CUDA Programming: Unified Memory

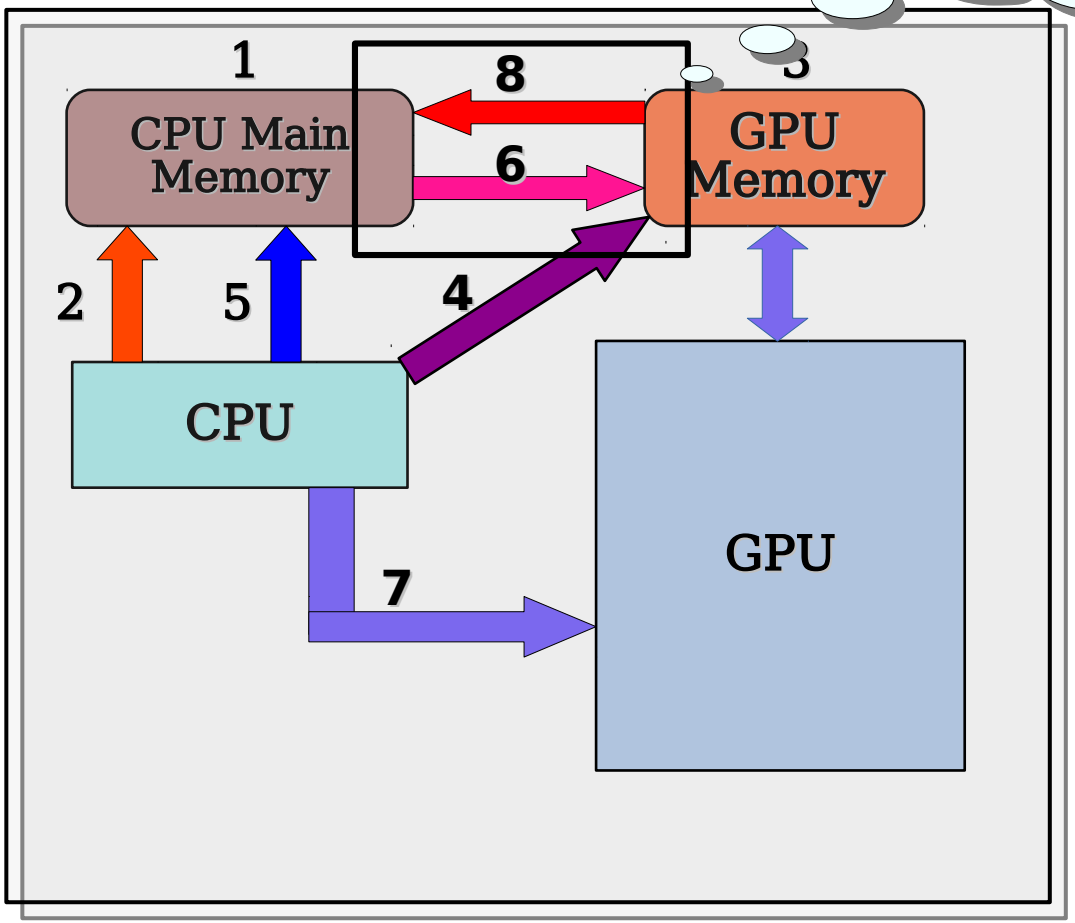
By

Dr. Nileshchandra K. Pikle

A certified CUDA Programming Instructor by NVIDIA

Email: nilesh.pikle@gmail.com
Contact No. +91 7276834418

Why Unified Memory



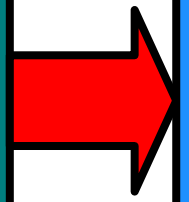
1. Declare CPU variables
2. Allocate memory to CPU variables
3. Declare GPU variables
4. Allocate memory to GPU variables
5. Initialize data in CPU memory
6. Copy data from CPU memory to GPU memory
7. CPU instruct to GPU for parallel Execution
8. Copy results back from GPU Memory to CPU memory

