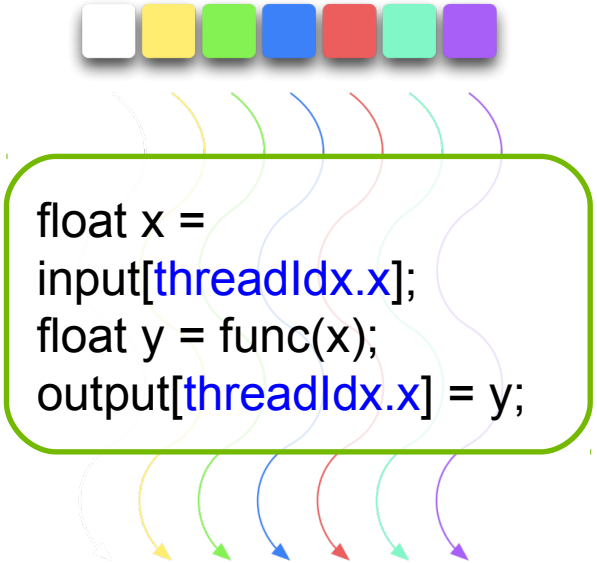
# Programming Model

April 24, 2017

# CUDA Kernels: Parallel Threads

- A kernel is a function executed on the GPU as an array of threads in parallel
- All threads execute the same code, but can take different paths
- Each thread has an ID
  - *Select input/output data*
  - *Control decisions*



# Scale Kernel

```
void scale(float alpha,
                        float* A,
                        float* C,
                        int m)
{
    int i = 0;
    for ( i=0; i<m; ++i)
        C[i] = alpha * A[i];
}
```

```
__global__ void scale(float alpha,
                        float* A,
                        float* C,
                        int m)
{
    int i = blockDim.x*blockIdx.x+threadIdx.x;
    if ( i < m)
        C[i] = alpha * A[i];
}
```

# Getting data in and out with Unified Memory

- GPU has separate memory, but transfers can be managed by runtime
- Allocate memory with cudaMallocManaged
- Free memory

# Allocate memory

```
cudaMallocManaged(T** pointer, size_t nbytes)
```

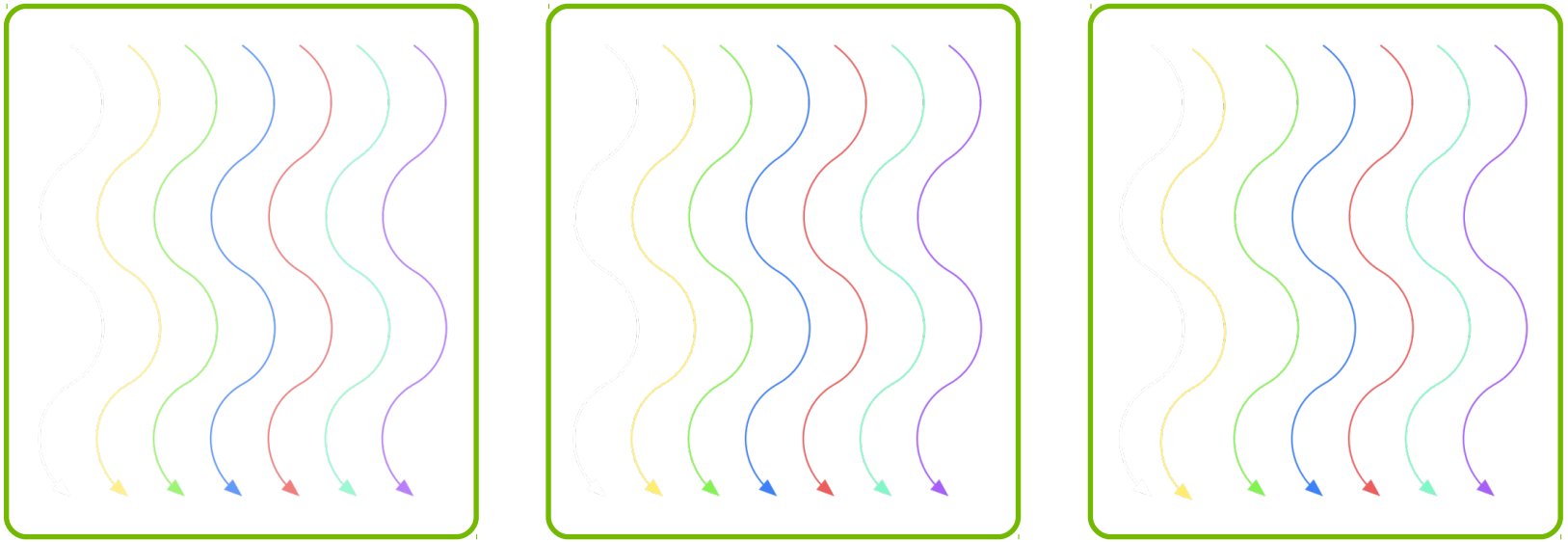
Example:

```
// Allocate a vector of 2048 floats for host and device
float * a;
int n = 2048;
cudaMallocManaged(&a, n * sizeof(float));
```

