Variable Qualifiers (GPU code)

- **__device__**
  - Stored in global memory (large, high latency, no cache)
  - Allocated with `cudaMalloc` (**device** qualifier implied)
  - Accessible by all threads
  - Lifetime: application

- **__shared__**
  - Stored in on-chip shared memory (very low latency)
  - Specified by execution configuration or at compile time
  - Accessible by all threads in the same thread block
  - Lifetime: thread block

**Unqualified variables:**
- Scalars and built-in vector types are stored in registers
- Arrays may be in registers or local memory

---

Coalesced access to global memory

- Global memory access of 32, 64, or 128-bit words by a half-warp of threads can result in as few as one or two transactions if certain conditions are met

**In Compute capability 1.0 and 1.1**

- K-th thread must access k-th word in the segment (or k-th word in 2 contiguous 128B segments for 128-bit words), not all threads need to participate

### Coalesces – 1 transaction

<table>
<thead>
<tr>
<th>128B aligned segment (32 floats)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64B aligned segment (16 floats)</td>
</tr>
</tbody>
</table>

### Out of sequence – 16 transactions

### Misaligned – 16 transactions
**Coalescing**

In Compute capability 1.2 and higher

- Coalescing is achieved for any pattern of addresses that fits into a segment of size: 32B for 8-bit words, 64B for 16-bit words, 128B for 32- and 64-bit words
- Smaller transactions may be issued to avoid wasted bandwidth due to unused words

**Shared Memory Architecture**

- Many threads accessing memory
  - Therefore, memory is divided into banks
  - Successive 32-bit words assigned to successive banks

- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks

- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized
Bank Addressing Examples

**No Bank Conflicts**
- **Linear addressing**
  - stride == 1

**Random 1:1 Permutation**

**2-way Bank Conflicts**
- **Linear addressing**
  - stride == 2

**8-way Bank Conflicts**
- **Linear addressing**
  - stride == 8
Shared memory bank conflicts

- Shared memory is ~ as fast as registers if there are no bank conflicts
- `warp_serialize` profiler signal reflects conflicts

**The fast case:**
- If all threads of a half-warp access different banks, there is no bank conflict
- If all threads of a half-warp read the identical address, there is no bank conflict (broadcast)

**The slow case:**
- Bank Conflict: multiple threads in the same half-warp access the same bank
- Must serialize the accesses
- Cost = max # of simultaneous accesses to a single bank

Maximize Use of Shared Memory

- Shared memory is hundreds of times faster than global memory
- Threads can cooperate via shared memory
  - Not so via global memory
- A common way of scheduling some computation on the device is to block it up to take advantage of shared memory:
  - **Partition the data set** into data subsets that fit into shared memory
  - **Handle each data subset with one thread block:**
    - Load the subset from global memory to shared memory
    - `__syncthreads()`
    - Perform the computation on the subset from shared memory
      - each thread can efficiently multi-pass over any data
    - `__syncthreads()` (if needed)
    - Copy results from shared memory to global memory
Transposing an $nxn$ matrix

- $B[j][i] = A[i][j]$ for $i=1,...,n$ and $j=1,...,n$
- Partitions $A$ into $k^2 = kxk$ tiles, each with $n/k$ rows and $n/k$ columns
- Launch a 2-D grid of $(k,k)$ thread blocks, each with $(n/k,n/k)$ threads
- Assign thread $(threadIdx.x, threadIdx.y)$ in block $(blockIdx.x, blockIdx.y)$ to handle element $(row, col)$ of the matrix $A$.

Naïve implementation exhibits non-coalesced access to global memory

Transposing a matrix (coalescing memory access)

- Each block copies the rows of its tile from global to shared memory (coalesced)
- Transpose in shared memory
- Copy rows of the transposed tile from shared to global memory (coalesced)

```c
__global__ void transpose(float* A, float* B, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
}
```
**Multiplication $C = A \times B$ of two $nxn$ matrices.**

- Partitions each matrix into $k^2 = k \times k$ tiles, each with $n/k$ rows and $n/k$ columns.
- Launch a 2-D grid of $(k, k)$ thread blocks, each with $(n/k, n/k)$ threads.
- Assign thread $(\text{threadIdx.x}, \text{threadIdx.y})$ in block $(\text{blockIdx.x}, \text{blockIdx.y})$ to handle element $(\text{row}, \text{col})$ of the matrix $C$.

```c
__global__ void mm_simple(float* C, float* A, float* B, int n)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0;
    for (int k = 0; k < n; k++)
        sum += A[row][k] * B[k][col];
    C[row][col] = sum;
}
```

**In the main program:**
- `cudaMalloc d_A, d_b and d_C`;
- `cudaMemcpy to d_A, d_B`;
- `dim3 threads(block_size, block_size);`  
- `dim3 grid(n / block_size, n / block_size);`  
- `mm_simple <<< grid, threads >>>(d_C, d_A, d_B, n);` 
- `__syncthreads();`  
- `cudaMemcpy back d_C`;

**Matrix-matrix multiplication using shared memory**

- Each thread that computes $C[\text{row}][\text{col}]$ accesses row $A[\text{row}][\ast]$ and column $B[\ast][\text{col}]$ from the global memory.
- Hence, each row of $A$ and each column of $B$ is accessed $n/k$ times by threads in the same thread block.
- To avoid repeated access to global memory, the block that computes a tile $C_{i,j}$ of $C$, executes the following:
  
