
Data transfer

- `cudaMemcpy()`
  - Memory transfer
    - Host to host (completeness)
    - Host to device
    - Device to host
    - Device to device (copy data)
  - Bandwidth limited by PCI-express
    - Typical 5-7 GB/s each way (host ←→ device)



Data transfer

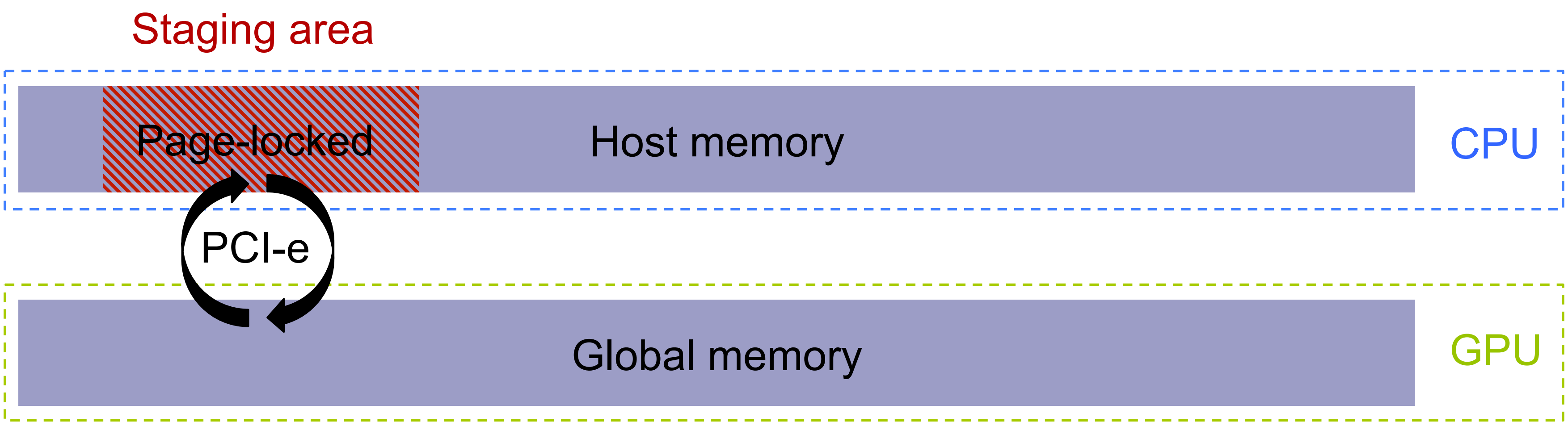
- `cudaMemcpy()`
  - Memory transfer
    - Host to host (completeness)
    - Host to device
    - Device to host
    - Device to device (copy data)
  - Bandwidth limited by PCI-express
    - Typical 5-7 GB/s each way (host ←→ device)



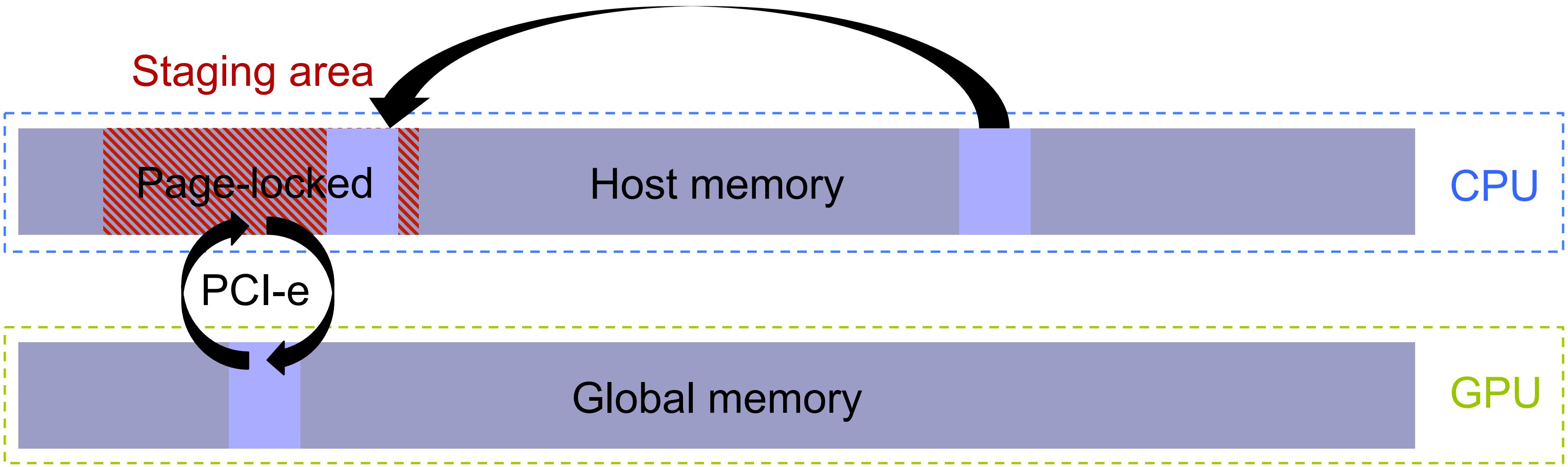

```
// Transfer data from host to device
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);

// Transfer data from device to host
cudaMemcpy(h_a, d_a, size, cudaMemcpyDeviceToHost);
```