Get size of a float

Address of pointer

# CUDA Kernels: Subdivide into Blocks



- Threads are grouped into blocks

# Define dimensions of thread block

```
dim3 blockDim(size_t blockDimX, size_t blockDimY,
              size_t blockDimZ)
```

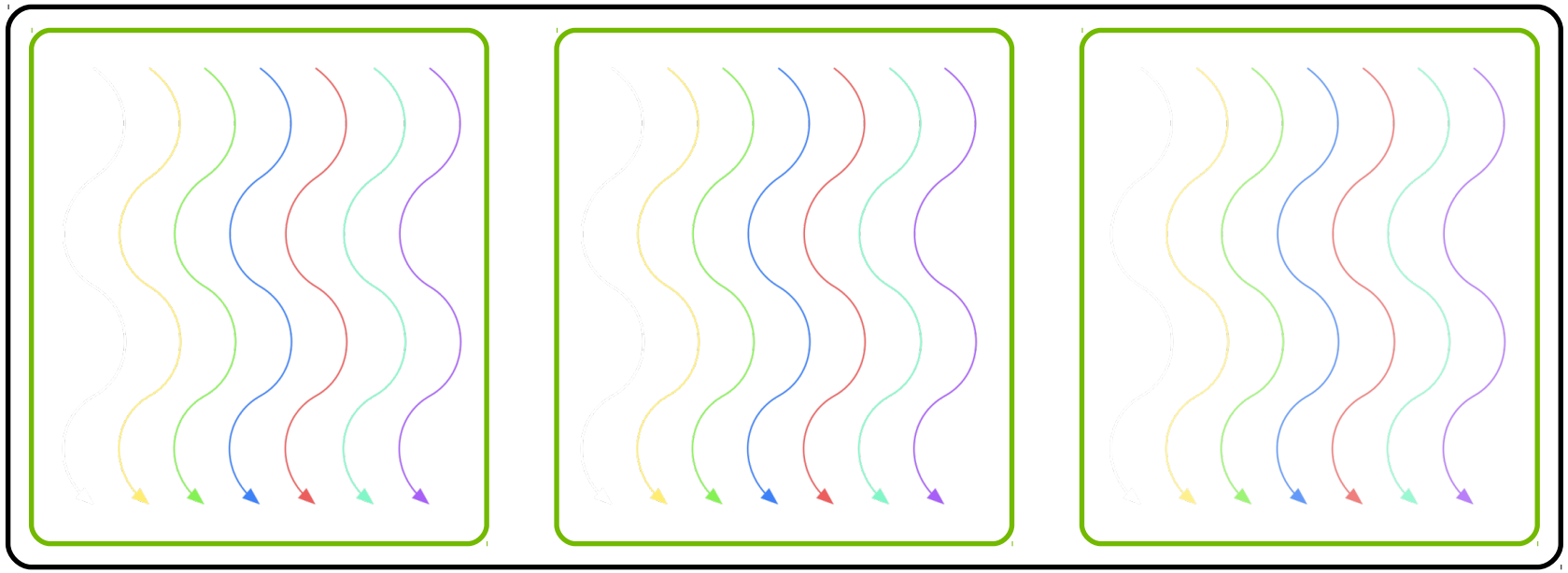
On JURECA & JURON:

- Max. dim. of a block: 1024 x 1024 x 64
- Max. number of threads per block: 1024

Example:

```
// Create 3D thread block with 512 threads
dim3 blockDim(16, 16, 2);
```

# CUDA Kernels: Subdivide into Blocks



- Threads are grouped into blocks
- Blocks are grouped into a grid

# Define dimensions of grid

```
dim3 gridDim(size_t blockDimX, size_t blockDimY,
             size_t blockDimZ)
```

On JURECA & JURON:

- Max. dim. of a grid: 2147483647 x 65535 x 65535

Example:

```
// Dimension of problem: nx x ny = 1000 x 1000
dim3 blockDim(16, 16) // Don't need to write z = 1
int gx = (nx % blockDim.x==0) ? nx / blockDim.x : nx / blockDim.x + 1
int gy = (ny % blockDim.y==0) ? ny / blockDim.y : ny / blockDim.y + 1
dim3 gridDim(gx, gy);
```

