CUDA

Part 2

Compute **U**nified **D**evice **A**rchitecture

Adam Simpson
Aug. 2011

VecAdd.cu

```
#include <stdlib.h>
#include <math.h>

// Write CUDA kernel

int main( int argc, char *argv[] )
{
    // Length of array
    int n = 100000;

    // Declare and allocate device arrays

    // Input arrays
    float *a;
    float *b;
    // Output array
    float *c;

    // Size, in bytes, of each array
    size_t bytes = n*sizeof(float);

    // Allocate memory for each array
    a = (float*)malloc(bytes);
    b = (float*)malloc(bytes);
    c = (float*)malloc(bytes);

    // Initialize content of input array
    int i;
    for(i=0; i<n; i++) {
        a[i] = sinf(i)*sinf(i);
        b[i] = cosf(i)*cosf(i);
    }
```

```
    // Copy input arrays to device

    // Setup and launch kernel

    // Copy output array to host

    // release device memory

    // Release memory
    free(a);
    free(b);
    free(c);

    return 0;
}
```

VecAdd.cuf

```
module kernel
    contains
    ! CUDA Fortran kernel
end module kernel

program main
    use cudafor
    use kernel

    ! Size of vectors
    integer :: n = 100000

    ! Declare and allocate device arrays

    ! Input vectors
    real,dimension(:),allocatable :: a
    real,dimension(:),allocatable :: b
    ! Output vector
    real,dimension(:),allocatable :: c

    integer :: i
    real :: sum

    ! Allocate memory for each vector
    allocate(a(n))
    allocate(b(n))
    allocate(c(n))
```

```
! Initialize content of input vectors
do i=1,n
    a(i) = sin(i*1.0)*sin(i*1.0)
    b(i) = cos(i*1.0)*cos(i*1.0)
enddo

! Copy input arrays to device

! Setup and launch kernel

! Copy output array to host

! Release device memory

! Release memory
deallocate(a)
deallocate(b)
deallocate(c)

end program
```

Vector Addition Kernel

```
// CUDA kernel
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
   // Get our global thread ID
   int id = blockIdx.x * blockDim.x + threadIdx.x;

   // Vector addition
   if (id < n)
      c[id] = a[id] + b[id];
}
```

VecAdd Kernel

```
! Kernel definition
attribute(global) subroutine vecAdd(n, A, B, C)
    integer, value :: n
    real, device :: A(n), B(n), C(n)
    integer :: id
    id =  (blockidx%x-1) * (blockdim%x) + threadidx%x

    if(id <= n) then
        C(id) = A(id) + B(id)
    endif
end subroutine vecAdd
```

VecAdd.cu

```
#include <stdlib.h>
#include <math.h>

// CUDA kernel
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
  // Get our global thread ID
  int id = blockIdx.x * blockDim.x + threadIdx.x;

  // Vector addition
  if (id < n)
    c[id] = a[id] + b[id];
}

int main( int argc, char *argv[] )
{
    // Length of array
    int n = 100000;

    // Declare and allocate device arrays

    // Input arrays
    float *a;
    float *b;
    // Output array
    float *c;

    // Size, in bytes, of each array
    size_t bytes = n*sizeof(float);

    // Allocate memory for each array
    a = (float*)malloc(bytes);
    b = (float*)malloc(bytes);
    c = (float*)malloc(bytes);

    // Initialize content of input array
    int i;
    for(i=0; i<n; i++) {
        a[i] = sinf(i)*sinf(i);
        b[i] = cosf(i)*cosf(i);
    }

    // Copy input arrays to device

    // Setup and launch kernel

    // Copy output array to host

    // Release device memory

    // Release memory
    free(a);
    free(b);
    free(c);

    return 0;
}
```

VecAdd.cuf

```
module kernel
    contains
    ! Kernel definition
    attributes(global) subroutine vecAdd(n, A, B, C)
      integer, value :: n
      real, device :: A(n), B(n), C(n)
      integer :: id
      id =  (blockidx%x-1)*(blockdim%x) + threadidx%x

      if(id <= n) then
          C(id) = B(id) + D(id)
      endif
    end subroutine vecAdd
end module kernel

program main
    use cudafor
    use kernel

    ! Size of vectors
    integer :: n = 100000

    ! Input vectors
    real,dimension(:),allocatable :: a
    real,dimension(:),allocatable :: b
    ! Output vector
    real,dimension(:),allocatable :: c

    integer :: i
    real :: sum
```

```
    ! Allocate memory for each vector
    allocate(a(n))
    allocate(b(n))
    allocate(c(n))

! Initialize content of input vectors
 do i=1,n
     a(i) = sin(i*1.0)*sin(i*1.0)
     b(i) = cos(i*1.0)*cos(i*1.0)
 enddo

 ! Copy input arrays to device

 ! Setup and launch kernel

 ! Copy output array to host

 ! Release device memory

 ! Release memory
 deallocate(a)
 deallocate(b)
 deallocate(c)

end program
```

Declare and allocate Arrays

```
// Host input arrays
float *h_A;
float *h_B;
// Host output array
float *h_C;

// Device input arrays
float *d_A;
float *d_B;
//  Device output array
float *d_C;

// Size, in bytes, of each array
size_t bytes = n*sizeof(float);

// Allocate host arrays
h_A = (float*)malloc(bytes);
h_B = (float*)malloc(bytes);
h_C = (float*)malloc(bytes);

// Allocate device arrays
cudaMalloc(&d_A, bytes);
cudaMalloc(&d_B, bytes);
cudaMalloc(&d_C, bytes);
```

Declare and allocate Arrays

```
! Input host arrays
real,dimension(:),allocatable :: h_a
real,dimension(:),allocatable :: h_b
! Output array
real,dimension(:),allocatable :: h_c

! Input device arrays
real,dimension(:),allocatable,device :: d_a
real,dimension(:),allocatable,device :: d_b
! Output device array
real,dimension(:),allocatable,device :: d_c

! Allocate host arrays
allocate(h_a(n))
allocate(h_b(n))
allocate(h_c(n))

! Allocate device arrays
allocate(d_a(n))
allocate(d_b(n))
allocate(d_c(n))
```

VecAdd.cu

```
#include <stdlib.h>
#include <math.h>

// CUDA kernel
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
  // Get our global thread ID
  int id = blockIdx.x * blockDim.x + threadIdx.x;

  // Vector addition
  if (id < n)
    c[id] = a[id] + b[id];
}

int main( int argc, char *argv[] )
{
    // Length of array
    int n = 100000;

    // Host input arrays
    float *h_A;
    float *h_B;
    // Host output array
    float *h_C;
    // Device input arrays
    float *d_A;
    float *d_B;
    //  Device output array
    float *d_C;

    // Size, in bytes, of each array
    size_t bytes = n*sizeof(float);
```

```
    // Allocate host arrays
    h_A = (float*)malloc(bytes);
    h_B = (float*)malloc(bytes);
    h_C = (float*)malloc(bytes);

    // Allocate device arrays
    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_B, bytes);
    cudaMalloc(&d_C, bytes);

    // Initialize content of input array
    int i;
    for(i=0; i<n; i++) {
        h_A[i] = sinf(i)*sinf(i);
        h_B[i] = cosf(i)*cosf(i);
    }

    // Copy input arrays to device

    // Setup and launch kernel

    // Copy output array to host

    // release device memory

    // Release memory
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

VecAdd.cuf