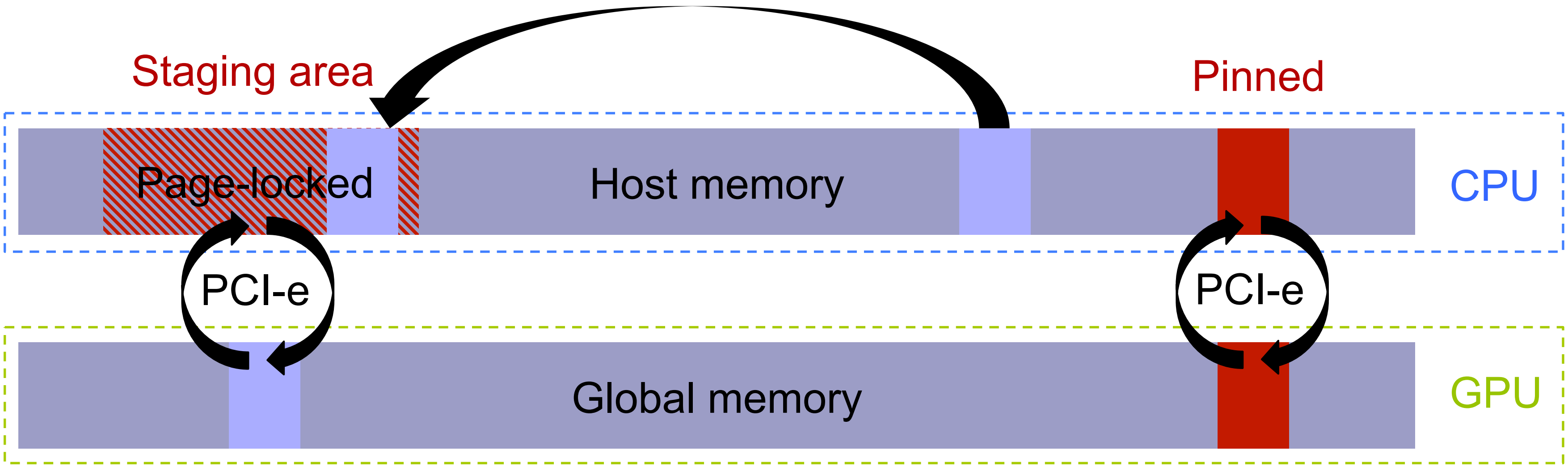
Pinned host memory



Pinned host memory



Pinned host memory



Allocating pinned host memory

- `cudaHostAlloc()`, `cudaMallocHost()`
  - Dynamically allocate page-locked memory on host
- `cudaFreeHost()`
  - Frees page-locked host memory
- `cudaHostRegister()`
  - Page-locks a range of memory allocated by `malloc()`

Global memory example

```
// Using different memory types in CUDA

__global__ void use_global_memory(int *a)
{
   // Variable tid is in local memory and private to each thread
   int tid = threadIdx.x + blockIdx.x * blockDim.x;

   // Parameter a is a pointer into global memory
   a[tid] = tid; // Sets a to [0,1,2,3,..]
}
```

Global memory example

```
#define N 30
int main() {
   // Array pointers on host and device
   int *h_a, *d_a;

   // Alloc mem on host and device
   cudaMallocHost((void **)&h_a, N * sizeof(int));
   cudaMalloc((void **)&d_a, N * sizeof(int));

   // Launch kernel using 6 threads per block
   use_global_memory<<<N/6, 6>>>(d_a);

   // Copy result back to host
   cudaMemcpy(h_a, d_a, N * sizeof(int), cudaMemcpyDeviceToHost);

   // Print result
   print_ints(h_a, N, "a: ");

   // Cleanup
   cudaFreeHost(h_a); cudaFree(d_a);
}
```

Shared memory allocation

- Static allocation using `__shared__`

```
#define N 128

__global__ void kernelFunc(...)
{
   __shared__ double smem[N]; // Static allocation
   ...
}
```

Shared memory allocation

- Static allocation using `__shared__`

```
#define N 128

__global__ void kernelFunc(...)
{
    __shared__ double smem[N]; // Static allocation
    ...
}
```