**Watch out!**

# Call the kernel

```
kernel<<<int gridDim, int blockDim>>>([arg]*)
```

Call returns immediately! → Kernel executes asynchronously
Example:

```
scale<<<m/blockDim, blockDim>>>(alpha, a, c, m)
```

# Calling the kernel

- Define dimensions of thread block

```
dim3 blockDim(size_t blockDimX, size_t blockDimY,
              size_t blockDimZ)
```

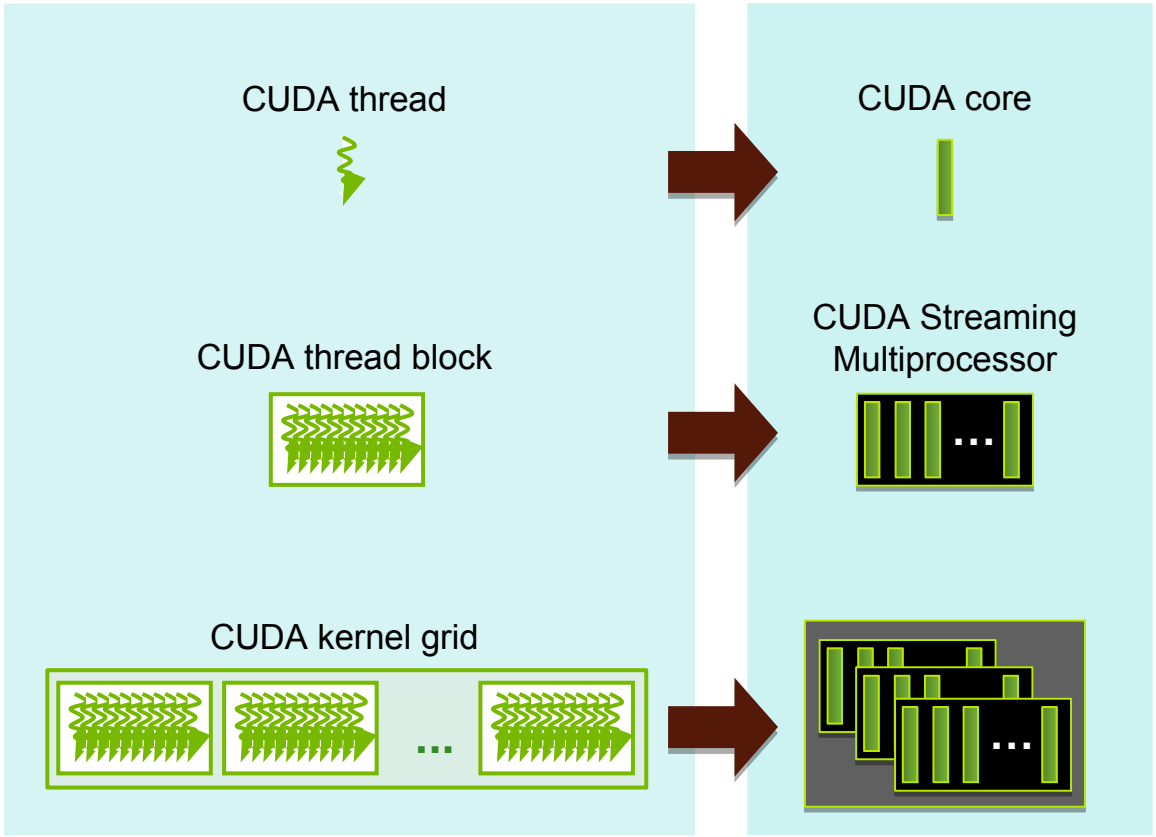
- Define dimensions of grid

```
dim3 gridDim(size_t gridDimX, size_t gridDimY,
             size_t gridDimZ)
```

- Call the kernel

```
kernel<<<dim3 gridDim, dim3 blockDim>>>([arg]*)
```

# Kernel Execution



# Free device memory

```
cudaFree(void* pointer)
```

Example:

```
// Free the memory allocated by a on the device
cudaFree(a);
```

# Exercise

CudaBasics/exercises/tasks/scale_vector

Compile with nvcc -o scale_vector scale_vector_um.cu

# Getting data in and out

- GPU has separate memory
- Allocate memory on device
- Transfer data from host to device
- Transfer data from device to host
- Free device memory

# Allocate memory on device

```
cudaMalloc(T** pointer, size_t nbytes)
```

Example:

```
// Allocate a vector of 2048 floats on device
float * a_gpu;
int n = 2048;
cudaMalloc(&a_gpu, n * sizeof(float));
```

Get size of a float

Address of pointer

# Copy from host to device

```
cudaMemcpy(void* dst, void* src, size_t nbytes,
           enum cudaMemcpyKind  dir)
```

Example:

```
// Copy vector of  floats a of length n=2048 to a_gpu on
device
cudaMemcpy(a_gpu, a, n * sizeof(float),
                                  cudaMemcpyHostToDevice);
```

