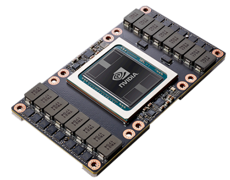
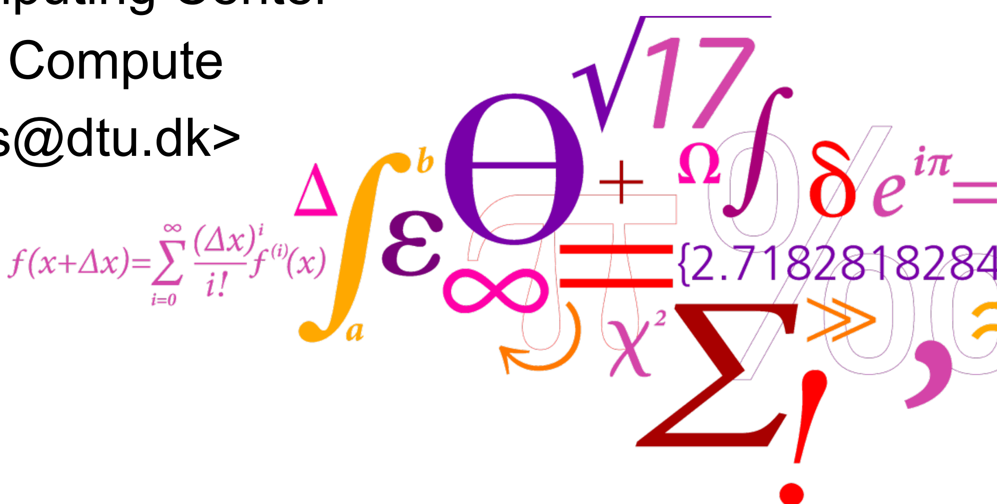
# CUDA Memory Model





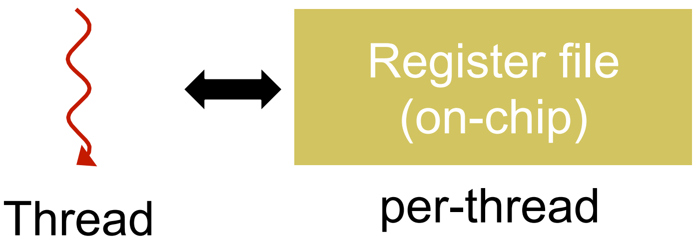
Hans Henrik Brandenborg Sørensen
DTU Computing Center
DTU Compute
<hhbs@dtu.dk>



# Overview

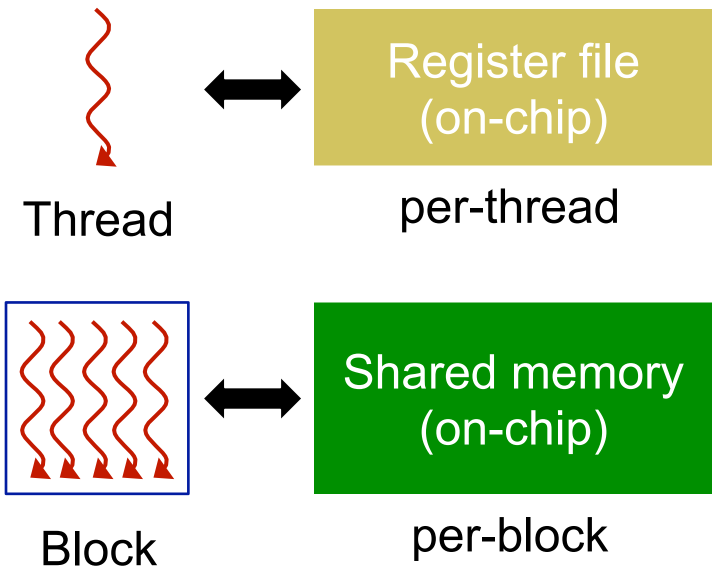
- Memory hierarchy
  - Four basic memory types
- Memory allocation
  - Declarations
  - Dynamic allocation
- Data transfer
  - Host to device
  - Device to host
- Unified memory
- Multi-GPU