9. Free both CPU & GPU memories

Why Unified Memory

Prior to CUDA 6.0

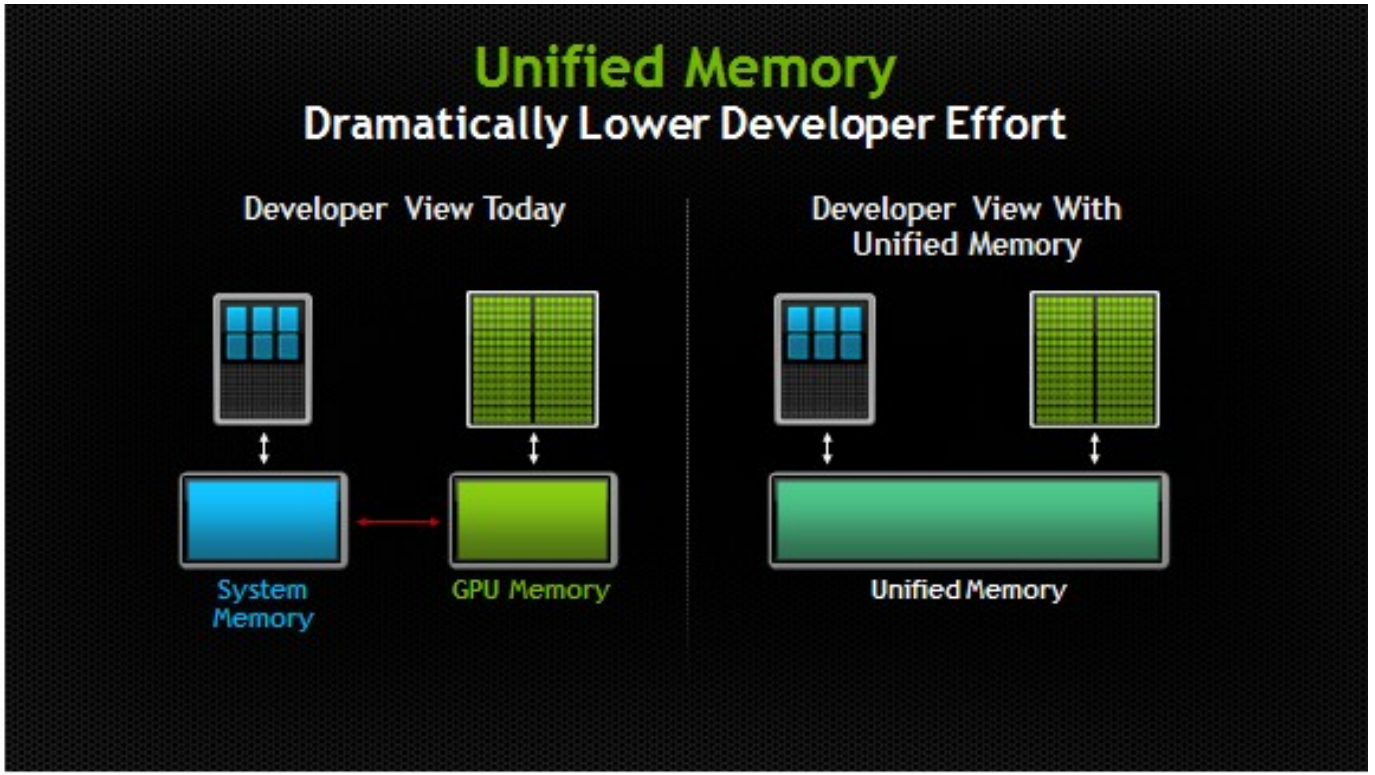
1. Declare Host variables
2. Declare Device variables
3. Allocate memory to Host variables
4. Allocate memory to Device variables
5. Transfer data from Host to Device
6. Parallel Kernel execution
7. Transfer data from Device to Host



From CUDA 6.0 onwards

1. **Declare variables**
2. **Allocate unified Memory**

Why Unified Memory



No need to invoke CUDA APIs such as cudaMalloc() cudaMemcpy()

Why Unified Memory

CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

Reduces lines of code
Simplicity
Reduces programmers efforts
Automatic data transfer

Unified Memory

- **Unified Memory** is a **single memory address space** accessible from any processor in a system.
- When **unified memory** is allocated it is **not** yet resident on Host or device.
- When either the host or device attempts to access the memory, **a page fault** will occur.
- At this point the host or device will migrate the needed data in batches.
- Similarly, at any point when the CPU, or any GPU in the accelerated system, attempts to access memory not yet resident on it, page faults will occur and trigger its migration.

Unified Memory Working

```
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
// initialize x and y arrays on the host
for (int i = 0; i < N; i++) {
  x[i] = 1.0f;
  y[i] = 2.0f;
}
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;

add<<<numBlocks, blockSize>>>(N, x, y);
 // Wait for GPU to finish before accessing on host

cudaDeviceSynchronize();
// Check for errors (all values should be 3.0f)
float maxError = 0.0f;
for (int i = 0; i < N; i++)
  maxError = fmax(maxError, fabs(y[i]-3.0f));
```

Unified Memory Working

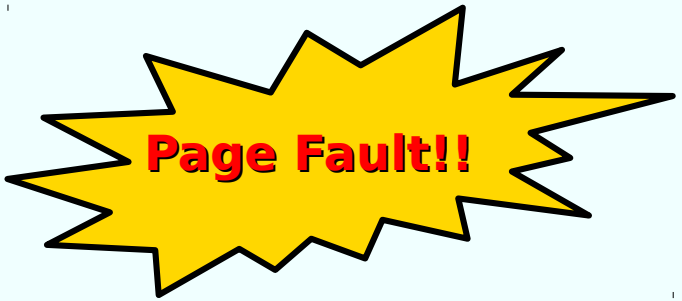
```
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
// initialize x and y arrays on the host
for (int i = 0; i < N; i++) {
  x[i] = 1.0f;
  y[i] = 2.0f;
}
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;

add<<<numBlocks, blockSize>>>(N, x, y);
 // Wait for GPU to finish before accessing on host

cudaDeviceSynchronize();
// Check for errors (all values should be 3.0f)
float maxError = 0.0f;
for (int i = 0; i < N; i++)
  maxError = fmax(maxError, fabs(y[i]-3.0f));d
```