```
module kernel
    contains
    ! Kernel definition
    attributes(global) subroutine vecAdd(n, A, B, C)
      integer, value :: n
      real, device :: A(n), B(n), C(n)
      integer :: id
      id =  (blockidx%x-1)*(blockdim%x) + threadidx%x

      if(id <= n) then
          C(id) = A(id) + B(id)
      endif
    end subroutine vecAdd
end module kernel

program main
    use cudafor
    use kernel

    ! Size of vectors
    integer :: n = 100000

    ! Input host arrays
    real,dimension(:),allocatable :: h_a
    real,dimension(:),allocatable :: h_b
    ! Output array
    real,dimension(:),allocatable :: h_c

    ! Input device arrays
    real, dimension(:),allocatable,device :: d_a
    real, dimension(:),allocatable,device :: d_b
    ! Output device array
    real, dimension(:),allocatable,device :: d_c

    integer :: i
    real :: sum
```

```
! Allocate host arrays
allocate(h_a(n))
allocate(h_b(n))
allocate(h_c(n))

! Allocate device array
allocate(d_a(n))
allocate(d_b(n))
allocate(d_c(n))

! Initialize content of input vectors
 do i=1,n
     h_a(i) = sin(i*1.0)*sin(i*1.0)
     h_b(i) = cos(i*1.0)*cos(i*1.0)
 enddo

 ! Copy input arrays to device

 ! Setup and launch kernel

 ! Copy output array to host

 ! Release device memory

 ! Release memory
 deallocate(h_a)
 deallocate(h_b)
 deallocate(h_c)

end program
```

Copy input arrays to device

```
// Copy host vectors to device
cudaMemcpy( d_A, h_A, bytes, cudaMemcpyHostToDevice );
cudaMemcpy( d_B, h_B, bytes, cudaMemcpyHostToDevice );
```

Copy input arrays to device

```
! Copy host vectors to device
d_A = h_A
d_B = h_B
```

VecAdd.cu

```
#include <stdlib.h>
#include <math.h>

// CUDA kernel
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
  // Get our global thread ID
  int id = blockIdx.x * blockDim.x + threadIdx.x;

  // Vector addition
  If (id < n)
    c[id] = a[id] + b[id];
}

int main( int argc, char *argv[] )
{
    // Length of array
    int n = 100000;

    // Device input arrays
    float *d_A;
    float *d_B;
    //  Device output array
    float *d_C;
    // Host input arrays
    float *h_A;
    float *h_B;
    // Host output array
    float *h_C;

    // Size, in bytes, of each array
    size_t bytes = n*sizeof(float);
```

```
    // Allocate host arrays
    h_A = (float*)malloc(bytes);
    h_B = (float*)malloc(bytes);
    h_C = (float*)malloc(bytes);

    // Allocate device arrays
    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_B, bytes);
    cudaMalloc(&d_C, bytes);

    // Initialize content of input array
    int i;
    for(i=0; i<n; i++) {
        h_A[i] = sinf(i)*sinf(i);
        h_B[i] = cosf(i)*cosf(i);
    }

     // Copy host vectors to device
     cudaMemcpy( d_A, h_A, bytes, cudaMemcpyHostToDevice );
     cudaMemcpy( d_B, h_B, bytes, cudaMemcpyHostToDevice );

    // Setup and launch kernel

    // Copy output array to host

    // Release device memory

    // Release memory
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

VecAdd.cuf

```
module kernel
    contains
    ! Kernel definition
    attributes(global) subroutine vecAdd(n, A, B, C)
      integer, value :: n
      real, device :: A(n), B(n), C(n)
      integer :: id
      id =  (blockidx%x-1)*(blockdim%x) + threadidx%x

      if(id <= n) then
          C(id) = A(id) + B(id)
      endif
    end subroutine vecAdd
end module kernel

program main
    use cudafor
    use kernel

    ! Size of vectors
    integer :: n = 100000

    ! Input host arrays
    real,dimension(:),allocatable :: h_a
    real,dimension(:),allocatable :: h_b
    ! Output array
    real,dimension(:),allocatable :: h_c

    ! Input device arrays
    real, dimension(:),allocatable,device :: d_a
    real, dimension(:),allocatable,device :: d_b
    ! Output device array
    real, dimension(:),allocatable,device :: d_c

    integer :: i
    real :: sum
```

```
    ! Allocate host arrays
    allocate(h_a(n))
    allocate(h_b(n))
    allocate(h_c(n))

    ! Allocate device array
    allocate(d_a(n))
    allocate(d_b(n))
    allocate(d_c(n))

    ! Initialize content of input vectors
    do i=1,n
        h_a(i) = sin(i*1.0)*sin(i*1.0)
        h_b(i) = cos(i*1.0)*cos(i*1.0)
    enddo

    ! Copy host vectors to device
    d_A = h_A
    d_B = h_B

    ! Setup and launch kernel

    ! Copy output array to host

    ! Release device memory

    ! Release memory
    deallocate(h_a)
    deallocate(h_b)
    deallocate(h_c)

end program
```

Setup and Launch Kernel

```
// Number of threads in each thread block
dim3 blockSize(256,1,1);

// Number of thread blocks in grid
int blocks = (int)ceil((float)n/blockSize.x);
dim3 gridSize(blocks,1,1);

// Execute the kernel
vecAdd<<<gridSize, blockSize>>>(d_A, d_B, d_C, n);
```

Setup and Launch Kernel

```
type(dim3) :: blockSize, gridSize

! Number of threads in each thread block
blockSize = dim3(256,1,1)

! Number of thread blocks in grid
gridSize = dim3(ceiling(real(n)/real(blockSize%x)) ,1,1)

! Execute the kernel
call vecAdd<<<gridSize, blockSize>>>(n,  d_A, d_B, d_C)
```

VecAdd.cu

```
#include <stdlib.h>
#include <math.h>

// CUDA kernel
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
  // Get our global thread ID
  int id = blockIdx.x * blockDim.x + threadIdx.x;

  // Vector addition
  if (id < n)
    c[id] = a[id] + b[id];
}

int main( int argc, char *argv[] )
{
    // Length of array
    int n = 100000;

    // Device input arrays
    float *d_A;
    float *d_B;
    //  Device output array
    float *d_C;
    // Host input arrays
    float *h_A;
    float *h_B;
    // Host output array
    float *h_C;

    // Size, in bytes, of each array
    size_t bytes = n*sizeof(float);

    // Allocate host arrays
    h_A = (float*)malloc(bytes);
    h_B = (float*)malloc(bytes);
    h_C = (float*)malloc(bytes);
```

```
    // Allocate device arrays
    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_B, bytes);
    cudaMalloc(&d_C, bytes);

    // Initialize content of input array
    int i;
    for(i=0; i<n; i++) {
        h_A[i] = sinf(i)*sinf(i);
        h_B[i] = cosf(i)*cosf(i);
    }

     // Copy host vectors to device
     cudaMemcpy( d_A, h_A, bytes,
                 cudaMemcpyHostToDevice );
     cudaMemcpy( d_B, h_B, bytes,
                 cudaMemcpyHostToDevice );

      // Number of threads in each thread block
    dim3 blockSize(1024,1,1);

    // Number of thread blocks in grid
    int blocks = (int)ceil((float)n/blockSize.x);
    dim3 gridSize(blocks,1,1);

    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(d_A, d_B, d_C, n);

    // Copy output array to host

    // release device memory

    // Release memory
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

VecAdd.cuf