# CUDA memory model



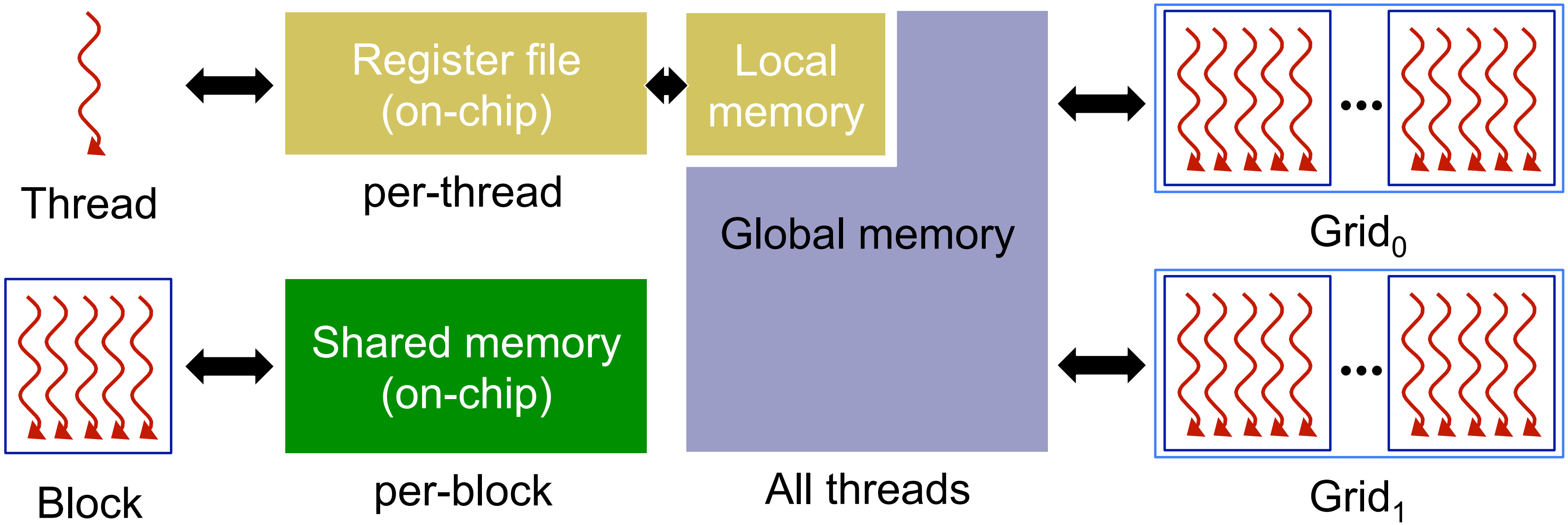

| | Register file |
|---|---|
| Size | 256KB / SM |
| Speed | N/A |

# CUDA memory model



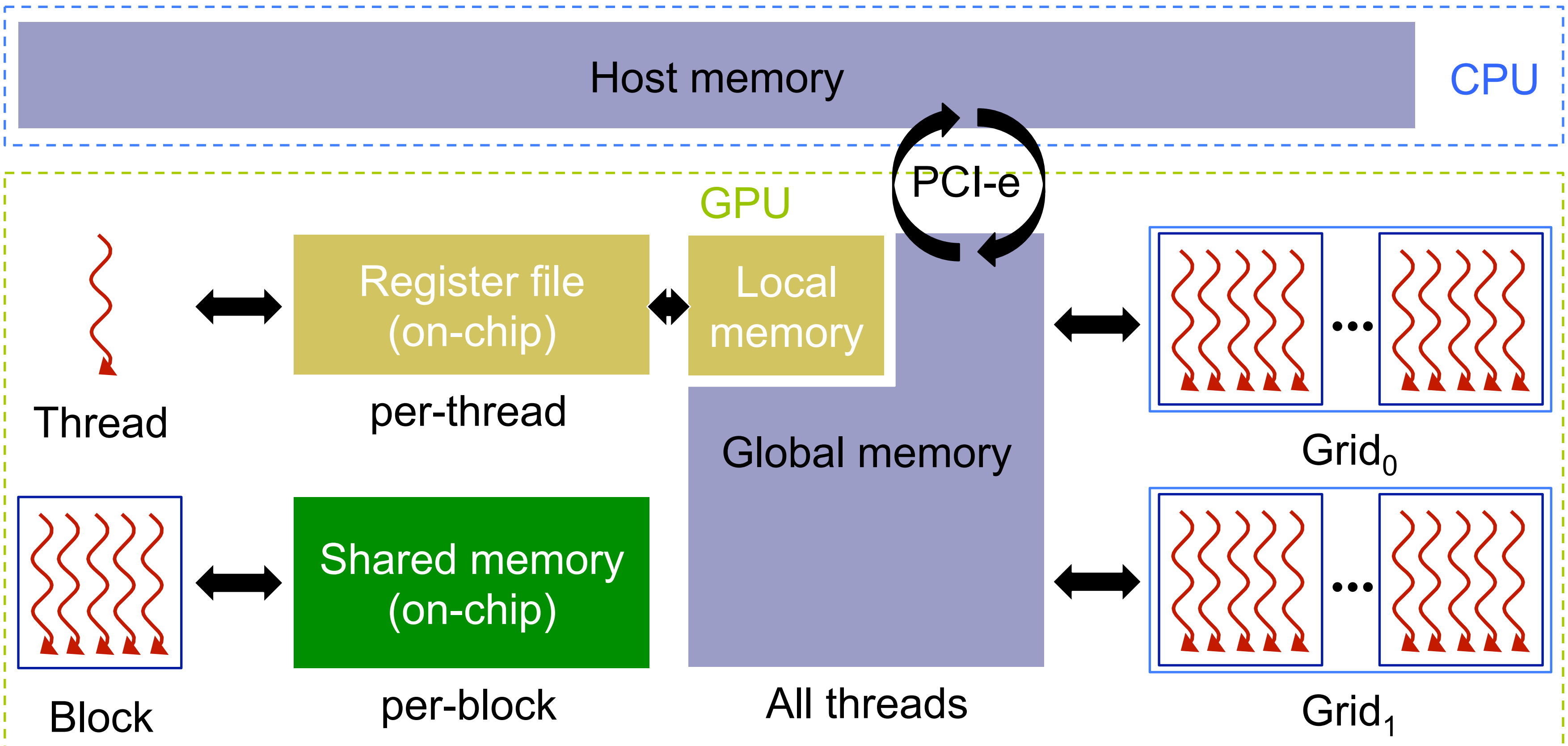

| | Register file | Shared |
|---|---|---|
| Size | 256KB / SM | Up to 164KB |
| Speed | N/A | >10TB/s aggr. |

# CUDA memory model




| | Register file | Shared | Global / local |
|---|---|---|---|
| Size | 256KB / SM | Up to 164KB | 16 – 80 GB |
| Speed | N/A | >10TB/s aggr. | 1555 GB/s |

# CUDA memory model




| | Register file | Shared | Global / local | CPU "Host" |
|---|---|---|---|---|
| Size | 256KB / SM | Up to 164KB | 16 – 80 GB | 192 – 768 GB |
| Speed | N/A | >10TB/s aggr. | 1555 GB/s | < 8 –16 GB/s |

# Register memory example

```
// Using different memory types in CUDA

__global__ void use_register_memory(double val)
{
   // Variable tid is in a register and private to each thread
   int tid;

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

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

# Register memory example

```
// Using different memory types in CUDA

__global__ void use_register_memory(double val)
{
   // Variable tid is in a register and private to each thread
   int tid;