- Dynamic allocation within `<<<...>>>`

```
kernelFunc<<<dimGrid, dimBlock, N * sizeof(double)>>>(...);

__global__ void kernelFunc(...)
{
    extern __shared__ double smem[]; // Dynamic allocation
    ...
}
```

Shared memory example

```
// Using different memory types in CUDA

__global__ void use_shared_memory(int *a)
{
   // Allocate shared memory statically
   __shared__ int smem[THREADS_PER_BLOCK];

   // Read from global memory to shared memory
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   smem[threadIdx.x] = a[tid];
   __syncthreads(); // Ensure all reads have completed

   // Adds even and odd elements in shared memory
   int i = 1 - 2 * (threadIdx.x & 1);
   smem[threadIdx.x] += smem[threadIdx.x + i];
   __syncthreads(); // Ensure all writes have completed

   // Write back to global memory
   a[tid] = smem[threadIdx.x]; // [0,1,2,3,..] -> [1,1,5,5,..]
}
```

Multi-GPU

Multi-GPU systems

- Multi-GPU systems appear in several flavors

Server

Nvidia Tesla K80

HPC Cluster (via MPI)

  - Nvidia Tesla K80, while physically occupying a single expansion slot, will appear to your CUDA applications as two separate GPUs

Multi-GPU systems

- Using multiple GPUs within the same application can improve the performance
  - Splitting the task (extra level of parallelism)
  - Scales the peak performance
  - Scales the memory bandwidth
  - Does NOT scale the PCI-e bandwidth!

Multi-GPU systems

- Using multiple GPUs within the same application can improve the performance
  - Splitting the task (extra level of parallelism)
  - Scales the peak performance
  - Scales the memory bandwidth
  - Does NOT scale the PCI-e bandwidth!
- The CUDA runtime requires that each GPU must be associated to a separate thread running on the CPU (started automatically)

Multi-GPU with CUDA

- `cudaGetDeviceCount()` gets the number of available GPUs
- `cudaSetDevice()` sets the device to run on
- `cudaGetDevice()` gets the current device

```
// Run independent kernel on each CUDA device
int numDevs = 0;
cudaGetDeviceCount(&numDevs);
...
for (int d = 0; d < numDevs; d++) {
   cudaSetDevice(d);
   kernel<<<dimGrid, dimBlock>>>(args);
}
...
```

Memory allocation / transfers

- You can handle memory on multiple GPUs by applying `cudaSetDevice()` multiple times

```
// Allocate half a matrix on two GPUs, copy top and bottom part to
each GPU and run independent kernels
...
cudaSetDevice(0);
double *d0_A;
cudaMalloc((void**)&d0_A, A_size/2);
cudaMemcpy(d0_A, h_A, A_size/2, cudaMemcpyHostToDevice);
kernel<<<dimGrid, dimBlock>>>(d0_A);

cudaSetDevice(1);
double *d1_A;
cudaMalloc((void**)&d1_A, A_size/2);
cudaMemcpy(d1_A, h_A + A_size/2, A_size/2, cudaMemcpyHostToDevice);
kernel<<<dimGrid, dimBlock>>>(d1_A);
...
```

Peer-to-peer memory access

- Use `cudaDeviceEnablePeerAccess()` to get unidirectional peer access to other GPUs

```
// Enable peer-to-peer access and run kernels
cudaSetDevice(0);
cudaDeviceEnablePeerAccess(1, 0); // (dev 1, future flag)
kernel<<<dimGrid, dimBlock>>>(d0_A, d1_A);

cudaSetDevice(1);
cudaDeviceEnablePeerAccess(0, 0); // (dev 0, future flag)
kernel<<<dimGrid, dimBlock>>>(d0_A, d1_A);
```

- Check peer access support with `deviceQuery`

Unified virtual memory

- Use `cudaMemcpyDefault`
  - cudaMemcpy() knows that our arrays are on different devices if you use `cudaMallocManaged()`.
  - It will do a P2P copy if possible
  - This will transparently fall back to a normal copy through the host if P2P is not available

```
// Transfer between host and multiple GPUs
cudaMemcpyDefault(d0_A, A, A_size, cudaMemcpyDefault);
cudaMemcpyDefault(d1_A, A, A_size, cudaMemcpyDefault);

// Transfer between two GPUs
cudaMemcpyDefault(d0_A, d1_A, A_size, cudaMemcpyDefault);
```

- Wrap up exercise 2
- Do exercise 3 (Mandelbrot set) now!
  - Use your code from last week as starting point
  - Please note that you are not to run CPU benchmarks on the GPU nodes (use batch job script as last week)

- Next presentation “CUDA Performance Tuning Introduction” at 9.00 (Tuesday)!

End of lecture