```
module kernel
    contains
    ! Kernel definition
    attributes(global) subroutine vecAdd(n, A, B, C)
      integer, value :: n
      real, device :: A(n), B(n), C(n)
      integer :: id
      id =  (blockidx%x-1)*(blockdim%x) + threadidx%x

      if(id <= n) then
          C(id) = A(id) + B(id)
      endif
    end subroutine vecAdd
end module kernel

program main
    use cudafor
    use kernel

    ! Size of vectors
    integer :: n = 100000

    ! Input host arrays
    real,dimension(:),allocatable :: h_a
    real,dimension(:),allocatable :: h_b
    ! Output array
    real,dimension(:),allocatable :: h_c

    ! Input device arrays
    real, dimension(:),allocatable,device :: d_a
    real, dimension(:),allocatable,device :: d_b
    ! Output device array
    real, dimension(:),allocatable,device :: d_c

    integer :: i
    real :: sum
    type(dim3) :: blockSize, gridSize
```

```
    ! Allocate host arrays
    allocate(h_a(n))
    allocate(h_b(n))
    allocate(h_c(n))

    ! Allocate device array
    allocate(d_a(n))
    allocate(d_b(n))
    allocate(d_c(n))

    ! Initialize content of input vectors
    do i=1,n
        h_a(i) = sin(i*1.0)*sin(i*1.0)
        h_b(i) = cos(i*1.0)*cos(i*1.0)
    enddo

    ! Copy host vectors to device
    d_A = h_A
    d_B = h_B

    ! Number of threads in each thread block
    blockSize = dim3(256,1,1)

    ! Number of thread blocks in grid
    gridSize = dim3(ceiling(real(n)/real(blockSize%x)) ,1,1)

    ! Execute the kernel
    call vecAdd<<<gridSize, blockSize>>>(n,  d_A, d_B, d_C)

    ! Copy output array to host

    ! Release device memory

    ! Release memory
    deallocate(h_a)
    deallocate(h_b)
    deallocate(h_c)

end program
```

Copy Output Array to Host

```
// Copy device vector to host
cudaMemcpy( h_C, d_C, bytes, cudaMemcpyDeviceToHost );
```

Copy input arrays to device

```
! Copy device vector to host
h_C = d_C
```

VecAdd.cu

```
#include <stdlib.h>
#include <math.h>

// CUDA kernel
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
  // Get our global thread ID
  int id = blockIdx.x * blockDim.x + threadIdx.x;

  //Vector addition
  if (id < n)
    c[id] = a[id] + b[id];
}

int main( int argc, char *argv[] )
{
    // Length of array
    int n = 100000;

    // Device input arrays
    float *d_A;
    float *d_B;
    //  Device output array
    float *d_C;
    // Host input arrays
    float *h_A;
    float *h_B;
    // Host output array
    float *h_C;

    // Size, in bytes, of each array
    size_t bytes = n*sizeof(float);

    // Allocate host arrays
    h_A = (float*)malloc(bytes);
    h_B = (float*)malloc(bytes);
    h_C = (float*)malloc(bytes);
```

```
    // Allocate device arrays
    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_B, bytes);
    cudaMalloc(&d_C, bytes);

    // Initialize content of input array
    int i;
    for(i=0; i<n; i++) {
        h_A[i] = sinf(i)*sinf(i);
        h_B[i] = cosf(i)*cosf(i);
    }

    // Copy host vectors to device
    cudaMemcpy( d_A, h_A, bytes, cudaMemcpyHostToDevice );
    cudaMemcpy( d_B, h_B, bytes, cudaMemcpyHostToDevice );

    // Number of threads in each thread block
    dim3 blockSize(1024,1,1);

    // Number of thread blocks in grid
    int grid = (int)ceil((float)n/blockSize.x);
    dim3 gridSize(grid,1,1);

    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(d_A, d_B, d_C, n);

    // Copy device vector to host
    cudaMemcpy( h_C, d_C, bytes, cudaMemcpyDeviceToHost );

    // release device memory

    // Release memory
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

VecAdd.cuf

```
module kernel
    contains
    ! Kernel definition
    attributes(global) subroutine vecAdd(n, A, B, C)
      integer, value :: n
      real, device :: A(n), B(n), C(n)
      integer :: id
      id =  (blockidx%x-1)*(blockdim%x) + threadidx%x

      if(id <= n) then
          C(id) = A(id) + B(id)
      endif
    end subroutine vecAdd
end module kernel

program main
    use cudafor
    use kernel

    ! Size of vectors
    integer :: n = 100000

    ! Input host arrays
    real,dimension(:),allocatable :: h_a
    real,dimension(:),allocatable :: h_b
    ! Output array
    real,dimension(:),allocatable :: h_c

    ! Input device arrays
    real, dimension(:),allocatable,device :: d_a
    real, dimension(:),allocatable,device :: d_b
    ! Output device array
    real, dimension(:),allocatable,device :: d_c

    integer :: i
    real :: sum
    type(dim3) :: blockSize, gridSize
```

```
    ! Allocate host arrays
    allocate(h_a(n))
    allocate(h_b(n))
    allocate(h_c(n))

    ! Allocate device array
    allocate(d_a(n))
    allocate(d_b(n))
    allocate(d_c(n))

    ! Initialize content of input vectors
    do i=1,n
        h_a(i) = sin(i*1.0)*sin(i*1.0)
        h_b(i) = cos(i*1.0)*cos(i*1.0)
    enddo

    ! Copy host vectors to device
    d_A = h_A
    d_B = h_B

    ! Number of threads in each thread block
    blockSize = dim3(256,1,1);

    ! Number of thread blocks in grid
    gridSize = dim3(ceiling(real(n)/real(blockSize%x)) ,1,1)

    ! Execute the kernel
    call vecAdd<<<gridSize, blockSize>>>(n,  d_A, d_B, d_C)

    ! Copy device vector to host
    h_c = d_c

    ! Release device memory

    ! Release memory
    deallocate(h_a)
    deallocate(h_b)
    deallocate(h_c)

end program
```

Release Device Memory

```
// Release device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Release Device Memory

```
! Release device memory
deallocate(h_a)
deallocate(h_b)
deallocate(h_c)
```

VecAdd.cu

```
#include <stdlib.h>
#include <math.h>

// CUDA kernel
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
  // Get our global thread ID
  int id = blockIdx.x * blockDim.x + threadIdx.x;

  //Vector addition
  if (id < n)
    c[id] = a[id] + b[id];
}

int main( int argc, char *argv[] )
{
    // Length of array
    int n = 100000;

    // Device input arrays
    float *d_A;
    float *d_B;
    //  Device output array
    float *d_C;
    // Host input arrays
    float *h_A;
    float *h_B;
    // Host output array
    float *h_C;

    // Size, in bytes, of each array
    size_t bytes = n*sizeof(float);

    // Allocate host arrays
    h_A = (float*)malloc(bytes);
    h_B = (float*)malloc(bytes);
    h_C = (float*)malloc(bytes);
```