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

   // Parameter val is in a register 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_register_memory<<<N/6, 6>>>(2.0);
   cudaDeviceSynchronize();
}
```

# Register memory limitations

- Hardware limits

| Query | Compute Capability | | |
|---|---|---|---|
| | 1.x (Tesla) | 2.x (Fermi) | 3.x (Kepler) – 8.x (Ampere) |
| Max 32-bit registers per thread | 128 | 63 | 255 |
| Max 32-bit registers per block | 8192 | 32768 | 65536 |

# Register memory limitations

- Hardware limits

| Query | Compute Capability | | |
|---|---|---|---|
| | 1.x (Tesla) | 2.x (Fermi) | 3.x (Kepler) – 8.x (Ampere) |
| 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 local memory
  - Variables are quickly cached in L1 and still fast to use

# Local memory example

```
// Using different memory types in CUDA

__global__ void use_local_memory()
{
   // Small statically allocated arrays are typically in registers
   double sum[8];

   // Large statically allocated arrays spill to local memory
   double array[256];

   // Both are private to each thread
   ...
}
```

# 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

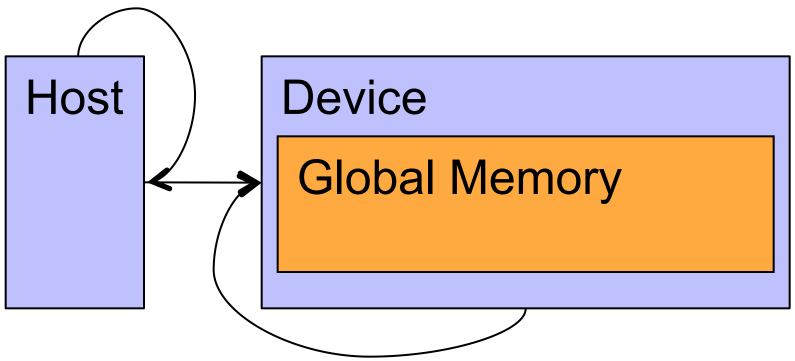
- `cudaMalloc()`
  - Dynamically allocate global memory on device
  - Requires two parameters
    - Address of a pointer of type void*
    - Number of bytes to allocate
- `cudaFree()`
  - Frees global memory

”`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);
```

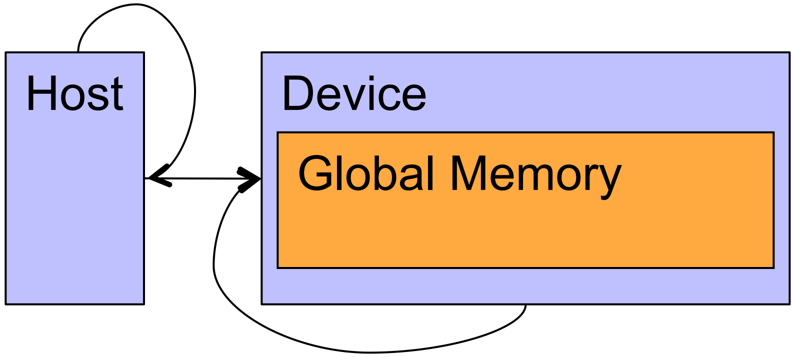
# Data transfer

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



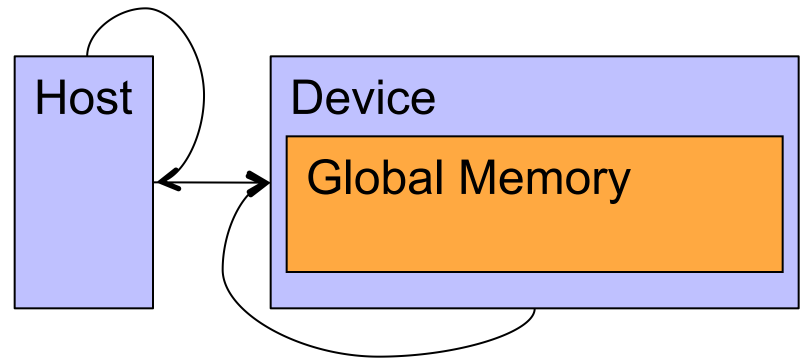
# Data transfer

- `cudaMemcpy()`
  - Memory transfer
    - Host to host (completeness)
    - Host to device
    - Device to host
    - Device to device (copy data)
  - Bandwidth limited by PCI-express
    - Typical ~8-14 GB/s each way (host ⇐⇒ device)



