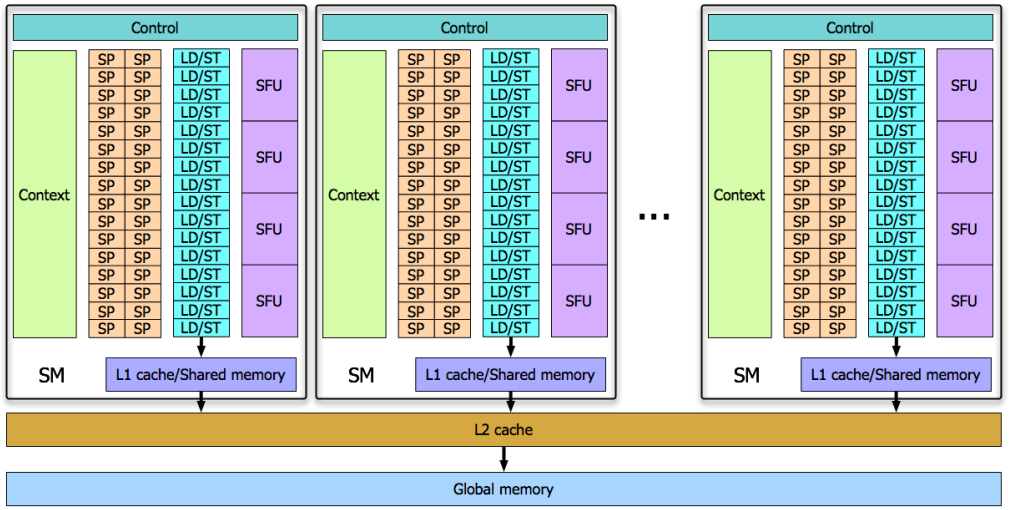
Lecture 21 – Shared Memory

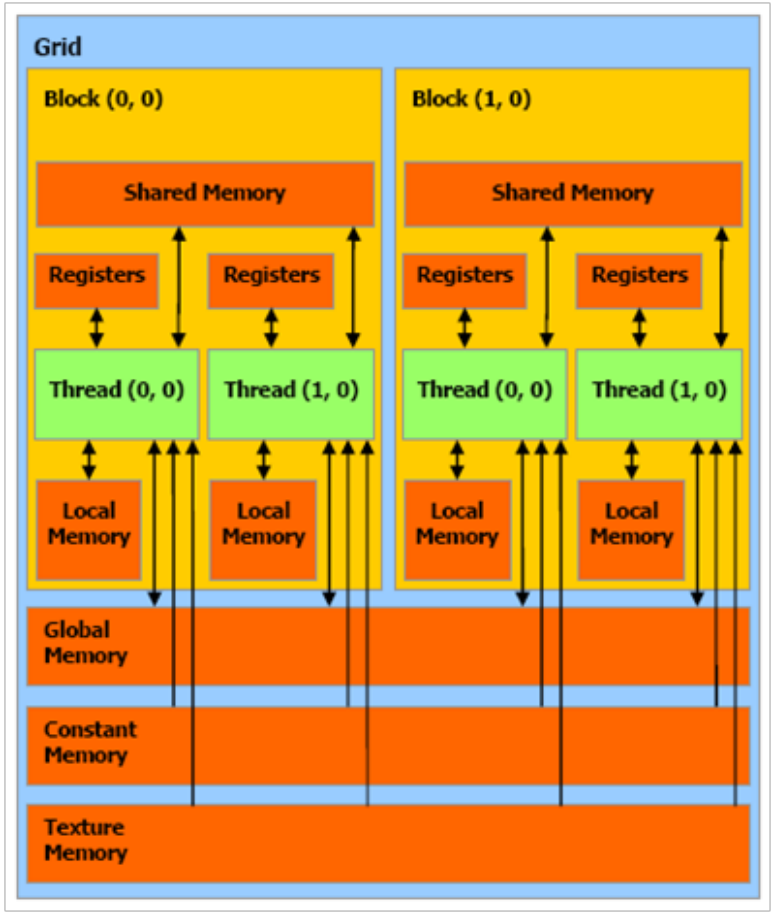
Jaejin Lee
Dept. of Computer Science and Engineering, College of Engineering
Dept. of Data Science, Graduate School of Data Science
Seoul National University
http://aces.snu.ac.kr