```
    // Allocate device arrays
    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_B, bytes);
    cudaMalloc(&d_C, bytes);

    // Initialize content of input array
    int i;
    for(i=0; i<n; i++) {
        h_A[i] = sinf(i)*sinf(i);
        h_B[i] = cosf(i)*cosf(i);
    }

    // Copy host vectors to device
    cudaMemcpy( d_A, h_A, bytes, cudaMemcpyHostToDevice );
    cudaMemcpy( d_B, h_B, bytes, cudaMemcpyHostToDevice );

    // Number of threads in each thread block
    dim3 blockSize(1024,1,1);

    // Number of thread blocks in grid
    int grid = (int)ceil((float)n/blockSize.x);
    dim3 gridSize(grid,1,1);

    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(d_A, d_B, d_C, n);


    // Copy device vector to host
    cudaMemcpy( h_C, d_C, bytes, cudaMemcpyDeviceToHost );

    // Release device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Release host memory
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

VecAdd.cuf

```
module kernel
    contains
    ! Kernel definition
    attributes(global) subroutine vecAdd(n, A, B, C)
      integer, value :: n
      real, device :: A(n), B(n), C(n)
      integer :: id
      id =  (blockidx%x-1)*(blockdim%x) + threadidx%x

      if(id <= n) then
          C(id) = A(id) + B(id)
      endif
    end subroutine vecAdd
end module kernel

program main
    use cudafor
    use kernel

    ! Size of vectors
    integer :: n = 100000

    ! Input host arrays
    real,dimension(:),allocatable :: h_a
    real,dimension(:),allocatable :: h_b
    ! Output array
    real,dimension(:),allocatable :: h_c

    ! Input device arrays
    real, dimension(:),allocatable,device :: d_a
    real, dimension(:),allocatable,device :: d_b
    ! Output device array
    real, dimension(:),allocatable,device :: d_c

    integer :: i
    real :: sum
    type(dim3) :: blockSize, gridSize

    ! Allocate host arrays
    allocate(h_a(n))
    allocate(h_b(n))
    allocate(h_c(n))

    ! Allocate device array
    allocate(d_a(n))
    allocate(d_b(n))
    allocate(d_c(n))

    ! Initialize content of input vectors
    do i=1,n
        h_a(i) = sin(i*1.0)*sin(i*1.0)
        h_b(i) = cos(i*1.0)*cos(i*1.0)
    enddo

    ! Copy host vectors to device
    d_A = h_A
    d_B = h_B

    ! Number of threads in each thread block
    blockSize = dim3(256,1,1);

    ! Number of thread blocks in grid
    gridSize = dim3(ceiling(real(n)/real(blockSize%x)) ,1,1)

    ! Execute the kernel
    call vecAdd<<<gridSize, blockSize>>>(n,  d_A, d_B, d_C)

    ! Copy device vector to host
    h_c = d_c

    ! Release device memory
    deallocate(h_a)
    deallocate(h_b)
    deallocate(h_c)

    ! Release memory
    deallocate(h_a)
    deallocate(h_b)
    deallocate(h_c)

end program
```

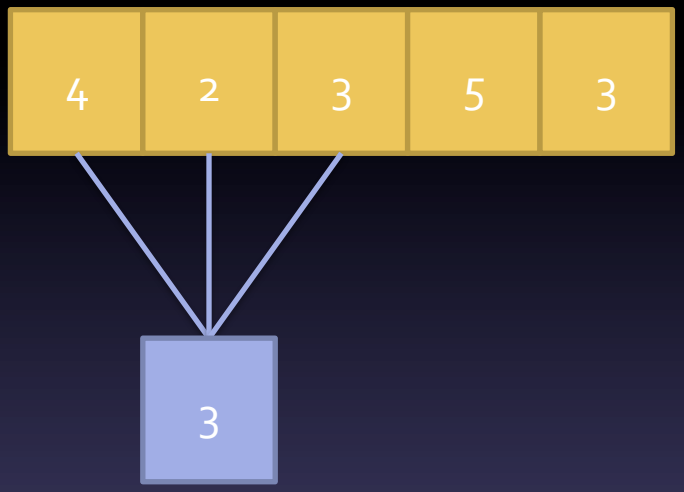
Compiling and Running

```
$ ssh user@lens.ccs.ornl.gov
$ module load cuda
$ qsub –A### -I –V –lwalltime=00:30:00,nodes=1:ppn=1
$ nvcc VecAdd.cu -o add.out
$ ./add.out
```

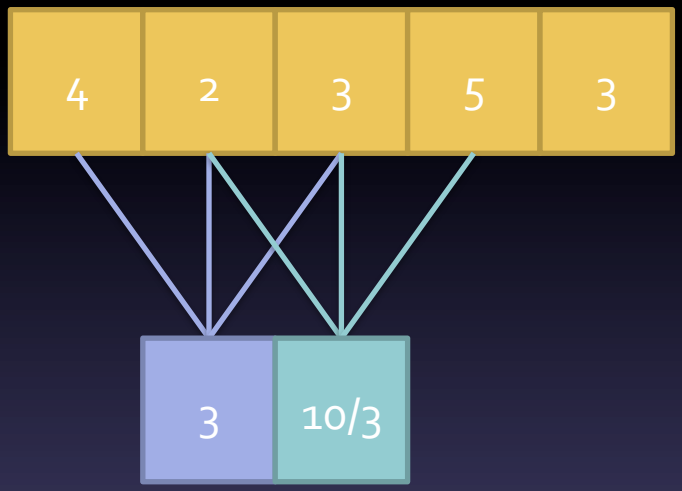
Compiling and Running

```
$ ssh user@lens.ccs.ornl.gov
$ qsub –A### -I –V –lwalltime=00:30:00,nodes=1:ppn=1
$ pgfortran VecAdd.cuf -o add.out
$ ./add.out
```

