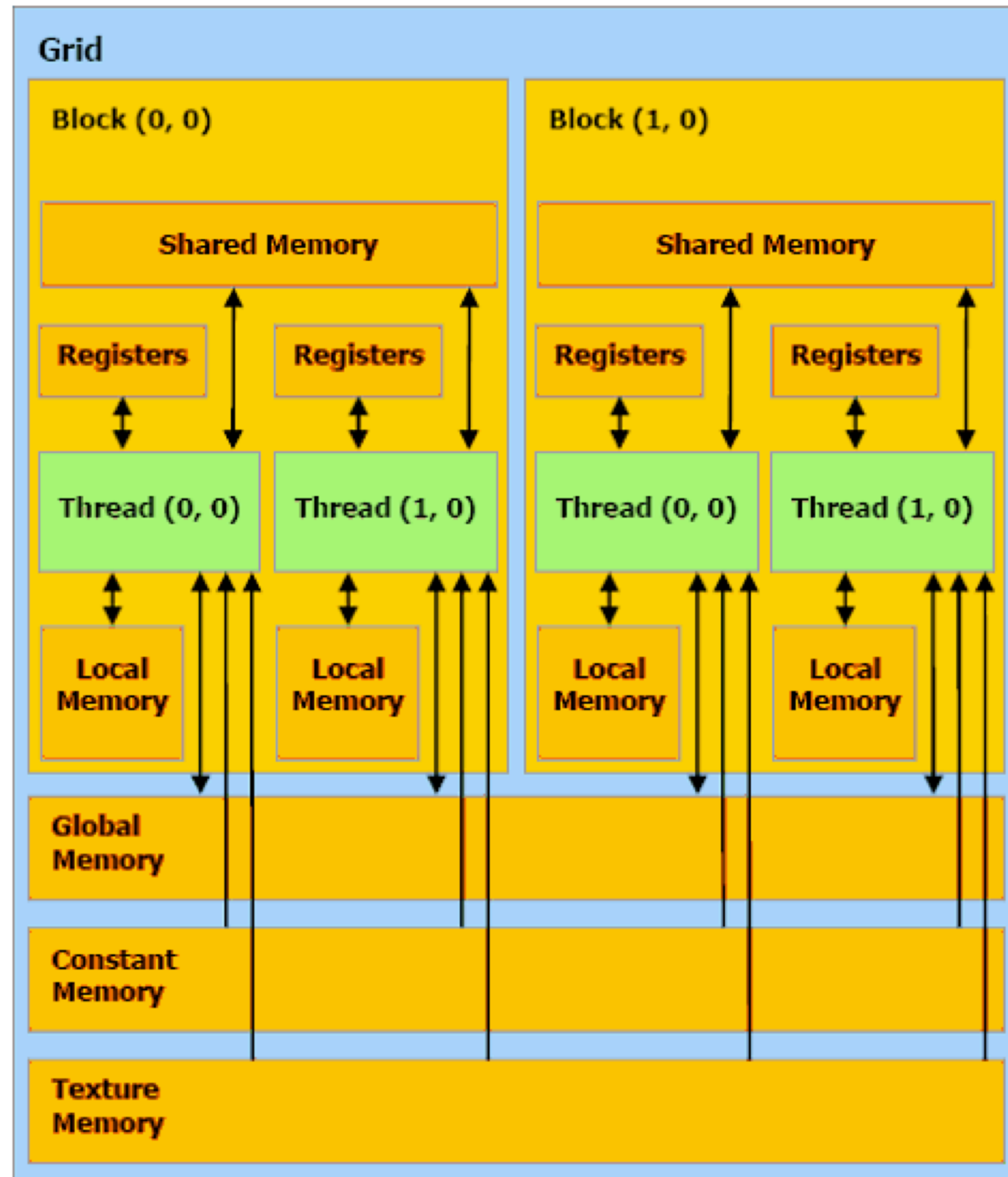
GPU Memory

- Local **registers** per thread.
- A parallel data cache or **shared** memory that is shared by all the threads.
- A read-only **constan**t cache that is shared by all the threads.
- A read-only **texture** cache that is shared by all the processors.
- A **local** cached memory like registers

Memory access:100 times more time to access local/global memory. Maximize shared/ register memory.