# Copy from device to host

```
cudaMemcpy(void* dst, void* src, size_t nbytes,
           enum cudaMemcpyKind  dir)
```

Example:

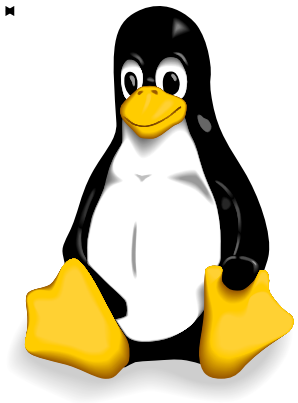
```
// Copy vector of floats a_gpu of length n=2048 to a on host
cudaMemcpy(a, a_gpu, n * sizeof(float),
                                  cudaMemcpyDeviceToHost);
```

Note the order

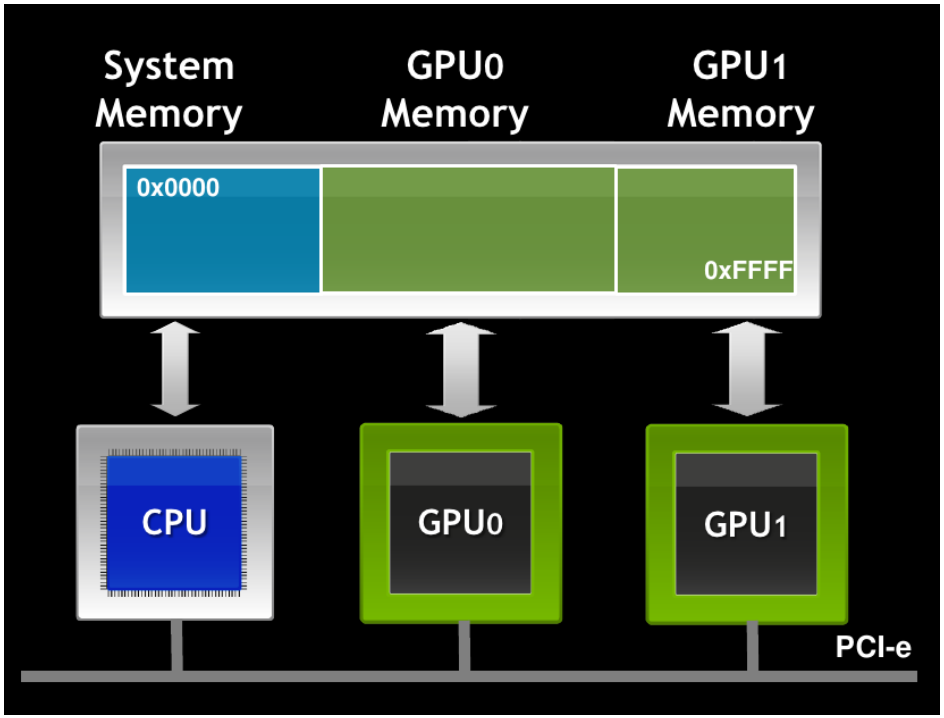
Changed flag

# Unified Virtual Address Space (UVA)

Windows



64bit




2.0

cudaMalloc*(...)
cudaHostAlloc(...) } return UVA pointers
cudaMemcpy*(..., cudaMemcpyDefault)

# Getting data in and out

- Allocate memory on device

```
cudaMalloc(void** pointer, size_t nbytes)
```

- Transfer data between host and device

```
cudaMemcpy(void* dst, void* src, size_t nbytes,
           enum cudaMemcpyKind  dir)
dir = cudaMemcpyHostToDevice
dir = cudaMemcpyDeviceToHost
```

- Free device memory

```
cudaFree(void* pointer)
```

# Exercise Scale Vector

Allocate memory on device

```
cudaMalloc(T** pointer, size_t nbytes)
```

Transfer data between host and device

```
cudaMemcpy(void* dst, void* src,
           size_t nbytes,
           enum cudaMemcpyKind dir)
 dir = cudaMemcpyHostToDevice
 dir = cudaMemcpyDeviceToHost
```

- Free device memory

```
cudaFree(void* pointer)
```

Define dimensions of thread block

```
dim3 blockDim(size_t blockDimX,
              size_t blockDimY,
              size_t blockDimZ)
```

Define dimensions of grid

```
dim3 gridDim(size_t gridDimX, size_t
             gridDimY,
             size_t gridDimZ)
```

Call the kernel

```
kernel<<<dim3 gridDim,
         dim3 blockDim>>>([arg]*)
```

# Exercise

CudaBasics/exercises/tasks/jacobi_w_explicit_transfer

Compile with make jacobi.