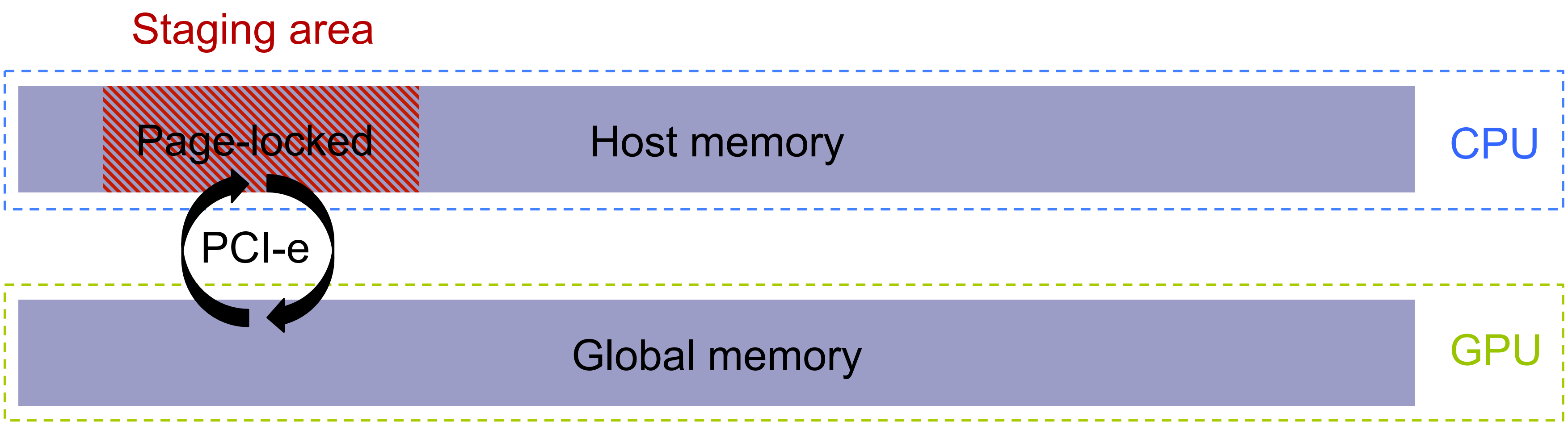
# Data transfer

- `cudaMemcpy()`
  - Memory transfer
    - Host to host (completeness)
    - Host to device
    - Device to host
    - Device to device (copy data)
  - Bandwidth limited by PCI-express
    - Typical ~8-14 GB/s each way (host ←→ device)