CHAMP
SNU THUNDER Research Group

CUDA Memory Hierarchy

Type	Read/write	Speed
Global memory	Read/write	slow, but cached
Texture memory	read only	cache optimized for 2D/3D access pattern
Constant memory	read only	where constants and kernel arguments are stored
Shared memory	read/write	fast
Local memory	read/write	used when it does not fit in to registers, just a part of global memory, slow but cached
Registers	read/write	fast





Access Speed

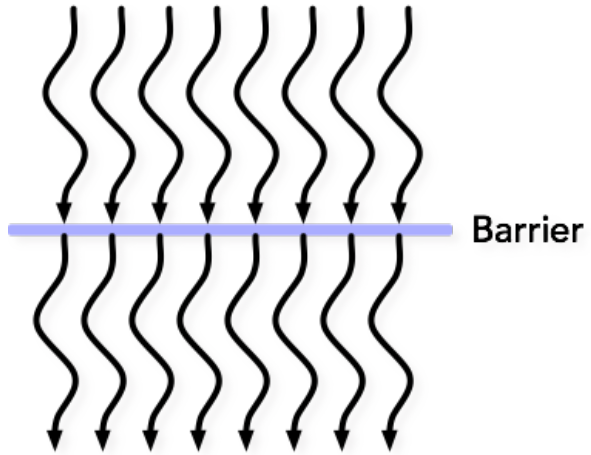
Declaration	Memory	Scope	Lifetime
int v	register	thread	thread
int v[10]	local	thread	thread
__shared__ int v	shared	block	block
__device__ int v	global	grid	application
__constant__ int v	constant	grid	application

Shared Memory

- On-chip
  - User-managed data caches (scratchpad memory)
- Much faster than local and global memory
  - Shared memory latency is roughly $100\times$ lower than uncached global memory latency
  - Threads can access data in shared memory loaded from global memory by other threads within the same thread block
- Memory access can be controlled by thread synchronization to avoid race condition
  - __syncthreads()

Barriers in CUDA

- __syncthreads() in CUDA
  - A block level synchronization barrier
  - Waits until all threads in the thread block have reached this point, and all global and shared memory accesses made by these threads prior to __syncthreads() are visible to all threads in the block



CUDA Memory Consistency Model

- CUDA adopts a relaxed memory consistency model to enable more aggressive compiler optimizations
- To explicitly force a certain ordering for program correctness, memory fences and barriers must be inserted
  - The only way to guarantee the correct behavior of a kernel
- Barriers
  - void __syncthreads()

CUDA Memory Consistency Model (cont'd)

- Memory fences
  - Ensure that any memory write before the fence is visible to other threads after the fence
  - Do not perform any thread synchronization
    - It is not necessary for all threads in a block to actually execute the fence
  - There are three variants of memory fences depending on the desired scope: block, grid, or system
- Memory fence functions only affect the ordering of memory operations by a thread
  - They do not, by themselves, ensure that these memory operations are visible to other threads
    - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#memory-fence-functions

CUDA Memory Consistency Model (cont'd)

- **`void __threadfence_block();`**
  - Ensures that all writes to shared memory and global memory made by a calling thread before the fence are visible to other threads in the same block after the fence
- **`void __threadfence();`**
  - Stalls the calling thread until all of its writes to global memory are visible to all threads in the same grid
- **`void __threadfence_system();`**
  - Stalls the calling thread to ensure all its writes to global memory, page-locked host memory, and the memory of other devices are visible to all threads in all devices and host threads