CUDA Memory Types

Global memory
- Slow and uncached, all threads

Texture memory (read only)
- Cache optimized for 2D access, all threads

Constant memory (read only)
- Slow, cached, all threads

Shared memory
- Fast, bank conflicts; limited; threads in block

Registers
- Fast, only for one thread

Local memory
- For what doesn’t fit in registers, slow but cached, one thread

Variables

Variable declaration	Memory	Scope	Lifetime
`int localVar;`	register	thread	thread
`int localArray[10];`	local	thread	thread
`__shared__ int sharedVar;`	shared	block	block
`__device__ int globalVar;`	global	grid	application
`__constant__ int constantVar;`	constant	grid	application

Access Penalty

Variable declaration	Memory	Performance penalty
`int localVar;`	register	1x
`int localArray[10];`	local	100x
`__shared__ int sharedVar;`	shared	1x
`__device__ int globalVar;`	global	100x
`__constant__ int constantVar;`	constant	1x

Memory

Registers:
The fastest form of memory on the multi-processor. Is only accessible by the thread. Has the lifetime of the thread.

Shared Memory:
Can be as fast as a register when there are no bank conflicts or when reading from the same address. Accessible by any thread of the block from which it was created. Has the lifetime of the block.

Constant Memory:
Accessible by all threads. Lifetime of application. Fully cached, but limited.

Global memory:
Potentially 150x slower than register or shared memory -- watch out for uncoalesced reads and writes. Accessible from either the host or device. Has the lifetime of the application—that is, it persistent between kernel launches.

Local memory:
A potential performance gotcha, it resides in global memory and can be 150x slower than register or shared memory. Is only accessible by the thread. Has the lifetime of the thread.

Where to declare variables?

Can Host access?

yes → Outside of any function:

global
constant

no → In kernel:

register
local
shared

Global Memory: DRAM on card

Found by deviceQueryDrv; I have 1024 Mb

Declared outside of any function

```
__device__ int globalArray[256];
```

Assigned by **cudaMemcpy**

cudaMemcpy is blocking transfer; host thread waits until transfer complete

or

```
int *myDeviceMemory = 0;
cudaMalloc(&myDeviceMemory, 256 * sizeof(int));
```

Global Memory

CUDA streams are a sequence of operations that execute on the device in the order they are issued by the host. Different streams can interleave operations.

When no stream specified, the default (null) stream is assumed.

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a)
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

All on the default stream. Kernel call waits until cudaMemcpy is complete. increment kernel has to complete before the cudaMemcpy to the host is started.

Alternatively:

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a)
myCpuFunction(b)   //this function will run while the DeviceToHost memcpy occurs
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Global Memory

Asynchronous transfer

Using pinned memory with cudaHostAlloc, then
cudaMemcpyAsync is a non-blocking version of cudaMemcpy.
Requires using a stream ID.

```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0)
kernel_function <<<grid, block >>>(a_d);
cpu_function();
```

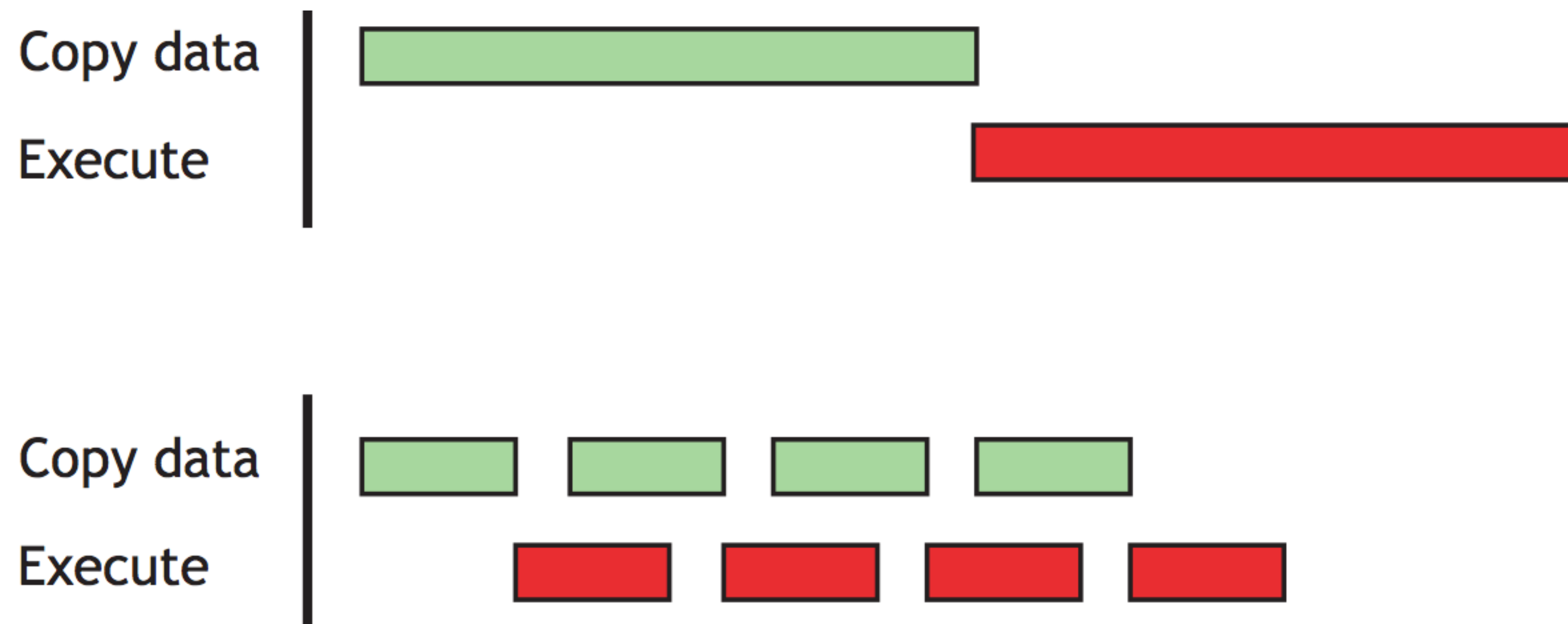
kernel function uses default stream 0, so no synchronization required as kernel uses default as well. Same as before.