Memory allocated using CudaMallocManaged() is neither on CPU nor on GPU

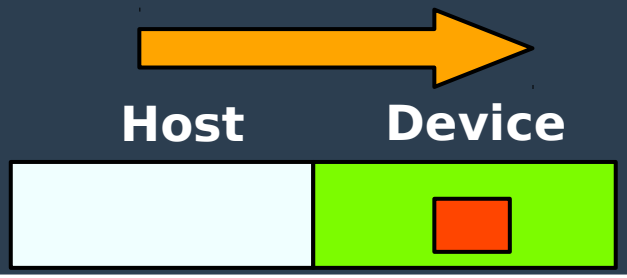
Unified Memory Working


```
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
// initialize x and y arrays on the host
for (int i = 0; i < N; i++) {
  x[i] = 1.0f;
  y[i] = 2.0f;
}
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;

add<<<numBlocks, blockSize>>>(N, x, y);
 // Wait for GPU to finish before accessing on host

cudaDeviceSynchronize();
// Check for errors (all values should be 3.0f)
float maxError = 0.0f;
for (int i = 0; i < N; i++)
  maxError = fmax(maxError, fabs(y[i]-3.0f));
```

Page fault occurs as first Time CPU accesses the Unified memory
Now data resides on CPU



Unified Memory Working




```
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
// initialize x and y arrays on the host
for (int i = 0; i < N; i++) {
  x[i] = 1.0f;
  y[i] = 2.0f;
}
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;

add<<<numBlocks, blockSize>>>(N, x, y);
 // Wait for GPU to finish before accessing on host

cudaDeviceSynchronize();
// Check for errors (all values should be 3.0f)
float maxError = 0.0f;
for (int i = 0; i < N; i++)
  maxError = fmax(maxError, fabs(y[i]-3.0f));
```

Page migration engine Transfers data data from Host to Device

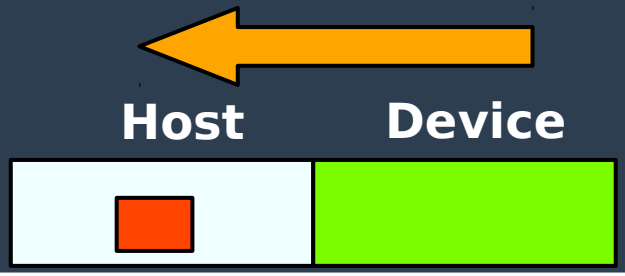
Page Fault!!

Unified Memory Working

```
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
// initialize x and y arrays on the host
for (int i = 0; i < N; i++) {
  x[i] = 1.0f;
  y[i] = 2.0f;
}
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;

add<<<numBlocks, blockSize>>>(N, x, y);
 // Wait for GPU to finish before accessing on host

cudaDeviceSynchronize();
// Check for errors (all values should be 3.0f)
float maxError = 0.0f;
for (int i = 0; i < N; i++)
  maxError = fmax(maxError, fabs(y[i]-3.0f));
```

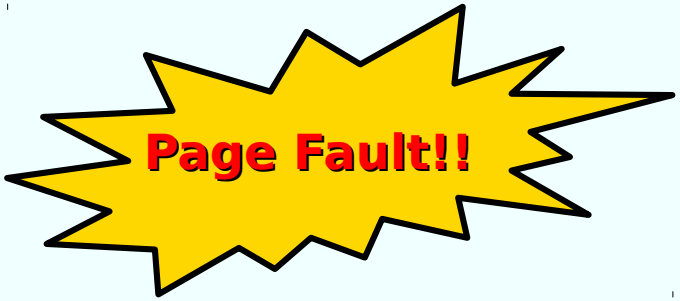
Synchronization ensures Host will wait till kernel Finishes execution

Unified Memory Working




```
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
// initialize x and y arrays on the host
for (int i = 0; i < N; i++) {
  x[i] = 1.0f;
  y[i] = 2.0f;
}
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;

add<<<numBlocks, blockSize>>>(N, x, y);
 // Wait for GPU to finish before accessing on host

cudaDeviceSynchronize();
// Check for errors (all values should be 3.0f)
float maxError = 0.0f;
for (int i = 0; i < N; i++)
  maxError = fmax(maxError, fabs(y[i]-3.0f));
```

Data transfered from Device to Host



Unified Memory

- Migrating data on demand incurs an **overhead cost** that would be better avoided.
- The use of unified memory is favourable when access patterns are **sporadic**
- If you are sure about the block of data required at CPU/GPU at compile time then use **cudaMemPrefetchAsync()**