Using Shared Memory

- On devices of compute capability 2.x and 3.x, each SM has 64KB of on-chip memory
  - Can be partitioned between L1 cache and shared memory
- For devices of compute capability 2.x, there are two settings:
  - 48KB shared memory and 16KB L1 cache, (default)
  - 16KB shared memory and 48KB L1 cache
- Can be configured at runtime
  - API from the host for all kernels using **`cudaDeviceSetCacheConfig()`**
  - per-kernel basis using **`cudaFuncSetCacheConfig()`**

Using Shared Memory (cont'd)

```
__global__ void reverse(int *d, int n) {
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}

int main(void) {
    const int n = 64;
    int a[n], r[n], d[n];
    for (int i = 0; i < n; i++) {
        a[i] = i; r[i] = n-i-1; d[i] = 0;
    }
    int *d_d; cudaMalloc(&d_d, n * sizeof(int));
    cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
    reverse<<<1,n>>>(d_d, n);
    cudaMemcpy(d, d_d, n*sizeof(int), cudaMemcpyDeviceToHost);
    for (int i = 0; i < n; i++)
       if (d[i] != r[i])
          printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);
}
```

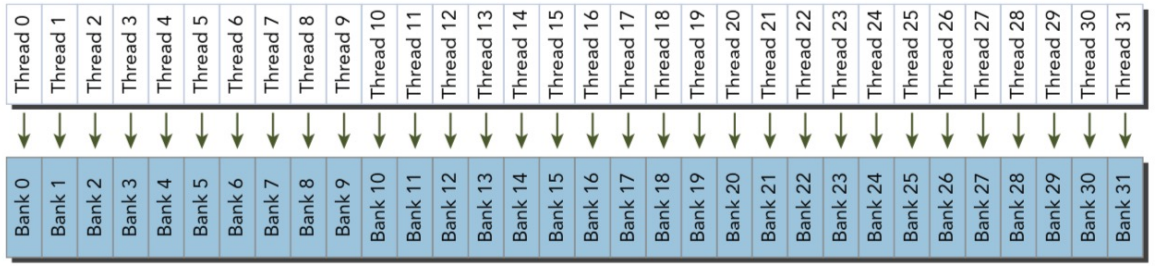
Using Shared Memory (cont'd)

```
__global__ void reverse(int *d, int n) {
    // Dynamic shared memory
    extern __shared__ int s[];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
int main(void) {
    const int n = 64;
    int a[n], r[n], d[n];
    for (int i = 0; i < n; i++) {
        a[i] = i; r[i] = n-i-1; d[i] = 0;
    }
    int *d_d; cudaMalloc(&d_d, n * sizeof(int));
    cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
    reverse<<<1,n,n*sizeof(int)>>>(d_d, n);
    cudaMemcpy(d, d_d, n*sizeof(int), cudaMemcpyDeviceToHost);
    for (int i = 0; i < n; i++)
       if (d[i] != r[i])
          printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);
}
```

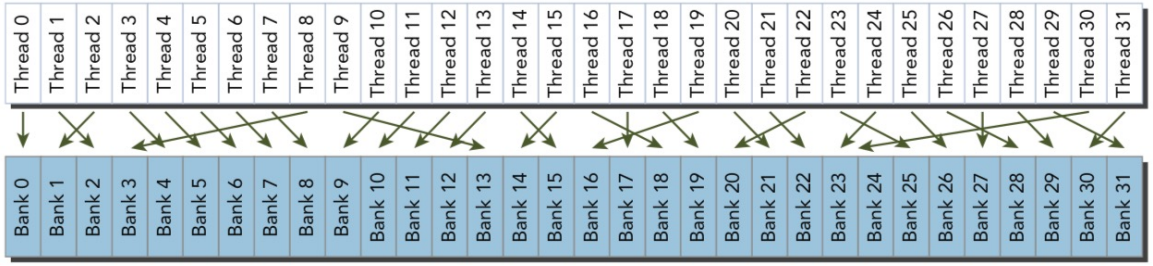
Bank Conflicts in Shared Memory

- To achieve high memory bandwidth, shared memory is divided into 32 equally-sized memory modules, called banks
  - Can be accessed simultaneously
  - There are 32 banks because there are 32 threads in a warp
  - Depending on the compute capability of a GPU, the addresses of shared memory are mapped to different banks in different patterns
- If a shared memory load or store operation issued by a warp does not access more than one memory location per bank, the operation can be serviced by one memory transaction
- Otherwise, the operation is serviced by multiple memory transactions