cpu_function does not wait for memory transfer nor kernel_function to finish.

Concurrent copy and kernel execution

Availability: DeviceQueryDrv:
"Concurrent copy and kernel execution: Yes with 1 copy engine(s)"

Compute_capability > 1.1




```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, stream1);
kernel<<<grid, block, 0, stream2>>>(otherData_d);
```

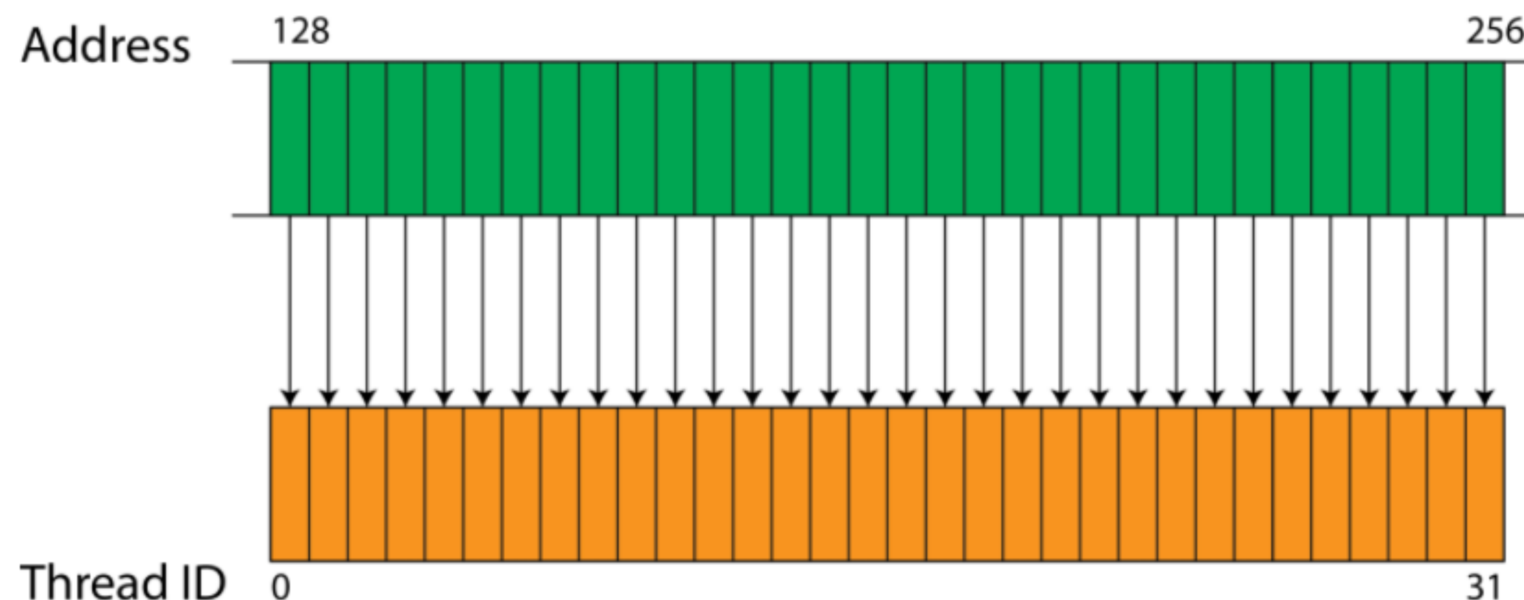
See http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/

Coalesced Global Memory Access

Threads in a block are computed a warp at a time (32 threads).