  For $k=0,\ldots,k-1$
  ```
  \{ 
  \quad \text{load tiles } A_{i,k} \text{ and } B_{k,j} \text{ into the shared memory} \\
  \quad C_{i,j} = C_{i,j} + A_{i,k} \times B_{k,j} \quad // \text{accumulate the product of } A_{i,k} \times B_{k,j} \\
  \}\n  ```
Multiplication $c = A \times b$ of an $n \times n$ matrix, $A$, by a vector $b$

- Assign the computation of each element $c_i$ of the result vector to a thread
- Partitions $c$ into $k$ parts, each with $n/k$ elements
- Launch $k$ thread blocks, each with $n/k$ threads
- Assign thread $threadIdx.x$ in block $blockIdx.x$ to compute $c_i$ of the result
where $i = blockIdx.x \times blockDim.x + threadIdx.x$

```
__global__ void mv(float* A, float* b, float* c, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = 0;
    for (int k = 0; k < n; k++)
        c[i] += A[row][k] * b[k];
}
```

GPU Thread Synchronization

- **void __syncthreads();**
- Synchronizes all threads in a block
  - Generates barrier synchronization instruction
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
  - Allowed in conditional code only if the conditional is uniform across the entire thread block
- To synchronize threads in different thread blocks, you need to use atomic operations on variables in global memory.

© NVIDIA Corporation 2009
GPU Atomic Integer Operations

- Requires hardware with compute capability $\geq 1.1$
  - G80 = Compute capability 1.0
  - G84/G86/G92 = Compute capability 1.1
  - GT200 = Compute capability 1.3

Atomic operations on integers in global memory:
- Associative operations on signed/unsigned ints
  - add, sub, min, max, ...
  - and, or, xor
  - Increment, decrement
  - Exchange, compare and swap

Atomic operations on integers in shared memory
- Requires compute capability $\geq 1.2$

Parallel reduction

```c
__global__ void plus_reduce(int *input, int N, int *total)
{
    int tid = threadIdx.x;
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    // Each block loads its elements into shared memory
    __shared__ int x[blocksize];
    x[tid] = (i<N) ? input[i] : 0; // last block may pad with 0's
    __syncthreads();

    // Build summation tree over elements.
    for(int s=blockDim.x/2; s>0; s=s/2)
    {
        if(tid < s) x[tid] += x[tid + s];
        __syncthreads();
    }

    // Thread 0 adds the partial sum to the total sum
    if( tid == 0 ) atomicAdd(total, x[tid]);
}
```
Occupancy

- Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep the hardware busy

\[ \text{Occupancy} = \frac{\text{Number of warps (threads) running concurrently on a multiprocessor}}{\text{Maximum number of warps (threads) that can run concurrently}} \]

- Limited by resource usage:
  - Registers
  - Shared memory

Blocks per Grid Heuristics

- # of blocks > # of multiprocessors
  - So all multiprocessors have at least one block to execute

- # of blocks / # of multiprocessors > 2
  - Multiple blocks can run concurrently in a multiprocessor
  - Blocks that aren’t waiting at a \texttt{__syncthreads()} keep the hardware busy
  - Subject to resource availability – registers, shared memory

- # of blocks > 100 to scale to future devices
  - Blocks executed in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations
Register Dependency

- **Read-after-write register dependency**
  - Instruction’s result can be read ~24 cycles later (assuming 6-stage pipelines, 8 cores per SM -- 4 cycles per warp instruction)

  **Scenarios:**

  **CUDA:**
  - \( x = y + 5; \)
  - \( z = x + 3; \)

  **PTX:**
  - \( \text{add.f32} \quad f3, f1, f2 \)
  - \( \text{add.f32} \quad f5, f3, f4 \)
  - \( s\_data[0] += 3; \)
  - \( \text{ld.shared.f32} \quad f3, [r31+0] \)
  - \( \text{add.f32} \quad f3, f3, f4 \)

  **To completely hide latency in the absence of forwarding:**
  - Run at least 192 threads (6 warps) per multiprocessor
  - At least 25% occupancy (1.0/1.1), 18.75% (1.2/1.3)
  - Warps do not have to belong to the same thread block

Register Pressure

- **Hide latency by using more threads per multiprocessor**

  **Limiting Factors:**
  - Number of registers per kernel
    - 8K/16K per multiprocessor, partitioned among concurrent threads
  - Amount of shared memory
    - 16KB per multiprocessor, partitioned among concurrent threadblocks

  **Compile with** \(-\text{ptxas-options}=-v\) **flag**

  **Use** \(-\text{maxrregcount}=N\) **flag to NVCC**

  - \( N = \) desired maximum registers / kernel
  - At some point “spilling” into local memory may occur
    - Reduces performance – local memory is slow
Optimizing threads per block

- **Choose threads per block as a multiple of warp size**
  - Avoid wasting computation on under-populated warps
  - Facilitates coalescing

- **More threads per block != higher occupancy**
  - Granularity of allocation
  - Eg. compute capability 1.1 (max 768 threads/multiprocessor)
    - 512 threads/block => 66% occupancy (can fit only one block)
    - 256 threads/block can have 100% occupancy (can fit 3 blocks)

- **Heuristics**
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 192 or 256 threads a better choice
    - Usually still enough regs to compile and invoke successfully
  - This all depends on your computation, so experiment!

Unified Memory (in recent Cuda releases)
Dramatically Lower Developer Effort

**Developer View Today**

- System Memory
- GPU Memory

**Developer View With Unified Memory**

- Unified Memory
Super Simplified Memory Management Code

**CPU Code**

```c
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**

```c
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);
}
```

Unified Memory Delivers

1. Simpler Programming & Memory Model
   - Single pointer to data, accessible anywhere
   - Tight language integration
   - Greatly simplifies code porting

2. Performance Through Data Locality
   - Migrate data to accessing processor
   - Guarantee global coherency
   - Still allows cudaMemcpyAsync() hand tuning