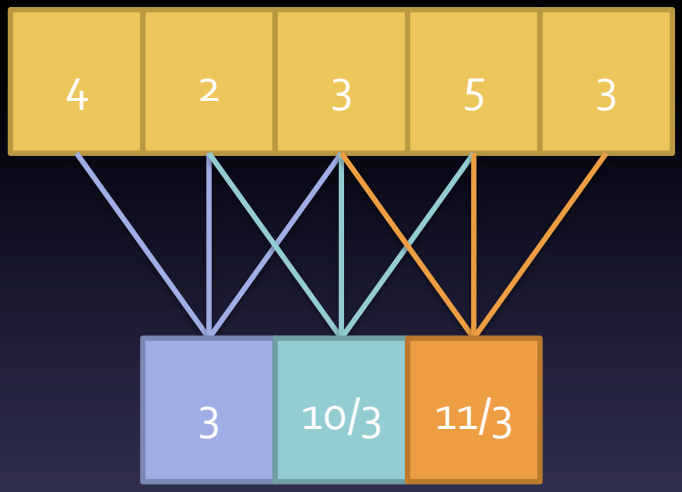
Moving Average



Moving Average



Moving Average

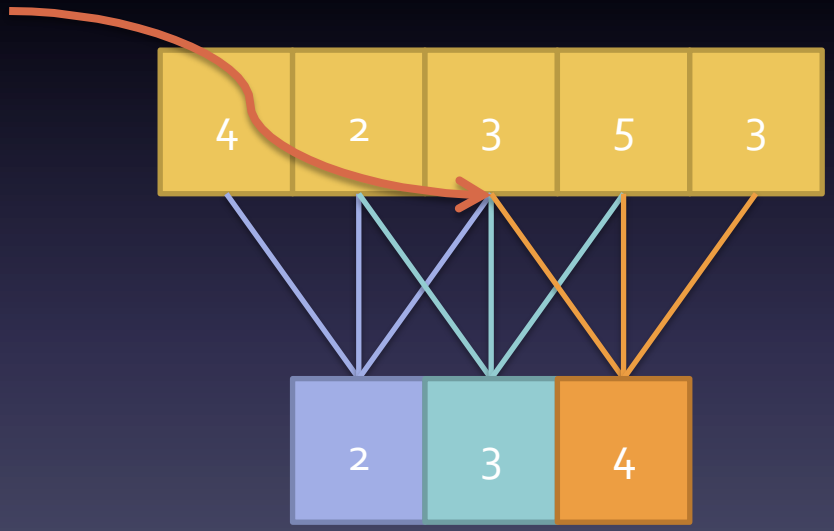


Global Moving Average

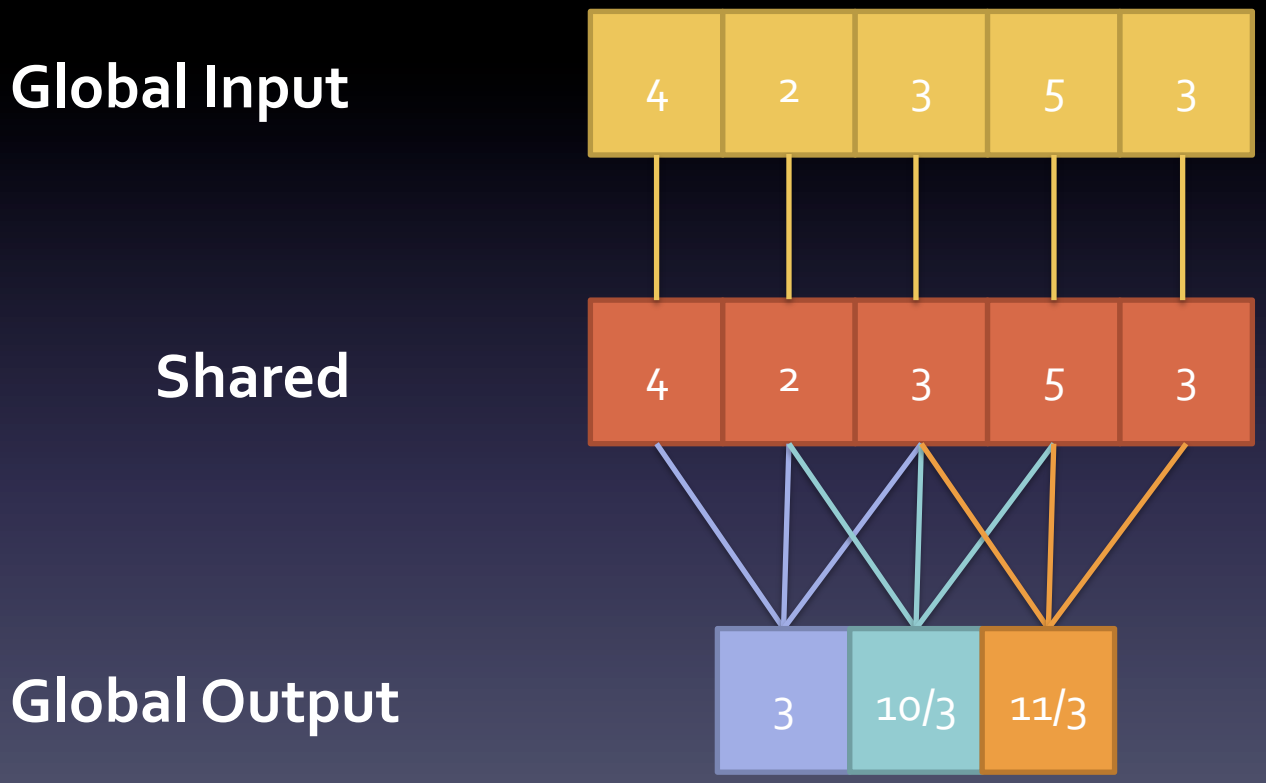
```
__global__ void Smooth(float *in, float *out, int n)
{
    // Get our global and block thread ID
    int id = blockIdx.x * blockDim.x + threadIdx.x;

    // Ignore the first and last threads
    if(id !=0 && id < (n-1)) {
        // Compute one element of out array
        float movAvg = (in[id-1] + in[id] + in[id+1]) / 3.0 ;

        // Write to result back to global memory
        out[id-1] = movAvg;
    }
}
```

Global Moving Average

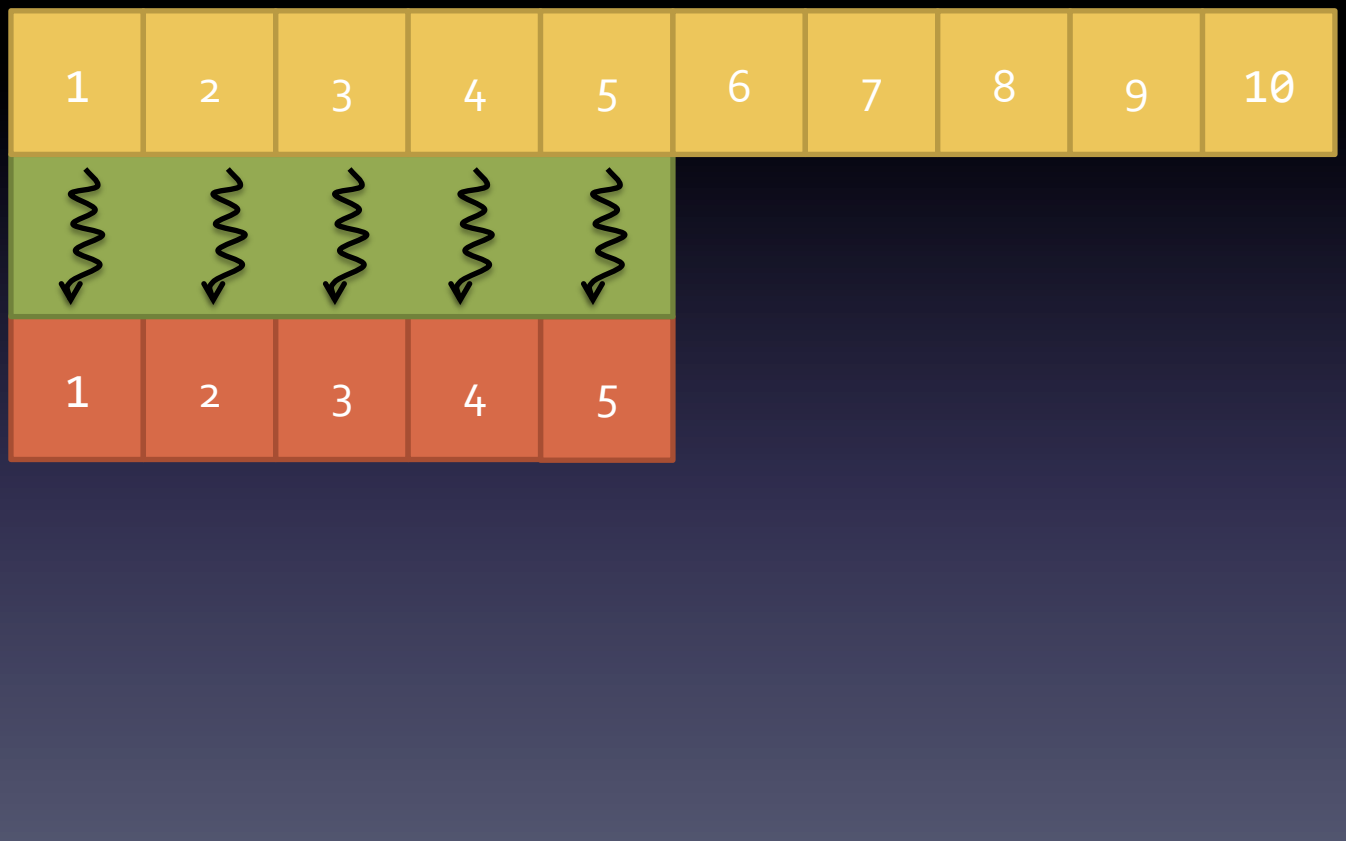
Same value read
from global memory
3 times = BAD