Global data is read or written in as few transactions as possible by combining memory access requests into a single transaction . This is referred to the device **coalescing** memory stores and reads.

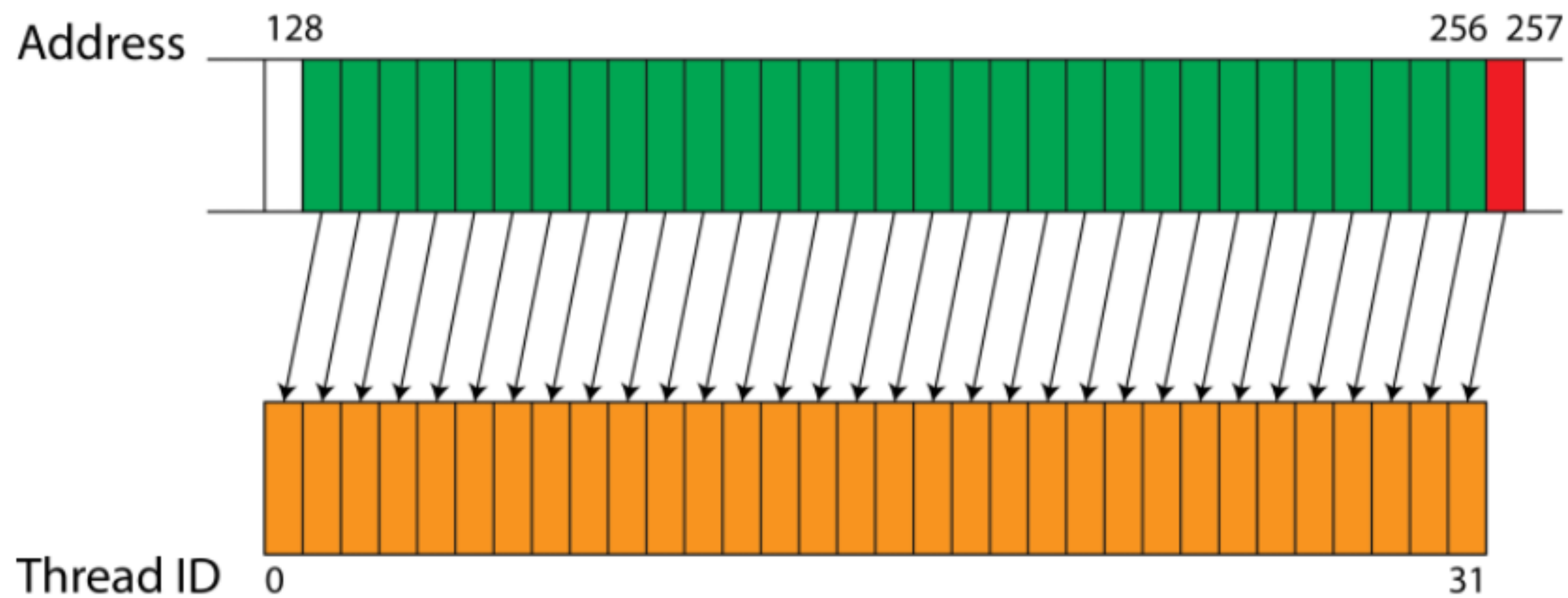
Every successive 128 bytes can be accessed by a warp (or 32 single precision words).



All 32 reads done in one step. So blockDim.x multiples of 32 better.

Non-aligned Memory

Not in successive 128 blocks of memory; **twice** as long to read



Constant Memory

For data that does not change over the course of the computation. It is read only.

64k memory available on my machine (see deviceQueryDrv), but it is cached, so there is only one clock cycle to read as opposed to 100+ for global memory

Using constant memory: Declare it outside of main()

```
__constant__  float cdata;  //available in all scopes
```

To get numbers into constant memory variable cdata, use

```
cudaMemcpytoSymbol( (const char * symbol, const void * src, size_t count ,
                    size_t offset=0, enum cudaMemcpyKind )
```

Best if all threads in warp read the same constant data, otherwise slower.

Constant Memory

deviceQueryDrv;   I have 65536 bytes (64 kB)

Declare outside of any function

```
__constant__  float  var;                    //available in all functions
```

Note: constant memory is available to all threads, like global memory; however, the data is cached.