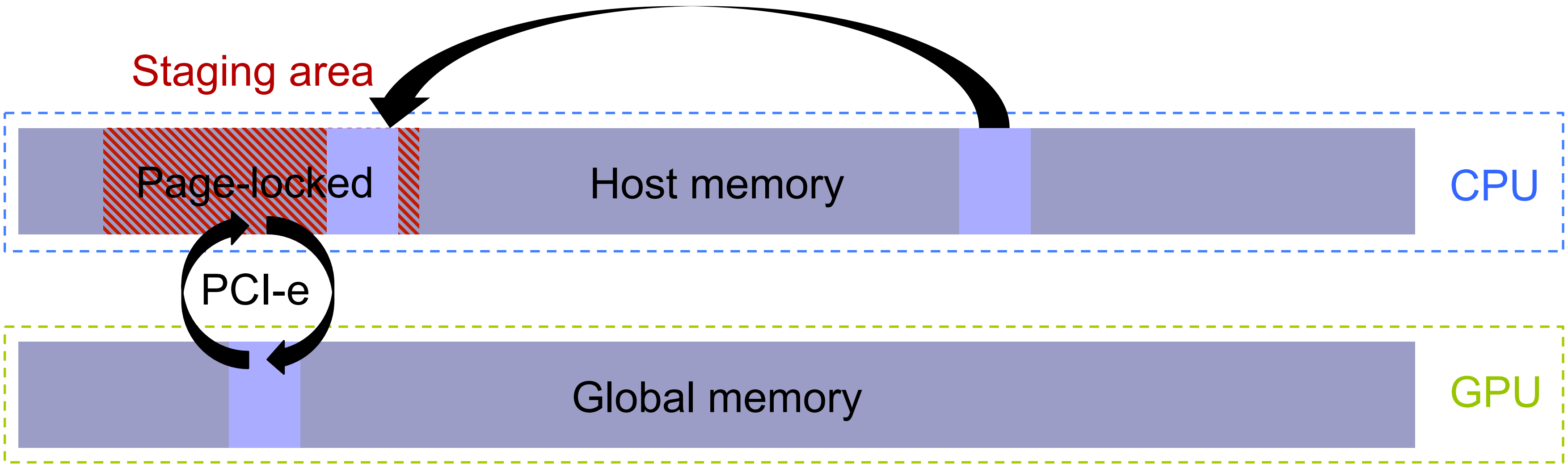



```
// Transfer data from host to device
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);

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

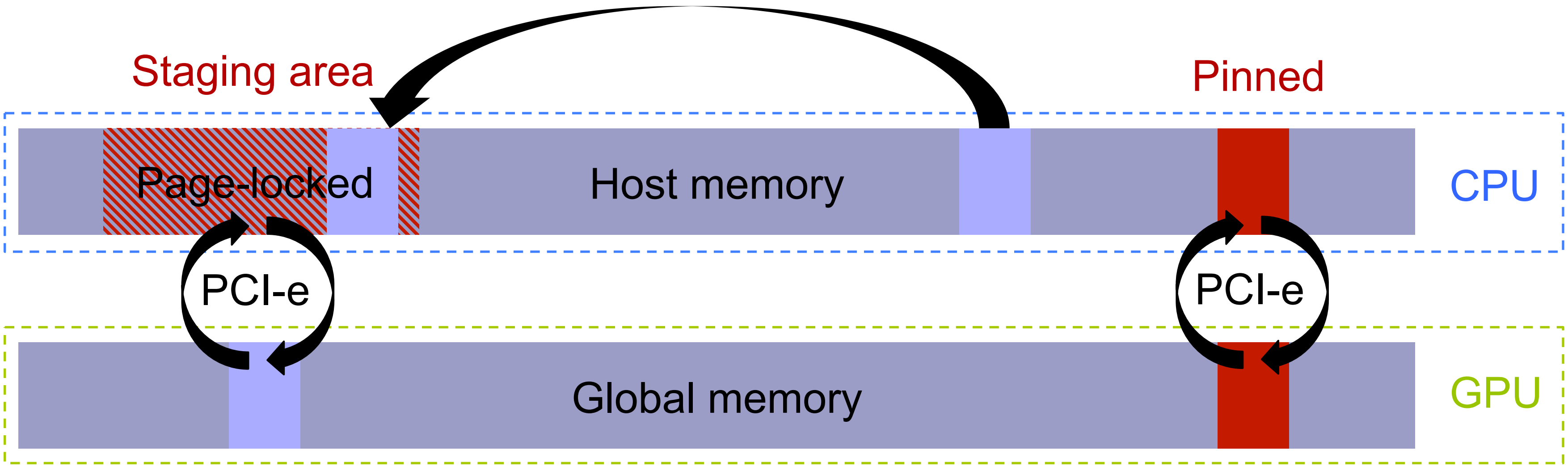
# Pinned host memory



# Pinned host memory



# Pinned host memory



# Allocating pinned host memory

- `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 a register 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

```
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 and 5 blocks
   use_global_memory<<<5, 6>>>(d_a);
   cudaDeviceSynchronize();