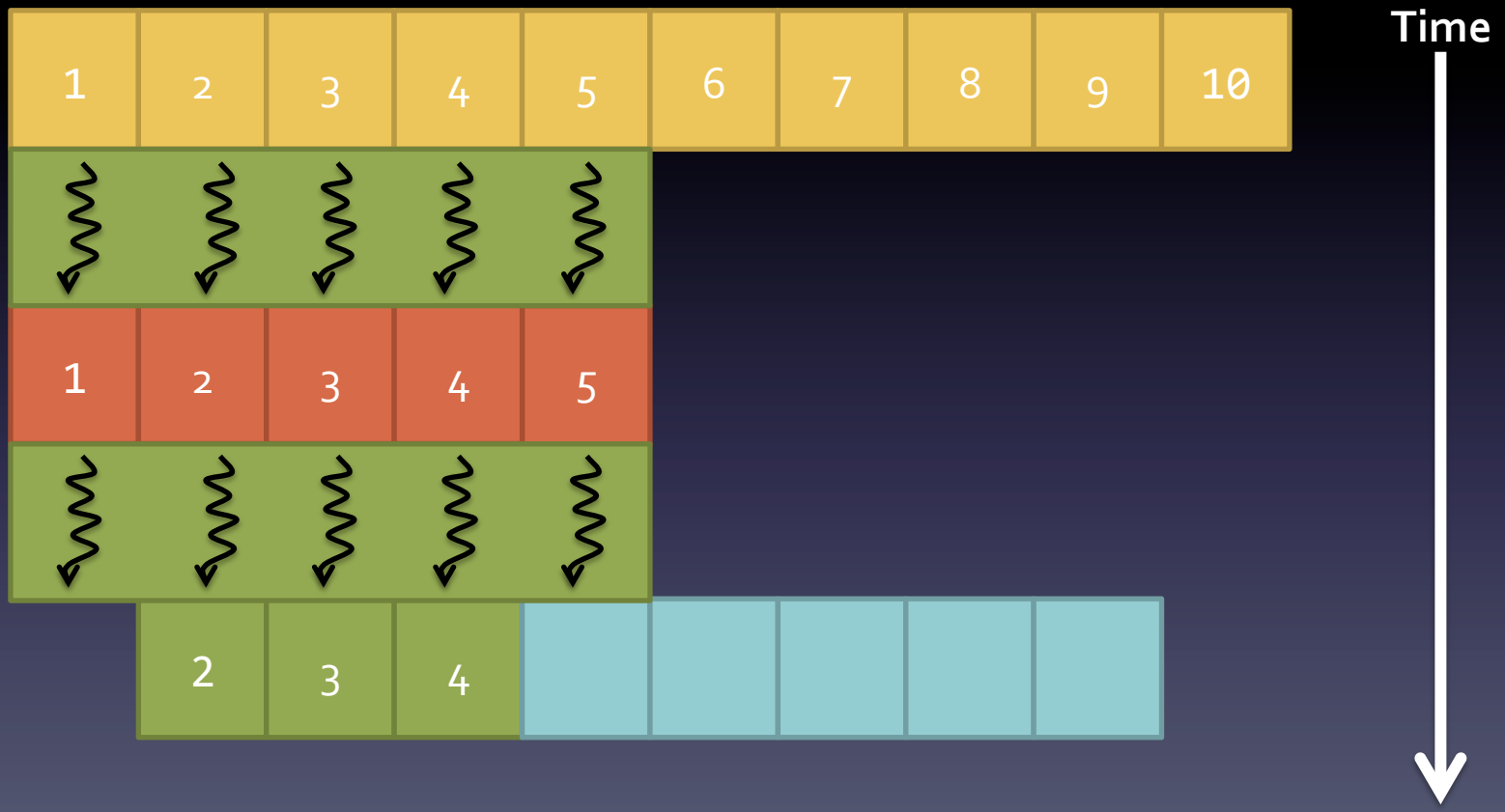
Shared Moving Average



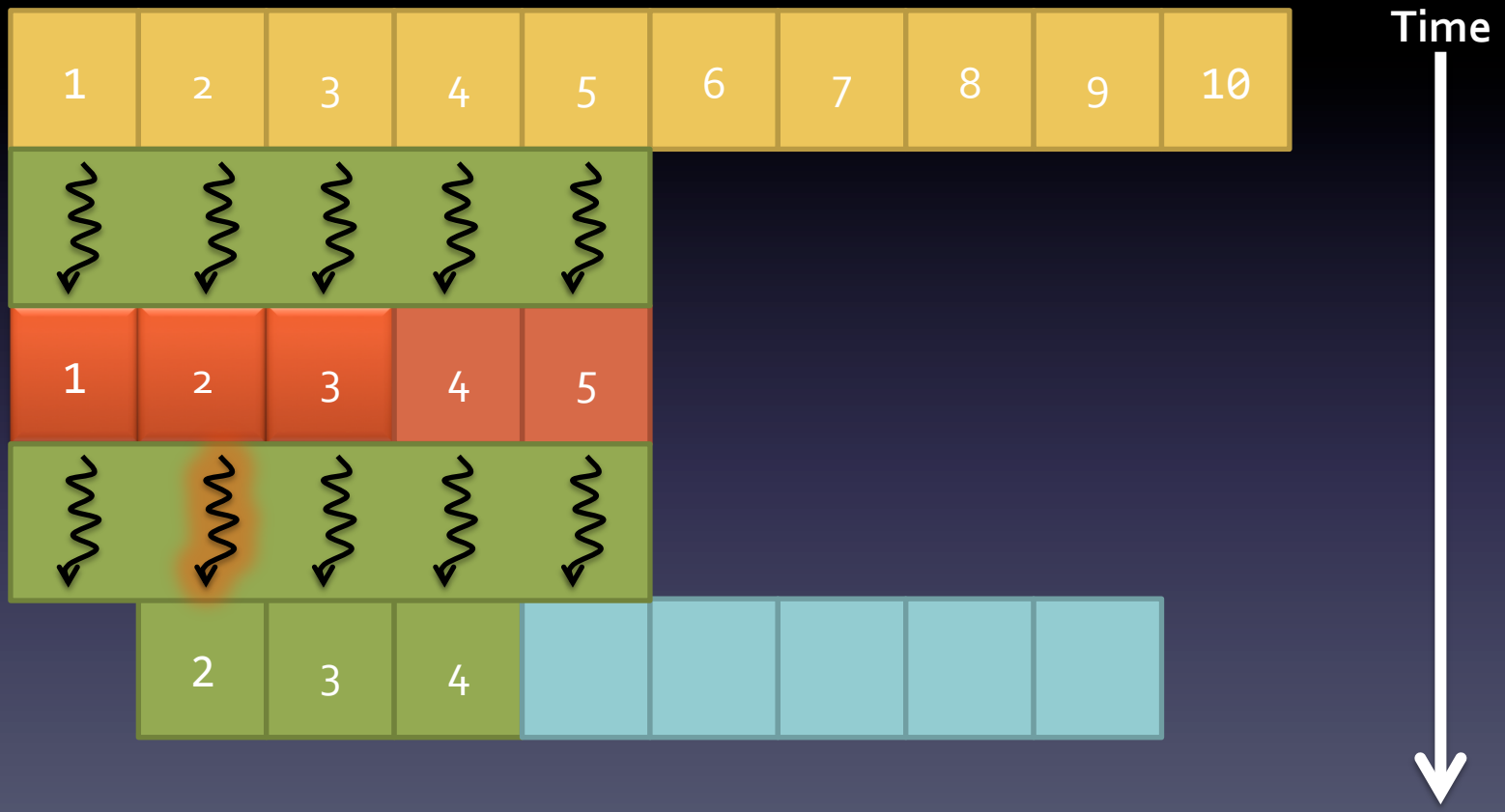
Shared Moving Average



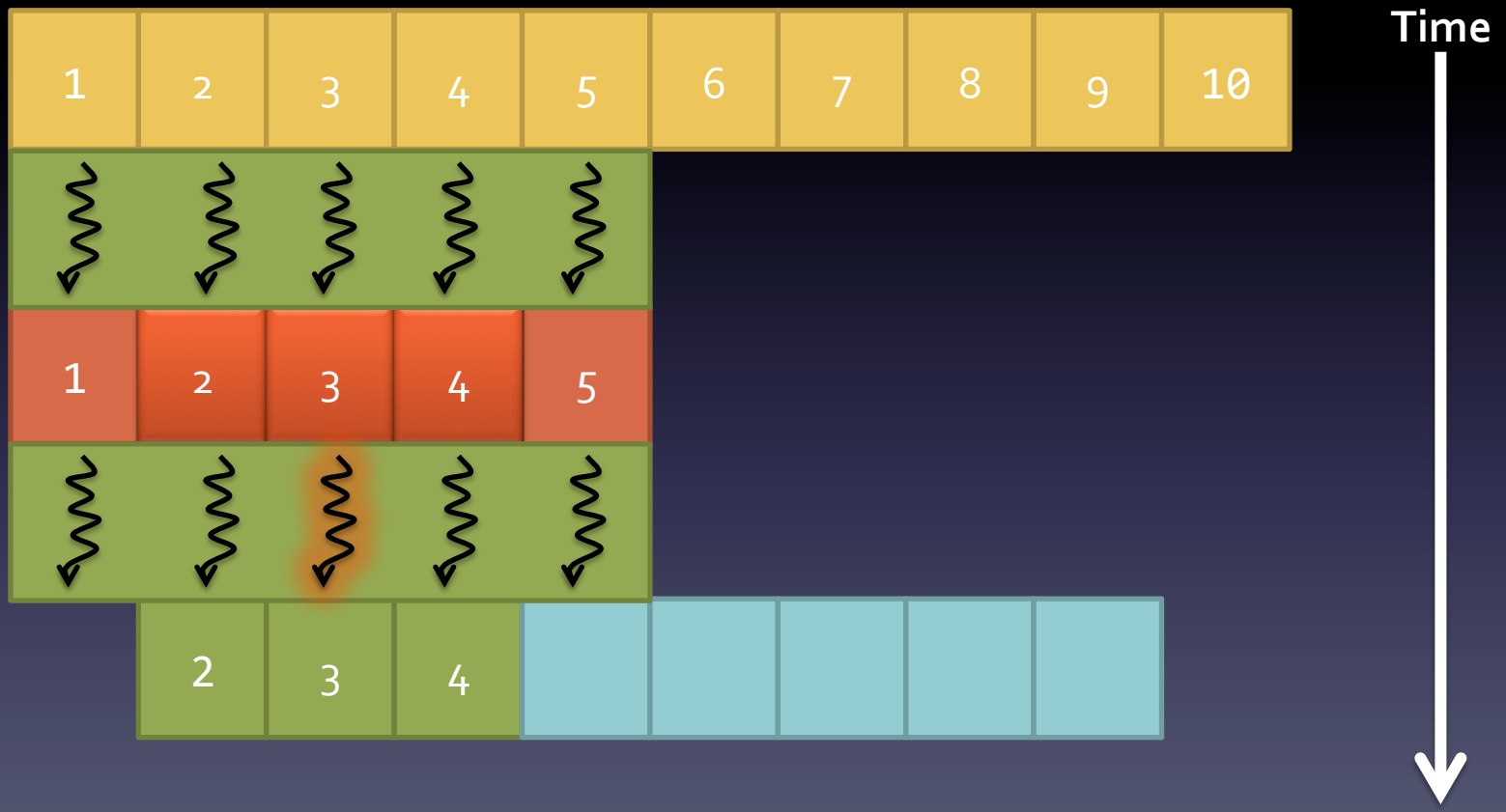
Shared Moving Average



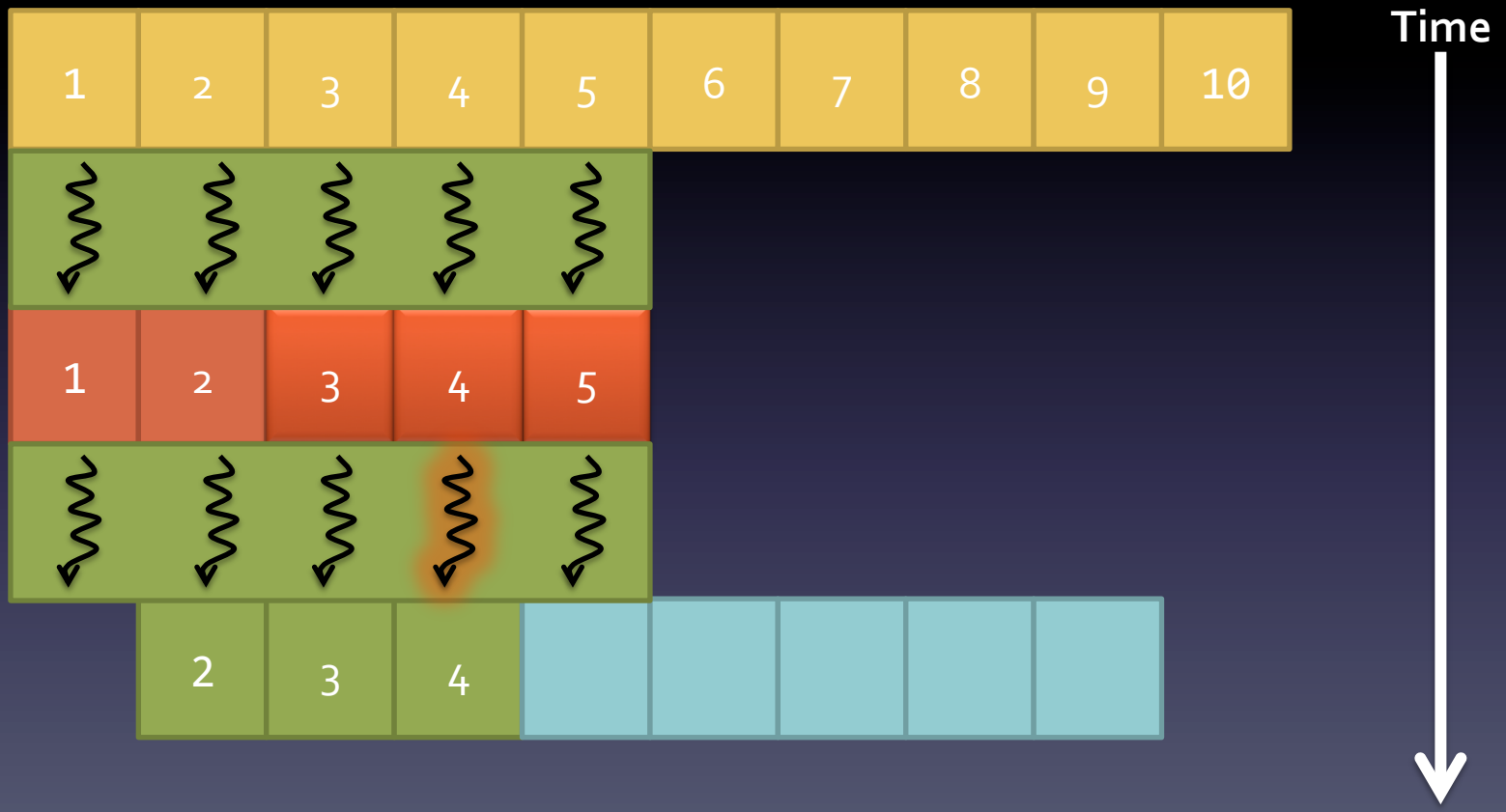
Shared Moving Average



Shared Moving Average

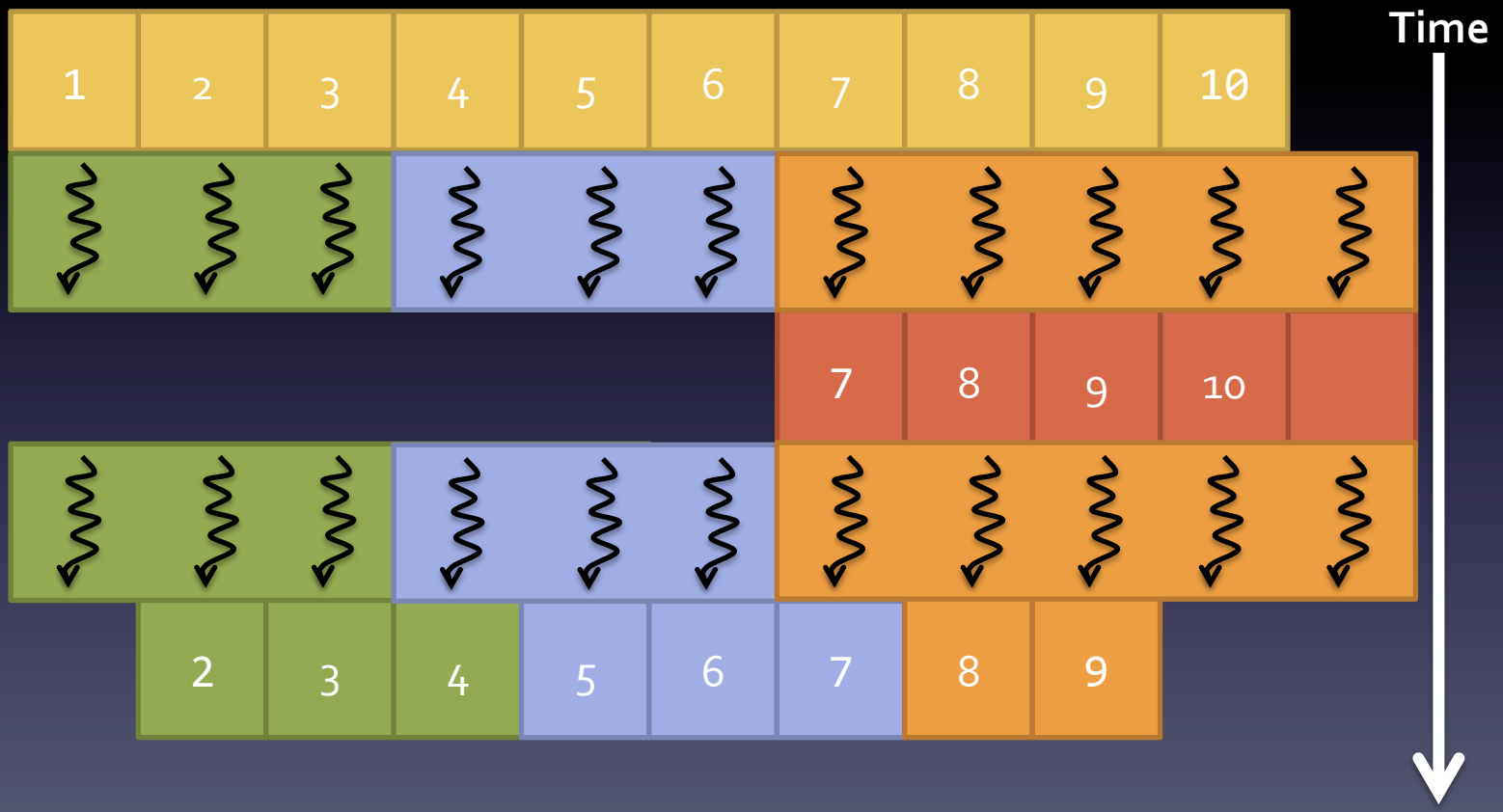


Shared Moving Average



Shared Moving Average

Shared Moving Average



Shared Moving Average

```
#define BLOCK_DIM 512

__global__ void Smooth(float *in, float *out, int n)
{
    //Declare memory per block
    __shared__ float s_mem[BLOCK_DIM];

    // Get our global and block thread ID
    int id = blockIdx.x * (blockDim.x-2) + threadIdx.x;
    int i  = threadIdx.x;

    if(id < n)
    {
        // Read from global into shared
        s_mem[i] = in[id];

        // Wait for all threads in a block to reach this point
        __synchthreads();

        if(i !=0 && i < (BLOCK_DIM-1) && id < (n-1)) {
            // Use shared memory
            float movAvg = (s_mem[i-1] + s_mem[i] + s_mem[i+1]) / 3.0 ;

            // Write to result back to global memory
            out[id-1] = movAvg;
        }
    }
}
```