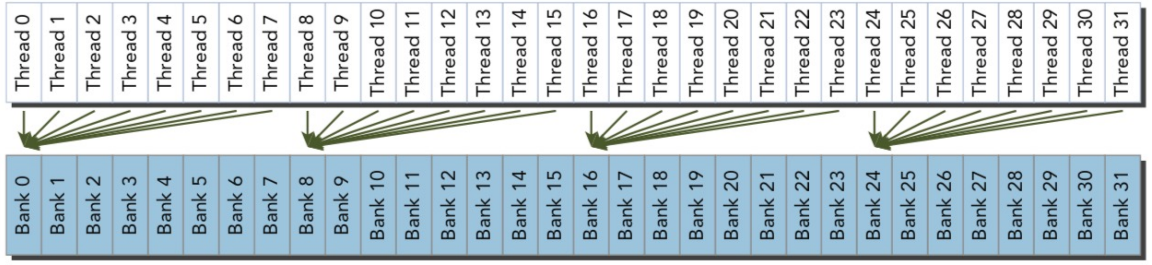
Bank Conflicts in Shared Memory (cont'd)



No conflict



No conflict



- No conflict (conflict-free broadcast access if threads access the same address within a bank)
- Bank conflicts (if threads access different addresses within a bank)

Using OpenCL Local Memory

- Local memory is local to a work-group
  - Shared by all work-items of work-group
  - Used to cache global memory
  - Low latency access

- Two ways to allocate it
  - Statically, inside the kernel
  - Dynamically, from the host as a parameter

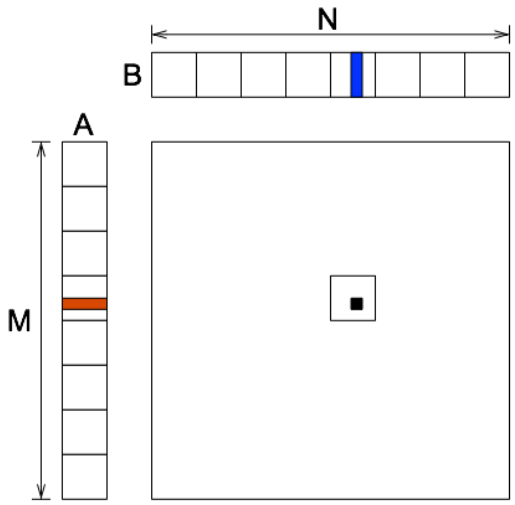
Allocating OpenCL Local Memory

- Inside a kernel
  - Declare local memory as a static array
  - Use the keyword **`__local`**

- Kernel with parameter in local memory
  - Allocation done during set of kernel arguments

Matrix Multiplication Using OpenCL Local Memory

- Every work-item takes care of one element in C




```
__kernel void MatMul(__global float* a,
                     __global float* b,
                     __global float* c,
                     int N)
{
    int row = get_global_id(1);
    int col= get_global_id(0);
    float sum = 0.0f;
    for (int i= 0; i < T_SIZE; i++) {
        sum += a[row*T_SIZE+i]
                * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

Matrix Multiplication Using OpenCL Local Memory (cont'd)

```
__kernel void TMatMul(__global float* a,
                      __global float* b,
                      __global float* c,intN,
                      __local float tile[T_SIZE][T_SIZE])
{
    int row = get_global_id(1);
    int col= get_global_id(0);
    float sum = 0.0f;
    int x = get_local_id(0);
    int y = get_local_id(1);

    tile[y][x] = a[row*T_SIZE+x];
    for (int i= 0; i< T_SIZE; i++) {
        sum += tile[y][i]* b[i*N+col];
    }
    c[row*N+col] = sum;
}
```