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

# Shared memory allocation

- Static allocation using `__shared__`

```
#define N 128

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

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

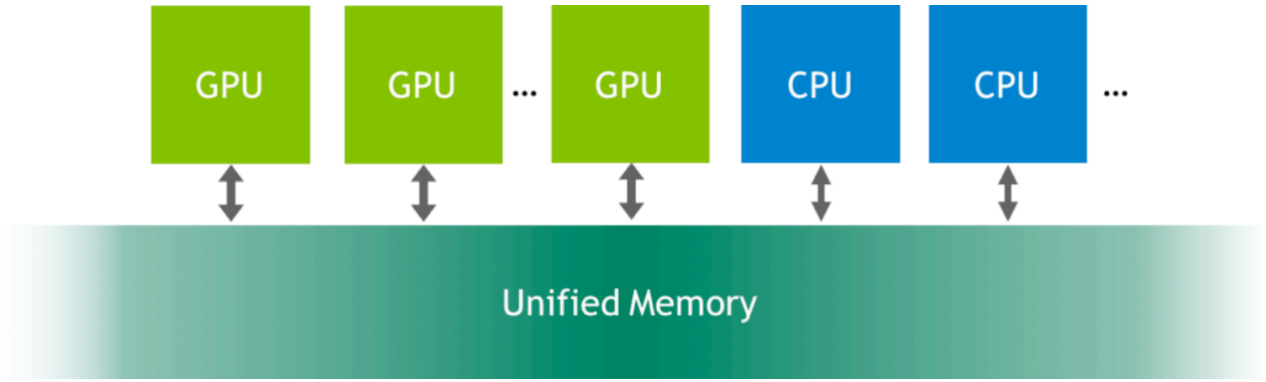
```
kernel<<<dimGrid, dimBlock, N * sizeof(double)>>>(...);

__global__ void kernel(...)
{
    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

    // Write back to global memory in reverse order (per block!)
    // <<<5,6>>>: [0,1,2,3,..] -> [5,4,..,0, | 11,10,..,6, | 17,..]
    a[tid] = smem[THREADS_PER_BLOCK - threadIdx.x - 1];
}
```

# Unified memory

# Unified memory




- CUDA creates a pool of managed memory that is shared between the CPU and GPU
- Managed memory is accessible on CPU and GPU from program using the same unique C pointer
- A true “virtual memory” divided into pages
- Data automatically migrates to CPU and GPU

# Unified memory

- Advantages
  - Ease of programming
  - Data is migrated on demand – no memCpys
    - Offers the performance of local data on the GPU
    - while providing the ease of use of globally shared data
  - Very efficient with complex data structures (e.g. linked lists, structures with pointers, ... )
- Disadvantages
  - The physical location of data is invisible to the programmer and may be changed at any time
  - Carefully tuned CUDA programs that efficiently over-lap execution with data transfers may perform better

# Allocating managed memory

- `cudaMallocManaged()`
  - Dynamically allocate managed memory
  - Same parameters as `cudaMalloc()`
- `cudaFree()`
  - Works also for managed memory
- `cudaMemPrefetchAsync()`
  - For explicitly migrating memory to the GPU if you know this would be best for performance
- `cudaMemAdvice()`
  - Provide hints on how the data will actually be used
    - E.g., **`cudaMemAdviseSetPreferredLocation`**

# Managed memory example

```
int main() {
   // Managed array pointer
   int *a;

   // Alloc mem in managed pool
   cudaMallocManaged((void **)&a, N * sizeof(int));

   // Launch kernel using 6 threads per block for 5 blocks
   use_global_memory<<<5, 6>>>(a);

   // Make sure we are finished - no race condition
   cudaDeviceSynchronize();

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

   // Cleanup
   cudaFree(a);
}
```

# 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 always scale the PCI-e bandwidth!

# 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_elms/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`

# Exercises

- Finish up the first two exercises
  - ex1_deviceQuery
  - ex2_helloworld

- Do the third exercises
  - ex3_mandelbrot

# End of lecture