As fast as a register if all threads read same address.

```
//copy data from host to constant memory in main():
cudaMemcpyToSymbol (var, &host_var, data_size );

//var is available to kernels
__global__ void kernel (float *a, int N)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        if (idx<N)
        {
                a[idx] = a[idx]* var;
        }
}
```

Class Problem

Write a GPU program to calculate the potential energy of N particles. The particles have random vertical positions between 0 and 100 m and the mass of each particle is the same (m = 0.2 kg).

N=2048

g= 9.81m/s^2 , put into constant memory

$$P.E. = \sum_{i=0}^{N} m g z_i$$

Note: #include <stdlib.h> provides rand() on host

(float) rand() %100 should give you a good random elevation

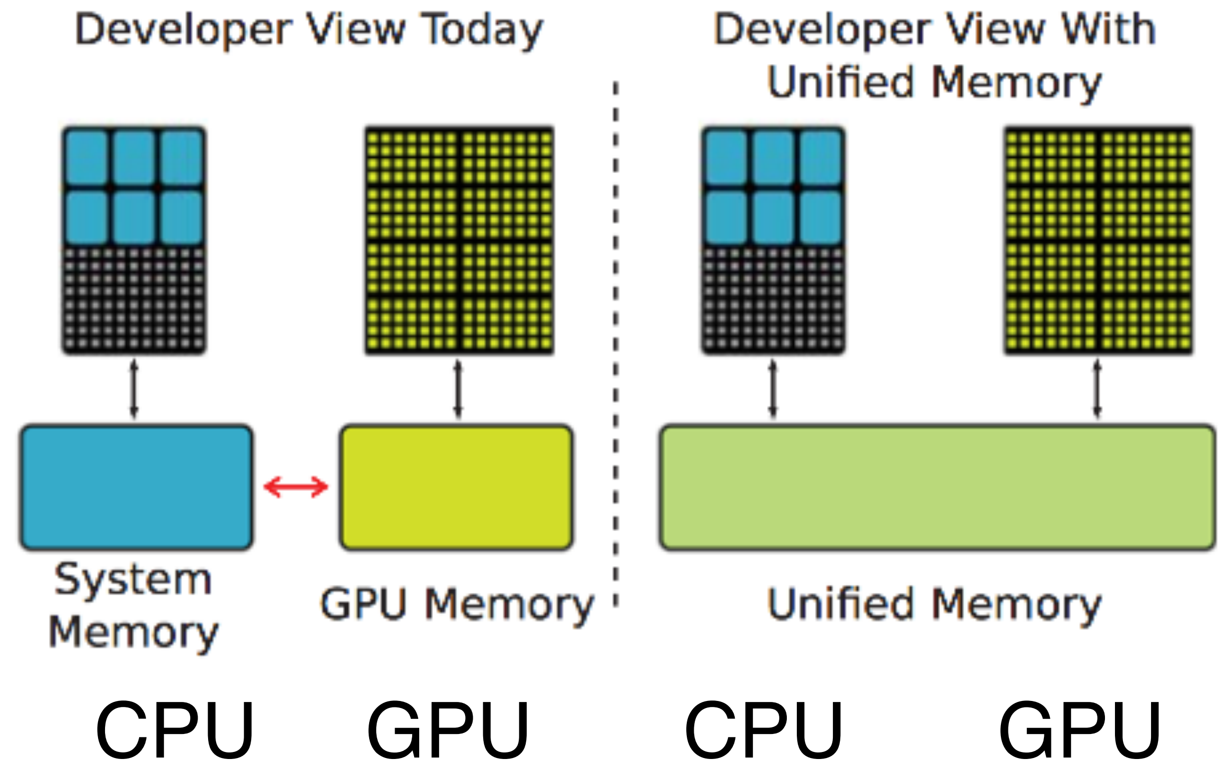
NOTE: rand() by itself is repeatable;
adding srand(time(NULL)) first, will randomize rand(); also #include <time.h>

PotentialEnergy.cu

Unified Memory: CUDA 6.0

Existing model:

Allocate memory on host
Allocate memory on device
Copy data from host to device
Operate on the GPU data
Copy data back to host



Unified memory model:

Allocate memory (same to CPU as malloc; same to GPU as cudaMalloc)
Operate on data on GPU

<u>NOTE: Linux or Windows only, right now</u>

New memory model : CUDA 6+

```
int N = 2048;
float * data;

cudaMallocManaged(&data, N);

…..generate data

kernel<<<.….>>> (data, N)

cudaDeviceSynchronize();

// Synchronize is to get GPU to finish before doing anything with the data on CPU

cudaFree(data);
```

— — — — — — — — — — — — — — —

See PotentialEnergyUnifiedMem.cu as compared to PotentialEnergy.cu

NOTE: no d_data, same data on both devices

PotentialEnergyUnifiedMem.cu